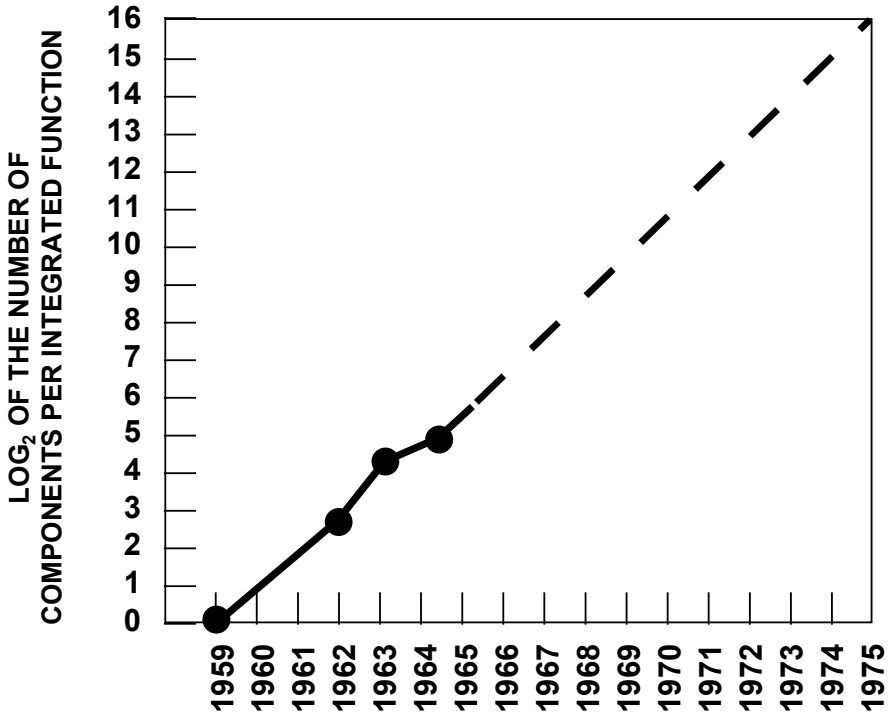
Αρχιτεκτονική Προηγμένων Υπολογιστών και Επιταχυντών

Άγγελος Αθανασιάδης

ακ. έτος: 2025-2026



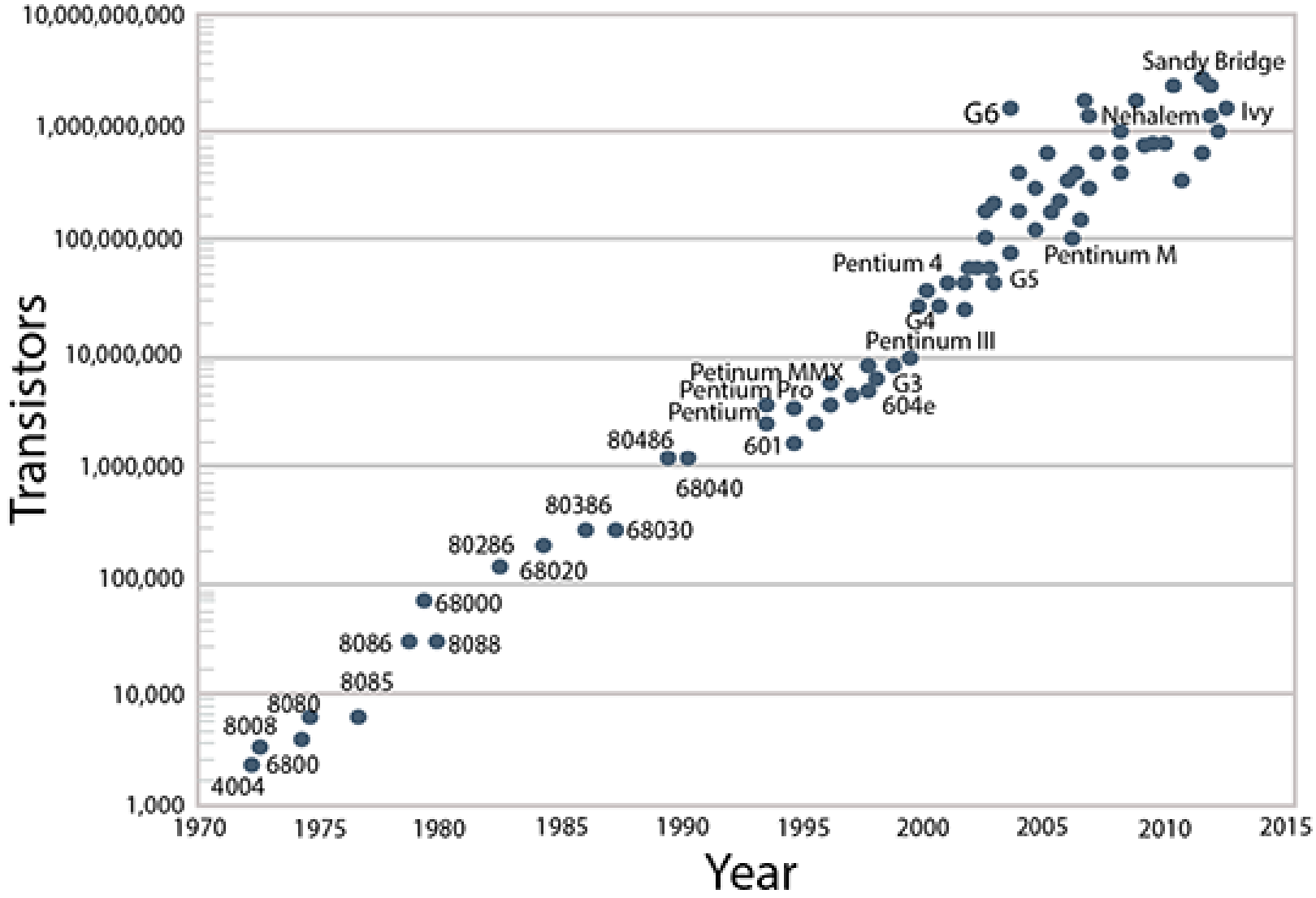
Moore's original law

Electronics, April 19, 1965.

Moore's law

- Based on economics, not physics (even though physics determine it in part)
- Needed to be able to **forecast** what will be the **performance of semiconductors** when my electronic **product will hit the market**
  - How many transistors will be available on a chip in 2, 5, 10, 20 years?
  - What will be their cost?
  - At which frequency will the chip work?
  - What will be its power and energy consumption?
- The unprecedented complexity of this forecasting effort required a unique industry-wide effort: the now-defunct International Technology Map for Semiconductors

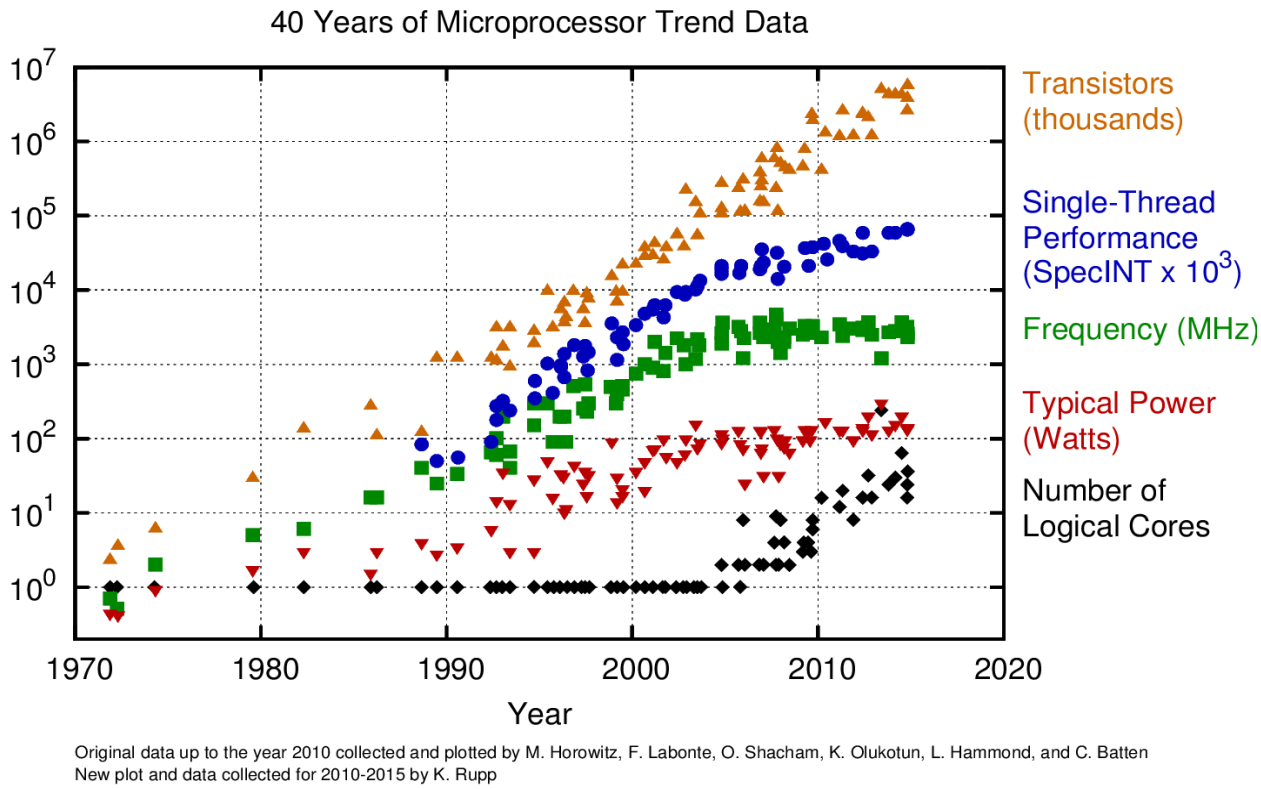
The basis of Moore's law



The number of transistors on a microprocessor doubles every 2 years

Courtesy, University of Wisconsin

The various aspects of Moore's law

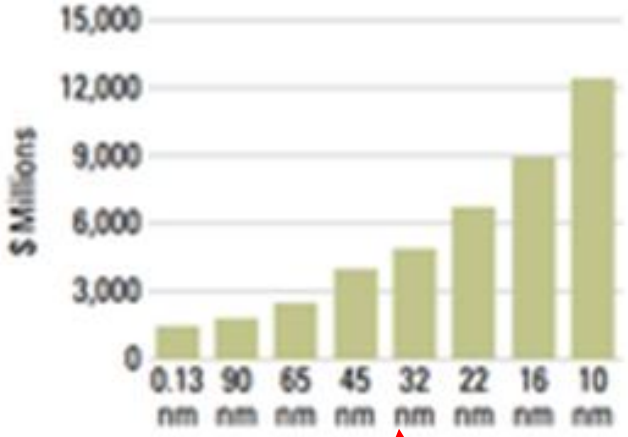


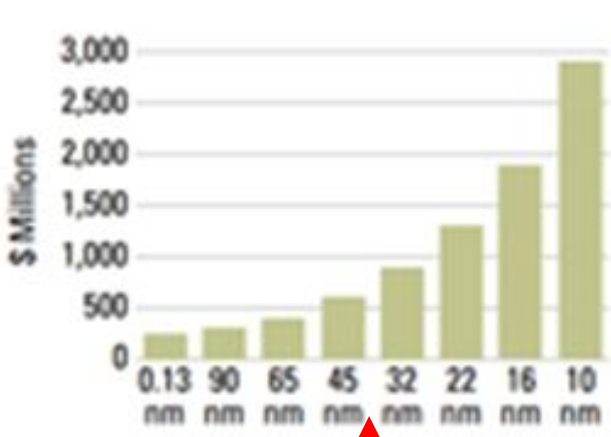
«This is the end, my friend»

- Transistor costs no longer go down
- It becomes more and more expensive to justify the adoption of new technologies

Technology	Gates/mm² (KU)	Gate utilization (%)	Used gates/mm² (KU)	Parametric yield impact (Δ from D_0 yield)	Actual used gates/mm² (KU)	Gates/ wafer (MU)	Wafer cost ($)	Wafer cost (Δ)	Cost per 100M gate ($)
90nm	637	86	546	97	532	33,831	1,357.62	–	4.01
65nm	1,109	83	919	96	885	56,330	1,585.71	16.8	2.82
45/40nm	2,139	78	1,677	92	1,538	97,842	1,898.83	19.7	1.94
28nm	4,262	77	3,282	87	2,855	181,658	2,361.84	24.4	1.30
20nm	6,992	65	4,524	73	3,293	209,541	2,981.75	26.2	1.42
16/14nm	10,488	64	6,712	67	4,497	286,140	4,081.22	36.9	1.43
10nm	14,957	60	8,974	62	5,564	354,013	5,126.35	25.6	1.45
7nm	17,085	59	10,080	60	6,048	384,813	5,859.28	14.3	1.52

The reason





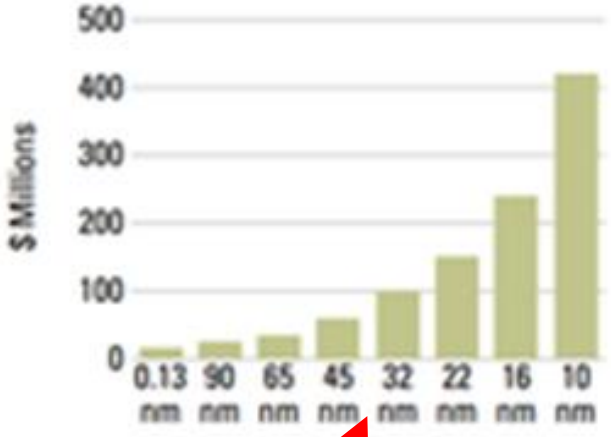


FIGURE 4. (L) Fab Costs by Node (in US$ billions) **FIGURE 5.** (C) Process Technology Development Costs by Node (US$ billions) **FIGURE 6.** (R) Chip Design Costs by Node (US$ millions)

- Fabrication line, technology development, mask costs grow faster than transistors shrink
- Design costs grow as much as mask costs, since getting working chips back at the first round is essential

Design and production costs

- NRE cost (Non-Recurring Engineering cost)
  - Paid once per design
  - Includes design and mask production costs
  - Independent of the number of manufactured chips N
- Unit production cost (U)
  - Paid once per manufactured chip
  - Includes raw material, plant depreciation (really an NRE but carefully hidden from most of the industry) and labor
  - Proportional to N

- Total per-product cost (C): depends on U, NRE, N
  - C = (NRE / N) + U

- The best amount of design effort depends on N…

The cost of flexibility

- SW-like programmability and flexibility has a cost
- About 1 order of magnitude loss in terms of unit cost, performance and energy between levels:
  - General-purpose CPU
    - Most flexible, lowest design cost
  - Application-Specific Instruction set Processor (ASIP)
    - DSP, GPU, vector processor
    - Customized datapath, memory subsystem
  - Field-Programmable Gate Array (FPGA)
    - Custom-like HW without mask costs
    - No more fetch-decode-execute cycle (wasting **energy**)
  - Application-Specific Integrated Circuit (ASIC)
    - Full customization of datapath, control and memory

Programmability versus specificity

- Programmability:
  - Loses efficiency (cost, performance energy, …)
  - Gains flexibility (ease of adaptation, robustness, …)

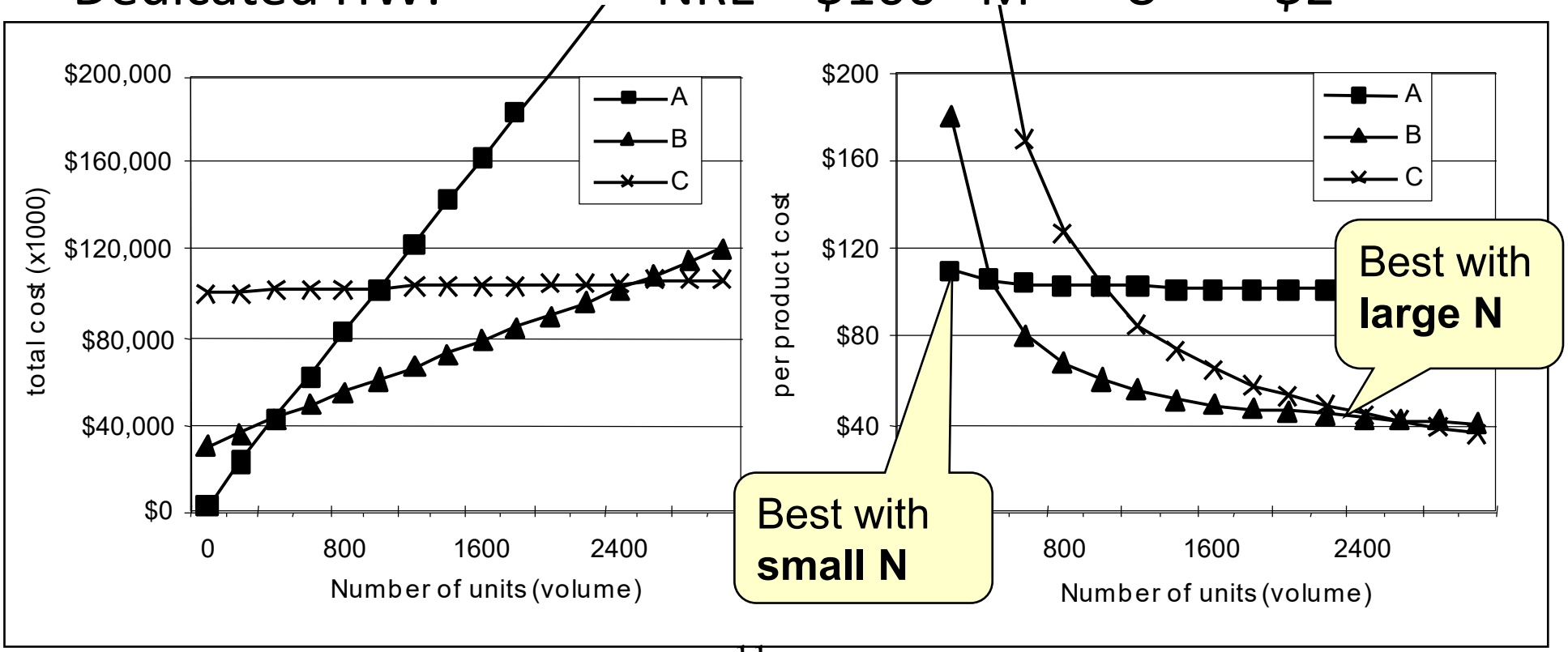
PC, tablet | Smart phone | Video-game console | Telecom switch | Engine control, camera | Washing machine, Elevator

