ELECH-409
Th01: Introduction to VHDL

Dragomir Milojevic
Université libre de Bruxelles

Today

1. Introduction
2. Logic Circuits
3. FPGAs
4. Hardware Description Languages (HDLs)
5. Design Flow
6. VHDL primer
7. Practical examples

1. Introduction

About these slides

- Slide notations:
  - *Italic* – most of the time external info (citations, article titles etc.)
  - **Bold** – trying to attract your attention to it
  - Blue – known concept (it should be!)
  - Purple – new concept
  - Red – important to remember, understand, connect
  - Brown – question to you, I am expecting you to think and provide an answer; I use this to measure your degree of understanding; obviously we will look into these during Q&A sessions if necessary
- What you have here is enough to successfully pass the exam, but you need to understand things
  - I will provide some extra references, you should look at them only if you really want
- If you see any issue with the slides, let me know by email in which lecture and clearly state the problem

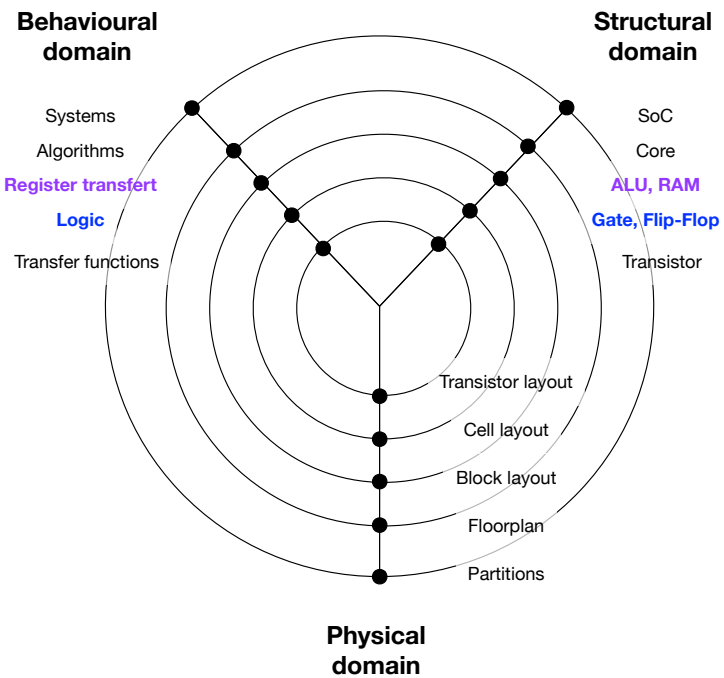
Course title explained

Digital Architecture & Design is about:

- Digital – we will use logic circuits that implement Boolean logic to process binary values (0,1) with a set of predefined logic gates
- Architectures – more or less complex systems that process digital information
  - Architecture could be anything: from a simple 8-bit micro-controller to a complex multi 64-bit cores processor (CPU), many core Graphical Processor Unit (GPU) or Artificial Intelligence core, etc.
  - We speak of System-on-Chip (SoCs), single Integrated Circuits (ICs) built using millions of logic gates on a single die in an Application Specific Integrated Circuit (ASIC)
  - SoCs and ASICs are implemented using CMOS technology to produce transistors, basic building block of all digital systems
- Design – is about all activities required to effectively build such system (and there are few!)

Focus

Y-Chart from Gasjki: structured view of digital systems & design



- Here we focus on gates (AND, OR, NAND, INV etc., 2nd circle) and will most of the time totally abstract the electronics (i.e transistors) behind
- We will also "look" one level above the gate (3rd circle): i.e. a small block of a system, (adders, counters, state machines, ALUs), but not the level above (e.g. SoC)
- By "look" in the above we mean:
  - ▷ Formalize & model digital circuits
  - ▷ Simulate these models to validate their functionality
  - ▷ Implement them in Field Programmable Gate Arrays (FPGAs)

2. Logic Circuits

Background

- In the next couple of slides we will cover the basics of digital logic circuits theory
- I do assume that all this is known, and that I am just refreshing your brain cells and synchronizing the vocabulary!
- Not all the concepts from digital logic circuits theory will be needed (e.g. manual logic synthesis and optimization), but some are essential to understand what we will be doing here, such as: combinatorial, sequential circuits, race conditions, flip-flops, synchronization etc.
- So if there are things that needs to be clarified, please rise the flag during the next QA session!
- Learning VHDL requires a solid foundation before going any further ...

Terminology & Logic circuit classes

- Logic circuits – systems that process digital information using Boolean logic functions implemented in hardware (HW)
- Boolean functions use inputs and **eventually** internal states to produce output and/or certain behavior (new system state)
- HW is about integrated electronic components (discrete & ICs)
- First used in early '50 to build control systems, today they are everywhere: on your desk, in your pocket and soon they will be all over & even inside your body!
- There are two logic circuit classes:
  - ▷ Combinatorial – use logic gates or Look-Up-Tables, but do not have feedback and/or memory
  - ▷ Sequential – asynchronous & synchronous [1] with feedback/memory
- Logic circuit design – how to formalize a circuit from verbal specifications, and derive information required to manufacture and physically implement given functionality

[1] Here we focus on synchronous circuits only

Combinatorial logic circuits – representation

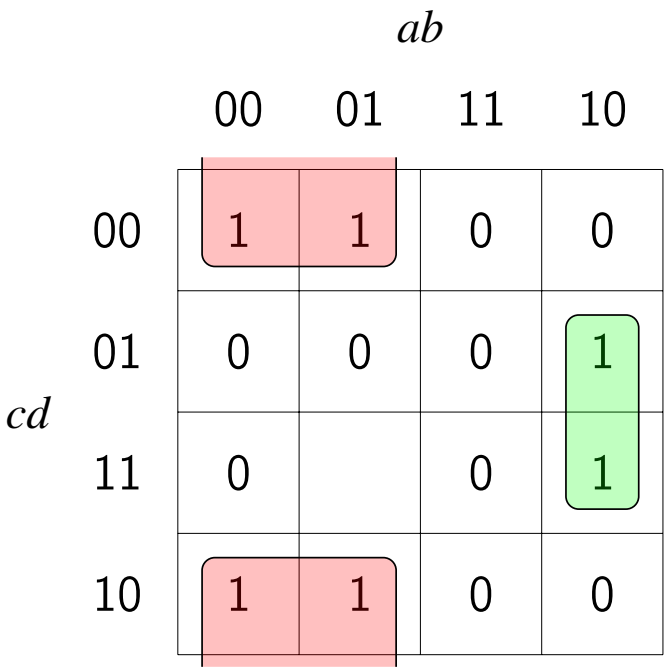
- Output(s) O is (are) function of inputs I **only**: $O = f(I)$
- Same input combination will always produce the same output
- Representation done using Truth Tables & Logic Functions
- Minterm is the combination of input variables for which $F = 1$ (or maxterm when $F = 0$)
- Logic function is then the sum of the minterms (or product of maxterms):

Dec.	a	b	c	d	F
0	0	0	0	0	**1**
1	0	0	0	1	**1**
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	**1**
7	0	1	1	1	0
8	1	0	0	0	**1**
9	1	0	0	1	**1**
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	**1**
15	1	1	1	1	0

$$F = a'b'c'd' + a'b'c'd + a'bcd' + ab'c'd' + ab'c'd + abcd'$$

Combinatorial logic circuits – optimisation

- Previous expression is not optimal, gate count could be LESS!
- Important to save **circuit resources** (i.e. cost, power, etc.)
- Many optimization methods: Boolean axioms/theorems, Karnaugh maps, Quinne-McCluskey, Espresso, etc.

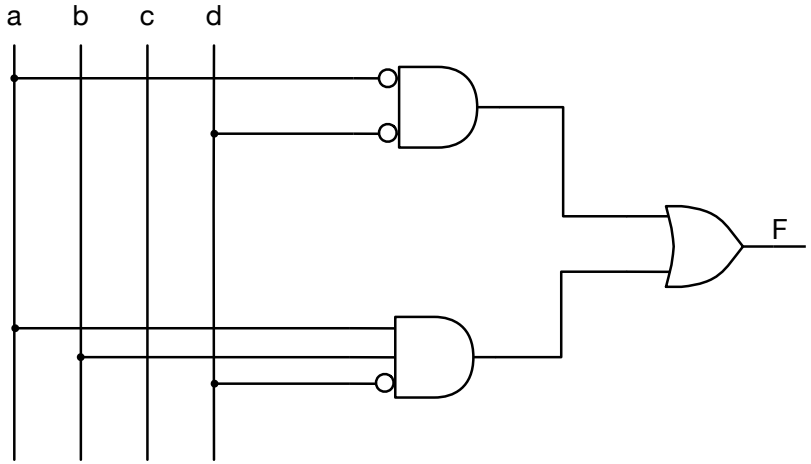


E.g. using K-Maps:

$$F = a'd' + ab'd$$

Combinatorial logic circuits (3/3)

- Optimized representations are often given in the form of Sum-of-Products (SoP) by looking into `1`s, or Product-of-Sums (PoS), by looking into `0`s
- Sop expression $F = a'd' + ab'd$ can be converted into a logic circuit:



- Note AND gates & one OR gate at the end; All logic functions could be implemented with a set of AND and a single OR gate (SoP)
- This is two-level logic, circuit delay is divided in two stages: first delay of AND stage and then the delay of OR gate stage
- Logic circuits today are implemented using multi-level logic, a series of AND/OR gates (more on this later on)

Sequential logic circuits (1/2)

- **Outputs** (O) depend on **inputs** (I) and system state (S)
- For the same combination of inputs we can have different outputs
- Future state (S^+) is calculated as a logic function of inputs AND previous state (S^-) represented using internal variables
- State can be a vector: multiple Boolean values/expressions
- System described with two sets of Boolean equations, one set for output(s) and one for next state(s) S^+:

$$\begin{aligned} O &= f_1(I, S^-) \\ S^+ &= f_2(I, S^-) \end{aligned}$$

- Sequential logic circuit can be seen as a combinatorial system with one (or more) feedback(s) on state variables
- Thus all sequential systems have this feedback! (crucial)

Sequential logic circuits (2/2)

