ELEC-H-409
Th05: Analysis of circuit performance: timing, power and area

Dragomir Milojevic
Université libre de Bruxelles

Today

1. Delays in logic circuits
2. Practical timing analysis and system performance
3. Design for performance
4. Design flow and Performance, Power & Area (PPA) in FPGAs
5. Physical implementation of FPGA circuits

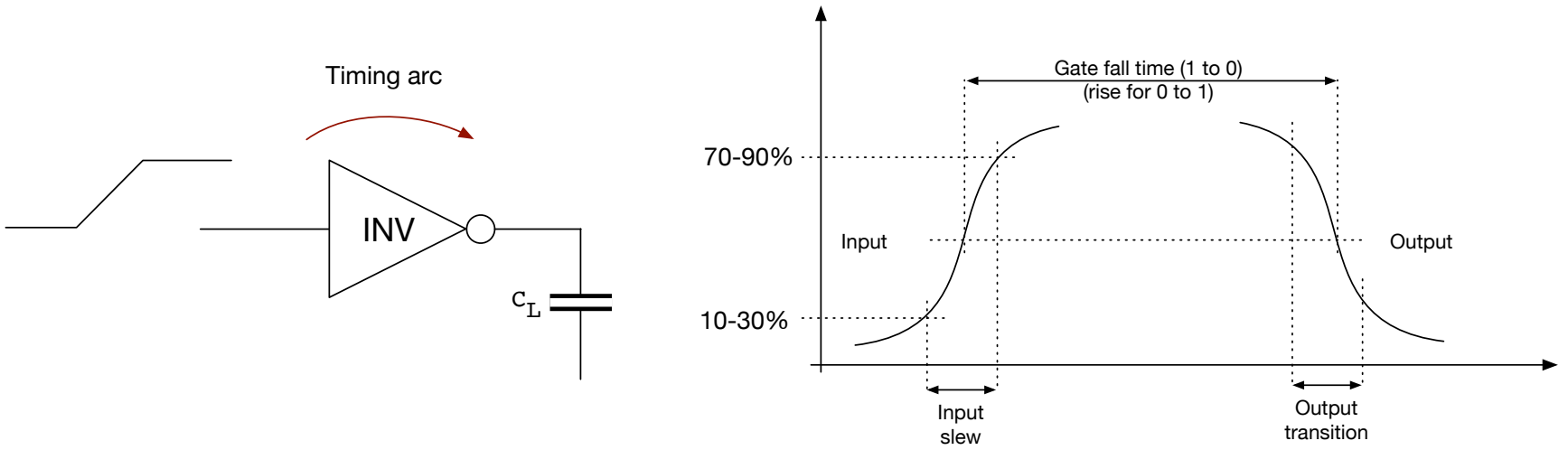
1. Delays in logic circuits

Circuit delays – overview

- Total circuit delay will depend on the delay of its components:
  A. Combinatorial delay – gates, memories, LUTs in FPGAs etc.
  B. Sequential delay – linked to flip-flops (FFs)
  C. Wire delay – due to interconnect (nets)
- Delays are calculated using resistance R, inductance L & capacitance C values of all components – parasitics
- **Exact delays** of the circuit will be known only after placement & route (P&R) of the design
- This is because during P&R the circuit will undergo different **circuit optimizations** driven by **design constraints**: form of logic circuit & gates could change, but the functionality will of course remain preserved
- If exact parasitics are not known, e.g. wire RLC during synthesis, they can be predicated using more or less complex estimators

A. Combinatorial delays 1/3

- We speak of timing arcs – paths from input(s) to output(s) for which we can define a delay; only one timing arc for an inverter:



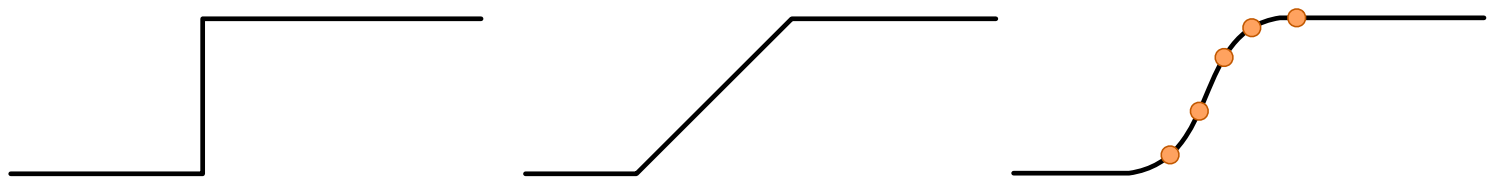
- Depending on the gate function and input data we speak of gate rise or gate fall time – time difference between input that reached 50% of it's value and the output reaching the same level
- Gate delay depend on:
  - ▷ Input transition time (slew) – time needed to move from 10 to 90% (or 30 to 70%for more recent CMOS) of the maximum value
  - ▷ Output load – input pin capacitance of the connected gate

A. Combinatorial delays 2/3

- Depending on how do we model transition times, we will have more or less accurate delay models:
  - Ideal circuits – no parasitics, no input transition times, zero load, so no rising or falling times $\rightarrow$ good for theory only
  - Linear models – combine transition time TT & load C_L; better approximation, used in the past for old CMOS technologies:

$$\delta = \delta_0 + \delta_1 TT + \delta_2 C_L$$

    where δ_i are constants depending on the technology
  - Non-Linear Delay Models (NLDM) – approximated with a number of points stored as 2D matrices: delay is specified as a function of TT and C_L



A. Combinatorial delays 3/3

- Once the logic functionality is assembled using transistors devices, electrical simulations are performed to extract different points of the NDLM model that are stored in a technology library, example:

```
pin (OUT) {
   max_transition : 1.0;
   timing() {
      related_pin : "INP1";
      timing_sense : negative_unate;
      cell_rise(delay_template_3x3) {
         index_1 ("0.1, 0.3, 0.7"); /* Input transition */
         index_2 ("0.16, 0.35, 1.43"); /* Output capacitance */
         values (
         /* 0.16 0.35 1.43 */
         /* 0.1 */ "0.0513, 0.1537, 0.5280"
         /* 0.3 */ "0.1018, 0.2327, 0.6476",
         /* 0.7 */ "0.1334, 1.43*/0.2973, 0.7252"
         );
```

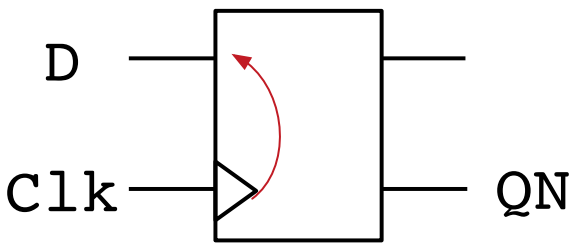
- During timing analysis SW tool will interpolate between different points – gate transistor model is abstracted (remember Gajski) & simulation process at gate-level is significantly accelerated
- For FPGAs circuits these are stored in (proprietary) device models

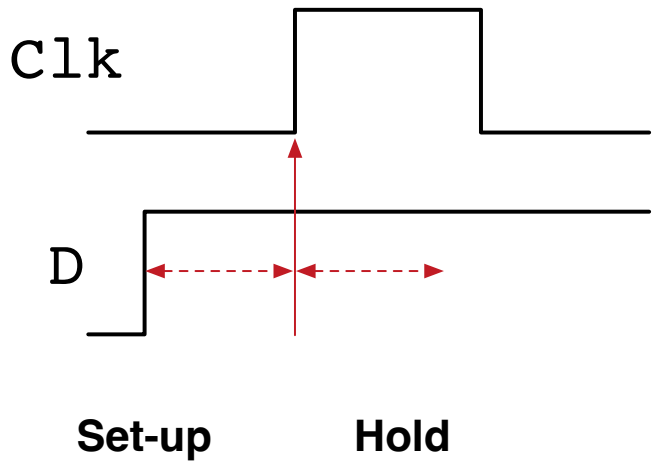
B. Timing for Flip-Flops (FF) 1/3

- Specification of D-FF:

  "*if control and data input signals change at the same time, it will be the value of the input data just before the control signal change that will be stored*"

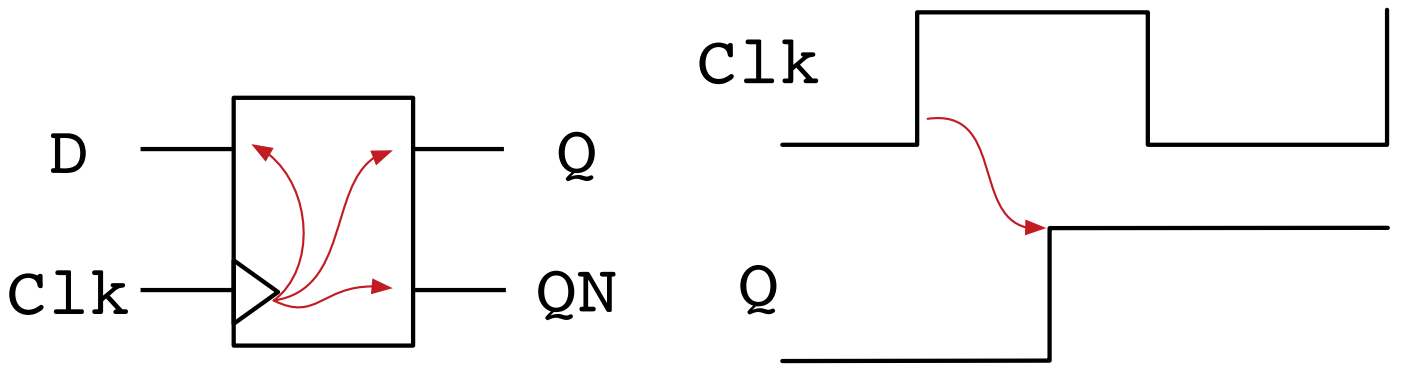
- This assumption is important to clarify what will happen in case of simultaneous events on `Clk` and `D` (inputs are seen as random)
- In another words input data must arrive before clock rising edge of the control signal (i.e. clock) – set-up time; data should also be kept stable sometime after the clock edge – hold-time