Programmability, flexibility

Efficiency

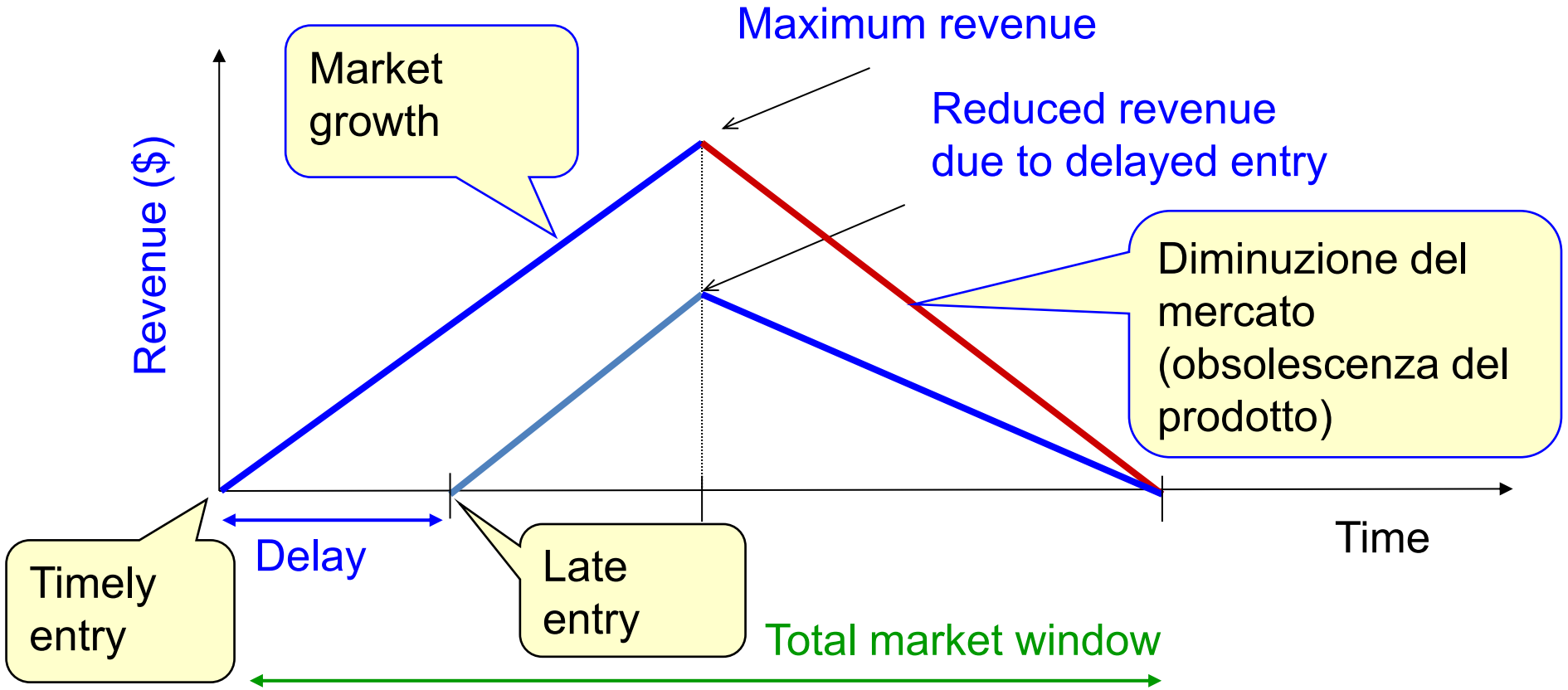
Various implementation options

- Mostly SW: NRE = $0.2M U = $100
- Programmable HW: NRE = $10 M U = $30
- Dedicated HW: NRE = $100 M U = $2



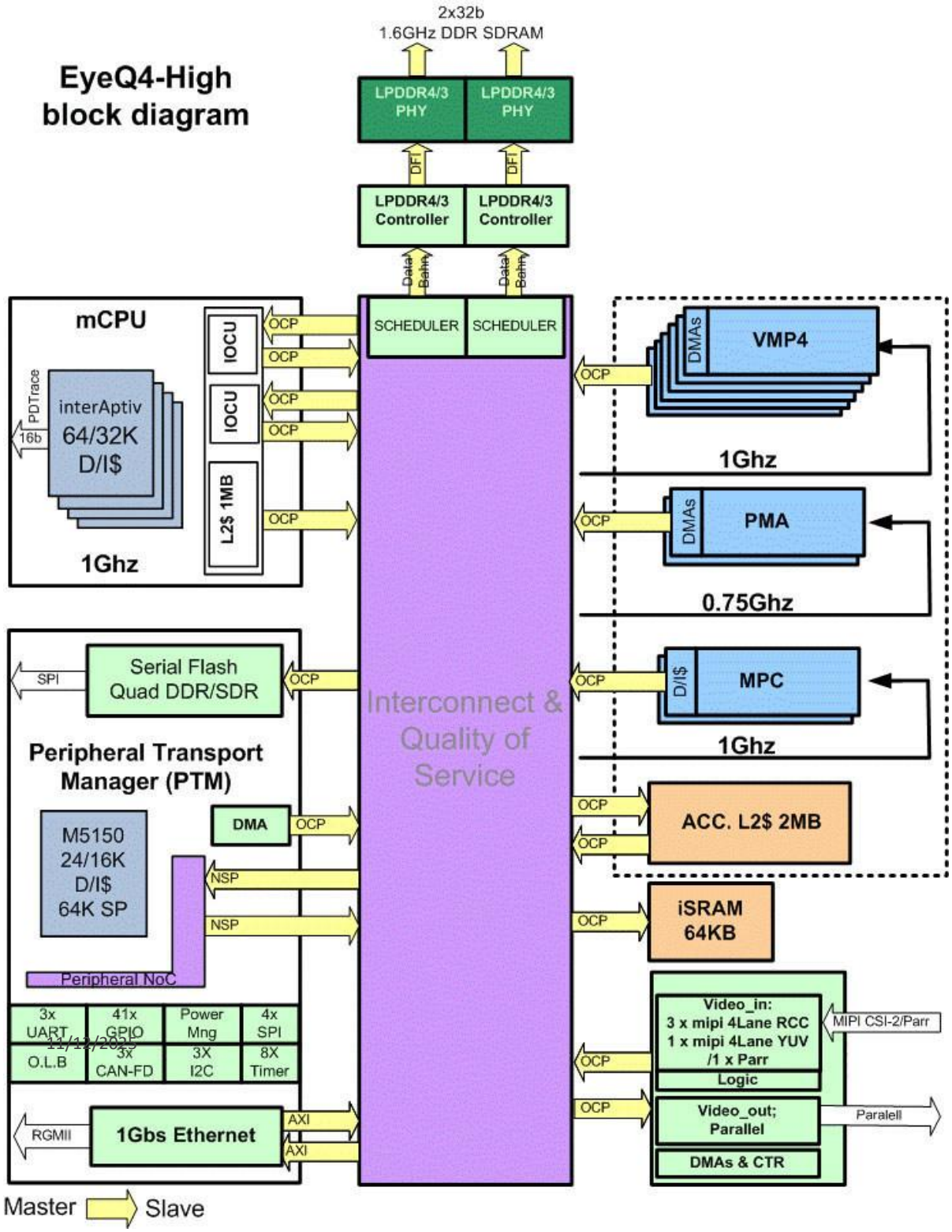
Time to market

- Excessive design time delays market entry, with a **quadratic** effect on total revenue



Outline

- Motivation:
  - Definition of embedded systems
  - Moore's law
  - The economics of electronic system design
- **Reduction of design costs and performance/cost optimization**
  - Platforms
  - Energy/performance trade-offs
- Modeling languages and expressive power
  - Data-dominated vs control-dominated
  - Turing machines and undecidability

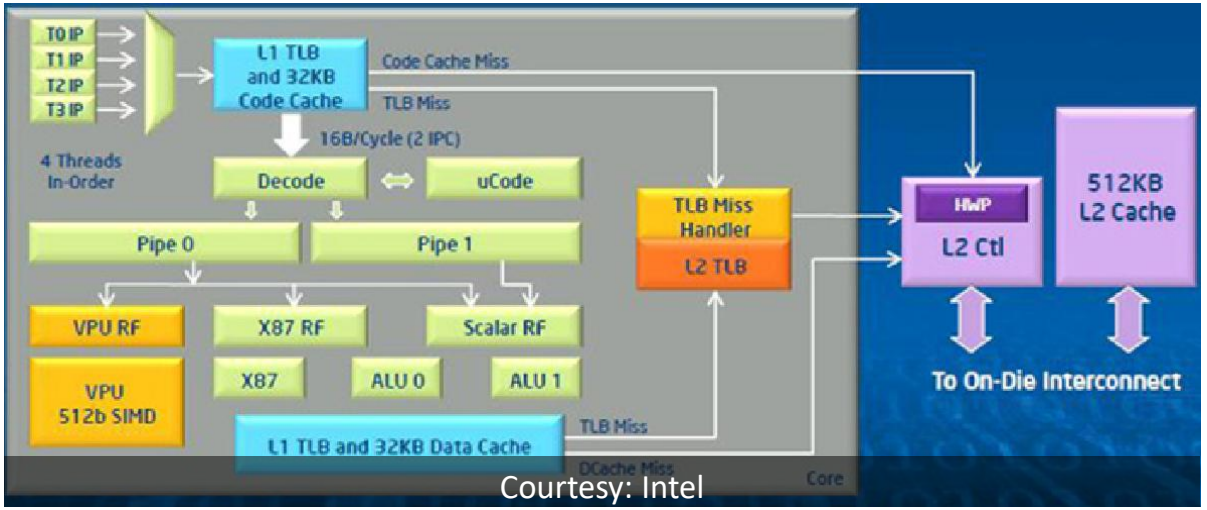


- Example: MobilEye EyeQ
  - Multi-core processor (MIPS-based)
  - SIMD Vector Multi-Processor for image processing
- Similar to, e.g., Kalray MPPA

Courtesy: MobilEye

Example of CPU platform

- Intel processor-based multi-core platforms (PCs, data centers)
  - Multi-core multi-threaded CPU
  - High-performance memory interface and interconnect
  - Vector processor (Xeon Phi), sharing memory interface with processor
  - Reduces fetch-decode-execute bottleneck by using SIMD architecture

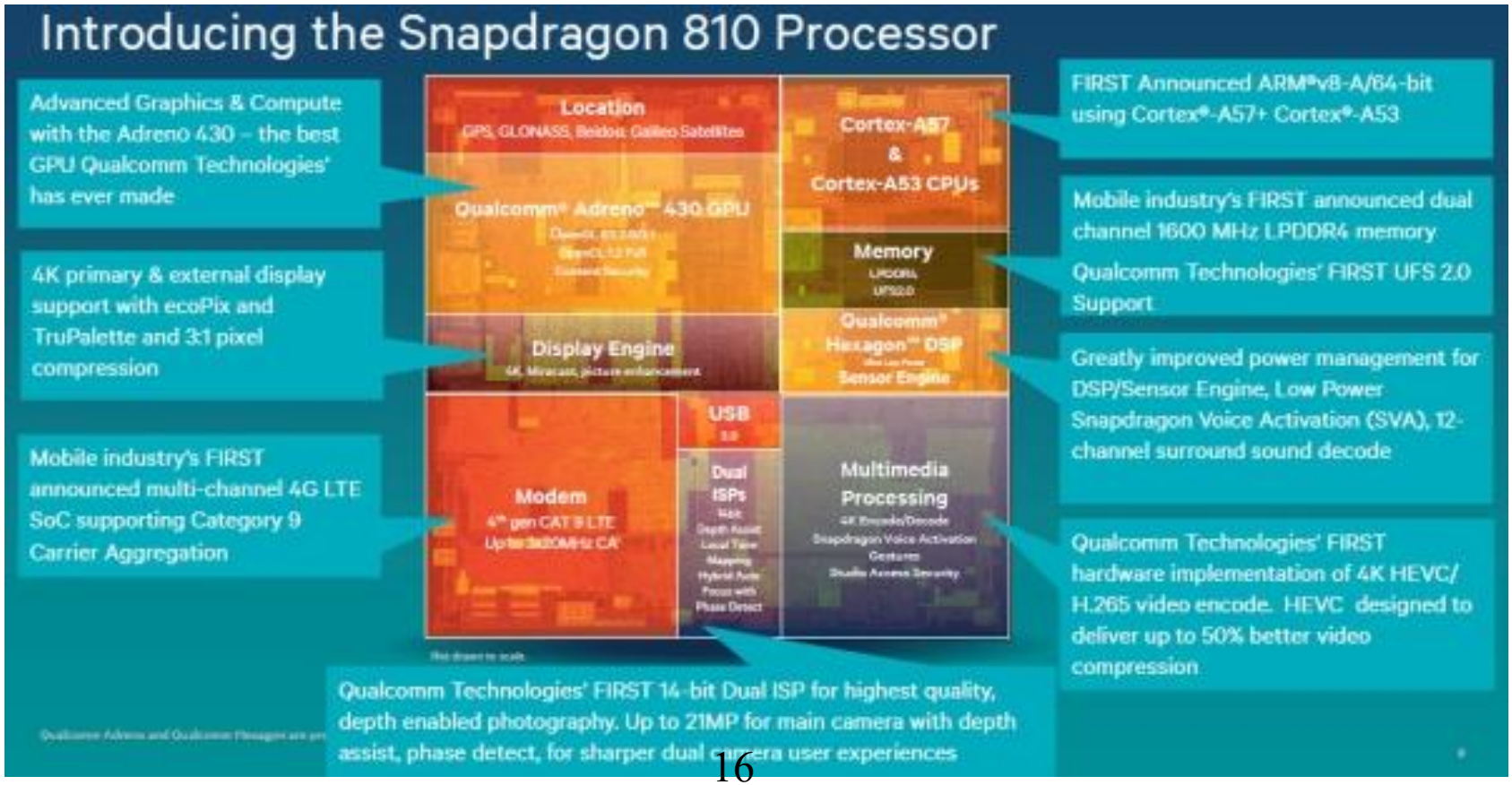


Courtesy: Intel

Example of SOC platform

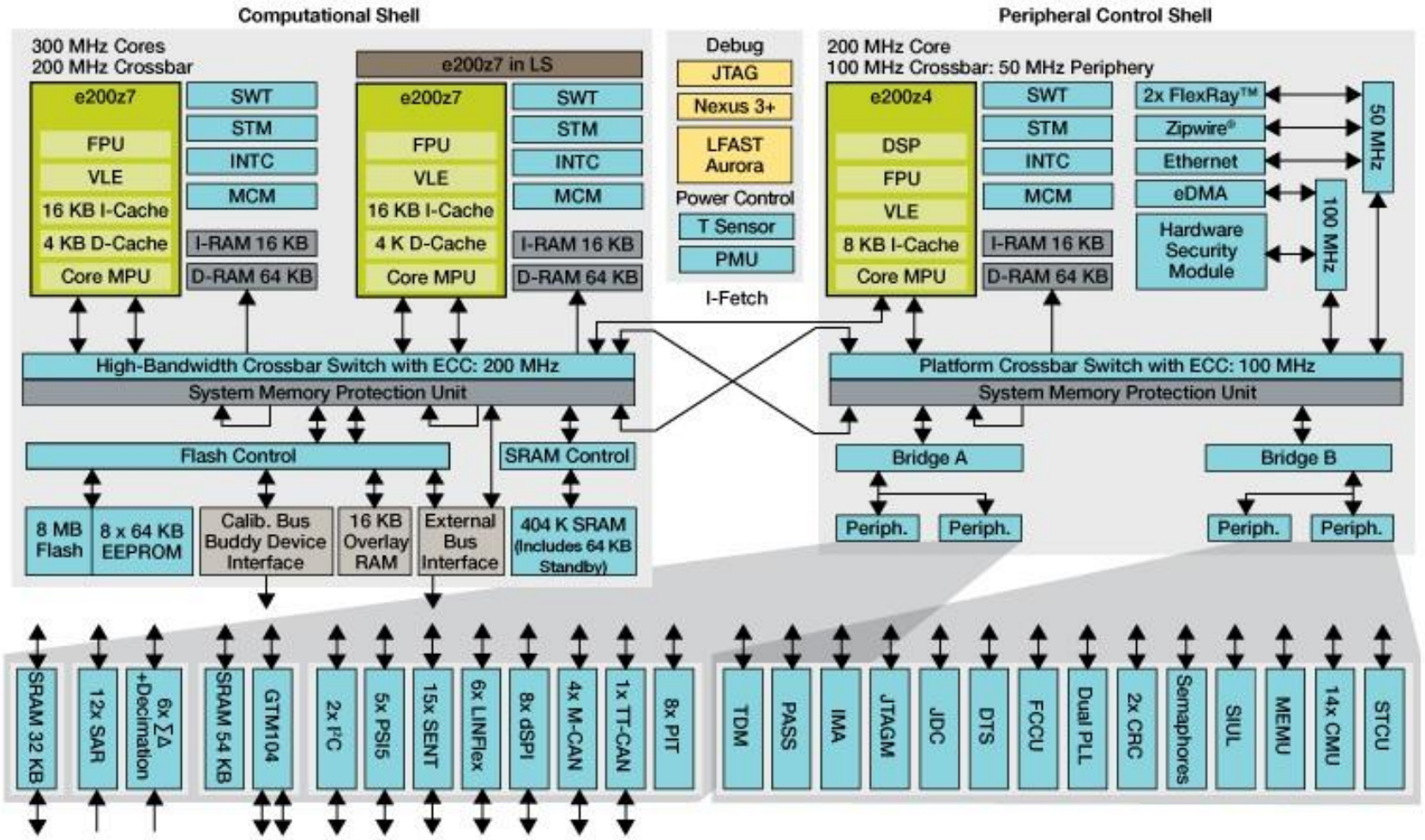
- Example: Snapdragon 810 cell phone SOC
  - Multi-core processor
  - Application-specific accelerators and peripherals

