Computer Architecture

Lecture 01 - Introduction

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi’an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

Course Administration

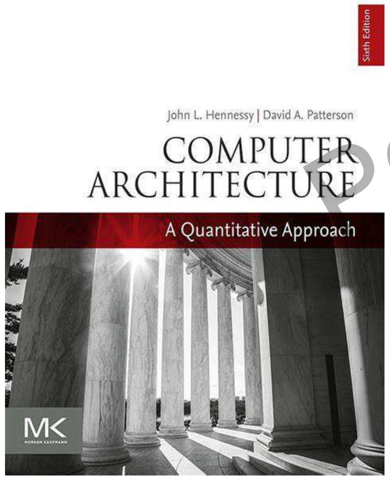
Instructor: Pengju Ren & Tian Xia

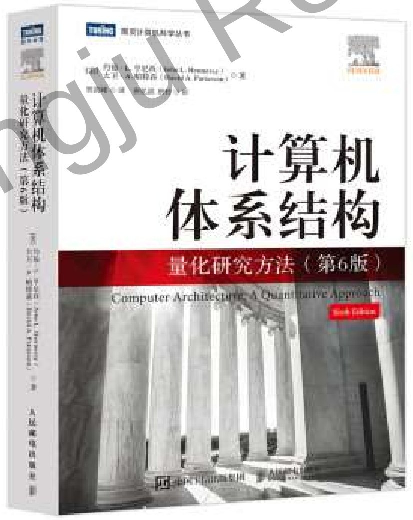
TA: Siyang Wang（Ph.D Candidate)

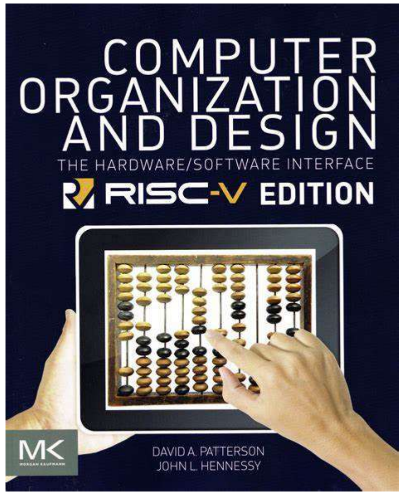
Lectures: Two 100-minute lectures a week

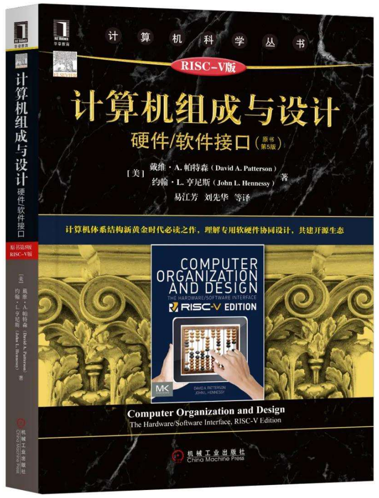
Textbook: Computer Architecture: A Quantitative Approach
6th Edition(2019) 中文版（2022.9月）

Prerequisite: Digital System Structure and Design





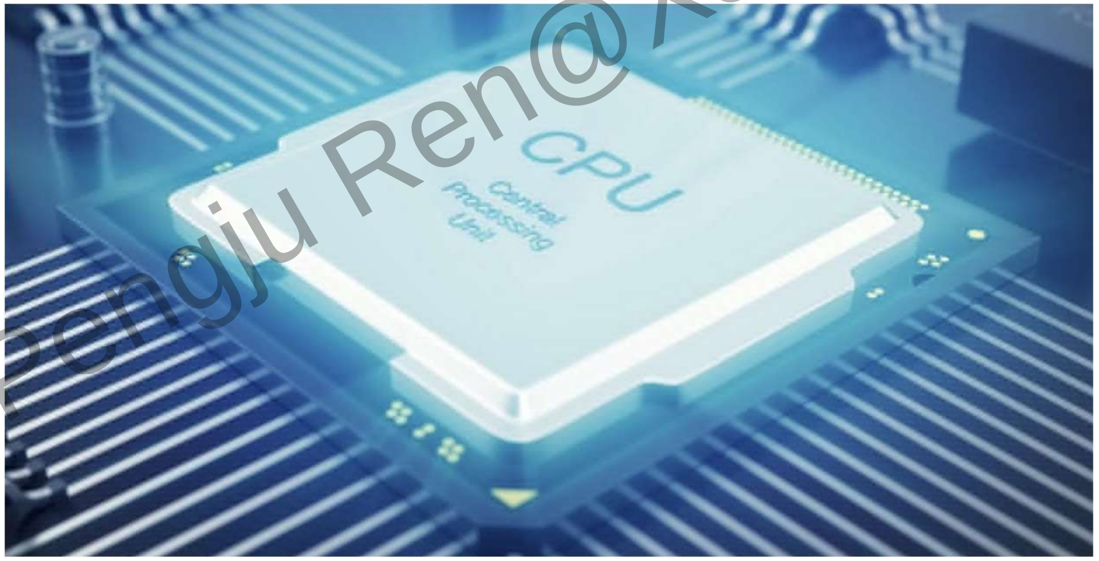

COMPUTER ORGANIZATION AND DESIGN
THE HARDWARE/SOFTWARE INTERFACE
RISC-V EDITION
DAVID A. PATTERSON
JOHN L. HENNESSY
MK



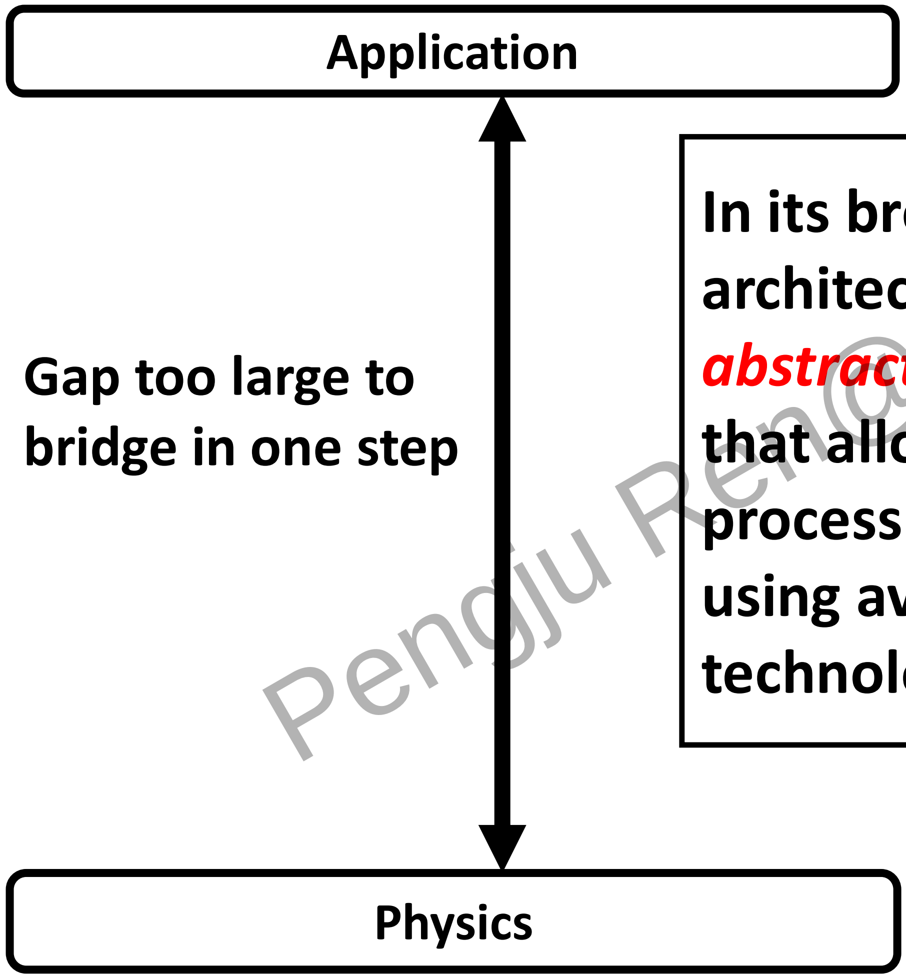
Preface

“The most beautiful thing we can experience is the mysterious. It is the source of all true art and Science.”

---Albert Einstein, What I believe, 1930

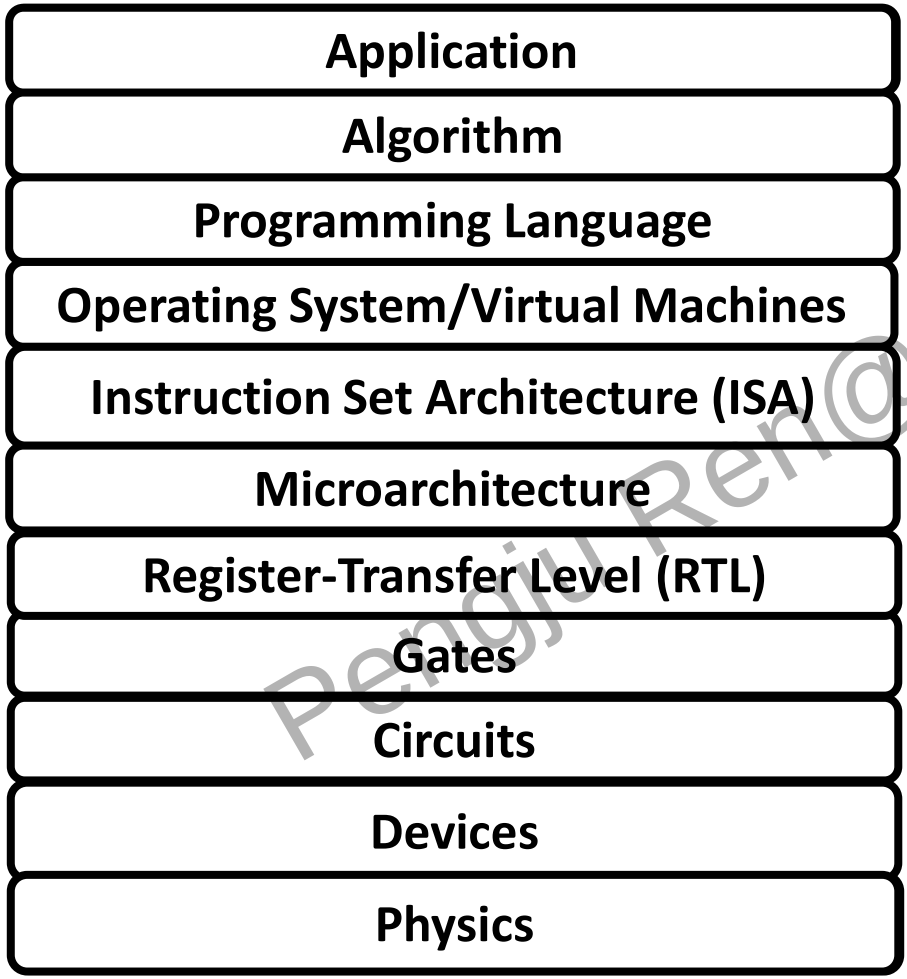


What is Computer Architecture

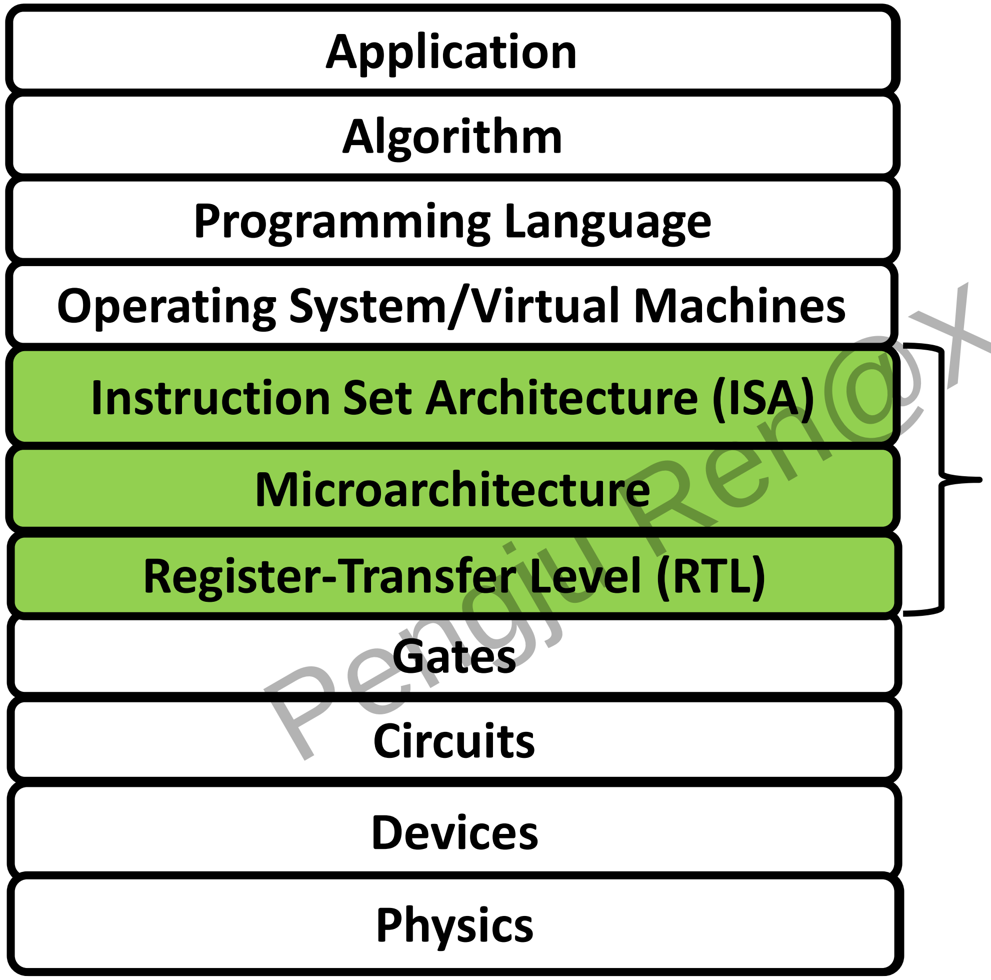


In its broadest definition, computer architecture is the *design of the abstraction/Implementation layers* that allow us to implement information processing *applications* efficiently using available manufacturing technologies.