B. Timing for Flip-Flops (FF) 2/3

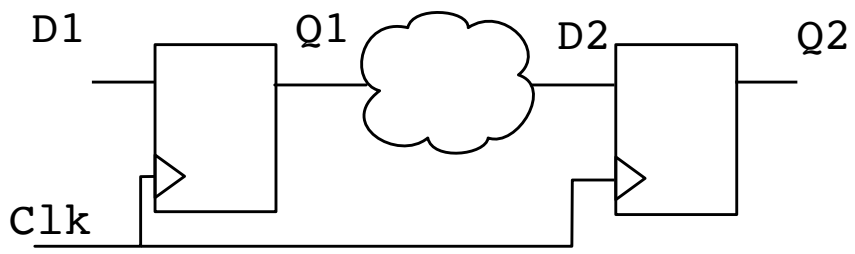
- If set-up and hold times are not respected, FF will NOT operate correctly – value stored is not deterministic, i.e. it can't be predicted, could be 0 or 1 (not handy); functional correctness of the memory element is compromised – metastability problem
- Any FF will have synchronous outputs (`Q`/`Qn`); data on these pins will appear after certain delay with respect to the clock edge – clock to output

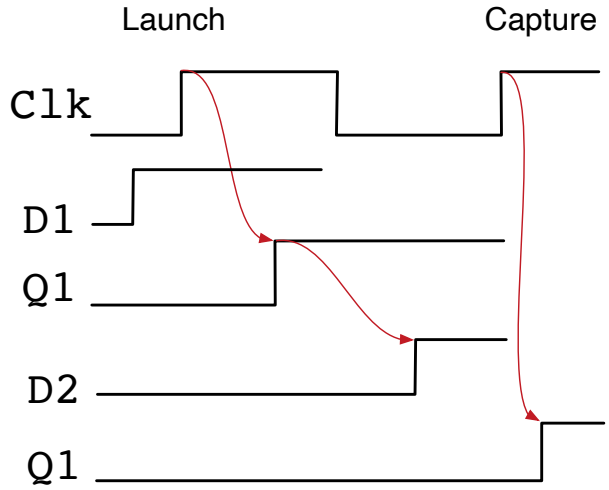


- Absolute set-up & hold and clock to output time will depend on CMOS device, just like for the INV; they will be characterized upfront and relevant information stored in technology library

B. Timing for Flip-Flops (FF) 3/3

- Typical digital circuits (even combinatorial ones) are implemented as a set of input (source) and output (destination) FFs with gates or LUTs in the cloud in between – this is why we speak of digital systems described as Register Transfer Logic – RTL





- On a first rising edge – clock lunch edge data is captured on `D1` & will become available at source FF output after some time
- Data then travels through logic & wires (LUTs) & should arrive before the next capture rising edge of the destination FF

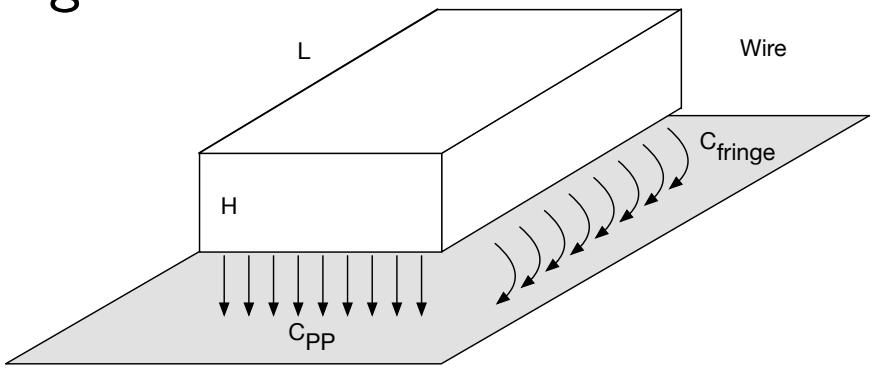
C. Wire delays 1/5

- In the early years we could neglect wire delays: they were too small compared to gate delays
- With CMOS scaling wire delays become more important: today not uncommon to have them contribute for 50% of critical path!
- Delays in wires depend on:
  - wire material – known in advance (Al, Cu, but other materials too)
  - wire cross-section – known in advance for ICs (including FPGAs)
  - wire length – this is design and place & route dependent
- It is thus because of this dependency to routing that the actual circuit timing could be known only after routing
- Timing after previous stages of the design flow will use wire delays models approximating the length
  - After **synthesis** – length predicted based on gate count & gate connectivity
  - After **placement** – length based on gates, LUTs x, y positions
  - After **routing** – real things since real physical nets (wires)

C. Wire delays 2/5

- For a wire of an arbitrary length L, width W & height H capacitance can be calculated using:

$$\begin{aligned} C &= C_{pp} + C_{fringe} \\ &= \frac{\varepsilon_{di}}{t_{di}} WL + \frac{2\varepsilon_{di}\pi}{log(t_{di}/H)} \end{aligned}$$



where C_{pp} & C_{fringe} is parallel plate & fringe capacitance, t_{di} is dielectric thickness and ε_{di} dielectric constant

- Resistance is computed using:

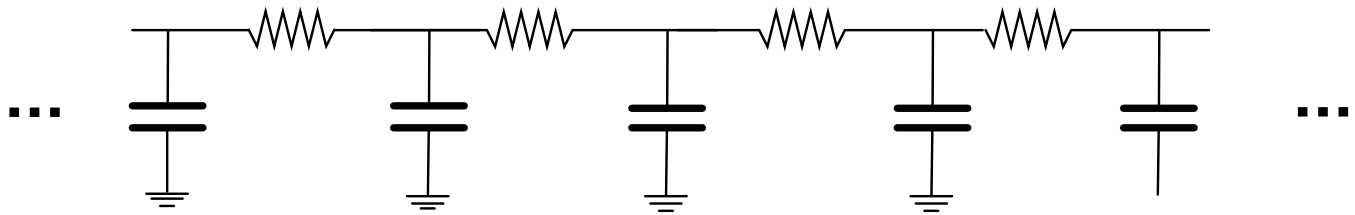
$$R = \frac{\rho L}{A} = \frac{\rho L}{HW}$$

where ρ is resistivity $[\Omega m]$

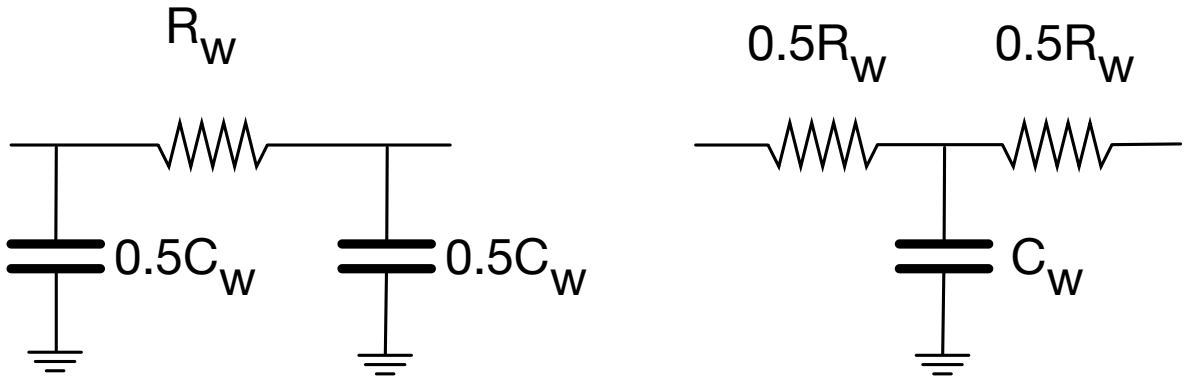
- Note that R is inversely & C directly proportional to W; note also that RC delay is proportional to L^2 (crucial for any IC)

C. Wire delay 3/5

- RC delay of a wire is represented using a distributed RC tree:



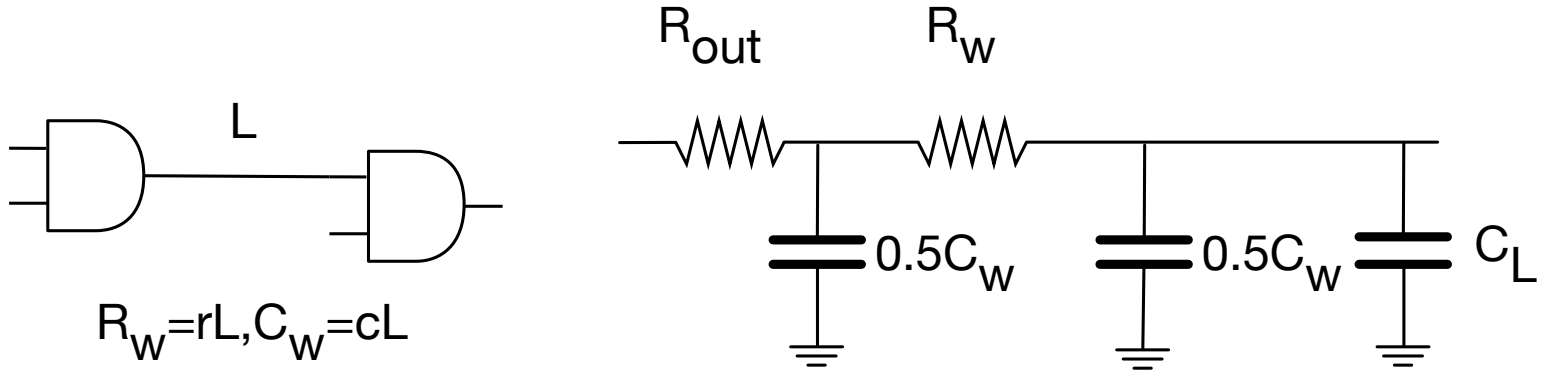
- Often simplified π-lumped model used instead – total wire capacitance split in half with a total wire resistance in the middle



- Alternative T-model splits the resistance value in half, and keeps total wire capacitance C_w in the middle

C. Wire delay 4/5

- We can then calculate a delay of a source and sink gate connected with a piece of wire of length L (so $R_w = rL$ and $C_w = cL$)
- Source gate has output resistance R_{out} driving a π-model of a wire connected to input pin with a cap value of C_L



- Delay of the circuit on the right can be then calculated using Elmore's delay approximation (quite rough):

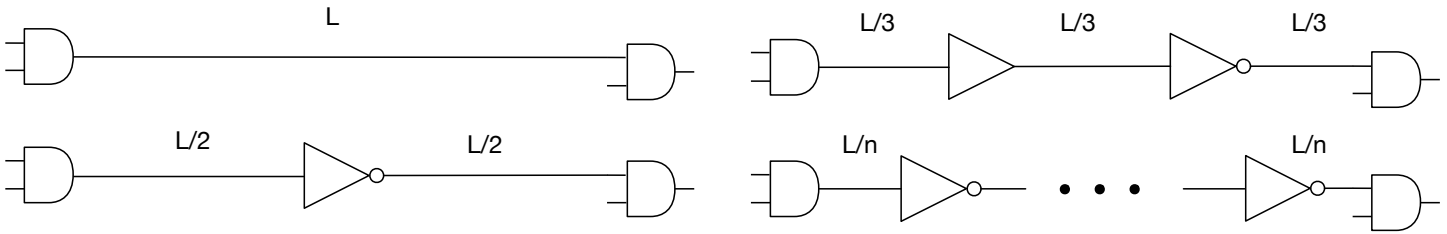
$$D = \frac{1}{2} R_{out} C_w + (R_{out} + R_w)(\frac{1}{2} C_w + C_L)$$

C. Wire delay 5/5

- Total C seen by the driver gate will determine the amount of current drawn; more $C \rightarrow$ more current is to be delivered
- We have more C:
  - If wires are smaller (scaled wires – technology), longer (design)
  - If we have more input capacitance at sink gate (technology)
  - If we have Fan-Out (FO) > 1, i.e. more than one gate connected to the same output (design dependent)
- To deliver necessary current we need to size gates; logic functionality is the same but gate internal architecture is more complex – we put more transistors in parallel to increase current delivery capacity; gates have different drive strengths
- Higher driver strength gates can deliver more current but for more area & power per gate
- To enable current delivery flexibility FPGAs implement fixed length connections with predictable loads – this comes at a price, reason why FPGAs will be always less optimal than ASICs

Impact of the wire length on delays

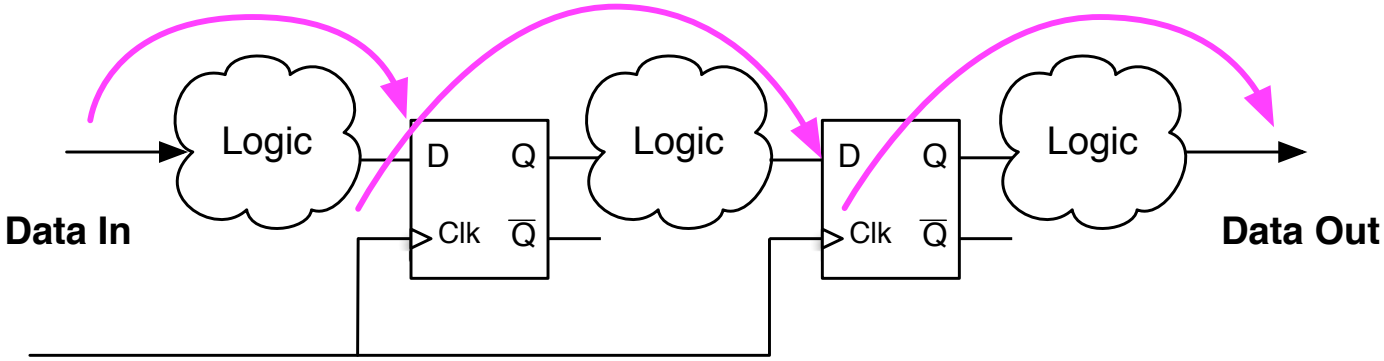
- If we scale logic gates, we need to scale wires too; cross section should get smaller but delay per unit length goes up – wire will contribute more to critical path; very little can de done here, this is fundamental problem of CMOS scaling
- Delay in the wire has quadratic dependence on L
- To reduce the impact of L we break down (long) wire by adding buffers or a even number of inverters – buffer insertion
  - Why even number of inverters?



- You will add extra inverter delays, but total delay is now linear function of $\frac{L}{n}$, where n is the number of segments
- This can not go on forever, at one point wire delay is small enough not to justify insertion any more; this can be calculated

Timing at complete circuit level

- For sequential circuits with FFs, with or without feedback, so including combinatorial logic, we can define different path group delays that need to be checked once the circuit is completed
  - Input port to register – from the IC pin to the first FF
  - Register-to-register – from clock input of the launching FF and it's output, to the capturing FF input and next clock edge
  - Register-to-output – from clock and FF output to output pin



- In FPGAs inputs can be registered at IOB level, (not automated); that way you make sure that input/output delays are minimal
- Because of many IOs and especially many register-to-register paths we need automation to analyze timing at circuit level

2. Practical timing analysis and system performance

Timing Analysis – TA (1/3)

- Place & route process is timing driven, so timing is calculated but locally on a limited parts of the circuit (divide & conquer)
- TA at circuit level is a separated design flow step using dedicated EDA tools (e.g. ModelSim, VCS or similar)
- During TA we derive all individual component delays (gates, wires, FFs) in the design & sum them for **for all paths** in the system; sort them, output worst ones & look at the worst
- TA is one of the most important success criteria of the design flow: we always want the maximum possible performance at lowest price (area & power)
- Two kinds of TA:
  - Static Timing Analysis (STA) – static means independent of inputs and assuming worst case; STA is thus pessimistic
  - Dynamic Timing Analysis – logic paths are data dependent; by assuming actual input vectors we can derive the actual logic paths; more accurate analysis at expense of more complex computations

Timing analysis (2/3)

- STA needs following inputs;
  - Design netlist – it defines logical content & connectivity plus placed & routed design data base which defines geometry aspect of the circuit
  - Timing constraints – target clock(s) period, input / output delays, explicit delays of certain paths, clock assumptions etc. supplied in text files with more or less standardized format
  - Technology information – gate and & wire delays for ASICs, LUTs and wires in FPGAs; information provided by technology manufacturer (CMOS foundry or FPGA vendor)
- During place & routing timing constraints are used to drive the implementation process; this is what we wish should happen at circuit level
- During STA same constraints are used to see how close or far final circuit is from the objective – difference between what wishes & reality

Timing analysis (3/3)

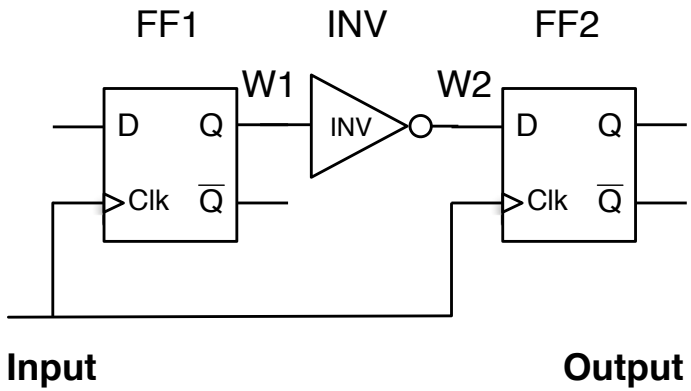
- Worst Negative Slack (WNS) – difference between actual arrival time in the critical path & the constraint time (aka worst delay)
  - If $WNS < 0$ constraints are too optimistic!
  - If $WNS > 0$ by little – timing is met
  - If $WNS >> 0$ we have a margin, you could actually increase the actual F or re-run everything with more aggressive timing constraints
- Total Negative Slack (TNS) – sum of slacks of all failed paths
  - Shows constraints relevance at system level; difference between few paths that fail by lot (you may optimize a given block), or many paths that fail by little (constraints are maybe to aggressive)
- To get a realistic idea of timing you may do the following:

$$TP_{i+1} = TP_i - WNS$$

  in the next iteration your target period TP_{i+1} is a corrected value of your previous target period TP_i with the value of WNS