Courtesy: Qualcomm



Example of SOC platform

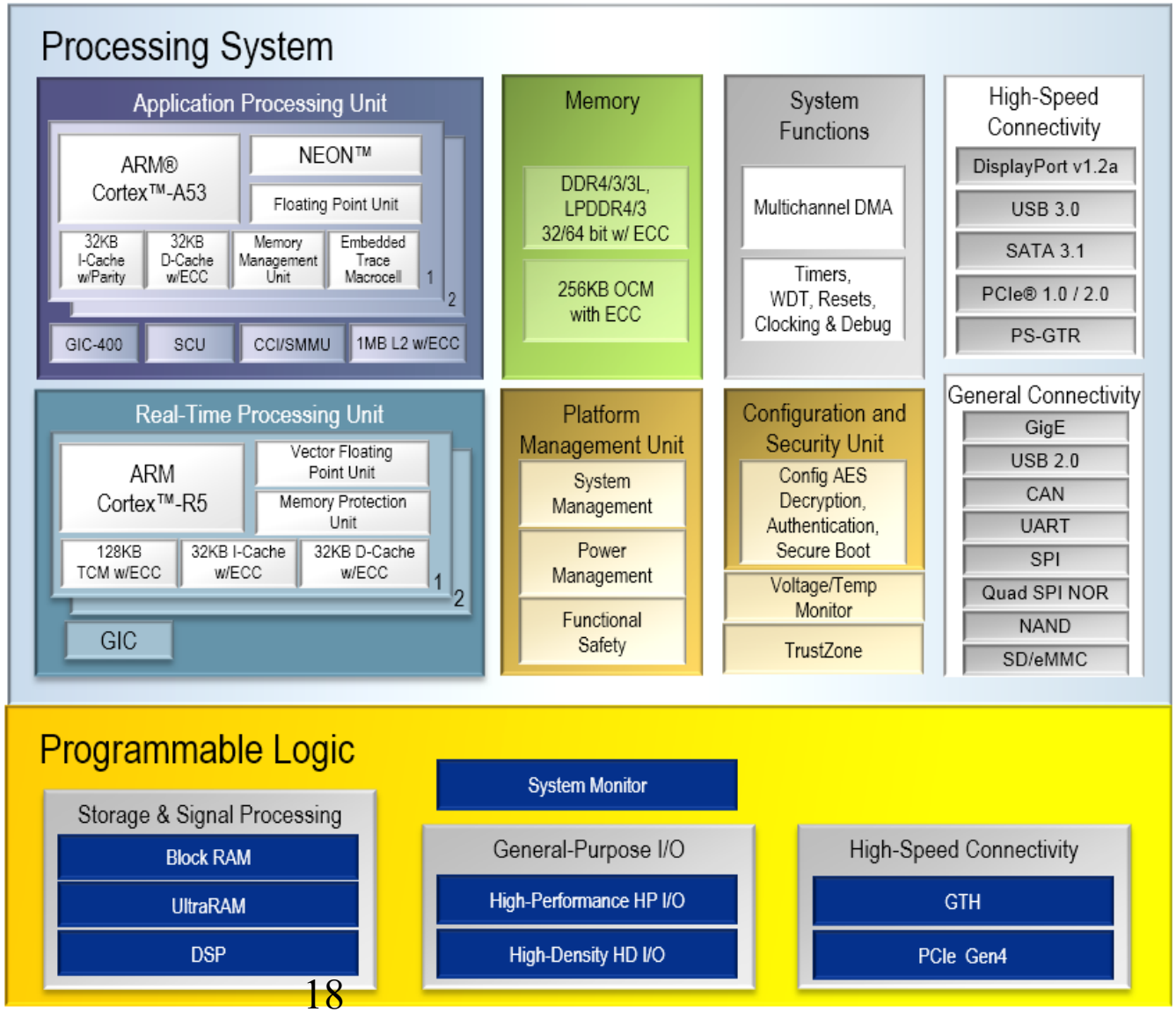
- Example: NXP MPC5777M automotive SOC
  - Multi-core processor (twin for safety)
  - Application-specific peripherals



Courtesy: NXP

Example of FPGA platform

- Xilinx Zynq
  - Multi-core processor
  - Programmable logic
  - Some ASIC-style peripherals
- Similar to, e.g. Altera Arria



Courtesy: Xilinx

Platform analysis

- Several common points:
  - All include one or more single-core or multi-core processors
  - Some processors are highly customized
    - SIMD in GPU
    - Vector processor in EyeQ
  - All include some HW
    - Application-specific HW for efficiency (e.g. modem, video accelerators, peripherals)
    - Reprogrammable HW for flexibility
- Key differentiators are application-specific accelerators and peripherals
- In all cases: HW/SW architecture design (including HW/SW partitioning problem)

Performance and energy efficiency

Type	Device	GFLOPS	Cost (€)	Power(W)	GFLOPS/€	GFLOPS/W
Multi-core	Intel E5-2630v3 8x2.4GHz	600	700	85	0.85	7.05
	Intel E5-2630v3 10x2.3GHz	740	1250	105	0.59	7.04
Many-core	Xeon Phi, knights corner, 16GB	2416	3500	270	0.69	8.94
	Xeon Phi, knights landing, 16GB	**7000**	3500	300	**2.00**	23.3
GPU	Nvidia GeForce Titan X	**7000**	1000	250	**7.00**	28
	Nvidia Tesla V100 (matr. mul)	**120000**	?	300	?	**400**
	Nvidia Tegra X1	512	450	**7**	**19.40**	**73**
	ARM Mali T880 MP16	374	?	**5**	?	**74?**
	Radeon firepro S9150	**5070**	3500	235	1.44	21.5
FPGA	Altera Arria 10	1500	2000	**30**	0.75	**50**
	Altera Stratix 10	**10000**	3000	125?	**3.33**	**80**
	Xilinx Zynq Ultrascale+	**5000**	2000	**40**	**2.30**	**115**

Courtesy: I. Mavroidis, ICS/FORTH

HW/SW co-design basics

- Models and methods to **reduce embedded system-level design time**, where **most of the cost/performance trade-offs occur**
- We will assume an underlying HW+SW platform
  - SW for flexibility and reduced design costs
  - HW for to improve performance, unit cost and energy consumption
- Ideal goal of Model-Based Design:
  - Model once
  - Verify once
  - Synthesize to many targets (SW, FPGA, HW, …)
- Requires verification and synthesis techniques, that will be covered in this course

Outline

- Motivation:
  - Definition of embedded systems
  - Moore's law
  - The economics of electronic system design
- Reduction of design costs and performance/cost optimization
  - Platforms
  - Energy/performance trade-offs
- **Modeling languages and expressive power**
  - Data-dominated vs control-dominated
  - Turing machines and undecidability

Control-dominated versus data-dominated

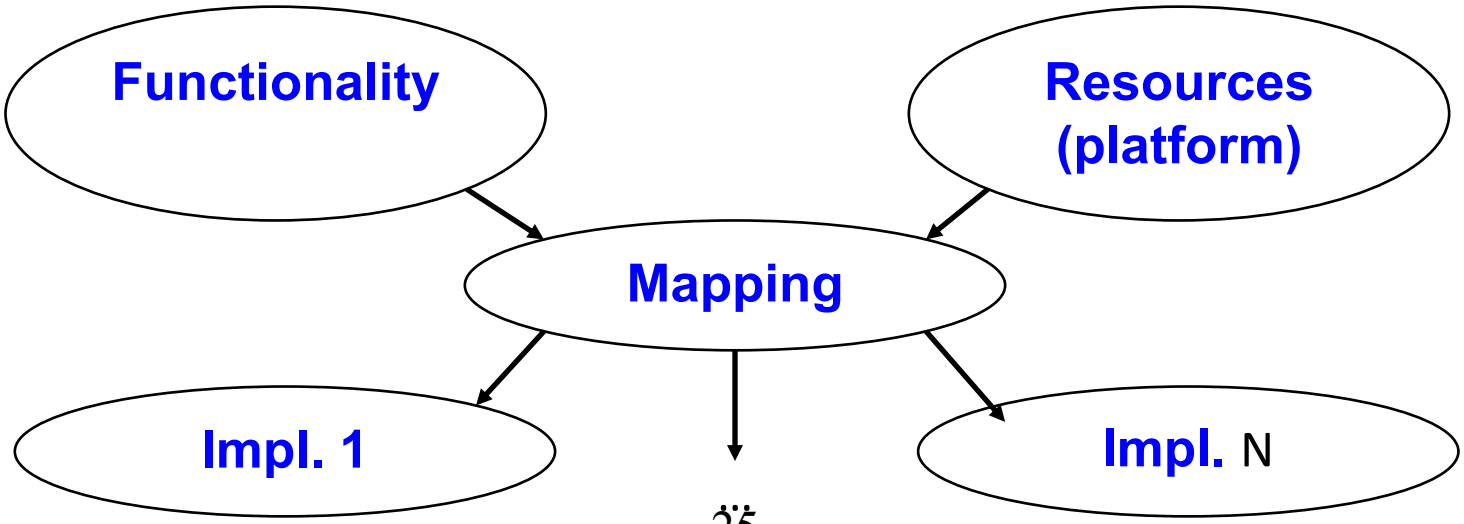
- No single model is suitable for every system
- Control-dominated systems emphasize:
  - Decisions
  - Rapid reaction to stimuli, latency
- Data-dominated systems emphasize:
  - Numerical computations
  - Throughput
- Control-dominated system design is based on extended, concurrent Finite State Machines
- Data-dominated system design is based on Dataflow Networks
- “Real” designs require a mix of both…

Expressive power versus analyzability

- Ideally, a design model should be:
  - **Expressive**, to represent anything a designer would like to design
  - **Compact**, because design time is proportional to the number of designed "objects" (from transistors to FFT stages)
  - **Executable**, to enable simulation-based verification
  - **Analyzable**, so that properties can be easily verified:
    - Correctness ("design verification")
    - Equivalence ("implementation verification")
  - **Synthesizable** and **optimizable**, so that many different implementations can be obtained and explored quickly
- Unfortunately, these properties are incompatible
- Expressiveness implies limited analyzability, which in turn implies limited optimizability

Platform-based design and Y-chart

- A design model should be **orthogonal**, to allow **separation of concerns** between:
  - Functional ("what") aspects
  - Non-functional ("at what cost and performance") aspects
- The implementation ("how") can then be derived by synthesis
  - Or, in the "old world", by manual refinement…



How do we actually use modelling methods to implement those systems ?

- So far, we have a nice mathematical theory, but how do we use it to ***optimize cost and performance of mixed HW/SW implementations of embedded data-dominated applications?***
- We need a ***cost function to optimize***
- We already found one aspect of the cost function: FIFO memory size (on-chip or off-chip RAM)
- From now on, we will assume ***SW execution on a single processor*** (we will look at HW later)
- Code size also matters (on-chip or off-chip RAM, ROM or FLASH)
- Estimating code size require hypotheses on how the code is implemented

Architectural assumptions

- We will first consider SW execution on a single ***Application-Specific Instruction set Processor (ASIP) optimized for data-dominated applications***
- Execution on multiple processors changes little about the code size discussion
- Different processor architectures change the cost function but ***do not change the design space exploration methodology***
  - Balance equations are the key to capture the full set of legal schedules
- Of course, ***multiple resources change performance*** a lot
  - We will discuss this more in the context of scheduling for ***HW***
  - Scheduling on multiple processors is like HW scheduling

Application-Specific Instruction set Processors

- Combine some advantages of software (low NRE, ease of update) with some of hardware (performance and power)
- Customize a processor's Instruction Set Architecture (ISA) and memory architecture for a ***specific application*** (or application area)
- "Make the common case fast"
- Remove little used instructions
- Examples:
  - encryption processor (good support for shift and exor)
  - Digital Signal Processor

Digital Signal Processor (DSP)

- Fully programmable (like a general-purpose CPU)
- Instruction Set Architecture and memory optimized to execute a particular class of algorithms: repetitive numerical calculations on relatively large vectors or matrices
  - Little decision making (if-then-else, data-dependent loops)
  - Lots of arithmetic operations
  - Fixed iteration count (data-independent) loops
- Examples:
  - Digital filters (Finite Impulse Response, Infinite Impulse Response)
  - Direct and Inverse Fast Fourier Transforms

Digital Signal Processor (DSP)

- Optimized for vector dot product (basis of FIR, correlation, …)
- Fast multiply and accumulate (MAC)
- Large memory bandwidth (will not discuss further)
- Fixed iteration (or infinite iteration) loops:
  - Single instruction, executed only one when loop starts (Zero-Overhead loops)
  - No pipeline flushing at the end of each iteration!
- No conditionals, i.e. no need for efficient:
  - Subroutine calls
  - Conditional jumps
- Good for real-time applications (predictable execution times)

SW architecture assumptions

- FIFOs mapped to RAM buffers, ***without memory sharing or dynamic allocation*** (fast, predictable code)
- Hardware support for zero-overhead loop
- Process code is ***inlined***, because ***subroutine calls are inefficient***:
  - Pipeline disruption
  - Cost of saving and restoring registers
- Standard code structure: “static looped code”

```
while (1) {
  for (i = 0; i < X_A; i++) { code of process A; }
  for (i = 0; i < X_B; i++) { code of process B; }
  …
}
```

Consequences of architecture assumptions

- FIFO memory size depends on the max size of each FIFO throughout the execution of the schedule
- Code size depends on the ***number of appearances of each process in looped code***
- Example: AAAABACCC

```
while (1) {
  for (i = 0; i < 4; i++) { code of process A; }
  code of process B;
  code of process A;
  for (i = 0; i < 3; i++) { code of process C; }
}
```

- Total size, ignoring cost of loop (1 instruction):
  2 * size(A) + size(B) + size(C)

High-level synthesis

- Typically divided into several phases:
  - ***Allocation***, i.e. choice of how many resources (adders, subtractors, ALUs, multipliers, …) will be used
  - ***Scheduling*** of the processes (i.e. operations) on the set of selected resources
  - ***Binding*** of operations to resources
- Simultaneous solution of these three problems would lead to better results (closer to the Pareto-optimal set), but it is typically ***too complex***
- Even finding the set of Pareto optimal schedules on multiple resources has exponential complexity in the number of processes and resources

HLS scheduling problems

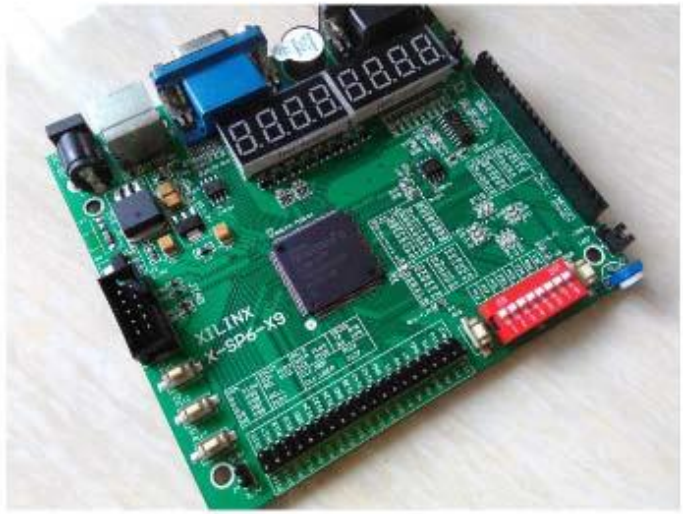
- Since the Pareto-optimal set has too many solutions, typically people pose and solve two variants:
  - Performance-constrained scheduling: find the schedule using the smallest set of resources and that can be executed within a given latency (number of clock cycles)
  - Resource-constrained scheduling: find the schedule that can be executed within the smallest latency on a given set of resources
- Most “real” problems are performance-constrained
- An algorithm solving one of the two can solve the other
  - E.g. given a resource-constrained scheduling algorithm, try to increase the set of resources until the desired latency is achieved

The Fundamentals

- Why use FPGAs?
- Applications
- Some Technology Background
- • FPGA programming technologies

FPGAs (Field-Programmable Gate Arrays)

- FPGAs (Field-programmable Gate Arrays)



FPGA Definitions (I)

- Field Programmable Gate Arrays (**FPGAs**) are digital **integrated circuits** (ICs) that contain **configurable (programmable) blocks of logic along with configurable interconnects** between these blocks. Design engineers can configure, or program, such devices to perform a tremendous **variety of tasks**.
- Depending on how they are implemented, **some FPGAs may only be programmed a single time**, while others may be reprogrammed over and over again. A device that can be programmed only once is referred to as one-time programmable (OTP).
- The "field programmable" portion of the FPGA's name refers to the fact that its **programming takes place "in the field"**. This is in contrast to devices whose internal functionality is hardwired by the manufacturer.

FPGA Definitions (II)

- In the field refers to that fact that, although FPGAs can be configured in the laboratory, w**e can modify the function of a device resident in an electronic system** that has already been deployed in the outside world.
- In this course we will be using the terms programmable logic devices (PLDs), application-specific integrated circuits (ASICs), application- specific standard parts (ASSPs) and FPGAs
- Just some notes for the history of FPGAs :
- •First FPGAs contained only a few thousand simple logic gates
- •The flows used to design these components, mostly schematic capture, were very easy to understand
- •Today, FPGAs are massively complex and there are many different design tools, flows and techniques

Historical development

- To give designers the **ability to implement different interconnections**
- in a single device
- One of the initial designs offered **two programmable planes**
- These planes provided any combination of **AND and OR gates**
- Common terms could also be shared between ANDs and ORs
- It offered flexible architectures
- When it was first introduced, the **wafer geometries** were significant (~10 um)
- This introduced a large propagation delay and made the devices relatively slow
- Timeline in the development of programmable logic has been:
  - **Standard logic products -> PLAs and PALs -> CPLDs -> FPGAs**

The PLA

Programmable Logic Array (PLA) first introduced in 1975

Two programmable planes

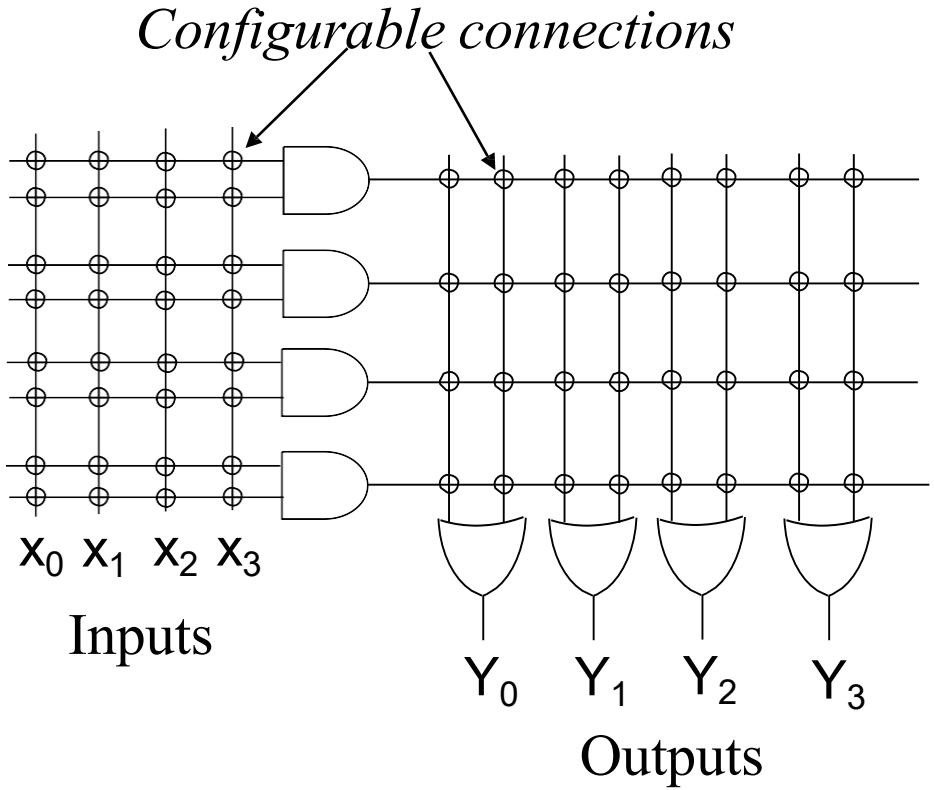
Any combination of ANDs and ORs

Sharing of AND terms across multiple ORs

Allows for a large number of logic functions to be synthesised (in a sum of products fashion)

