



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# Advanced Computer Architecture

## 高级计算机体系结构

---

### 第2讲：量化设计分析基础(2)

张献伟

[xianweiz.github.io](https://xianweiz.github.io)

DCS5367, 9/14/2021



中山大學  
SUN YAT-SEN UNIVERSITY



# 一些通知

---

- 下周课取消 (09/21)

- 选课人数
  - 86 (120 - 34)

- 课程QQ群
  - **962501738**

Advance CA fall...



- 课程主页: [xianweiz.github.io/](http://xianweiz.github.io/) (> Teaching)

- 问卷调查: <https://wj.qq.com/s2/8993787/debf/>

# How to Summarize Performance

- Arithmetic mean (weighted arithmetic mean)[算术平均]
  - Considering the frequencies of programs in the workload
  - E.g.,: tracks execution time:  $\sum_{i=1}^n \frac{T_i}{n}$  or  $\sum_{i=1}^n W_i * T_i$
- Harmonic mean (weighted harmonic mean) of rates[调和平均]
  - E.g.,: track MFLOPS:  $\frac{n}{\sum_{i=1}^n \frac{1}{Rate_i}}$
- **Normalized** execution time is handy for scaling performance (e.g., X times faster than Pentium 4)
- Geometric mean  $\Rightarrow \sqrt[n]{\prod_{i=1}^n execution\_ratio_i}$  [几何平均]
  - The execution ratio is relative to a reference machine
    - Based on relative performance to a reference machine

# Performance Evaluation[性能评估]

---

- **Execution time** and **power** are the main measure of computer performance
- Good products created when we have
  - Good benchmarks
    - For better or worse, benchmarks shape a field
  - Good ways to summarize performance
    - Reproducibility is important (should provide details of experiments)
- Given that sales is a function, in part, of performance relative to competition, companies invest in improving performance summary
  - If benchmarks/summary are inadequate, then choose between improving product for real programs vs. improving product to get more sales ==> Sales almost always wins!

# Quantitative Principles (§1.8)[量化原则]

---

- Guidelines and principles that are useful in the design and analysis of computers
- Take advantage of **parallelism**[并行]
  - System level: multiple processors, multiple disks
  - Individual processor: instruction parallelism, e.g., pipelining
  - Detailed digital design: cache, memory
- Principle of **locality**[局部性]
  - Programs tend to reuse data and insts they have used recently
    - A program spends 90% of its execution time in only 10% of the code
- Focus on the **common case**[一般情况]
  - To make a trade-off, favor the frequent case over infrequent

# Amdahl's Law [阿姆达尔定律]

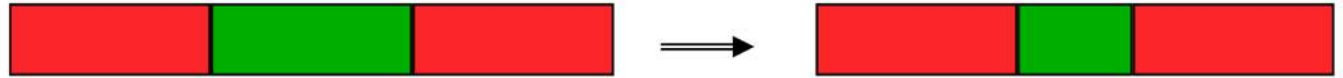
---

- The performance **improvement** to be gained from using some faster mode of execution is limited by the **fraction** of the time the faster mode can be used
- 系统中对某一部件采用更快执行方式所能获得的系统**性能改进程度**，取决于这种执行方式被使用的**频率**，或所占总执行时间的比例
- Amdahl's law defines the speedup that can be gained by using a particular feature
  - Speedup due to some enhancement E:

$$Speedup_{overall} = \frac{ExTime_{withoutE}}{ExTime_{withE}} = \frac{Performance_E}{Performance_{withoutE}}$$

# Amdahl's Law (cont.)

- Suppose that enhancement  $E$  accelerates a fraction of the task by a factor  $S$ , and the remainder of the task is unaffected



$ExTime_{withE}$

$$= ExTime_{withoutE} * \left[ (1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{S} \right]$$

