

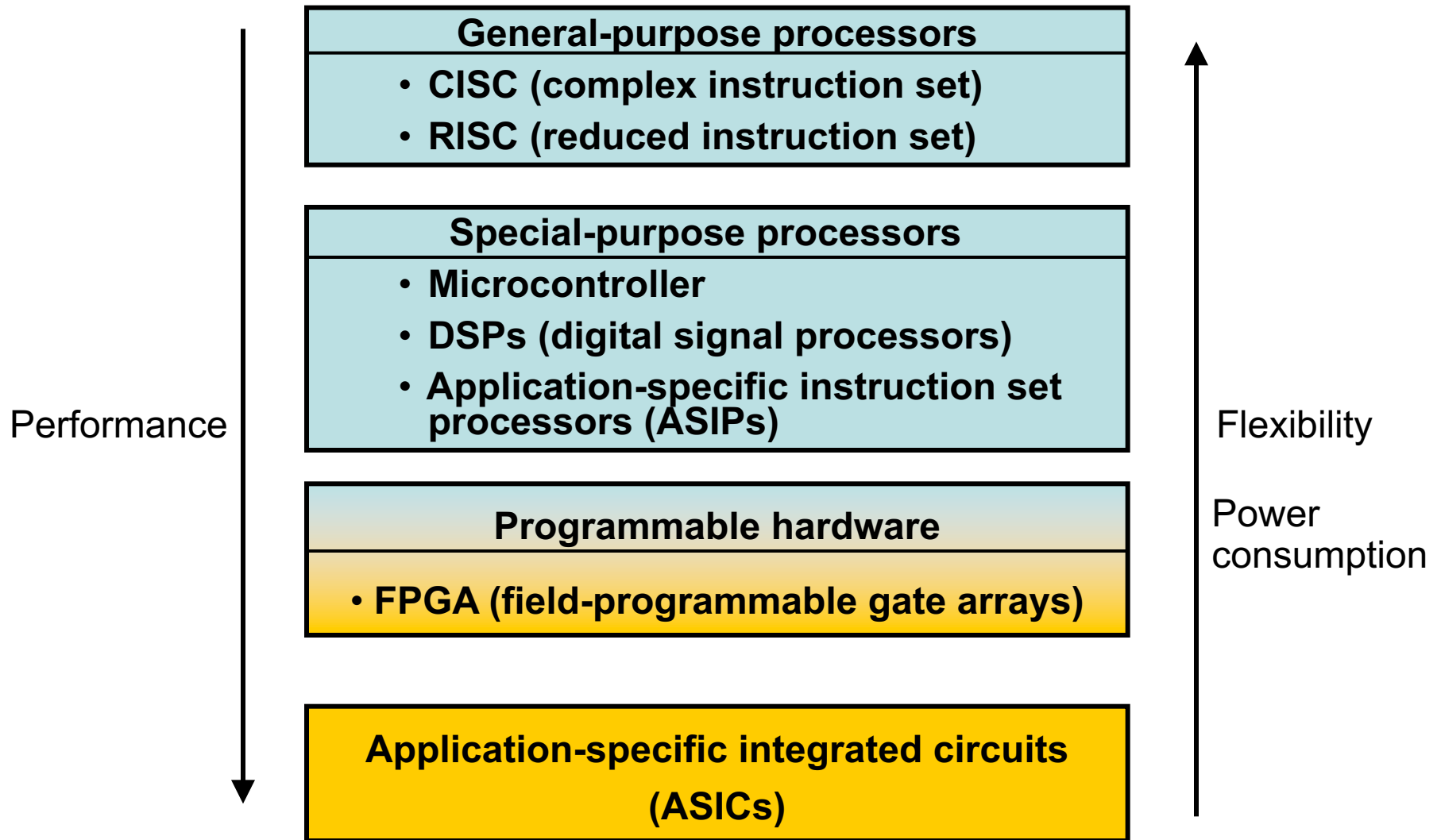
Target Architectures for HW/SW Systems

SS 2017

Prof. Dr. Christian Plessl

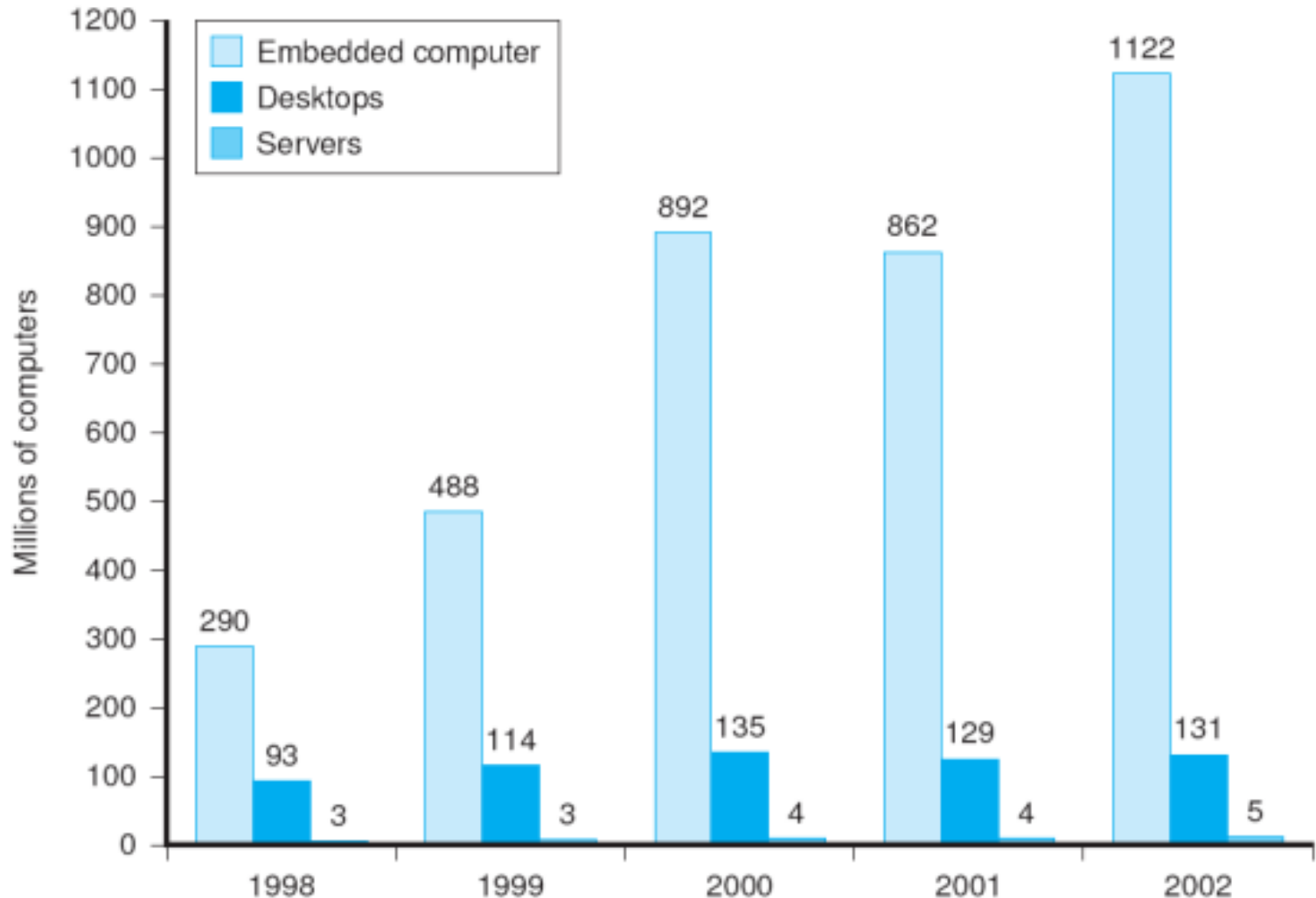
High-Performance IT Systems group
Paderborn University

Implementation Alternatives

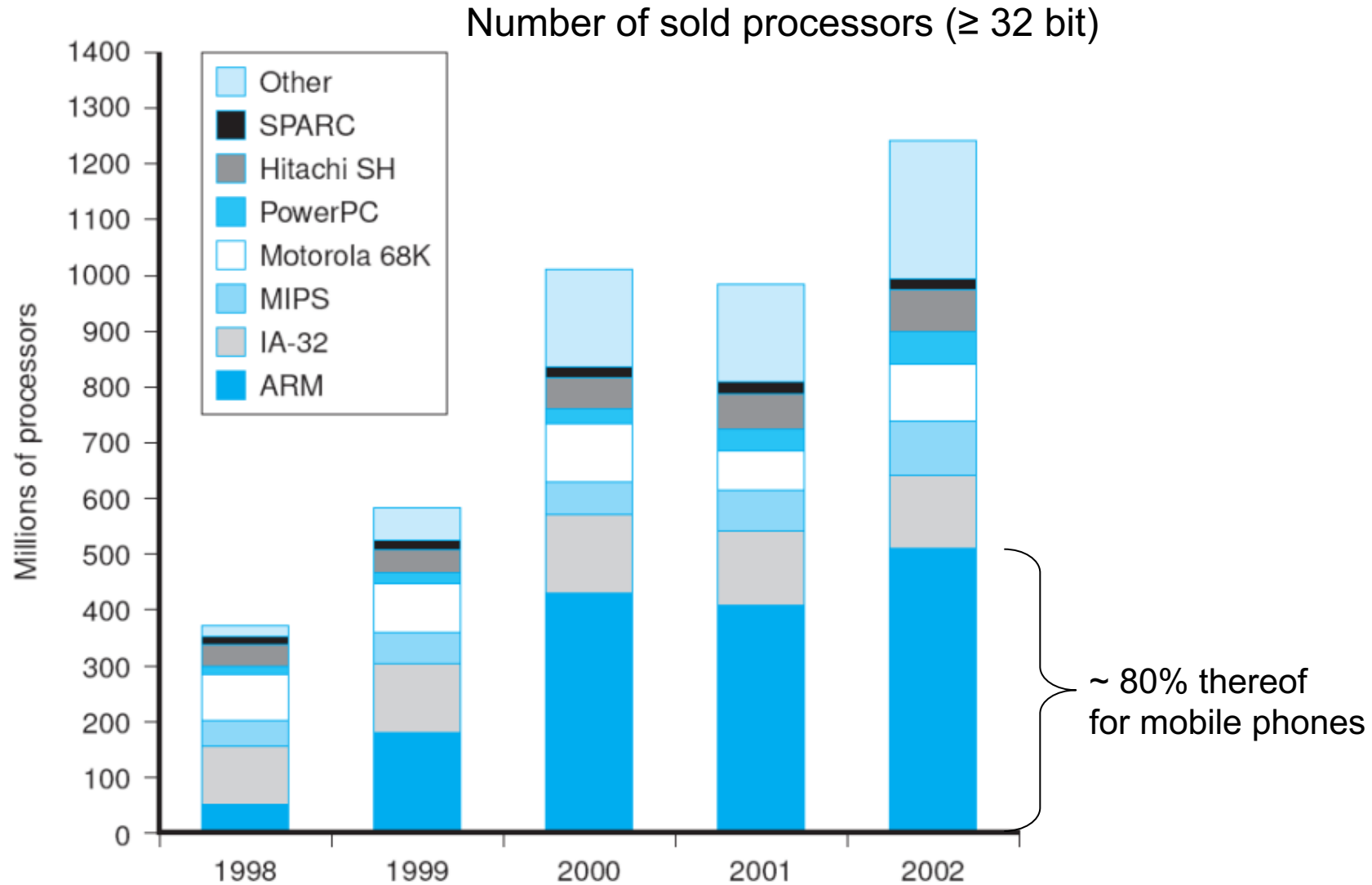


Processor Market

Number of sold processors (embedded, ≥ 32 bit) and systems (desktop, server)



Instruction Set Architectures

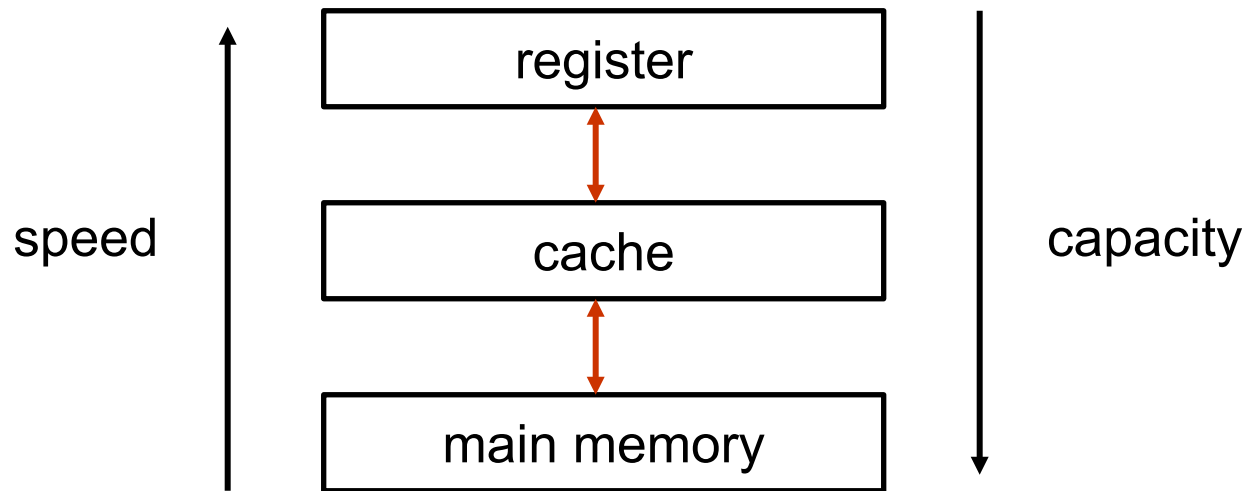


General-Purpose Processors

- characteristics
 - high performance for large application mix, not optimized for a single application
 - high power consumption
- development
 - highly optimized circuit structures
 - design time >100 person years
 - profitable in large volumes only
- application domains
 - PCs, workstations, game consoles, servers, supercomputers

How General Purpose CPUs achieve High Performance

- exploitation of parallelism
- several layers of memory hierarchy



- use of leading semiconductor technology
 - gate count, clock rate

Exploitation of Parallelism

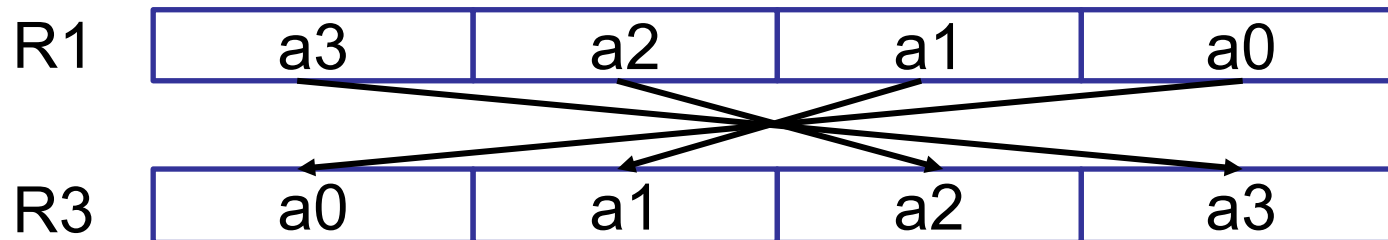
- **bit level**
 - wider data paths, e.g. 8 bit → 16 bit → 32 bit → 64 bit → ...
- **word level**
 - SIMD/vector set extensions, e.g. IA-32/x86-64 MMX/SSE/AVX
32/64-bit registers and ALUs split into smaller units
instructions work in parallel on these units
- **instruction level**
 - pipelining
 - multiple issue, e.g. superscalar processors, VLIW processors
- **thread level**
 - multithreaded processors
- **thread/process level**
 - multicore processors
 - multiprocessor computer (SMT)