What is Computer Architecture

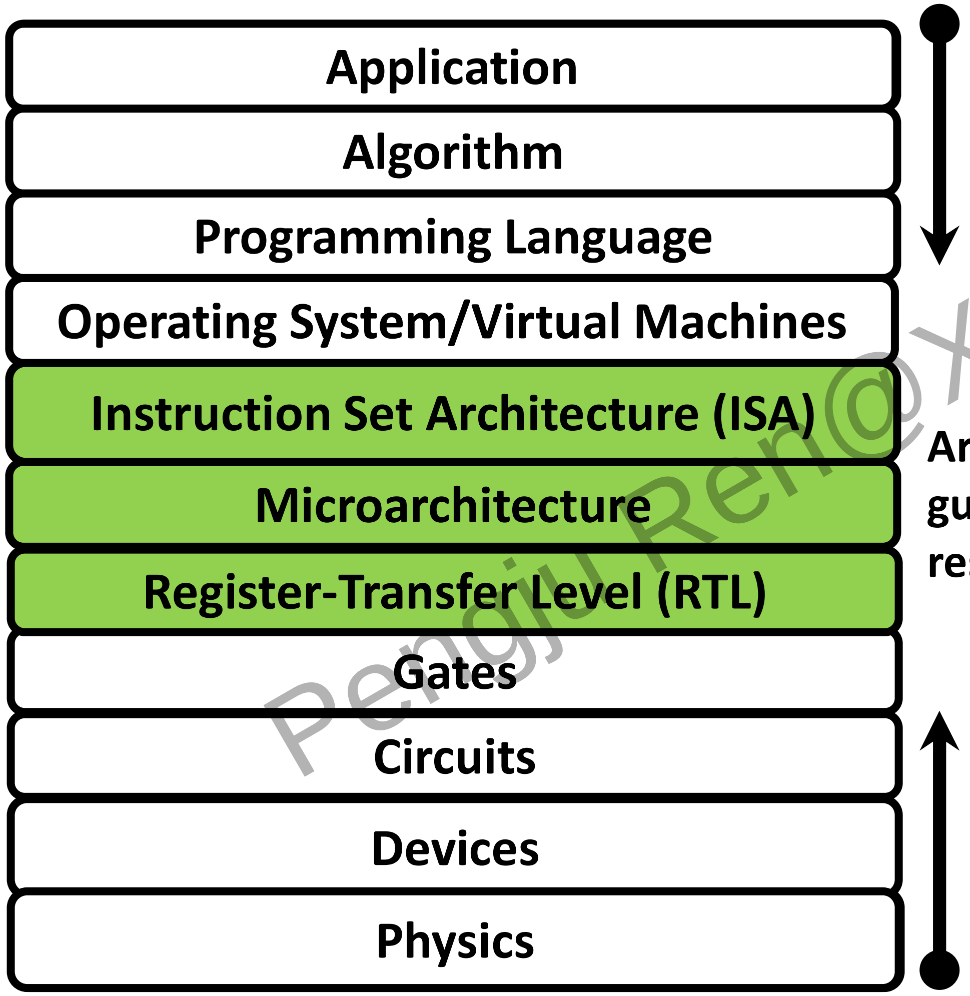

Application
Algorithm
Programming Language
Operating System/Virtual Machines
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Gates
Circuits
Devices
Physics

What is Computer Architecture


Application
Algorithm
Programming Language
Operating System/Virtual Machines
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Gates
Circuits
Devices
Physics

This course will start you thinking about designing and analyzing the underlying hardware computer system

What is Computer Architecture


Application
Algorithm
Programming Language
Operating System/Virtual Machines
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Gates
Circuits
Devices
Physics

Application Requirement:

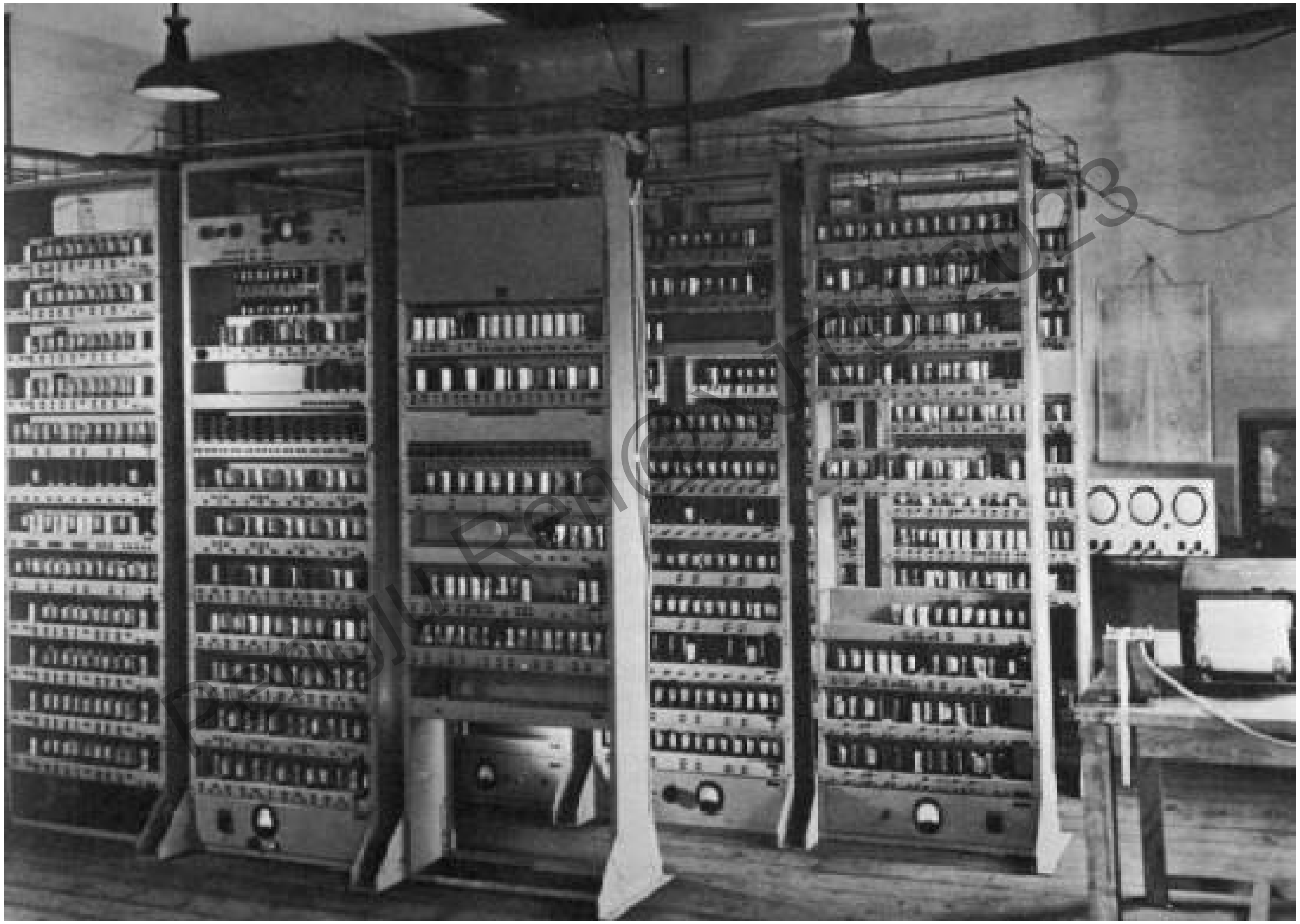
- Suggest how to improve architecture
- Provide revenue to fund development

Architecture provides feedback to guide application and technology research directions

Technology Constraints:

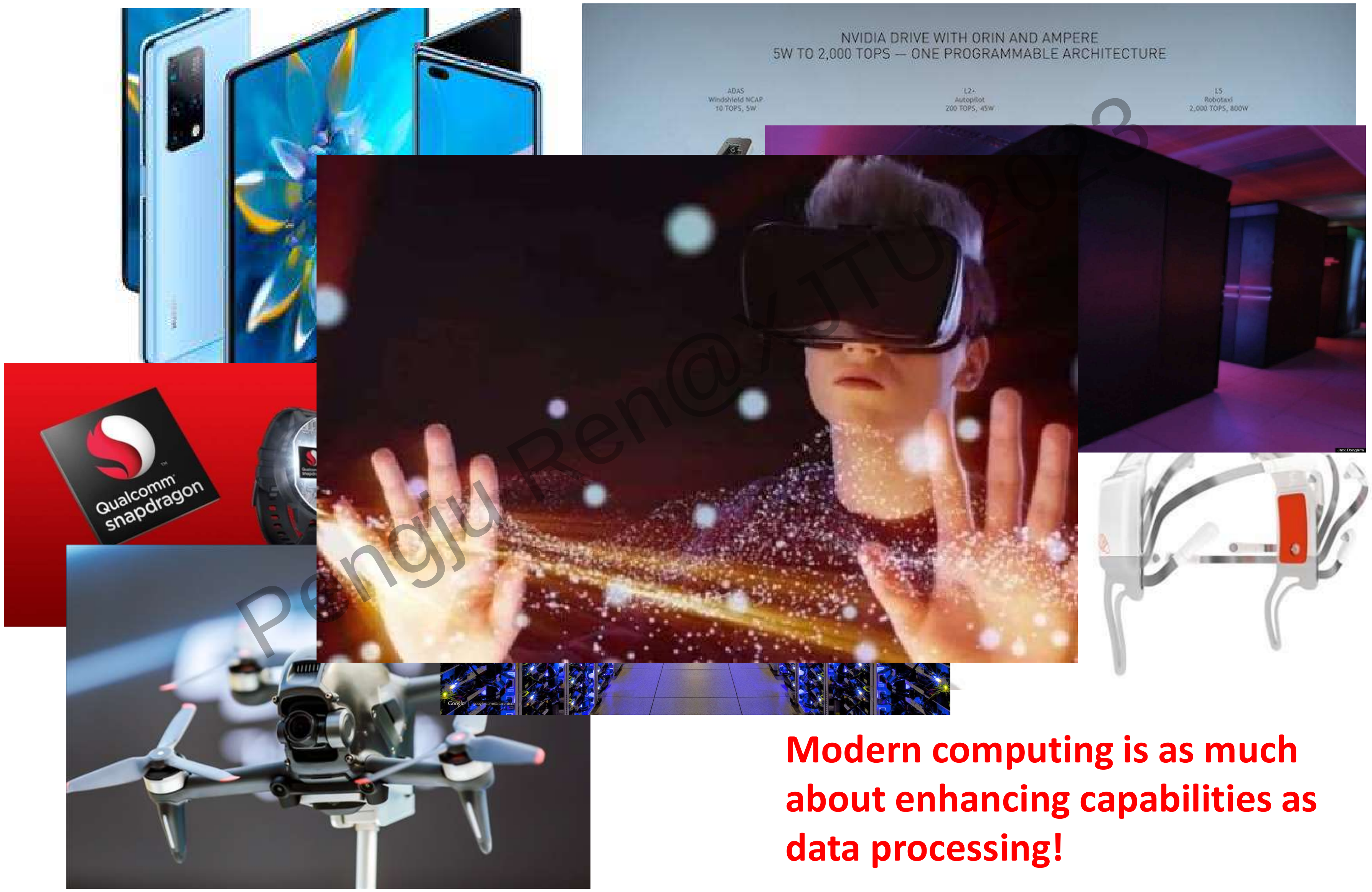
- Restrict what can be done efficiently
- New technologies make new arch possible

Computing Devices Then…



EDSAC, University of Cambridge, UK, 1949

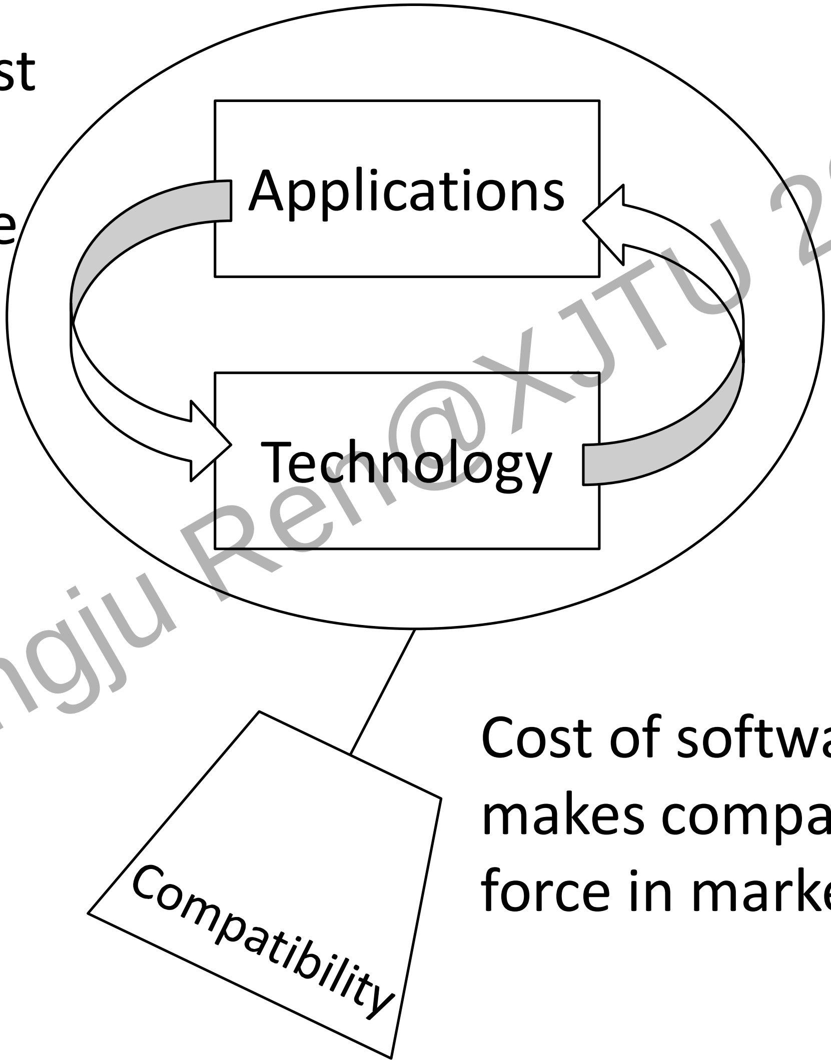
Computing Devices Now


NVIDIA DRIVE WITH ORIN AND AMPERE
5W TO 2,000 TOPS — ONE PROGRAMMABLE ARCHITECTURE
ADAS Windshield NCAP 10 TOPS, 5W
L2+ Autopilot 200 TOPS, 45W
L5 Robotaxi 2,000 TOPS, 800W
Qualcomm snapdragon

Modern computing is as much about enhancing capabilities as data processing!

