

Class Topics (클래스 홈페이지 참조)

- ❑ Part 1: Fundamental concepts and principles
- ❑ Part 2: 빠른 컴퓨터를 위한 ISA design
 - Topic 1 Computer performance and ISA design (Ch. 1)
 - ❑ 1-1 Performance evaluation & performance models
 - ❑ 1-2 RISC versus CISC, power limit
 - Topic 2 RISC (MIPS) instruction set (Chapter 2)
 - Topic 3 Computer arithmetic and ALU (Chapter 3)
- ❑ Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)

Performance and ISA Design

Part 2

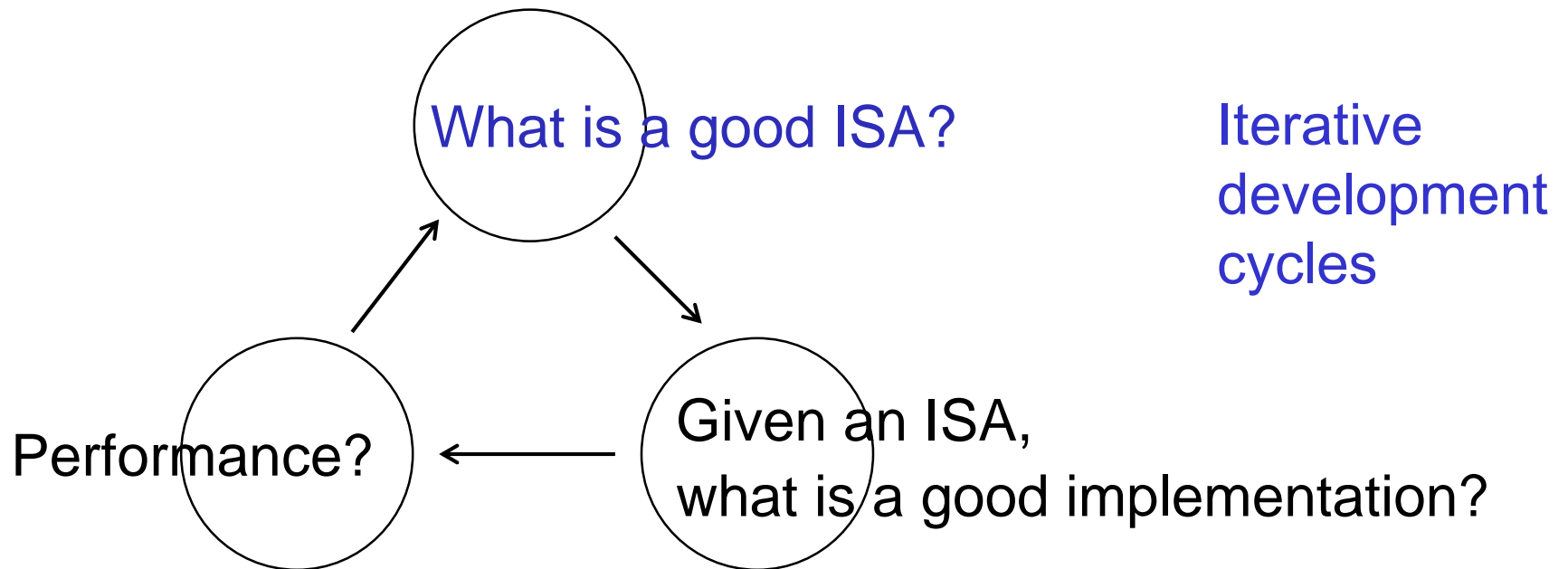
- 1) RISC vs. CISC
- 2) Amdahl's law
- 3) Power Limit and Multicores

References:

1. Computer Organization and Design & Computer Architecture, Hennessy and Patterson (slides are adapted from those by the authors)

Big Picture - Iterative Cycles (반복)

□ $\text{CPU time} = \text{IC} * \text{CPI} * \text{clock cycle time}$



(Some of) ISA Design Issues

❑ Operations (opcode)

- How many, what types of instructions
 - ALU, data transfer, branch, others

❑ Operands

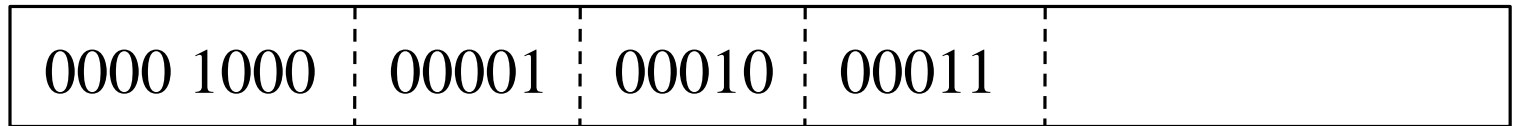
- How to specify the locations of operands
 - Addressing modes: register, direct, immediate, ...
- Operand types (data types - more later)
- How many operands in ALU instructions?
 - Number of memory operands

❑ Instruction encoding: how to pack all in words

Assembly vs. Binary Languages (반복)

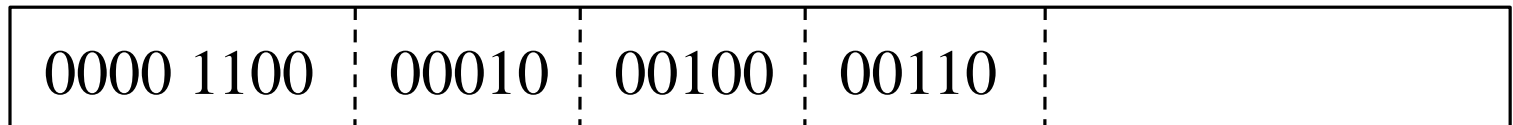
- ALU instructions: 32-bit long (opcode, operands)

ADD R1, R2, R3



Opcode(8) Reg(5) Reg(5) Reg(5) unused(9)

OR R2, R4, R6



- The two are identical - both called machine languages
 - Simple 1:1 translation
 - Assembly language: mnemonic

Assembly vs. Binary Language (반복)

- ❑ Load and store instructions: 32-bit long

LD R1, R31(8)

0000 0010	00001	11111	00 0000 0000 1000
Opcode(8)	Reg(5)	Reg(5)	Constant(14)

ST R2, R31(4)

0000 0011	00010	11111	00 0000 0000 0100
-----------	-------	-------	-------------------

- ❑ Why not use absolute memory address?
- ❑ What is instruction decode?

ISA Classes

How many operands in ALU instructions?

- Number of memory operands

ISA Classes

(Hennessy and Patterson, Computer Architecture, Morgan Kaufmann)