$$Speedup = \frac{ExTime_{withoutE}}{ExTime_{withE}}$$

$$= \frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{S}}$$

# Amdahl's Law (cont.)

---

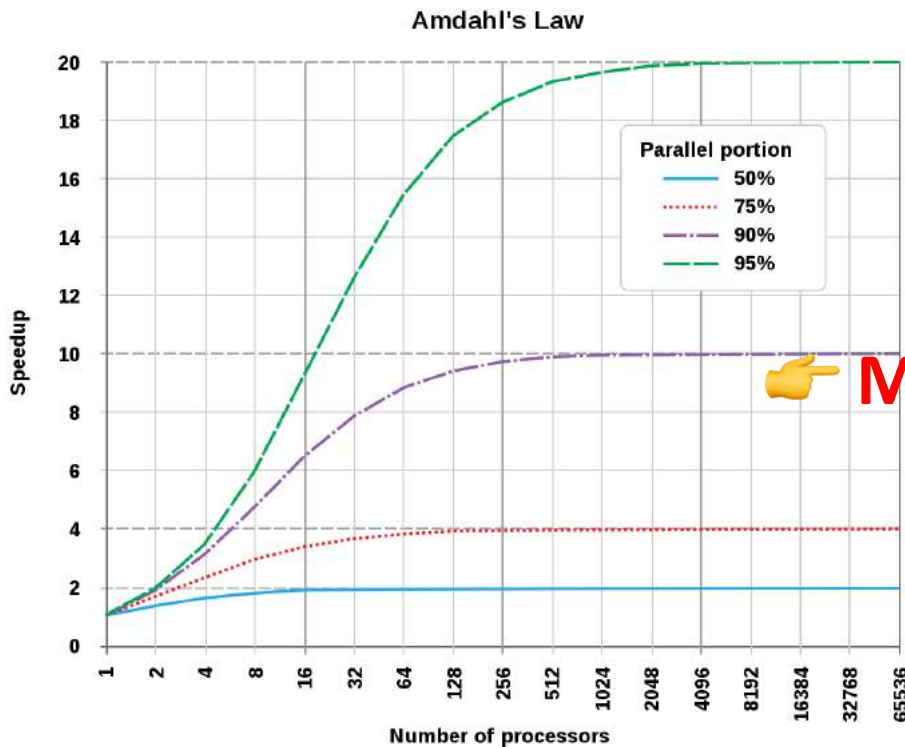
- Example 1: Floating point instructions can be improved to run 2X; but only 10% of actual instructions are FP. What is the overall speedup?
  - $\text{Fraction}_E = 10\%$ ,  $S = 2$ ,  $\text{Speedup} = 1/(90\% + 10\%/2) = 1.05$
- Example 2: Assume we need to improve the performance of a graphics engine (assume 20% inst are FP Square root, 50% for all FP inst). Which choice is better?
  - Choice one: Speed up FP Square root by 10x  
 $1/(80\% + 20\%/10) = 1.22$
  - Choice two: Speed up all FP instruction by 1.6x  
 $1/(50\% + 50\%/1.6) = 1.23$

👉 **Focus on the common case !**



# Amdahl's Law (cont.)

- A program's speedup is limited by its serial part
  - For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be **20x**



**Make the fast case common !**

# Computing CPU time

---

- CPU @ 2.5GHz
  - 2.5G ticks per second  $\rightarrow 1/2.5G \text{ s/tick} = 0.4\text{ns} / \text{tick}$
  - Tick == clock == clock cycle
- CPU time for a program, i.e., #clock cycles to execute
  - CPU time = CPU clock cycles for a program x Clock cycle time
  - CPU time = CPU clock cycles for a program / Clock rate
- Clock cycles per instruction (CPI)
  - CPI = CPU clock cycles for a program / Instruction count
  - Reverse of IPC (instructions per cycle)
- CPU time = Inst count x CPI x Clock cycle time
  - $\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}}$

# Computing CPU time (cont.)

- *Average Cycles per Instruction (CPI) =  $\sum_{j=1}^n CPI_j * F_j$* 
  - Where  $CPI_j$  is the number of cycles needed to execute instructions of type  $j$
  - and  $F_j$  is the percentage (fraction) of instructions that are of type  $j$

**Example:** Base Machine (Reg / Reg)

Op	Freq	Cycles	$CPI_j * F_j$	(% Time)
ALU	50%	1	.5	(33%)
Load	20%	2	.4	(27%)
Store	10%	2	.2	(13%)
Branch	20%	2	.4	(27%)
			<u>1.5</u>	

Typical Mix

- *CPU time = Cycle time \*  $\sum_{j=1}^n CPI_j * I_j$* 
  - $I_j$  is the number of instructions of type  $j$ , and Cycle time is the inverse of the clock rate.

# Computing CPU time (cont.)

---

- CPI is a function of the machine and program.
  - The CPI depends on the actual **instructions** appearing in the program—a floating-point intensive application might have a higher CPI than an integer-based program.
  - It also depends on the **CPU implementation**. For example, a Pentium can execute the same instructions as an older 80486, but faster.
- It is common to each instruction took one cycle, making  $CPI = 1$ .
  - The CPI can be  $>1$  due to memory stalls and slow instructions.
  - The CPI can be  $<1$  on machines that execute more than 1 instruction per cycle (superscalar).