Architecture continually changing

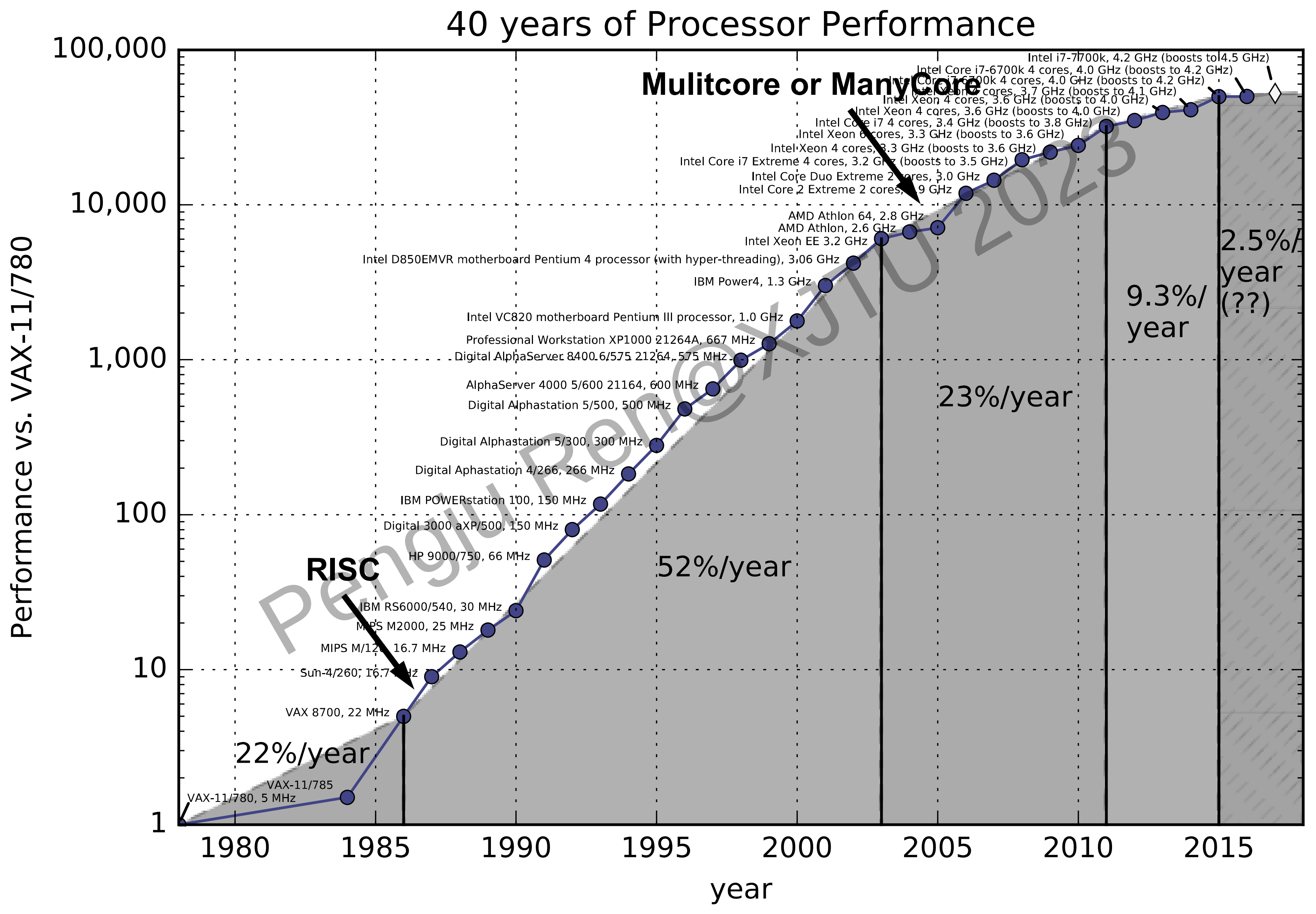
Applications suggest how to improve technology, provide revenue to fund development



Improved technologies make new applications possible

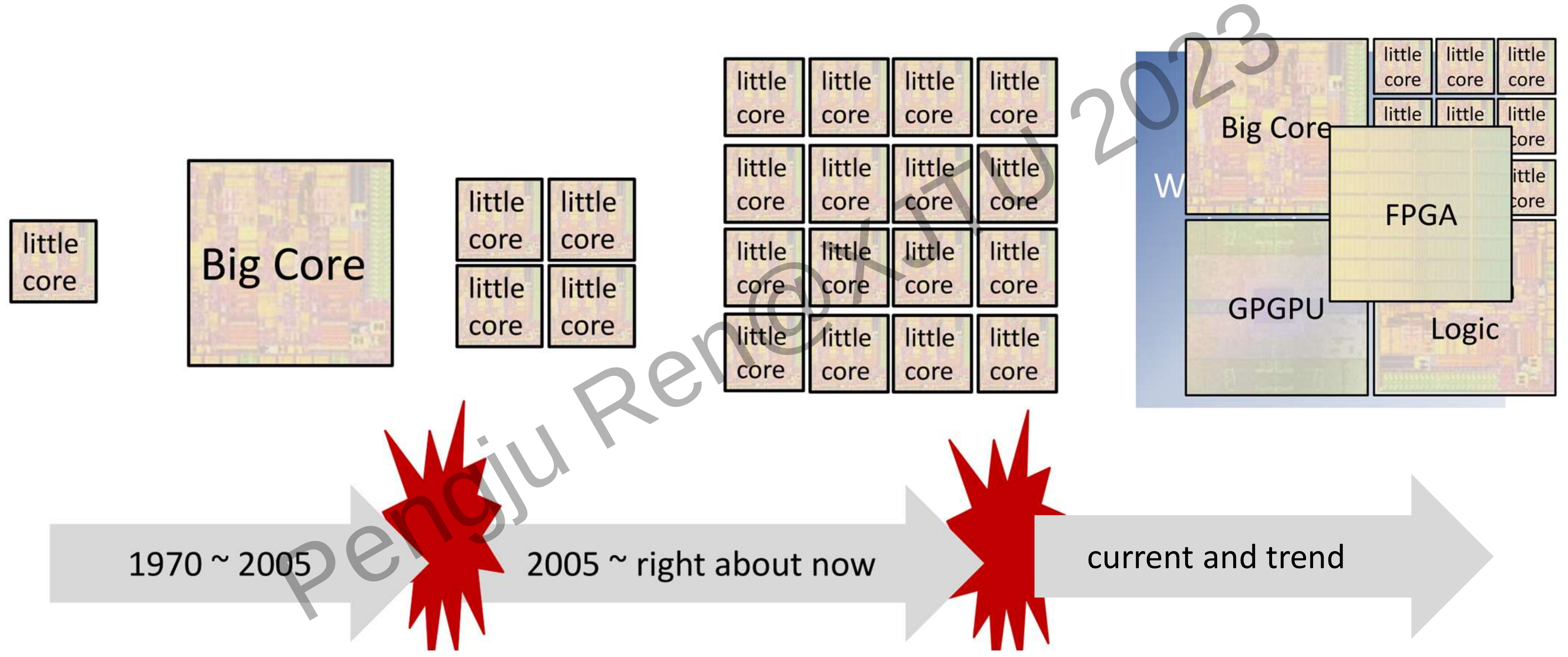
Cost of software development makes compatibility a major force in market

Single-Thread(Sequential) Processor Performance



[Hennessy & Patterson, 2017]

Moore's Law Scaling with Cores



Global Semiconductor Market

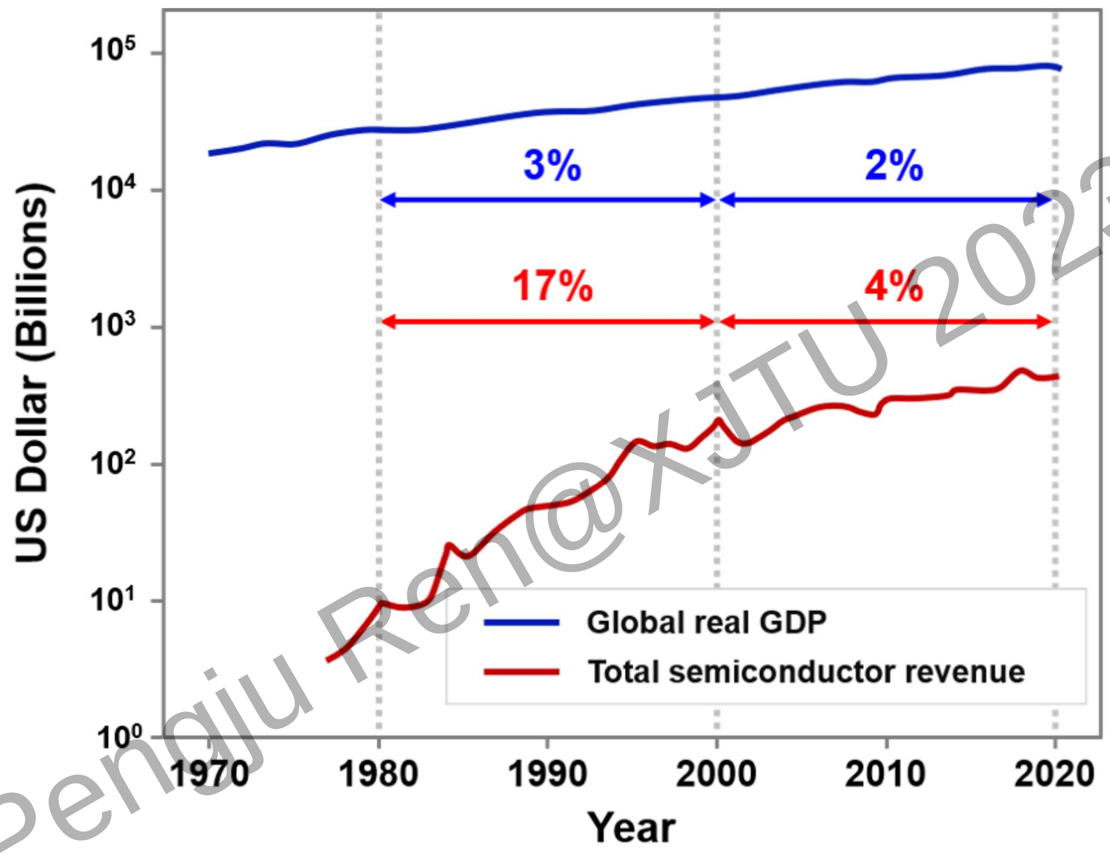
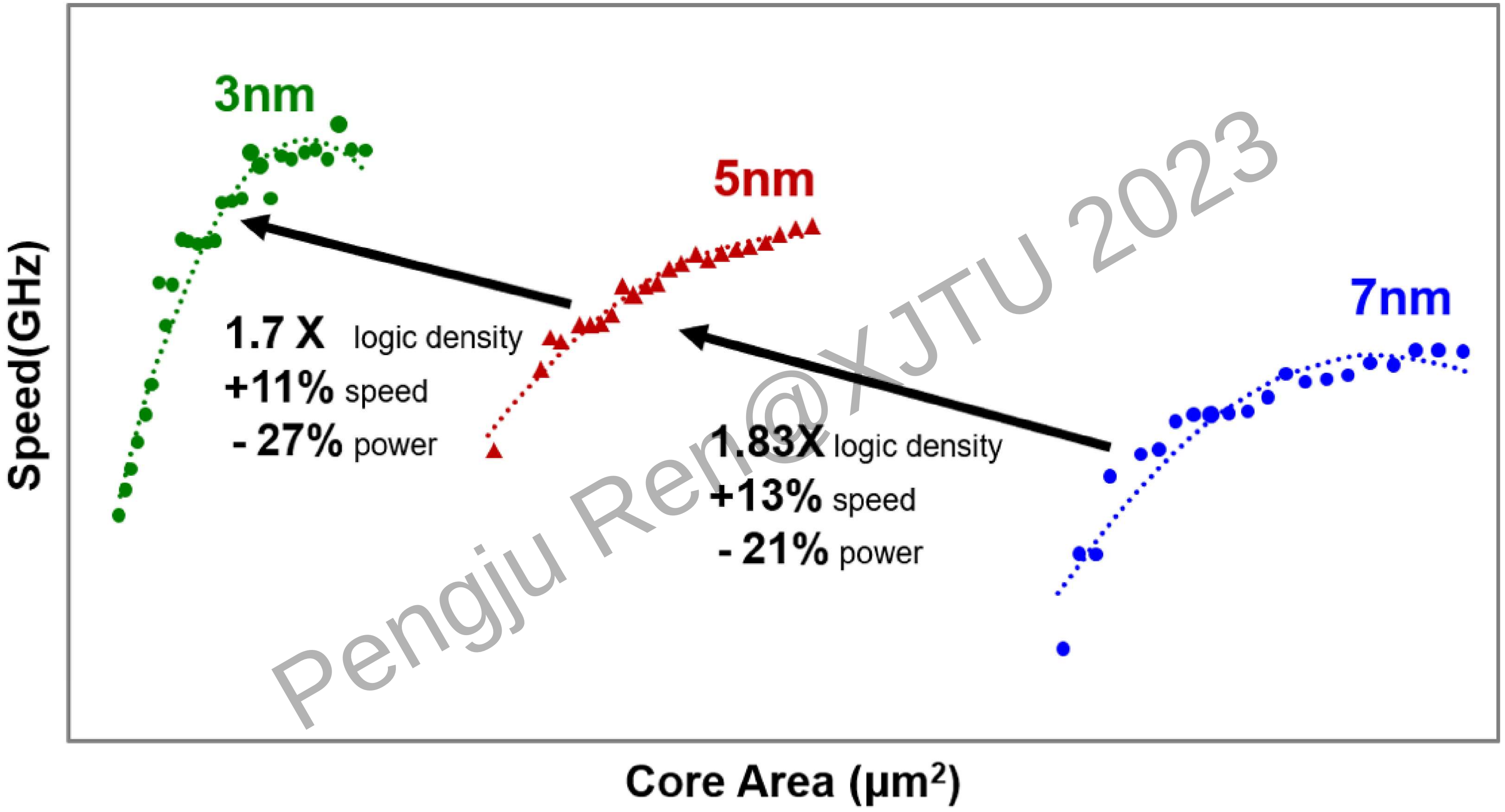


Figure 1.1.1: (a) The growth rate of revenue of semiconductors parallels those of the gross world product (GWP) for the past 20 years. After the initial fast growth period around the 1990s, worldwide semiconductor sales grow at a similar rate as the gross world product.