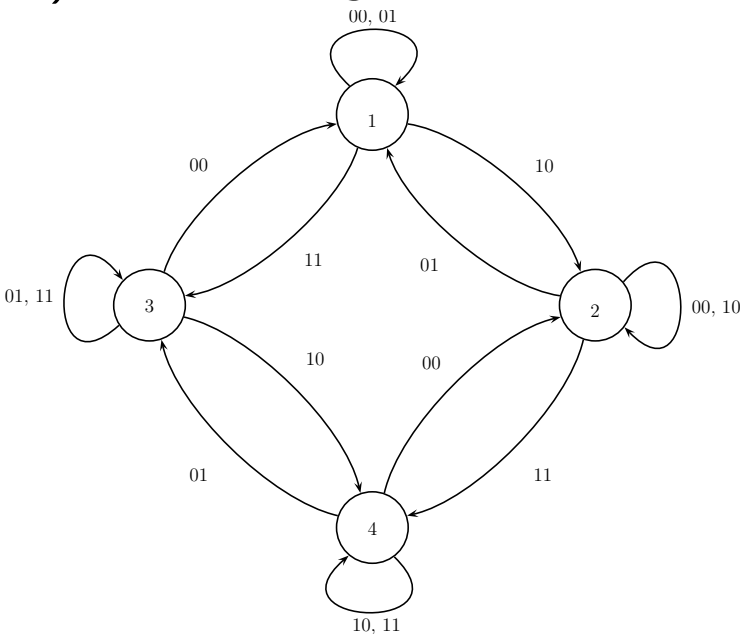
Many different ways to represent sequential systems:

A) State tables:

- In bold – stable states
- Otherwise transitions
- This is a Mealy machine
  - ▷ Ever heard of Moore/Mealy?

	ab 00	01	11	10
1	**1/1**	**1/0**	3	2
2	**2/0**	1	4	**2/1**
3	1	**3/1**	**3/0**	4
4	2	3	**4/1**	**4/0**

B) State diagram:



C) Boolean logic equations:

$J1 = ab,\ K1 = a'b'$

$J2 = y_1a' + y_1'a,\ K2 = y_1b' + y_1'b$

$Z = b'y_1y_2 + ay_1'y_2$

Logic circuit design or synthesis

- Starts with some initial **specification of the system** – it has to be correct otherwise your system will not do what you want it to do (which is embarassing)
- Synthesis can be quite complex knowing how much logic gates we can pack in a mm^3 these days
- Formal specification – we translate the above spec into some kind of a **formal model** of digital circuit
- Logic optimization – logic description transformation to reach certain objectives (Performance, Power, Area, Cost – PPAC)
  - Optimizations can be done manually ...
  - If you followed ELECH305 or ELECH310 you should know this
  - If NOT: you should at least have some ideas on how this is done and why it is so important
- Logic functions implementation – circuit assembley

Logic circuit optimization

- Optimization is a very important part of digital system design
  - Why this is the case?
- Optimization of **combinatorial circuits**:
  - Boils down to Boolean logic functions optimisations
  - Could be done using logic axioms and theorems with variable results
  - K-maps for problems with $\leq$ then 5 variables ($>$ 6 is tough)
  - When $>$ 6 variables manual or automated Quinne-McCluskey
- Optimization of **sequential circuits**:
  - State tables and their combinatorial logic
  - State table reduction: equivalences and state fusion
  - Flip-flop choices (SR, JK, T and D) used to store the values of the state variables influence the complexity of logic functions used to generate their control signals

Manual digital synthesis

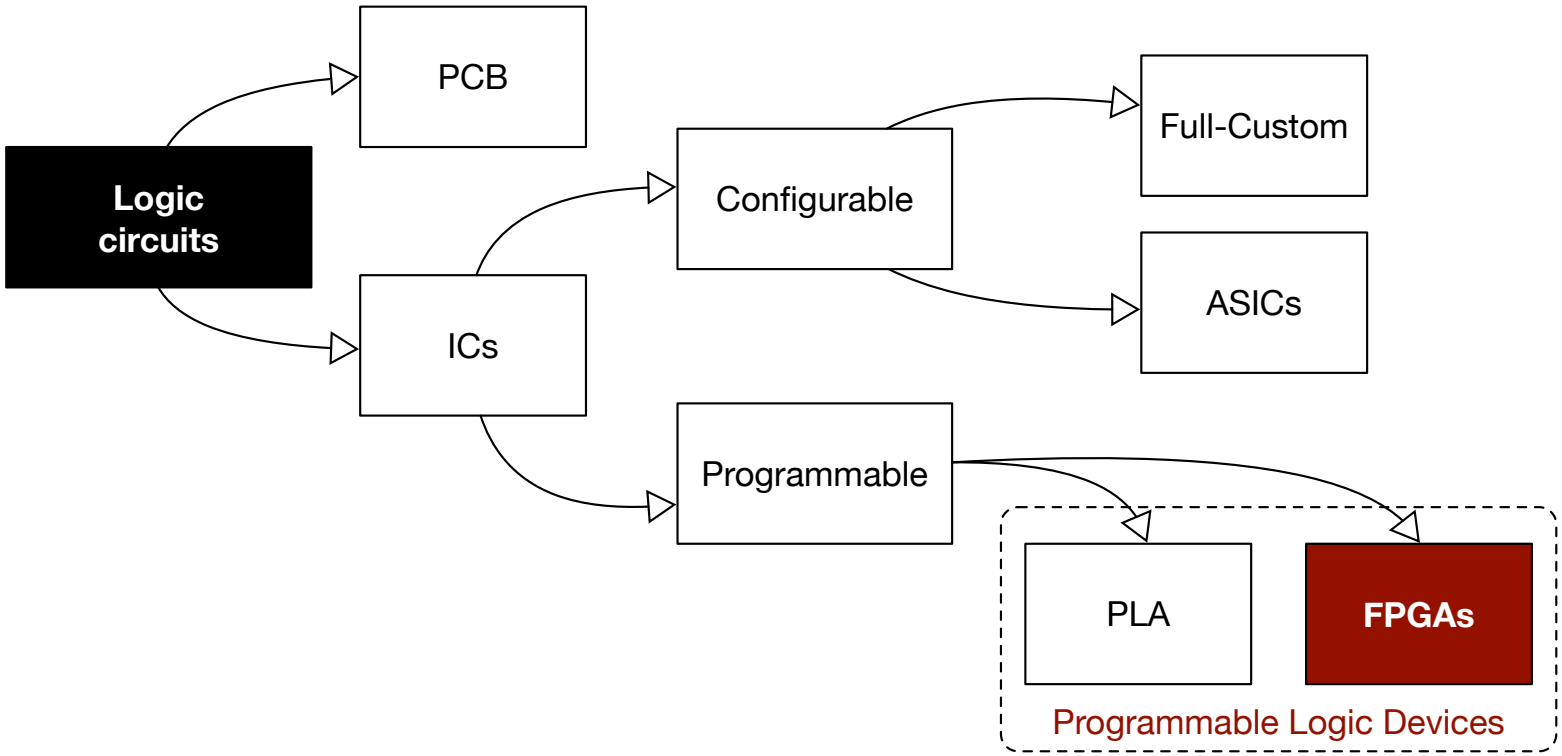
- Is lengthy, error prone, even for simple circuits, simple being:
  - 5 to 6 variables for combinatorial circuits
  - $<$10 states (2-4 state variables) for sequential systems
- Question is what to do with more complexes problems, with:
  - **hundreds of logic variables** for combinatorial systems, or
  - **hundreds of states** for sequential systems
- And this is what's happening with current CMOS integration that allow implementation of millions of logic gates per mm^3 of silicon: 10, 7 & 5nm process technology and 3D integration, reason why we speak of mm^3

Need for alternative ways to describe, optimise and implement digital logic circuits

Welcome to HDLs & EDA: subject of these lectures

How to implement logic circuits?

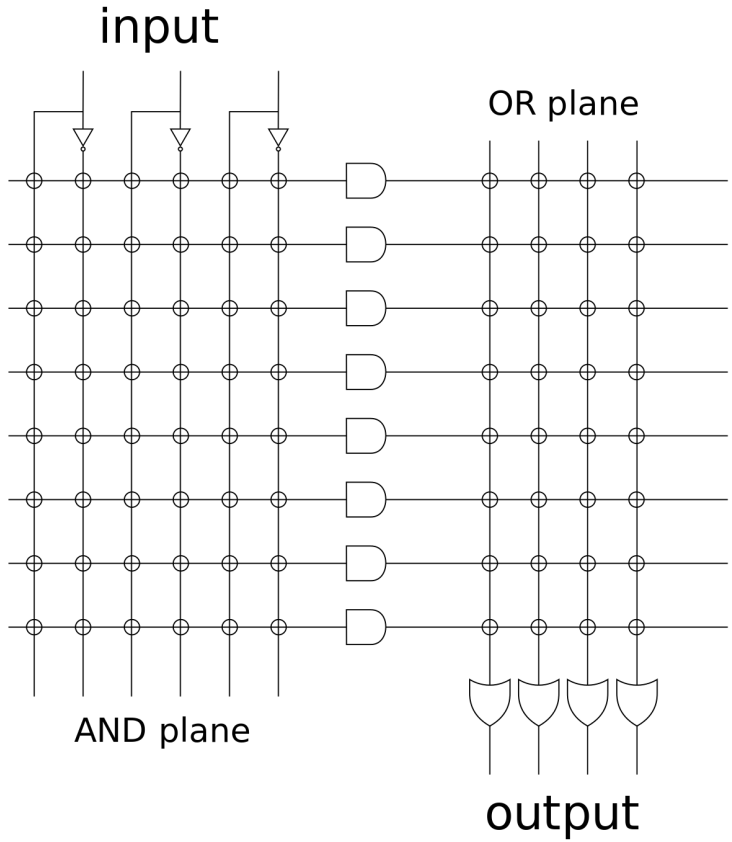
Many different options !