How to perform “arithmetic and logic” computation

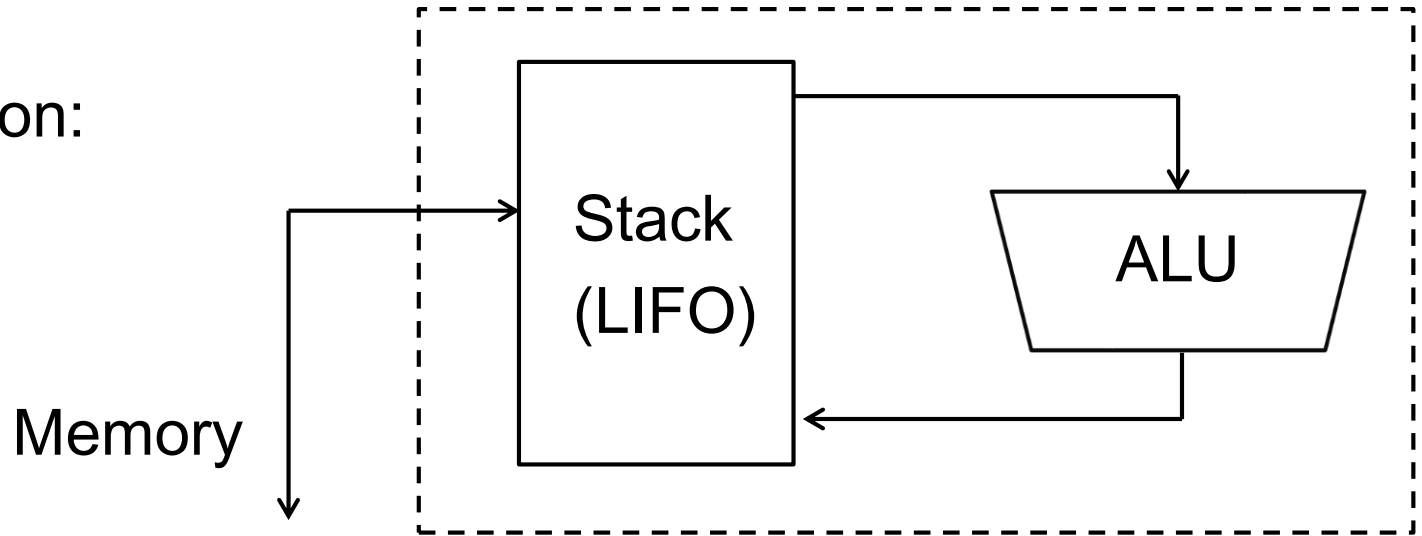
$C = A + B;$ // A, B, C in memory

❑ Code sequence for four classes of instruction sets

Stack	Accumulator	GPR (Register-memory)	GPR (Register-register)
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

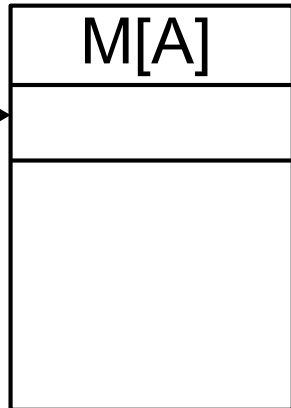
Stack ISA: Zero-Operand Arch.

Implementation:

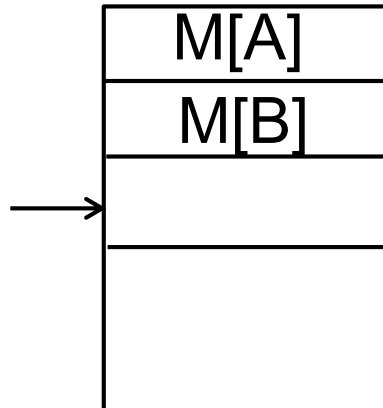


(C = A + B;)

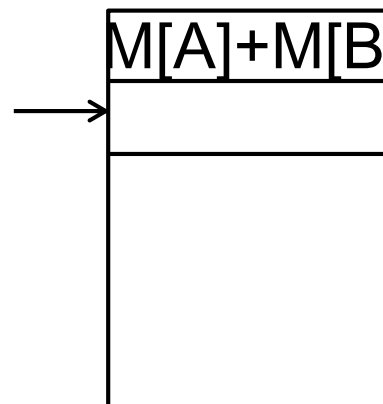
Push A



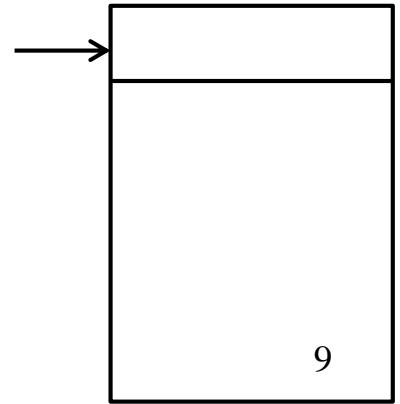
Push B



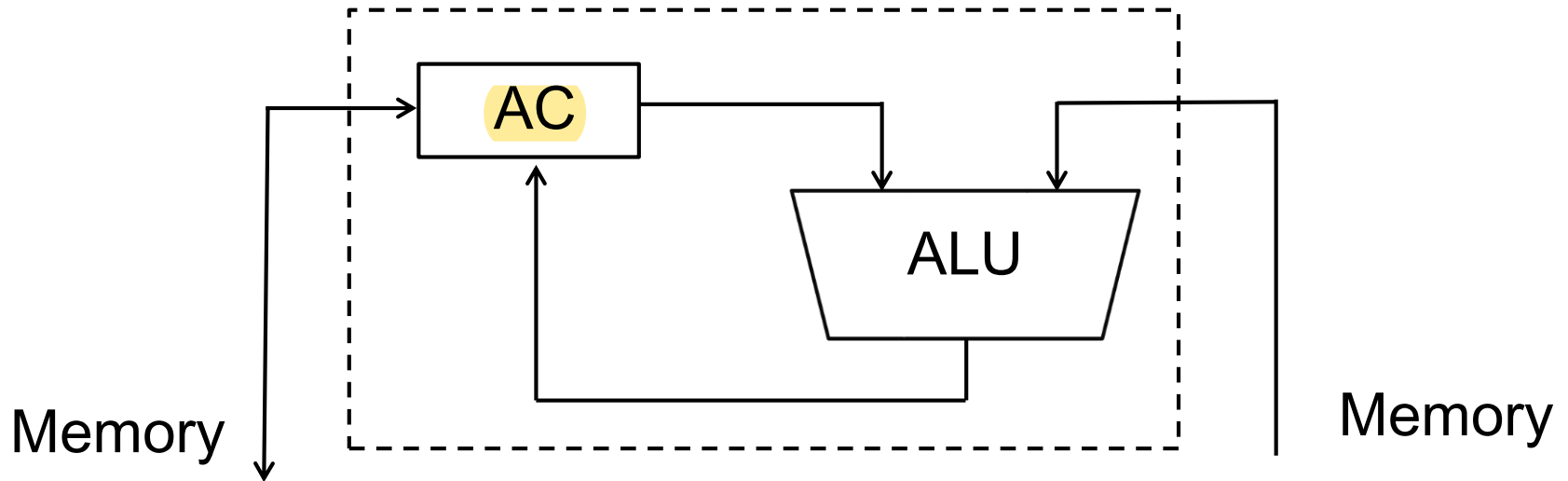
Add



Pop C



Accumulator ISA



□ Single-operand architecture ($C = A + B$;))

Load A // $AC \leftarrow M[A]$

Add B // $AC \leftarrow AC + M[B]$

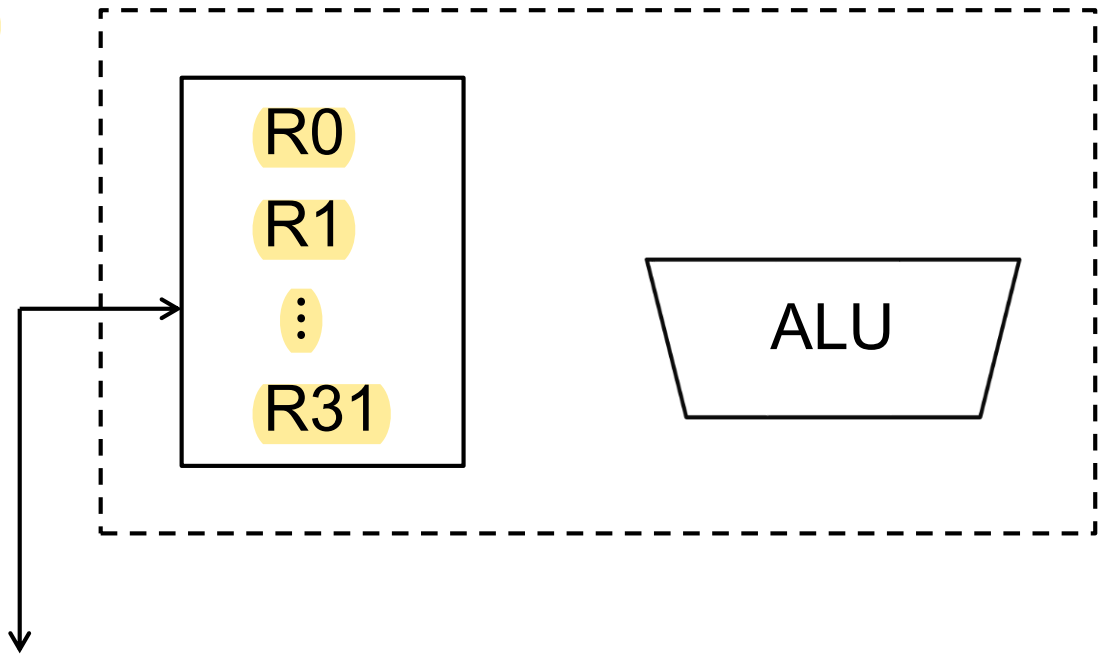
Store C // $AC \rightarrow M[C]$

GPR ISA

General-purpose registers

(for executing applications; programmers or compilers)