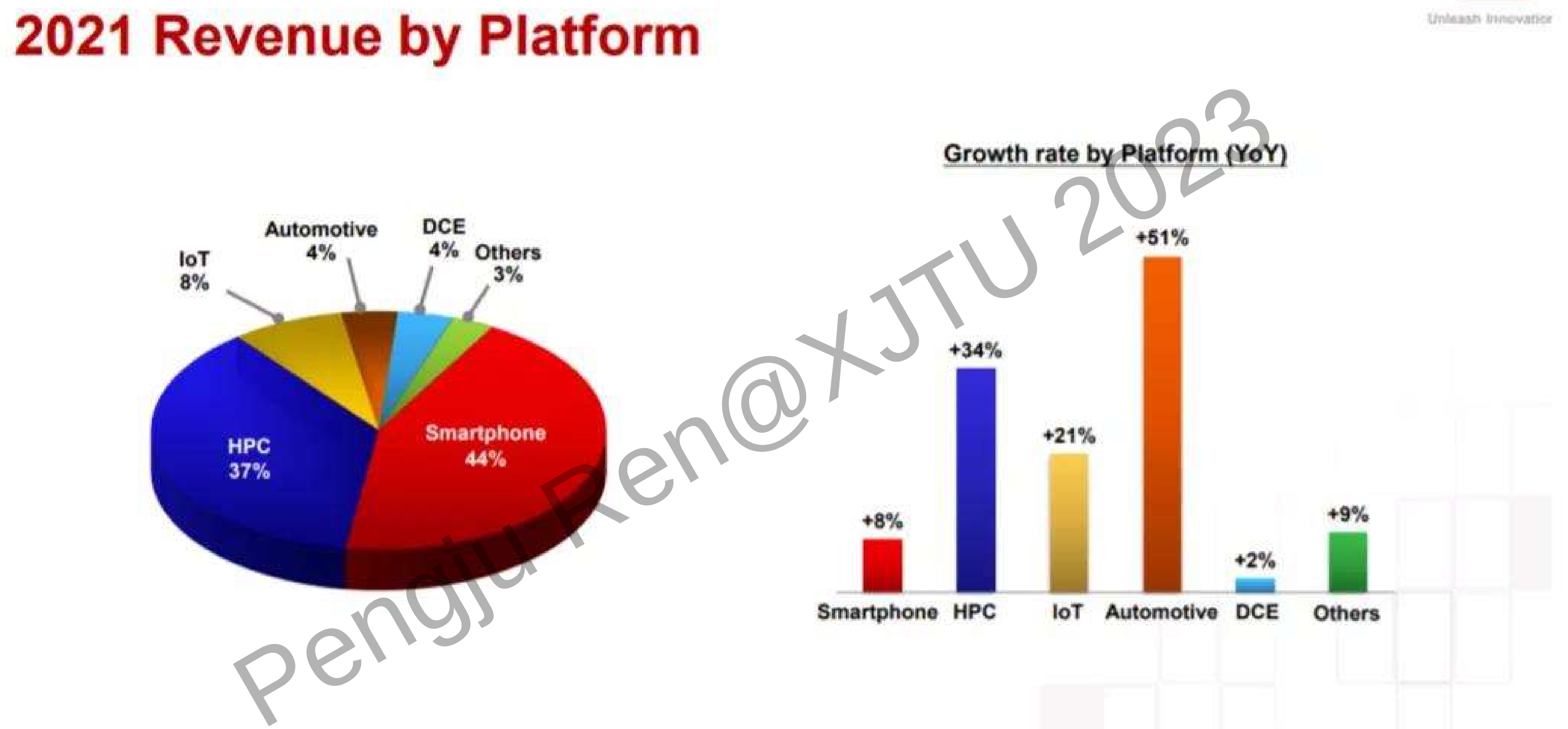
The global semiconductor market is estimated at **$450 billion** USD in revenue for 2020. Products using these semiconductors represent global revenues of **$2 trillion** USD, or **around 3.5% of global gross domestic product (GDP)**

Advanced Tech nodes continue provide value



Steady progress in two-dimensional transistor scaling and a variety of device enhancement techniques have sustained energy-efficiency improvement and device density gains from one technology generation to the next

TSMC（2022.1.13）



全年收入+24.9%，达到570亿美元（毛利53-55%, 净利42-44%【500强No.1】）

HPC、IoT 和 Automotive 分别实现 34%、21% 和 51% 的强劲增长

Huawei 2021销售收入约900亿美元（净利率10%左右)

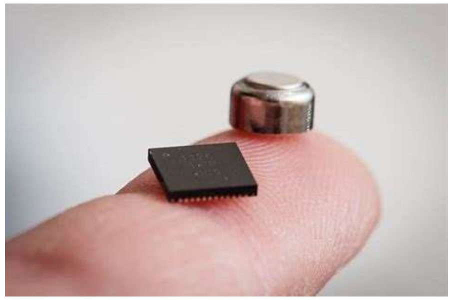
Upheaval in Computer Design

- Most of last 50 years, Moore’s Law ruled
  - Technology scaling allowed continual performance/energy improvements without changing software model
- Last decade, technology scaling slowed/stopped
  - Dennard (voltage) scaling over (supply voltage ~fixed)
  - Moore’s Law (cost/transistor) over?
  - No competitive replacement for CMOS anytime soon
  - Energy efficiency constrains everything
- No “free lunch” for software developers, must consider:
  - Parallel systems
  - Heterogeneous systems

Today’s Dominant Target Systems

- Mobile (smartphone/tablet)
  - >1 billion sold/year
  - Market dominated by ARM-ISA-compatible general-purpose processor in system-on-a-chip (SoC)
  - Plus sea of custom accelerators (radio, image, video, graphics, audio, motion, location, security, etc.)

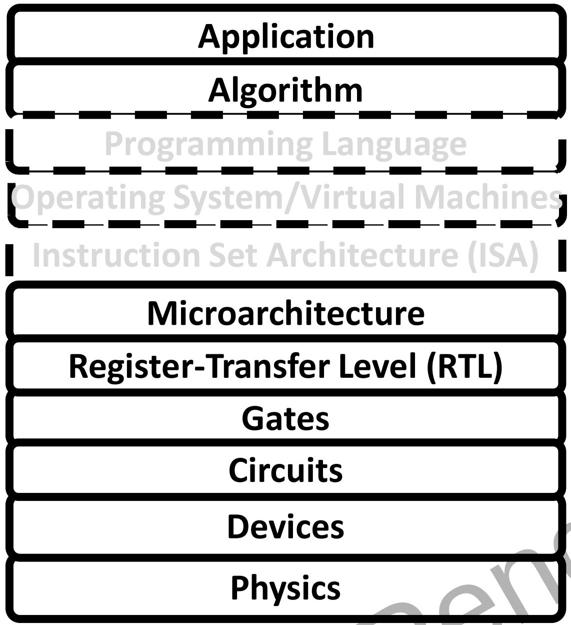
- Warehouse-Scale Computers (WSCs)
  - 100,000’s cores per warehouse
  - Market dominated by x86-compatible server chips
  - Dedicated apps, plus cloud hosting of virtual machines
  - Now seeing increasing use of GPUs, FPGAs, custom hardware to accelerate workloads



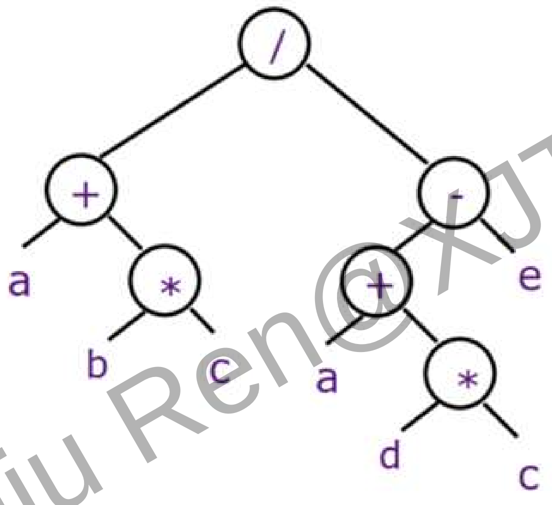
- Embedded computing
  - Wired/wireless network infrastructure, printers
  - Consumer TV/Music/Games/Automotive/Camera/MP3
  - Internet of Things!

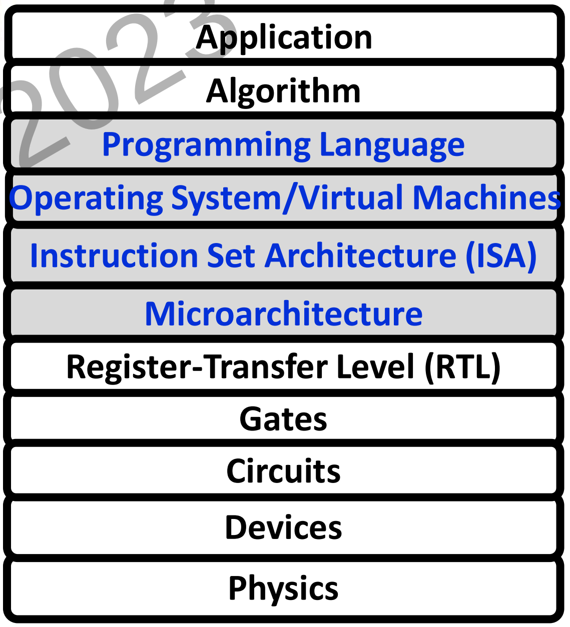
Evaluation of Expressions (ASIC v.s Processor)

App: Polynomial operation


Application
Algorithm
Programming Language
Operating System/Virtual Machines
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Gates
Circuits
Devices
Physics

(a + b * c) / (a + d * c - e)




Application
Algorithm
Programming Language
Operating System/Virtual Machines
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Gates
Circuits
Devices
Physics

Application Specific Design

(High Efficiency, Dedicated)

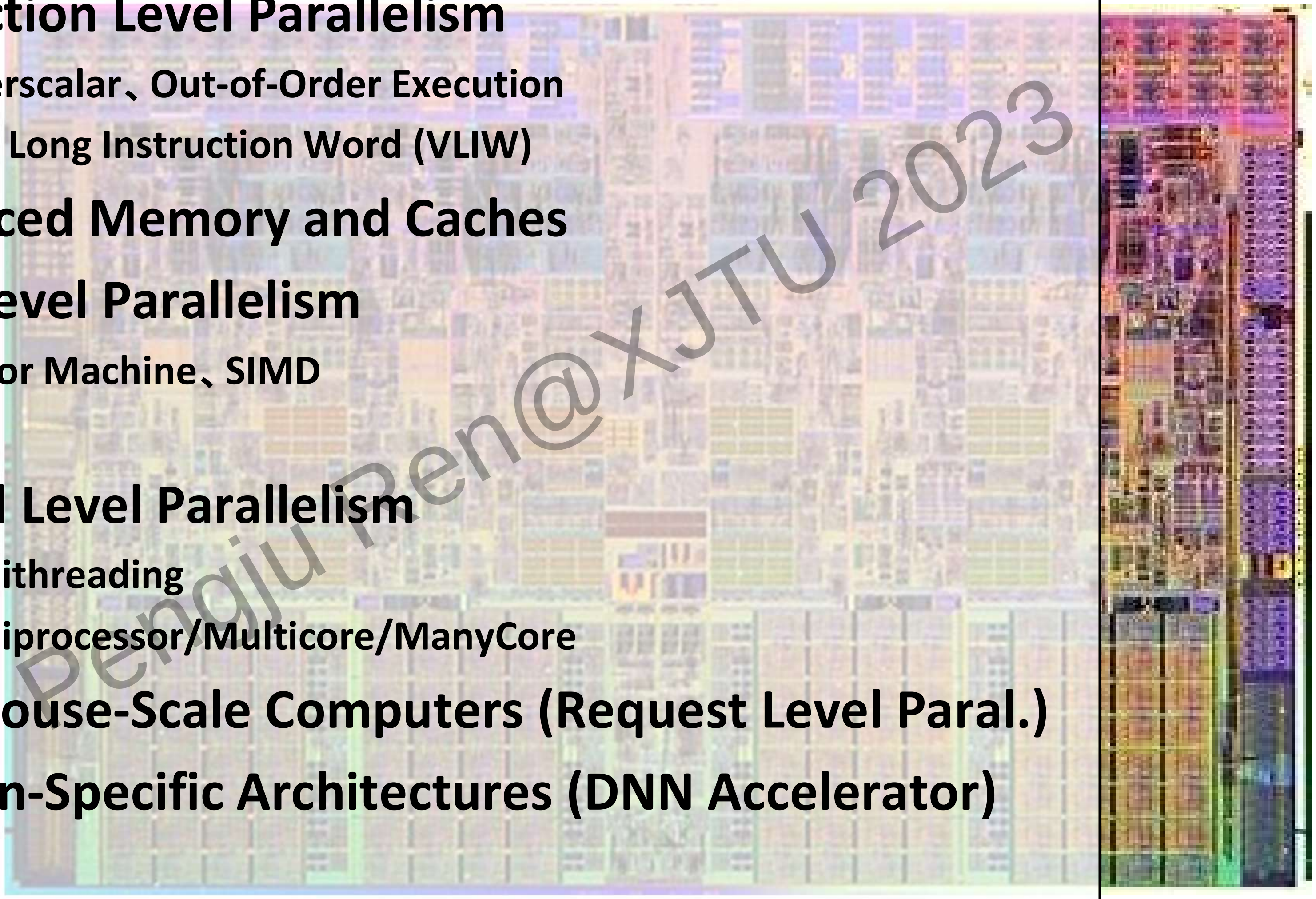
大四课程：人工智能芯片设计导论

General Design

(Programable, Flexible)

本课程：计算机体系结构

Course Content Computer Architecture



- **Instruction Level Parallelism**
  - Superscalar、Out-of-Order Execution
  - Very Long Instruction Word (VLIW)
- **Advanced Memory and Caches**
- **Data Level Parallelism**
  - Vector Machine、SIMD
  - GPU
- **Thread Level Parallelism**
  - Multithreading
  - Multiprocessor/Multicore/ManyCore
- **Warehouse-Scale Computers (Request Level Paral.)**
- **Domain-Specific Architectures (DNN Accelerator)**

Intel Nehalem Processor, Core i7

Architecture vs. Microarchitecture

“Architecture”/Instruction Set Architecture:

- Programmer visible state (Memory & Register)
- Operations (Instructions and how they work)
- Execution Semantics (interrupts)
- Input/Output
- Data Types/Sizes

Microarchitecture/Organization:

- Tradeoffs on how to implement ISA for some metric (Speed, Energy, Cost)
- Examples: Pipeline depth, number of pipelines, cache size, silicon area, peak power, execution ordering, bus widths, ALU widths, etc.