SIMD Instructions

ADD R1, R2 → R3

R1	a3	a2	a1	a0
	+	+	+	+
R2	b3	b2	b1	b0
	=	=	=	=
R3	a3+b3	a2+b2	a1+b1	a0+b0

PERMUTE R1 (pattern 0 1 2 3) → R3



Processor Performance

$$T_{exe} = I_c \times CPI \times T$$

T_{exe} execution time for a program

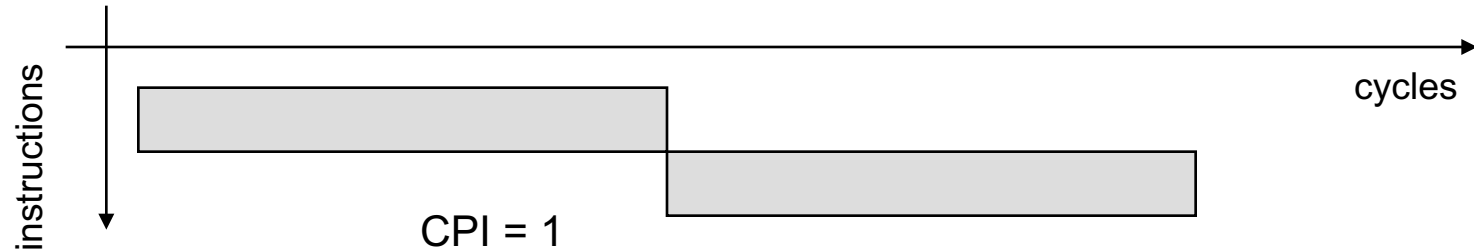
I_c instruction count, number of instructions to execute for a program run

CPI cycles per instruction, average number of clock cycles per instruction

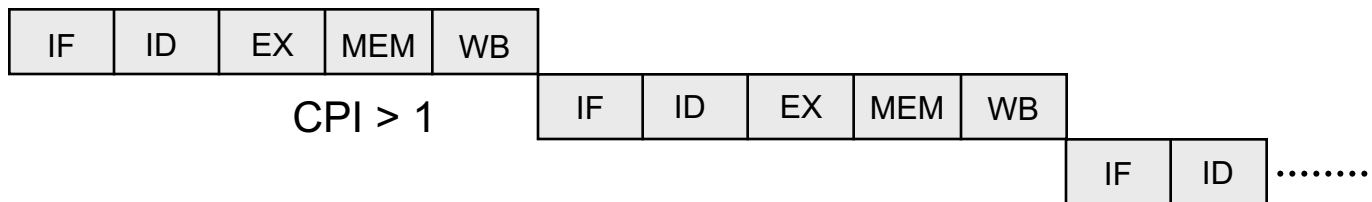
T clock period

Processor Implementations

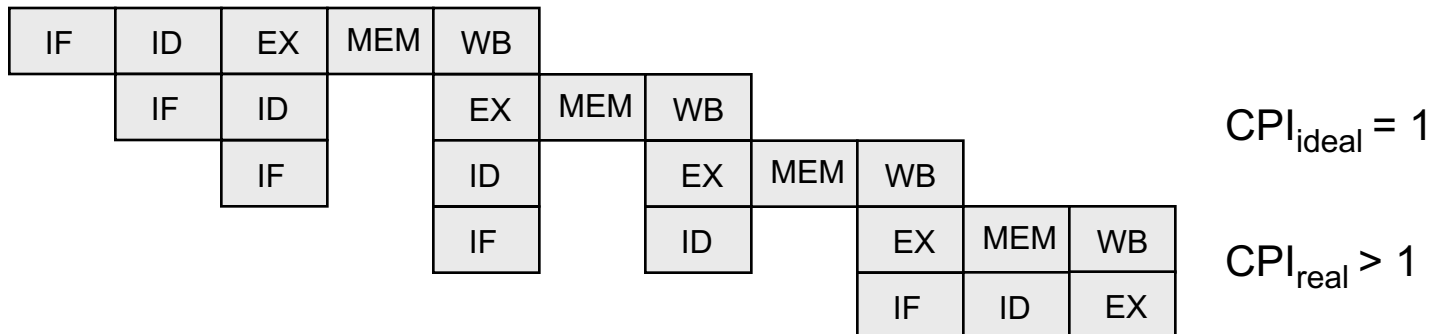
- single cycle implementation



- multi-cycle implementation

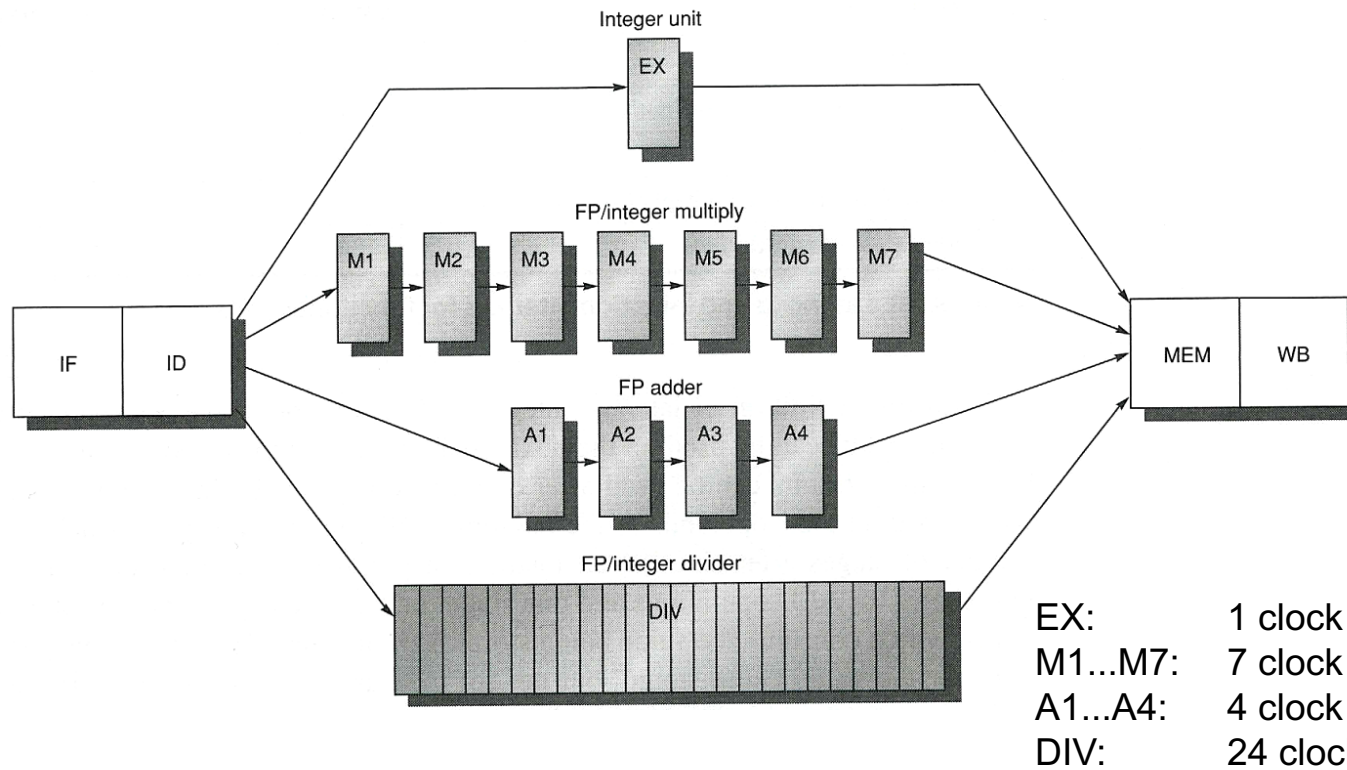


- pipelining implementation



Deep Pipelining

- operations in the EX phase often take much longer than one clock cycle → pipeline performance drops
 - use several execution units with different latencies
 - the execution units can also be pipelined

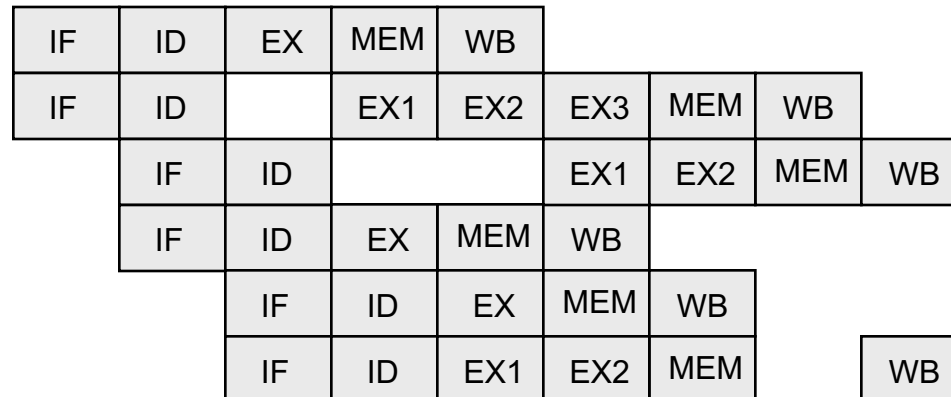


- in-order execution
 - instructions are assigned to the execution units in the original program order (in-order issue, in-order execution)
 - the results of the instructions may be out-of-order → resolve conflicts
- reduce conflicts by reordering of instructions
 - static pipeline scheduling
 - by the software (compiler) at compilation time
 - compiler has to know characteristics of the actual processor implementation
 - code is efficient only for the targeted processor implementation
 - dynamic pipeline scheduling (out-of-order execution)
 - by the hardware (processor) at runtime
 - huge hardware effort required, limited algorithms for instruction reordering
 - simplifies compiler design
 - code runs efficiently on every implementation of the instruction set architecture (!)
 - modern systems use a combination of static/dynamic pipeline scheduling

Multiple Issue Processors (1)

- pipeline scheduling tries to bring the real CPI close to the ideal CPI
 - although at any given time many instructions are in the execution units, only one new instruction is started per clock cycle $\rightarrow \text{CPI}_{\text{ideal}} = 1$
- multiple issue processors start (issue) more than one instruction per clock cycle
 - m -issue $\rightarrow \text{CPI}_{\text{ideal}} = 1/m$
 - the processor has to fetch m instructions per cycle from the cache $\rightarrow$ the cache bandwidth must be increased

2-issue:

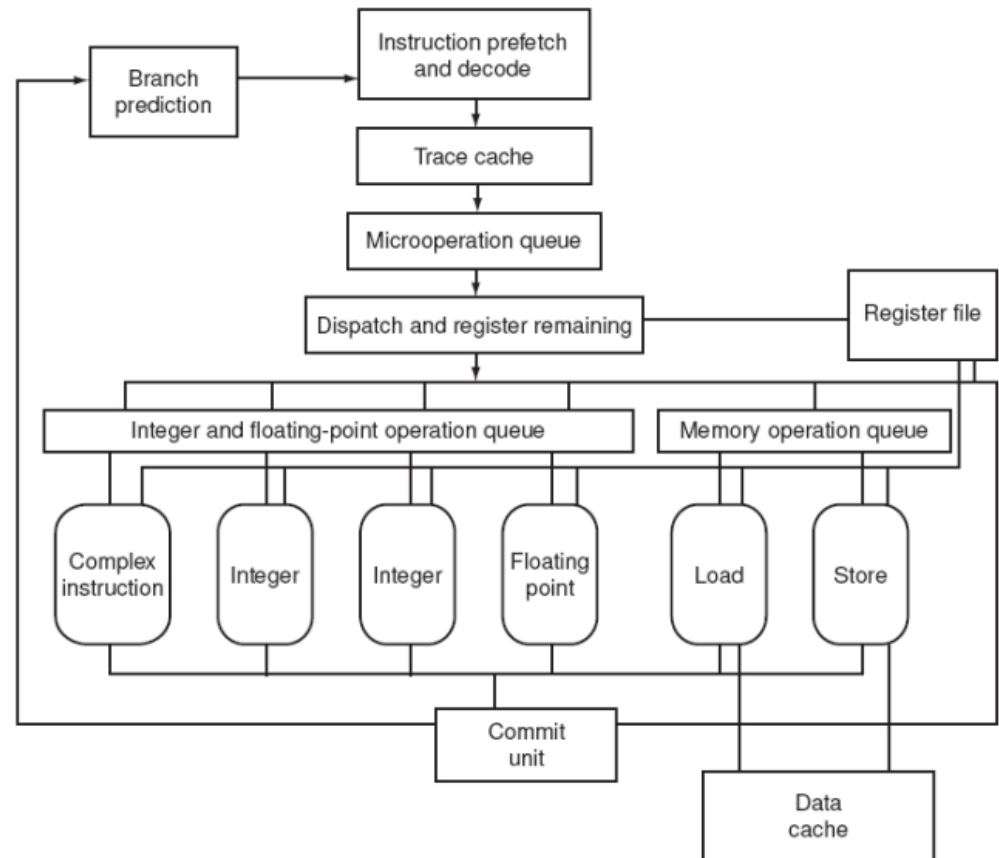


Multiple Issue Processors (2)

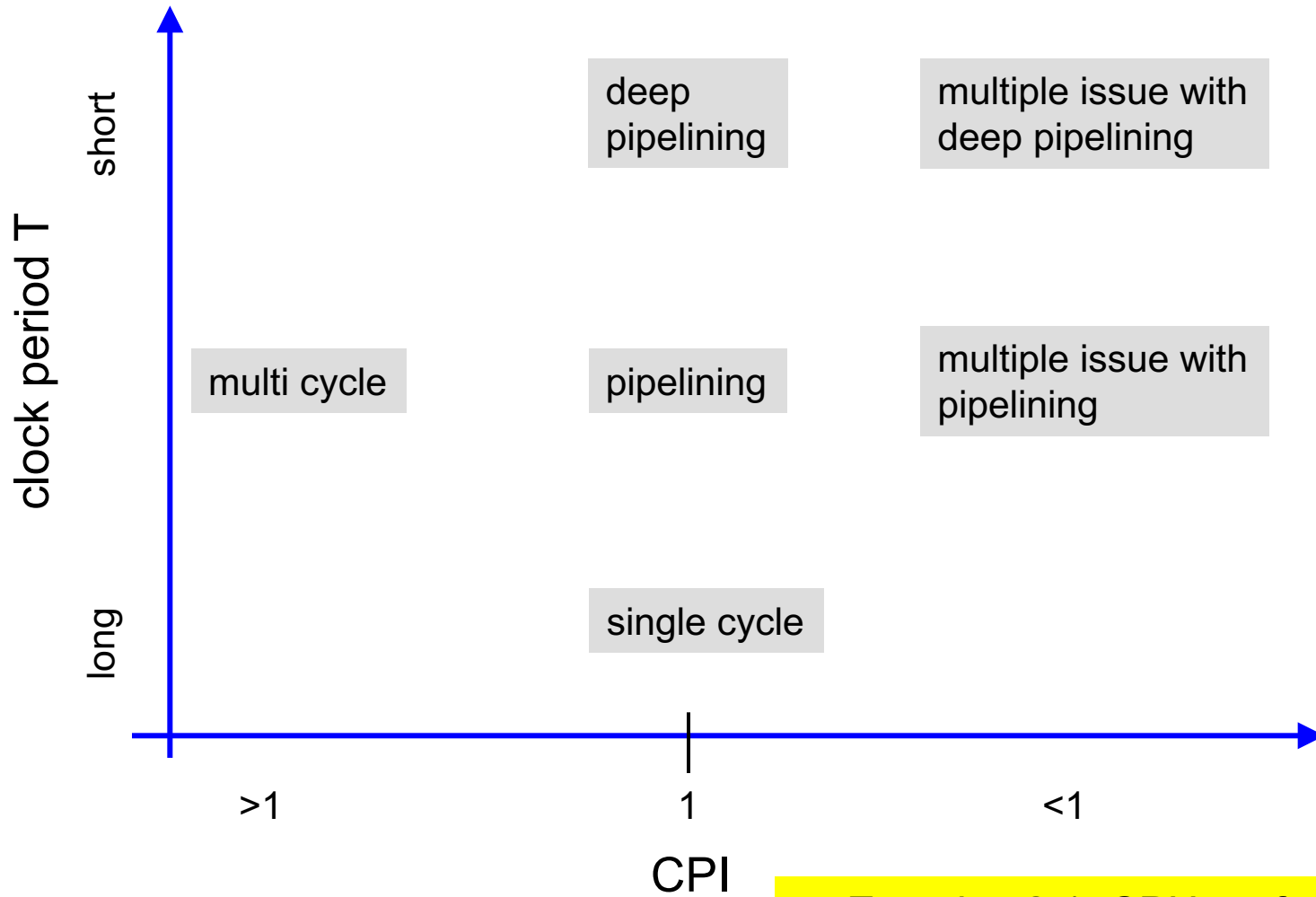
- static multiple issue (VLIW processors)
 - compiler assigns instructions to execution units
 - the single instructions are grouped into issue packets
 - originally denoted as very long instruction word (VLIW)
 - simplifies processor design, complicates compiler design
 - code is not binary compatible to other processors of the same instruction set family (!)
 - Example: Intel IA-64 (Itanium, Itanium 2), high-end DSPs
- dynamic multiple issue (superscalar processors)
 - processor assigns instructions to execution units
 - complicates processor design, simplifies compiler design
 - code is binary compatible to other processors of the same instruction set family (!)
 - Example: Intel IA-32 (Pentium 4)