Memory



❑ Registers as cache

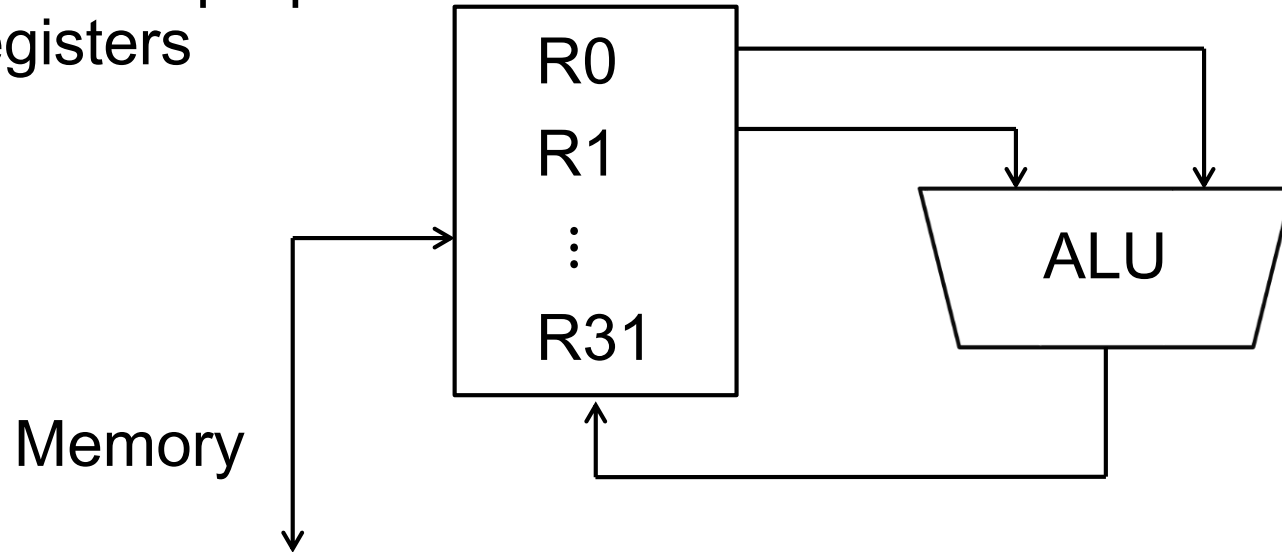
- Faster, reduce memory traffic, support parallelism

GPR Architectures

- ❑ Three types
 - Register-register architecture
 - Register-memory architecture
 - Memory-memory architecture

Register-Register Architecture

General-purpose
registers



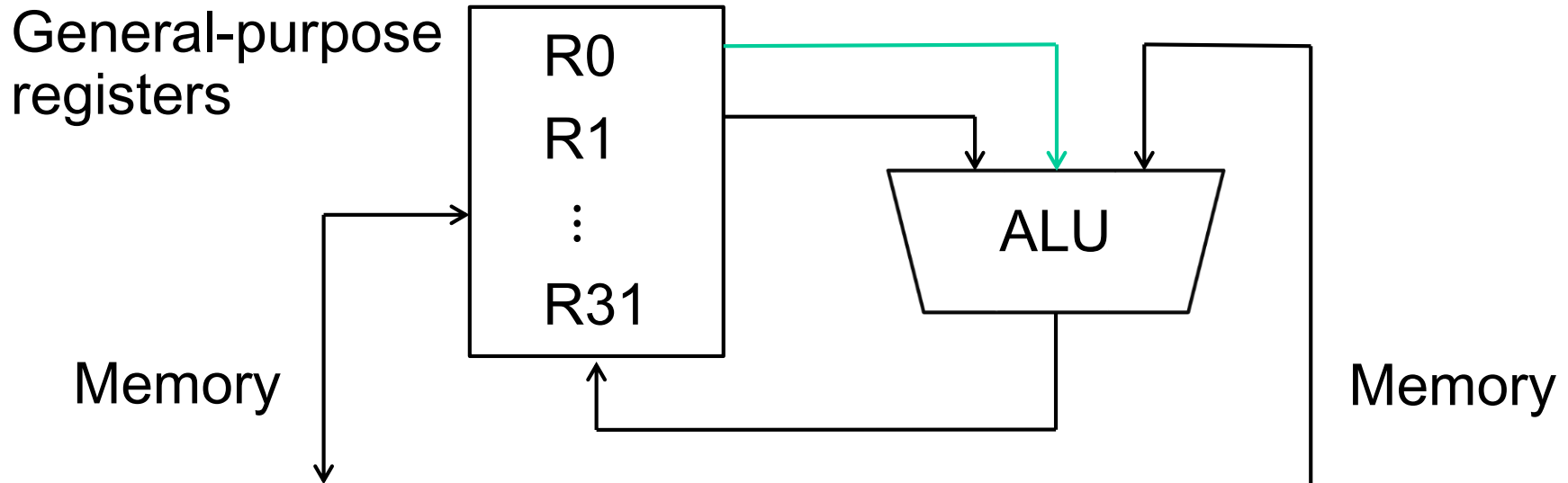
(C = A + B;)

Load R1, A	// R1 ← M[A]
Load R2, B	// R2 ← M[B]
Add R3, R1, R2	// R3 ← R1 + R2
Store R3, C	// R3 → M[C]

Register-Register Architecture

- ❑ “Load-store” architecture
 - Access memory only through load and store
 - All ALU instructions: register-based
- ❑ What is called “RISC” architecture today
 - For general-purpose computers
 - PowerPC, PA-RISC, MIPS, SPARC, Alpha
 - † Only exception is Intel (internally RISC)
 - For mobile embedded systems (e.g., ARM)
 - Competitive performance, small, low-power

Register-Memory Architecture



(C = A + B;)

Load	R1, A	// R1 ← M[A]
Add	R3, R1, B	// R3 ← R1 + M[B]
Store	R3, C	// R3 → M[C]

- ❑ Will R-M support "Add R3, R1, R2"?
- ❑ Same amount of work, fewer instructions, a little complex¹⁵

ISA Classes (Interfaces) (반복)

(Hennessy and Patterson, Computer Architecture, Morgan Kaufmann)

