Computer Systems Design

Lesson 3

Abstraction levels of programmable computer systems

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

ITMO

Outline the lesson

- Common levels of programmable computer systems
- Difference between processor architecture, microarchitecture, physical implementation
- Difference between hardware and software operation principles
- Instruction set architectures
- Processor’s performance: iron law, CPU scalability, Amdahl’s law

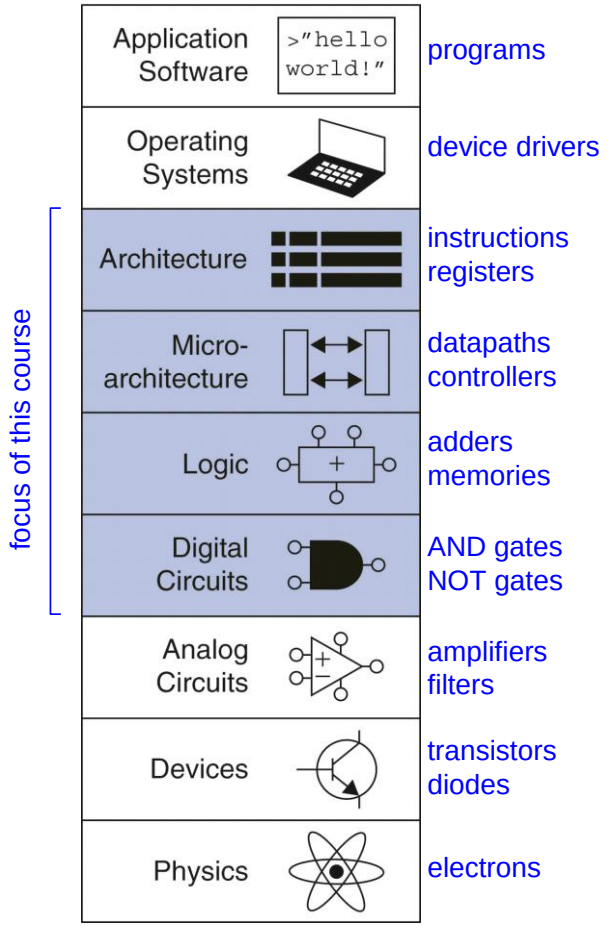
Common levels of programmable computer system organization

EDA (Electronic Design Automation) – computer aided design of electronics (IC, PCBs, …)

The promise of designing in a better way is as old as EDA itself. The evolution towards ***higher abstractions*** is rooted in EDA's DNA *

Computer architecture is a ***multi-layer structure***, descending from ***application-specific models*** down to ***generic mechanisms*** of computational process organization

Our focus: layers capturing ***functional aspects*** of computer systems architecture "around" HW/SW interface



** T. Bollaert. High-Level Synthesis Blue Book. Mentor Graphics, 2010*

Picture: D.M.Harris, S.L.Harris Digital Design and Computer Architecture, 2nd Ed., 2012

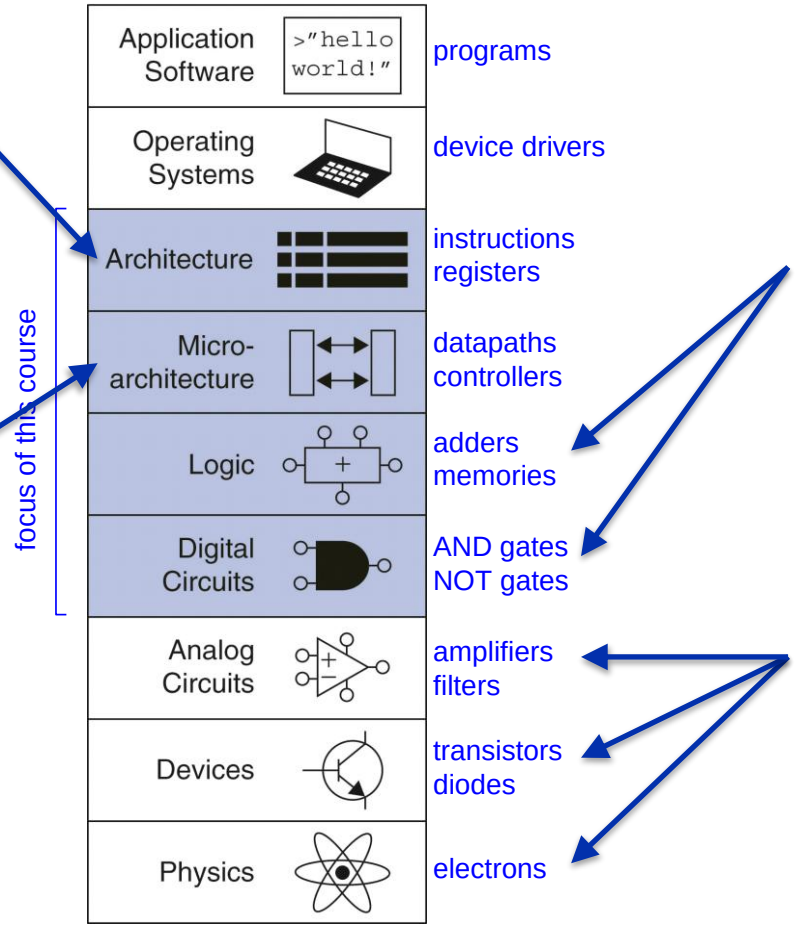
Typical separation of concerns for programmable processors