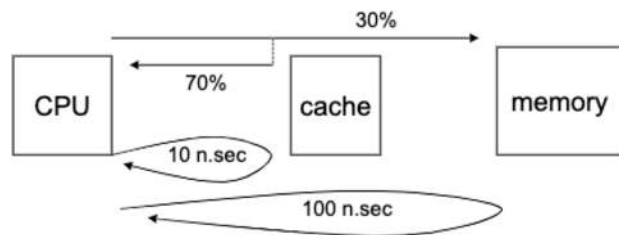
# Aspects of CPU Performance

• 
$$CPU\ time = \frac{Seconds}{program} = \frac{Instructions}{program} * \frac{Cycles}{Instructions} * \frac{Seconds}{Cycles}$$

	Inst Count	CPI	Clock Rate
Program	○		
Compiler	○	○	
Inst. Set	○	○	
Organization		○	○
Technology			○

# Improving CPI using caches

- An example



What is the improvement (speedup) in memory access time? :

- Caching works because of the principle of locality:
  - Locality found in memory access instructions
    - **Temporal locality**: if an item is referenced, it will tend to be referenced again soon
    - **Spatial locality**: if an item is referenced, items whose addresses are close by tend to be referenced soon
  - 90/10 locality rule
    - A program executes about 90% of its instructions in 10% of its code
  - We will look at how this principle is exploited in various microarchitecture techniques



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# Advanced Computer Architecture

## 高级计算机体系结构

---

### 第2讲：ISA and ILP (1)

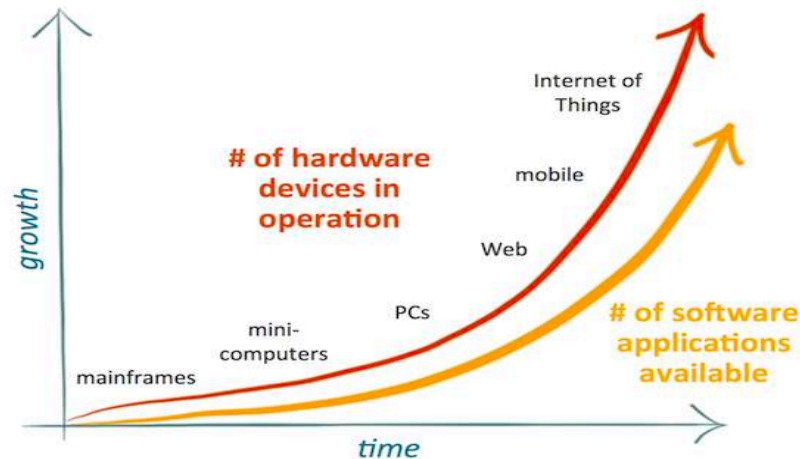
张献伟

[xianweiz.github.io](https://xianweiz.github.io)

DCS5367, 9/14/2021

# The History

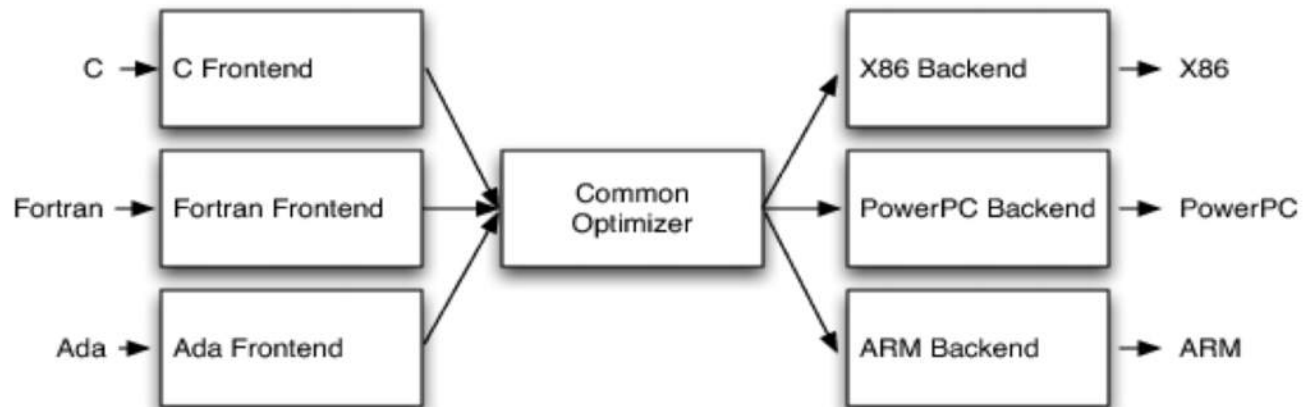
- For more than 50 years, we have enjoyed exponentially increasing compute power[算力急剧增长]
- The growth is based on a fundamental contract between HW and SW[得益于软硬件之间的协议]
  - HW may change radically “under the hood”
    - Old SW can still run on new HW (even faster)
  - HW looks the same to SW, always speaking the same language
    - The **ISA**, allows the decoupling of SW development from HW dev