Slower than Programmable Array Logic (PALs)

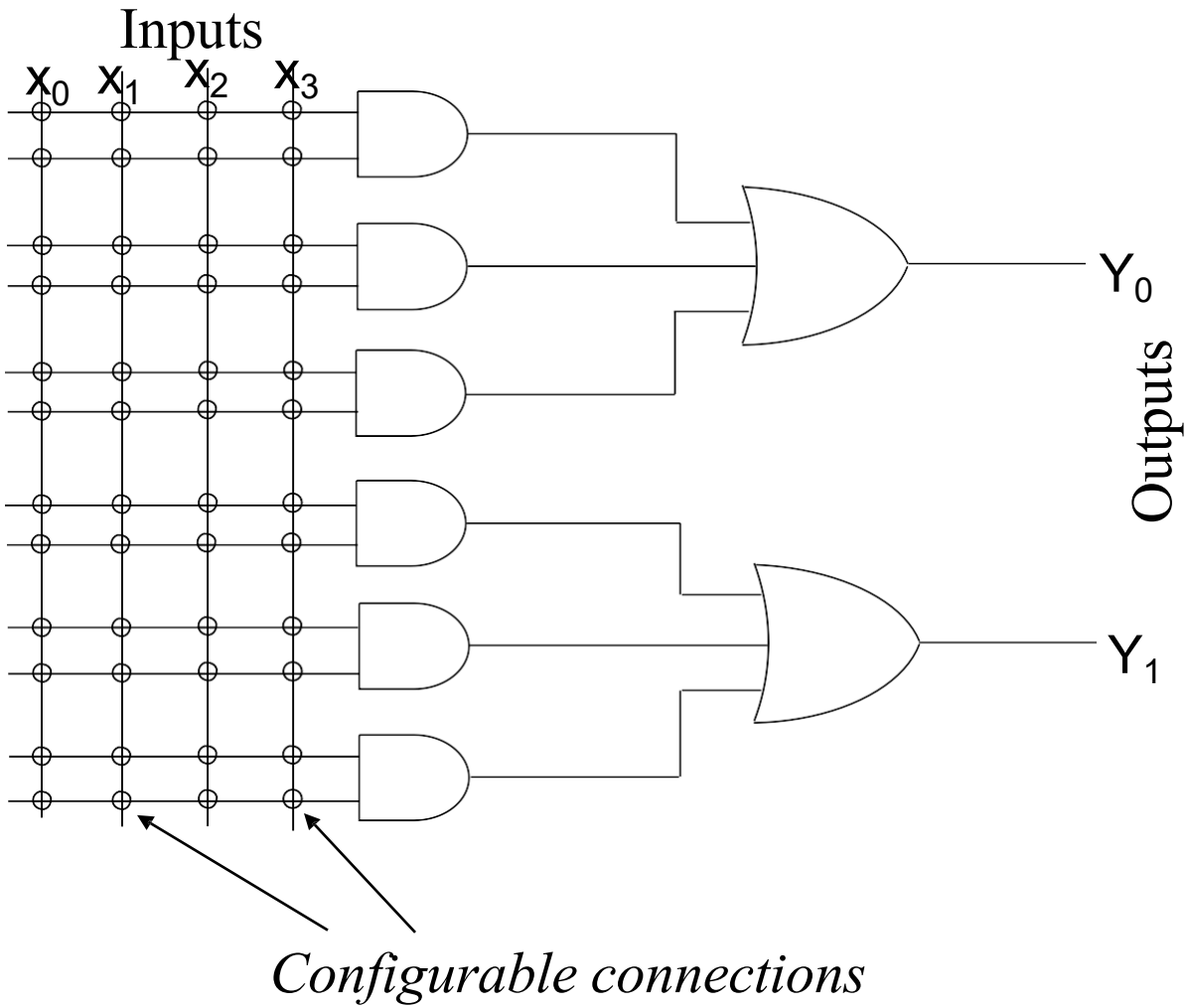
High fuse count -slower than PALs



The PAL (I)

- Programmable Array Logic (PAL)
- **One programmable plane is fixed (the OR array)**
- **One programmable plane free to interconnect (the AND array)**
- Added benefit of smaller propagation delays
- Less complex software
- Not the flexibility of the PLA structure
- **Also classified as belonging to a category of devices called Simple Programmable Logic Devices (SPLDs)**
- **Medium logic density available**
- **Lower fuse count** – faster than PLAs
- **A few hundred gates**

The PAL (II)



The PAL (III)

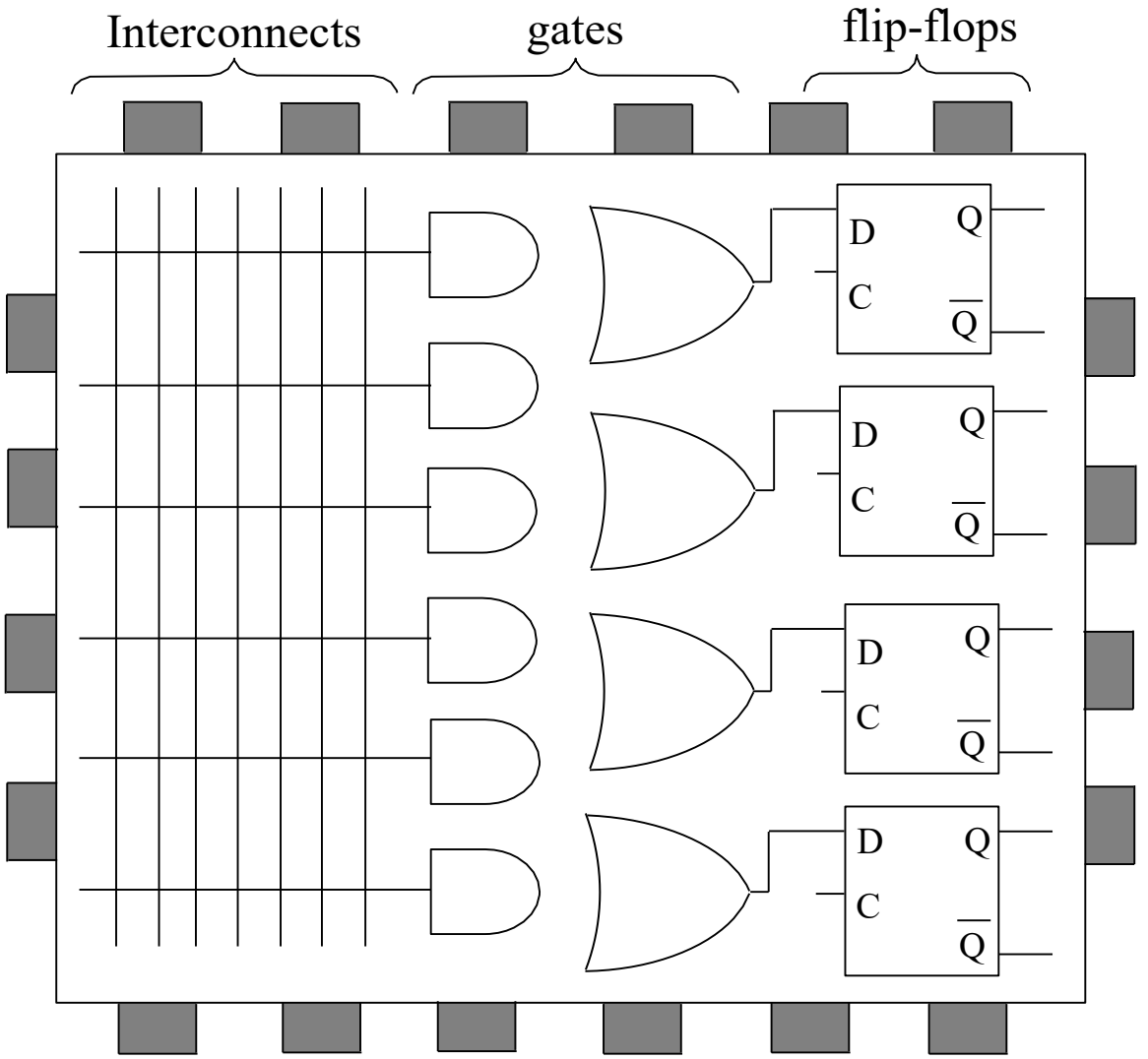
- The architecture was comprised of a mesh with horizontal and vertical interconnect tracks
- At each junction there was a fuse
- Through software programming, **the designer could choose how to connect each junction**
- This was achieved by “blowing” all unwanted fuses
- **Input pins were connected to vertical tracks**
- **AND and OR gates (also known as product terms) were connected to horizontal tracks**
- The product terms were **connected to dedicated flip-flops** the outputs of which were connected to output pins
- **PLDs provided as much as 50 times more gates** in a single package than existing discrete logic devices

CPLDs (I)

- Complex Programmable Logic Devices (CPLDs)
- They extend the density of SPLDs
- The concept is to have a **few PLD blocks (or *macrocells*) on a single device with a general-purpose interconnect in between**
- Simple logic paths are implemented within an individual logic block
- More sophisticated logic can utilise multiple blocks and the general-purpose interconnect
- CPLDs are efficient in handling complex circuits at speeds up to 200 MHz (~ 5 nanoseconds)
- Easy to calculate the timing model of your design (input – output speed)

CPLDs (II)

- Central, global interconnect
- Deterministic timing
- Easily routed
- Wide and fast complex gating



CPLDs (III)

- Can be synthesised to **build circuit prototypes**
- Allow for **debugging to take place at hardware level**
- **Any changes can be introduced at software level**
- They offer short development cycles
- They have non-volatile memory (do not require an external ROM on start-up)
- **Large number of gates (typically in the order of thousands to tens of thousands)**
- Allow the implementation of moderately complicated data processing devices
- Low cost

Why use FPGAs?

- **PLDs are devices whose internal architecture is predetermined by the manufacturer, but can be configured in-field to perform a variety of functions.**
- They **contain few logic gates and can implement only simple functions**
- **At the other end, ASICs can contain hundreds of millions of logic gates and can implement incredibly complex functions.**
  - However, they are custom-designed to address a specific application.
  - Also, designing and building one is an extremely time-consuming and expensive process.
  - The final design is “frozen” in silicon and cannot be modified without manufacturing a new version.
- FPGAs occupy the middle ground!!

Technology trade-offs

- **The cost of an FPGA is much lower than that of an ASIC**
- **ASIC components only become cheaper in large production runs**
- Implementing design changes is much easier in FPGAs
- Time to market for FPGAs is much faster
- **Hence, FPGAs make small design companies viable because they allow engineers to realise their hardware and software concepts on (relatively) economical FPGA-based test platforms**

The Last 30 Years

- First FPGA introduced in 1984 by Xilinx
- The term was popularized in 1988 by Actel

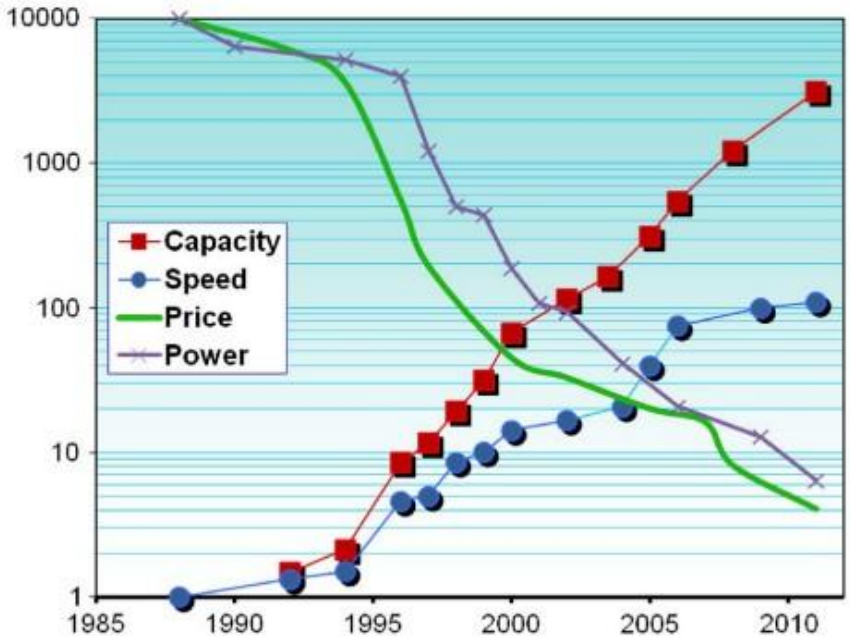
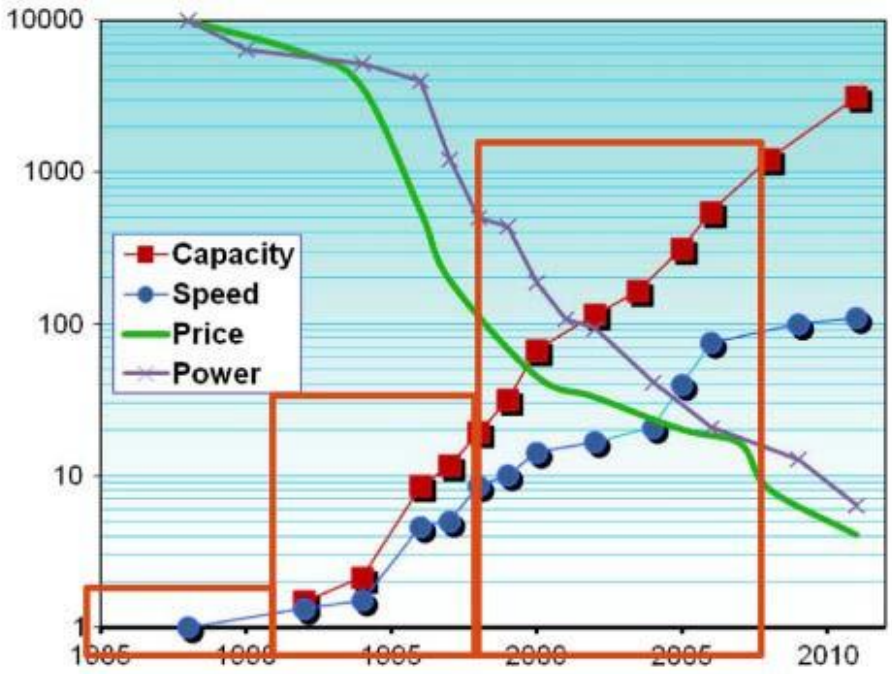


Fig. 1. *Xilinx FPGA attributes relative to 1988. Capacity is logic cell count. Speed is same-function performance in programmable fabric. Price is per logic cell. Power is per logic cell. Price and power are scaled up by 10 000×. Data: Xilinx published data.*

- Over the last 30 years...
  - capacity increased by a factor of 10,000
  - speed increased by a factor of 100
  - price decreased by a factor of 1,000
  - power consumption decreased by a factor 1,000

The three “ages” of FPGAs



- Age of Invention: 1984 – 1991
- Age of Expansion: 1992 – 1999
- Age of Accumulation: 2000 – 2007

Fig. 1. ***Xilinx FPGA attributes relative to 1988. Capacity is logic cell count. Speed is same-function performance in programmable fabric. Price is per logic cell. Power is per logic cell. Price and power are scaled up by 10 000×. Data: Xilinx published data.***

Stephen M. Trimberger, “Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology”

2012 – today Age of **Wide-Use** …. Will it continue ??