Architecture, Instruction Set Architecture (ISA): "programmer's view" of the processor

- *binary encoding of instructions*
- *number of software-addressable registers*
- *memory address spaces*
- *ordering of memory requests (memory model)*
- *byte ordering (endianness)*
- *exceptions/interrupts handling*
- *execution modes: user, privileged, etc.*

Microarchitecture, internal logical organization of the processor

- *processor pipeline structure*
- *number of hardware execution units*
- *control unit structure*
- *clock cycles needed for instruction execution*
- *hardware instruction scheduling algorithm*
- *instruction execution ordering*
- *instruction execution speculations*
- *cache memory levels and sizes*



Synthesized hardware

- *network of basic hardware elements:*
  - *combinational logic*
  - *registers*
  - *memories*

Physical implementation

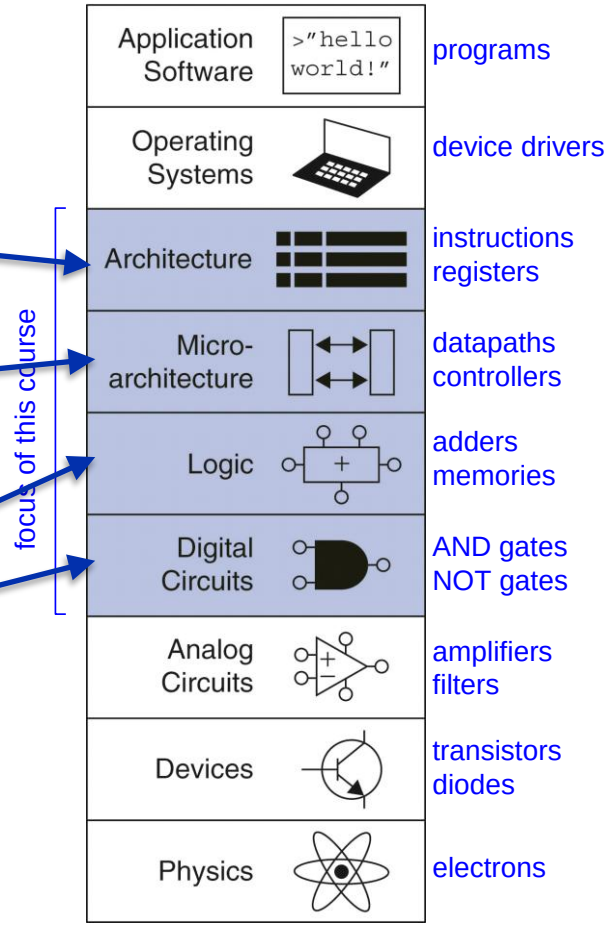
- *technology node / FPGA platform*
- *clock frequencies*
- *logical elements placement*
- *signals routing*

Levels covered in the course

Lectures 4-8

Lectures 9-16

Will be related eventually
(more focus in Functional Circuit Design disciplines)