# Program Compilation[程序编译]

- **Program** written in a “high-level” programming language
  - C/C++, Java, Python
  - Hierarchical, structured control: loops, functions, conditionals
  - Hierarchical, structured data: scalars, arrays, pointers, structures
- **Compiler**: translates program to assembly
  - Parsing and straight-forward translation
  - Compiler also optimizes



# What is ISA?

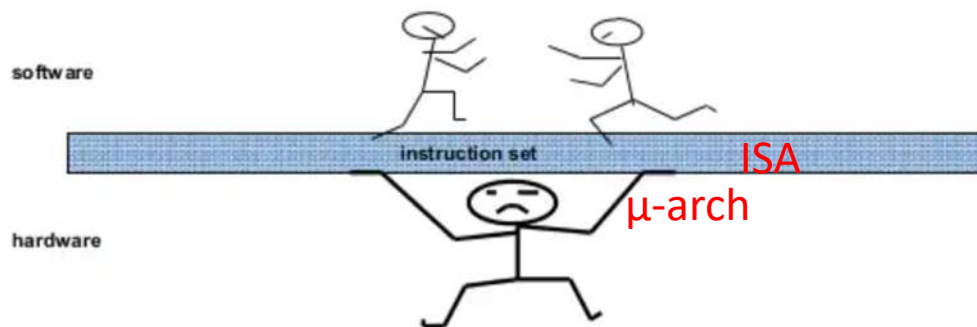
---

- Instruction Set == A set of instructions
- The HW/SW **contract**[软硬件协议]
  - Compiler correctly translates source code to the ISA[编译器]
  - Assembler translates to relocatable binary[汇编器]
  - Linker solidifies relocatables into object code[连接器]
  - HW promises to do what the object code says[硬件执行]
- Not in the “contract”: non-functional aspects[非协议]
  - How operations are implemented
  - Which operations are fast and which are slow and when
  - Which operations take more power and which take less

# ISA + $\mu$ -arch = Arch

- “Architecture” = ISA + microarchitecture
- ISA[指令集架构]
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture ( $\mu$ -arch)[微架构]
  - Specific implementation of an ISA
    - Implementation of the ISA under specific design constraints and goals
  - Not visible to the software

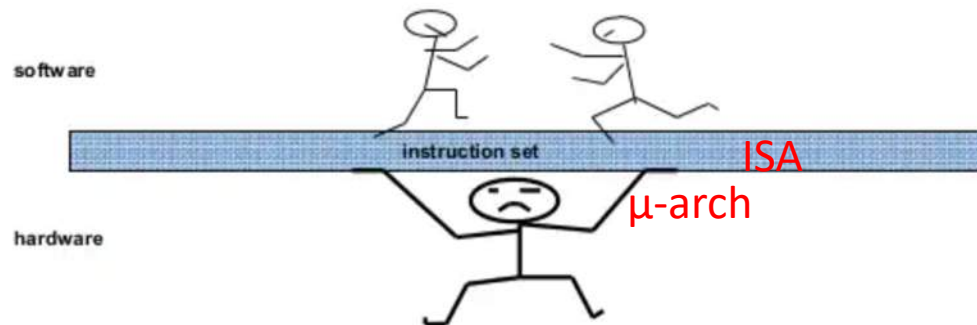
Problem
Algorithm
Program/Language
Runtime System (VM, OS, MM)
ISA (Architecture)
Microarchitecture
Logic
Circuits
Electrons



# ISA vs. $\mu$ -arch (cont.)

---

- Implementation ( $\mu$ -arch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, ...
- $\mu$ -arch usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha) but many u-archs



# What Makes a Good ISA?

---

- Programmability[可編程性]
  - Easy to express programs efficiently?
- Implementability[可實現性]
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power implementations?
    - Easy to design high-reliability implementations?
    - Easy to design low-cost implementations?
- Compatibility[兼容性]
  - Easy to maintain programmability (implementability) as languages and programs (technology) evolves?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2...

# How to Design ISA?[设计]

---

- Design decisions must take into account[考虑因素]
  - Technology
  - Machine organization
  - Programming languages
  - Compiler technology
  - Operating systems
- Issues in instruction set design[一些问题]
  - Operand storage in CPU (stack, registers, accumulator)
  - Number of operands in an instruction (fixed or variable number)
  - Type and size of operands (how is operand type determined)
  - Addressing modes
  - Allowed operations and the size of op-codes
  - Size of each instruction.