Choice will depend on what you want to do,
in these lectures we focus on **FPGA circuits only**, but do mention ASICs from time to time so that you can get a feeling of it

Programmable Logic Array (PLA)

- First programmable logic devices to appear where circuit functionality is **not** fixed during manufacturing, but can be decided later!
  - ▷ Programming process is permanent though
- PLA implement Sum-of-Products expressions in hardware
- Regular structure of AND & OR gates (planes) with **programmable connectivity**
- Limited to appx. **100 IOs** and relatively simple logic functions
- As CMOS integration evolved, more IC density allowed more complex PLA circuits $\rightarrow$ FPGAs

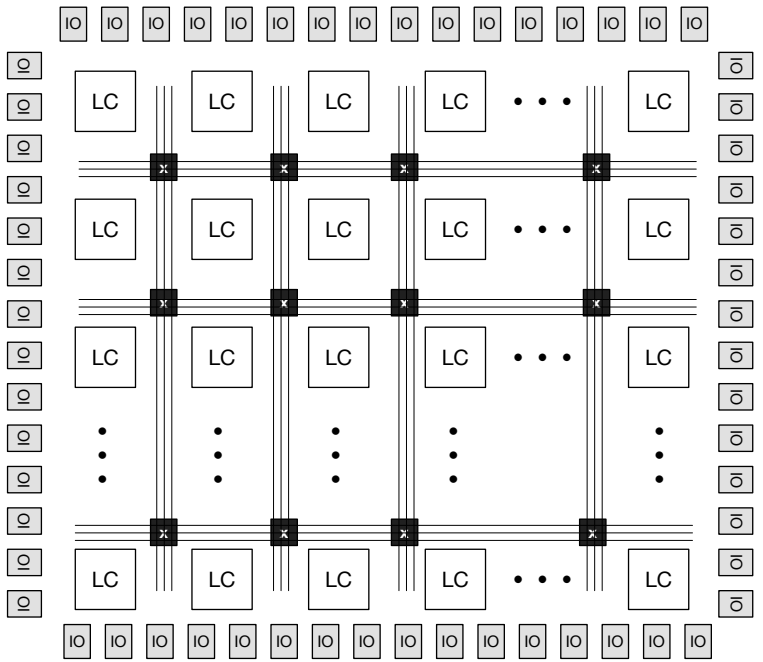


3. FPGAs

Field Programmable Gate Arrays (FPGA)

Regular (typically **mesh**) structure of:

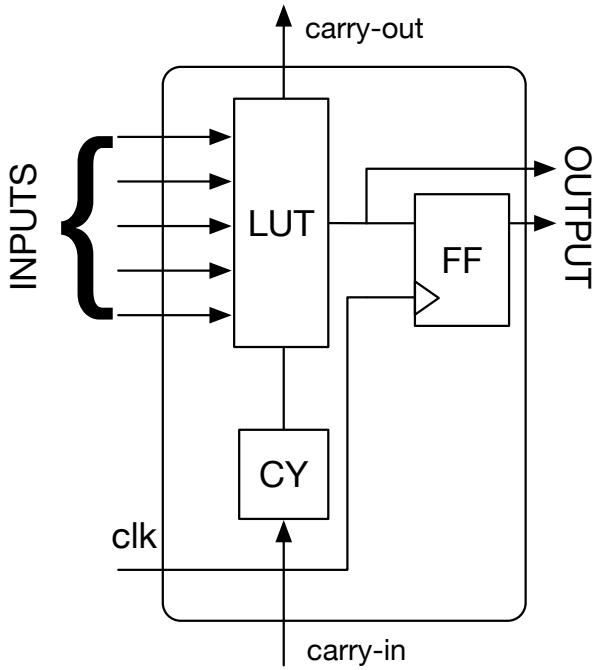
- Logic Cells (LCs)[2] – implement logic functions using memories to store Truth Tables (logic function output) as function of input (addresses)
- Programmable interconnect – can connect any two (or more) arbitrary points inside the circuit
- I/O blocks – off-chip connection
- Memories (SRAMs) – any digital design needs a memory, often half of the resources could be used for storage (and even more)
- Dedicated logic CPUs, DSPs etc.



[2] CLE, CLB, slice, LC ... depending on manufacturer, FPGA family etc.

Typical Logic Cell (LC) architecture

- Each LC has at least one Look Up Table – LUT, i.e. a memory that stores Truth Table of the logic function to be implemented
- LUT is an SRAM: 1-bit data with few address bits ($addr < 10$)
- Thus the SRAM is small, so it can have fast READ times
  - What is the size of the LUT?
  - Why LUT needs to be fast?
  - How many different logic functions a single LUT can implement?

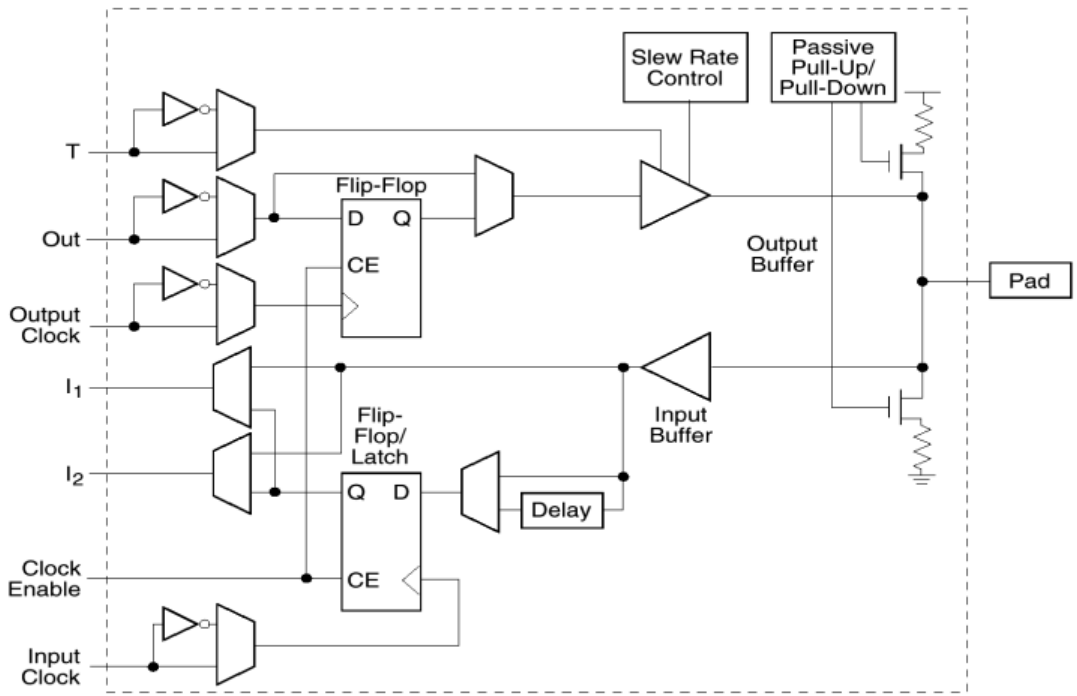


- LUT output is either connected to combinatorial LC output or through FF for a sequential (registered) output
- Some LCs have supplementary logic resources to perform certain functionality, typically carry generation logic (CY)

Input/Output (IO) blocks

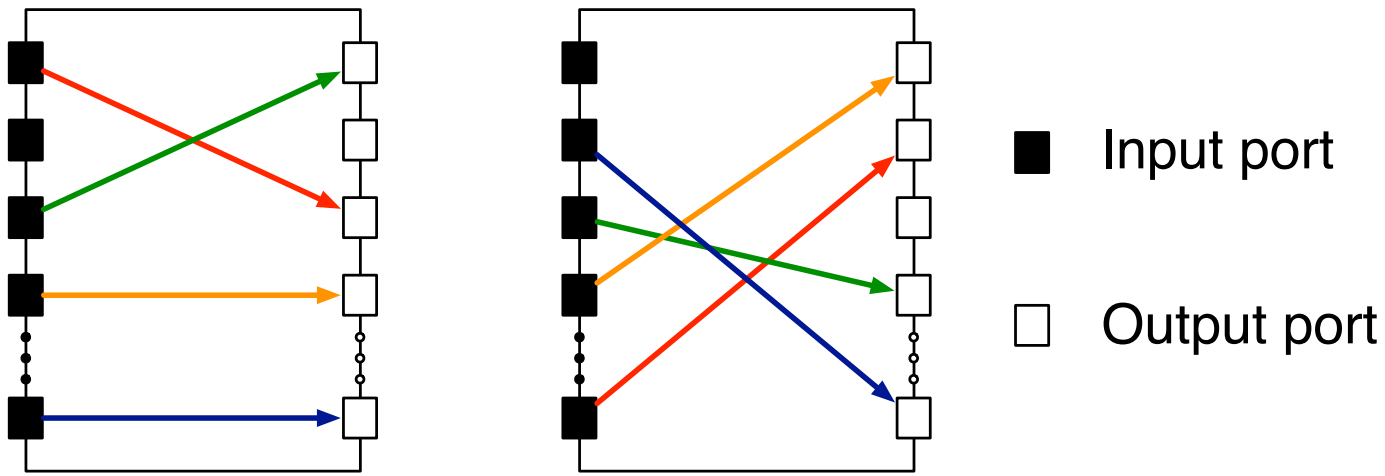
- Enables communication between FPGA & the outside world through the connection of the FPGA package: the pad
- Also allows to adapt different electrical standards used to connect external components (there is a huge variety of IO standards)
- Enables synchronization of inputs with internal logic using internal Flip-Flops

Note that
Input, Output directions
are programmable



Programmable interconnect

- Enables connectivity between different LCs: any LC can be connected with any other LC in the FPGA !
- Each LC has something called Programmable Switch Matrix (PSM) (this is a terminology of a particular FPGA vendor)
- PSM is a **crossbar**: a circuit that connects n inputs to n outputs
- Connection is programmable (per FPGA configuration!)