Example: Superscalar Processor

- IA-32 Pentium 4
 - CISC instructions are translated to RISC-like micro operations
 - 7 execution units, deep pipeline (20 cycles on average)
 - multiple issue with 3 micro-operations per clock cycle
 - in a single clock cycle, up to 126 micro operations can be "on-the-fly"



Design Space: (CPI x T)

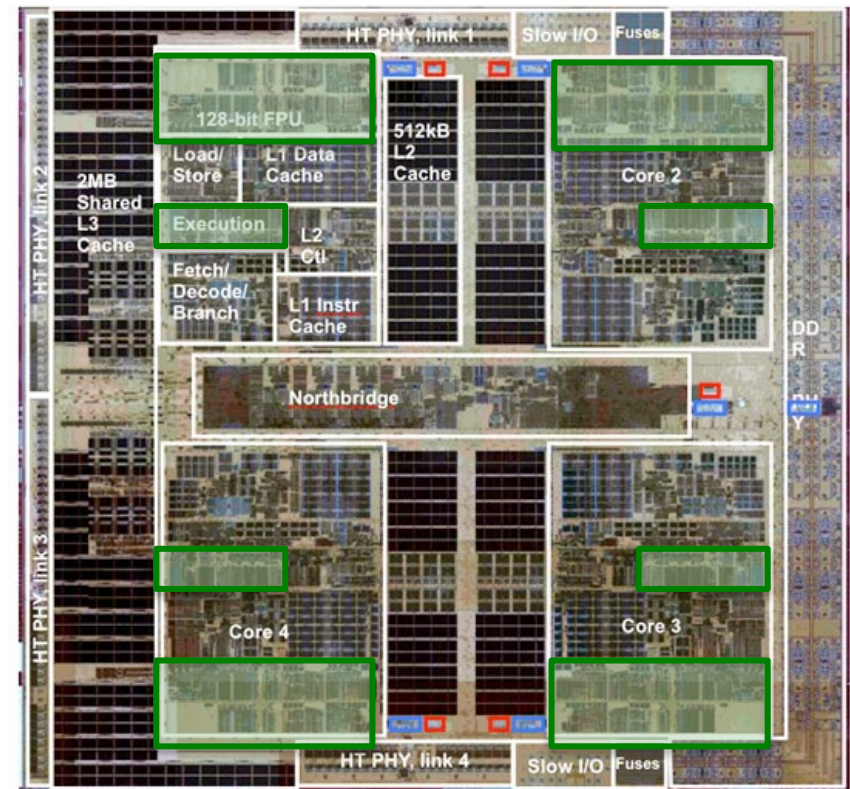


 Exercise 2.1: CPU performance

Disadvantages of General-Purpose CPUs

- general-purpose CPUs make no assumptions about
 - kind of applications
 - parallelism of operations
 - memory access patterns
- compensating this lack of knowledge using
 - generic instructions
 - complex execution controllers
 - large caches
- result
 - low energy efficiency
 - large and complex chips
 - majority of chip area does not contribute to computing

die shot of a 4-core
AMD Barcelona CPU



chip area that contributes to
actual computation

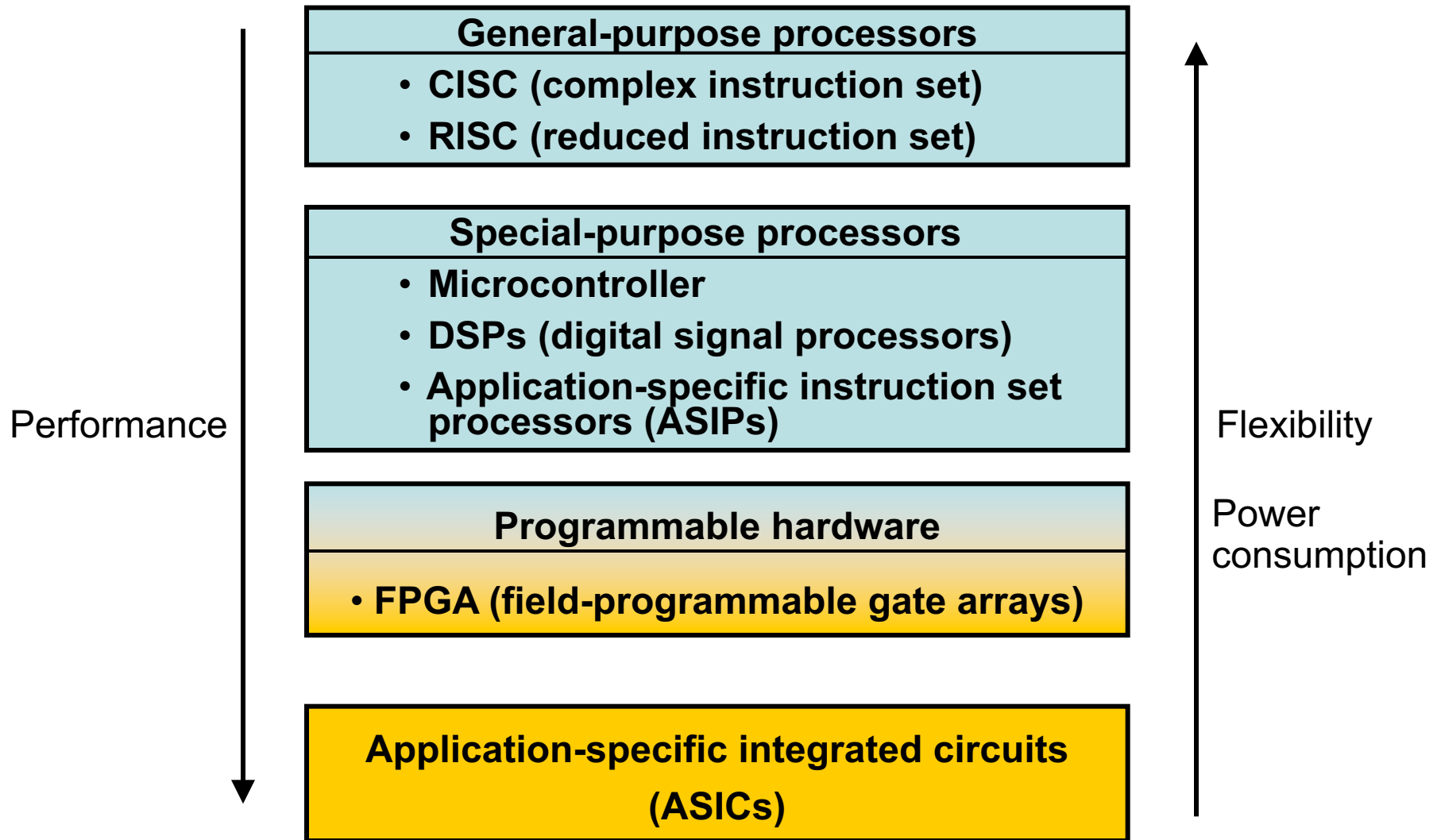
General-Purpose Processors for Embedded Systems

- execution times difficult to predict due to
 - dynamic scheduling
 - caching
 - branch prediction

➡ general-purpose processors are not suitable for hard real-time

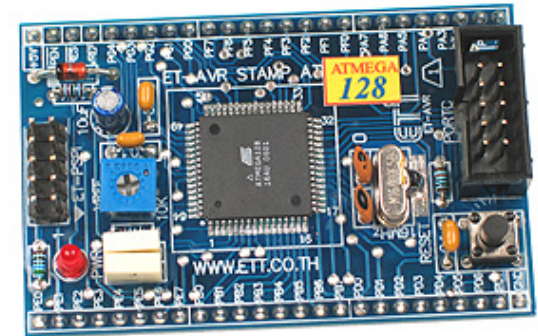
- high power consumption
- complex I/O- and memory interfaces
- very expensive

Implementation Alternatives



- control-dominant embedded computing
 - control-dominant code (many branches and jumps)
 - few arithmetic operations, low data throughput
 - multitasking
- application/market requirements
 - energy efficiency (sleep modes)
 - low-cost
- microcontrollers are optimized for that
 - simple processor architecture
 - fast context switches and interrupt processing (e.g. shadow registers)
 - efficient bit- and logic operations
 - integrated peripheral units (analog/digital and digital/analog converters, USB, CAN, timer, ...)
 - integrated memory (SRAM, Flash EEPROM, EEPROM, etc.)

- 8-bit low-cost RISC microcontroller
 - 32 general purpose registers
 - instructions execute in 1-4 cycles
- included volatile and non-volatile memory
 - 128kB Flash, 4kB EEPROM, 4kB SRAM
- integrated peripherals
 - 4 timers/counters
 - 6 pulse-width modulation channels
 - 24 parallel I/O ports
 - 8-channel 10bit analog/digital converter, programmable gain
 - serial RS232 interface + two-wire interface
- part of a large family of devices
 - different combination of memory size and peripherals (USB, CAN, LCD)



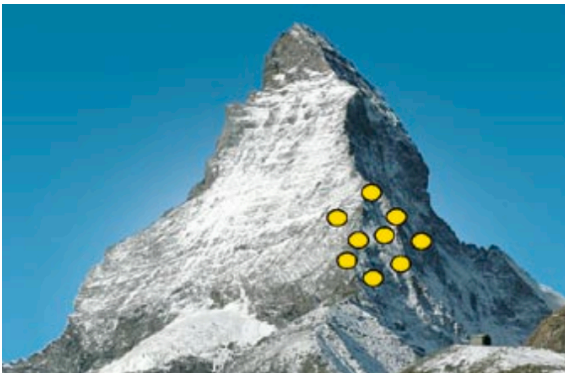
Texas Instruments MSP430F2619

- 16-bit ultra-low power microcontroller
 - low power modes: active $\sim 365\mu\text{A}$, standby $\sim 0.6\mu\text{A}$
 - a rechargeable battery of 1.5V 2500mAh could
 - keep the controller running for >9 month (active)
 - retain data in RAM for about 475 years (sleep)
 - wakeup time from standby $\sim 1\mu\text{s}$
 - 16 registers (12 general purpose)
- included volatile and non-volatile memory
 - 96kB Flash, 256B EEPROM, 4kB SRAM
- integrated peripherals
 - 2 timers/counters
 - 1-channel 12bit A/D converter, 2-channel D/A converter
 - 4 multi-standard serial I/O interfaces
 - 48-bit parallel I/O interface



Case Study: Wireless Sensor Networks

- goal: monitoring of permafrost regions (PermaSense project)
 - measurements: temperature, humidity, rock movement
 - extreme conditions: -40° to 60° , rock fall, lightening strokes, ice coating
 - high accuracy measurements
- HW/SW co-design challenge
 - goal run 3 years from a single battery while using wireless communication
 - dividing functionality between specialized hardware and uC (TI MSP430)
- approaches
 - careful low-power HW design covering all components (microcontroller, AD converters, power regulators, ...)
 - aggressive power management techniques in SW, system 99.9% asleep

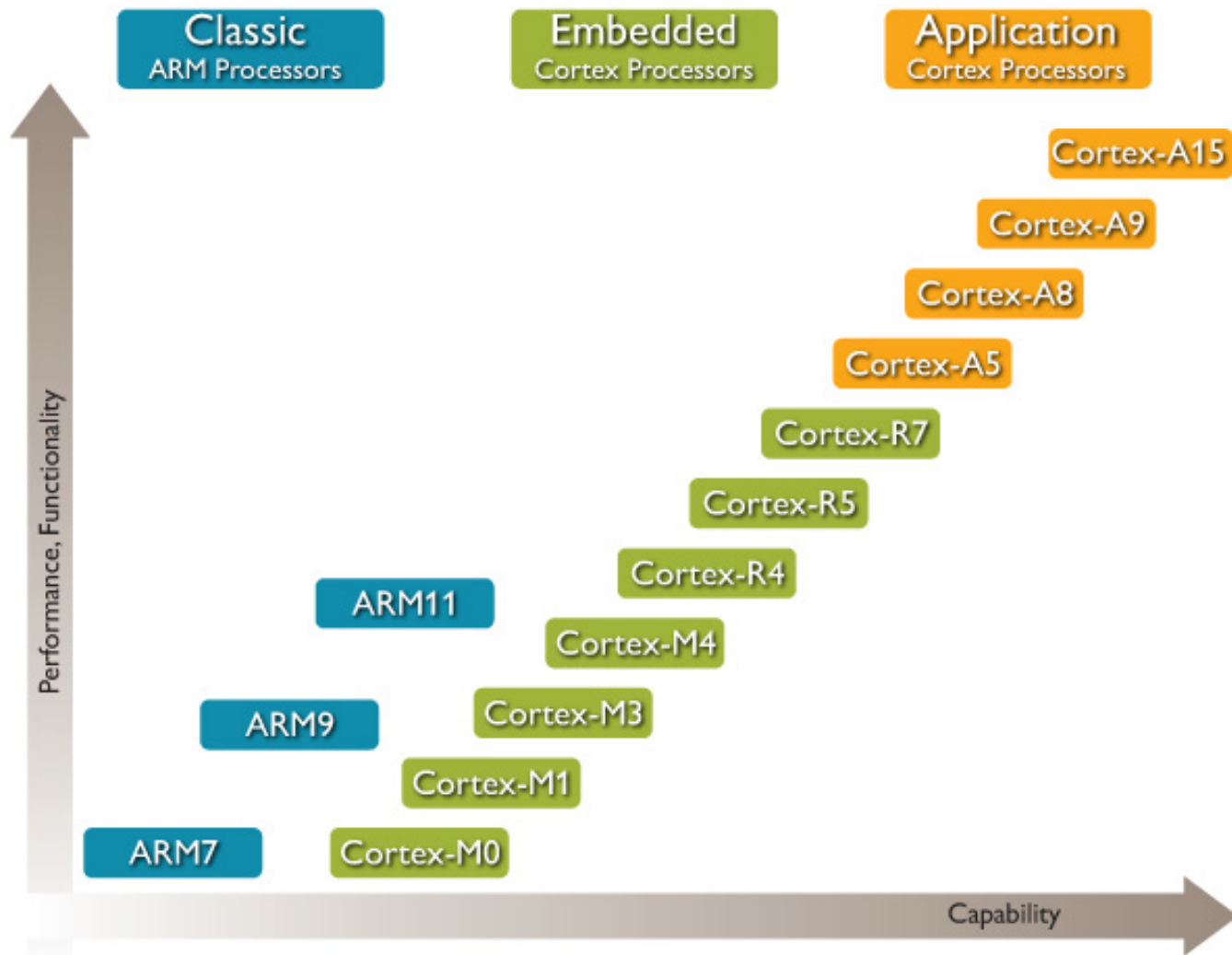


- used as building block in vendor-specific system-on-chip
 - wide range of processors based on a common architecture
 - good balance between performance/power consumption/chip area
 - ARM does license core to semiconductor companies (ARM itself is fabless)
 - huge ecosystem (cores, compilers, tools, libraries, debuggers, ...)
- RISC design principles of basic ARM architecture
 - 32 uniform registers / load-store RISC architecture / simple addressing modes
- key features to improve code density
 - instructions that combine a shift with an ALU operation
 - auto-increment/decrement addressing modes to improve loops
 - load/store-multiple operations to maximize data throughput
 - predicated (conditional) execution of almost all instructions to maximize execution throughput