# ISA Classifying (§A.2)[分类]

---

- Type of internal storage in a processor
  - Major choices: stack, accumulator, registers
- Stack architecture[栈]
  - Operands are implicitly on the top of the stack
- Accumulator architecture[聚集器]
  - One operand is implicitly the accumulator
- General-purpose register (GPR) architecture[通用寄存器]
  - Only explicit operands – either registers or memory locations
  - Two subclasses
    - Register-memory: can access any memory as part of any instruction
    - **Load-store**: can access memory only with load and store instructions
    - ~~Memory-memory: all operands in memory~~

# Example

---

- $C = A + B$

- Stack

Push A

Push B

Add

Pop C

- Accumulator

Load A

Add B

Store C

- Register (register-memory)

Load R1, A

Add R3, R1, B

Store R3, C

- Register (load-store)

Load R1, A

Load R2, B

Add R3, R1, R2

Store R3, C



# Memory Addressing (§A.3)[内存寻址]

- How memory addresses are **interpreted** and how they are **specified**[解释和指定]
  - Interpretation: what object is accessed as a function of the address and the length?
  - Addressing modes: the ways addresses are specified
- Addressing modes[寻址模式]
  - Register, immediate, indexed, ...
- Effective address[有效地址]
  - The actual address to access a memory location



# Memory Addressing (cont.)

---

- Register[寄存器] Add R4, R3
  - operand = content of register
- Immediate[立即] Add R4, 3
  - operand = in instruction
- Register indirect[寄存器间接] Add R4, (R1)
  - operand = in memory = Mem(R1)
  - address = content of register
- Displacement[偏移] Add R4, 100(R1)
  - operand in memory = Mem[(R1) + base]
  - address = content of register + base
- Indexed[索引] Add R3, (R1 + R2)
  - operand in memory = Mem[(R1) + (R2)]
  - address = content of R1 + content of R2

Note: (R) means content of R and Mem[A] means content of memory address A

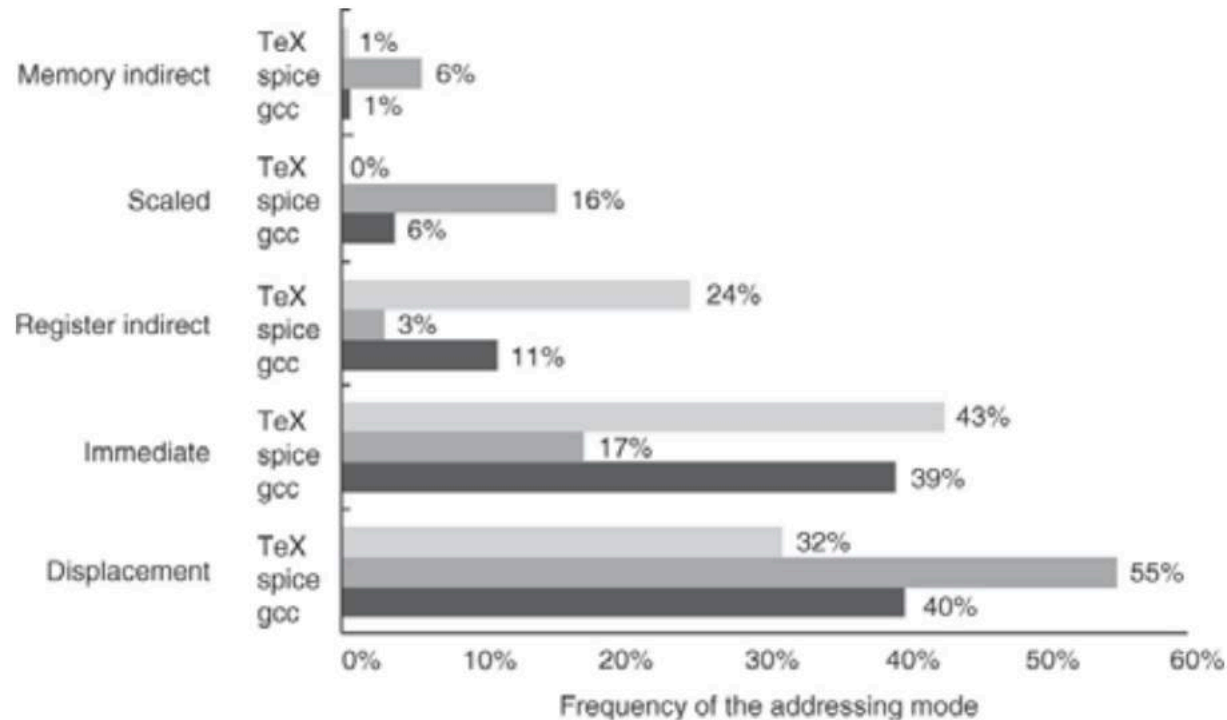
# Memory Addressing (cont.)