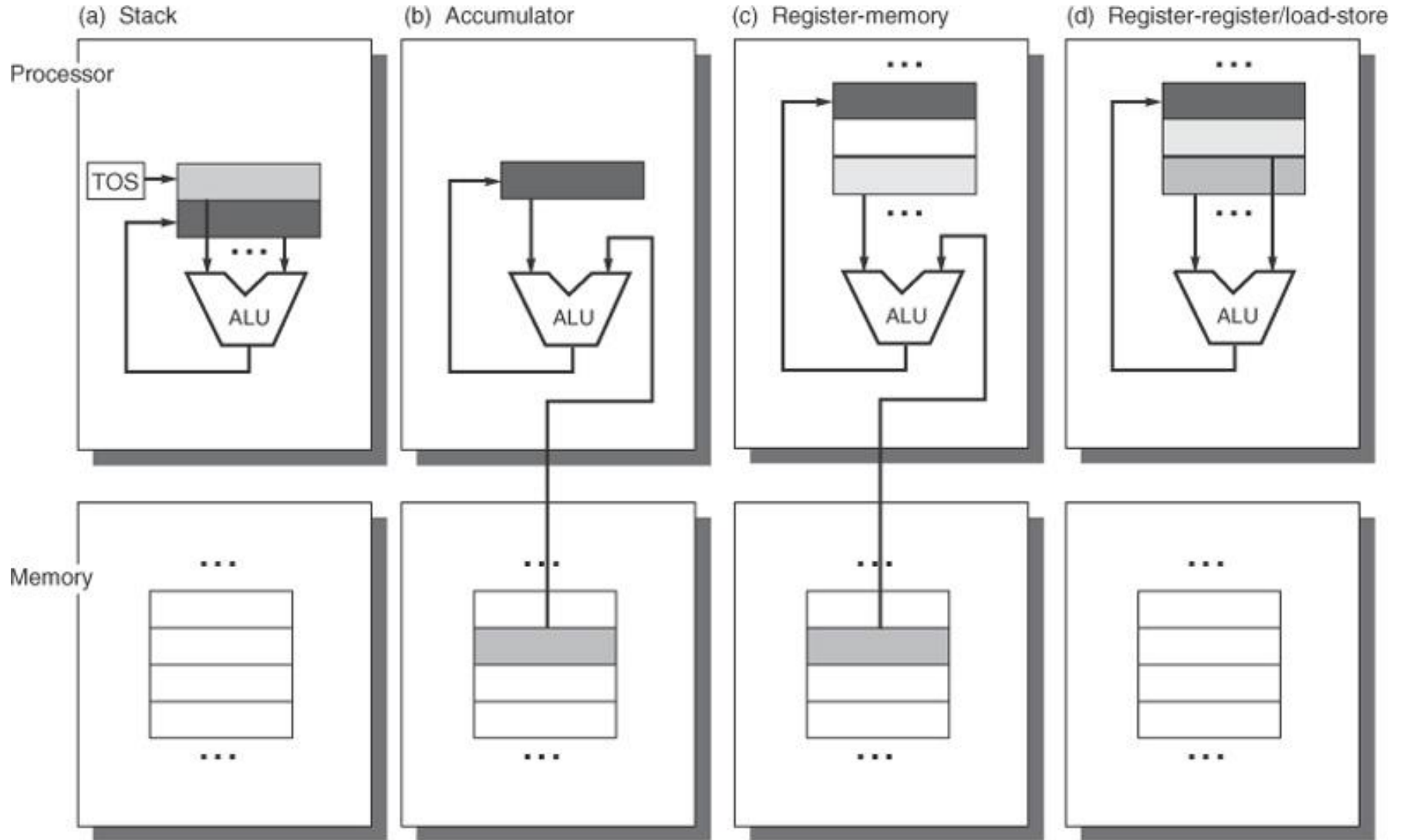
How to perform “arithmetic and logic” computation

$C = A + B;$ // A, B, C in memory

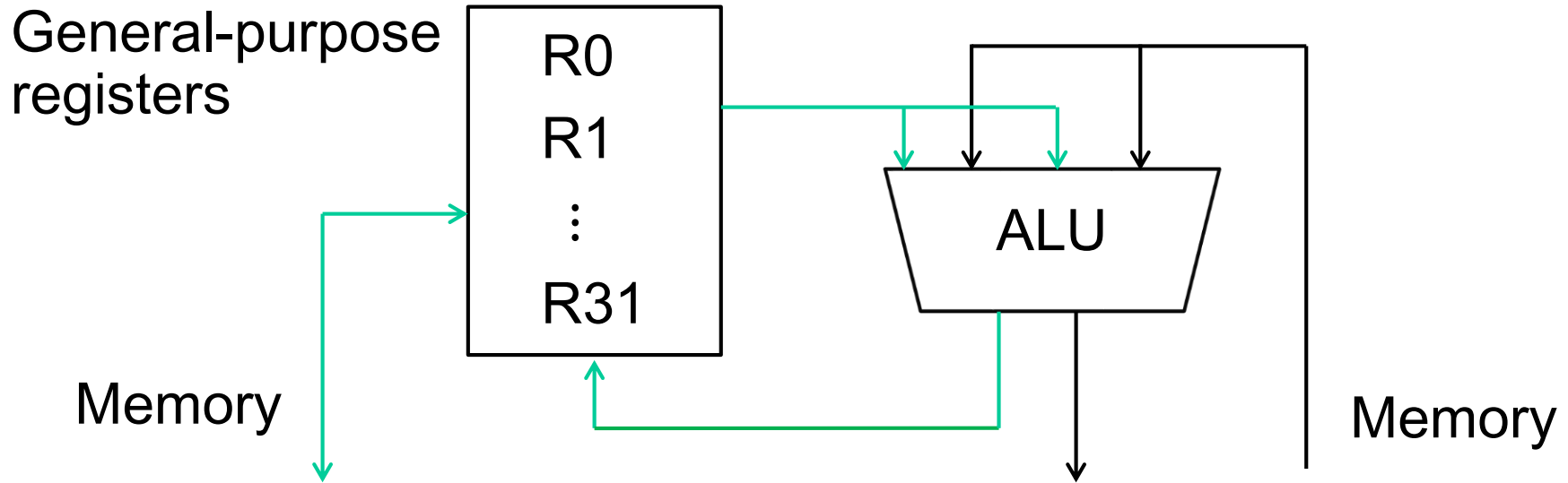
❑ Code sequence for four classes of instruction sets

Stack	Accumulator	GPR (Register-memory)	GPR (Register-register)
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

ISA Classes (Implementations) (참고)



Memory-Memory Architecture



(C = A + B;)

Add C, A, B // $M[C] \leftarrow M[A] + M[B]$

- ❑ Will M-M support "Add R3, R1, R2" or "Add R3, R1, B"?
- ❑ Same amount of work, fewer instructions, more complex
 - Obsolete (CISC)

GPR Architectures

(Hennessy and Patterson, Computer Architecture, Morgan Kaufmann)

- ❑ Three (or two) operands
 - 2-operand: destructive (Add R1, R2), shorter instruction
- ❑ Operands in ALU instructions

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-reg.	Alpha, ARM, MIPS, PowerPC, SPARC
1	2	Register-mem.	IBM 360/370, Intel 80x86, Motorola 68000
2	2	Memory-mem.	VAX (also has 3-operand formats)
3	3	Memory-mem.	VAX (also has 2-operand formats)

RISC and CISC

(Reduced Instruction Set Computer,
Complex Instruction Set Computer)

CISC

❑ Complex Instruction Set Computer

(in contrast with RISC appeared in market in 1980s)

- More (and complex) operations
- More (and complex) addressing modes
 - e.g., register-memory, memory-memory
- Diverse instruction format (consequently)
 - VAX: 1B to 53B long, 1 to x00 cycles to execute