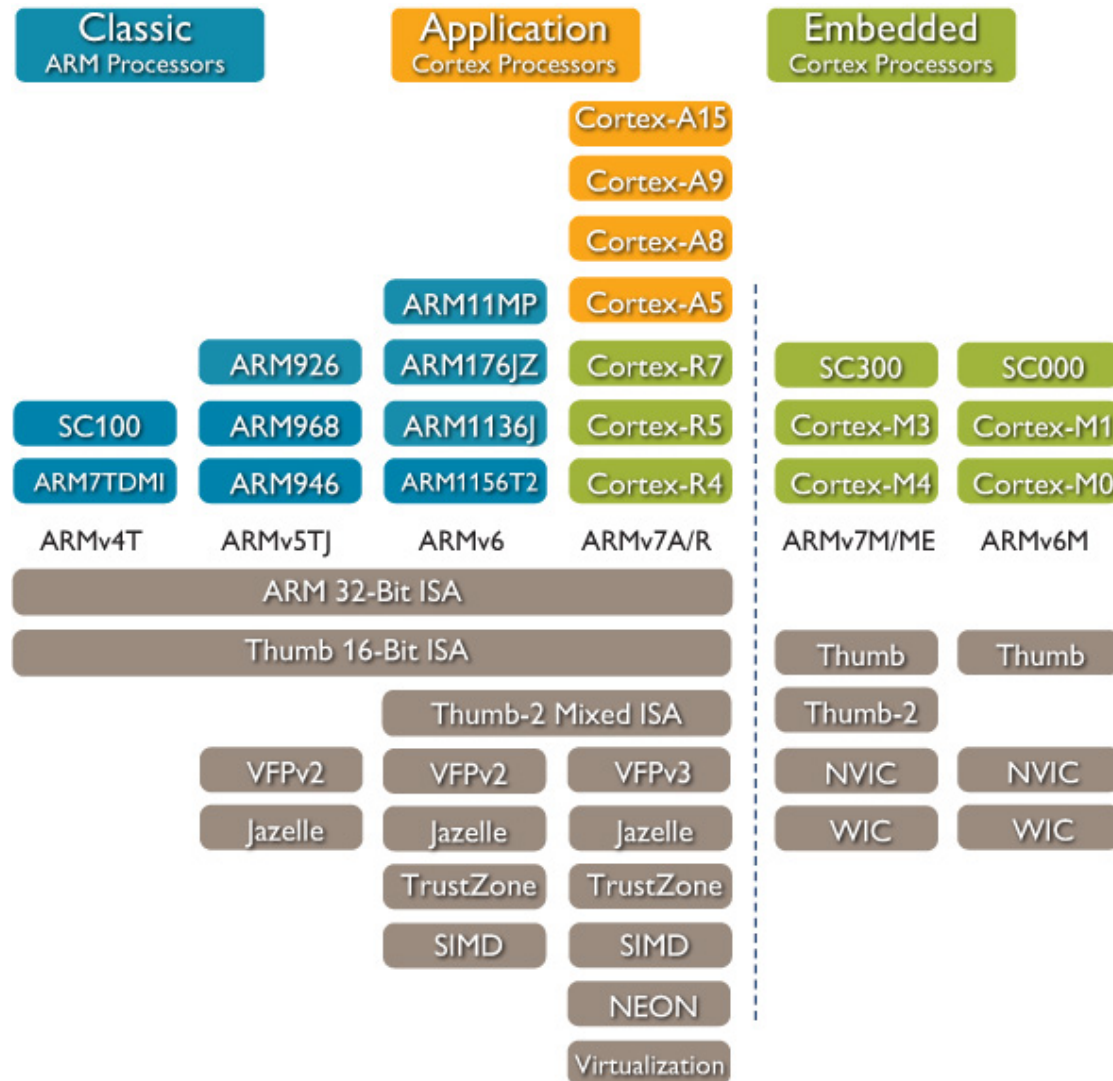
 Excursus: Predicated Execution

- various extensions
 - Thumb: alternative 16bit instruction encoding for most instructions to improve code density
 - Jazelle: support for virtual machines (JVM/.NET)
 - NEON: SIMD instruction set
- ARM Cortex family, defines 3 profiles (subfamilies) based on same basic ARMv7 architecture
 - **Cortex M** (microcontroller profile): fast interrupt processing, no virtual memory, no caches, minimum chip area, low-cost
 - **Cortex R** (real-time profile): protected memory, for real-time control systems in industrial automation, automotive, storage devices, ...
 - **Cortex A** (application profile): application profile, virtual memory, SIMD units, floating-point, for high-end consumer electronic devices, networking devices, cell phones, tablets, ...

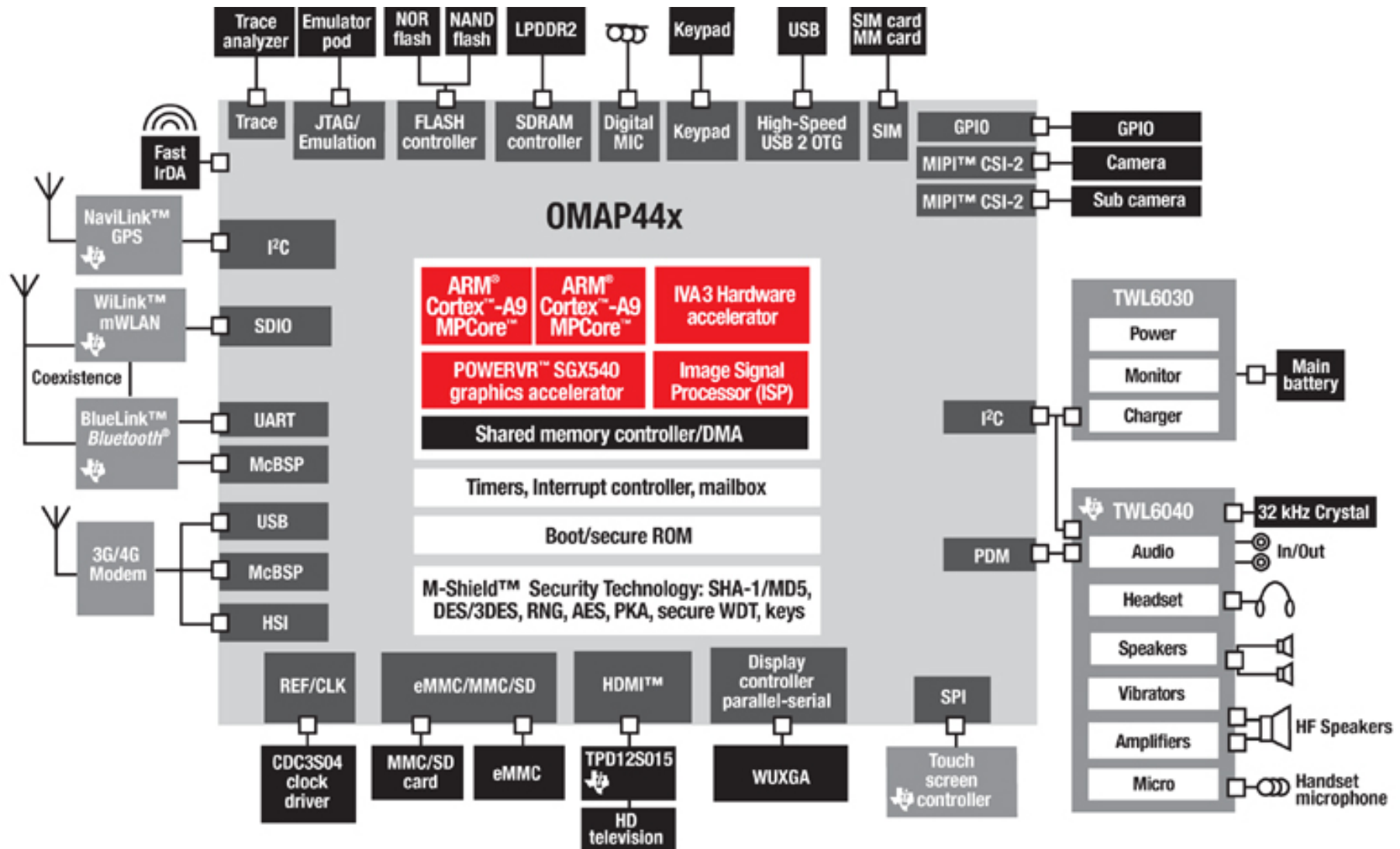
ARM Cortex – Roadmap



ARM Cortex – Features



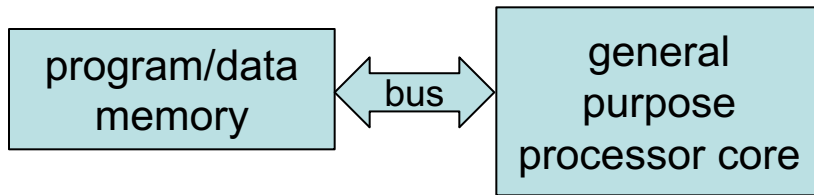
Example: ARM Cortex-based System-on-Chip



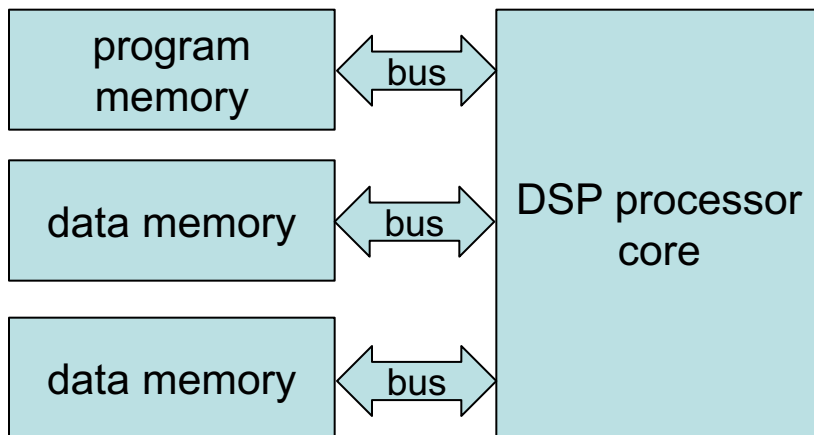
Digital Signal Processors (DSP)

- signal processing applications
 - dataflow-dominant code, high data throughput
 - regular arithmetic operations, few branches and jumps
- DSPs support these characteristics through
 - explicit parallelism
 - Harvard architecture for concurrent data access
 - concurrent operations on data and addresses
 - optimized control flow and background processing
 - zero-overhead loops
 - DMA controllers
 - special addressing modes
 - distinction of address, data and modifier registers
 - versatile address computation for indirect addressing
 - specialized instructions
 - single-cycle hardware multiplier
 - multiply accumulate (MAC) instruction, also known as fused-multiply-add (FMADD)
 - single instruction to multiply two operands and add results to third operand

Harvard Architecture



- unified external memory for program and data
- all operands in registers



- separate program and data memories
- operands also in memory
- concurrent access to
 - instruction word
 - one or several data words
- example:

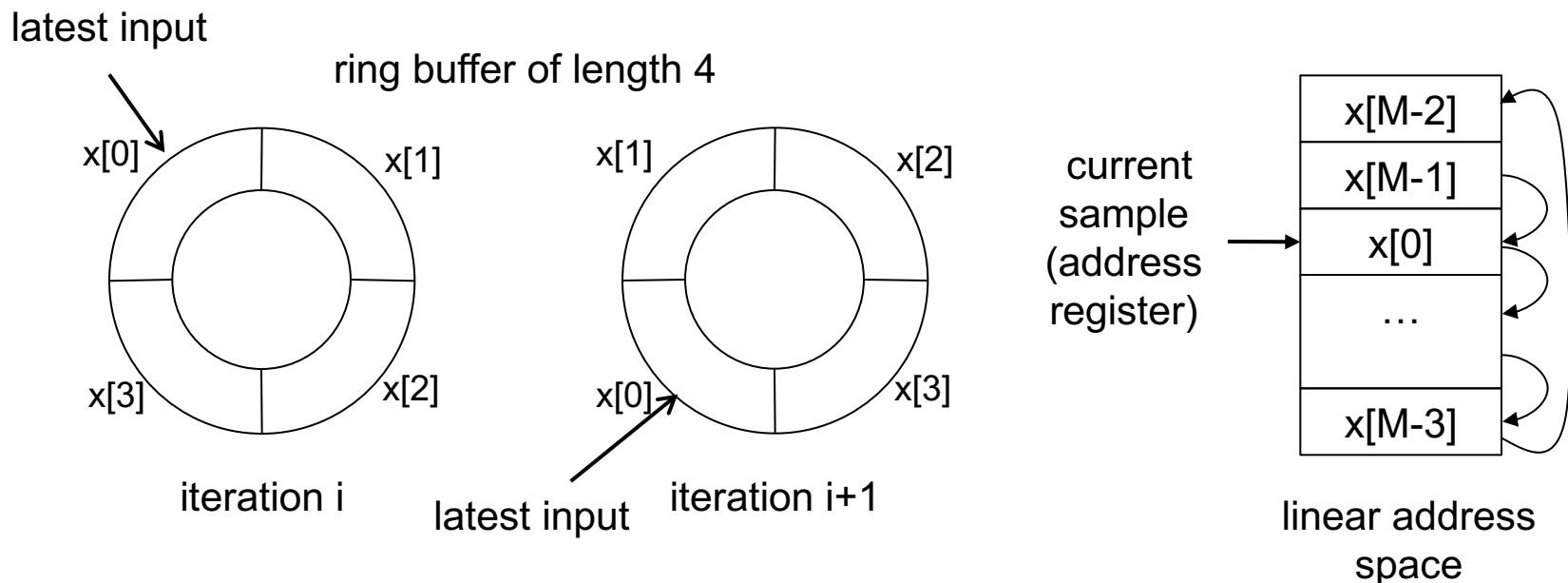
MPYF3	* (AR0) ++,	* (AR1) ++,	R0
↓	↓	↓	↓
instruction from program memory	from data memory (address in address register AR0)	from data memory (address in address register AR1)	store result in data register R0