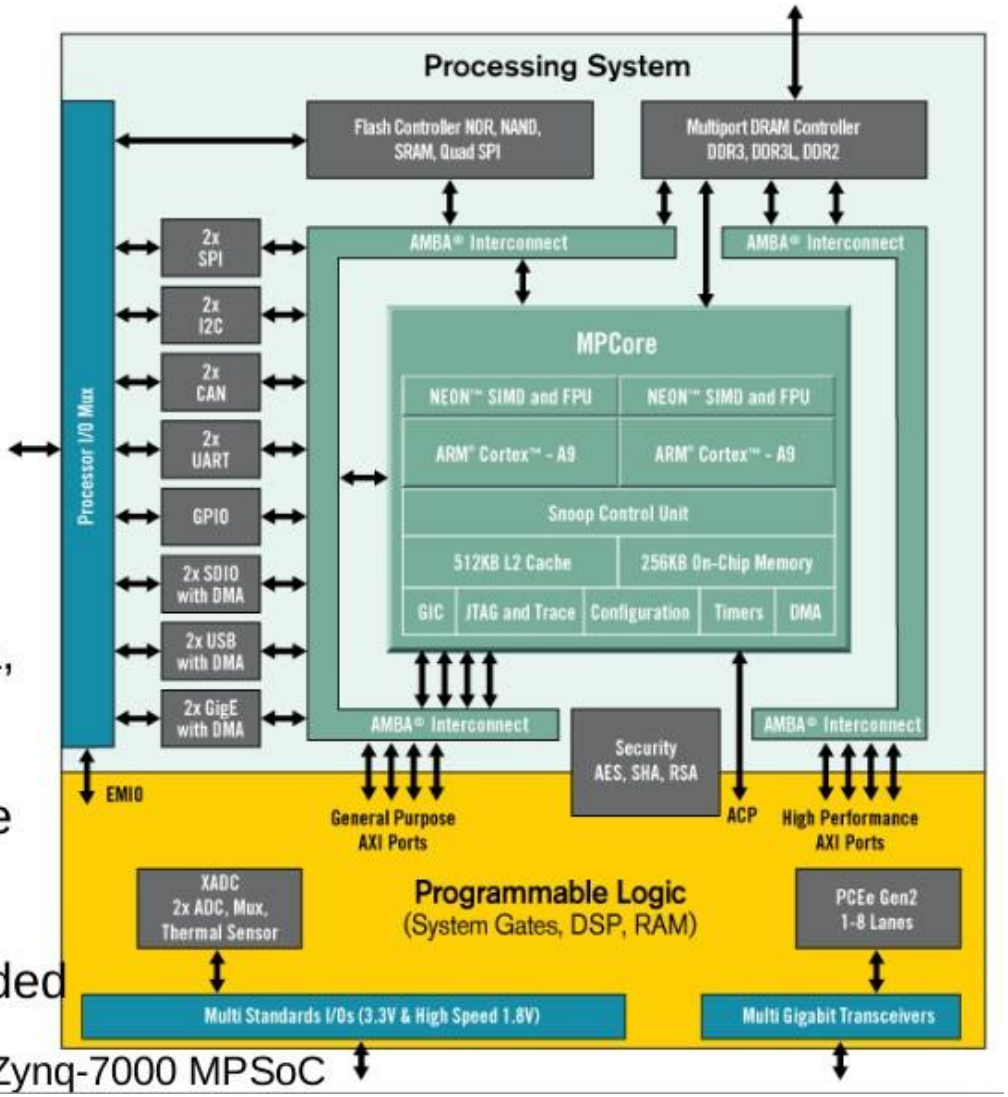
The current age of FPGAs

- Previously, FPGAs moved from arrays of gates to hard blocks and programmable logic
- Added programmability introduced a design burden (major adoption drawback in comparison with multi-core processors and GPUs)
- The need for lowering costs and power consumption remains and the end of Dennard scaling does not help
- FPGA crossover point at the level of millions
  (ASIC chips over that point: microprocessors, memories, cell phone processors)
- ASICs evolved to ASSP SoCs (Application-specific standard product system on chip, fixed-function blocks and microprocessors)
- FPGAs evolved to programmable SoCs (full programmable SoC with memory, microprocessors, on-chip network, and programmable logic)

Today's programmable

- Software and Hardware programmable devices
- Full environment of
  - caches
  - interconnenct
  - peripherals
    etc

- Design effort?
  - Increasingly explored solutions include high-level synthesis based on C, Cuda, OpenCL
  - Vendors provide libraries of logic and functions to reduce design costs
  - Widely needed functionality being done by vendor-provided tools automatically



Xilinx Zynq-7000 MPSoC

FPGA Applications

- **When they first arrived, i.e. mid-1980s**, they were mostly used as glue logic, **medium complexity state machines and very limited data processing tasks**
- **Early 1990s, mostly in telecommunications and networking, i.e. processing large blocks of data and pushing that data around**
- End of 1990s, **strong presence in consumer, automotive and industrial applications**
- **FPGAs are often used to prototype ASIC designs** or to provide a HW platform on which to verify the physical implementation of algorithms
- They can implement just about anything, e.g. communication devices, radar, image, DSP applications, system-on-chip (SoC) components and many more
- Hence, they are also found in final products

Major FPGA Market Segments

- ASIC and Custom Silicon
- Digital Signal Processing
- Embedded Microcontrollers and Microprocessors
- Physical Layer Communications
- Reconfigurable Computing

FPGA ARCHITECTURES

Programming technologies (continued)

- SRAM-based devices
- Antifuse-based devices
- E^2PROM/FLASH devices
- Hybrid FLASH-SRAM devices

SRAM-based devices

- Majority of FPGAs are based on the use of SRAM configuration cells
- New designs can be quickly implemented and tested
- **Also, when the system is first powered up, the FPGA can be programmed to run a self-test and then be reprogrammed to perform its main task**
- **But, they have to be reconfigured every time the system is powered up**
- This either **requires the use of a special externa memory** device or of an on-board microprocessor
- Hence, it **becomes difficult to protect your design** from theft since the configuration file is stored in some form of external memory
    - Safety mechanisms exist, such as **bitstream encryption**
    - **Encryption key** stored in a special SRAM-based register in the FPGA
- Requires battery backup

Antifuse-based devices

- **These devices are programmed off-line using a special device programmer**
- They are **non-volatile**, i.e. immediately available when powered on
- They do **not need an external memory chip** to store their configuration data
- **Their interconnect structure is relatively immune to the effects of radiation**, very important in the case of aerospace applications
- Any additional electronic components, such as flip-flops, usually implement the triple **redundancy** design
- The antifuse-based device's configuration data are buried deep within the device itself, making it resistant to reverse engineering efforts
- Disadvantage – they are OTP!
- **Compared to SRAM-based, they consume less operational as well as standby power**
- They are behind in terms of technology generations compared to SRAM

E^2PROM/FLASH-based devices

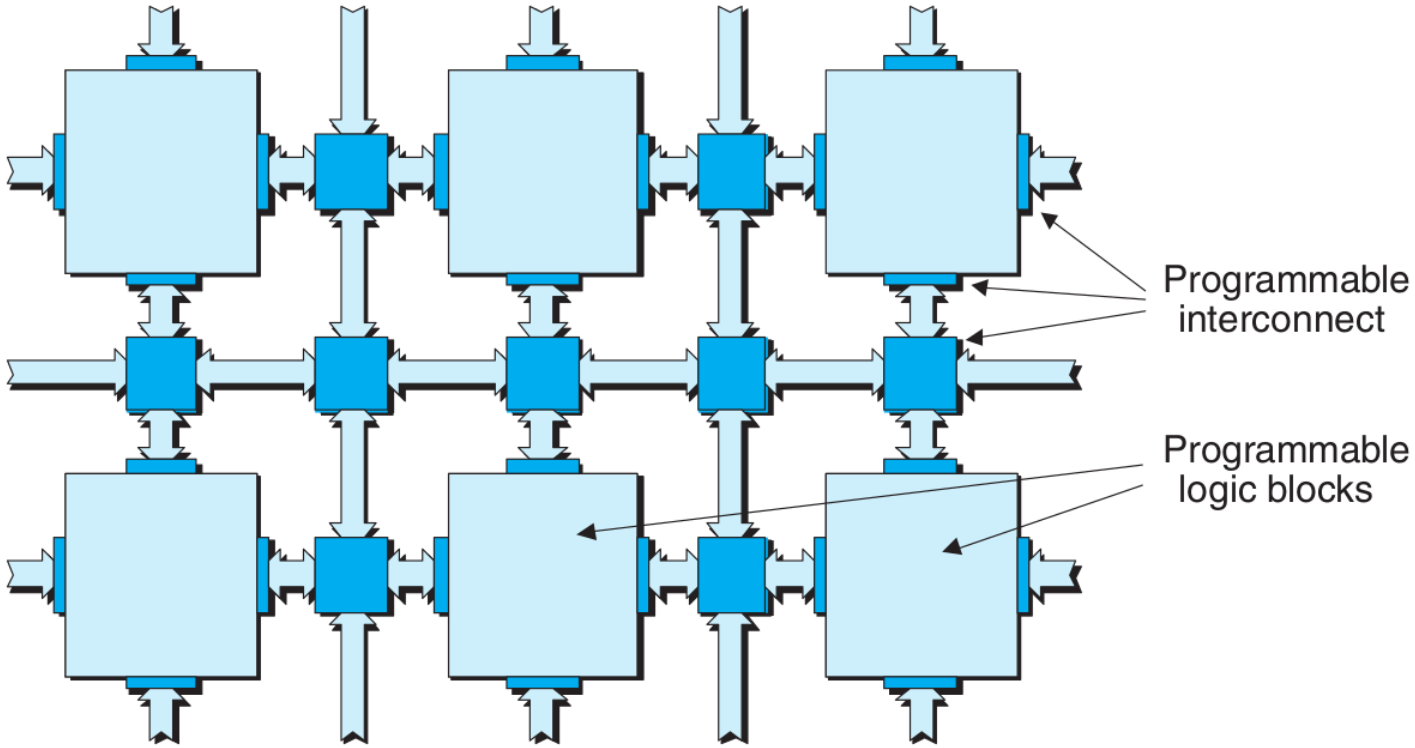
- **Usually configured off-line but some versions are in-system programmable**
- Programming time, however, is three times that of an SRAM equivalent
- **They are non-volatile, i.e. instantly ON once power is provided**
- Can use the concept of mutltibit key for security applied **through the JTAG port (brute force attack would take billions of years due to port's low operating frequency)**
- **E^2PROM cells are considerably smaller than the SRAM ones, so the rest of the logic can be much closer, thereby reducing interconnect delays**
- Disadvantage – They require additional process steps on top of standard CMOS, hence, they lag behind SRAM in terms of technology generations
- **Also, tend to have high static power consumption**

Hybrid FLASH-SRAM devices

- Some FPGA vendors offer esoteric combinations of programming technologies
- For instance, a device with each configuration element formed from the combination of a FLASH cell and an associated SRAM cell
- The FLASH elements can be preprogrammed
- At power up, the FLASH contents are copied into their corresponding SRAM cells
- **This combines non-volatility with in-field reconfiguration**

Fine-, medium-, and coarse-grained architectures (I)

- It is common to categorise FPGAs using the terms above
- The underlying **fabric of an FPGA consists of large numbers of relatively simple programmable logic block "islands" embedded in a "sea" of programmable interconnect**



Fine-, medium-, and coarse-grained architectures (II)

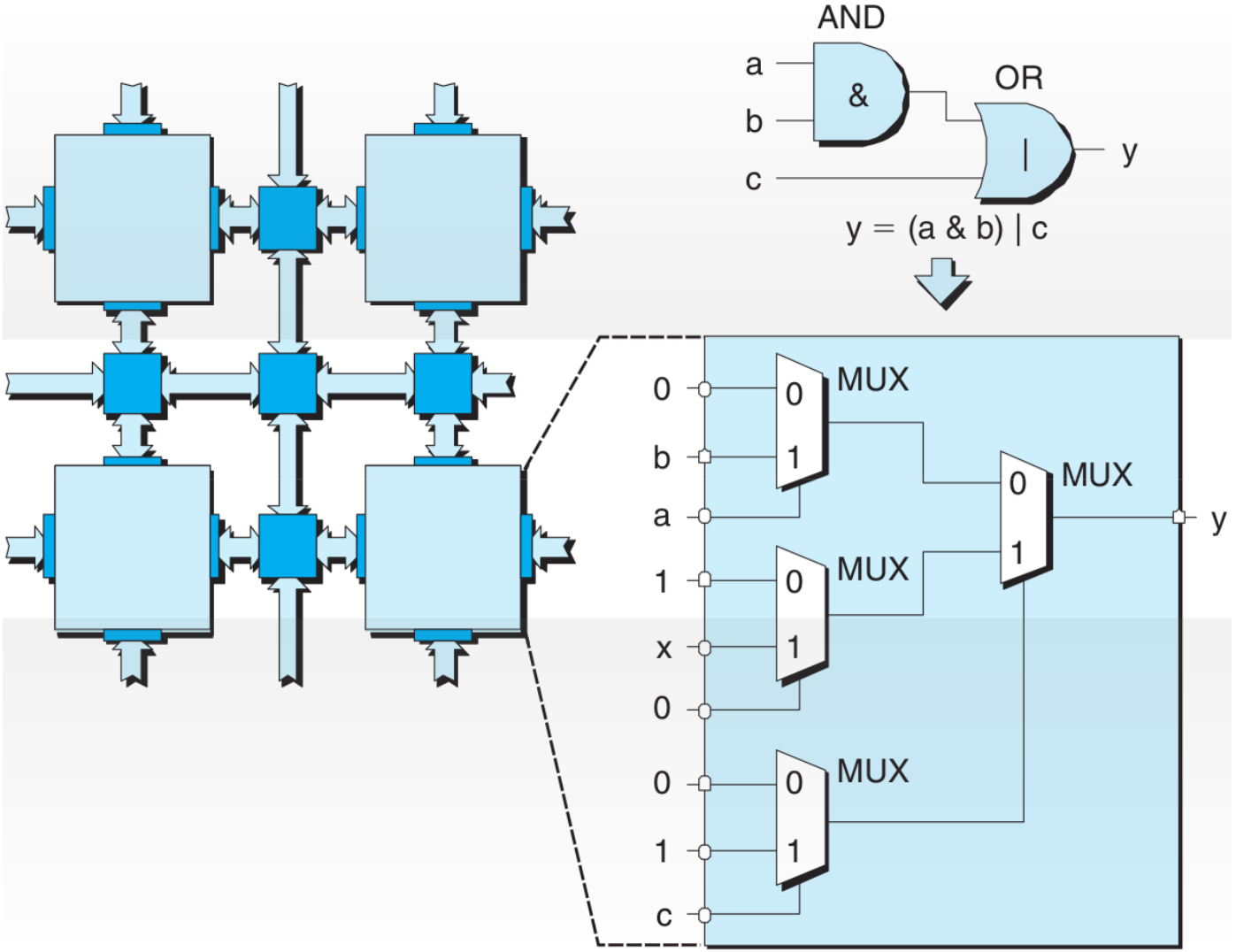
- Fine-grained architecture: Each logic block can be used to **implement only a very simple function**, such as a 3-input function or a storage element
- Coarse-grained architecture: Each logic block contains a relatively large amount of logic, for instance, four 4-input LUTs, four multiplexers, four D- type flip-flops and some fast carry logic
- Fine-grained implementations require a relatively large number of connections into and out of a logic block compared to the amount of functionality they contain
- As the granularity increases, the amount of connections decreases compared to the amount of functionality they support
- Hence, interblock delays play less role the coarser the granularity becomes
- Fine-grained architectures are particularly efficient when executing systolic algorithms

Logic Blocks

- There are two fundamental implementations for the programmable logic blocks
- The MUX (multiplexer)-based and the LUT (lookup table)-based

- **MUX-based**:
  - •Consider one way in which the 3-input function $y = (a \,\&\, b) \mid c$ could be implemented using a logic block containing only multiplexers
  - •The device is programmed such that each input to the block is presented with a logic 0, a logic 1, or the true or inverse version of a signal coming from another block or from a primary input
  - •This way each block can be configured in many different ways

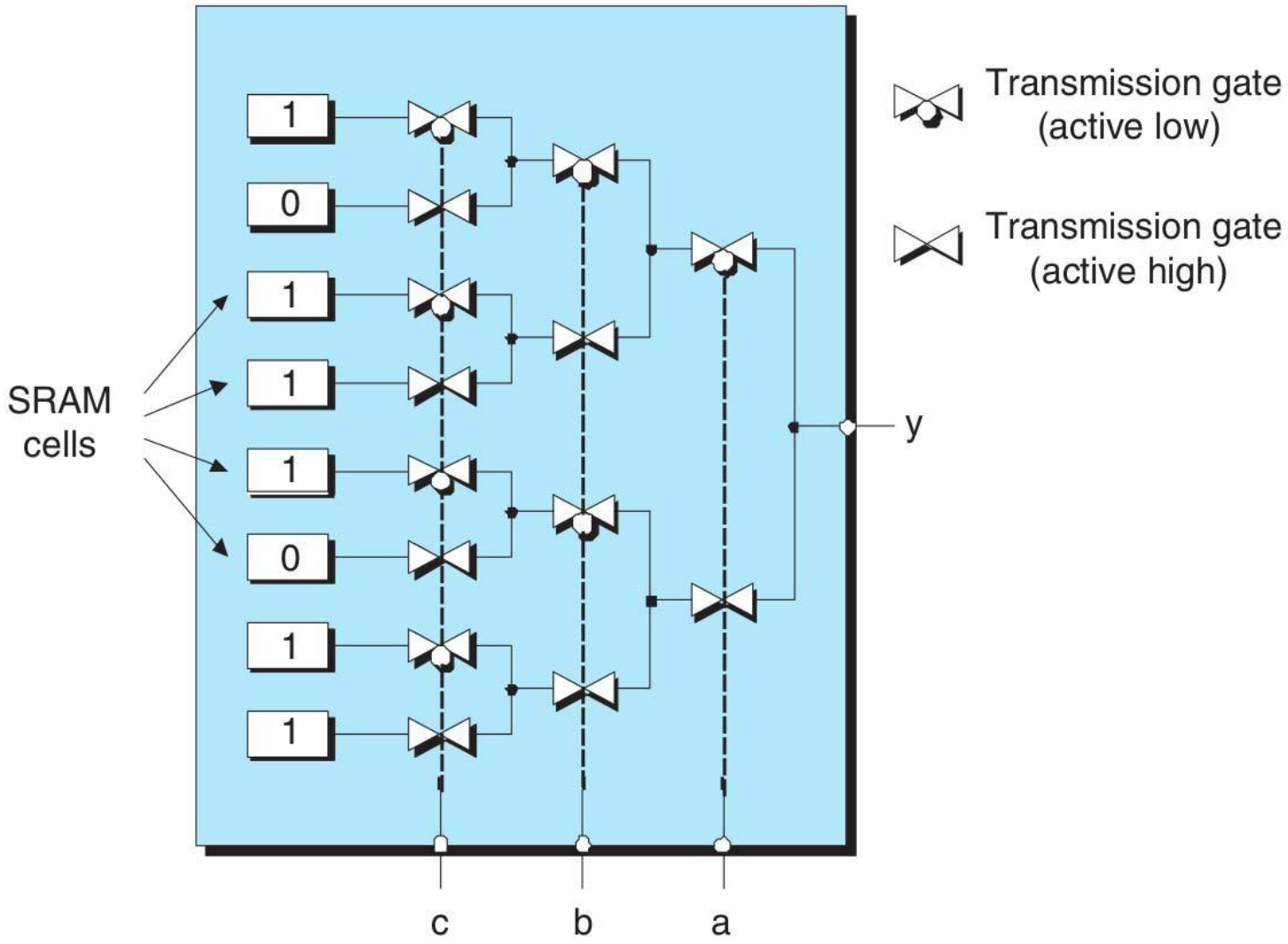
MUX-based logic block



LUT-based logic block (I)

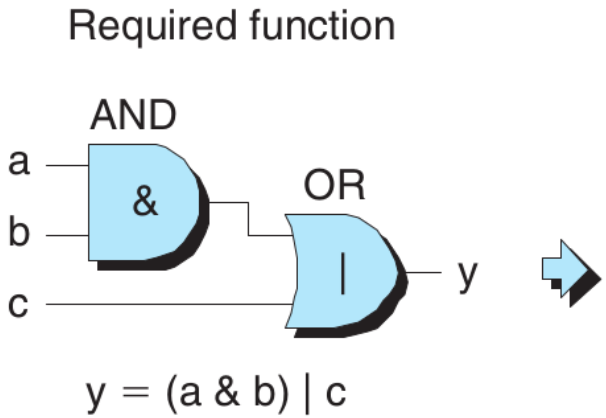
- Simple concept
- A group of input signals is used as an index to a lookup table
- The contents of the table are arranged such that the cell pointed to by a specific combination contains the correct and desired value
- For instance, consider *y = (a & b) | c*
- This can be achieved by loading a 3-input LUT with appropriate values
- **A commonly used technique is to use the inputs to select the desired SRAM cell using a cascade of transmission gates**
- If a transmission gate is enabled , it passes the signal from its input to its output
- Hence, different input combinations can be used to select the contents of different SRAM cells