Same Architecture Diff Micro-Architecture

AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

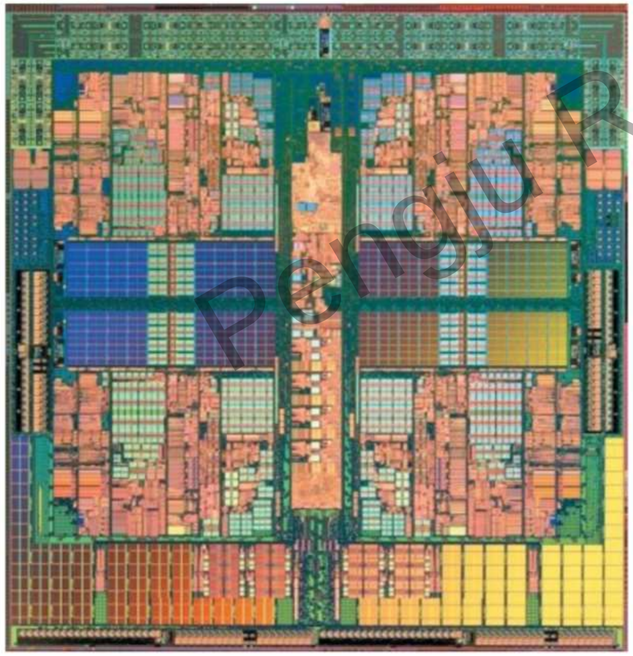


Image Credit: AMD

Intel Atom

- X86 Instruction Set
- Single Core
- 2W
- Decode 2 Instructions/Cycle/Core
- 32KB L1 I Cache, 24KB L1 D Cache
- 512KB L2 Cache
- In-order
- 1.6GHz

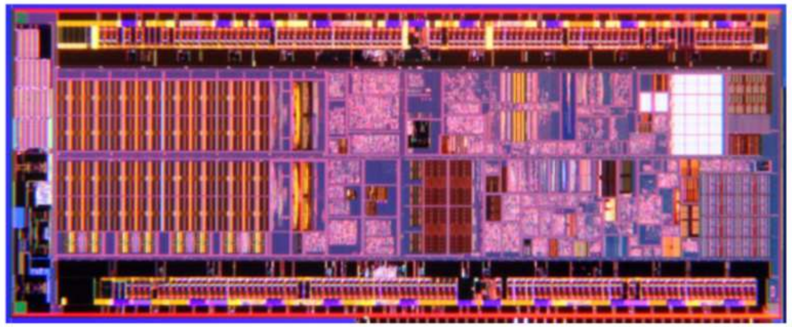


Image Credit: Intel

Diff Architecture Diff Micro-Architecture

AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

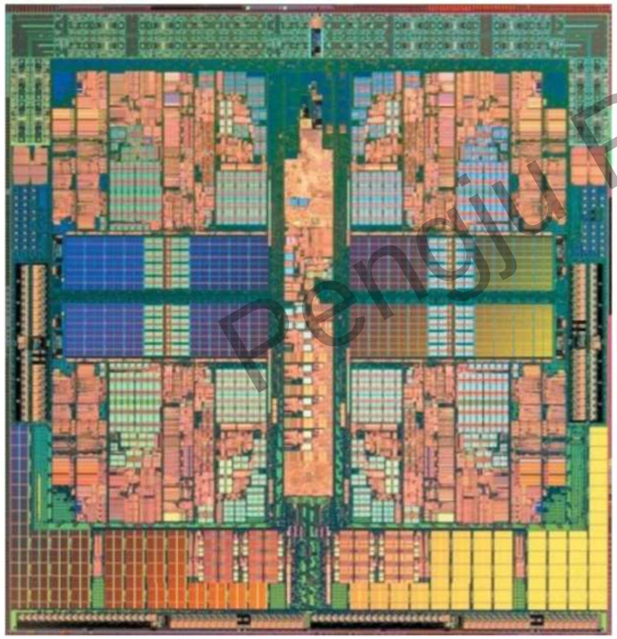


Image Credit: AMD

IBM POWER7

- Power Instruction Set
- Eight Core
- 200W
- Decode 6 Instructions/Cycle/Core
- 32KB L1 I Cache, 32KB L1 D Cache
- 256KB L2 Cache
- Out-of-order
- 4.25GHz

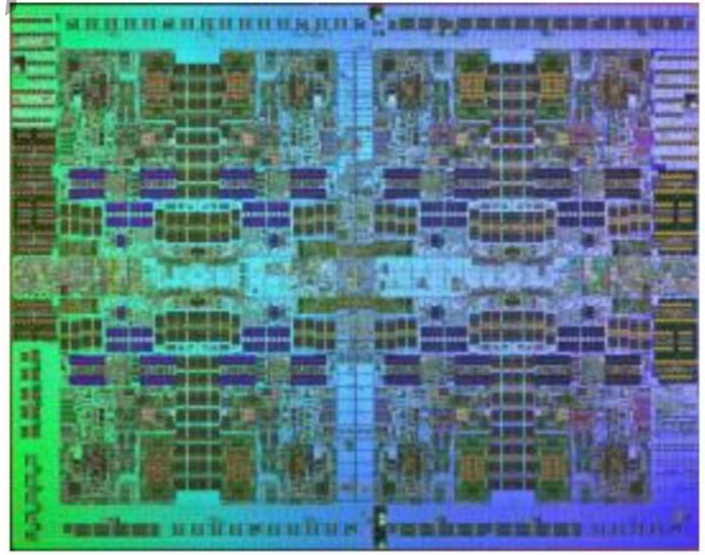
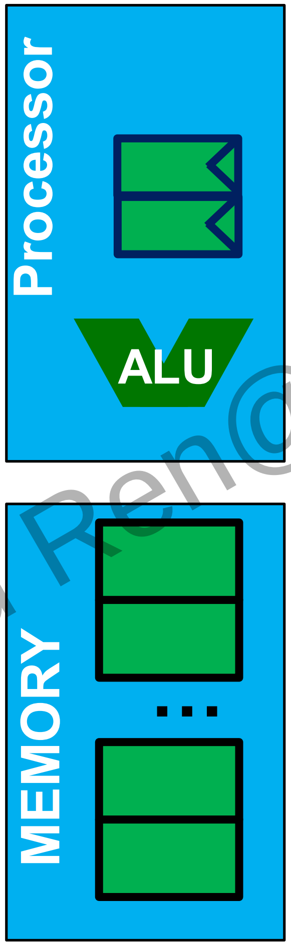
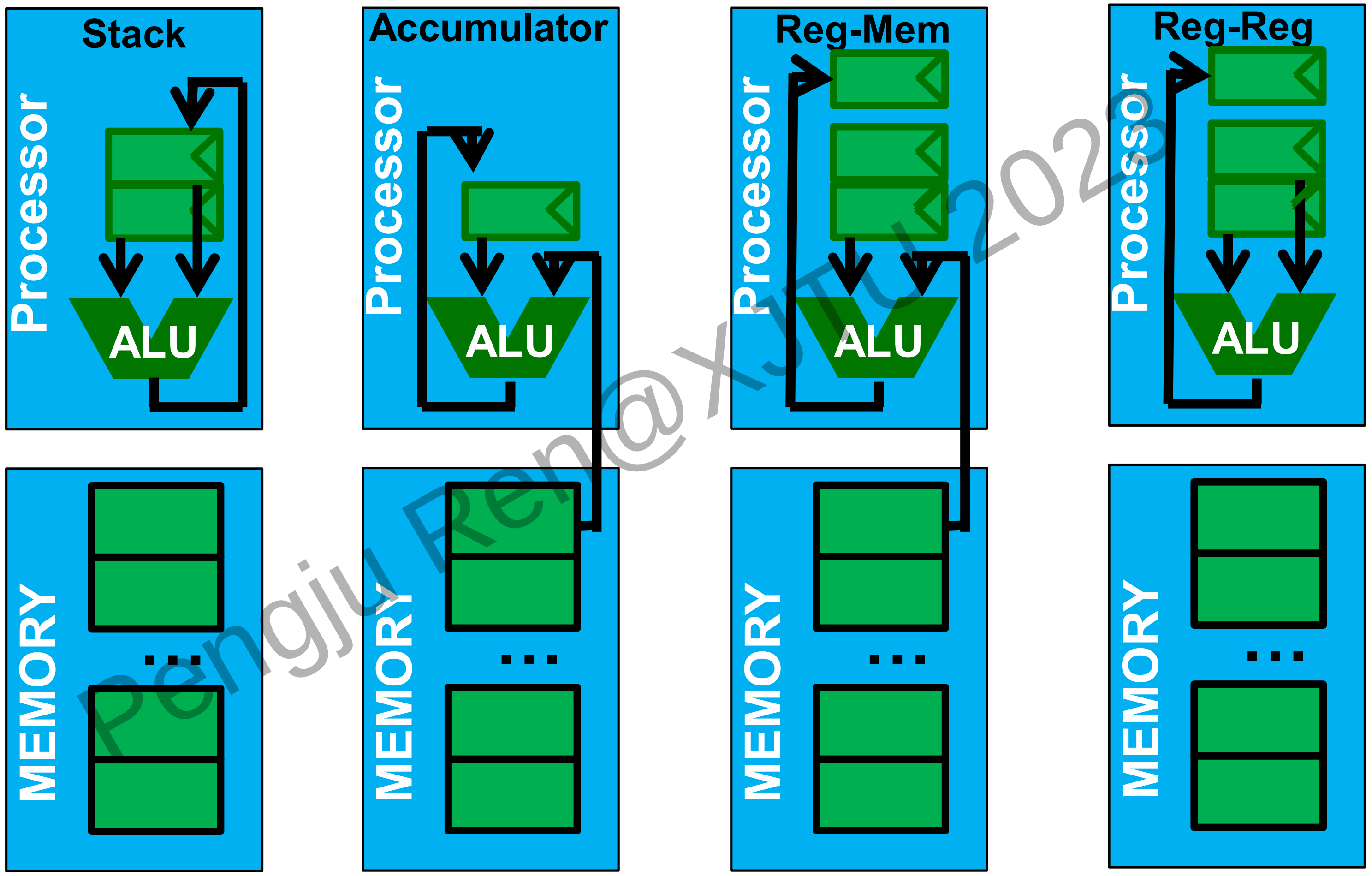


Image Credit: IBM
Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

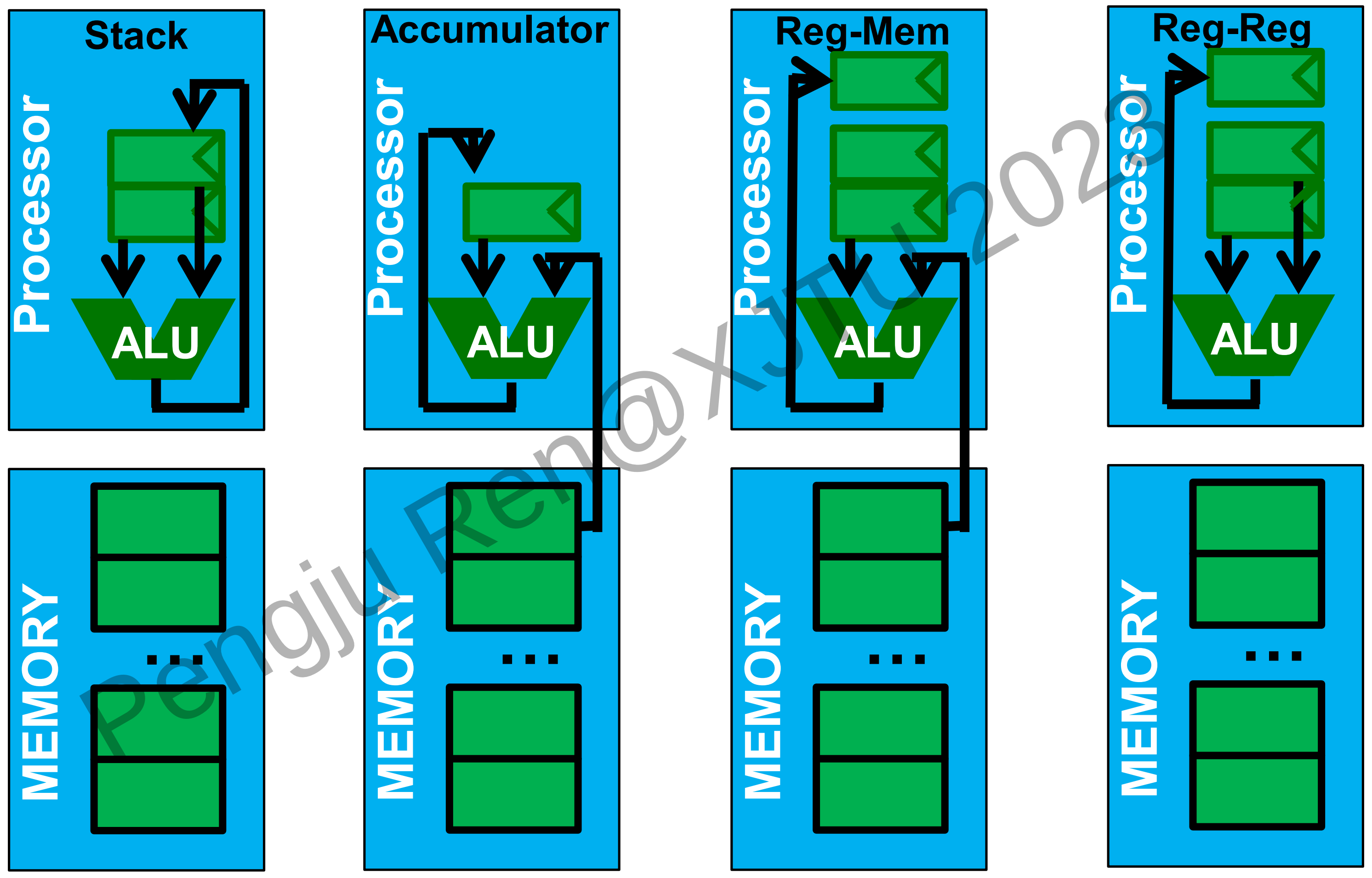
Where do Operands come from and Where do Results Go ?



Where do Operands come from and Where do Results Go ?