ITMO

Examples of processor architectures and designs

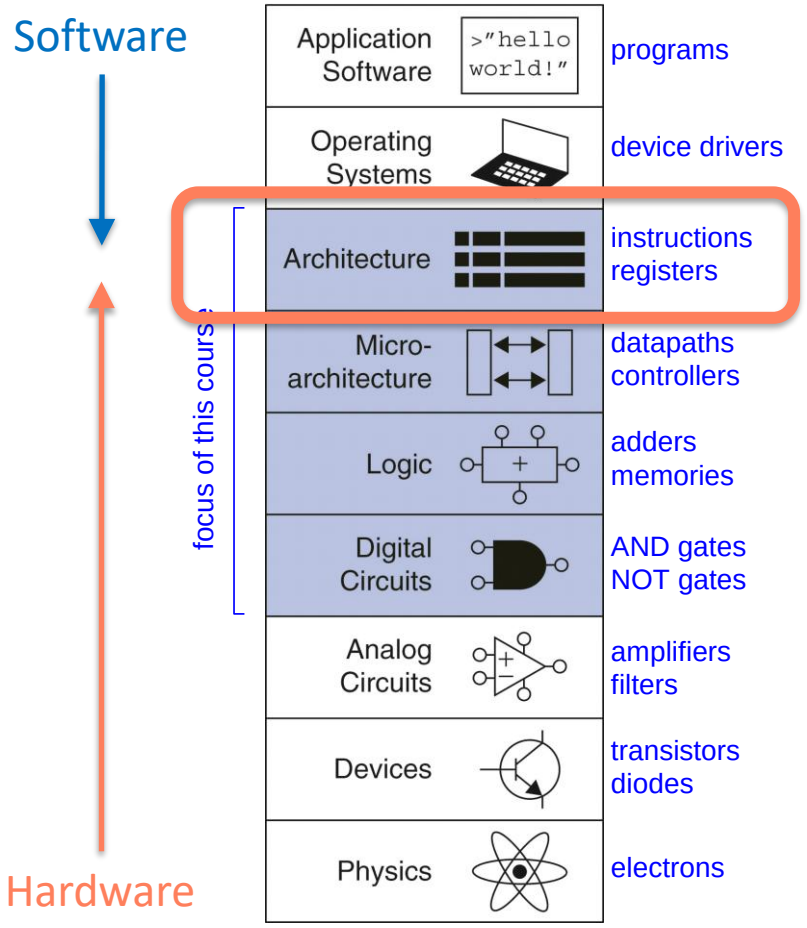
- **x86/x86-64**
  - **Developers of architectures:** Intel, AMD
  - **Developers of designs:** Intel, AMD, Zhaoxin, Cyrix, VIA Technologies
- **ARMv7/ARMv8**
  - **Developers of architectures:** ARM
  - **Developers of designs:** ARM, Apple, AppliedMicro, Broadcom, Cavium, Digital Equipment Corporation, Intel, Nvidia, Qualcomm, Samsung Electronics
- **MIPS**
  - **Developers of architectures :** MIPS Technologies, Imagination Technologies
  - **Developers of designs :** MIPS Technologies, Imagination Technologies, Broadcom, Cavium, NXP Semiconductors
- **RISC-V**
  - **Developers of architectures:** Berkeley, RISC-V Foundation/International
  - **Developers of designs:** SiFive, Western Digital, ETH Zurich & Università di Bologna, Andes, Codasip, ...

Instruction Set Architecture (ISA)

ISA: “programmer’s view” of the processor

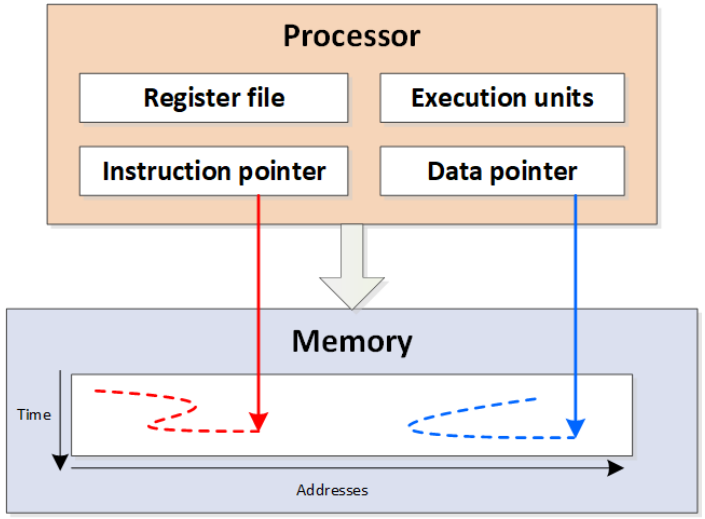
Defines a ***contract between HW and SW*** (SW “meets” HW)