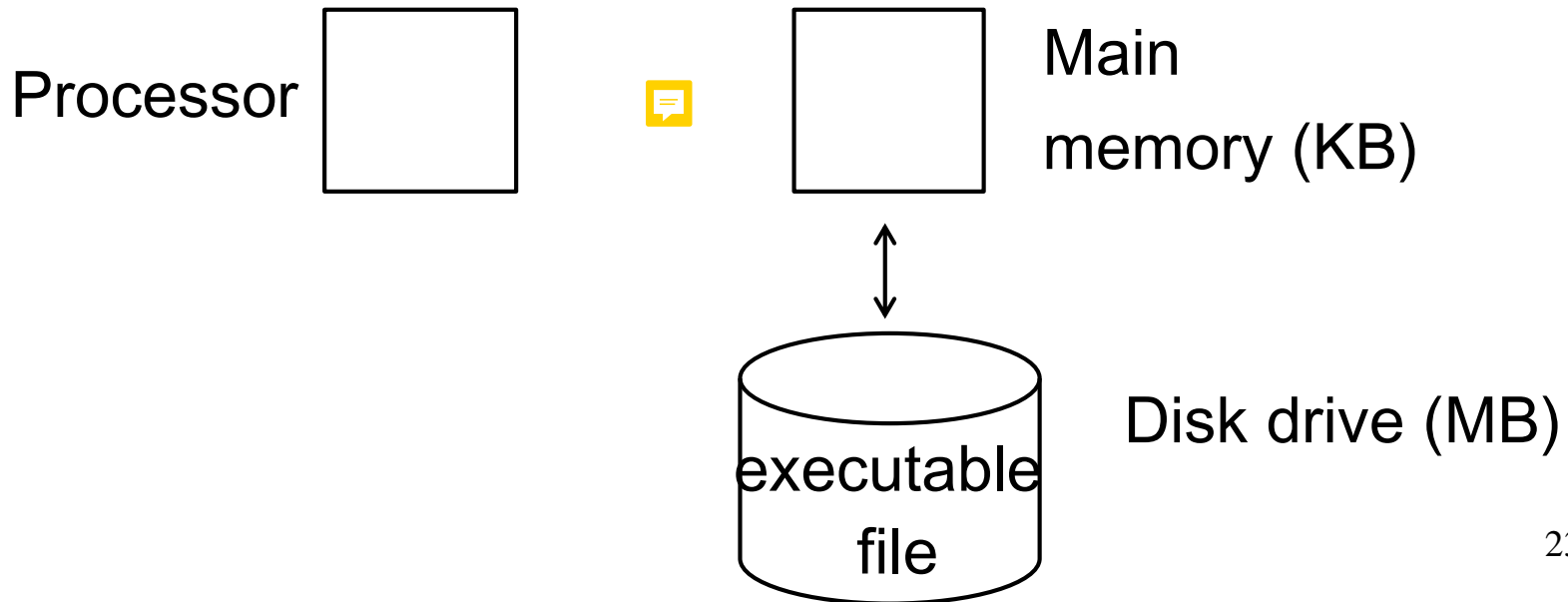
Rationale behind CISC

- ❑ Until around 1980, memory quite expensive and small
- ❑ Imagine a minicomputer in that era



Rationale behind CISC

- ❑ Until around 1980, memory quite expensive and small
 - Size of executable file determines performance
 - I/O time for code (not just data)
 - Processor designers react to improve code density



RISC and CISC

- ❑ Same work done; what is different?
- ❑ CISC: 상황에 따라 가장 compact 한 instruction 사용
 - Smaller executable files (high code density)

(C = A + B;)



RISC:

```
load  R1, A
load  R2, B
add   R3, R1, R2
store R3, C
```

* all instructions are 32-bit

CISC: 다음도 제공

```
add1  R1, A, B
add2  C, A, R2
add3  C, R2, B
add4  C, A, B
```

* memory address: register + offset
(A, B, C: not necessarily 32-bit)

RISC and CISC

❑ Vector add example

C code: for (i=0; i < n; i++)
 C[i] = A[i] + B[i];

CISC: **addv** C, B, A, n

* memory address: register + offset
(A, B, C: not necessarily 32-bit)

RISC: **add R10, R0, R0** // clear R10
 load R1, R4(0) // A[i]
 load R2, R5(0) // B[i]
 add R3, R1, R2
 store R3, R6(0) // C[i]
 addi R4, R4, 4
 addi R5, R5, 4
 addi R6, R6, 4
 add R10, R10, 1
 bne R10, R11, -10 // n in R11

* all instructions are 32-bit

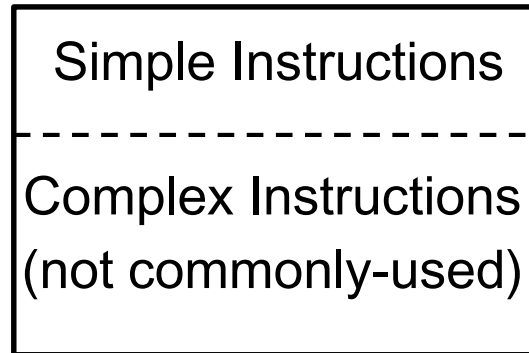
❑ Complex instructions for commonly-used patterns

Observations (early 1980s)

- ❑ Semiconductor technology - Moore's law
 - Memory becoming bigger and less expensive (exponentially)
 - Not true: size of executable \approx performance
- ❑ Is CISC (reducing the size of executable) valid ?

Observations (early 1980s)

CISC CPU die



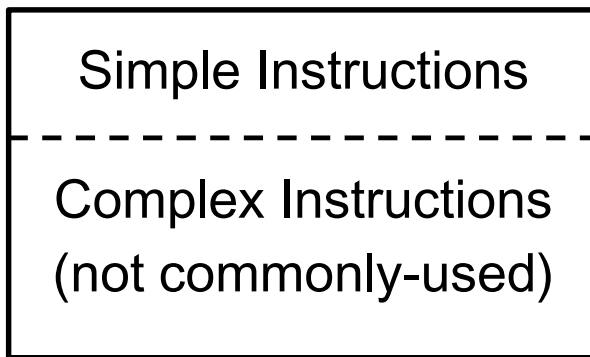
Difficult to reduce CPI, cct

- ❑ Is CISC valid? Are we using "die area" efficiently?
 - Additional operations and addressing modes
 - Require hardware (i.e., die size) to implement
 - Not used frequently
 - Resulting complex instruction formats
 - Efficient implementation (i.e., pipelining) difficult

Observations (early 1980s)

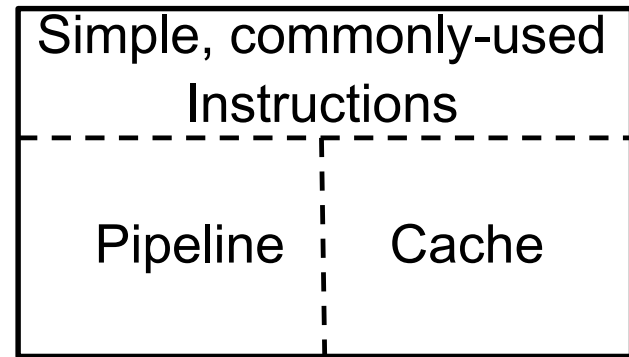
- ❑ What's the best way to use "die area"?
 - Provide only simple, commonly-used instructions
 - Use die area for efficient implementation (pipeline, cache memory)