LUT-based logic block (II)



LUT-based logic block (III)

•LUTs are useful when wanting to represent a group of logic gates several layers deep, it is strong in terms of resource utilisation and delays



Truth table

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

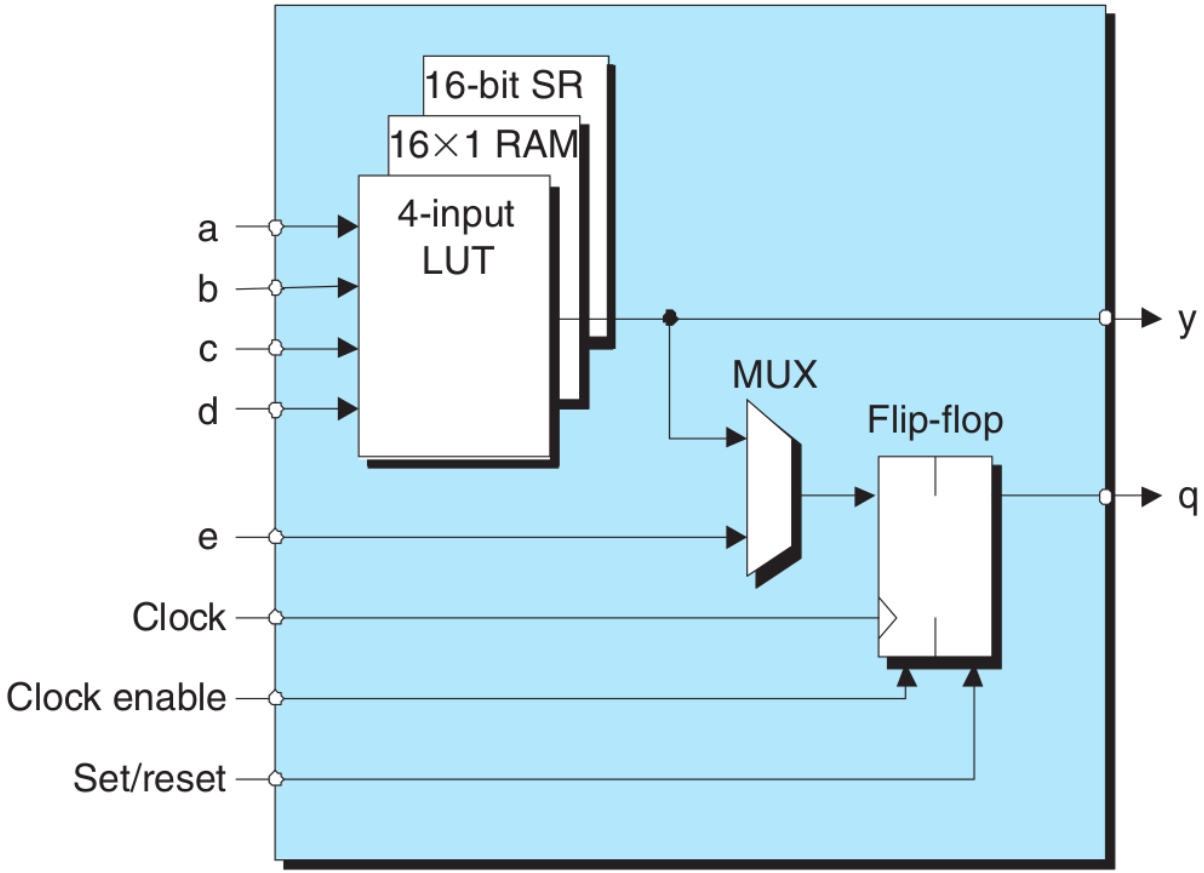
•However, if you want to implement a small function you are “wasting” an entire LUT and the associated delays are significant for a simple function

LUT-based logic block (IV)

- MUX-based logic blocks can be advantageous in terms of performance and silicon utilisation for designs containing large numbers of independent simple logic functions
- That is because it is often possible to gain access to the intermediate values from the signals connecting specific logic gates to specific MUXes
- LUT-based are most commonly used in arithmetic processing while MUX-based offer some advantages at implementing control logic
- Contemporary LUT-based logic blocks use 4-input LUTs
- Note that a LUT-based programmable logic block also contains other elements, such as multiplexers and registers

Logic Cells/Logic Elements (I)

- Each FGPA vendor uses unique terminology for their building blocks
- For example, the core building block in a Xilinx FPGA is called *logic cell*
- It is comprised of a 4-input LUT, a multiplexer and a register



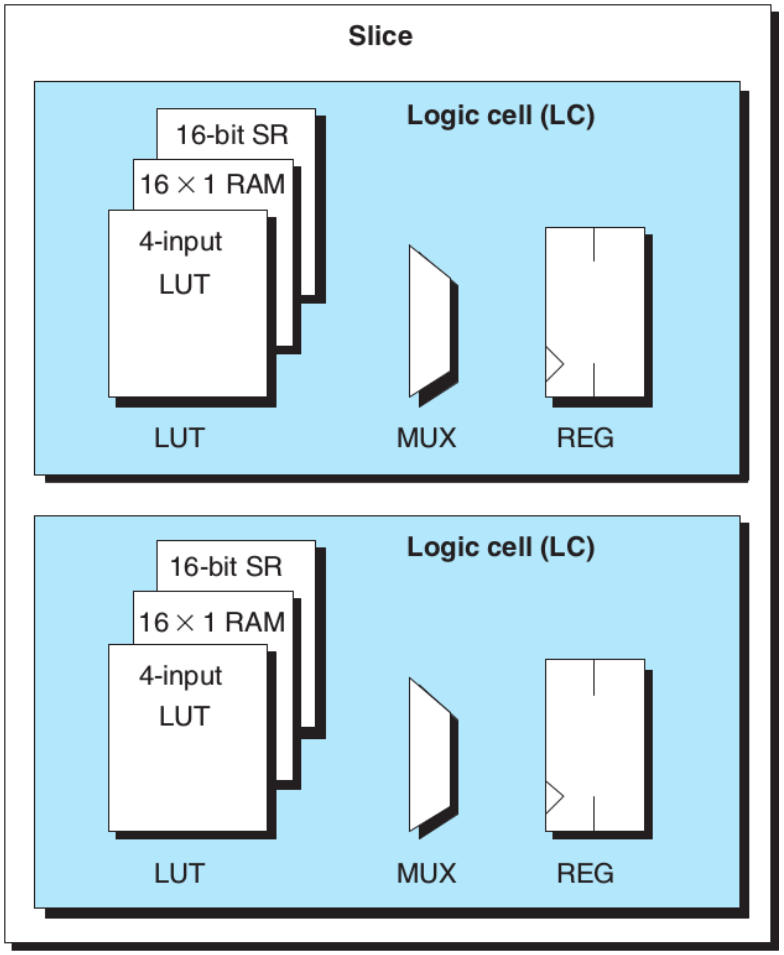
Logic Cells/Logic Elements (II)

- The register can be configured to act as a flip-flop or as a latch
- The polarity of the clock is also configurable as well as the clock enable and set/reset signals
- The equivalent core building block in an Altera FPGA is called *logic element* with considerable differences to the *logic cell*

Slices

- The next step in the hierarchy (for Xilinx) is the *slice*
- A slice contains two logic cells

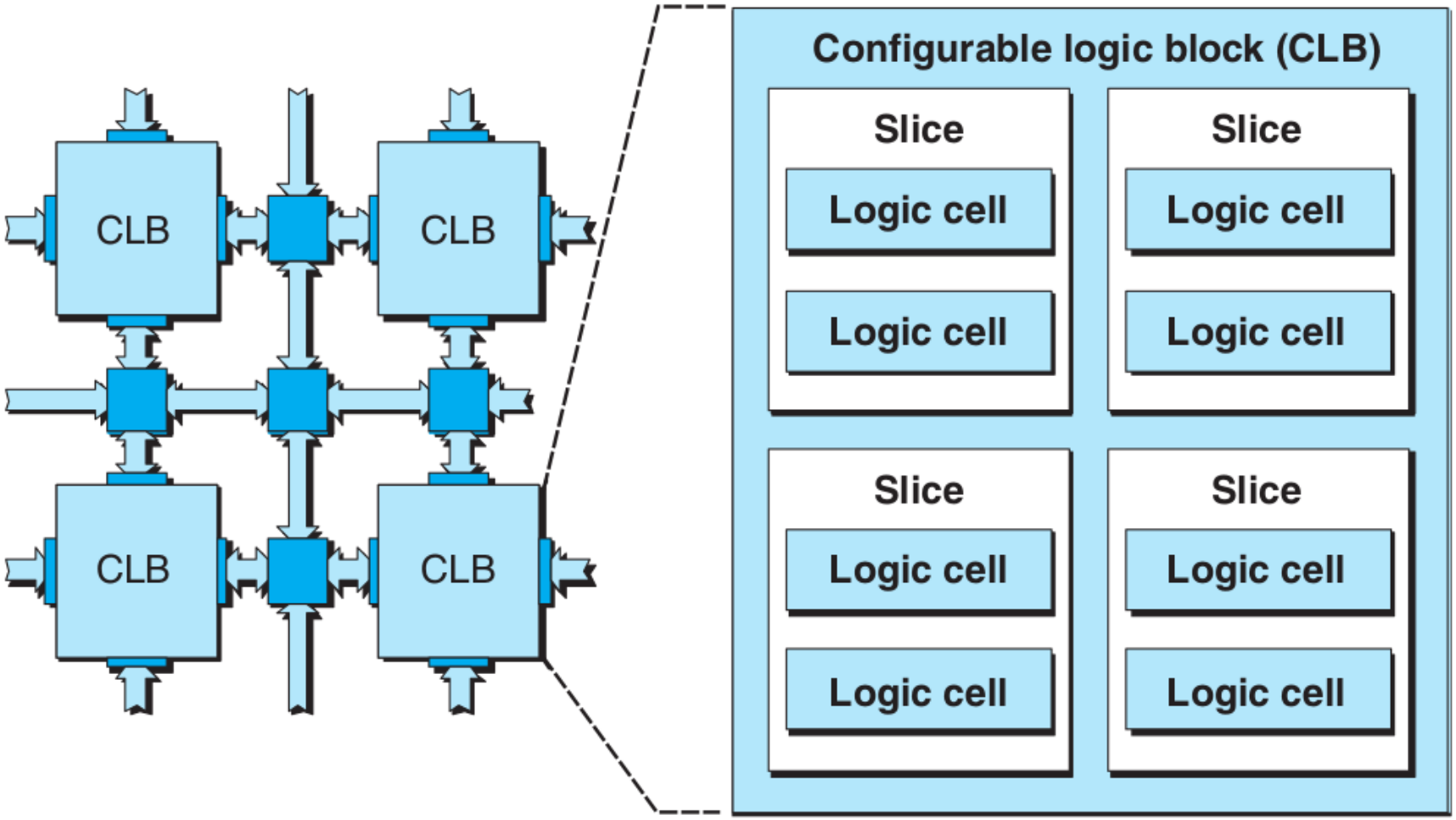
  - The LUT, MUX and register have their own inputs and outputs
  - But, clock, clock enable and set/reset signals are common in a slice



Configurable Logic Block (CLB) (I)

- One more level up the hierarchy, we come to the Xilinx *CLB* (Altera calls the equivalent block *logic array block* (LAB))
- Some Xilinx FPGAs have two slices in each CLB, while others have four
- A CLB commonly equates to a single logic block in the previous virtualisation of “islands” of programmable logic in a “sea” of programmable interconnect
- The reason for having this type of hierarchy, LC-Slice-CLB, is that it is complemented by an equivalent hierarchy in the interconnect
- There is a fast interconnect between the LCs in a slice
- A slightly slower interconnect between slices in a CLB and
- a slightly slower interconnect between CLBs
- This achieves an optimum trade-off between connecting things easily without excessive interconnect-related delays

Configurable Logic Block (CLB) (II)



Distributed RAMs and Shift Registers

- Each 4-bit LUT can also be used as a 16 x 1 RAM
- Assuming the four slices-per-CLB configuration, all of the LUTs within a CLB can be configured together to implement:

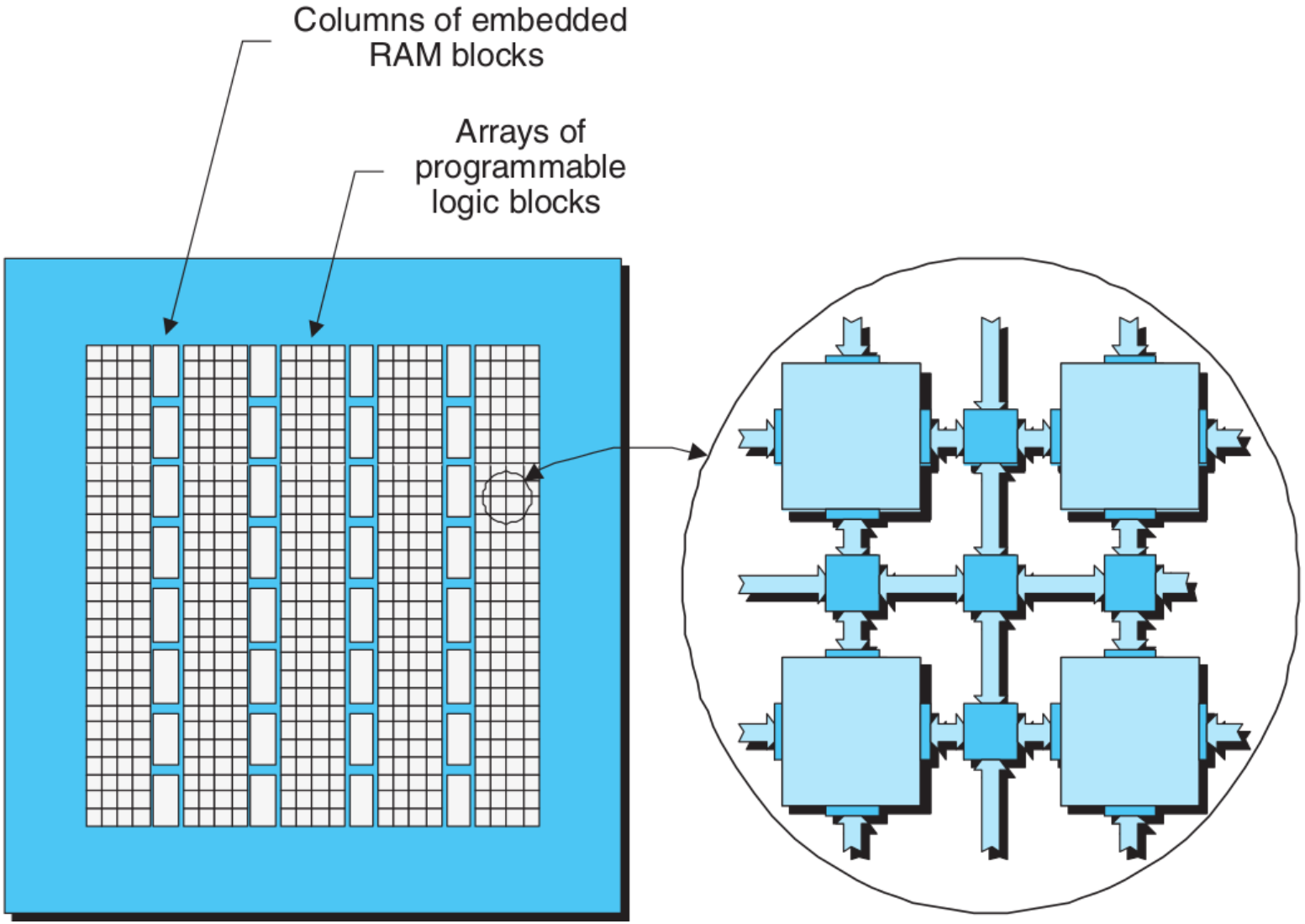
Single-port 16	x 8 bit RAM
Single-port 32	x 4 bit RAM
Single-port 64	x 2 bit RAM
Single-port 128	x 1 bit RAM
Dual-port 16	x 4 bit RAM
Dual-port 32	x 2 bit RAM
Dual-port 64	x 1 bit RAM

- Alternatively, each 4-bit LUT can be used as a 16-bit shift register
- Hence, the LUTs within a single CLB can be configured together to implement a shift register containing up to 128 bits

Embedded RAMs (Block RAMs) (I)

- Many applications require the use of memory
- FPGAs include large chunks of embedded RAM called *block RAM*
- (BRAM)
- Depending on the architecture, these blocks might be positioned around the periphery of the device, scattered across the face of the chip or organised in columns
- Depending on the device, a BRAM can hold from a few thousand to tens of thousands of bits
- A device may contain from tens to hundreds of BRAMs, thereby providing a total storage capacity between a few hundred thousand bits to several million bits (kilobytes to megabyte)
- A BRAM can be used independently or combined with others to implement standard single or dual port RAMs, FIFOs, state machines etc.