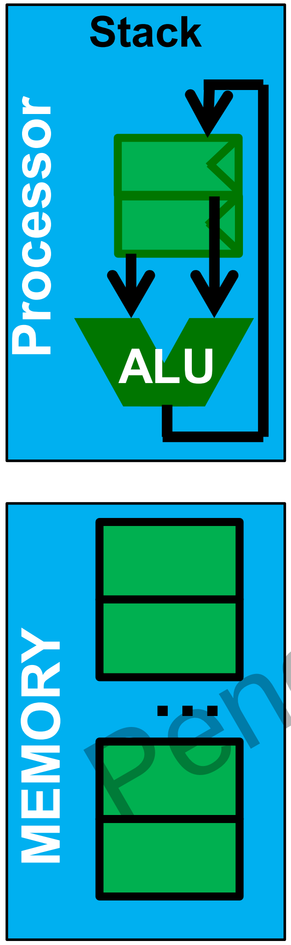


Where do Operands come from and Where do Results Go ?



Number Explicitly Named Operands	0	1	2 or 3	2 or 3

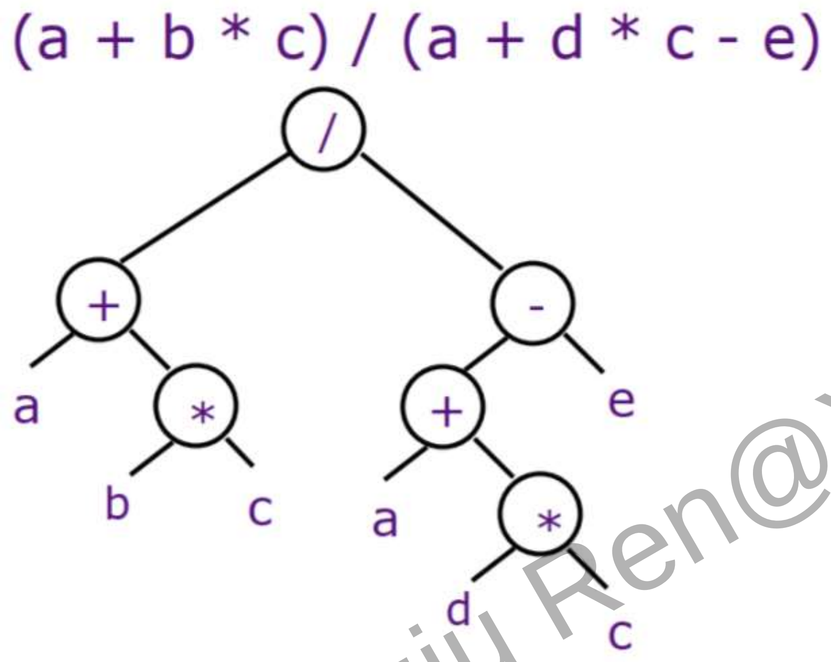
Stack-Based Instruction Set Architecture(ISA)



Burrough's B5000 (1960)
- Burrough's B6700
- HP 3000
- ICL 2900
- Symbolics 3600
- Inmos Transputer

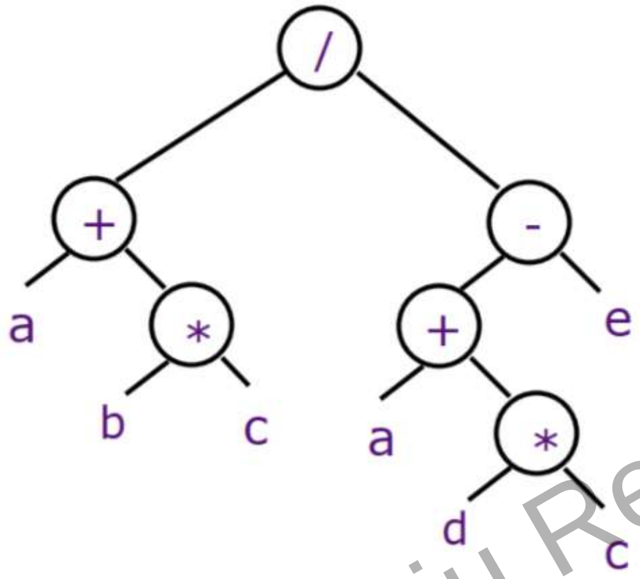
Modern
- Forth machines
- Java Virtual Machine
- Intel x87 Floating Point Unit

Evaluation of Expressions



Evaluation of Expressions

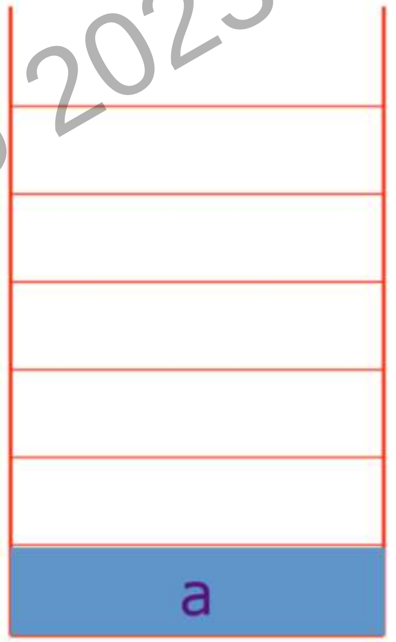
(a + b * c) / (a + d * c - e)



Reverse Polish

a b c * + a d c * + e - /

push a

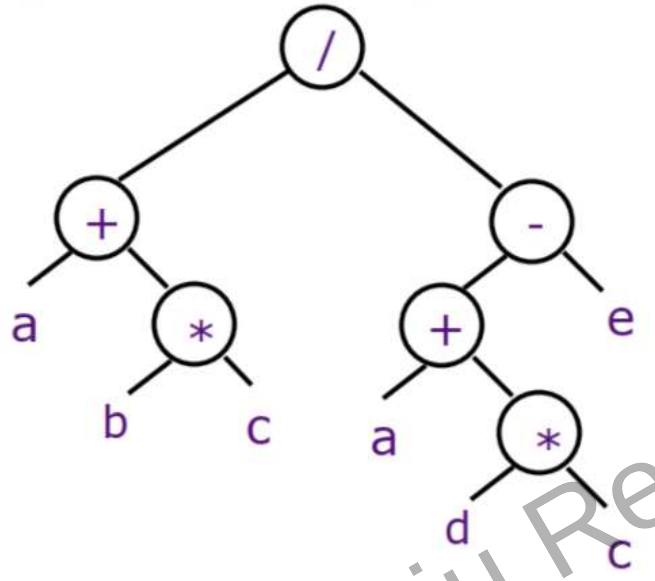


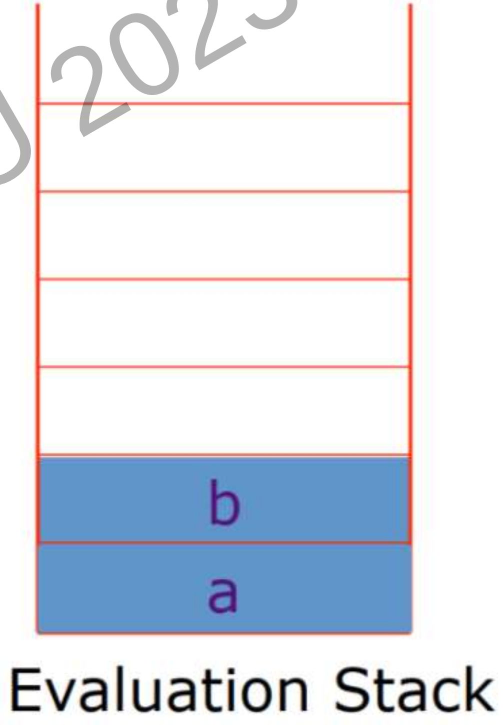
Evaluation Stack

52

Evaluation of Expressions

(a + b * c) / (a + d * c - e)





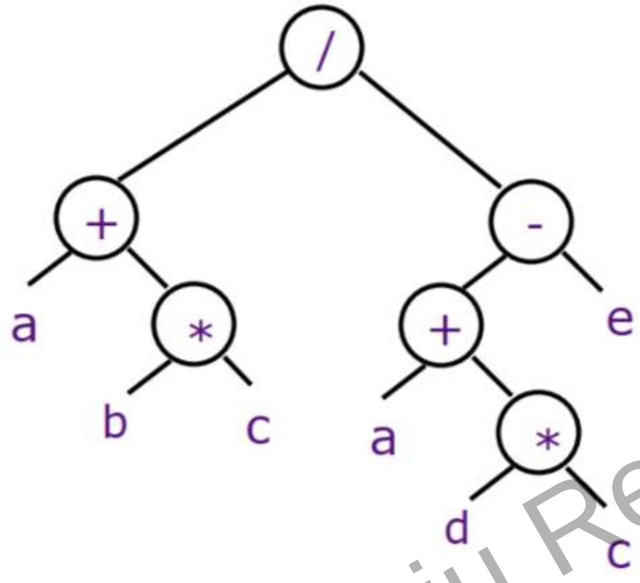
Reverse Polish

a b c * + a d c * + e - /

push b

Evaluation of Expressions

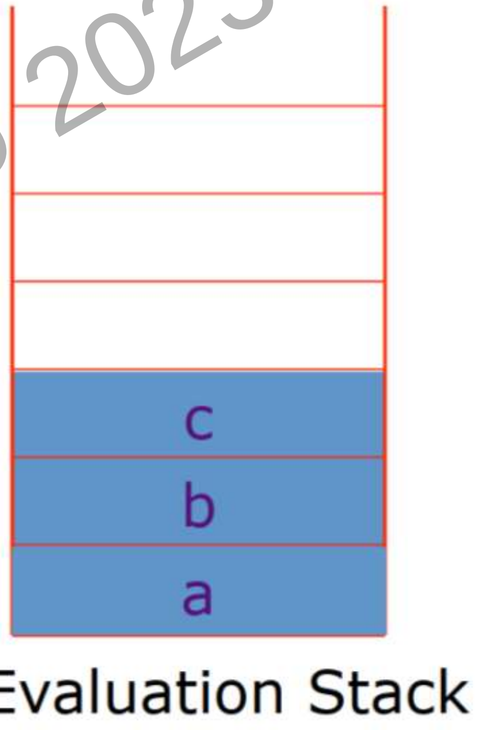
(a + b * c) / (a + d * c - e)



Reverse Polish

a b c * + a d c * + e - /

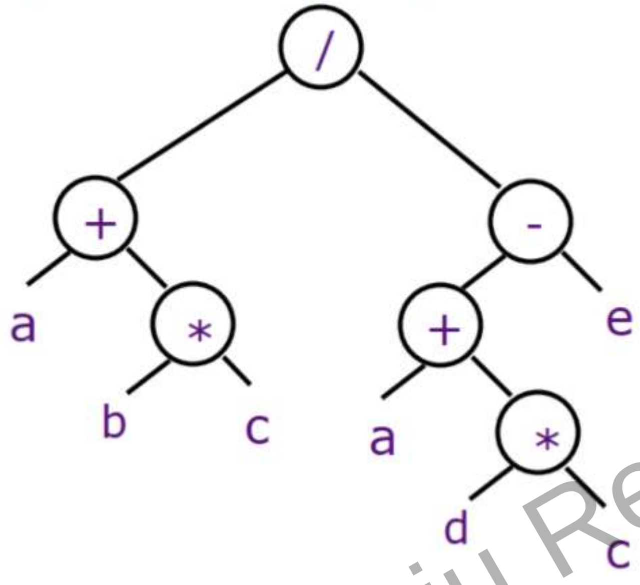
push c



56

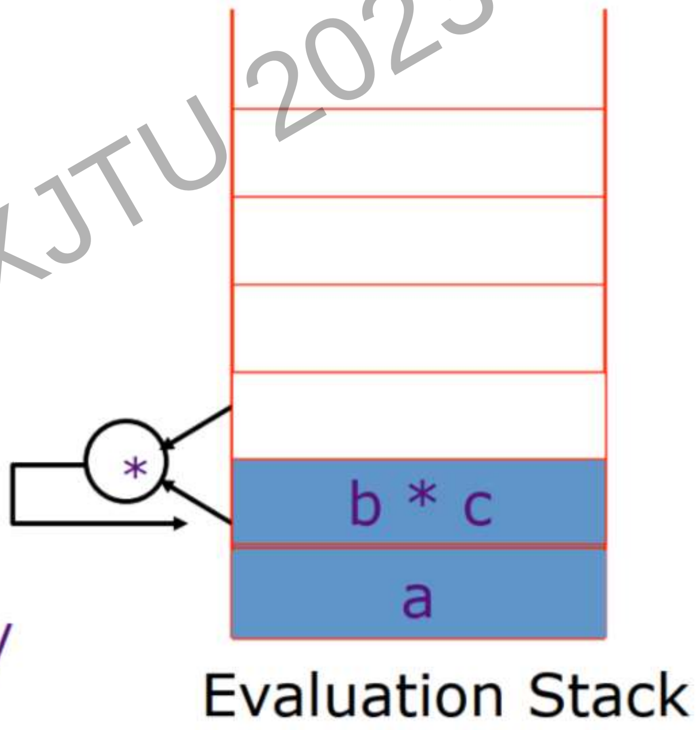
Evaluation of Expressions

(a + b * c) / (a + d * c - e)



Reverse Polish

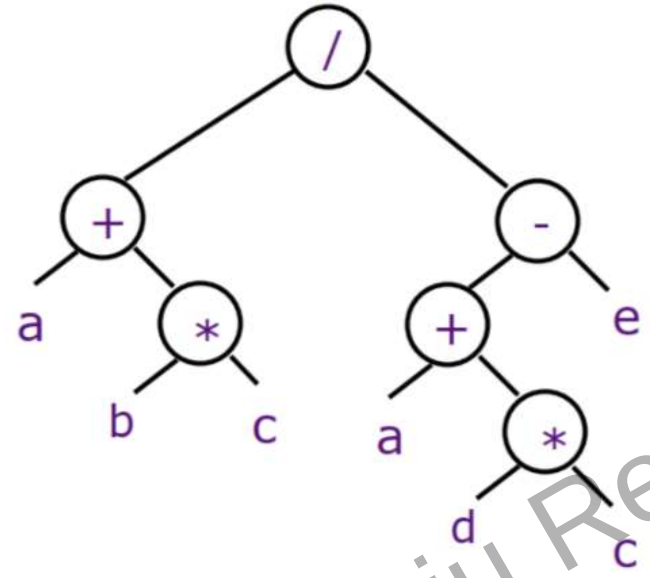
a b c * + a d c * + e - /



↑multiply

Evaluation of Expressions

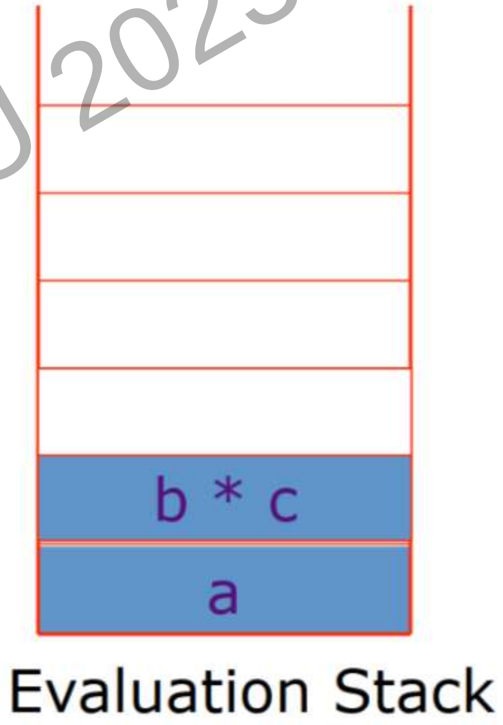
(a + b * c) / (a + d * c - e)