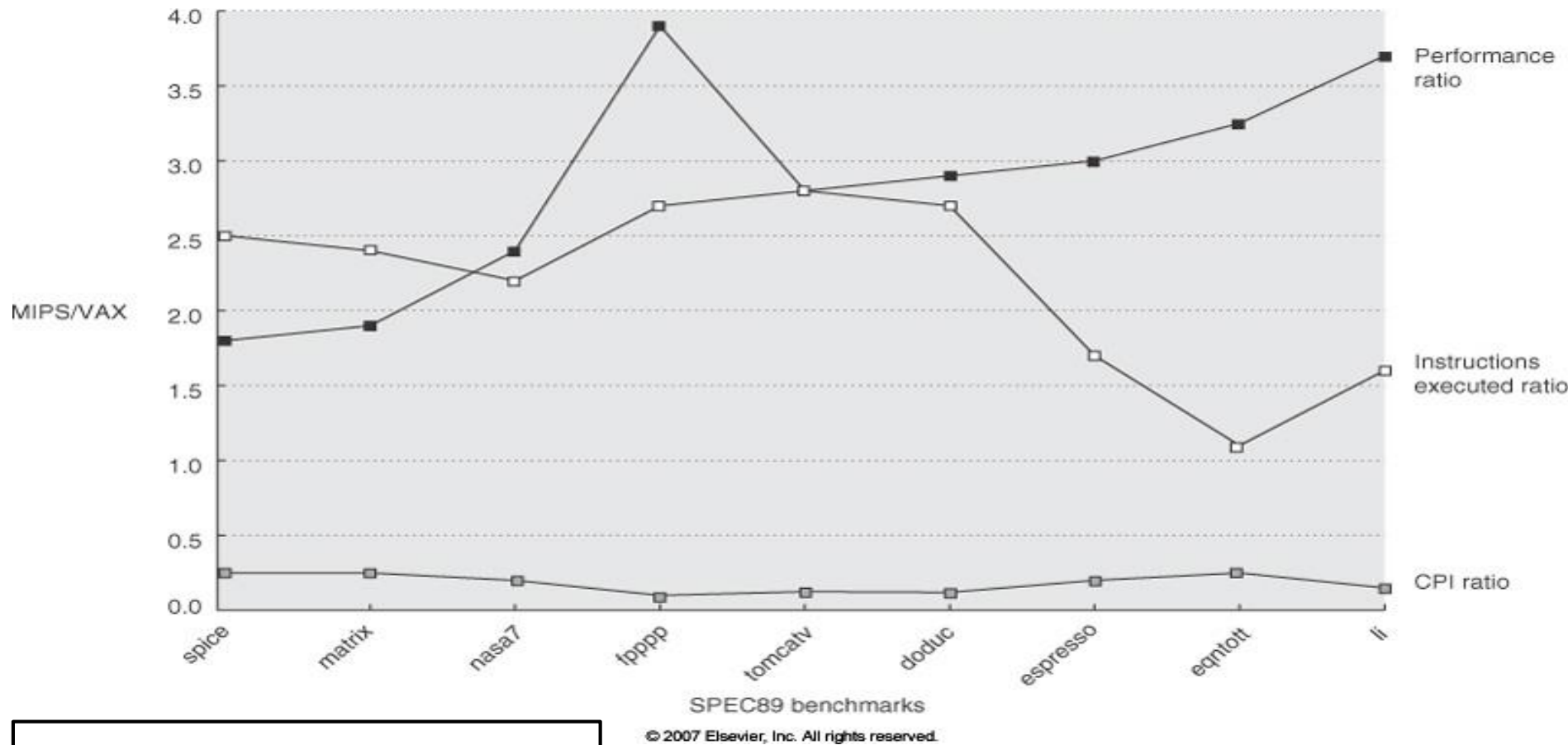
CISC CPU die



(difficult to reduce CPI, cct)

RISC CPU die





Same cct and die size

MIPS VAX7800

IC 2 :

CPI 1 :

RISC Established in 1980s

- ❑ Processor designers back to right performance model
 - $\text{CPU time} = \text{IC} \cdot \text{CPI} \cdot \text{clock cycle time}$
- ❑ Characteristics of RISC (32-bit)
 - Fewer operations
 - Fewer addressing modes
 - Fixed and easy-to-decode instruction format
 - Single cycle execution ($\text{CPI} \approx 1$)
 - Pipelining and cache memory
 - Use of optimizing compilers (to reduce IC)
 - Access memory only through Load and Store

GPR Architectures

(Hennessy and Patterson, Computer Architecture, Morgan Kaufmann)

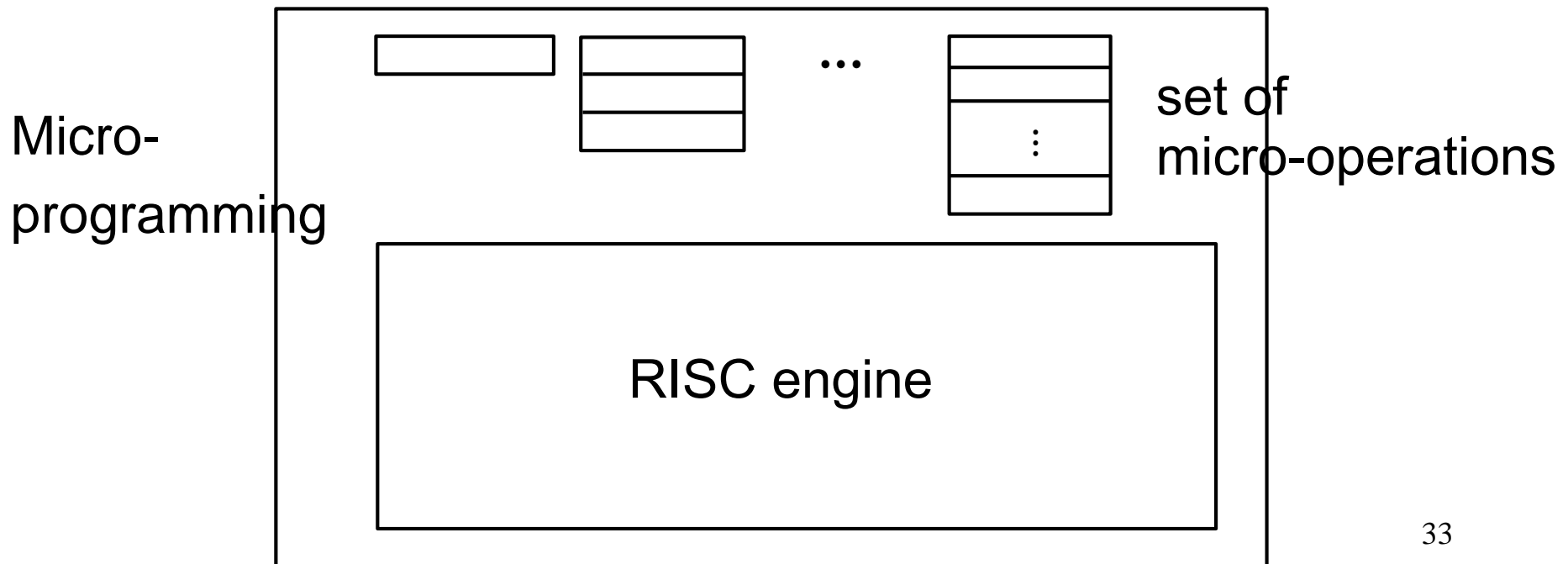
Type	Advantages	Disadvantages
Register-register (0,3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute.	Higher instruction count. More instructions and lower instruction density leads to a larger programs.
Register-memory (1,2)		
Memory-memory (2,2) or (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size. Large variation in work per instruction. Memory accesses create memory bottleneck . (Not used today.)

Instruction Set Architecture (반복)

- ❑ Processor design in 1970s (what we call CISC)
 - Constraint: memory expensive
- ❑ 1980s: renaissance of processor design
 - Semiconductor technology
 - Memory became cheaper (move to RISC style)
 - Open Unix operating system
 - High-level programming
- ❑ Emergence of powerful 32-bit RISC processors
 - PowerPC, PA-RISC, MIPS, SPARC, Alpha, ARM
 - Exception is Intel x86 ISA

Performance of x86 Architecture

- ❑ How did Intel manage to compete with RISC processors?
 - Fetch a complex x86 instruction
 - Execute a set of RISC-like instructions



Performance and ISA Design

Part 2

- 1) RISC vs. CISC
- 2) Amdahl's law
- 3) Power Limit and Multicores

Amdahl's Law

(Law of Diminishing Returns)

* 여기서부터는 Chapter 6
(multicores, multiprocessors)
내용이 포함되어 있음

Amdahl's Law

- ❑ Multiplication accounts for 80% of execution time
 - Let's improve multiplier performance

Others	20s
Mult.	80s

Improve	Mult 2x	Mult 4x	Mult 10x	Mult ∞
	20	20	20	20
Execution time	+ 40	+ 20	+ 8	+ 0
	= 60s	= 40s	= 28s	= 20s
Speedup	$\frac{100}{60}$	$\frac{100}{40}$	$\frac{100}{28}$	$\frac{100}{20}$

Amdahl's Law

- ❑ Example: multiply accounts for 80s out of 100s
 - How much improvement in multiply performance to get $5\times$ overall?

$$20 = \frac{80}{n} + 20 \quad // \text{ Can't be done!}$$

- ❑ Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

- ❑ Corollary: make the common case fast

Example

- ❑ Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if half of the 10 seconds is spent executing floating-point instructions?

$$5 + 5/5 = 6 \text{ seconds}$$

Example

- We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?

$$(100 - x) + x/5 = 100/3$$
$$\therefore x = 250/3$$

Misuse of Amdahl's Law (참고)

- ❑ Early parallel (or supercomputer) projects
 - Years of effort for HW, OS, compiler, applications

serial	20%
parallel	80%

Number of processors	4	16	64	∞
	20	20	20	20
Exec. time	+ 20	+ 5	+ 1.3	+ 0
	= 40	= 25	= 21.3	= 20
Speedup	$\frac{100}{40}$	$\frac{100}{25}$	$\frac{100}{21.3}$	$\frac{100}{20}$

- ❑ Amdahl's law: curse to parallel computer projects?