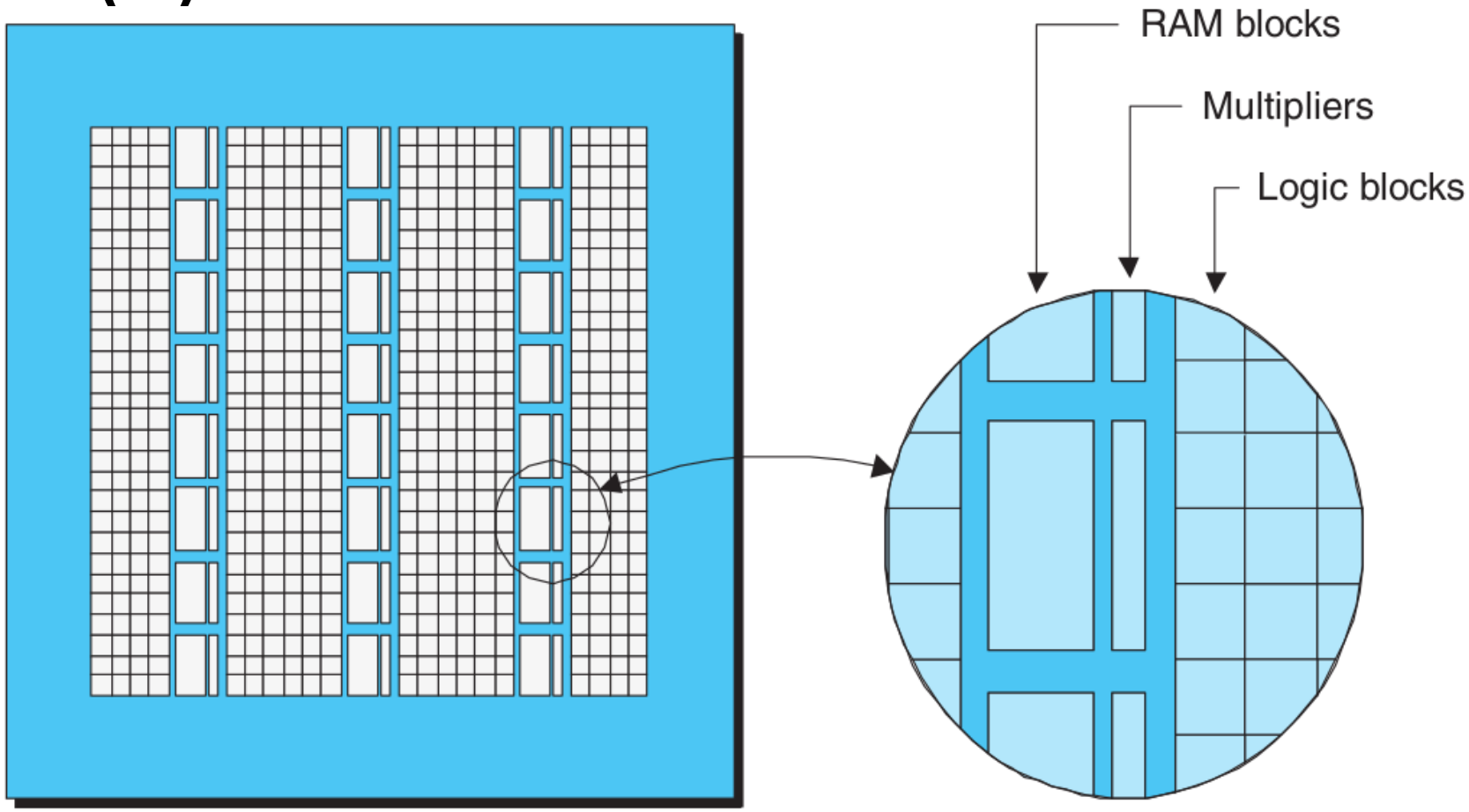
Embedded RAMs (Block RAMs) (II)



Embedded Multipliers, Adders etc. (I)

- Some functions, e.g. multiplier, are very slow if they are implemented by connecting a large number of programmable logic blocks together
- Since many applications require these functions, many FPGAs incorporate special hardwired multiplier blocks
- They are usually located close to BRAMs since a lot of the time they are used together
- Similarly, some FPGAs offer dedicated adder blocks
- One very popular DSP-type operation is the multiply-and-accumulate (MAC)
- If your FPGA supplies only multipliers, you have to combine them with an adder formed from a number of programmable logic blocks
- The result is stored in a flip-flop, BRAM or distributed RAM
- Some FPGAs provide MACs as embedded functions

Embedded Multipliers, Adders etc. (II)



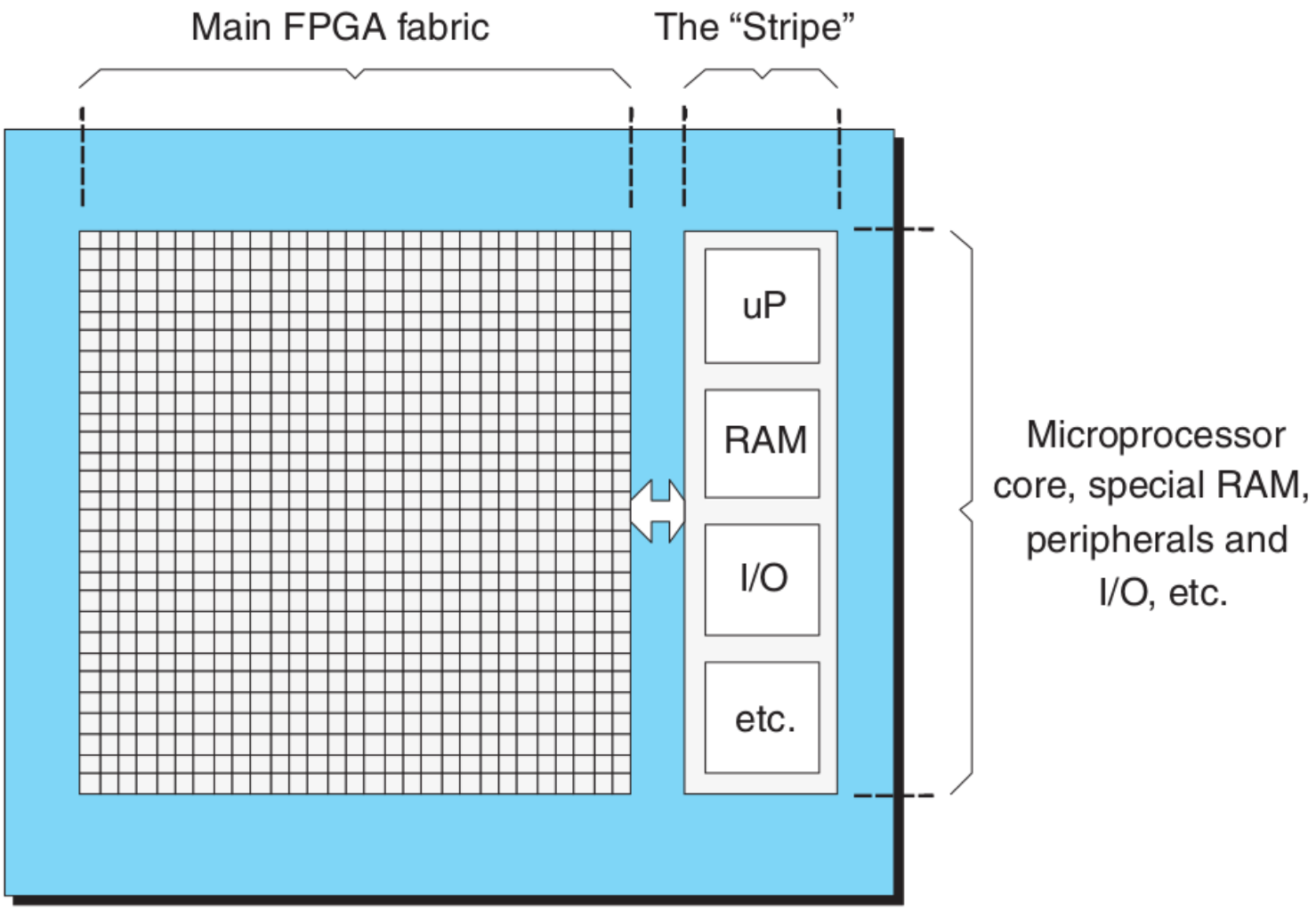
Embedded Processor Cores (I)

- An electronic design can be executed in hardware using logic gates, registers etc.
- It can also be executed in software as instructions to be executed on a microprocessor
- One of the main criteria is speed, i.e. how fast you wish the various functions to perform their tasks
- Pico- and nanosecond logic is for HW, millisecond logic is for microprocessors and microsecond logic is the middle ground
- The majority of designs make use of microprocessors in one form or another
- FPGAs usually contain one or more embedded microprocessors, referred to as microprocessor cores
- Split into two categories, i.e. *hard* microprocessor cores and *soft*
- microprocessor cores

Embedded Processor Cores (II)

- A hard microprocessor core is implemented as a dedicated, predefined block
  - Located in a strip, called “the stripe”, to the side of the main FPGA fabric
    - All of the components are typically formed on the same silicon chip
    - An advantage is that the main FPGA fabric remains the same for devices with and without a microprocessor core
    - Also, additional functions can be included in the stripe to support the microprocessor, such as memory, special peripherals etc.
  - Embed the microprocessor(s) into the main FPGA fabric
    - Many-core implementations are available
    - The design tools must take into account the presence of these cores
    - Any memory used by a core is formed from BRAMs
    - Any peripheral functions are formed from groups of general-purpose programmable logic blocks
    - Some potential speed advantages over the previous approach

Embedded Processor Cores (III)

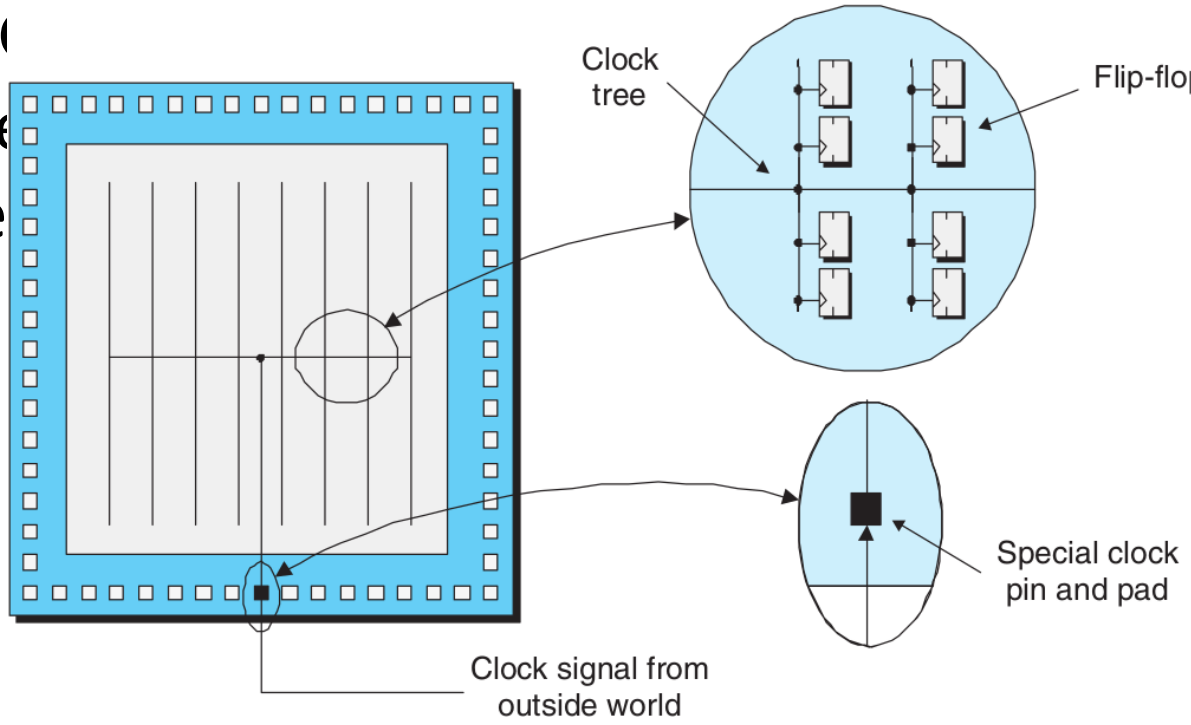


Embedded Processor Cores (IV)

- Instead of embedding a microprocessor physically into the fabric of the chip, it is possible to configure a group of programmable logic blocks to set up a soft microprocessor core
- Soft cores are simpler and slower than hard microprocessor cores
- They run at 30 to 50 percent of the speed of a hard core
- They have the advantage, however, of implementing a core only if you need it and that you can create as many cores as required until you run out of resources

Clock Managers (I)

- All of the synchronous elements inside an FPGA need to be driven by a clock signal
- Such signal typically originates in the outside world, comes into the FP ıted through the
- A tree-like nly used

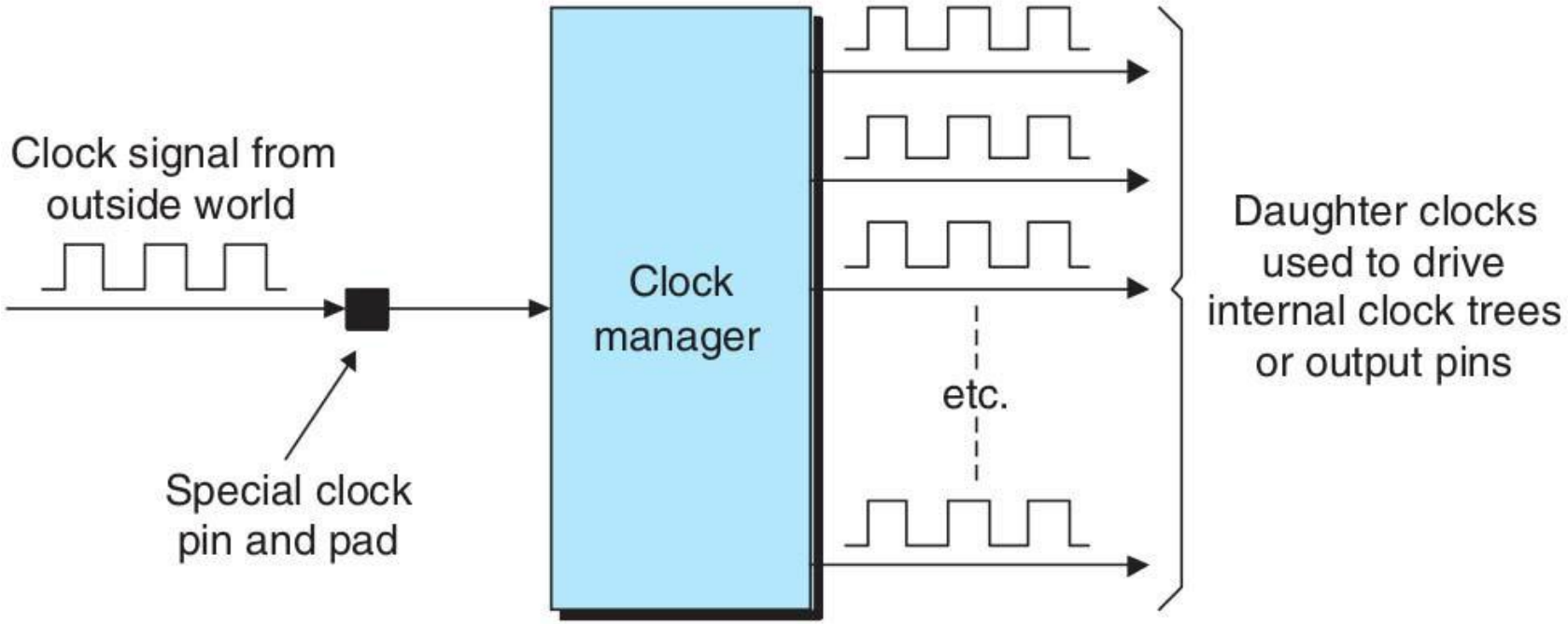


Clock Managers (II)

- The main clock signal branches to the flip-flops (leaves of the tree) on the end of the branches
- It ensures that all of the flip-flops see the clock signal as close together as possible
- Otherwise considerable deviations could occur between the clock signals of different flip-flops, which would introduce skew
- Skew can cause major problems
- Even with a clock tree, some amount of skew between the registers on a branch and between branches, exists
- The clock tree is implemented using special tracks and is separate from the general-purpose programmable interconnect

Clock Managers (III)

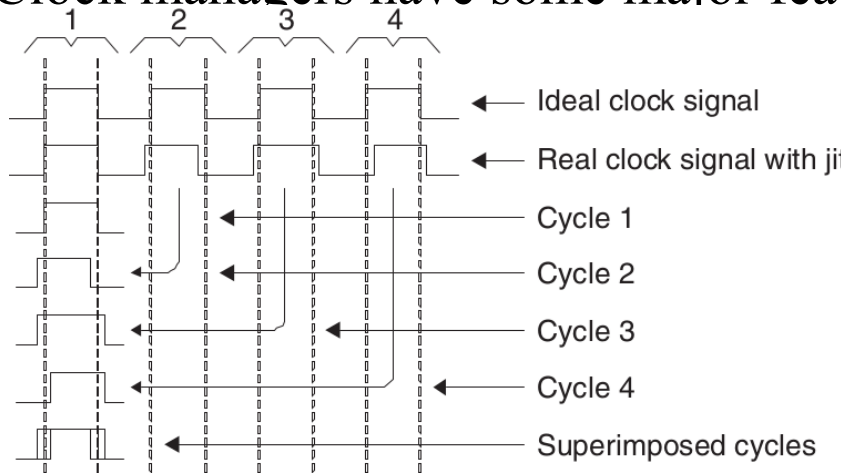
- In reality, *clock managers* are used
- That is, the clock pin drives a special hard-wired function (block) that generates a number of daughter clocks

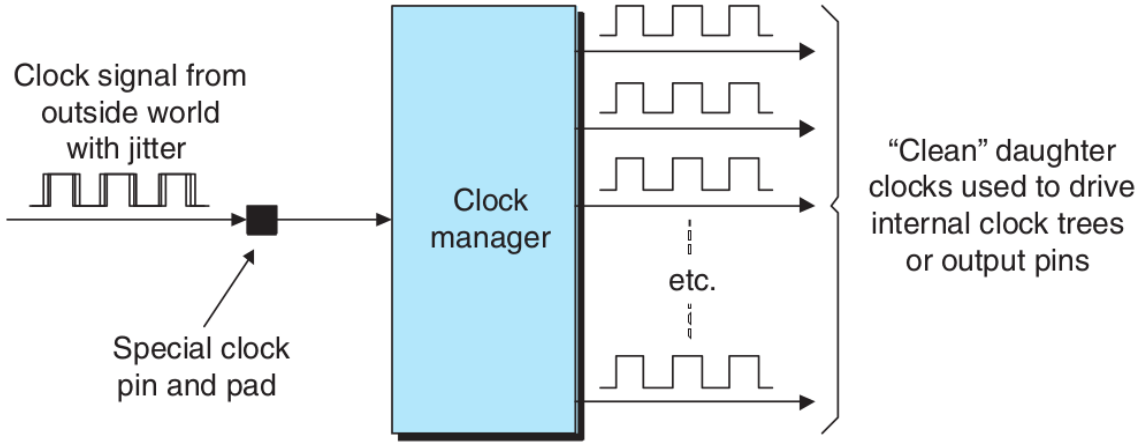


Clock Managers (IV)

•The daughter clocks can be used to drive internal clock trees or external output pins that provide clocking services to other devices

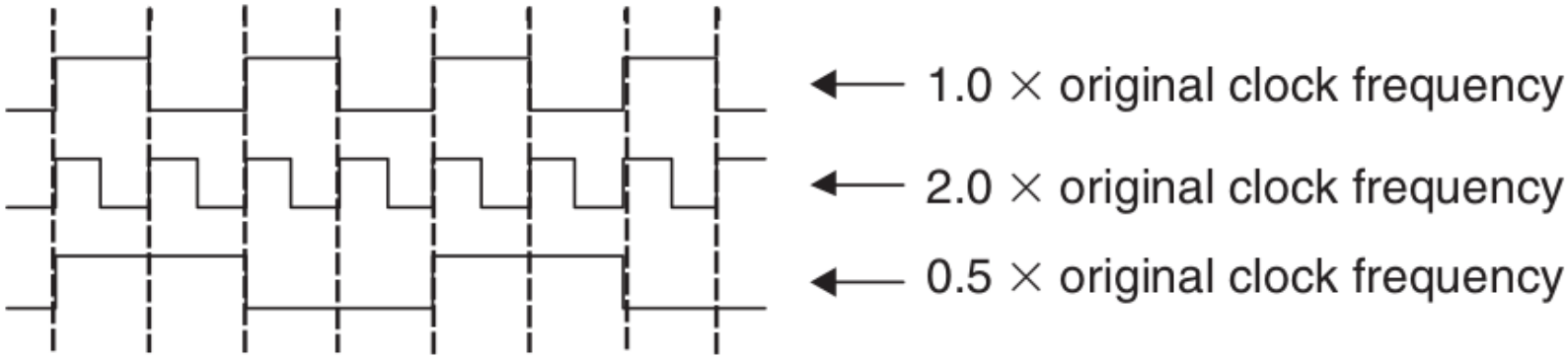
• Clock managers have some maior features such as jitter removal