Maximum frequency

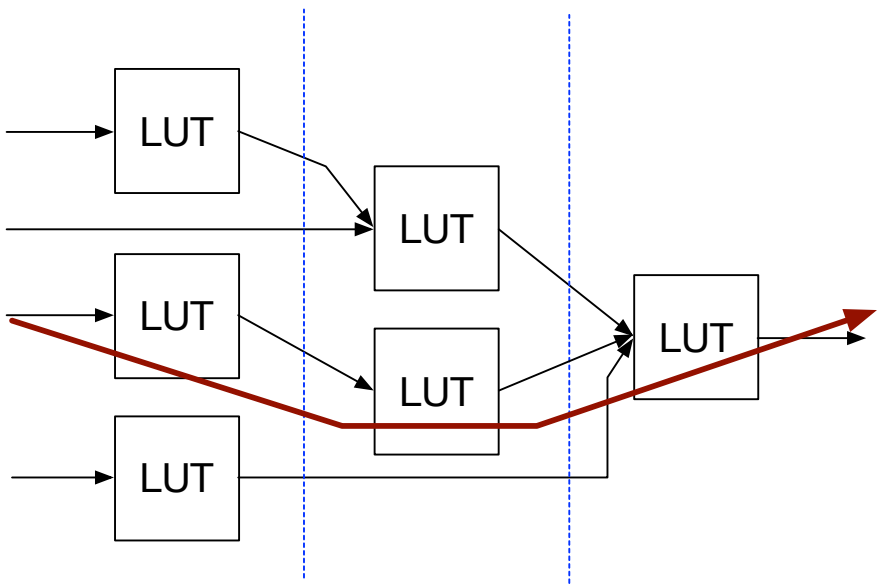
- In FPGA design flows STA is automatically run at each step and max frequency F_{max} is always reported (loot at generated txt logs)
- When you will NOT have the F_{max} reported?
- Assume valid data on input pin of the launch FF, so set-up and hold time with respect to rising `Clk` edge, we can then compute minimum clock period T_{min} for a given circuit
- T_{min} is the sum of delays:
  - Clock to output of first FF (FF1)
  - Wire delay W1 (FF1.Q to INV.I)
  - Propagation delay in the inverter (INV)
  - Wire delay W2 (INV.O to FF2.D)
  - Set-up of second FF (FF2)



- Maximum frequency is then $F_{max} = \frac{1}{T_{min}}$

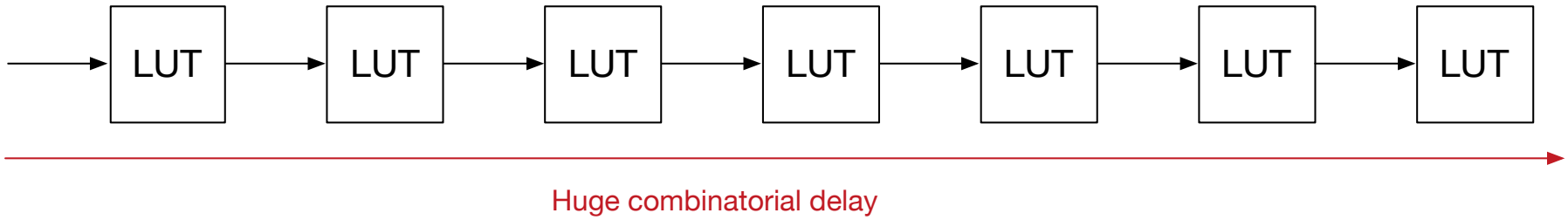
Logic complexity (FPGAs)

- Complex combinatorial circuits will be decomposed into multiple LUTs: this decomposition can be VERY sophisticated depending on optimization targets (see next section)
- Optimization will trade-off:
  - amount of concurrency – number of functionally independent LUTs that could run in parallel (LUTs 11, 12, 13 below)
  - logic depth – number of LUTs in series (functionally dependent) number of LUTs delays before the output (below 3 LUTs)
- Logic depth is crucial since it defines system critical path(s)



Logic depth and critical path

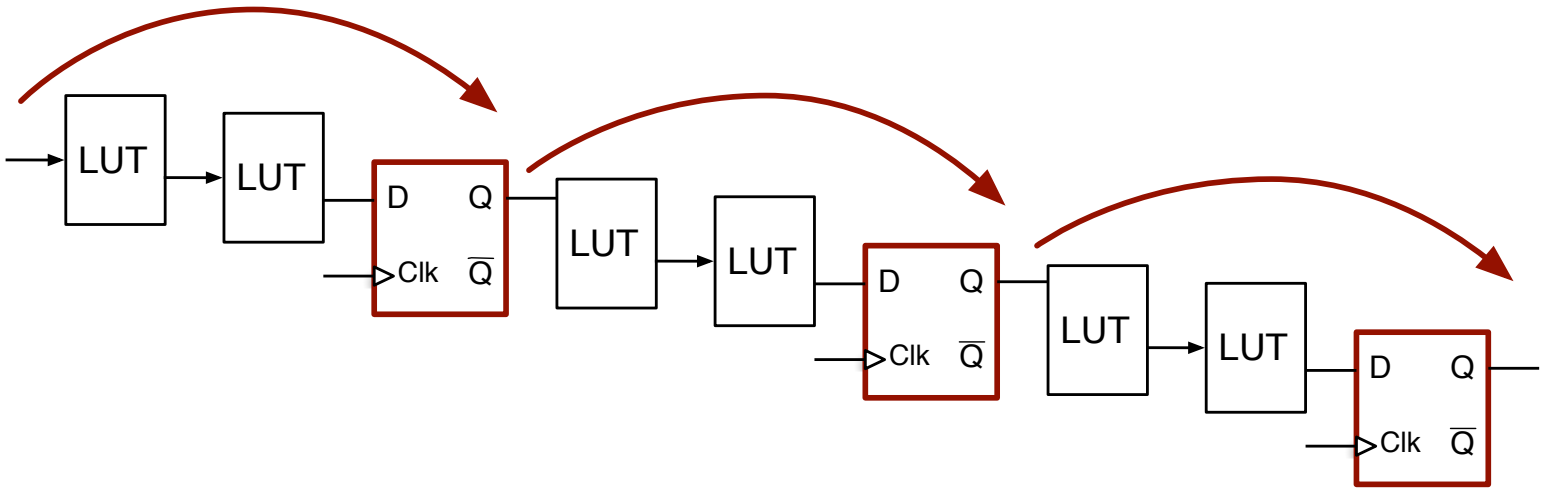
- Very complex logic functions might result in logically deep circuits:



- Logically deep circuits result in huge minimum period, so low operating frequency (bad performance)
- The circuit above is not going to be very fast since there are so many accumulated LUT delays
- Question is how to improve the performance of such circuit; the above situation is going to happen in real circuits
- Modify the computational algorithm, if possible and development time permits ... or insert a FF – very common solution

Improving clock speed and system latency

- Long combinatorial paths can be broken by inserting a certain number of intermediate FFs to evenly break the critical path



- Minimum period T_{min} will become shorter, so maximum frequency F_{max} goes up; design can run faster
- However the output will be available only after certain number of cycles – latency, number of cycles that delay the output from the input (3 in the above example)

Delay vs. Latency trade-off

- For complex circuits inserting latency on purpose is often mandatory, because the critical paths could be very deep
  - Think of instruction decoding stage that has to reach all sub-systems in a micro-processor, or complex arithmetic sub-circuits
- We speak of Flip-Flop insertion, or pipelining; this is the link between the physical design and the system architecture; e.g. in a processor the approach translates into sub-circuits to complete fetch, decode, execute, write operations in pipelined fashion
- When do we apply pipelining?
  - If we have to compute the same thing over many different data! The approach is not good for a single data unit computation
  - This is because the total computation time per unit does not change; you have better F_{max}, but more cycles to complete
  - By supplying one input data at each clock cycle, we will have one output data at each cycle; this results in higher data throughput and reduction in total execution time

Improving design performance at RTL/flow level

- Design flow tuning – manipulate different design optimization objectives: chose between performance, area or power as a prime optimization target; easiest thing to do since just a choice of an EDA SW parameter: you change, run sit & wait for result; you only pay run-time (could be significant for bigger designs)
- Constraints tuning – tightening (or sometimes relaxing) timing objectives for your design; they drive optimization process during place & route; could be easy or complex depending on the design and your understanding of it
- RTL rewriting for critical path modules – tracing down critical paths and enabling local RTL modifications; this can be easy ... or VERY complex
- Micro-architecture change – complete system change and re-design; **VERY** complex, done only if really needed

Improving design performance at device level

- Easiest way to do it: use more aggressive technology; better CMOS devices for ASICs/FPGAs at the expense of power & cost
- CMOS process is subject to local variations due to IC manufacturing process; circuits from the same wafer may have differences and will be binned according to their performance; faster devices will sell for more; this difference is more significant with advanced CMOS technologies
- So, FPGAs come with different speed grades ranging from lower to higher performance (same FPGA device family, and even ICs made on the same wafer)
- Faster FPGA device will have better timing of different components (LUTs, memory devices etc.); you can try this out, but use relatively complex design
- If justified you could pick a faster FPGAs to improve your design performance, but you will have to pay more (that difference may be less than the dev time!)