---

- Direct (absolute)[直接/绝对] Add R1, (1001)
  - operand in memory = Mem[C]
  - address = a constant in the instruction
- Memory indirect[内存间接] Add R1, @(R3)
  - operand in memory = Mem[Mem[(R3)]]
  - address = the content of Mem[(R3)]
- Auto-increment (or decrement)[自增] Add R1, (R2)+
  - operand in memory = Mem[(R2)]
  - The content of R2 is incremented
- Scaled[比例] Add R1, 100(R2)[R3]
  - operand in memory = Mem[C+(R2)+(R3)\*d]

# Memory Addressing (cont.)

- The usage of various addressing modes is critical in helping the architect what to include [模式使用的影響]
  - Can significantly reduce instruction counts
  - Can also add to the complexity of building a computer and may increase CPI



# Operations (§A.5)[操作]

---

- Operators supported by most ISAs
  - Arithmetic/logical: add, sub, mult, div, shift (arith,logical), and, or, not, xor ...
  - Data transfer: copy, move, load, store, ..
  - Control: branch, jump, call, return, trap, ...
  - System: operating system call, virtual memory management, ...
  - Floating point: add, mult, div, ...
  - Decimal: add, multi, decimal-to-character conversions
  - String: move, copy, compare, search
  - Graphics: pixel operations, compression, ...
- Rule of thumb: most widely executed instructions are the simple operations of an instruction set

# Operations (cont.)

---

- 10 simple insts account for 96% of insts executed for a collection of integer programs running on Intel 80x86
  - Common case, make them fast

Rank	80x86 instruction	Integer avg % total executed
1	Load	22%
2	Conditional branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move register-register	4%
9	Call	1%
10	Return	1%
<b>Total</b>		<b>96%</b>

# Encoding (§A.7)[编码]

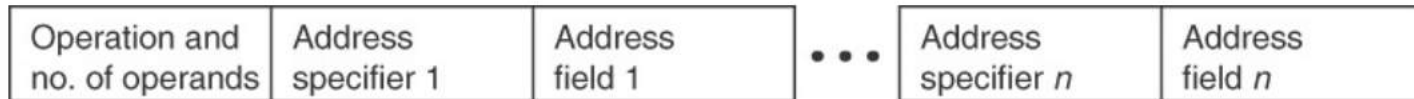
---

- Encoding: instructions → binary representation
  - Affects the size of the compiled program
  - Affects the implementation of processors
    - Decode instruction to quickly find the operation (*opcode*) and its operands
- How to encode the addressing modes with the operations
  - Depends on the range of addressing modes and the degree of independence between opcodes and modes
    - Older computers: 1-5 operands with 10 addressing modes for each → separate address specifier for each operand
    - Load-store computers: only one memory operand and 1-2 addressing modes → encode mode as part of the opcode

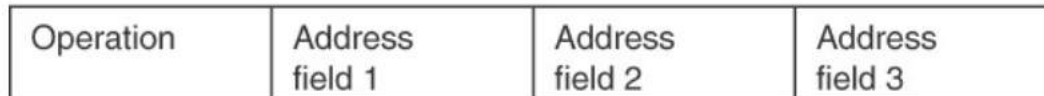
# Encoding (cont.)

---

- Popular choices
  - Variable: allows all addressing modes to be with all operations
  - Fixed: few addressing modes and operations
  - Hybrid
- Trade-off: size of the program vs. ease of decoding
  - Variable: use as few bits as psbl to represent the program, but individual instructions can vary widely



(a) Variable (e.g., Intel 80x86, VAX)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