- PSM introduces a supplementary delay compared to a simple propagation delay of a wire
  - ▷ Make sure that you can explain why

FPGA manufacturing technologies & usage

- Depending on the manufacturing technology used to build the IC of the FPGA we can have:
  - FPGA devices that are **programmable once** – anti-fuse
  - FPGA devices that are **reprogrammable** – LUTs are implemented using SRAMs; most common devices used today
    - Re-programmable FPGAs enable reconfigurable computing
- FPGA vs. ASIC: lower operating frequencies, but not necessarily lower performance, they are less area/power efficient, but easier to design & recoverable if errors in HW (since you can just re-program the circuit in case of an error)
- FPGA use:
  - Low volume production – ASICs are very expensive and justifiable only if you have a huge volume (millions of components sold)
  - Prototyping – implementation and test before ASIC manufacturing
  - Reconfigurable systems – run-time configurability; change silicon functionality on the fly !!!

ASIC vs. FPGA

Now that you know what an FPGA is, you should be able to answer the following questions:

- Explain how logic functions are implemented in an FPGA?
- Will we really implement logic functions in FPGAs as SoP?
  - Take
$$F(a,b,c,d) = abc' + ab'd + a'bd' + a'b'c' \tag{1}$$
  - Such function will be decomposed using **Factoring, Shannon, Boolean** decomposition theorem that enables multi-level logic:
$$F = a(bc' + b'd) + a'(bd' + b'c') \tag{2}$$
  - What is the trade-off of two equations above?
- What should be the difference between FPGA and ASIC in terms of logic implementation process?

FPGA example: family summary (1/3)

Summary of Virtex-6 FPGA Features

- Three sub-families:
  - Virtex-6 LXT FPGAs: High-performance logic with advanced serial connectivity
  - Virtex-6 SXT FPGAs: Highest signal processing capability with advanced serial connectivity
  - Virtex-6 HXT FPGAs: Highest bandwidth serial connectivity
- Compatibility across sub-families
  - LXT and SXT devices are footprint compatible in the same package
- Advanced, high-performance FPGA Logic
  - Real 6-input look-up table (LUT) technology
  - Dual LUT5 (5-input LUT) option
  - LUT/dual flip-flop pair for applications requiring rich register mix
  - Improved routing efficiency
  - 64-bit (or two 32-bit) distributed LUT RAM option per 6-input LUT
  - SRL32/dual SRL16 with registered outputs option
- Powerful mixed-mode clock managers (MMCM)
  - MMCM blocks provide zero-delay buffering, frequency synthesis, clock-phase shifting, input-jitter filtering, and phase-matched clock division
- 36-Kb block RAM/FIFOs
  - Dual-port RAM blocks
  - Programmable
    - Dual-port widths up to 36 bits
    - Simple dual-port widths up to 72 bits
  - Enhanced programmable FIFO logic
  - Built-in optional error-correction circuitry
  - Optionally use each block as two independent 18 Kb blocks
- High-performance parallel SelectIO™ technology
  - 1.2 to 2.5V I/O operation
  - Source-synchronous interfacing using ChipSync™ technology
  - Digitally controlled impedance (DCI) active termination
  - Flexible fine-grained I/O banking
  - High-speed memory interface support with integrated write-leveling capability
- Advanced DSP48E1 slices
  - 25 x 18, two's complement multiplier/accumulator
  - Optional pipelining
  - New optional pre-adder to assist filtering applications
  - Optional bitwise logic functionality
  - Dedicated cascade connections
- Flexible configuration options
  - SPI and Parallel Flash interface
  - Multi-bitstream support with dedicated fallback reconfiguration logic
  - Automatic bus width detection
- System Monitor capability on all devices
  - On-chip/off-chip thermal and supply voltage monitoring
  - JTAG access to all monitored quantities
- Integrated interface blocks for PCI Express® designs
  - Compliant to the PCI Express Base Specification 2.0
  - Gen1 (2.5 Gb/s) and Gen2 (5 Gb/s) support with GTX transceivers
  - Endpoint and Root Port capable
  - x1, x2, x4, or x8 lane support per block
- GTX transceivers: up to 6.6 Gb/s
  - Data rates below 480 Mb/s supported by oversampling in FPGA logic.
- GTH transceivers: 2.488 Gb/s to beyond 11 Gb/s
- Integrated 10/100/1000 Mb/s Ethernet MAC block
  - Supports 1000BASE-X PCS/PMA and SGMII using GTX transceivers
  - Supports MII, GMII, and RGMII using SelectIO technology resources
  - 2500Mb/s support available
- 40 nm copper CMOS process technology
- 1.0V core voltage (-1, -2, -3 speed grades only)
- Lower-power 0.9V core voltage option (-1L speed grade only)
- High signal-integrity flip-chip packaging available in standard or Pb-free package options

FPGA example: different components (2/3)

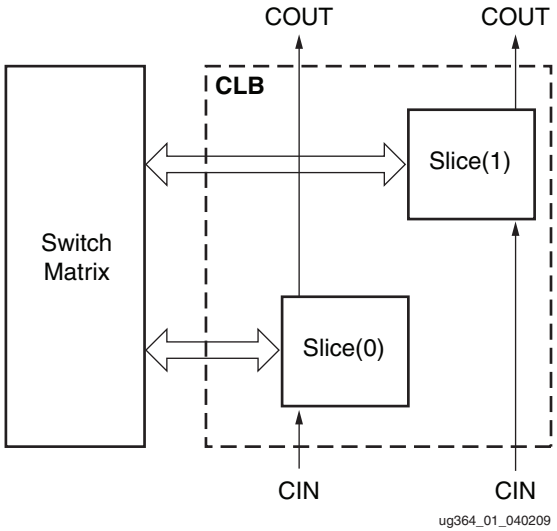
Virtex-6 FPGA Feature Summary

Table 1: **Virtex-6 FPGA Feature Summary by Device**

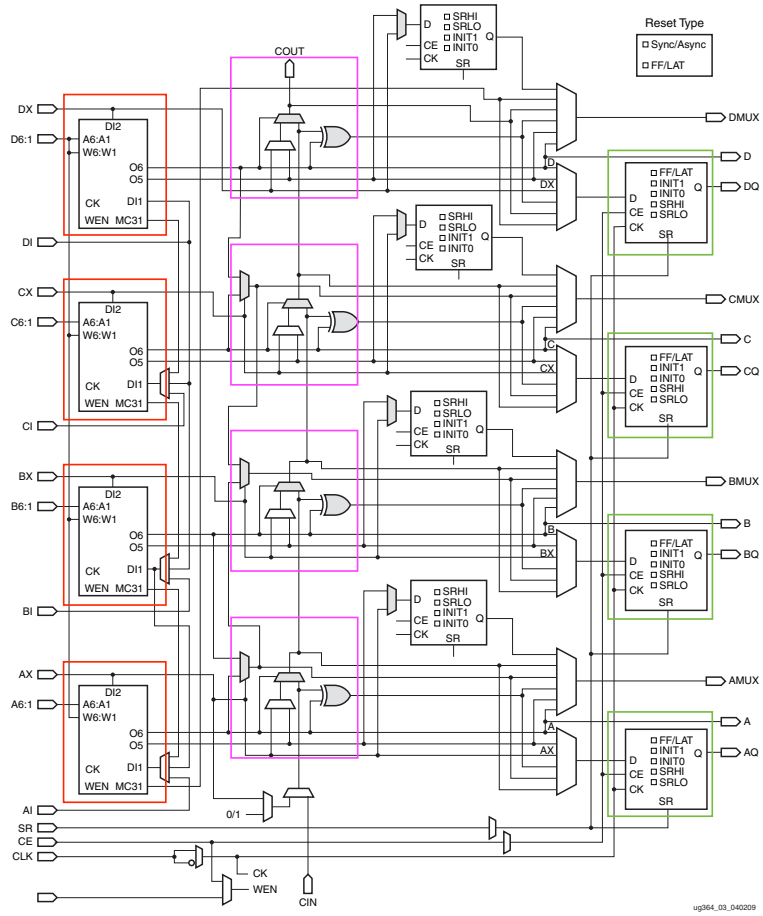
Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices[2]	Block RAM Blocks			MMCMs[4]	Interface Blocks for PCI Express[5]	Ethernet MACs[6]	Maximum Transceivers		Total I/O Banks[7]	Max User I/O[8]
		Slices[1]	Max Distributed RAM (Kb)		18 Kb[3]	36 Kb	Max (Kb)				GTX	GTH		
XC6VLX75T	74,496	11,640	1,045	288	312	156	5,616	6	1	4	12	0	9	360
XC6VLX130T	128,000	20,000	1,740	480	528	264	9,504	10	2	4	20	0	15	600
XC6VLX195T	199,680	31,200	3,040	640	688	344	12,384	10	2	4	20	0	15	600
XC6VLX240T	241,152	37,680	3,650	768	832	416	14,976	12	2	4	24	0	18	720
XC6VLX365T	364,032	56,880	4,130	576	832	416	14,976	12	2	4	24	0	18	720
XC6VLX550T	549,888	85,920	6,200	864	1,264	632	22,752	18	2	4	36	0	30	1200
XC6VLX760	758,784	118,560	8,280	864	1,440	720	25,920	18	0	0	0	0	30	1200
XC6VSX315T	314,880	49,200	5,090	1,344	1,408	704	25,344	12	2	4	24	0	18	720
XC6VSX475T	476,160	74,400	7,640	2,016	2,128	1,064	38,304	18	2	4	36	0	21	840
XC6VHX250T	251,904	39,360	3,040	576	1,008	504	18,144	12	4	4	48	0	8	320
XC6VHX255T	253,440	39,600	3,050	576	1,032	516	18,576	12	2	2	24	24	12	480
XC6VHX380T	382,464	59,760	4,570	864	1,536	768	27,648	18	4	4	48	24	18	720
XC6VHX565T	566,784	88,560	6,370	864	1,824	912	32,832	18	4	4	48	24	18	720

Look at the density of the last device, and this 40nm (iPhone 12 is 5nm!)