SW built for certain ISA can run correctly on any HW implementing it (with various performance)

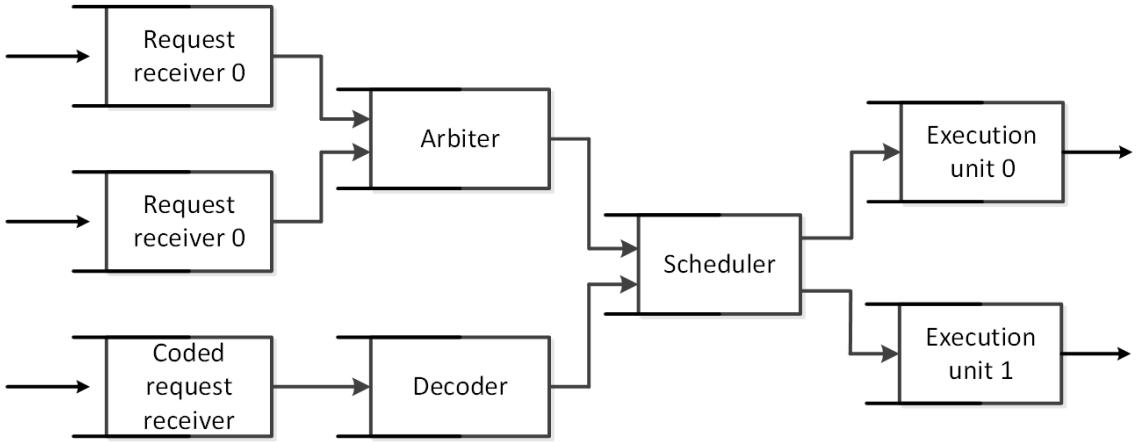


HW and SW: principles of operation

Software:
CPU core traversing the program, directed by instruction pointer



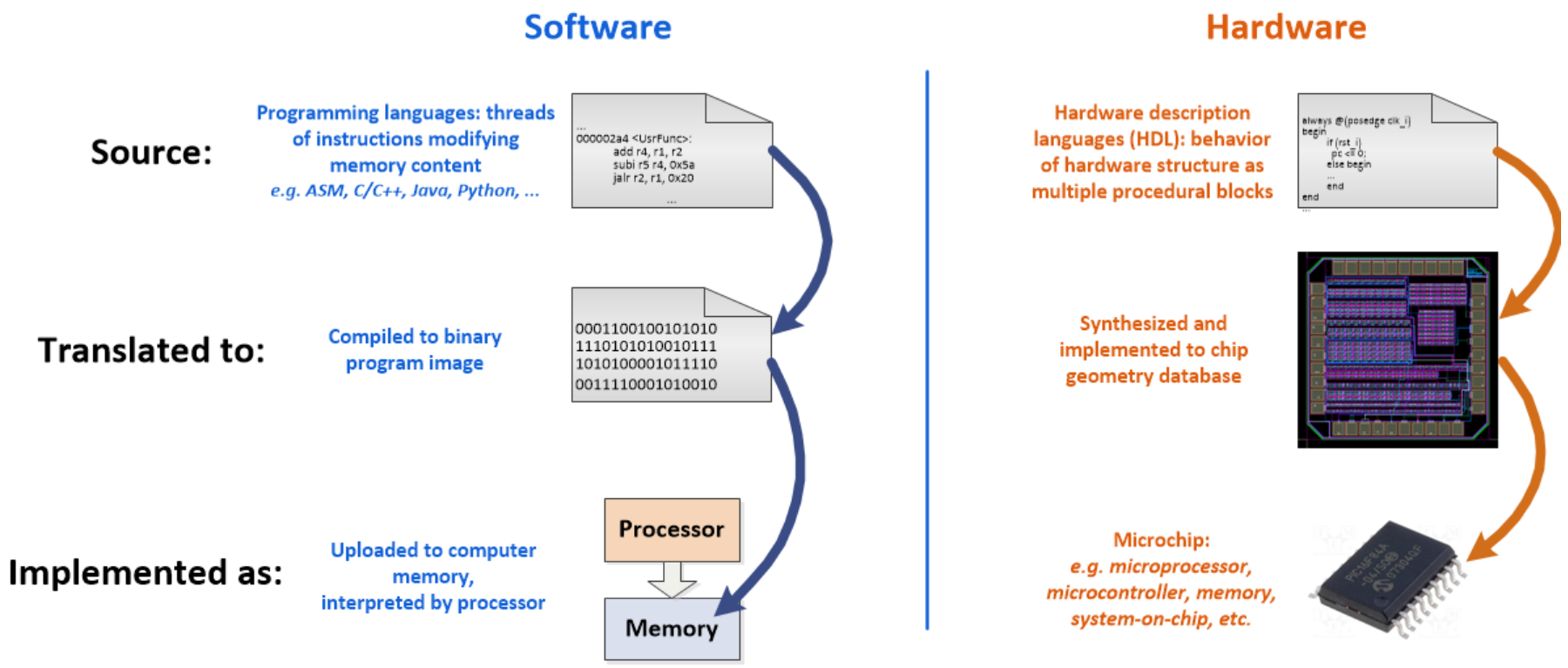
Hardware:
streams of transactions simultaneously traversing through the structure