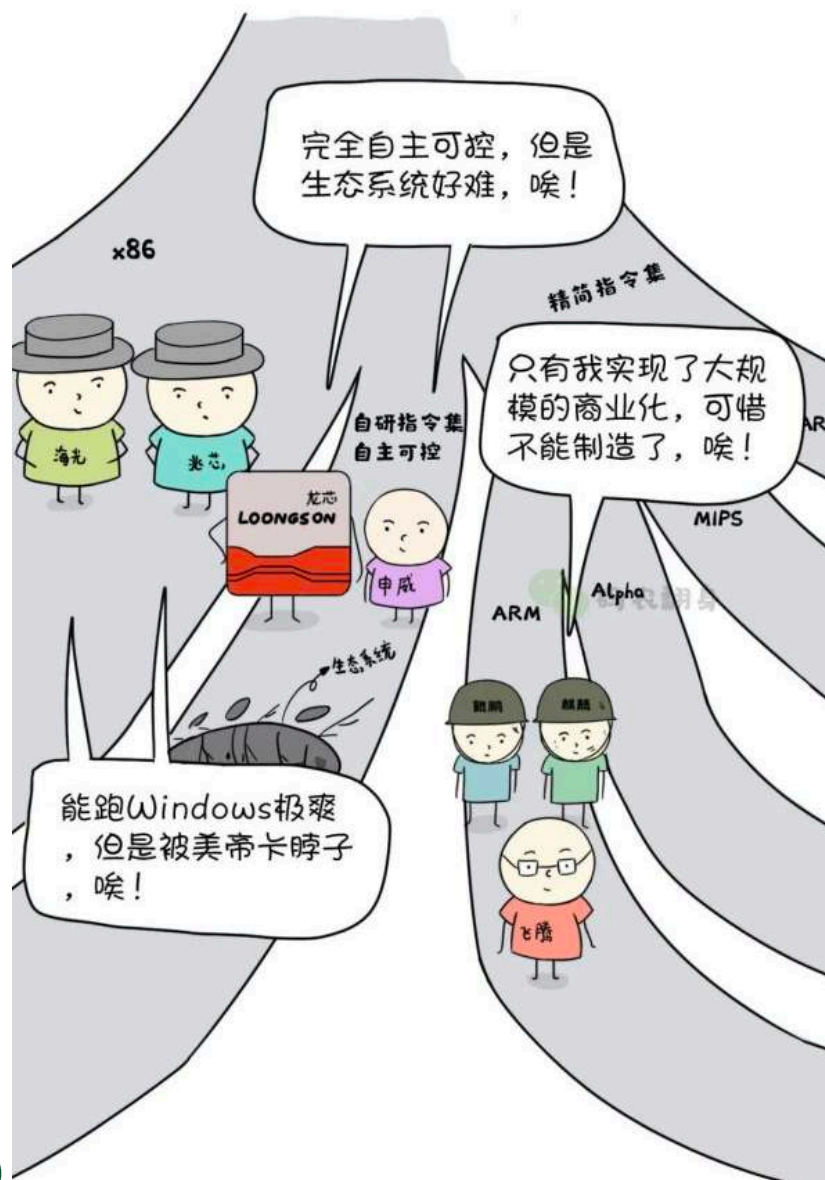


# Existing ISAs

- RISC: reduced-instruction set computer[精简指令集]
  - Coined by Patterson in early 80's
  - RISC-I (Patterson), MIPS (Hennessy), IBM 801 (Cocke)
  - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC
- CISC: complex-instruction set computer[复杂指令集]
  - Term didn't exist before "RISC"
  - Examples: x86, VAX, Motorola 68000, etc.



# 国产架构



- x86
  - 曙光/海光
- ARM
  - 华为、飞腾
- 自主
  - 龙芯、申威

\* CPU及指令集演进 (漫画 | 20多年了，为什么国产CPU还是不行?)

# Performance Argument[性能的争论]

---

- Performance equation:
  - $(\text{instructions/program}) * (\text{cycles/instruction}) * (\text{seconds/cycle})$
- CISC
  - Reduce “instructions/program” with “complex” instructions
    - But tends to increase CPI or clock period
  - Easy for assembly-level programmers, good code density
  - Idea: give programmers powerful insts, fewer insts to complete the work
- RISC
  - Improve “cycles/instruction” with many single-cycle instructions
  - Increases “instruction/program”, but hopefully not as much
    - Help from smart compiler
  - Idea: compose simple insts to get complex results

# CISC vs. RISC

---

- **Instructions:** multi-cycle complex vs. single-cycle reduced
- **Addressing modes:** many vs. few
- **Encoding:** many formats and lengths vs. fixed-length instruction format
- **Performance:** hand assemble to get good performance vs. reliance on compiler optimizations
- **Registers:** few vs. many (compilers are better at using them)
- **Code size:** small vs. large



# CISC vs. RISC (cont.)

---

- The war started in mid 1980's
  - CISC won the high-end commercial war (1990s to today)
    - Compatibility a stronger force than anyone (but Intel) thought
  - RISC won the embedded computing war
- CISC: winner on revenue[贏在收益]
  - X86 was the first 16-bit microprocessor
    - No competing choices → historical inertia and “financial feedback”
  - Moore’s law was the helper
    - Most engineering problems can be solved with more transistors
- RISC: winner on volume[贏在數量]
  - First ARM chip in mid-1980s → 150 billion chips
  - Low-power and embedded devices (e.g., cellphones)

# x86 → ARM → RISC-V [进行中的变革]

- But now, things are changing ...
  - Fugaku: ARM-based supercomputer (Top1)
  - Apple Inc.: ARM-based M1 chip
  - Amazon Inc.: AWS Graviton processor
- RISC-V: a freely licensed open standard (Linux in hw)
  - Builds on 30 years of experience with RISC architecture, “cleans up” most of the short-term inclusions and omissions
    - Leading to an arch that is easier and more efficient to implement



# What is RISC-V?

- Fifth generation of RISC design from UC Berkeley[第五代]
- A high-quality, license-free, royalty-free RISC ISA[自由]
- Experiencing rapid uptake in both industry and academia[快速发展]
- Supported by growing shared software ecosystem[生态]
- Appropriate for all levels of computing system, from microcontrollers to supercomputers[普适]
  - 32-bit, 64-bit, and 128-bit variants
- Standard maintained by non-profit RISC-V Foundation