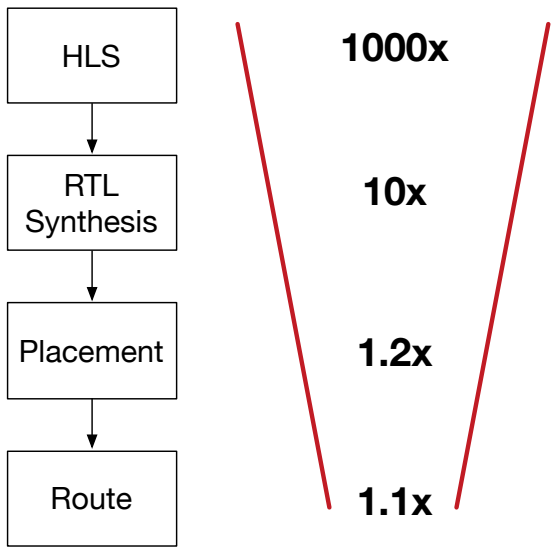
3. Design for performance

What is in your hands?

- Designers impact is most important (block & system integration)

  Micro-architecture definition and quality RTL coding
  are key for a good digital circuit design

- Wrong micro-architecture & bad RTL can never produce a good circuit no matter how good design automation is (10x impact at synthesis and 1.2 at place & route levels)
- Today we speak of High-Level Synthesis – from C/C++ to HDLs
- Trust me on this: if you do not know what you are doing, no HLS tool **will be able to generate a great IC**
- In real-world significant efforts will be spend in the back-end implementation, since even few % do make a difference



Impact on performance

Tools and their limits

- While current EDA tools are embedding state of the art knowledge of computer science, algorithms & programming, most of the problems that these SW tools deal with are very complex (placement, clock tree synthesis & routing)
- Exact solutions are out of the question & their optimality is always relative; good solutions make a good trade-off between performance, power & area & EDA tool run-time
- To make things worst current CMOS technology (5nm) enable very dense IC designs; these days we see commercial ICs with 60 billion transistors even for general purpose computers
- EDA SW need to deal with huge number of objects (gates, pins etc.) and thus require tremendous CPU power, loads of memory and significant run-times (couple of days are not uncommon)
- Real-world design flow tools come with detailed design methodologies – recipes on how to improve design process

Methodology example for FPGAs and Xilinx

Document below – tons of EDA tool & design related information with RTL coding advices for advanced designs

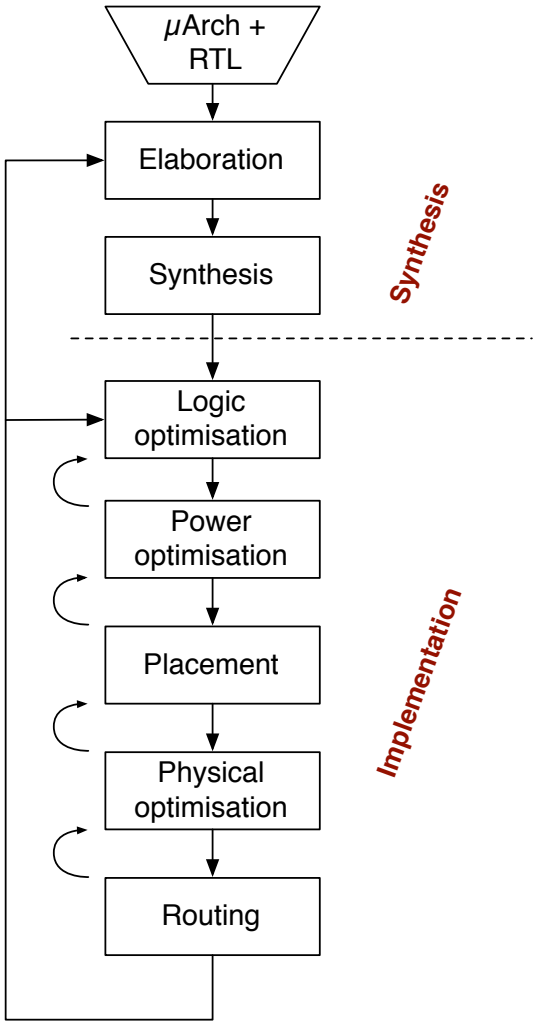
UltraFast Design Methodology Guide for the Vivado Design Suite

UG949 (v2015.3) November 23, 2015

4. Design flow and Performance, Power & Area (PPA) in FPGAs

Xilinx FPOGA design flow – optimization steps

- Actual flow adds extra steps on top of synthesis, place & route
- These aim optimizations to get best possible PPA; ICs are expensive, designs are complex and we & EDA tools should optimise PPA
- FPGA flow (Vivado) could be seen as push-button tool, and will work OK for small designs in one design iteration (if your design & constraints are OK)
- Real life (complex) designs may require MANY iterations at various stages
- Ultimate feedback is when you make a micro-architecture change; this is to be avoided since most time consuming



I. Elaboration and Synthesis

- Elaboration – produces the logical view from the input RTL
  - Generated models are good for functional analysis and simulation, but doesn't (yet) involve technology dependent mapping and optimization
  - Analysis of elaborated design informs you on how good your RTL spec is against micro-architecture you have in mind
- Synthesis – provides technology dependent and somehow optimal (logical) view of your design; physical implementation properties are only estimated since the design is not routed yet:

  a) Area – LUT/FF count and other resource usage
  b) Timing – actual vs. supplied timing constraints
  c) Power – statistically or simulation based
- If one of the above parameters doesn't meet the spec, there is no point of going further in the flow, since things will get worse with place & route!

II. Logic & power optimization

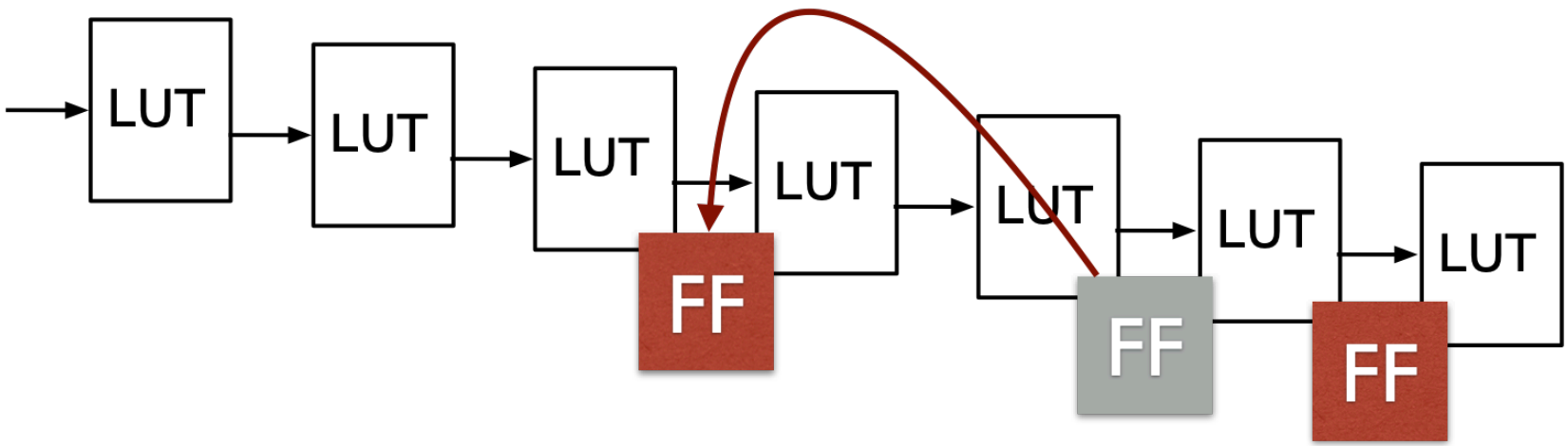
- Post synthesis gate-level netlist is manipulated by the tool to further improve PPA during placement & routing
- Logic optimization methods include:
  - Logic trimming – suppression of unused, or not properly used logic
  - Logic remapping – further effort into logic-to-LUT translation that could possibly lead to even better overall LUT usage
  - RAM optimisations – FPGAs do contain dedicated memory instances, when do you use LUTs for memory and when the real memory instances?
- Power optimization – pre- or post- placement **but not both**;
  - Pre-placement power optimization transforms gate-level netlist and thus can affect timing for better or worse
  - Post-placement keeps the timing intact, but generally results in less power savings