Reverse Polish

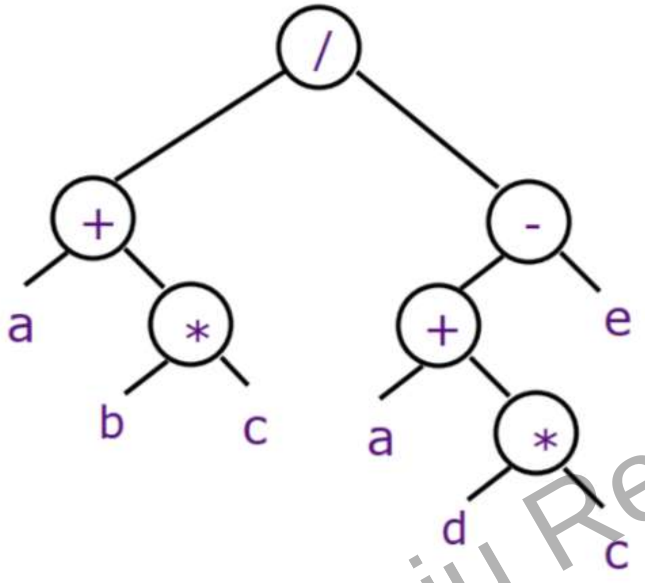
a b c * + a d c * + e - /

add



Evaluation of Expressions

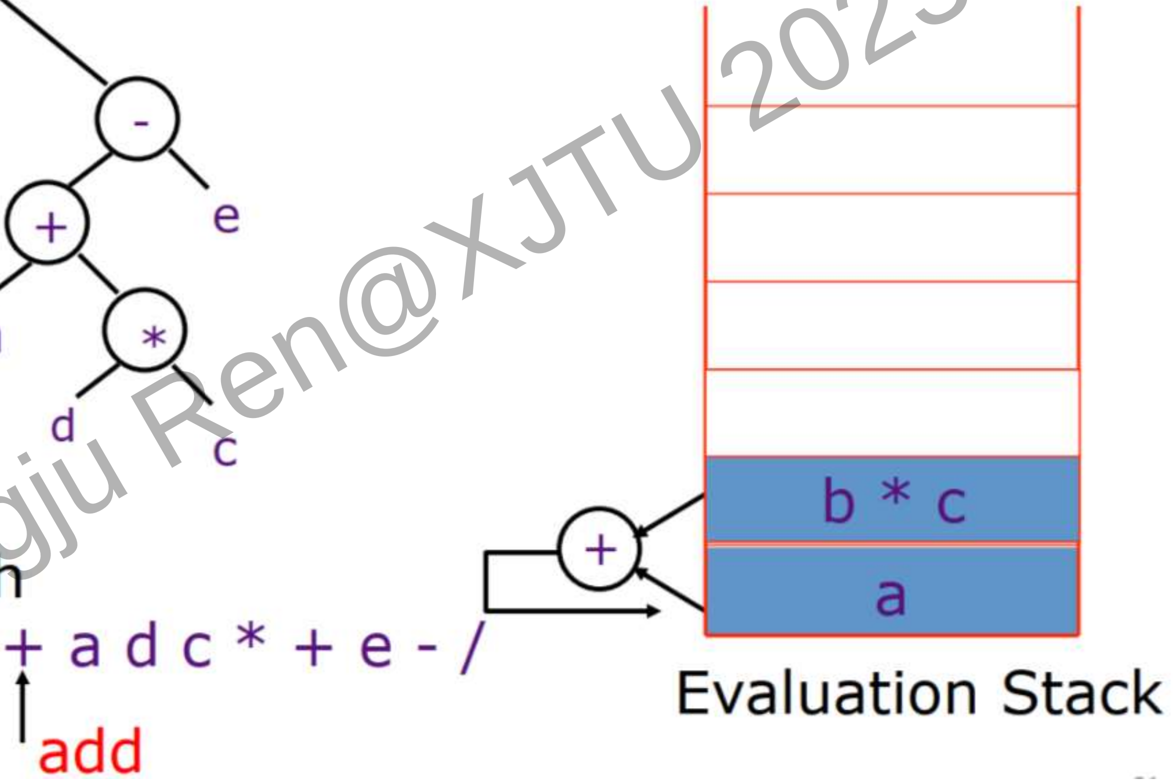
(a + b * c) / (a + d * c - e)



Reverse Polish

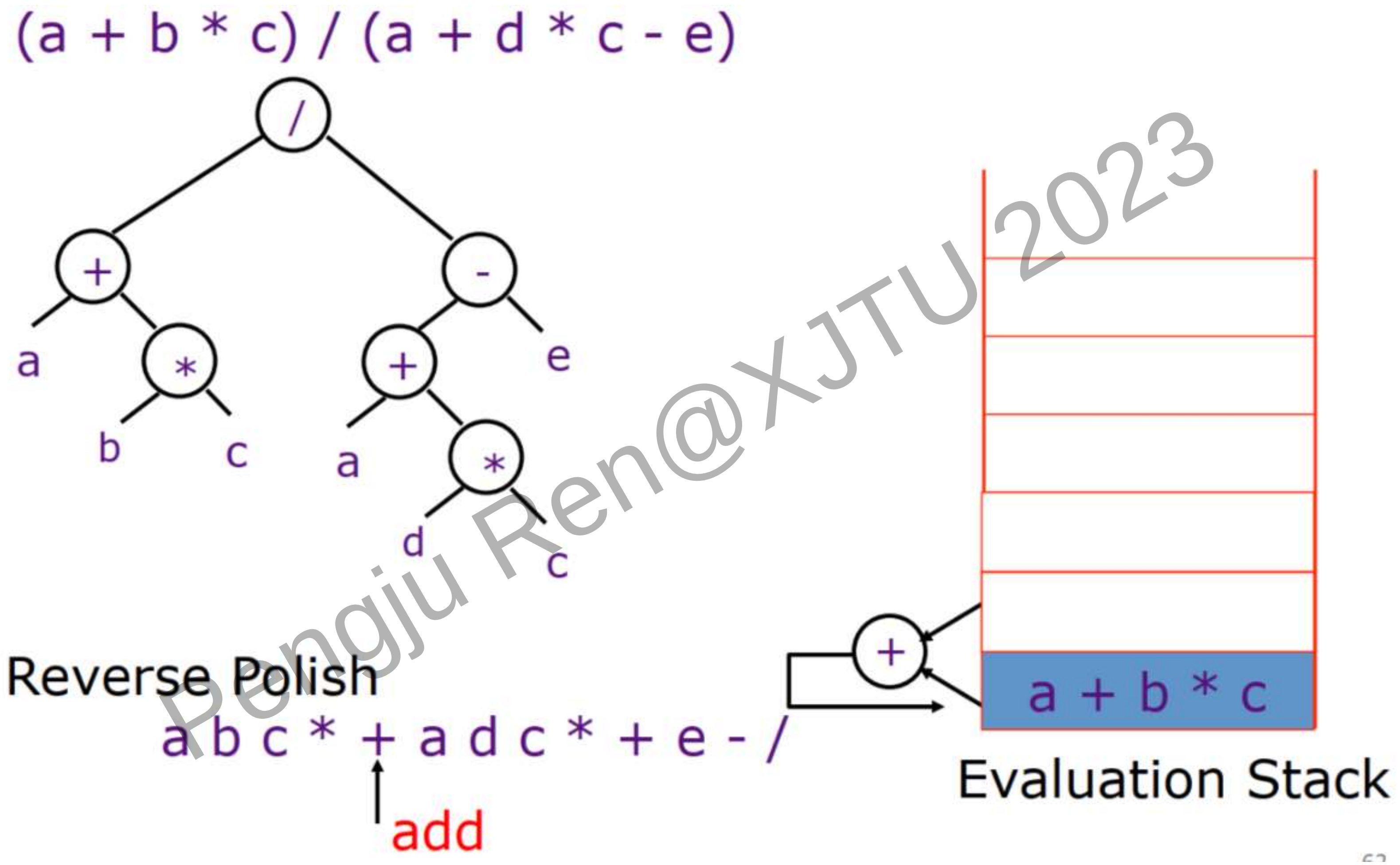
a b c * + a d c * + e - /

add



Evaluation Stack

Evaluation of Expressions



Hardware Organization of the Stack

Stack is part of the processor state
  ⇒ stack must be bounded and small
    ≈ number of Registers, not the size of main memory
Conceptually stack is unbounded
  ⇒a part of the stack is included in the
      processor state; the rest is kept in the main memory

Stack Operations/Implicit Memory References

Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

Each *push* operation ⇒ 1 memory reference

pop operation ⇒ 1 memory reference

Better performance by keeping the top N elements in registers, and memory references are made only when register stack overflows or underflows.

Stack Size and Memory References

a b c * + a d c * + e - /

program	*stack (size = 2)*	*memory refs*
push a	R0	a
push b	R0 R1	b
push c	R0 R1 R2	c, ss(a)
*	R0 R1	sf(a)
+	R0	
push a	R0 R1	a
push d	R0 R1 R2	d, ss(a+b*c)
push c	R0 R1 R2 R3	c, ss(a)
*	R0 R1 R2	sf(a)
+	R0 R1	sf(a+b*c)
push e	R0 R1 R2	e,ss(a+b*c)
-	R0 R1	sf(a+b*c)
/	R0	

Four Store and Fetch

Stack Size and Memory References

a b c * + a d c * + e - /

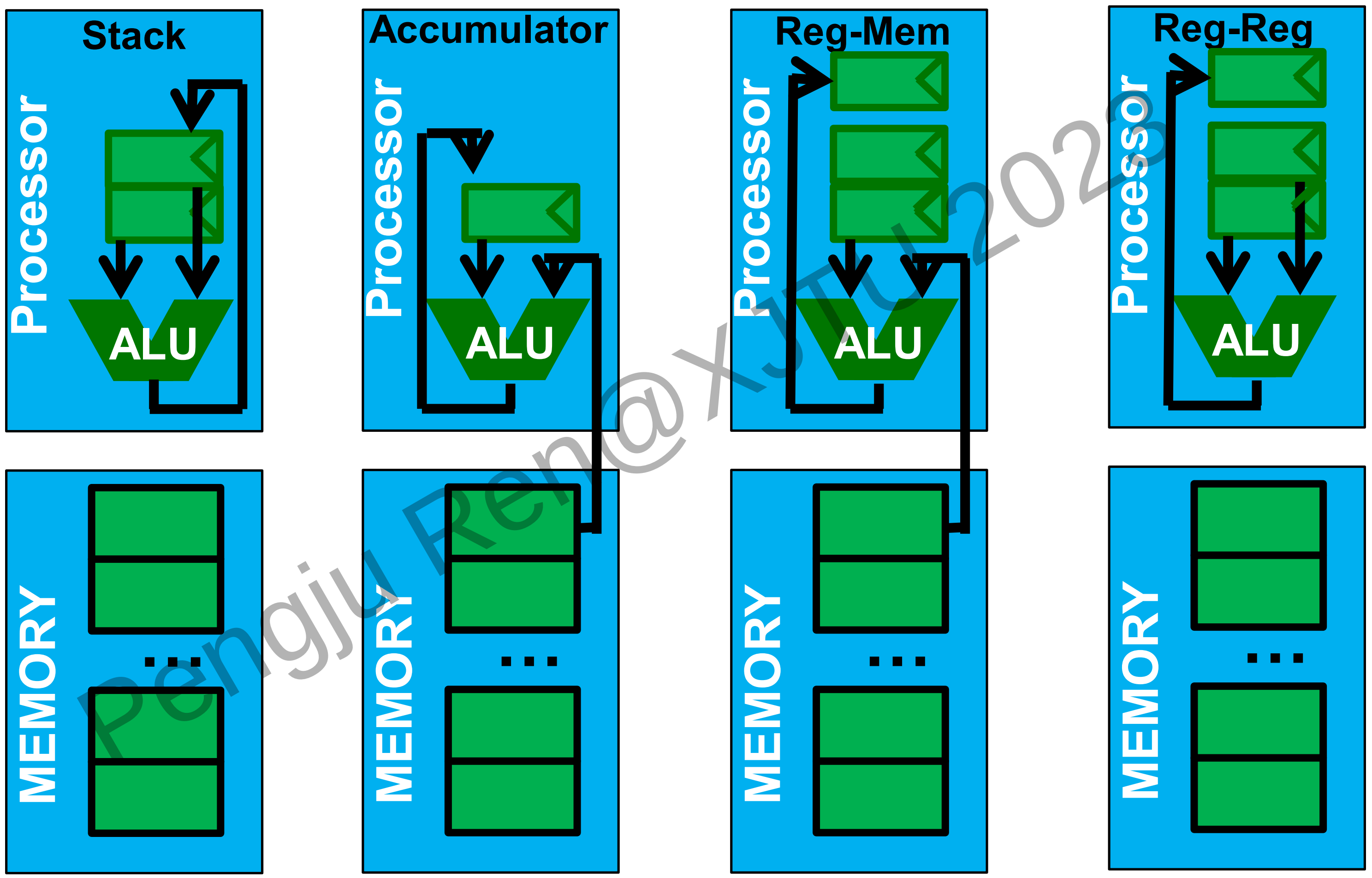
a and c are "loaded" twice

$\Rightarrow$

not the best use of registers!

program	*stack (size = 4)*
push a	R0
push b	R0 R1
push c	R0 R1 R2
*	R0 R1
+	R0
push a	R0 R1
push d	R0 R1 R2
push c	R0 R1 R2 R3
*	R0 R1 R2
+	R0 R1
push e	R0 R1 R2
-	R0 R1
/	R0

Where do Operands come from and Where do Results Go ?