Specialized Addressing Modes

- many DSPs distinguish address registers from data registers
- additional ALUs for address computations
 - useful for indirect addressing (register points to operand in memory)
`ADDF3 *(AR0)++, R1, R1`
 - operations on address registers **in parallel** with operations on data registers, **no extra cycles**
 - behavior depends on instruction and contents of special purpose registers (modifier registers)
- typical address update functions
 - increment/decrement by 1 (`AR0++`, `AR0--`)
 - increment/decrement by constant specified in modifier register (`AR0 += MR0`, `AR0 -= MR5`)
 - circular addressing (`AR0 += 1` if `AR0 < upper limit`, else `AR0 = base address`), see example

Circular Addressing

- goal: implementation of ring buffers in linear address space
 - implementation variants
 - copy data with data access, or
 - use circular addressing (don't copy data, wrap pointers)
 - supported by addressing modes
 - data access and move operations
 - increment operators that wrap around at buffer boundaries



Zero-overhead Loops

- goal
 - reduce overhead for executing loops
 - general purpose processors
 - initialize loop counter
 - execute loop body
 - check loop exit condition
 - branch to loop start or exit loop
 - digital signal processors
 - initialize loop counter
 - execute loop body
 - ~~▪ check loop exit condition~~
 - ~~▪ branch to loop start or exit loop~~

example: add first 100 values in array a and store result in R1

TMS320C3x-like assembler

```
LDI    @a, AR0
LDI    0.0, R1
RPTS  99
ADDF3  *(AR0)++, R1, R1
...
```

RPTS N repeats next instruction N+1 times

Putting it Together: Scalar Product

```
sum = 0.0;
for (i=0; i<N; i++)
    sum = sum + a[i]*b[i];
```

TMS320C3x assembler

```
LDI    @a, AR0
LDI    @b, AR1
LDF    0, R0
LDF    0, R1
RPTS   N-1
MPYF3  *(AR0)++, *(AR1)++, R0
|| ADDF3 R0, R1, R1
ADDF3  R0, R1, R1
```

zero-overhead loop

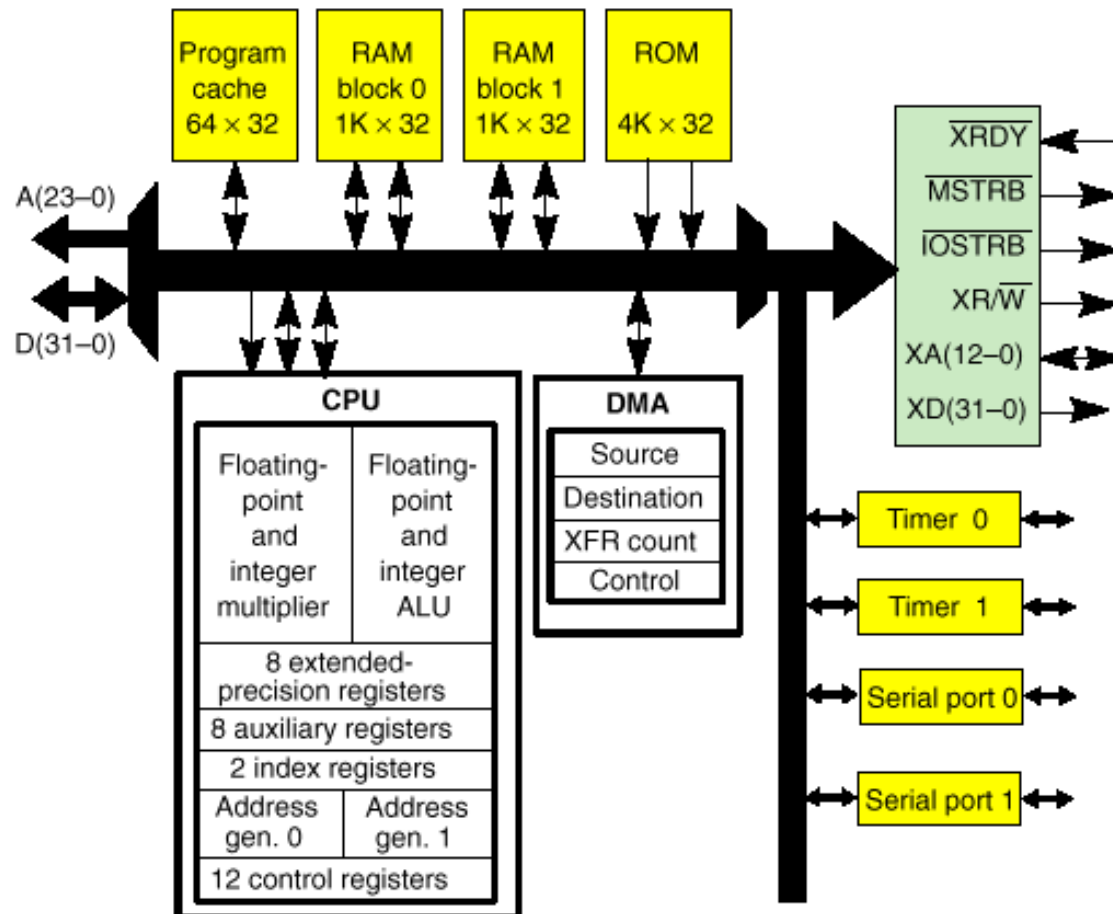
exploit harvard
architecture, read two data
operands in one cycle

MAC - instruction

address arithmetic (auto increment)

Example: TMS320C3x DSP Block Diagram

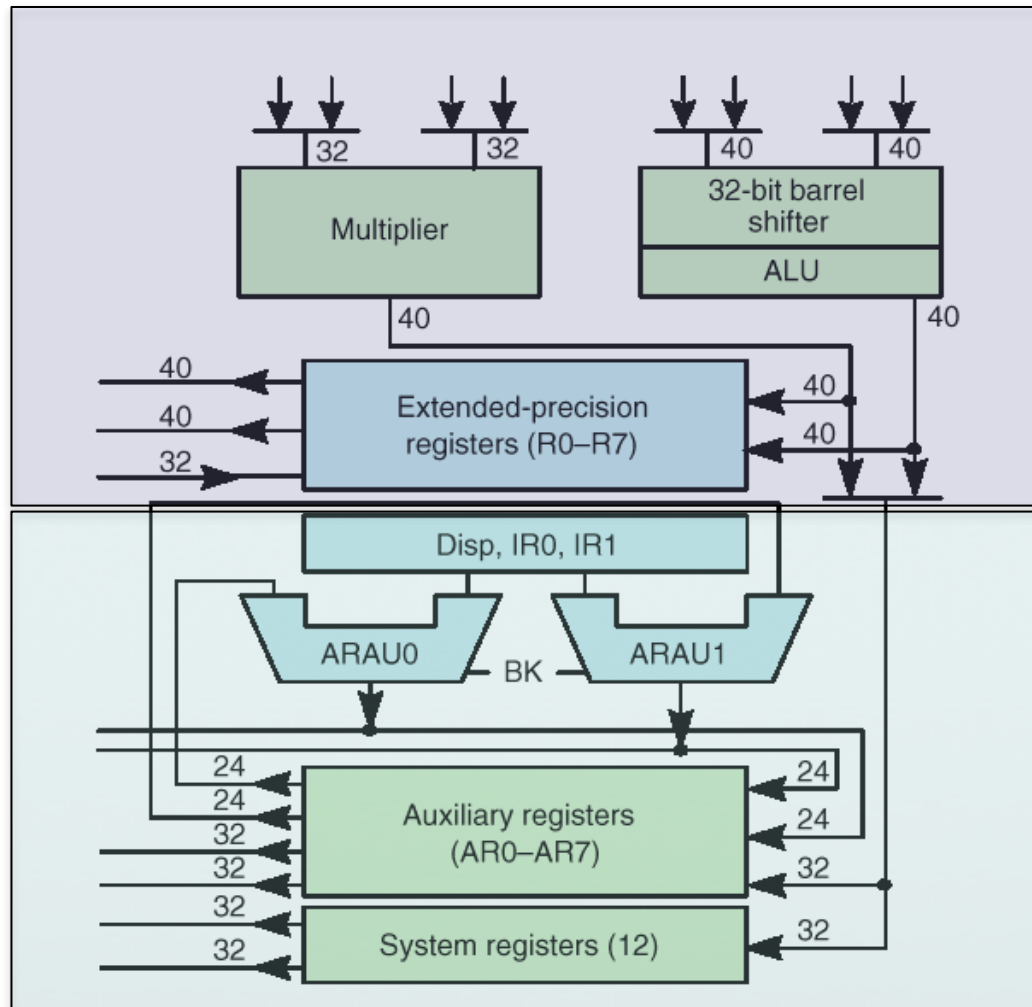
TMS320C30 Block Diagram



note: the TMS320C3x architecture is outdated today

TMS320C3x DSP Datapath

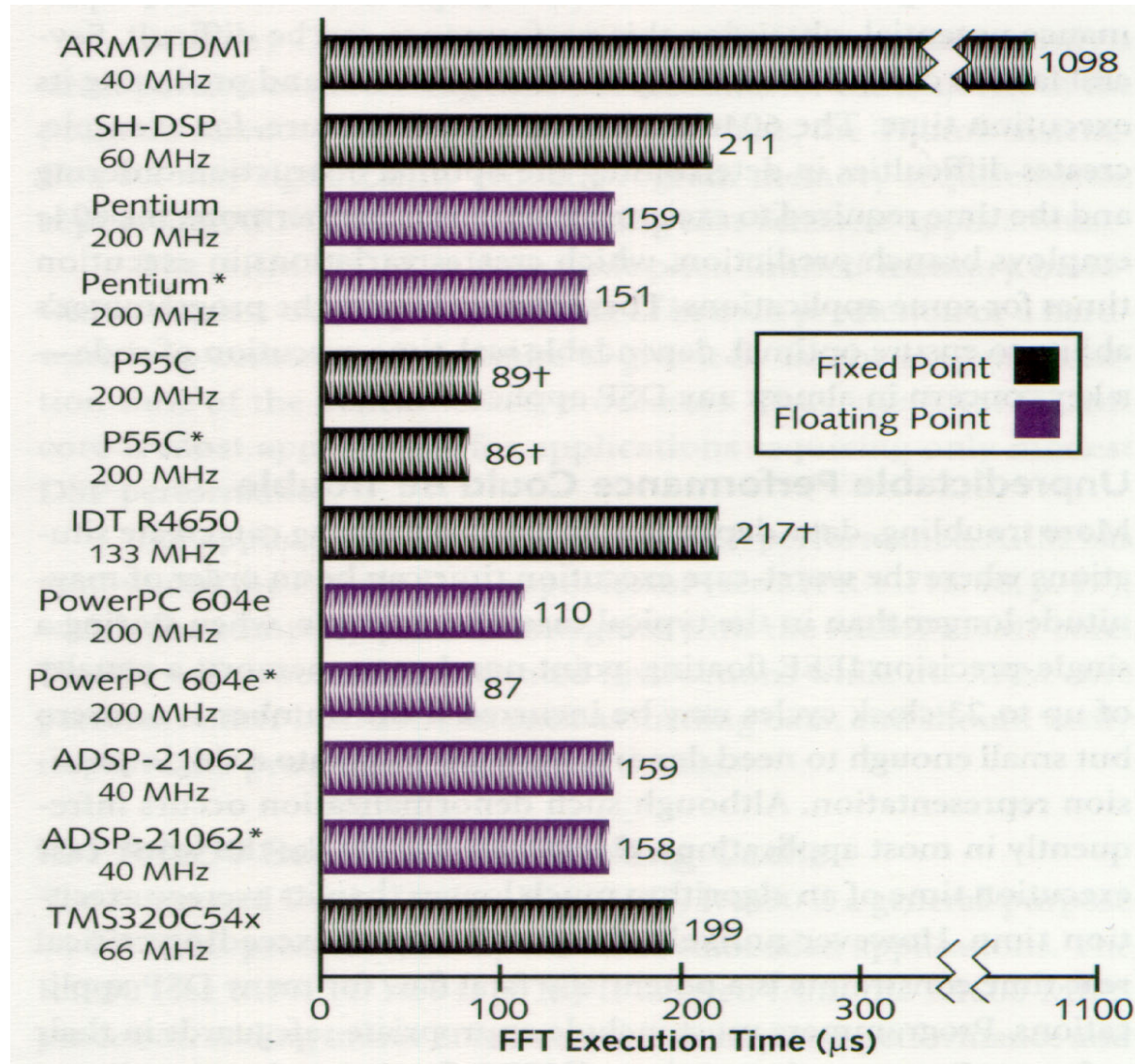
TMS320C3x CPU Block Diagram



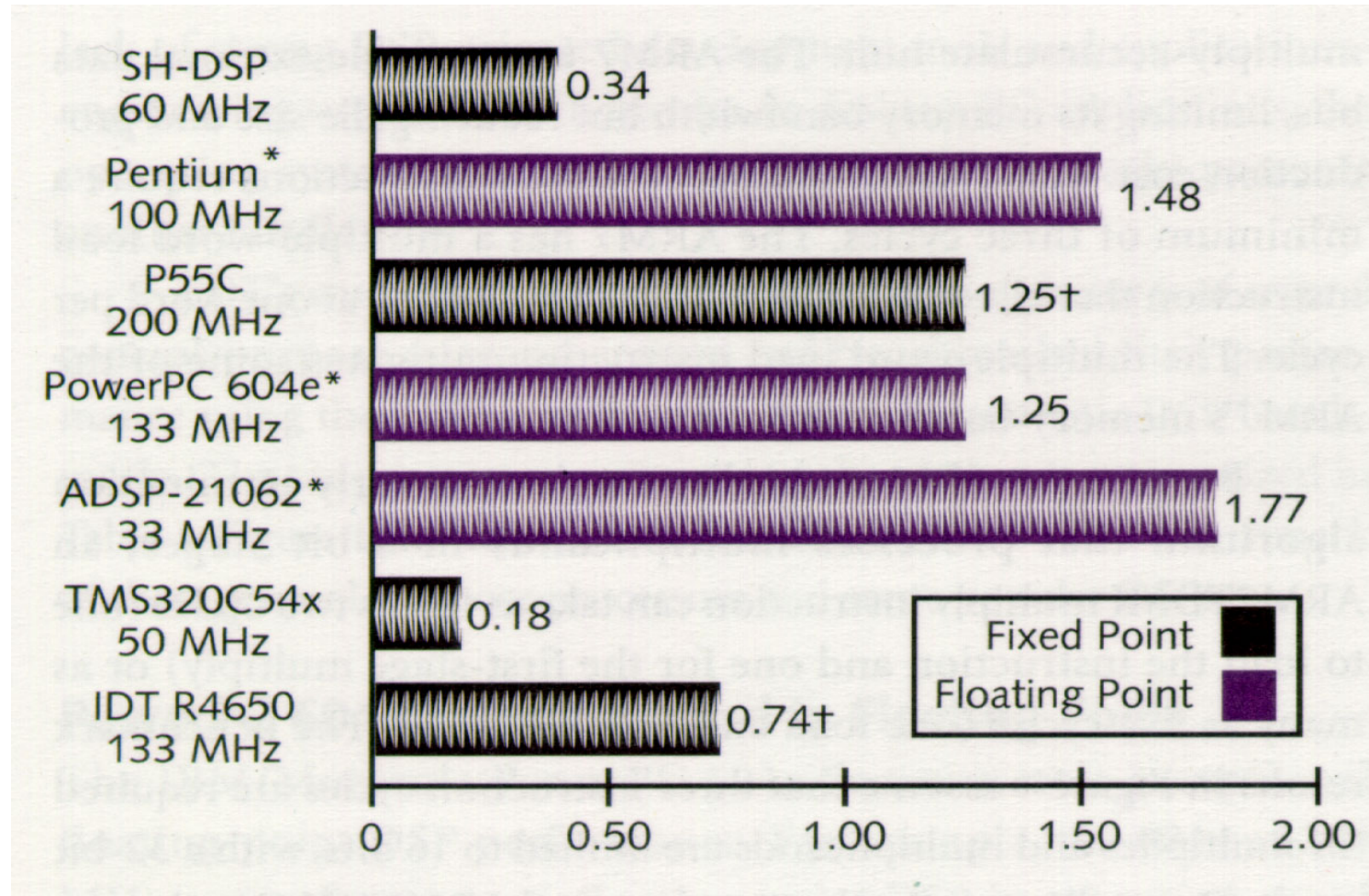
arithmetic units
for data

arithmetic units
for address
computations

DSP BDTI Performance Benchmarks (1)