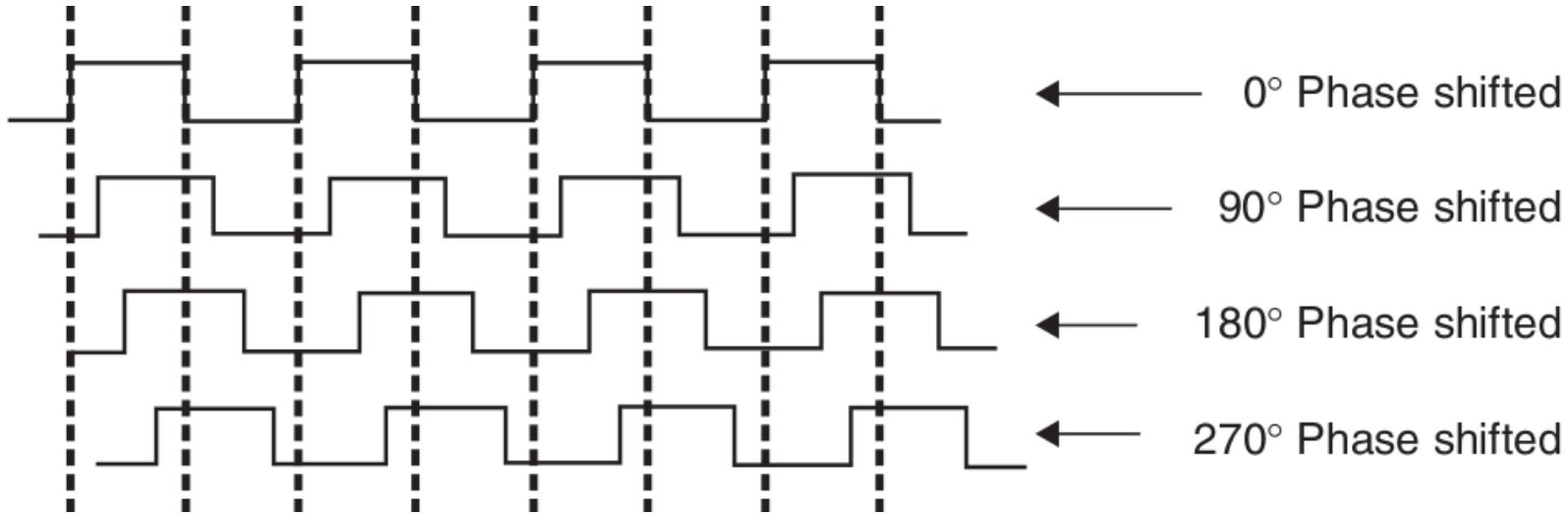
Clock Managers (V)

- A second important feature is frequency synthesis
- Sometimes the frequency provided is not the desired one
- The clock manager can generate daughter clocks at frequencies that are multiples or sub-multiples of the original signal frequency



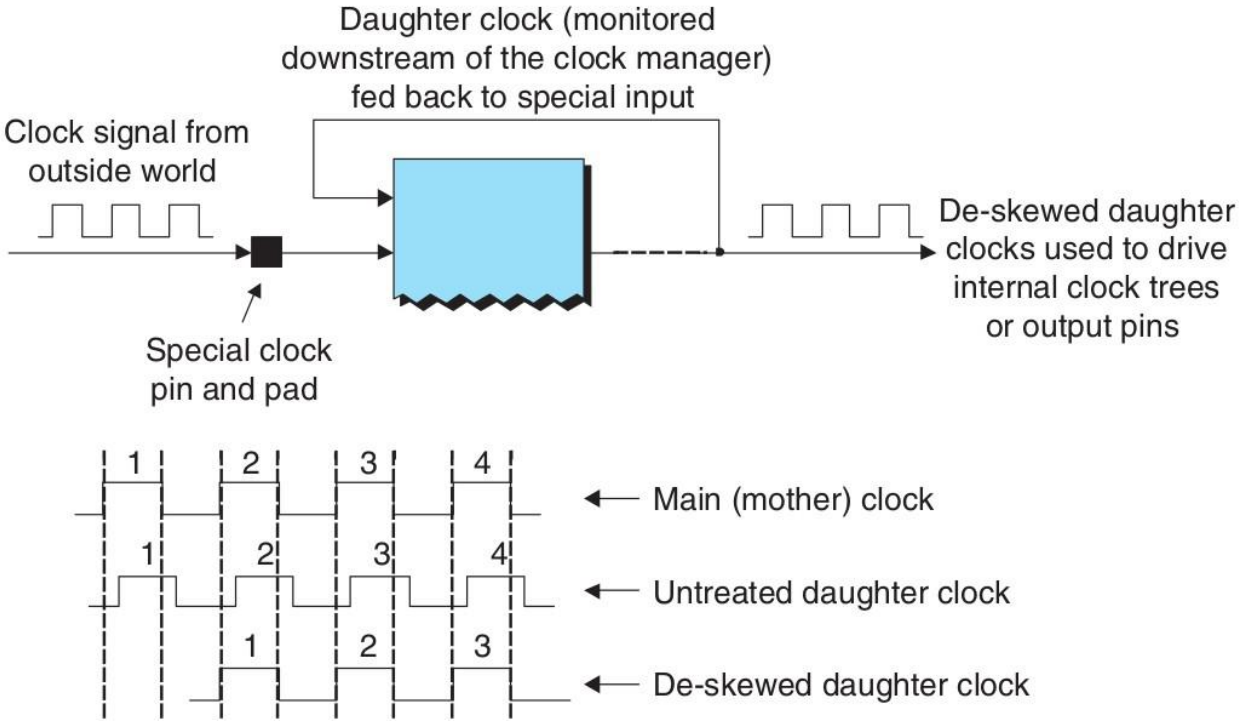
Clock Managers (VI)

- A third important feature is phase shifting
- Designs may also require the use of clocks that are phase shifted with respect to each other
- Some clock managers allow the selection from fixed phase shift values
- Others allow you to configure the exact amount of phase shift



Clock Managers (VII)

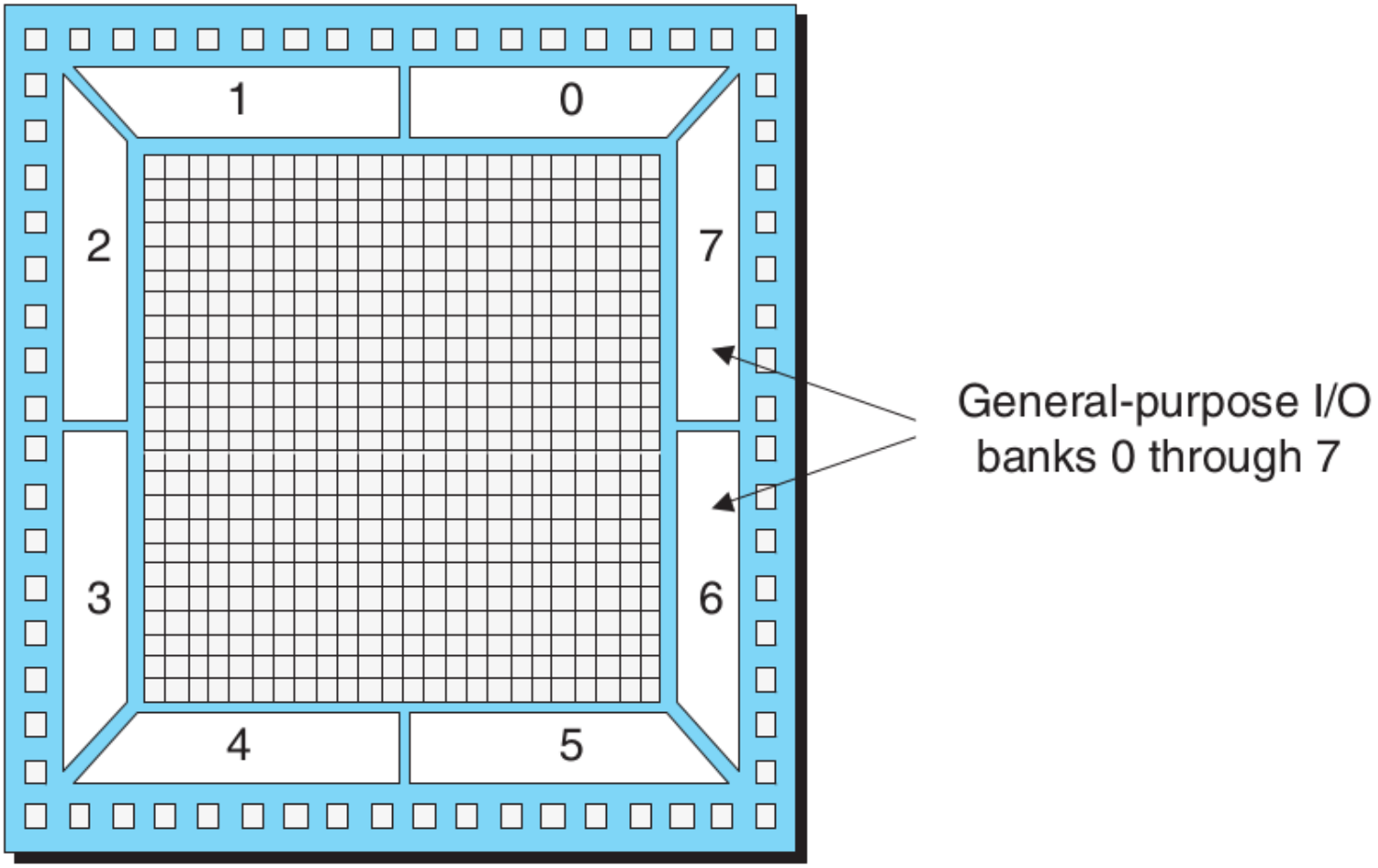
- A fourth important feature is <u>auto-skew correction</u>
- It addresses the added delays to the daughter clock signals due to the clock manager itself, driving gates and interconnect deployed in the signals' distribution
- Hence, the manager adds additional delay to the daughter clock in order to realign it with the main clock



General-Purpose I/O (I)

- Today's FPGA packages can have a thousand or more pins, arranged as an array across the base of the package
- For simplicity, however, we will assume that all connections to the chip are in the form of a ring around the circumference of the device
- FPGAs offer General Purpose I/Os
- This way the I/O can be configured to accept and generate signals conforming to a range of different standards
- This is important since a multitude of different standards exist!
- The general-purpose I/O signals are split into a number of banks
- Each bank can be configured individually for a particular standard
- Hence, the FPGA can work with multiple standards and interface between different standards

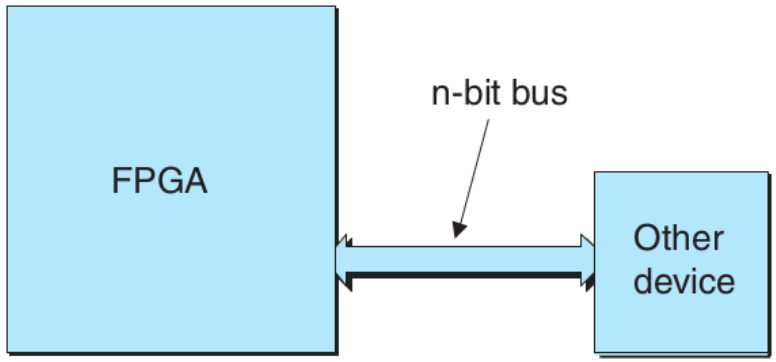
General-Purpose I/O (II)



General-Purpose I/O (III)

•The traditional way to move large amounts of data between devices is to use a bus

•That is a collection of signals that carry related data and perform a common function



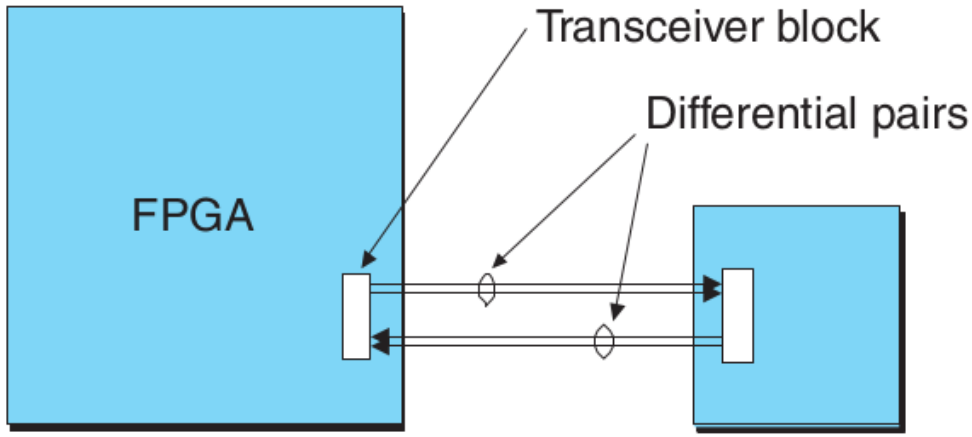
•With the need to push more data around faster, buses grew to 16 bits, then to 32 bits and then to 64 bits

•This requires a lot of pins and routing these tracks becomes overly complicated as well as ensuring signal integrity

General-Purpose I/O (IV)

•Today's high-end FPGAs include special hard-wired gigabit-transceiver blocks for serial high-speed communication

•They use a pair of differential signals to transmit (TX) data and another pair to receive (RX) data



•They operate at incredibly high-speeds and transmit and receive billions of bits of data per second

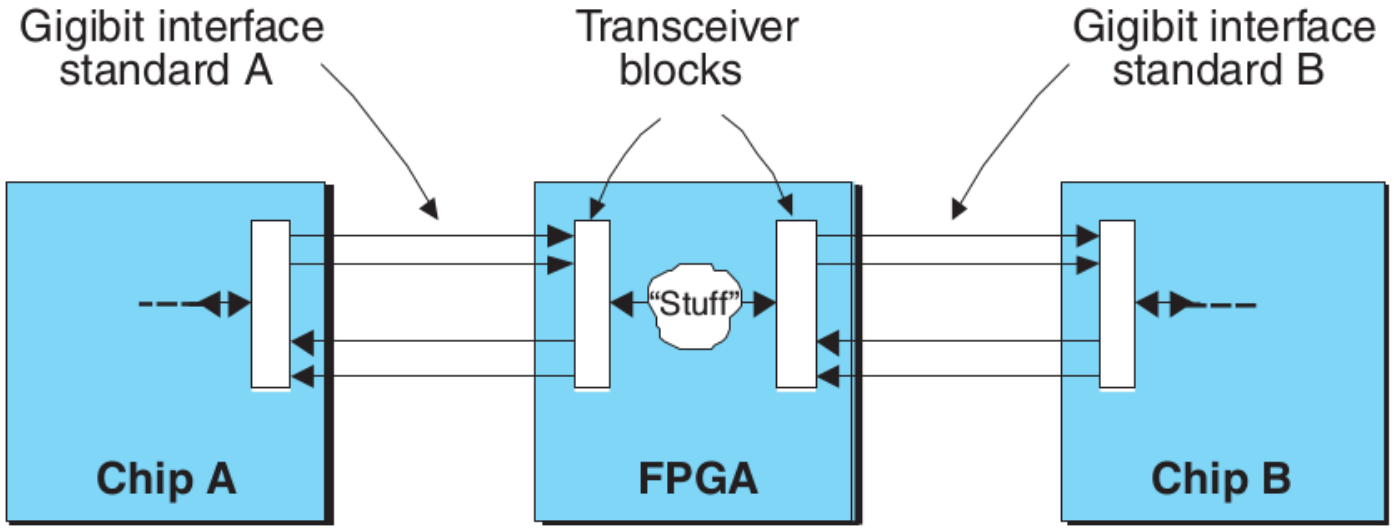
•Each block supports a number of such transceivers and an FPGA contains a number of blocks

General-Purpose I/O (V)

•Multiple standards exist for this technology, such as Fibre Channel, Infiniband, PCI Express, RapidIO, SkyRail and 10-gigabit Ethernet

•Off-the-shelf devices typically support only one gigabit standard, or a small subset

•In this case, an FPGA may be used to act as an interface between two or more standards



Intellectual Property (IP) (I)

- Today's designs are very big and complex
- It is impractical to create every design from scratch
- Re-use existing functional blocks
- They are usually referred to as Intellectual Property (IP)
- The main sources of IP are FPGA vendors and third-party providers
- Each FPGA vendor offers a selection of hard, firm and soft IP
- Hard IP is pre-implemented blocks, such as microprocessor cores, gigabit transceivers, multipliers, adders etc.
- They are designed to be as efficient as possible in terms of power consumption, silicon real estate, and performance
- Soft IPs refer to a source-level library of high-level functions
- These functions are usually represented using a hardware description language

Intellectual Property (IP) (II)

- Soft IPs are incorporated into the main body of the design
- They are synthesised along with the rest of the design into a group of programmable logic blocks
- Firm IP also comes in the form of a library of high-level functions
- However, these functions have already been optimally mapped, placed and routed into a group of programmable logic blocks
- IPs can be **handcrafted** and the end-user can either purchase it as *i)* Blocks of unencrypted source code, *ii)* Encrypted RTL level, *iii)* Unplaced-and-unrouted netlist level, or iv) Placed-and-routed netlist level
- An advantage of IP use is that the provider has tuned the synthesis engine to provide optimal implementations
- However, the designer cannot remove any unwanted functionality
- Also, the IP block is tied to a specific vendor and device family

Intellectual Property (IP) (III)

- Another common practise is for FPGA vendors to provide special tools that act as IP block/core generators
- They allow you to specify the widths and depths of buses and functional elements
- First you select from a list of components and then you parametrise them
- The designer can also include or omit certain functional attributes of the core
- In addition to the different types of IP, the generator may also output a cycle- accurate C/C++ model of your IP for functional simulation

Ευχαριστώ για την προσοχή σας