C= A+B

Stack	Accumulator	Reg-Mem	Reg-Reg
Push A	Load A	Load R1 A	Load R1, A
Push B	Add B	Add R3 R1, B	Load R2, B
Add	Store C	Store R3, C	Add R3, R1, R2
Pop C			Store R3, C

Classes of Instructions

- **Data Transfer**
  - LD, ST, MFC1, MTC1, MFC0, MTC0
- **ALU**
  - ADD, SUB, AND, OR, XOR, MUL, DIV, SLT, LUI
- **Control Flow**
  - BEQZ, JR, JAL, TRAP, ERET
- **Floating Point**
  - ADD.D, SUB.S, MUL.D, C.LT.D, CVT.S.W,
- **Multimedia (SIMD)**
  - ADD.PS, SUB.PS, MUL.PS, C.LT.PS
- **String**
  - REP MOVSB (x86)

ISA Encoding

Fixed Width: Every Instruction has same width
- Easy to decode

(RISC Architectures: MIPS, PowerPC, SPARC, ARM…)

Ex: MIPS, every instruction 4-bytes

Variable Length: Instructions can vary in width
- Takes less space in memory and caches

(CISC Architectures: IBM 360, x86, Motorola 68k, VAX…)

Ex: x86, instructions 1-byte up to 17-bytes

Mostly Fixed or Compressed:
- Ex: MIPS16, THUMB (only two formats 2 and 4 bytes)
- PowerPC and some VLIWs (Store instructions compressed, decompress into Instruction Cache

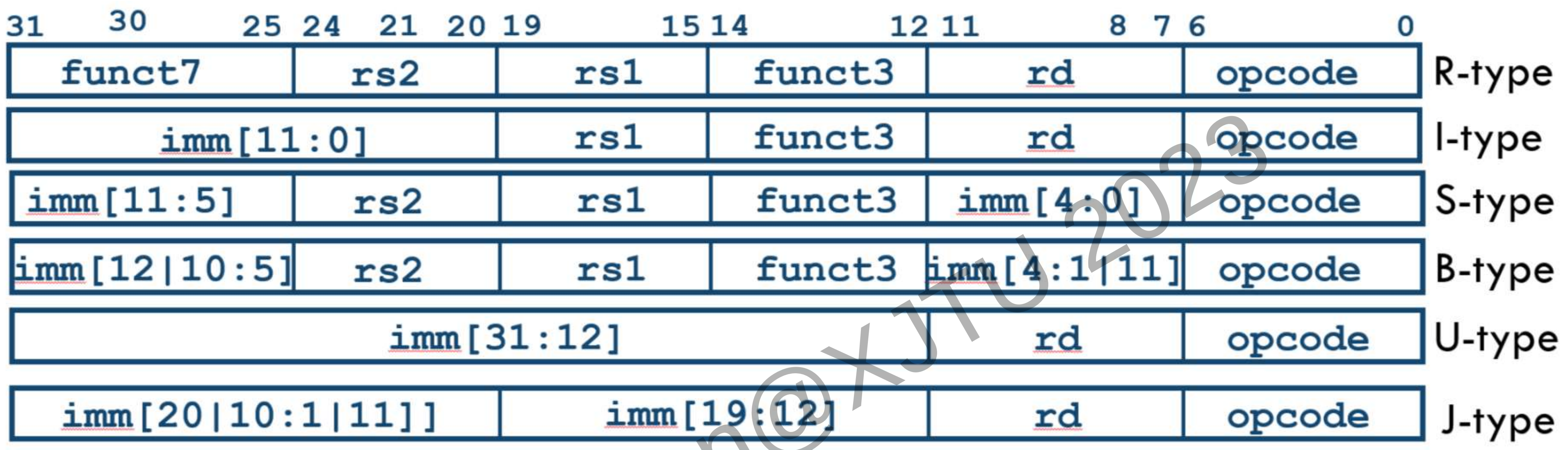
(Very) Long Instruction Word:
- Multiple instructions in a fixed width bundle
- Ex: Multiflow, HP/ST Lx, TI C6000

Case study: X86(IA-32) Instruction Encoding

Instruction Prefixes	Opcode	ModR/M	Scale, Index, Base	Displacement	Immediate
Up to four Prefixes (1 byte each)	1,2, or 3 bytes	1 byte (if needed)	1 byte (if needed)	0,1,2, or 4 bytes	0,1,2, or 4 bytes

x86 and x86-64 Instruction Formats
Possible instructions 1 to 18 bytes long

RISC-V Instruction Encoding(1)

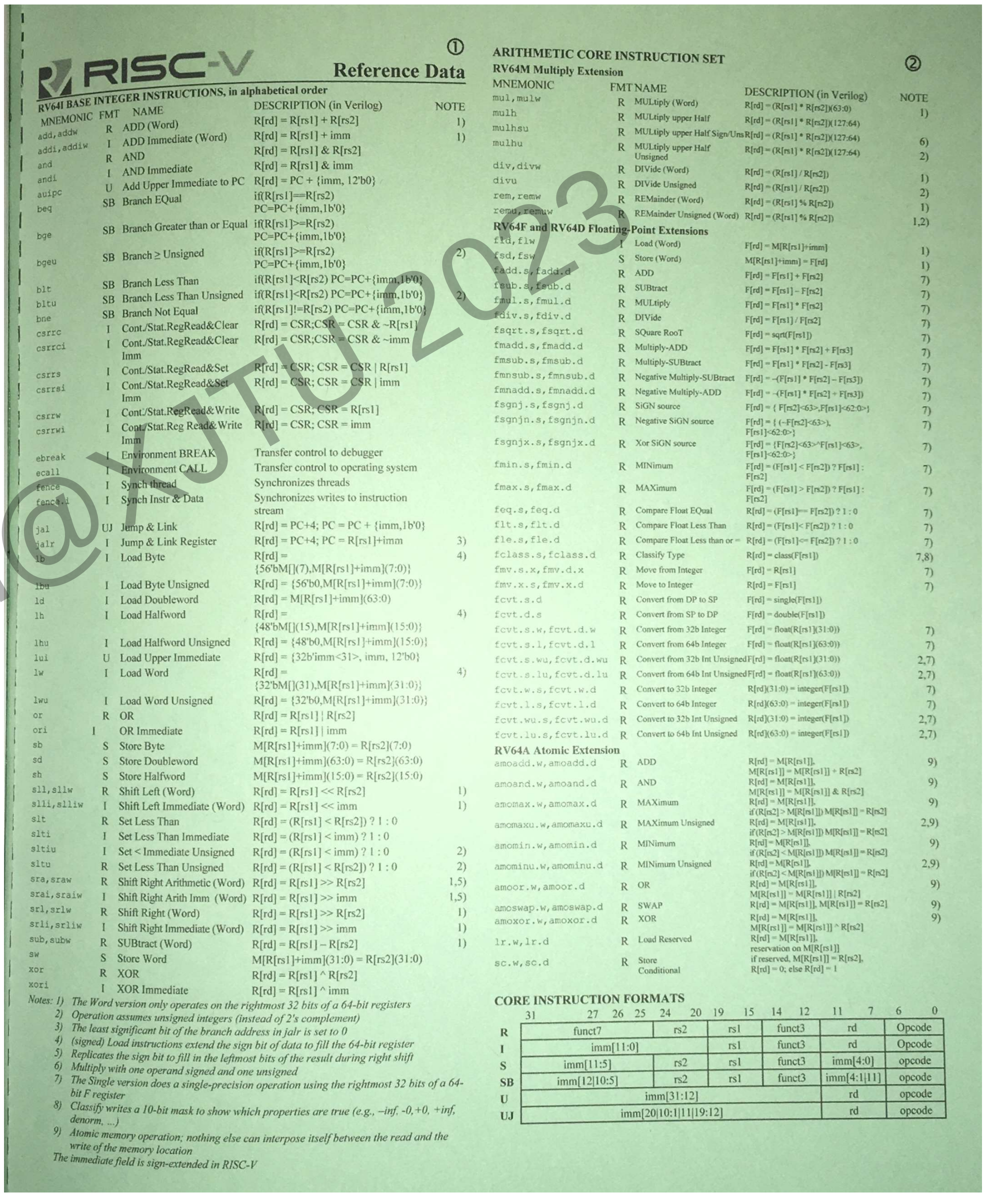

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12|10:5] | rs2 | rs1 | funct3 | imm[4:1|11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20|10:1|11]] | imm[19:12] | rd | opcode | J-type

- R-Format: instructions using 3 register inputs
  - add, xor, mul —arithmetic/logical ops
- I-Format: instructions with immediates, loads
  - addi, lw, jalr, slli
- S-Format: store instructions: sw, sb
- SB-Format: branch instructions: beq, bge
- U-Format: instructions with upper immediates
  - lui, auipc —upper immediate is 20-bits
- UJ-Format: jump instructions: jal

RISC-V Instruction Encoding(2)

New open-source, license-free ISA spec

- Supported by growing shared software ecosystem
- Appropriate for all levels of computing system, from microcontrollers to supercomputers
- 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)



RISC-V Reference Data ①

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] - R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., -inf, -0,+0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply upper Half	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
mulhu	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] - F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fmnsub.s,fmnsub.d	R	Negative Multiply-SUBtract	F[rd] = -(F[rs1] * F[rs2] - F[rs3])	7)
fmnadd.s,fmnadd.d	R	Negative Multiply-ADD	F[rd] = -(F[rs1] * F[rs2] + F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (~F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert from DP to SP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert from SP to DP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomic Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

http://www-inst.eecs.berkeley.edu/~cs61c/fa18/img/riscvcard.pdf

Real World Instruction Sets

Arch	Type	# Oper	# Mem	Data Size	# Regs	Addr Size	Use
Alpha	Reg-Reg	3	0	64-bit	32	64-bit	Workstation
ARM	Reg-Reg	3	0	32/64-bit	16	32/64-bit	Cell Phones, Embedded
MIPS	Reg-Reg	3	0	32/64-bit	32	32/64-bit	Workstation, Embedded
SPARC	Reg-Reg	3	0	32/64-bit	24-32	32/64-bit	Workstation
TI C6000	Reg-Reg	3	0	32-bit	32	32-bit	DSP
IBM 360	Reg-Mem	2	1	32-bit	16	24/31/64	Mainframe
x86	Reg-Mem	2	1	8/16/32/ 64-bit	4/8/24	16/32/64	Personal Computers
VAX	Mem-Mem	3	3	32-bit	16	32-bit	Minicomputer
Mot. 6800	Accum.	1	1/2	8-bit	0	16-bit	Microcontroler

Why the Diversity in ISAs?

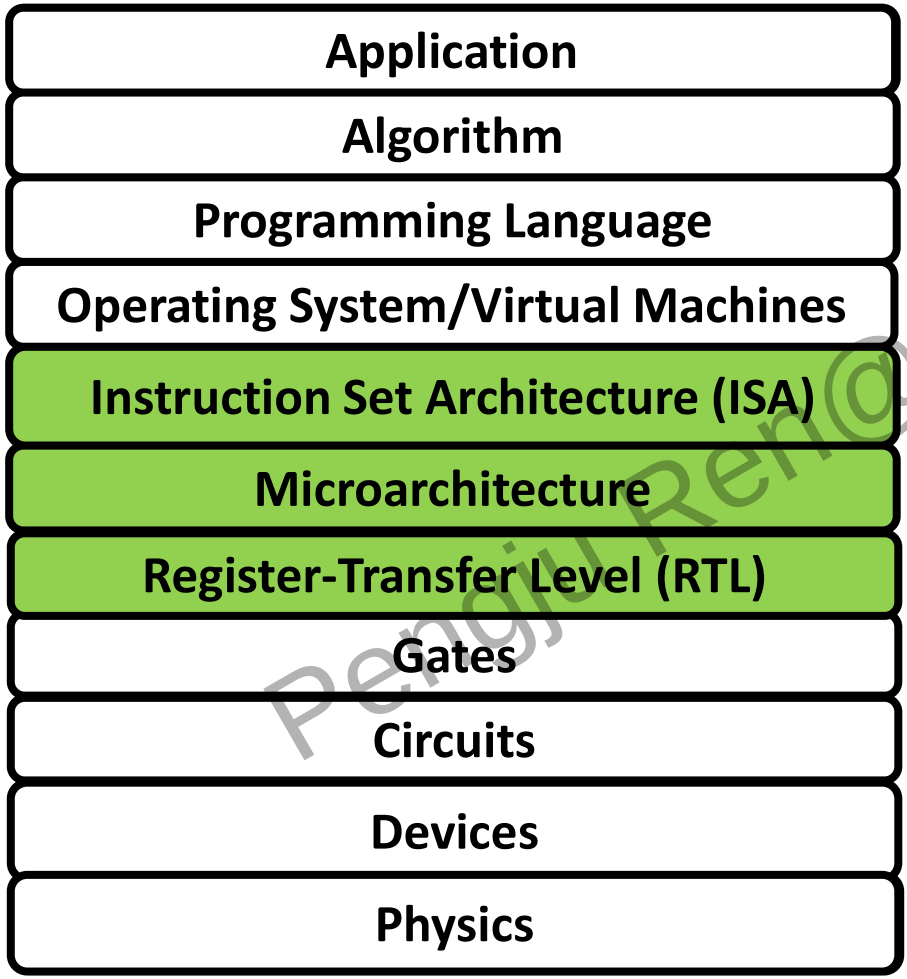
Application Influenced ISA

- Instructions for Applications
  - DSP instructions
- Compiler Technology has improved
  - SPARC Register Windows no longer needed
  - Compiler can register allocate effectively

Technology Influenced ISA

- Storage is expensive, tight encoding important
- Reduced Instruction Set Computer
  - Remove instructions until whole computer fits on die
- Multicore/Manycore
  - Transistors not turning into sequential performance

Recap


Application
Algorithm
Programming Language
Operating System/Virtual Machines
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Gates
Circuits
Devices
Physics

ISA vs Micro-Architecture

ISA Characteristics

- Machine Models
- Encoding
- Data Types
- Instructions
- Addressing Modes

And in conclusion …

- Computer Architecture >> ISAs and RTL
- Computer Architecture is about interaction of hardware and software, and design of appropriate abstraction layers
- Computer architecture is shaped by technology and applications
- Computer Science at the crossroads from sequential to parallel computing
  - Salvation requires innovation in many fields, including computer architecture
- Read Chapter 1 & Appendix A for next time! (6th)

Next Lecture： RISC-V ISA, Datapath & Control

(ISA and Micro-Architecture)