DSP BDTI Performance Benchmarks (2)



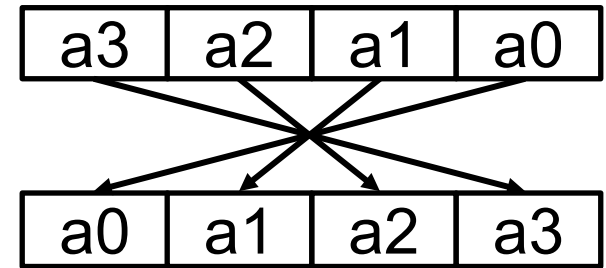
→
cost x execution time (cost-efficiency)

Application-Specific Instruction Set Processors

- application-specific specialization of a basic configurable CPU architecture
- specialization can cover
 - the instruction set
 - e.g. operator chaining (multiply-accumulate)
 - the functional units
 - e.g. saturating arithmetic, $1/\sqrt{x}$
 - the memory architecture
 - e.g. multiple memory blocks with parallel access
- advantages over regular processors
 - higher performance
 - lower cost (smaller chip area, fewer pins)
 - smaller code size
 - lower power consumption

Example for a Custom Instruction

- byte swap operation
 - convert between little endian and big endian
 - requires many instructions in software
trivial to implement in hardware

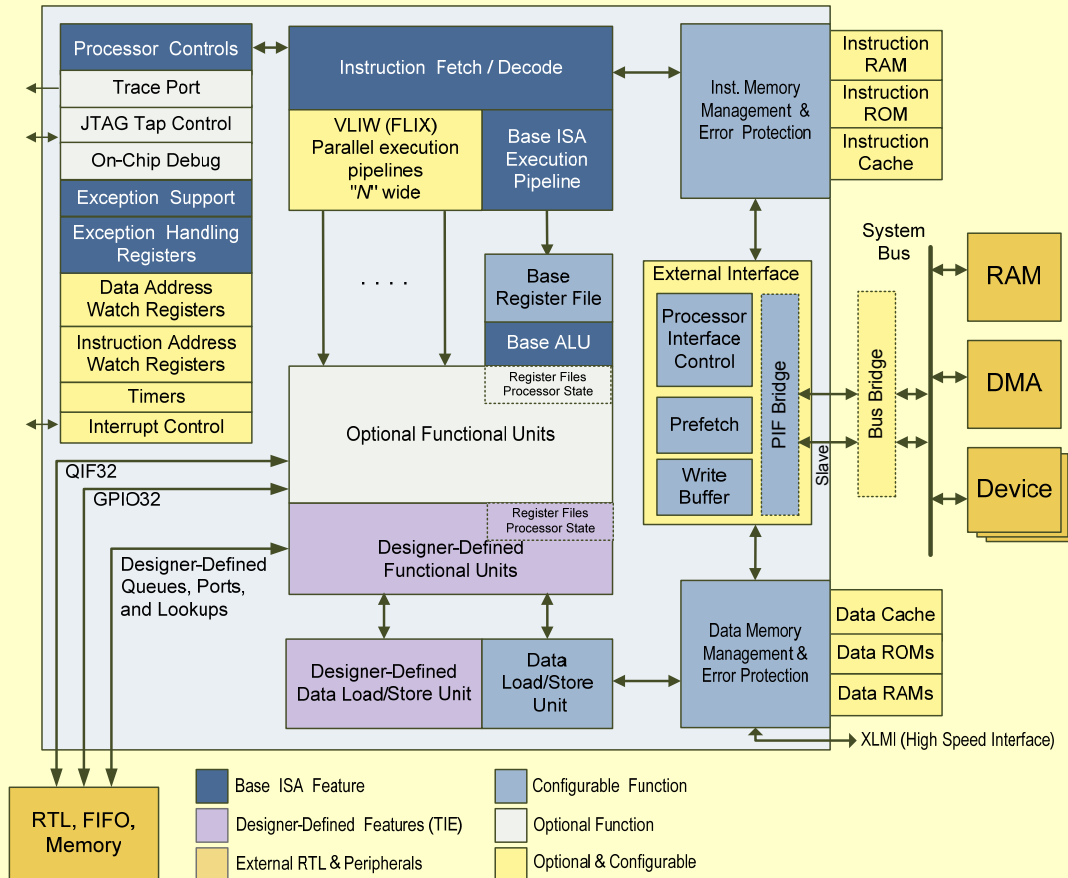


```
int byteswap(int x){
    return
    ((x << 24) & 0xFF000000) |
    ((x << 8) & 0x00FF0000) |
    ((x >> 8) & 0x0000FF00) |
    ((x >> 24) & 0x000000FF);
}
```

```
_byteswap:    @ @byteswap
@ BB#0:      @ %entry
    push {r0}
    mov r2, #16711680 // =0xFF0000
    mov r1, #65280    // =0xFF00
    and r2, r2, r0, lsl #8
    and r1, r1, r0, lsr #8
    orr r2, r2, r0, lsl #24
    orr r1, r2, r1
    orr r0, r1, r0, lsr #24
    add sp, sp, #4
    mov pc, lr
```

Example: Tensilica XTensa Configurable Processor

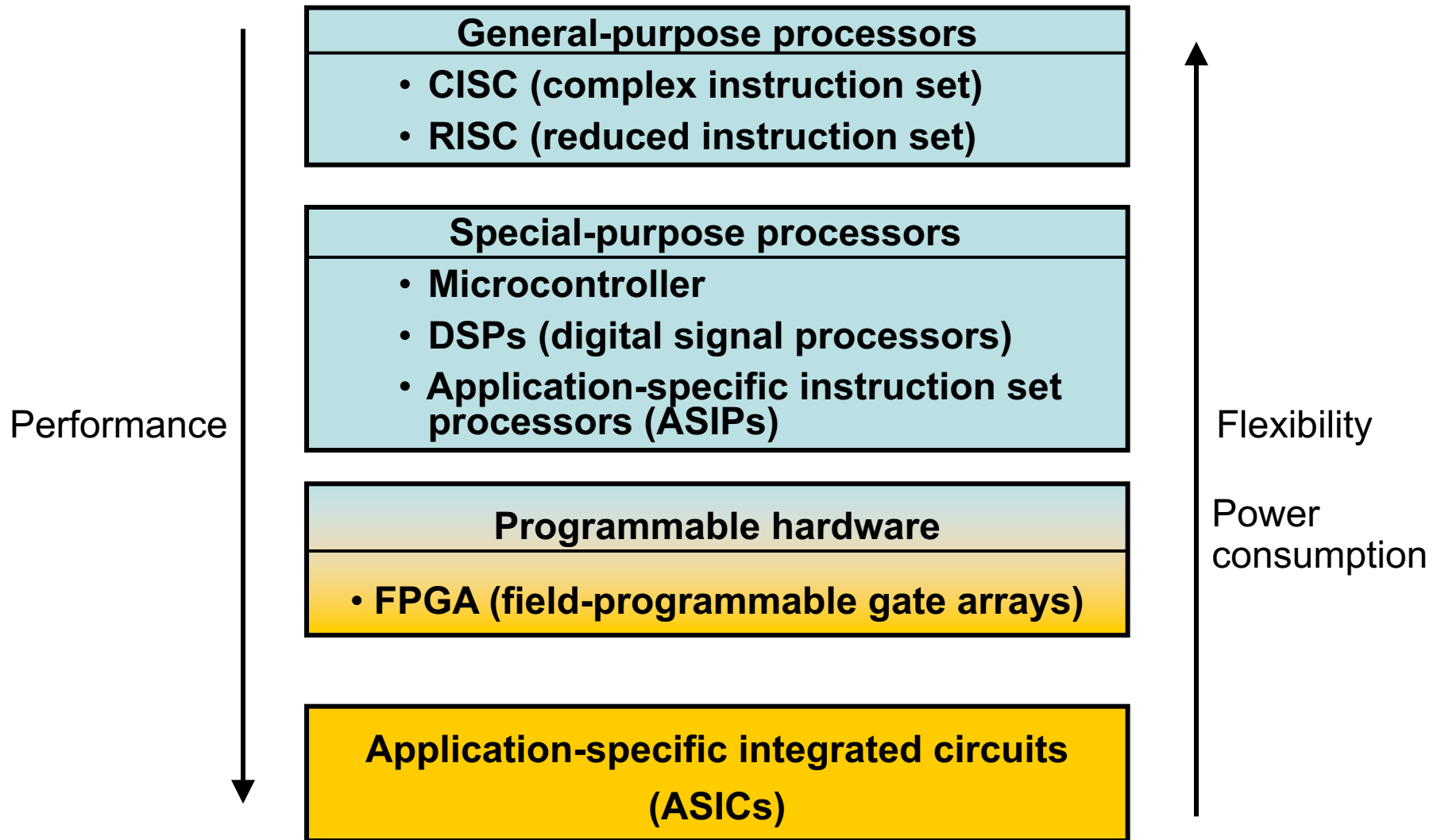
Xtensa LX4 DPU Architecture



- tool for automatically creating
 - IP core
 - simulator
 - compiler
 - power estimation
- user-defined scalar and VLIW instructions can be added
- resulting core can be implemented in silicon or in FPGA

Exercise 2.3: Processor Architectures

Implementation Alternatives



Phases – Integrated Circuits

1

Design



Modeling
Synthesis &
Optimization
Verification

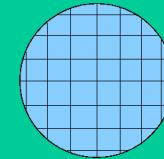
2

Fabrication

Masks



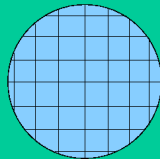
Wafer



3

Testing

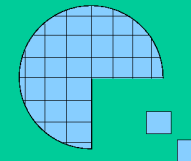
10000
01010
11001



4

Packaging

Slicing

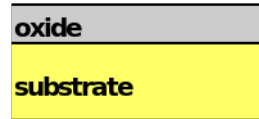


Packaging

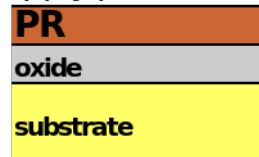


Photolithographic Processing of Silicon

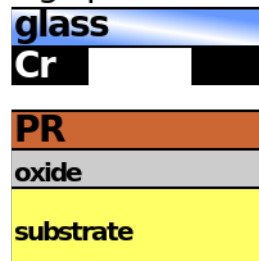
a. Prepare wafer



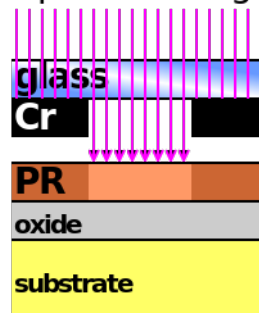
b. Apply photoresist



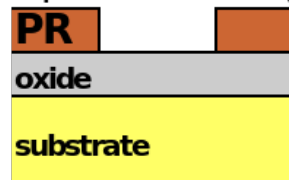
c. Align photomask



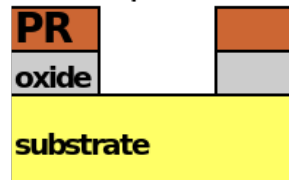
d. Expose to UV light



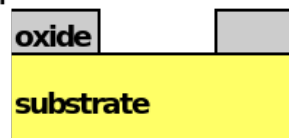
e. Develop and remove photoresist exposed to UV light