However, many optimizations are conceptually similar

HW and SW: design languages

Both HW and SW is designed mostly using ***textual languages***



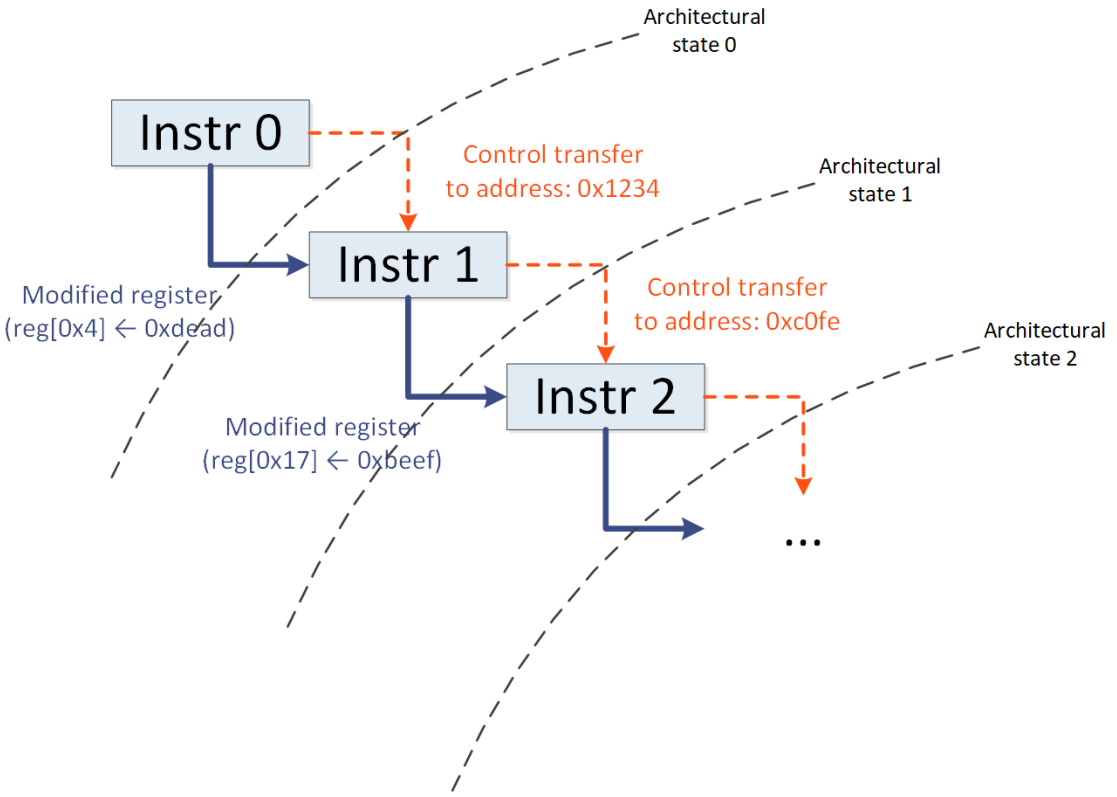
ITMO

ISA in instruction flow processors

Program: sequence ***(thread)*** of basic instructions operating on ***addressable registers/memory pool***

Each instruction modifies ***architectural (software-visible) state***:

1) modifies the registers/memory state
2) passes control to some next instruction (if not branch – to next instruction in memory)



ITMO

Iron law of processor performance

Program execution time can be computed as:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Number of instructions in program depends on:

- source code
- compiler
- ISA

Number of cycles per instruction (CPI) depends on:

- ISA
- microarchitecture

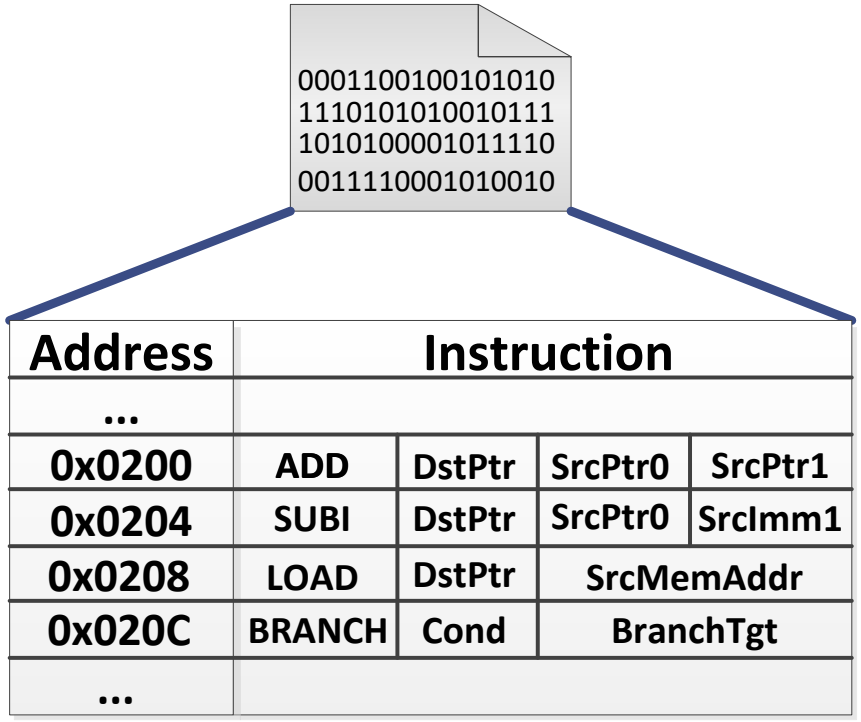
Clock cycle period depends on:

- microarchitecture
- implementation technology

Instruction contents

Processor instructions have typical set of fields:

1) opcode: defines instruction type
   arithmetical/logical operation, memory load/store, branches, special functions
2) source data pointer(s)
3) destination data pointer(s)
4) immediate values
5) other information
   extra details, conditions, performance hints, etc.



Address	Instruction			
...				
0x0200	ADD	DstPtr	SrcPtr0	SrcPtr1
0x0204	SUBI	DstPtr	SrcPtr0	SrcImm1
0x0208	LOAD	DstPtr	SrcMemAddr	
0x020C	BRANCH	Cond	BranchTgt	
...				

Memory addressing features

Addressable memory objects:

1) bit (mostly microcontrollers)
2) byte (8 bits)
3) half word (16 bits)
4) word (32 bits)
5) double word (64 bits)

Addressing schemes:

1) byte addressing (common)
2) word addressing

Endianness – bytes ordering in accessed word:

1) Little-endian
   low address – low byte
2) Big-endian
   low address – big byte