FPGA example: logic cell configuration (3/3)



Two slices form one CLB connected to PSM



One CL has 4 $\times$ 6-inputs LUTs, 4 Carry Logic, and 4 programmable FlipFlops

4. Hardware Description Languages (HDLs)

HDLs – Hardware Description Languages

- Note the plural on *Languages*, indicates that there are few of these (but much less than in software)
- HDLs enable us to formalize verbal specifications of circuits (e.g. adders, CPUs, GPUs etc.)
  - *Formal* means complete, precise and without ambiguities
  - Formal models are manipulated using computers (automation)
  - This is essential, since we need to cross the frontier of the human language (inaccurate verbal specification) to accurate HDL model
- HDLs allow automated synthesis of logic circuits (using some SW)
  - Synthesis – Transform higher-level (aka behavioral) models into logic gates & descriptions for physical implementation
  - Formally described logic circuit (in HDLs) at **higher abstraction levels** is translated into a description using logic gates (i.e. **lower abstraction levels**)
  - Could be seen as schematic but in reality it is not graphical representation, rather a data-base called gate-level netlist

HDLs and simulation

- HDLs enable automated simulation to verify functional properties of the design before physical implementation; some manual preparation of the simulation environment is always necessary (scripting), but simulation process itself is fully automated
- More generally we speak about model verification at various stages of the flow
- HDLs can be used to **automatically** generate physical design description for a given target technology: FPGA or ASIC of a given CMOS technology node
- HDLs have great capacity for **structuring**, i.e. they allow description of very complex systems through hierarchical modeling top-down or bottom-up
- Complex systems: all of the above can be done for circuits with millions of logic gates & flip-flops **in reasonable amount of time** (hours, days, rarely weeks)

Abstraction levels

- HDLs can describe logic circuits at various abstraction levels
- For example we can describe an Arithmetical Logic Unit (ALU) that will perform addition **without** necessarily describing **how** this operation will be done
  - Options for addition operation: Ripple Carry or Carry Look Ahead (& many others...)
  - How would you derive logic functions of these two adder circuits?
  - Different adder architecture trade-off power, performance, area ...
- ALU can be described and **simulated** using a normal computer
- Computers can do addition; so we do not need to decide which adder architecture we chose at this stage of the system design (simulation)
- Then only later we can chose how to effectively implement the adder circuit (or let the tool do the synthesis for us)

HDL language classes

- Simulation
  - Used to make more precise function specification, validate some system micro-architecture assumptions through simulation
  - NOT intended for physical implementation because models don't have necessary levels of detail
  - Simulation is in general fast even for complex systems
- Implementation
  - Target **physical implementation** of circuits
  - Models describe functionality in detail to allow physical implementation (full custom CMOS IC, ASIC or FPGA)
  - Very precise description of the system; it can be simulated too
  - Simulations are in general slow, especially for lower abstraction levels (one of the reasons that motivated simulation languages)
- In reality most HDLs can do both! but we say that **only a subset** of what can be described using HDLs **can be effectively synthesized**

Practical HDL languages

- SystemC – extensions to C language (SW) to enable simulation of HW systems; Used to model complex HW systems & decide on some architectural choices **before doing actual system design**; Part of it can be synthesized to logic but don't expect magic there is no easy way to convert SW to HW!
- VHDL, Verilog – mainly used for physical circuit simulation & implementation; Attention! **part of VHDL is not synthesizable**: some constructs are used for **simulation only** (they will be ignored during synthesis); Also some synthesis tools do not allow certain constructs
  - ▷ Guideline – If there is a construct to use, check well before using!
- Important: even if the modeling language is universal, adopted HDL coding style could be influenced by the target technology used for circuit implementation (e.g. ASIC/FPGA); Technology/ software tool dependent models/optimisations are not uncommon

VHDL origins and usage

- Acronym of an acronym:
  VHDL = VHSIC Hardware Description Language
  VHSIC = Very High Speed Integrated Circuits
- Developed by Department of Defense of USA in early '80
- Objective: enable automated modeling, synthesis and implementation of complex digital circuits to minimize development time, cost and errors through:
  - Formal specification of digital systems and verification environment
    - Verification – definition of input stimuli through test-benches, that will be applied to the system to gather outputs through simulation
  - Automated syntax and semantic consistency, circuit synthesis and physical implementation
  - The above is done using Electronic Design Automation (EDA), software tools that despite their super efficient algorithms need human intervention !!!
- The above is true for Verilog too

5. Design Flow

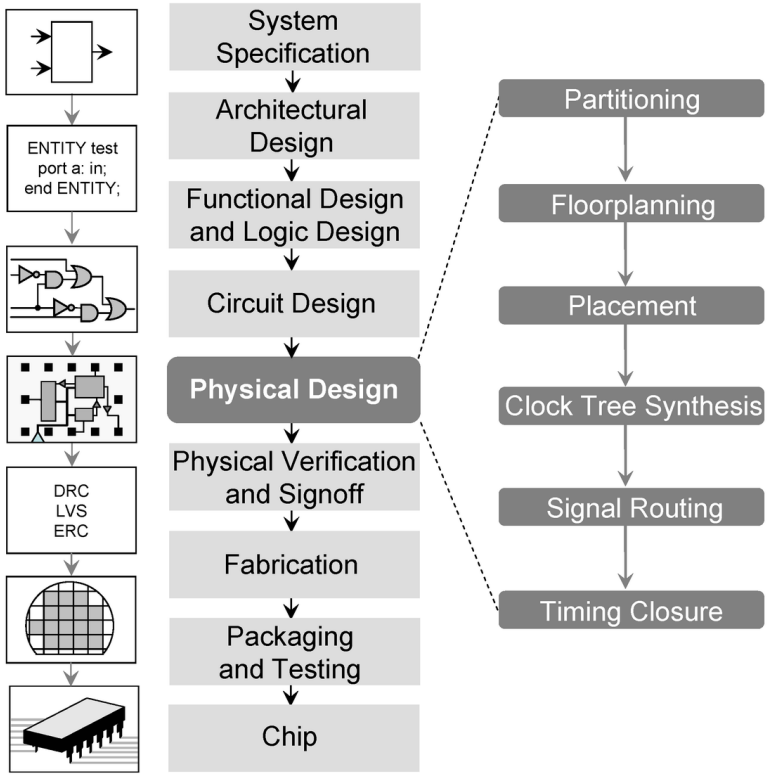
Overview of the digital system design

- Designing digital system is a complex and tedious task, so **divide & conquer** strategy is a must
- Digital circuit design process is decomposed in a sequence of different implementation steps, called design flow
- The goal of the design flow is to transform the initial system model, typically described at higher abstraction levels such as behavioral models, into a detailed low-level physical circuit (gates and transistors) that allows actual system implementation (assembled PCB, programmed FPGA, manufactured ASIC etc.)
- This is done in a progressive manner, with many, MANY, MANY iterations due to circuit complexity & complexity of transforms involved in the design flow
- Different steps are usually executed by different engineering teams, using many different tools, easy interfacing between design teams (& tools) is (very) important

Design flow steps – coarse view (ASIC or FPGA)

- System specification – initial description that can be less formal (verbal specs) or more formal (sometimes using **high-level description languages**); designers typically decide on global system micro-architecture
- RTL modeling & functional simulation – establishment of the HDL model (VHDL, Verilog and/or SystemC) that is tested & validated for functionality only (Boolean correctness)
  - RTL stands for Register Transfer Logic a paradigm in which logic data flow travels between gates and flip-flops from inputs to outputs; designates a digital system model
- Synthesis – transforms HDL descriptions into Gate-Level Netlist logic representation of the circuit
- Placement & Routing – decide on:
  - **where** components will be instantiated, so (x, y) location and
  - how they are **interconnected**, i.e. wires
- Sign-off – various properties of the circuits are validated for manufacturing (timing, power, thermal, mechanical etc.)

Design flow steps (more detailed view)



Above is done using EDA,
i.e. SW tools (note the plural) that put in practice solutions to many fundamental computer science (read complex) problems

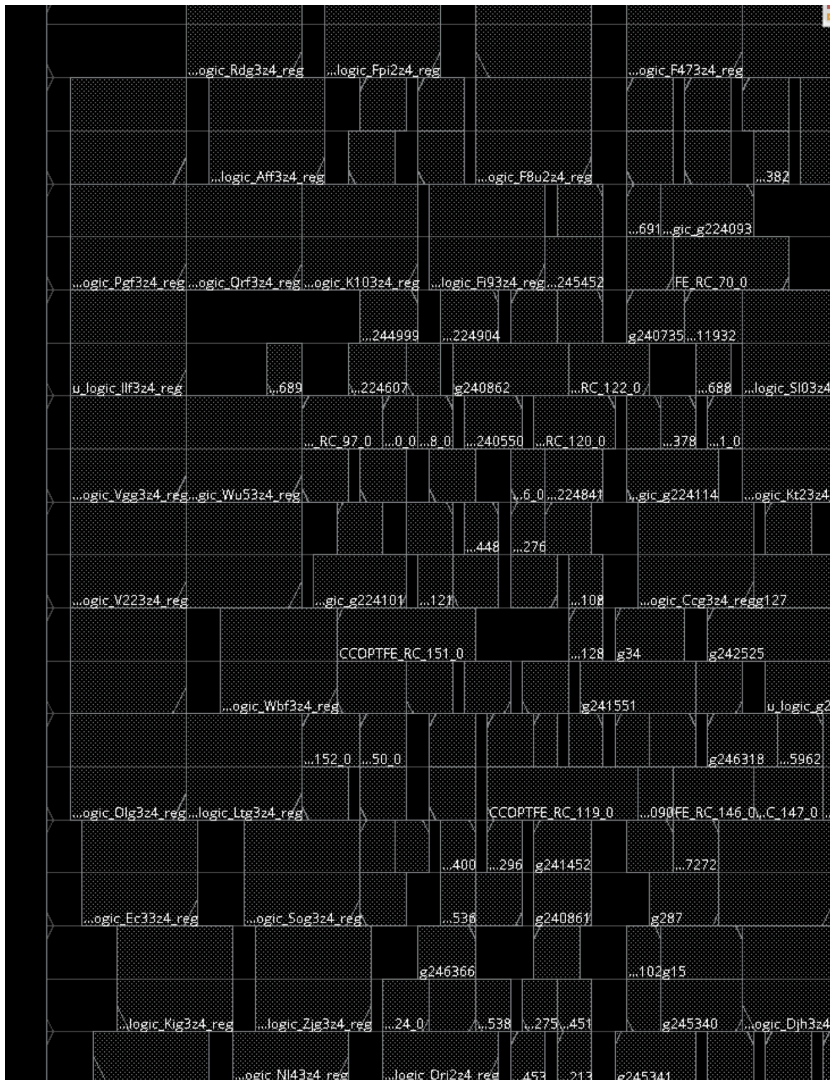
This means we often don't have optimal solution,
but try to get the best possible one