<https://riscv.org/>






# RISC-V (cont.)

- The free and open RISC instruction set architecture
  - Free and open ISA enabling a new era of processor innovation through open standard collaboration [彻底开放]
  - RISC-V ISA delivers a new level of open, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation

## What's Different About RISC-V? ("RISC Five", fifth UC Berkeley RISC)

- Free and Open
  - Anyone can use
  - More competition  
⇒ More innovation
  - Pick ISA, then vendor
- For Cloud & Edge
  - From large to tiny computers
- Secure/Trustworthy
  - Design own secure core
  - Open cores ⇒ no secrets
- Simple, Elegant
  - 25 years later, learn from 1st gen RISCs\*
  - Far simpler than ARM and x86
  - **Can add custom instructions**
  - **Input from software/architecture experts BEFORE finalize ISA**
- Community designed
  - RISC-V Foundation owns RISC-V ISA



# The RISC-V Architecture[架构]

---

- 32, 64-bit general purpose registers (GPRs)
  - called x0, ... , x31 (x0 is hardwired to the value 0).
- 32, 64-bit floating point registers - FPRs (each can hold a 32-bit single precision or a 64-bit double precision value)
  - called f0, f1, ... , f31 (or f0, f2, ... , f30)
- A few special purpose registers (example: floating point status),
- Byte addressable memories with 64-bit addresses
- 32-bit instructions
- Only immediate and displacement addressing modes (12-bit field)

Data transfer operations: ld, lw, lb, lh, flw, sd, sw, sb, sh, fsw, ...

Arithmetic/logical operations: add, addi, sub, subi, slt, and, andi, xor, mul, div, ...

Control operations: beq, bne, blt, jal, jalr, ...

Floating point operations: fadd, fsub, fmult, fsqrt, ...

# $\mu$ -ops[微操作]

---

- x86: RISC inside
  - Maintains x86 ISA externally for compatibility
  - But executes RISC  $\mu$ ISA internally for implementability
    - x86 code is becoming more “RISC-like”
  - Different  $\mu$ ops for different designs
    - Not part of the ISA specification, not publicly disclosed

- Example:

`push $eax`

becomes (we think, uops are proprietary)

`store $eax, -4($esp)`

`addi $esp,$esp,-4`

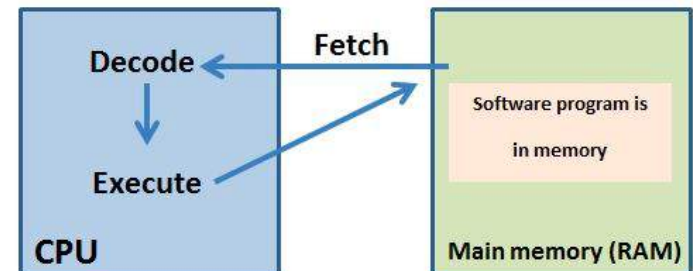
# Translation and Virtual ISAs[翻译和虚拟]

---

- New compatibility interface: ISA + translation software
  - Binary-translation: transform static image, run native
  - Emulation: unmodified image, interpret each dynamic inst
  - Typically optimized with just-in-time (JIT) compilation
  - Examples: 龙芯 x86 → LoongArch
  - Performance overheads reasonable (many recent advances)
- Virtual ISAs: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Goals: Portability (abstract hardware nastiness), flexibility over time
  - Examples: Java Bytecodes, NVIDIA's "PTX"

# Instruction Execution

- Instruction fetch (IF)
  - Fetch the next instruction from memory (and update PC to the next sequential instruction)
- Instruction decode/register fetch (ID)
  - Decode the inst and read the registers corresponding to register source specifiers
- Execution/effective address (EX)
  - Operate on the operands prepared in the prior cycle
- Memory access (MEM)
  - Load: read using the effective address, store: write to memory
- Write-back (WB)
  - Writes the result into the register



# Pipelining (§C.1)

- Pipelining: an implementation technique whereby multiple instructions are overlapped in execution
  - Just like an assembly line
  - Takes advantage of parallelism that exists among the actions needed to execute an instruction
  - Pipelining is the key technique to make fast processors

Instr. No.	Pipeline Stage						
	IF	ID	EX	MEM	WB		
<b>1</b>	IF	ID	EX	MEM	WB		
<b>2</b>		IF	ID	EX	MEM	WB	
<b>3</b>			IF	ID	EX	MEM	WB
<b>4</b>				IF	ID	EX	MEM
<b>5</b>					IF	ID	EX
<b>Clock Cycle</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>