Byte addressing

Address	Data
…	…
0x200	0x34
0x201	0xFA
0x202	0x80
…	…

Word addressing

Address	Data
…	…
0x200	0xe490aa10
0x201	0xc456a372
0x202	0xbadc0ffe
…	…

Address	Data
…	…
0x200	0x34
0x201	0xFA
0x202	0x80
0x203	0xCD
…	…

Little endian

0x200	0xCD80FA34

Big endian

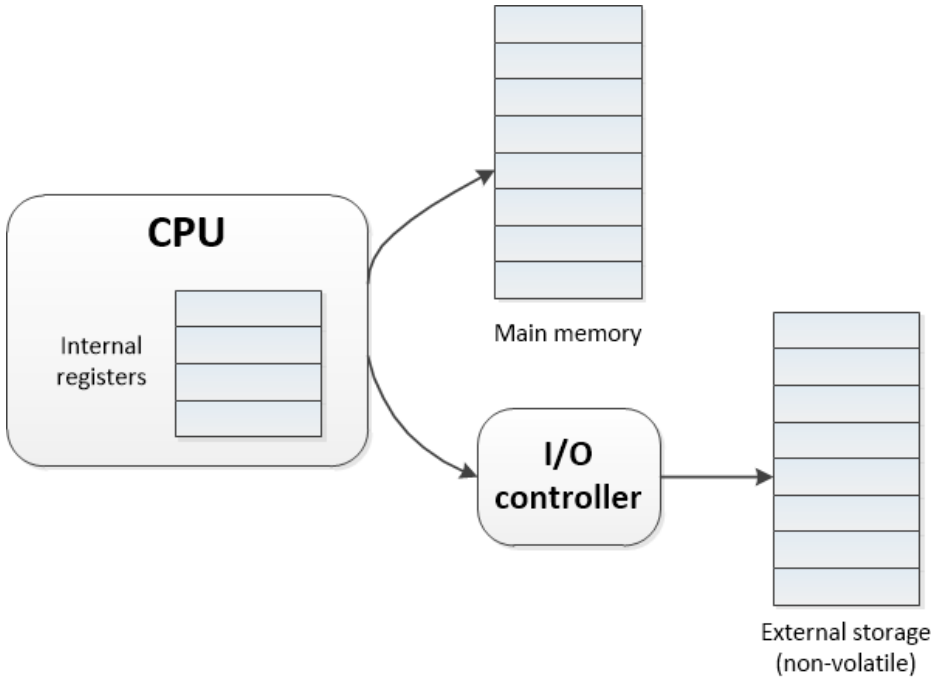
0x200	0x34FA80CD

Typical addressable memory pools

1) Registers inside CPU
   - *small size (units/tens of words)*
   - *accessible by all instructions*
   - *low latency (units of CPU clock cycles)*
   - *volatile*

2) Main memory
   - *bigger size (units of gigabytes or more)*
   - *accessible by some instructions*
   - *larger latency (hundreds of CPU clock cycles)*
   - *volatile*

3) External storage
   - *potentially infinite size*
   - *indirectly accessible (through I/O controller)*
   - *large latency (millions of CPU clock cycles)*
   - *non-volatile*



ITMO

Alignment

Addressing	*0x000*	*0x001*	*0x002*	*0x003*	*0x004*	*0x005*	*0x006*	*0x007*
Byte	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
Half	Aligned		Aligned		Aligned		Aligned	
Half		Misaligned		Misaligned		Misaligned		Misaligned
Word	Aligned				Aligned			
Word		Misaligned				Misaligned		
Word			Misaligned				Misaligned	
Word				Misaligned				Misaligned
Double	Aligned							
Double		Misaligned						
Double			Misaligned					
Double				Misaligned				
Double					Misaligned			
Double						Misaligned		
Double							Misaligned	
Double								Misaligned

ITMO

Basic addressing modes

- Many algorithms rely on computed addresses for data
- ISAs typically implement immediate addressing for constants, register addressing (by constant register addresses) for computed data and indirect register addressing for computable addresses of data (stored in memory)
- Other addressing modes can be implemented in SW using the basic ones

Addressing mode	Code example	Mnemonics	Description
Immediate	Add R4,3	Regs[R4]←Regs[R4]+3	Value is instruction field (constant)
Register	Add R4,R3	Regs[R4]←Regs[R4]+Regs[R3]	Value is in register
Register indirect	Add R4,(R1)	Regs[R4]←Regs[R4]+Mem[Regs[R1]]	Using computed address from the register to access memory

Problem: trade-off between ISA complexity and instructions/program

ITMO

Advanced addressing modes

Addressing mode	Code example	Mnemonics	Description
Immediate	Add R4,3	Regs[R4]←Regs[R4]+3	Value is instruction field (constant)
Register	Add R4,R3	Regs[R4]←Regs[R4]+Regs[R3]	Value is in register
Register indirect	Add R4,(R1)	Regs[R4]←Regs[R4]+Mem[Regs[R1]]	Using computed address from the register to access memory
Displacement	Add R4,100(R1)	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	Indirect with offset
Indexed	Add R3,(R1+R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs[R2]]	Useful for accessing arrays at variable addresses and offsets
Direct or absolute	Add R1,(1001)	Regs[R1]←Regs[R1]+Mem[1001]	Useful for accessing static data (with big constant address)
Memory indirect	Add R1,@(R3)	Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]	Accessing data by pointer residing in memory
Autoincrement/ -decrement	Add R1,(R2)+	Regs[R1]←Regs[R1]+Mem[Regs[R2]]; Regs[R2]←Regs[R2]+D	Useful for passing through an array (R2 is auto-incremented by element size D)
Scaled	Add R1,100(R2)[R3]	Regs[R1]←Regs[R1] +Mem[100+Regs[R2]+Regs[R3]*D]	Used for randomly indexing array at dynamic location
Implicit/implied	Add R1	Regs[R1]←Regs[R1]+A	A is taken from pre-defined location