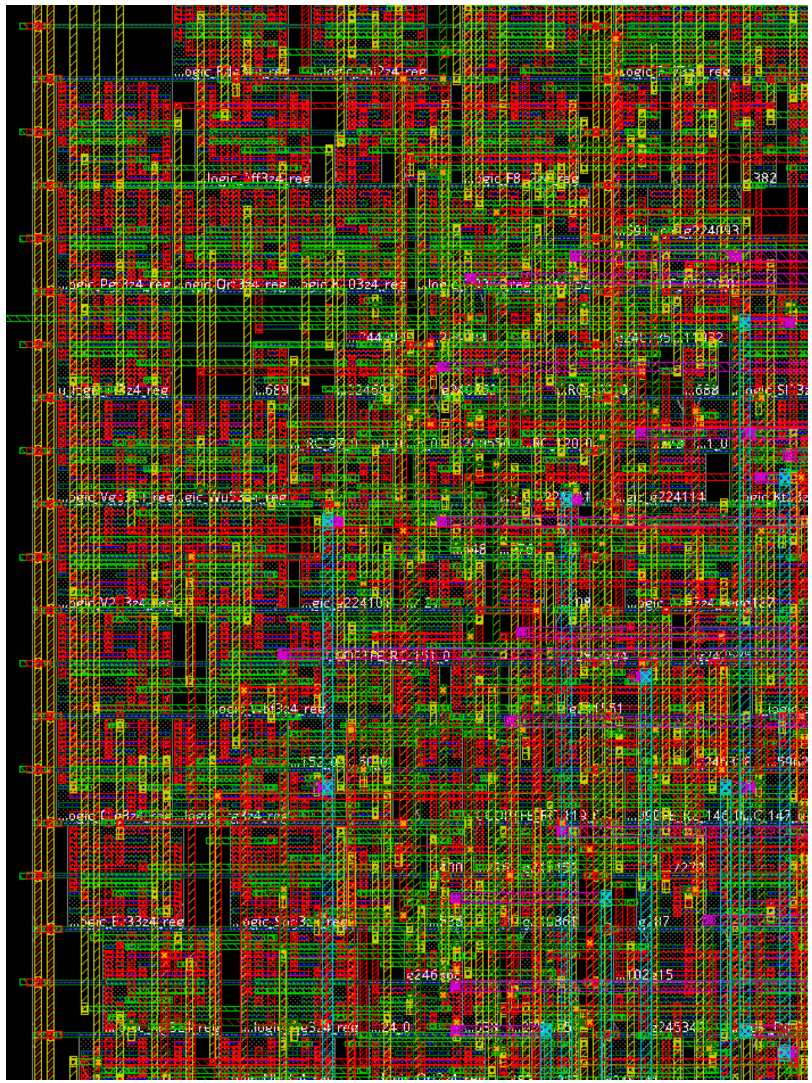
Illustration: from RTL to netlist

```
    ram_resp[initvar] = _GEN_1[1:0];
  `endif
  `ifdef RANDOMIZE_REG_INIT
  _GEN_2 = {1{$random}};
  _T_41 = _GEN_2[0:0];
  `endif
  `ifdef RANDOMIZE_REG_INIT
  _GEN_3 = {1{$random}};
  _T_43 = _GEN_3[0:0];
  `endif
  `ifdef RANDOMIZE_REG_INIT
  _GEN_4 = {1{$random}};
  maybe_full = _GEN_4[0:0];
  `endif
  end
`endif
  always @(posedge clock) begin
    if(ram_id__T_52_en & ram_id__T_52_mask) begin
      ram_id[ram_id__T_52_addr] <= ram_id__T_52_data;
    end
    if(ram_resp__T_52_en & ram_resp__T_52_mask) begin
      ram_resp[ram_resp__T_52_addr] <= ram_resp__T_52_data;
    end
    if (reset) begin
      _T_41 <= 1'h0;
    end else begin
      if (do_enq) begin
        _T_41 <= _T_59;
      end
    end
    if (reset) begin
      _T_43 <= 1'h0;
    end else begin
      if (do_deq) begin
        _T_43 <= _T_64;
      end
    end
    if (reset) begin
      maybe_full <= 1'h0;
    end else begin
      if (_T_65) begin
        maybe_full <= do_enq;
      end
    end
  end
endmodule
module Queue_11(
  input   clock,
  input   reset,
  output  io_enq_ready,
  input   io_enq_valid,
  input  [3:0] io_enq_bits_id,
  input  [63:0] io_enq_bits_data,
  input  [1:0] io_enq_bits_resp,
```

```
FA1STKD1 g802651(.A (n_6841), .B (r246), .CI (r187), .CO
     (UNCONNECTED3563), .S (n_67106));
FA1STKD1 g802652(.A (r726), .B (r689), .CI (n_6764), .CO
     (UNCONNECTED3564), .S (n_67107));
FA1STKD1 g802653(.A (r477), .B (r410), .CI (n_6825), .CO
     (UNCONNECTED3565), .S (n_67108));
FA1STKD1 g802654(.A (r330), .B (r228), .CI (n_6629), .CO
     (UNCONNECTED3566), .S (n_67109));
FA1STKD1 g802655(.A (r1910), .B (r1898), .CI (n_6566), .CO
     (UNCONNECTED3567), .S (n_67110));
AOI21D1 g802656(.A1 (n_67111), .A2 (n_41783), .B (n_43033), .Z
     (n_67112));
INR2D0 g802657(.A1 (n_32181), .B1 (n_50676), .ZN (n_67111));
AOI21D1 g802658(.A1 (n_67113), .A2 (n_63733), .B (n_34644), .Z
     (n_67114));
INR2D0 g802659(.A1 (n_32183), .B1 (n_48554), .ZN (n_67113));
AOI21D1 g802660(.A1 (n_67115), .A2 (n_41900), .B (n_41124), .Z
     (n_67116));
INR2D0 g802661(.A1 (n_31794), .B1 (n_48551), .ZN (n_67115));
AOI21D1 g802662(.A1 (n_67117), .A2 (n_41497), .B (n_38424), .Z
     (n_67118));
INR2D0 g802663(.A1 (n_31798), .B1 (n_46304), .ZN (n_67117));
AOI21D1 g802664(.A1 (n_67119), .A2 (n_41880), .B (n_38427), .Z
     (n_67120));
INR2D0 g802665(.A1 (n_31796), .B1 (n_46301), .ZN (n_67119));
AOI21D1 g802666(.A1 (n_67121), .A2 (n_41766), .B (n_38284), .Z
     (n_67122));
INR2D0 g802667(.A1 (n_31792), .B1 (n_46298), .ZN (n_67121));
AOI21D1 g802668(.A1 (n_67123), .A2 (n_63733), .B (n_38286), .Z
     (n_67124));
INR2D0 g802669(.A1 (n_31645), .B1 (n_46295), .ZN (n_67123));
AOI21D1 g802670(.A1 (n_67125), .A2 (n_40308), .B (n_34632), .Z
     (n_67126));
INR2D0 g802671(.A1 (n_31800), .B1 (n_46307), .ZN (n_67125));
AOI21D1 g802672(.A1 (n_67127), .A2 (n_41900), .B (n_34732), .Z
     (n_67128));
INR2D0 g802673(.A1 (n_31647), .B1 (n_44240), .ZN (n_67127));
XNR2D0 g802674(.A1 (n_67129), .A2 (n_9822), .ZN (n_67130));
XOR2D1 g802675(.A1 (n_1466), .A2 (r1101), .Z (n_67129));
FA1STKD1 g802676(.A (n_2508), .B (n_6519), .CI (n_25738), .CO
     (UNCONNECTED3568), .S (n_67131));
FA1STKD1 g802677(.A (n_17082), .B (n_4392), .CI (n_1196), .CO
     (UNCONNECTED3569), .S (n_67132));
FA1STKD1 g802678(.A (n_829), .B (n_4396), .CI (n_14984), .CO
     (UNCONNECTED3570), .S (n_67133));
FA1STKD1 g802680(.A (n_20329), .B (n_3475), .CI (n_161), .CO
     (UNCONNECTED3571), .S (n_67135));
FA1STKD1 g802681(.A (n_3524), .B (n_3405), .CI (n_18483), .CO
     (UNCONNECTED3572), .S (n_67136));
FA1STKD1 g802682(.A (r982), .B (r904), .CI (n_14487), .CO
     (UNCONNECTED3573), .S (n_67137));
FA1STKD1 g802683(.A (n_14683), .B (n_1494), .CI (n_2133), .CO
     (UNCONNECTED3574), .S (n_67138));
FA1STKD1 g802684(.A (n_11698), .B (n_1923), .CI (n_501), .CO
     (UNCONNECTED3575), .S (n_67139));
FA1STKD1 g802685(.A (n_17497), .B (r1299), .CI (r1266), .CO
```

Illustration: placed & routed design (ASICs)





Design flow: ASICs vs. FPGA

- **Almost** the same: we always need to place & route something
- **Almost** – difference is what we need to place & route:
  - in ASICs we place standard cells – i.e. predefined logic gates, assembled from basic transistors to fulfill different logical (AND2, AND3, AOI) & electrical requirements (drive strength of the cell, i.e. capacity of the current delivery)
  - in FPGAs we place (& route) logic cells (LC), memory blocks or pre-existing DSPs
    - LC placement is LUT/SRAM **allocation** since they are already placed on the FPGA die in a regular fashion
- For FPGAs we also need to decide on LC content, so the optimisation process is different (Can you explain?)
- Routing in FPGAs is about choosing the pre-existing routes and connecting PSMs (in an ASIC we create dedicated routes)
- Final output is also different: for FPGAs we have bitstreams (i.e. configuration files) and for ASICs we have databases of drawings!