🗨️ Amdahl's Law and ISA Design

- ❑ Interpretation by RISC designers
 - Amdahl: make common case faster (common sense)
- ❑ What is a good ISA? Level of abstraction adequate?
 - Complex question
- ❑ Common sense: make common case faster
 - Common case: simple operations (are you sure?)
 - RISC: common sense approach
 - Explainable design!

RISC

- ❑ Make common case faster (Amdahl, common sense)
 - Common case: simple operations
- ❑ How do we prove the claim?
 - Analyze benchmark programs
 - Compile and count frequently used operations
 - † Use RISC compiler or CISC compiler?
 - If you do the above, you end up with RISC style ISA

RISC

- ❑ Performance model under today's technology

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{cct}$$

- ❑ RISC strategy

- IC: smart compilers (code density low)
- CPI: pipeline, cache memory
- Clock cycle time: pipeline, cache memory

RISC

- ❑ All newly-designed processors since 1980
 - Proven technology
 - Recent challenges
 - Multimedia applications (Chapter 3)
- 🗨 † Evolution of general-purpose processors
 - Diminishing return on investment
 - Power consumption
- ❑ Parallel revolution (around 2006)
 - All desktop/server companies ship multicore processors

Measuring and Modeling IC, CPI

(skip)

- ❑ Profile-based approach
 - Dynamic execution profile
 - IC for given input
- ❑ Trace-driven approach
 - Trace of memory references
 - Memory system simulation (CPI)
- ❑ Execution-driven approach
 - Processor pipeline (CPI)
 - May combine with memory system simulation

Power Limit and Multicores

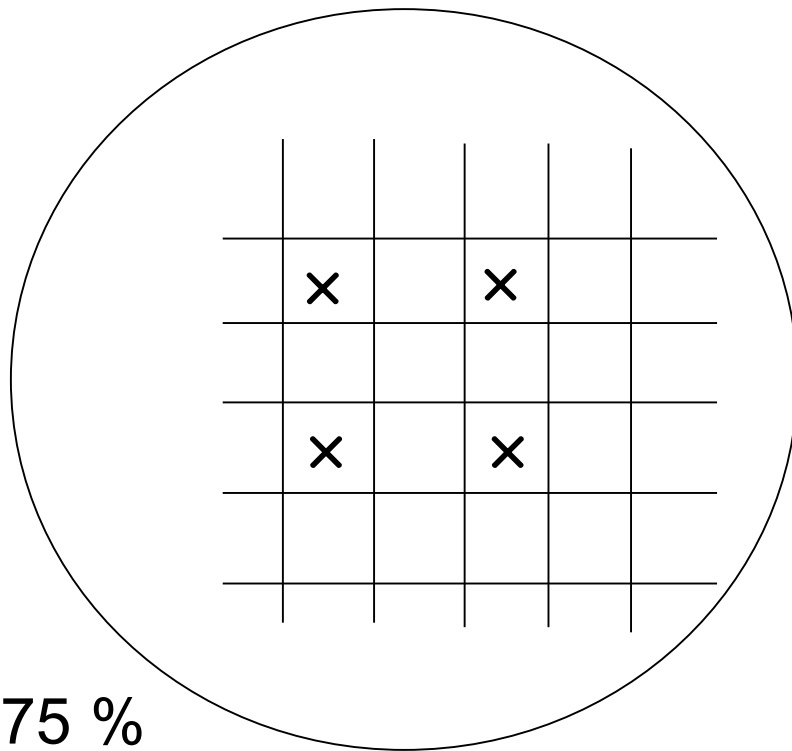
Trends in Technology (반복)

- ❑ Processor technology
 - Transistor density: 35%/year
 - Die size: 10-20%/year (pipelines, cache memory)
 - Integration overall: 40-55%/year
- ❑ DRAM capacity: 25-40%/year
- ❑ Flash memory capacity: 50-60%/year
 - 15-20X cheaper/bit than DRAM
- ❑ Magnetic disk capacity: 40%/year
 - 15-25X cheaper/bit than Flash
 - 300-500X cheaper/bit than DRAM

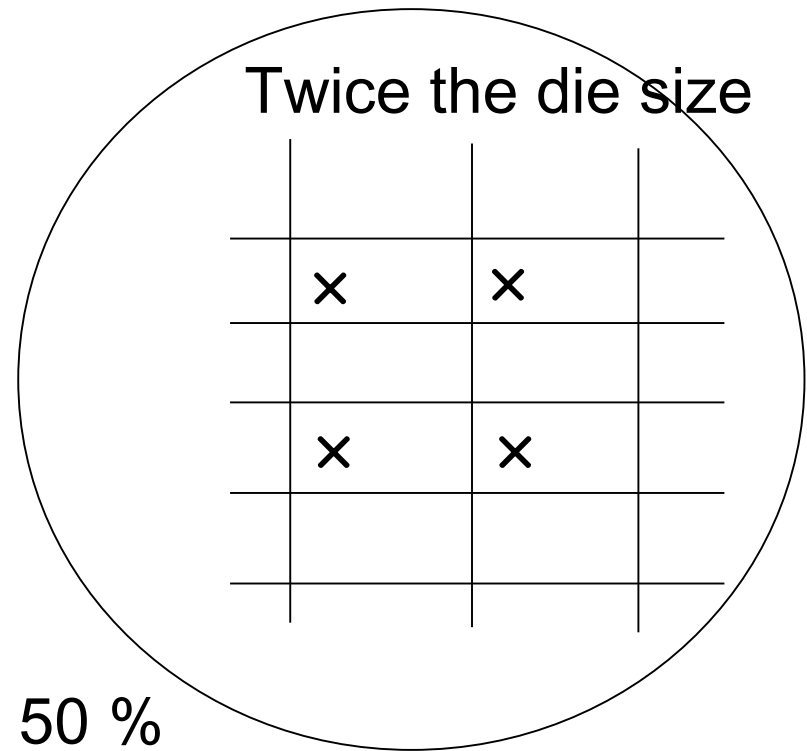
Speed
vs.
Capacity

Yield: Proportion of Good Die

- ❑ Smaller transistors, larger die sizes
 - Increased power consumption in processor



75 %



50 %

(in reality, can be much lower)

Integrated Circuit Cost (skip)