ITMO

Functionality in SW (vN architecture) vs. HW

Design aspect	SW	HW
Static bit manipulations (shift, ranging, concatenation)	Require extra CPU instructions	Free, implemented by signals' routing
Activity of functionality	Code activates when pointed by CPU instruction pointer. "Passively" resides in memory in all remaining time.	The entire functionality can be active in each moment of time and can potentially do useful processing (i.e. in parallel and pipelined manner)
Data accessibility	Free access to the whole RAM, can cause stalls due to cache misses (for unpredictable patterns) and cache coherence communications (in multi-cores)	Needs to be routed to all consumers, constrained by wire lengths and fanout

ITMO

Functionality in SW (vN architecture) vs. HW

Design aspect	SW	HW
Timing severity	Relaxed. Changed timing weakly affects correctness (if not hard real time)	Strict. Processing should be split in clock cycles and stages, failed timing paths may ruin correctness
Scheduling dynamism	Dynamic. CPUs continuously construct schedules according to data availability, events, etc.	Static. Schedule is fixed in design time and embedded in HW control logic
Time/space folding automation	Automatic, superscalar out-of-order CPUs route ready instructions to execute on multiple execution units	Manual (needs re-design of datapath and controller)
Performance scalability	Limited. Single-core: limited by Pollack's rule. Multi-core: limited by Amdahl's Law	Defined by application and design (up to linear)

ITMO

Single-core CPU scalability: Pollack's rule (~2008)

Fred Pollack: microprocessor engineer, Intel (iAPX 432, i960)

Empirical observation for superscalar processors:

"Performance increase is roughly proportional to [the) ***square root*** *of [the] increase in complexity".*

- 2x complexity → 1.4x perf
- 4x complexity → 2x perf
- 16x complexity → 4x perf
- …

Serves as justification of multi-core computing

ITMO

Multi-core CPU scalability: Amdahl's law (1967)

Gene Amdahl: computer scientist, IBM (1922-2015)

- p – parallelizable workload fraction
- (1-p) – non-parallelizable workload fraction
- N – number of processors
- p/N – speedup for parallel workload part

$$Speedup = \frac{1}{(1-p) + p/N}$$

Examples (for infinite processors count):

- 50% sequential code → 2x speedup limit
- 25% sequential code → 4x speedup limit
- 10% sequential code → 10x speedup limit
- …

Processing latency is fundamentally limited by non-parallelizable workload fraction
Justifies workload ("application-level") parallelism

ITMO

Functionality in SW (vN architecture) vs. HW

Design aspect	SW	HW
Processing variability	Infrequently used branches and functions consume memory, but weakly affect performance	All processing options consume area, power (if not gated), can affect frequency and performance
Complexity/ performance balancing	Automatic, new functions can be added/removed, performance decreases/increases accordingly	Manual (needs re-design)
Performance predictability	Loosely predictable. Distributed dynamic hardware optimizers (schedulers, caches, branch predictors) in CPUs disrupt precise performance evaluation	Precisely predictable. Simulation gives clock cycle precision, clock cycle period can be evaluated from implementation

ITMO

Functionality in SW (vN architecture) vs. HW

Design aspect	SW	HW
Implementation speed	Fast. Includes: preprocessing, compilation, optimization, linking, running	Slow. Includes: preprocessing, synthesis, translation, placement, routing, bitstream/geometry generation, production (for ASICs)
Updatability of functionality	Yes (if instruction memory writable), straightforward self-modification.	Restricted to pre-defined RAMs, registers, and configuration ports. Extreme case: FPGA (entire logic defined by programmable LUTs and interconnects, trading order of efficiency).
Hiring engineering workforce	Relatively easy	Relatively difficult

Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025