6. VHDL primer

Basic syntax elements

```
1  -- 1. Comments
2  -- If you start the line with this, you can write whatever you want
3  -- in language you want (this is not VHDL) ===> super useful
4
5  -- 2. Identifiers
6  -- Can contain any character but have to start with a letter!
7  -- And they are not case sensitive !
8  -- Some examples
9  My2Name                              -- good
10 my2nAME                              -- good, but same as previous
11 2My2Name                             -- not good
12
13 -- 3. Numbers
14 -- Can use various (useful) bases: decimal (10), 2, 8 and 16
15 -- Number that precedes the sign # indicates the base
16 -- Some examples (make sure you know how to convert)
17 16#C4#                               -- int hexadecimal and in decimal?
18 8#304#                               -- int octal and in decimal 196
19 2#1100_0100#                         -- binary 8 bits
20 16#F.FF#E2                           -- float : 4095.0
```

Data Types

- Data types are extremely important in VHDL
  - Type conversions are possible but take extra care when doing it!

```
1  -- Binary - predefined types
2  bit                                          -- True False only
3  std_logic                                    -- True False plus others,
4                                               -- like don't care etc.
5                                               -- you will use this one extensively
6
7  -- User defined types
8
9  -- Integers
10 type byte_int is range 0 to 255;             -- 8b unsigned
11 type signed_word_int is range -32768 to 32767; -- 8b signed
12 bit_index is range 31 downto 0;              -- 32 bit vector
13 -- Floats
14 type signal_level is range -10.00 to +10.00;  -- signed float
15 probability is range 0.0 to 1.0;
16 -- Arrays, Matrices
17 type word is array (31 downto 0) of bit;
18 type memory is array (address) of word;
19 type transform is array (1 to 4, 1 to 4) of real;
20 type register_bank is array (byte range 0 to 132) of integer;
```

- VHDL is even more strongly typed (what is this?) than C
- Can you explain why we need precise data types in HW?

Assignments, operators & expressions

```
1  -- 1. Assignment
2  -- Use symbol <= to assign a value to an identifier
3  -- Value could be constant or another identifier
4  -- Think of assignment as soldering a piece of wire between two ends
5  -- Obviously left and right of <= should match
6  -- Example of use:
7  value <= tmp;
8
9  -- 2. Logic operators
10 -- AND, OR, XOR, NAND, NOR, XNOR, NOT
11 -- Example of logic expression and assignment
12 c <= a or b;
13
14 -- 3. Arithmetic operators
15 -- Usual ones: +, * , / -
16 -- Example:
17 c <= a + b;
18
19 -- 4. Relations
20 =           -- equal
21 /=          -- unequal
22 >           -- strictly greater then
23 <           -- strictly less then
```

Basic building block

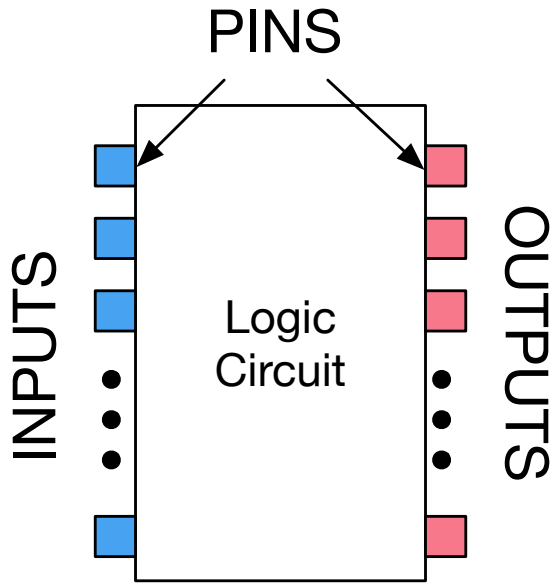
- In VHDL basic building block is called entity
- An entity will define a HW module in your design
  - ▷ Good practice is to write one HW entity (module) per VHDL file
- We can define the following template for a HW entity:

```
1  -- 1. Libraries provide various definitions
2  --    Signal types (data types)
3  --    Logic/arithmetic libraries (these are mandatory)
4  library ieee;
5  use ieee.std_logic_1164.all;
6
7  -- 2. Entity Declaration
8  -- a) Module name
9  -- b) Definitions of all inputs/outputs (module interface)
10 entity myBlackBox is port(
11    -- IO definition
12    ...
13 );end myBlackBox;
14
15 -- 3. Architecture: how to compute outputs, or state variables
16 --    Boolean functions of inputs and/or internal variables
17 architecture synthesis of myBlackBox is
18    -- Circuit description
19    ...
20 end synthesis;
```

Entity & circuit view basic building block

- Module interface defines ports: define name, direction and type (always see this as one or more wires)
- Ports are used to establish connection (IO) with the external world (other blocks or outside the chip)
- Ports are logical entities, once they become physical, i.e. after implementation, we speak of pins (shapes that are connected, e.g. soldered, to wires)

```
1  -- Port definitions (MUST) have:
2  --      a) Name (identifier)
3  --      b) certain data type
4  --         and
5  --      c) direction
6  --         (input, output or bidirectional)
7
8  entity myBlackBox is port(
9  a    : IN std_logic;
10 b    : IN std_logic;
11 c    : IN std_logic;
12 z    : OUT std_logic
13 );end myBlackBox;
```



Example of a simple combinatorial circuit

- We use assignment operator (<=) to connect inputs to outputs using some logic in between!
- Example – implement logic function: $z(a,b,c) = a \cdot b \cdot c + a' + b' \cdot c'$

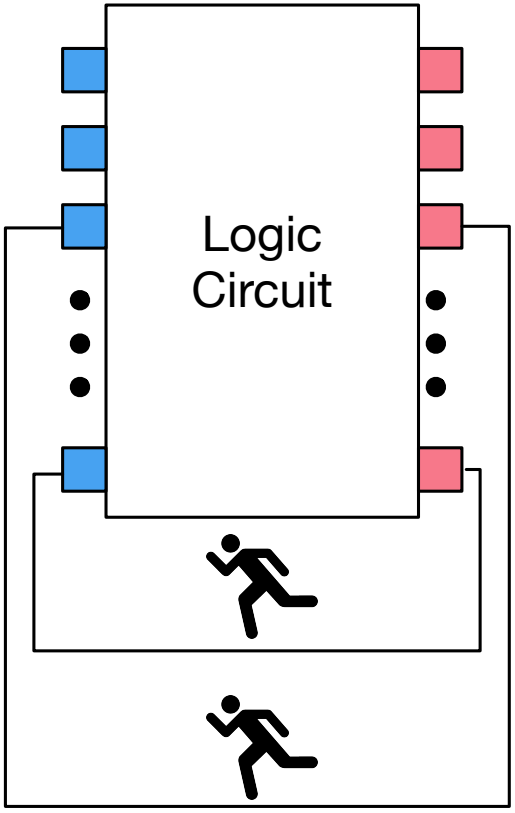
```
1  -- Libraries
2  library ieee;
3  use ieee.std_logic_1164.all;
4  -- Entity declaration: we need 1 output and 3 inputs. Why?
5  entity myBlackBox is port(
6  a    : in std_logic;
7  b    : in std_logic;
8  c    : in std_logic;
9  z    : out std_logic
10 );end myBlackBox;
11 -- Architecture definition
12 architecture synthesis of myBlackBox is
13 -- Here you can define (internals) signals for this entity
14 -- These are like local variables in C, but do not forget :
15 -- THESE ARE SIGNALS, NOT VARIABLES, more on this later
16 -- And when we speak of signals, think of wires !!!
17 -- None here, since we map inputs to output
18 begin
19 -- Boolean expression(s) ===> assignment: inputs map to outputs, using <=
20 z <= (a and b and c) or (not a) or ((not b) and not(c))
21 end synthesis;
```

- We will further discuss assignments in the next two lectures

Port assignment: fundamental rule!

No output on the right hand side of the assignment!

- This is because it would mean **feedback** on signals! (output connected to input)
- Circuits with feedback are sequential, but they are asynchronous! & asynchronous circuits are subject to race conditions
  - ▷ What are race conditions?
- If race conditions are not handled properly the circuit will not operate as it should!!!



Race conditions are solved using synchronization, resulting in synchronous sequential logic circuits

Example of the assignment

```
1  -- Libraries
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  -- Entity declaration
6  entity myBlackBox is port(
7  b    : IN std_logic;
8  c    : IN std_logic;
9  zo   : OUT std_logic
10 );end myBlackBox;
11
12 -- Architecture definition
13 architecture syn of myBlackBox is
14 begin
15 -- Boolean expression(s)
16 zo <= zo and b and c
17 end synthesis;
```

- Knowing this you can conclude that this model will never work
- Synthesis/simulation tools will not allow you to do that and you will get an error
- Or, this is totally NORMAL in programming languages
  - ▷ Can you explain why it is different here?

ALWAYS REMEMBER:
VHDL IS NOT A PROGRAMMING LANGUAGE !!!
(forget about C/C++/Python etc. when designing circuits)

Although some literature/people do say so ... but they are wrong!