III. Placement

- Placement engine positions cells (LUTs) from the netlist onto specific sites in the target FPGA device (x, y locations)
- Placement constraints affect the placement engine
  - in a good way – they will provide hints where to look in the design space, and where not to spend efforts; improves PPA and run-time
  - in a bad way – bad constraints increase run-time with no visible effect, so this is waste of time & (electrical) energy (think of servers that run EDA SW)
- Quality constraints are crucial for a good design & are typically elaborated in iterative way: implement design (step-by-step), analyze PPA, modify constraints and re-run; constraints concern:
  - I/O or pin position
  - Location constraints for a given block (explicit physical constraints)
  - Timing Constraints – fundamental !!!
  - Netlist Constraints – ex. `LOCK_PINS`: used to specify mapping between logical LUT inputs and LUT physical input pins

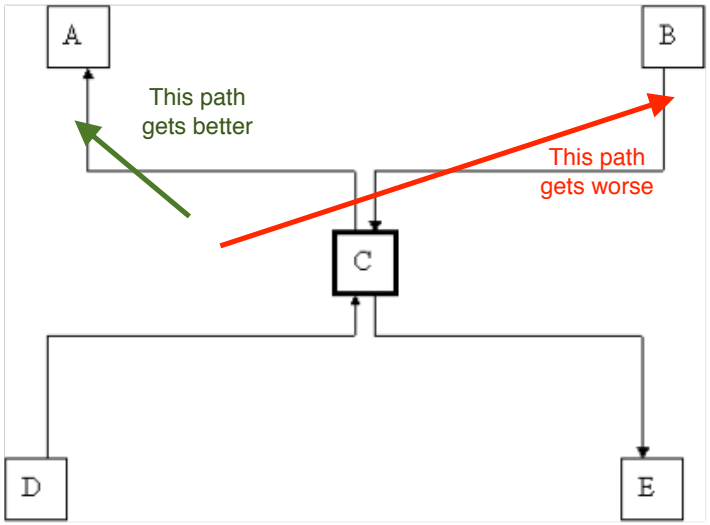
Physical optimization (1/2)

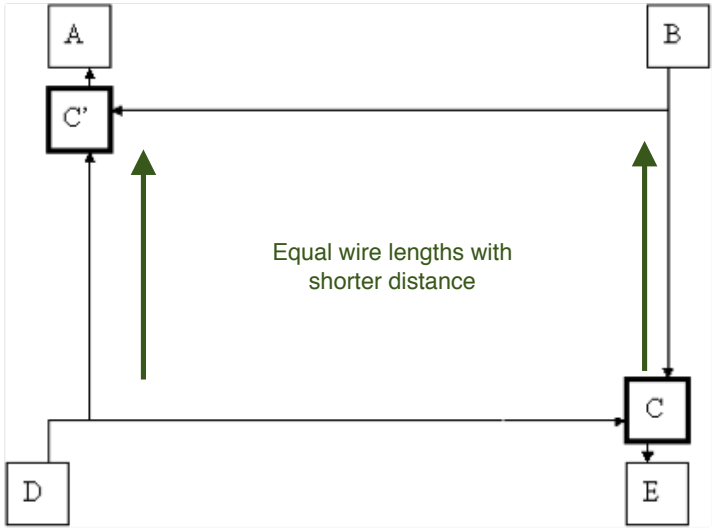
- Physical optimization is an **optional step** in the design flow
- It performs timing-driven optimization of all the paths that did not reach the timing
- Two methods: retiming and logic replication
  - Retiming – restructuring of FFs, i.e. changing the FF position to allow better timing on critical paths (better logic depth balance)
  - Latency remains unchanged since the same number of FFs
  - It is just the position of the FF in the critical path that changes (you could potentially do this yourself in RTL, but with effort)



Physical optimization (2/2)

Logic replication – this is inverse to logic optimization, if applied it will increase area to gain performance





If block C is in the critical path (other blocks are fixed) moving C to any other location will make things worse; if timing not met at this stage, we can't do anything

By replicating the block C we offer more flexibility for placement, contribution of wiring to critical path (logic will not be changed) can be reduced!

IV. Routing

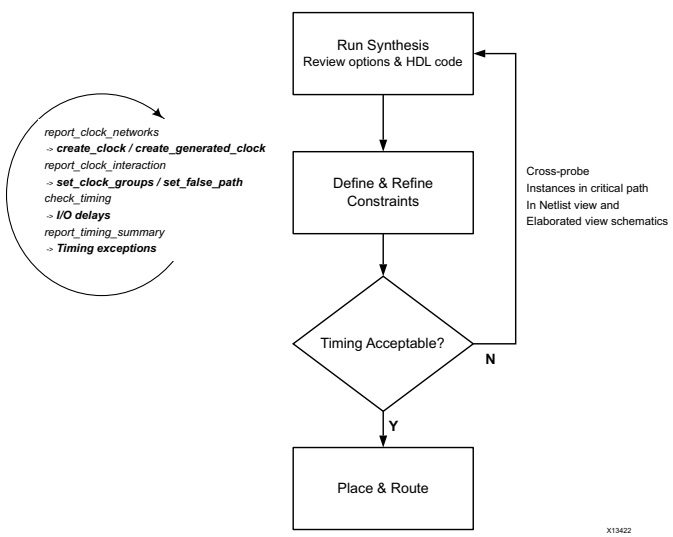
- All pins (terminals) in the design are connected using physical nets
  - Pins = all IO & intermediate logic, LUTs, memory, DSPs etc.
- Routing quality depends on target technology, design complexity (number of nets and pin) and constraints (timing or other)
- Routing is timing driven, optimization will aim to resolve all timing violations in the system
- Poor routing results are often due to poor/incorrect timing constraints!
- Routing can be fast, but unrealistic timing constraints could lead to uncontrolled run times, thus we usually limit the number of trials

```
Design Route Status
                                                :      # nets :
  ------------------------------------------- : ----------- :
  # of logical nets..................... :       12400 :
      # of nets not needing routing..... :        4014 :
          # of internally routed nets.... :        4014 :
      # of routable nets................. :        8386 :
          # of fully routed nets......... :        8386 :
      # of nets with routing errors..... :           0 :
  ------------------------------------------- : ----------- :
```

Routing report of a small design

V. Timing closure

- Process of completing the full place & route flow for a given HDL inputs, and EDA tools parameters that will produce a solution that satisfies all constraints in the design
- Constraints are typically refined as we move along with the flow and only when timing constraints are met at previous level, and with a certain margin, we can proceed with the next steps
- Good timing after synthesis will not guarantee good timing after routing
- In FPGA circuits clock tree synthesis (clock distribution to all FFs) & power delivery is fully automated and uses predefined resources; these two tasks could be quite complex in ASICs
- In FPGAs routing produces final design, extracted metrics are those of a circuit



Constraints example

- Below we have **two clock periods** for two different clock nets
- Physical constraints for **FPGA pin positions**
  - These constraints are necessary to provide IC design that could be compatible with the existing PCB layout
  - If not specified, place & route tool does his own IO placement optimization (top-down, PCB to IC, or bottom-up, IC to PCB)
  - Placement of the LUTs will be driven by the IO placement

```
# LOC and IOSTANDARDS set for all IOs which is required for generating a Bitstream
create_clock -period 10 -name wbClk [get_ports wbClk]
create_clock -period 5 -name bftClk [get_ports bftClk]

# Since this sample design is not meant to be stand alone, but exist in a larger
# design, there are no input/output delays specified. Timing will not be done
# on these interfaces by default.
set_property PACKAGE_PIN P20 [get_ports {wbOutputData[23]}]
set_property PACKAGE_PIN V22 [get_ports {wbOutputData[9]}]
set_property PACKAGE_PIN E21 [get_ports {wbInputData[18]}]
set_property PACKAGE_PIN M17 [get_ports {wbInputData[4]}]
set_property PACKAGE_PIN R17 [get_ports {wbOutputData[22]}]
set_property PACKAGE_PIN T18 [get_ports {wbOutputData[8]}]
```

- Note that physical constraints are ignored during synthesis

a) Area reporting 1/3

- Total Logic Area (A_{TLA}) – sum of the area of all logic resources in the design calculated after synthesis, placement & route
  - Logic resources = LUTs (and others) in FPGAs or gates in ASICs
- Ideally the total cell area should be equal to total core (chip, IC) area A_{IC}, that is the actual silicon die area
- In practice this never happens, we always have some unused area due to place & route complexity
  - In ASICs this is some empty space in the IC, and in FPGAs this will be all unused LUTs (all those that are left non-programmed)
- Obviously unused area is to be minimized!
- To express how well silicon is used EDA tools use the notion of Design Utilization – DU

$$DU = \frac{A_{TLA}}{A_{IC}}$$

a) Area reporting 2/3

- Typically DU depends on the design complexity (mostly system connectivity), micro-architecture, designers skills, constraint realism, EDA tool chain ... and is probably the most important parameter designers are looking for since related to cost
- In ASICs DU can be from 75% and up to, but rarely, above 90%; in most recent CMOS nodes DU could be even less
- Sometimes designer accept exceptionally low DU (as low as 30%), but only for some parts of the design; certain functional blocks have complex connectivity and hence could result in complex place & route that needs space
- In FPGAs: you can pick the device density that suits the best the needs of the design, but don't expect utilization to be 100%
- Trying to get the high density will most likely degrade the system performance, since place&route have less freedom

a) Area reporting 3/3

Example – post synthesis report provides: LUT usage (LUT implementing logic & memory!), flip-flops (latches) at global and hierarchical (per module) levels