f. Etch exposed oxide

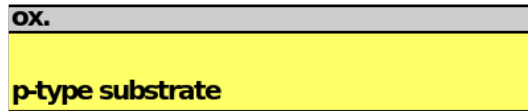


g. Remove remaining photoresist

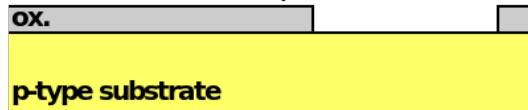


Semiconductor Fabrication

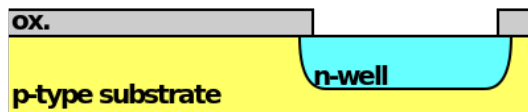
1. Grow field oxide



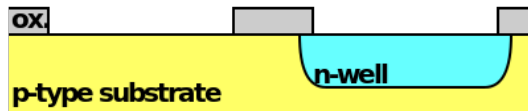
2. Etch oxide for pMOSFET



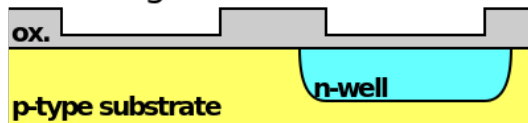
3. Diffuse n-well



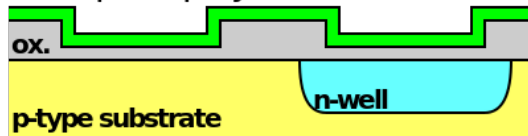
4. Etch oxide for nMOSFET



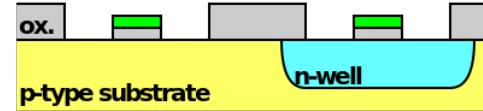
5. Grow gate oxide



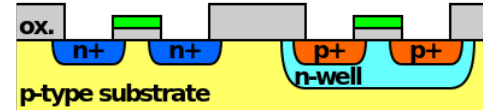
6. Deposit polysilicon



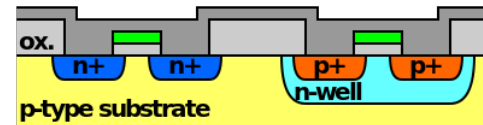
7. Etch polysilicon and oxide



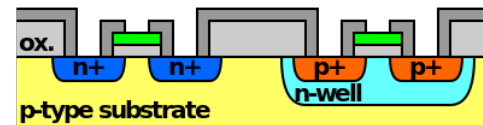
8. Implant sources and drains



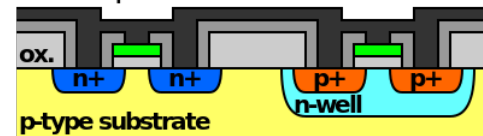
9. Grow nitride



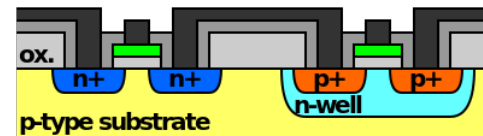
10. Etch nitride

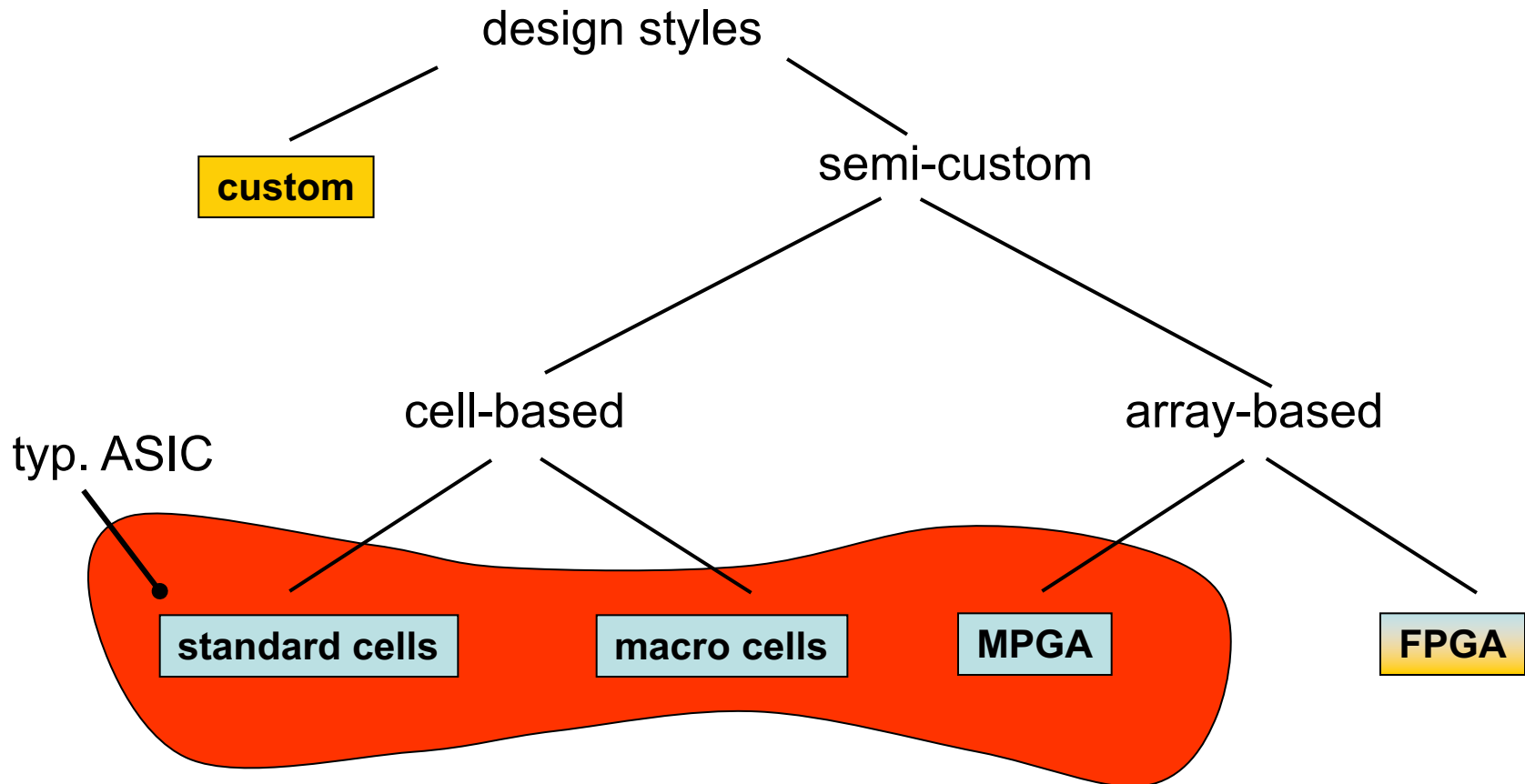


11. Deposit metal



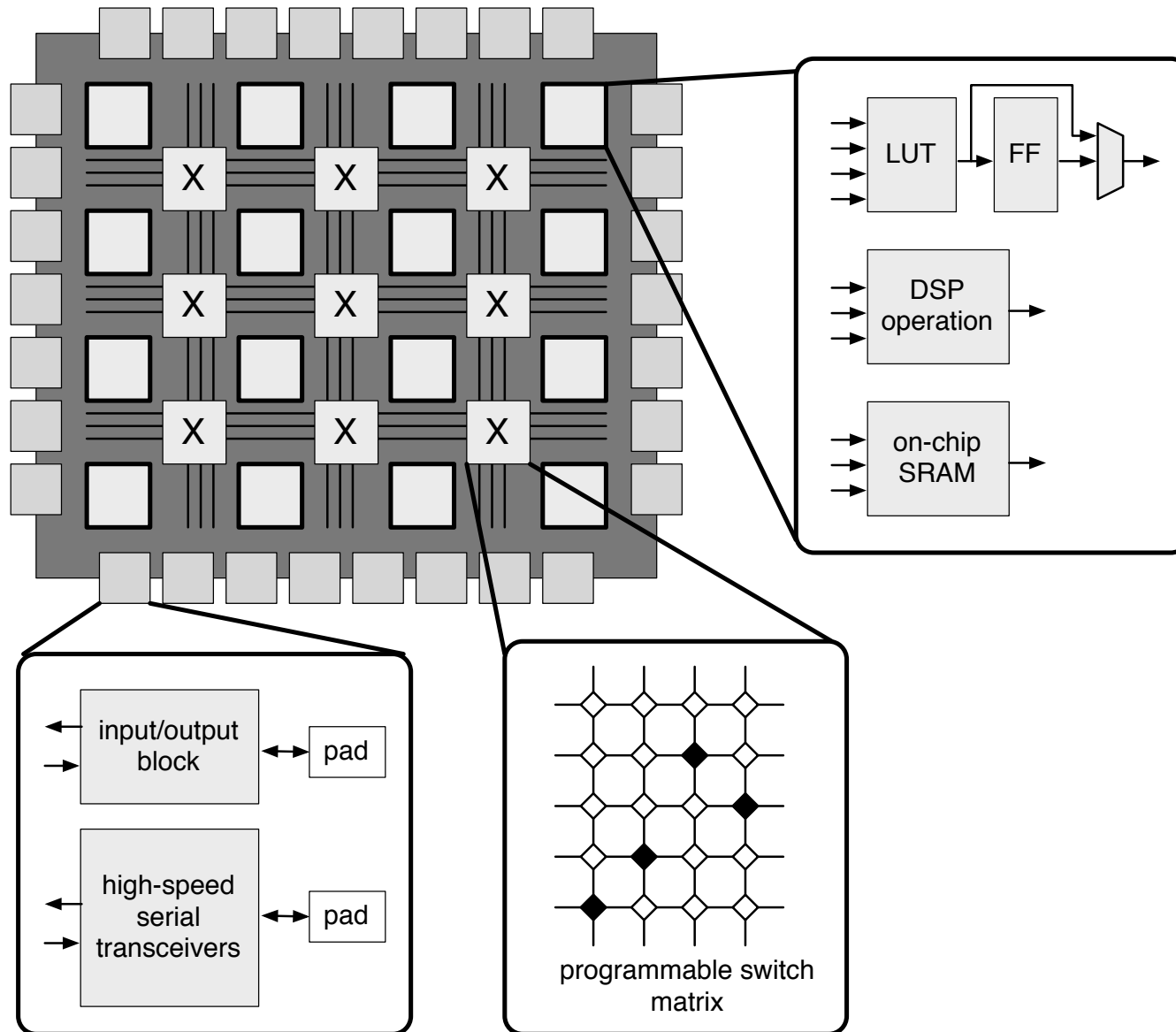
12. Etch metal





- MPGA: mask-programmable gate array („programmed” by the manufacturer)
- FPGA: field-programmable gate array („programmed” by the user)

FPGA – Basic Structures



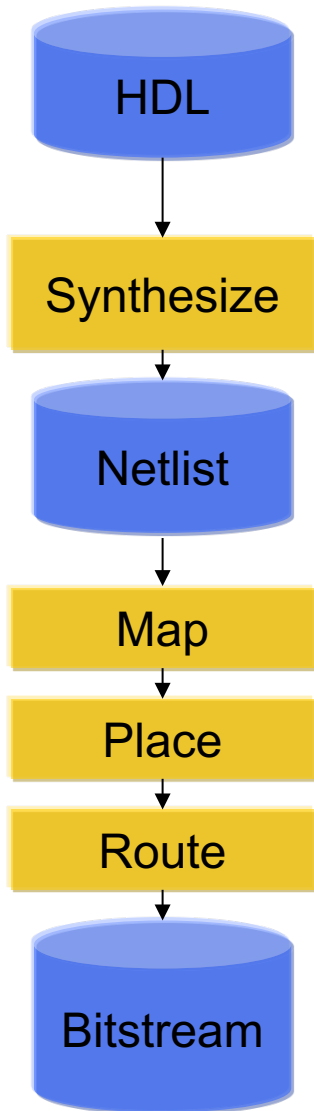
FPGA – Configuration and Application Domains

- configuration
 - all FPGAs components are software programmable (logic cell, DSP, IO-block functions, routing, ...)
 - configuration data (bitstream) is stored in SRAM cells
 - bitstream loaded from non-volatile memory at boot time
 - some devices can be re-configured at runtime
- application domains
 - glue logic
 - rapid prototyping, emulation
 - embedded systems
 - configurable system-on-chip
 - ASIC replacement
 - reconfigurable computing
 - computing without CPUs
 - combine processor-like programmability with ASIC-like performance

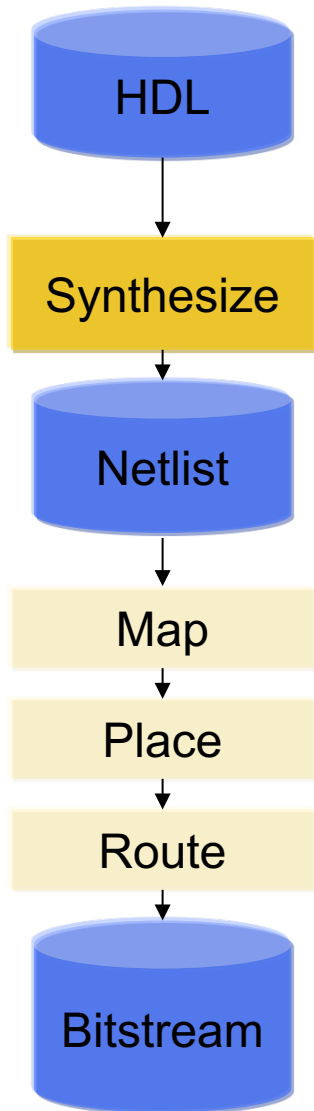
FPGA – Example Device Family

- Xilinx 7 series FPGA family
 - 1 logic cell contains 4 look-up tables and 8 flip-flops
 - very large capacity
 - e.g. Microblaze 32bit microcontroller softcore uses about 5000LUTs, i.e.
 - largest Kintex-7 device can implement around 400 CPU cores

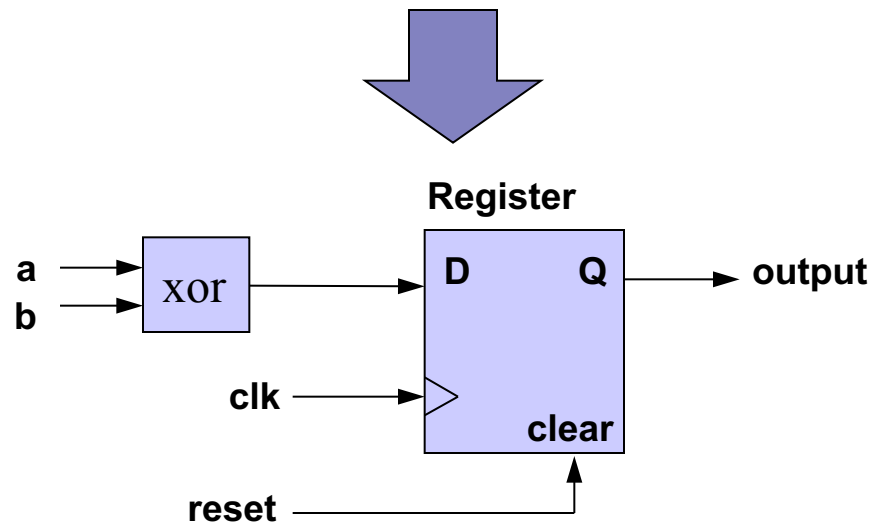
Maximum Capability	Artix-7 Family	Kintex-7 Family	Virtex-7 Family
Logic Cells	348K	478K	1,955K
Block RAM ⁽¹⁾	19 Mb	34 Mb	84 Mb
DSP Slices	1,040	1,920	5,280
Peak DSP Performance ⁽²⁾	1,129 GMACS	2,450 GMACS	6,737 GMACS
Transceivers	16	32	96
Peak Transceiver Speed	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Peak Serial Bandwidth (Full Duplex)	211 Gb/s	800 Gb/s	2,784 Gb/s
PCIe Interface	x4 Gen2	x8 Gen2	x8 Gen3
Memory Interface	1,066 Mb/s	1,866 Mb/s	1,866 Mb/s
I/O Pins	600	500	1,200
I/O Voltage	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V
Package Options	Low-Cost, Wire-Bond, Lidless Flip-Chip	Low-Cost, Lidless Flip-Chip and High-Performance Flip-Chip	Highest Performance Flip-Chip



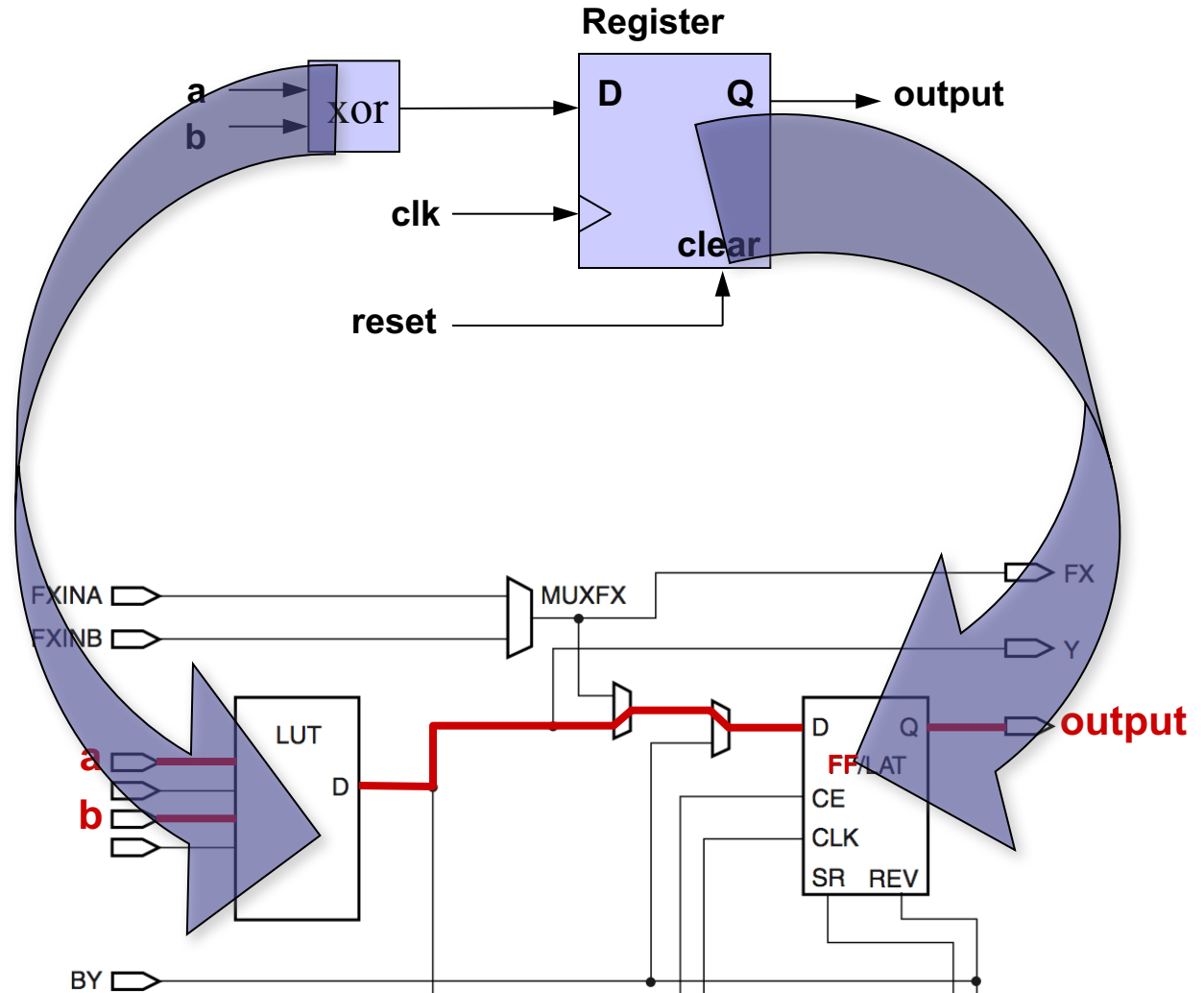
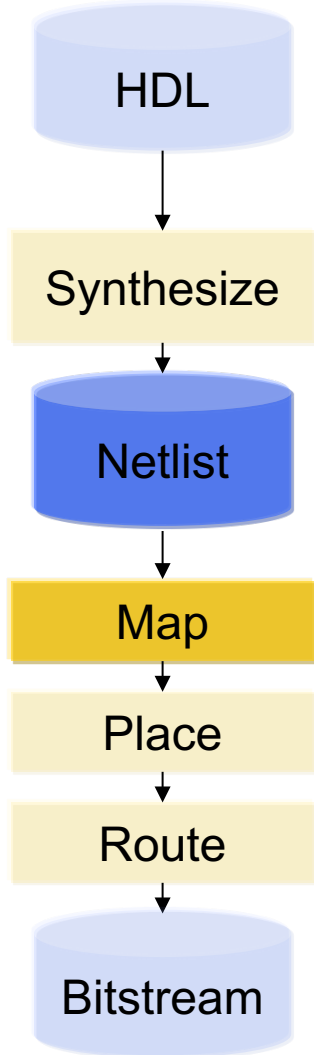
- Hardware design is traditionally done by modeling the system in a hardware description language (e.g. VHDL or Verilog)
- An FPGA synthesis tool (compiler) generates an netlist of basic logic elements,
 - which is then translated (mapped) to components available on the FPGA,
 - which are placed on the chip,
 - and the connecting signals are routed through the interconnection network.
- The resulting configuration data (bitstream) for programing the FPGA is created