We do not program FPGAs,
we configure them with some logic

Sequential circuits – overview

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity myBlackBox is port(
5  ... -- IO definition
6  );end myBlackBox;
7
8  architecture syn of myBlackBox is
9  -- first begin to start module
10 begin
11 -- 1. Here we define the process p_1
12 -- p_1 is the name used to track
       process
13 -- during simulation and/or
       implementation
14
15 -- Sensitivity list
16 -- Given between the parentheses
17 -- immediately after
18 -- the process statement
19 --
20    p_1 : process(...)
21    -- Note the second begin
22    -- after the process statement
23       begin
24       -- Statements that describe
25       -- sequential machine
26       --
27       end process;
28 end syn;
```

- Statement `process` is used to describe sequential synchronous & asynchronous circuits
- Here we assume **synchronous circuits only**
- Activation of a process is done through a sensitivity list
- If one of the signals in this list changes value, this will trigger functional description evaluation (statements within the process)
- Attention! process "activation" above refers to simulation only! (more on this later)

VHDL `process` and clocks

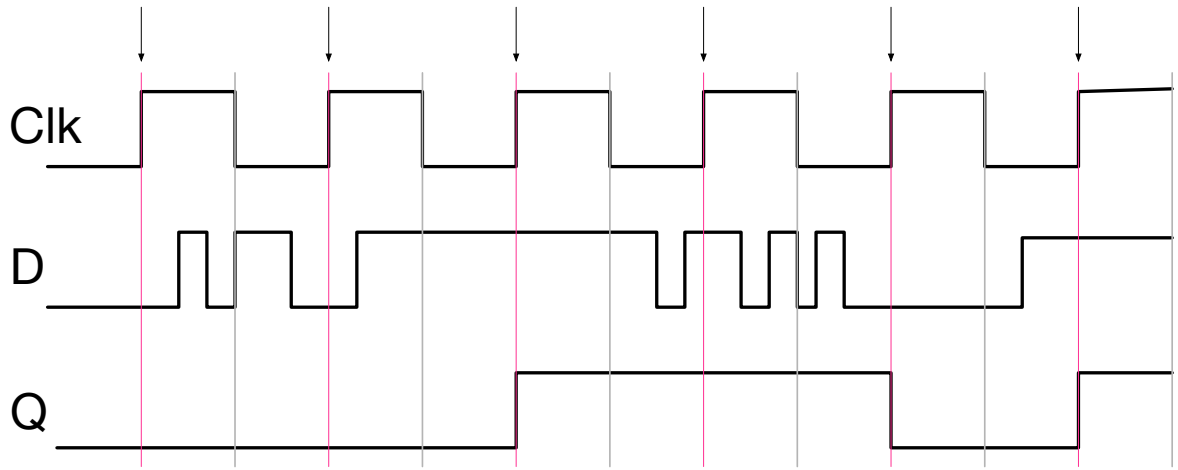
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity myBlackBox is port(
5  ... -- IO definition
6  );end myBlackBox;
7
8  architecture syn of myBlackBox is
9  -- first begin to start
10 -- module description
11 begin
12
13 -- p_1 has the sensitivity list
14 -- in this list we put CLK
15
16 p_1 : process(clk)
17    begin
18    -- Detect clk rising edge
19    if (clk'event) and (clk = '1')
         then
20       --  if branch taken
21    end process;
22 end syn;
```

- If the signal in the sensitivity list is some periodic signal, such as clock (CLK), then if the "branch" is taken, whatever occurs after will occur only during the transition from 0 to 1 of CLK i.e. during the rising edge
- Making circuits that react on the rising edge is a fundamental property of sequential logic circuits
- Flip-flop memories are typical example of such circuits

Current dense IC could have millions of logic gates, and hundreds of thousands of Flip-Flops

Sequential circuits – D Flip-Flop

- Previous VHDL template can be used to describe the behavior of the mother of all storage elements (bi-stables): D Flip-Flop (DFF)
- DFF has two inputs `D`, `CLK` and one output `Q`
- The output `Q` will have the value of `D` only when the signal `CLK` changes the value from 0 to 1
- Outside the rising edge of the `CLK`, DFF keeps the old value of `D`, no matter what is the value of `D`; this is the mechanism of memorization



- Note that if `D` and `CLK` change simultaneously (inputs could be seen as random variables, so this could happen), it is the previous value that will be stored in the DFF (much more on this later on)

Sequential circuits – D Flip-Flop

- Let's use sequential system template (slide 54), and the process that will describe the DFF will be thus sensitive to `CLK`

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity DFF is port(
5  D, clk   : in std_logic;
6  Q        : out std_logic
7  );end DFF;
8
9  architecture syn of DFF is
10 begin
11 p_1 : process(clk)                          -- Clk in the sensitivity list
12    begin
13    if (clk'event) and (clk = '1') then      -- Detect clk rising edge
14       Q <= D;                               -- Assign output
15    end process;                             -- If not, do nothing, keep previous value
16 end syn;
```

- If you interpret the model above in human language you will see that the model describes the behavior of the system; this is why VHDL modeling is sometimes referred to as behavioral modeling
- Designer describes the system behavior in relatively abstracted way, & the SW tools will understand the relative HW component

Insisting on memorization mechanism

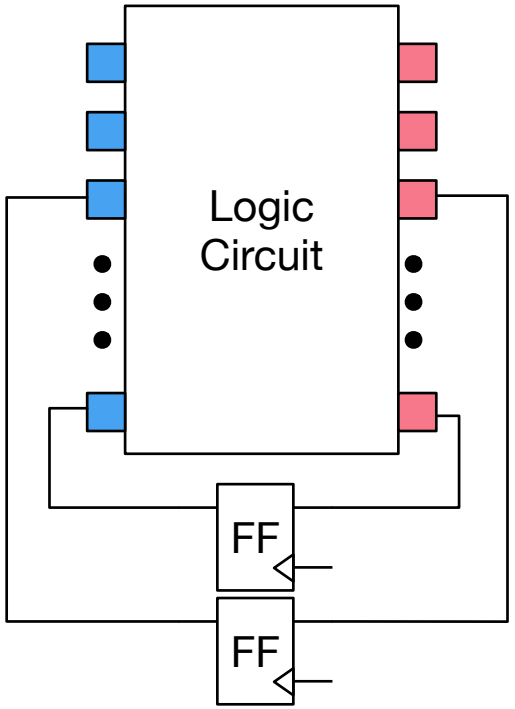
- Behavior of the DFF is understood by the synthesis engine that will infer a memory element; How does it do?
- Look at the model:

```
1 process (clk)
2   begin
3   -- 1. Detect rising edge
4     if (clk'event) AND (clk = '1') THEN
5      -- 2. If rising edge: assign to output q input d
6       Q <= D;
7     end if;
8 end process;
```

- We first detect rising edge (1.) and then we "copy" the input `D` to output `Q` (2.)
- Since there is no assignment for "outside the rising edge" (other branch for the `if` statement), the system should keep the previous value; or `Q` and `D` are wires: we must insert a memory to keep that value for the future
- We have just described a behavior that could be understood by synthesis

Synchronous sequential logic circuits

- To avoid race conditions we add synchronous elements on the signal path: **a Flip-Flop** (FF)
- FF will be active on the rising edge of a periodic signal (`clk`); when this occurs value on the input will be stored in the FF
- This value is kept until the next rising edge occurs; any change on the input will be ignored until the next rising edge (transients are filtered out)



- This is how we make a difference between previous and current value of the same logical net (i.e. wire) in sequential digital systems that are now synchronous (due to FFs)
- The above mechanism is also used for data transfer between different functional blocks – remember RTL?!

Multiple processes

- Logic circuits are concurrent by nature, so more then one process can be defined inside a single VHDL module [3]
- Processes are **independent** one from the other
- Example: 1 entity has 3 processes; synthesis will produce 3 independent logic circuits

```
1  entity myBlackBox is port(
2  ... -- We should have here some I/O definitions
3  );end myBlackBox;
4
5  architecture synthesis of myBlackBox is
6  begin
7   p_1 : process(a,b)
8    begin -- some VHDL below
9    end process;
10  p_2 : process(c,d)
11   begin -- some VHDL below
12   end process;
13  p_3 : process(clk)
14   begin -- some VHDL below
15   end process;
16 end synthesis;
```

[3] Good design practice would be to minimize the number of processes per per module

7. Practical examples

Example 1 – Simple logic circuit

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity circuit1 is port(                                  -- 1. Def. of entity
5     a : in std_logic;                                      -- 2. Inputs  outputs
6     b : in std_logic;
7     c : in std_logic;
8     z : out std_logic
9  );end circuit1;
10
11 architecture synthesis of circuit1 is
12    begin
13       signal tmp : std_logic;
14       tmp <= (a and (not b)) or ((not a) and c);          -- 3. Assignment
15       p_1 : process(clk)
16          begin
17          if (clk'event) and (clk = '1') then              -- 4. Detect clk rising edge
18          z <= tmp;                                        -- 5. Assignment
19       end process;
20 end synthesis;
```

- Analyze different comments in the model above
- What type of logic circuit this is?
- Derive the circuit schematic from the model below

Example 2 – Slightly more complex circuit

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity circuit1 is port(
4     a : in std_logic;
5     b : in std_logic;
6     c : in std_logic;
7     d : in std_logic;
8     z1 : out std_logic ;                          -- 1. We add output
9     z2 : out std_logic
10 );end circuit1;
11 architecture synthesis of circuit1 is
12 begin
13    signal tmp1 : std_logic;
14    signal tmp2 : std_logic;
15
16    tmp1 <= (a and (not b))  or ((not a) and c);     -- 2. Two logic functions
17    tmp2 <= (b and d))  or (a and c)
18
19    p_1 : process(clk)
20    begin
21       if (clk'event) and (clk = '1') then
22          z1 <= tmp1;                             -- 3. Assignments
23          z2 <= tmp2;
24 end process;
25 end synthesis;
```

- Analyze different comments in the model above
- How the above logic functions will be processed?
- How different this is from the previous example?

Example 3 – VHDL is not a programming language

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity circuit1 is port(
   a : in std_logic;
   b : in std_logic;
   c : in std_logic;
   d : in std_logic;
   z1 : out std_logic ;
   z2 : out std_logic
);end circuit1;
architecture synthesis of circuit1 is
begin
   signal tmp1 : std_logic;
   signal tmp2 : std_logic;
   tmp1 <= (a and(not b))or((not a)and c);
   tmp2 <= (b and d)) or (a and c);
   p_1 : process(clk)
      begin
      if (clk'event) and (clk = '1') then
         z1 <= tmp1;
         z2 <= tmp2;
   end process;
end synthesis;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity circuit1 is port(
   a : in std_logic;
   b : in std_logic;
   c : in std_logic;
   d : in std_logic;
   z1 : out std_logic ;
   z2 : out std_logic
);end circuit1;
architecture synthesis of circuit1 is
begin
   signal tmp1 : std_logic;
   signal tmp2 : std_logic;
   p_1 : process(clk)
      begin
      if (clk'event) and (clk = '1') then
         z1 <= tmp1;
         z2 <= tmp2;
   end process;
   tmp2 <= (b and d)) or (a and c);
   tmp1 <= (a and(not b))or((not a)and c);
end synthesis;
```

- Draw the schematic of the circuits above
- Explain the difference between these two models?
- How will this difference affect the circuit?