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	4035	0	17600	22.92
LUT as Logic	4035	0	17600	22.92
LUT as Memory	0	0	6000	0.00
Slice Registers	3098	0	35200	8.80
Register as Flip Flop	3098	0	35200	8.80
Register as Latch	0			
F7 Muxes	0			
F8 Muxes	0			

Hierarchy

Name	Slice LUTs* (17600)	Slice Registers (35200)	Block RAM Tile (60)	DSPs (80)	Bonded IOB (100)	BUFGCTRL (32)
bft	4036	3098	16	64	71	2
arnd1 (round_1)	768	384	0	16	0	0
arnd2 (round_2)	768	384	0	16	0	0
arnd3 (round_3)	768	384	0	16	0	0
arnd4 (round_4)	768	384	0	16	0	0
egressLoop[0].egressFi...	53	94	1	0	0	0
egressLoop[1].egressFi...	53	94	1	0	0	0
egressLoop[2].egressFi...	53	94	1	0	0	0
egressLoop[3].egressFi...	85	94	1	0	0	0
egressLoop[4].egressFi...	54	94	1	0	0	0
egressLoop[5].egressFi...	85	94	1	0	0	0
egressLoop[6].egressFi...	54	94	1	0	0	0
egressLoop[7].egressFi...	88	94	1	0	0	0
ingressLoop[0].ingress...	54	94	1	0	0	0
ingressLoop[1].ingress...	54	94	1	0	0	0
ingressLoop[2].ingress...	54	94	1	0	0	0
ingressLoop[3].ingress...	54	94	1	0	0	0
ingressLoop[4].ingress...	54	94	1	0	0	0

b) Static Timing Analysis reports

- Gives us the following information:
  - Worst Negative Slack (WNS) – it is negative, the timing is not met for set-up! By adding 1.227ns to the target period you may end up with a right frequency target (you may re-run the full implementation flow with a new constraint)
  - Total Negative Slack (TNS) – 778.829ns; quite big compared to WNS
  - Finally the number of failing endpoints (number of failing paths); in the example below: 960 paths out of 7973 failed, roughly 10% this is a lot
  - Improving timing with HDL re-design could be hard; constraints are too aggressive and should be relaxed

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1.227 ns	Worst Hold Slack (WHS):	0.077 ns	Worst Pulse Width Slack (WPWS):	1.971 ns
Total Negative Slack (TNS):	-778.829 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	960	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7973	Total Number of Endpoints:	7973	Total Number of Endpoints:	3164

Timing report at different implementation stages

- Below are shown two timing reports for the same design, first report is post synthesis, and the other is post place & route
- Design clearly did not met timing specifications

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1.227 ns	Worst Hold Slack (WHS):	0.077 ns	Worst Pulse Width Slack (WPWS):	1.971 ns
Total Negative Slack (TNS):	-778.829 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	960	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7973	Total Number of Endpoints:	7973	Total Number of Endpoints:	3164

Design Timing Summary

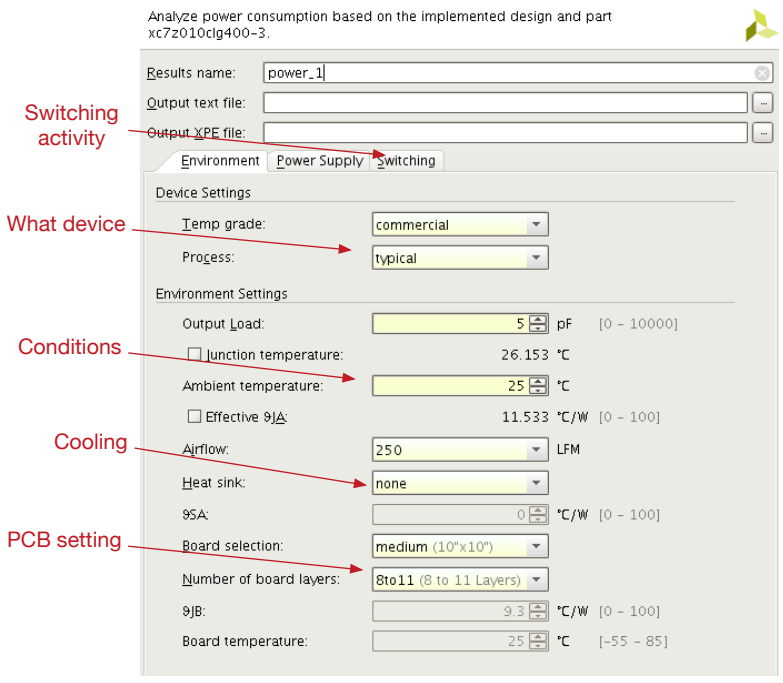
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1.400 ns	Worst Hold Slack (WHS):	0.058 ns	Worst Pulse Width Slack (WPWS):	2.000 ns
Total Negative Slack (TNS):	-657.205 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	940	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7941	Total Number of Endpoints:	7941	Total Number of Endpoints:	3132

Timing constraints are not met.

- There are slight differences; Can you explain why?

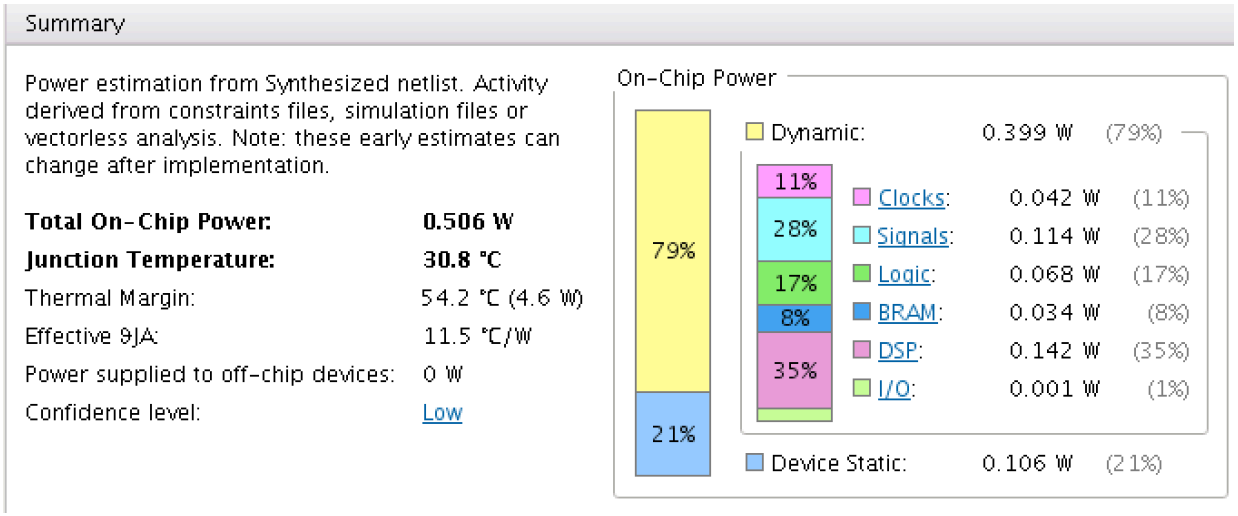
c) Power reporting – setup

- Power simulation are run to derive electrical and thermal needs of the FPGA (power supply design and cooling)
- Assumptions are made on FPGA package, cooling and PCB; also we need logic activity; for quick power estimates toggle statistics are used, i.e. how much often gates/wires switch; common parameter for all gates – could under or over estimate
- Accurate power simulation – can be calculated only after routing & functional simulation that will provide exact switching activity of all gates & wires, i.e. which gate and wire switches and when; this is time consuming so done only at very end of the flow



c) Power reporting – analysis

- Total power & temperature reported; note the dynamic (due to switching) and static (due to leakage) power components
- Detailed breakdown per IC component (Clock, wires, logic etc.)
  - ▷ What can you say about the breakdown?