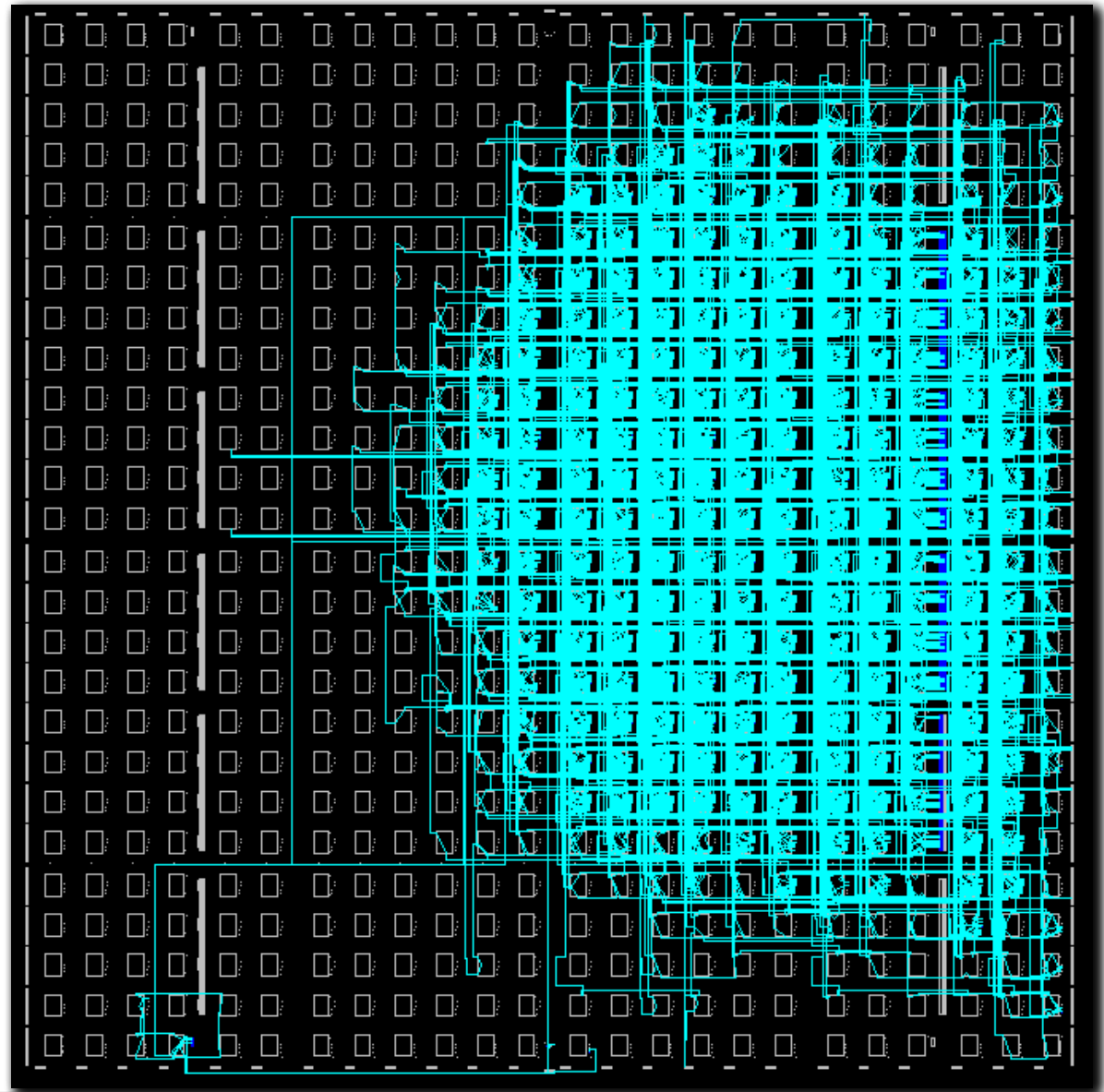
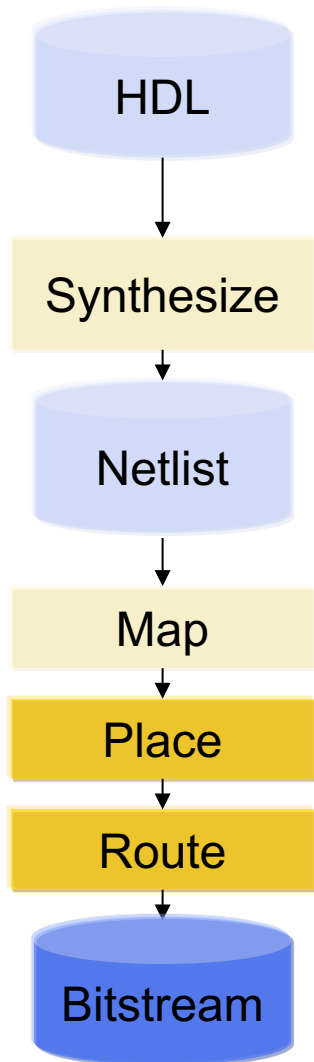


```
process(clk, reset)
begin
  if reset = '1' then
    output <= '0';
  elsif clk'event AND clk = '1' then
    output <= a XOR b;
  end if;
end process;
```



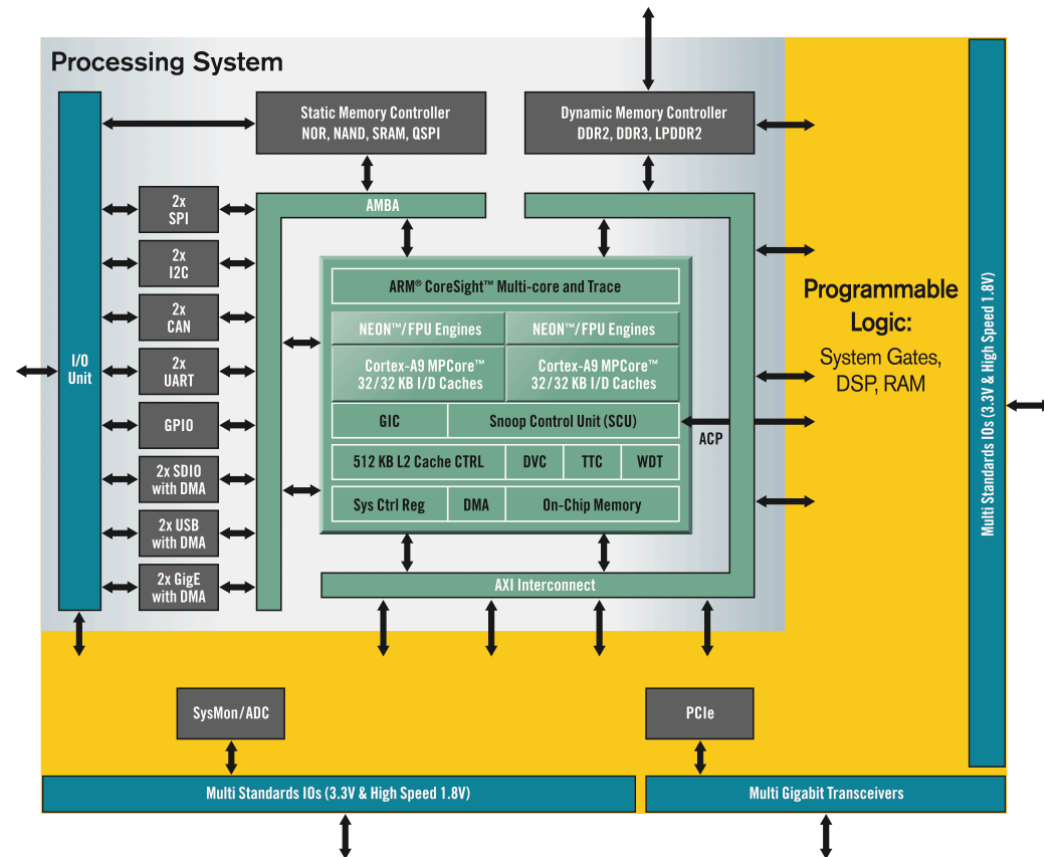
Technology Mapping





CPU/FPGA Hybrids

- integrating CPUs in FPGAs
 - implement soft-CPU core with programmable FPGA logic
 - integrate dedicated “hard” CPU cores
- hybrid CPU/FPGA architectures
 - Xilinx ZYNQ (FPGA+ dual-core ARM Cortex A9)
 - Xilinx Virtex FX (FPGA+ up to 2 PowerPC cores)
 - Intel Stellarton (Altera FPGA + Intel Atom)
- basis for configurable system-on-chip



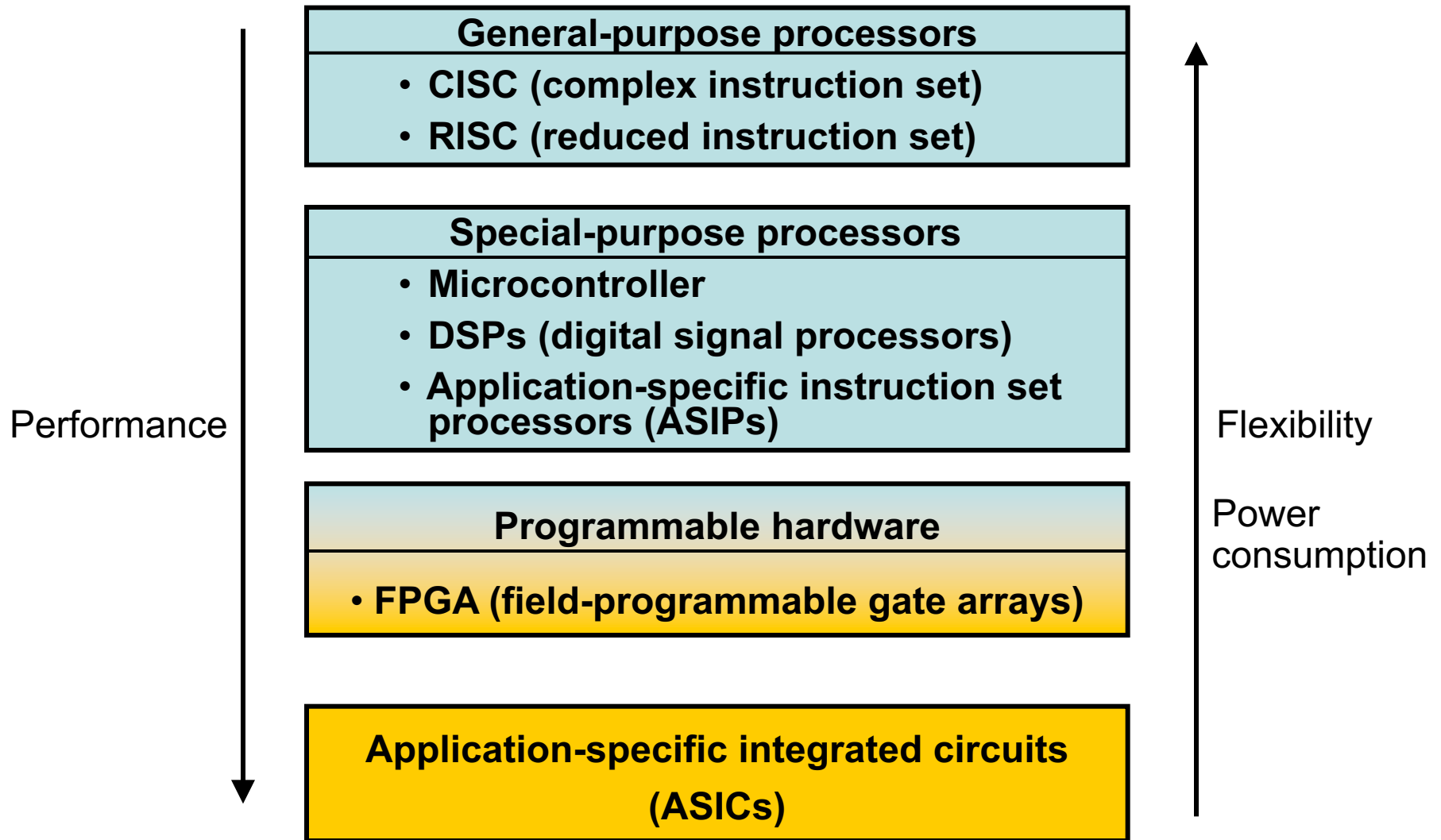
✎ Exercise 3.1: Digital Architecture Design

✎ Exercise 3.2: FPGA Lookup Table Mapping

Design Methodologies - Comparison

	Custom	Cell-based	MPGA	FPGA
Density	very high	high	high	medium-low
Performance	very high	high	high	medium-low
Design time	very long	short	short	very short
Manufacturing time	medium	medium	short	very short
Cost-low volume	very high	high	high	low
Cost-high volume	low	low	low	high

Implementation Alternatives



- v1.6.0 (2017-04-23)
 - minor updates for SS2017
- v1.5.2 (2016-05-02)
 - fix xor sign on gate on slides 51 and 52
 - design methodologies slide move towards end of presentation
- v1.5.1 (2016-04-18)
 - fix type on title slide, garbled symbols on slide 8
- v1.5.0 (2016-04-08)
 - minor updates for SS2016

- v1.4.0 (2015-04-13)
 - minor updates for SS2015
- v1.3.0 (2014-04-09)
 - minor updates for SS2014
- v1.2.2 (2013-04-26)
 - corrected semantics of RPTS description on p.33, and byteswap example on p.40
- v1.2.1 (2013-04-18)
 - add references to exercises and excursions
- v1.2.0 (2013-04-10)
 - minor updates for SS2013
- v1.1.1 (2012-04-13)
 - added illustration for MSP430 application (wireless sensor networks)
- v1.1.0 (2012-04-10)
 - added more slides illustrating the FPGA toolflow