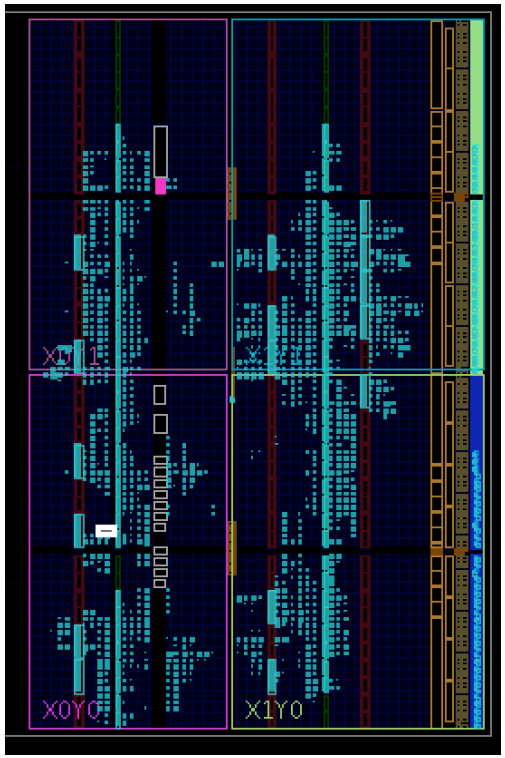


- Note that the above report has been generated after synthesis, with no actual toggle statistics, hence the confidence is said to be low; this can be used as a rough guide

5. Physical implementation of FPGA circuits

Implemented design

- Layout view of a placed circuit in an FPGA
- Current Vivado tool uses analytical placement & route engine to get more predictable results
- Nevertheless designers can use randomness when running the flow; this randomness could cause PPA differences (better or worse) of the implementation at the expense of longer run-times



- Using randomness could yield totally different layouts, although keeping exactly the same functionality
- Designers do not care how the design is implemented as long as:
  - it is a functionally correct,
  - that satisfies given timing & power constraints,
  - with minimum area since this will save cost

FPGA configuration

- Once the timing is closed (i.e. constraints are validated), we can generate the corresponding configuration file, also known as a bitstream
  - What exactly an FPGA configuration file stores?
- Bitstream file contains design configuration information required to "program" a given FPGA device (you know why I use brackets)
  - When you start an FPGA project the first thing that you should do is to pick a certain FGPA device
  - FPGA manufactures group FPGAs in families, all devices in the same family share same architecture & CMOS node, but have different capacity and speed grade
  - FPGA device choice will influence design PPA & cost
- After the configuration file is loaded into FPGA device, design will run at specified (and supplied) frequency F; FPGA devices do have their own internal clock generators, but you have limited choice

Reconfigurable systems

- More & more computational systems are becoming heterogeneous: a combination of traditional general purpose CPUs and HW accelerators implemented using FPGAs
- Intel has bought Altera – 2nd FPGA manufacturer after Xilinx
  - ▹ Why do you think they did so?
- Some computing architectures use FPGAs to change their internal architecture depending on the computational problem they need to solve – reconfigurable computer architecture; new paradigm in computer architecture
- You have a library of configurations and you pick the one to be loaded in the FPGA circuit depending on the needs
- Of course you need to keep the interface of your reconfigurable architecture with everything else the same, but this is doable

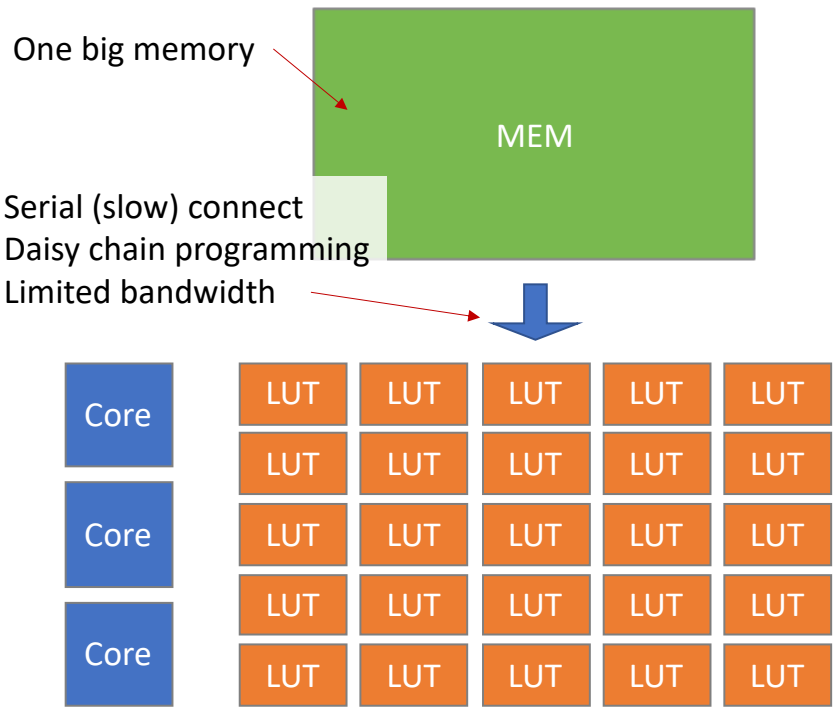
Dynamically reconfigurable systems

- Some FPGA devices allow dynamic reprogrammability – you define a part of the FPGA that could be modified on the fly, while the other part still remains operational – dynamically reconfigurable computing; a totally different view of computing systems compared to traditional programmable computing
- Connection between the configuration file source, typically some kind of re-programmable memory, & the FPGA itself is **serial** to limit configuration pin count (IC pins are expensive) and **not that fast** to allow use of longer cables
- Thus, the FPGA configuration takes some non-negligible time and this will limit how often the functionality is modified (these days you wouldn't do that that often)
- But tomorrow things may change ...
- 3D IC technologies could be used to integrate very dense connections between FPGA and memory layers built on top

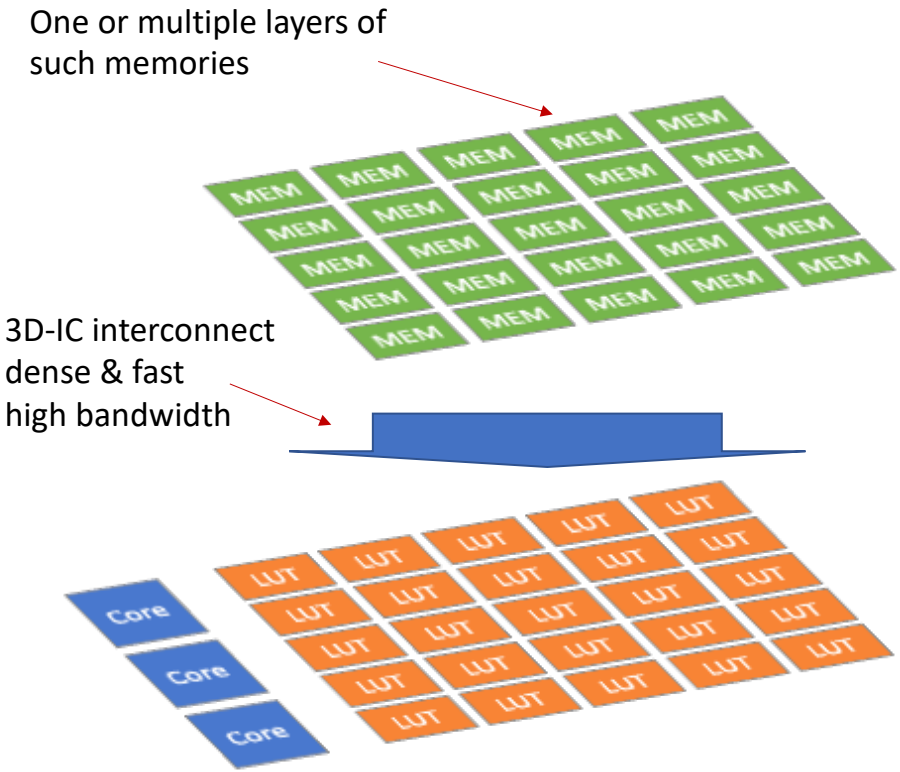
3D FPGA: is this the future?

By distributing the configuration memory and implementing it on-chip, reconfiguration time could be pushed to few memory cycles

Separate package with one configuration memory



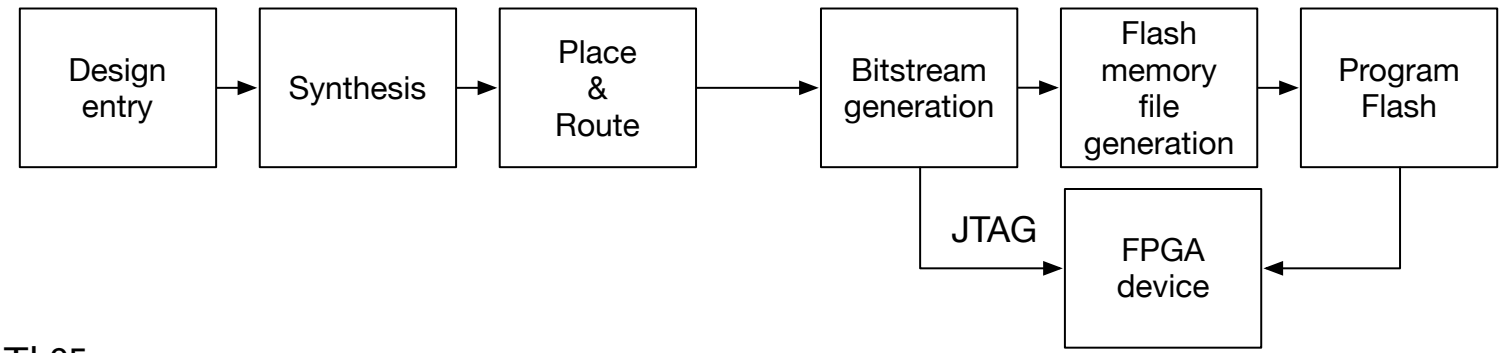
In-package many configuration memories with



7

Practical FPGA configurability

- Programming of the FPGA device can be done:
  - Directly – using your computer, FPGA IDE and some cable (JTAG is a commonly used standard) to directly program the device from the host computer
  - Indirectly – you store the bitstream into some kind of memory (e.g. flash); configuration stream is uploaded to the FPGA automatically during FPGA boot sequence on power-up
- When connected to a host computer and JTAG cable, designs can be tested live at run-time using probes to check values at certain points in the design
- Most of the FPGAs IDE provide some kind of support to recover probe values and display them for live debugging



That's all folks!