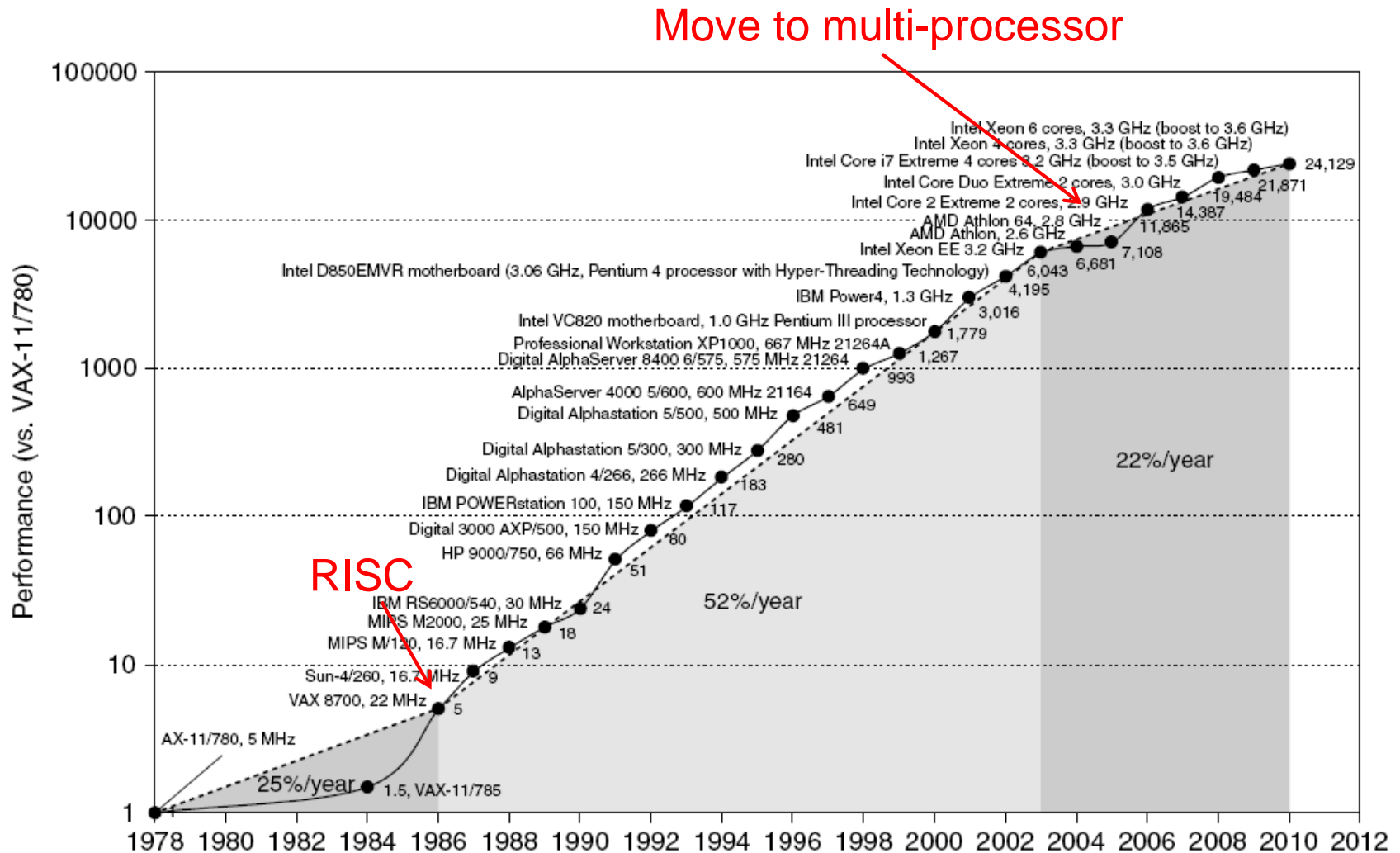
$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

$$\text{Dies per wafer} \approx \text{Wafer area} / \text{Die area}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area} / 2))^2}$$

- ❑ Nonlinear relation to area and defect rate
 - Wafer cost and area are fixed
 - Defect rate determined by manufacturing process
 - Die area determined by architecture and circuit design

Single Processor Performance (반복)



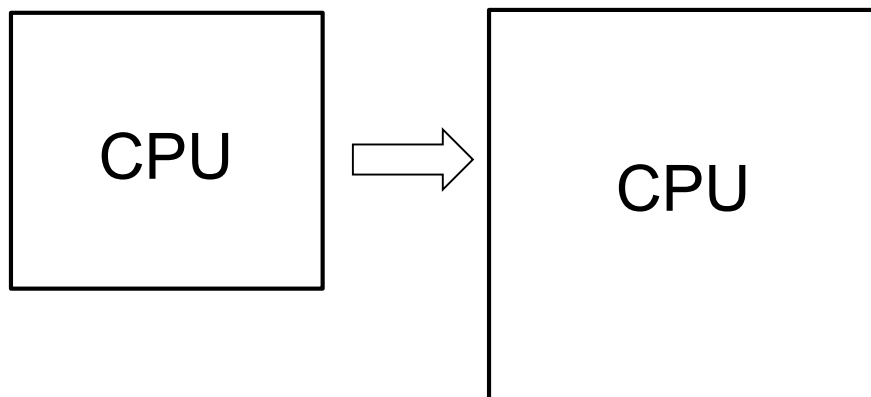
Game of Response Time

- ❑ Before 1980: 25%, technology
- ❑ 1980 - 2002: 52%, technology + architecture (RISC)
- ❑ Since 2002: 20%
 - Diminishing return on investment
 - Available instruction-level parallelism
 - Long memory latency
 - Power limit

Diminishing Return

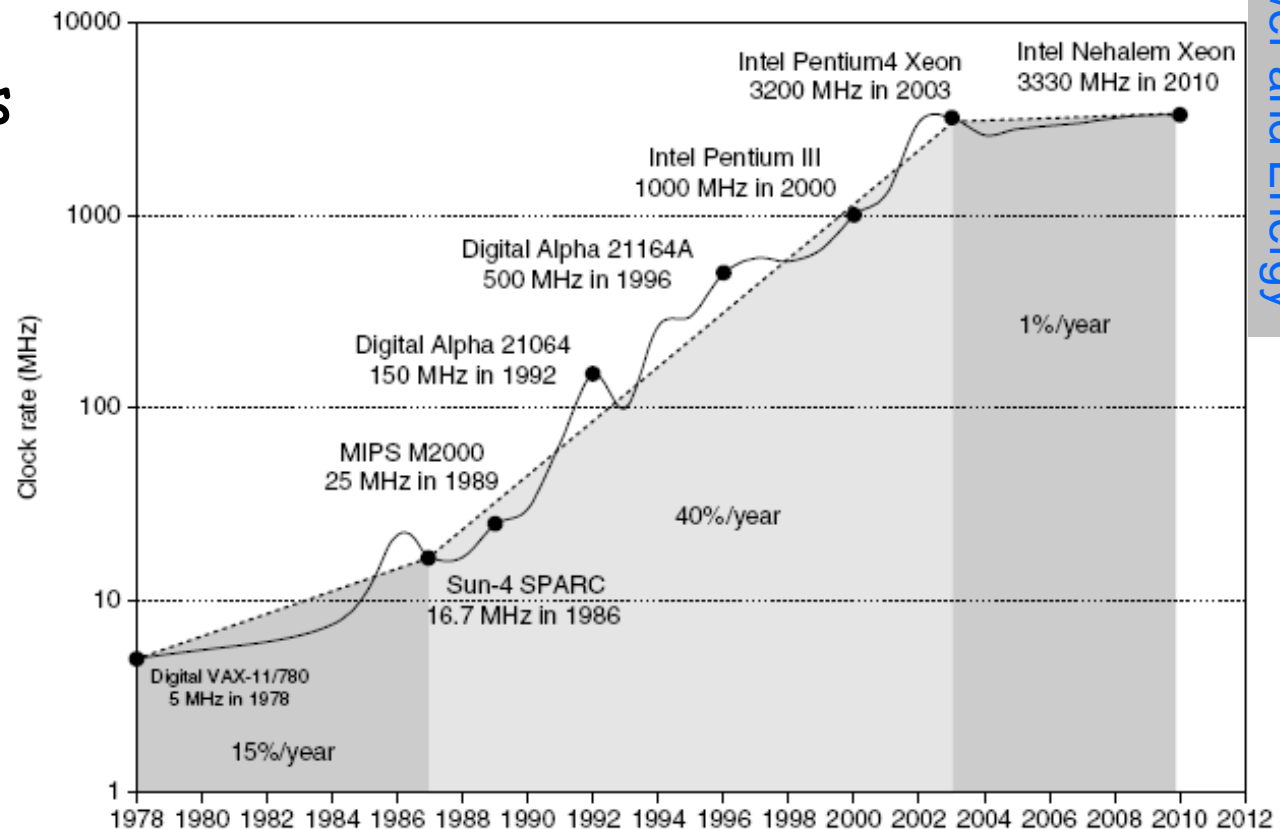
- ❑ Smaller transistors, increased die size (pipeline, cache)
- ❑ Diminishing return on investment in 1990s
 - Engineering perspective
 - Business perspective
- ❑ Pollack's rule (die size vs. performance/power)

Die size	1 : 2
Performance	1 : $\sqrt{2}$
Power	1 : 2

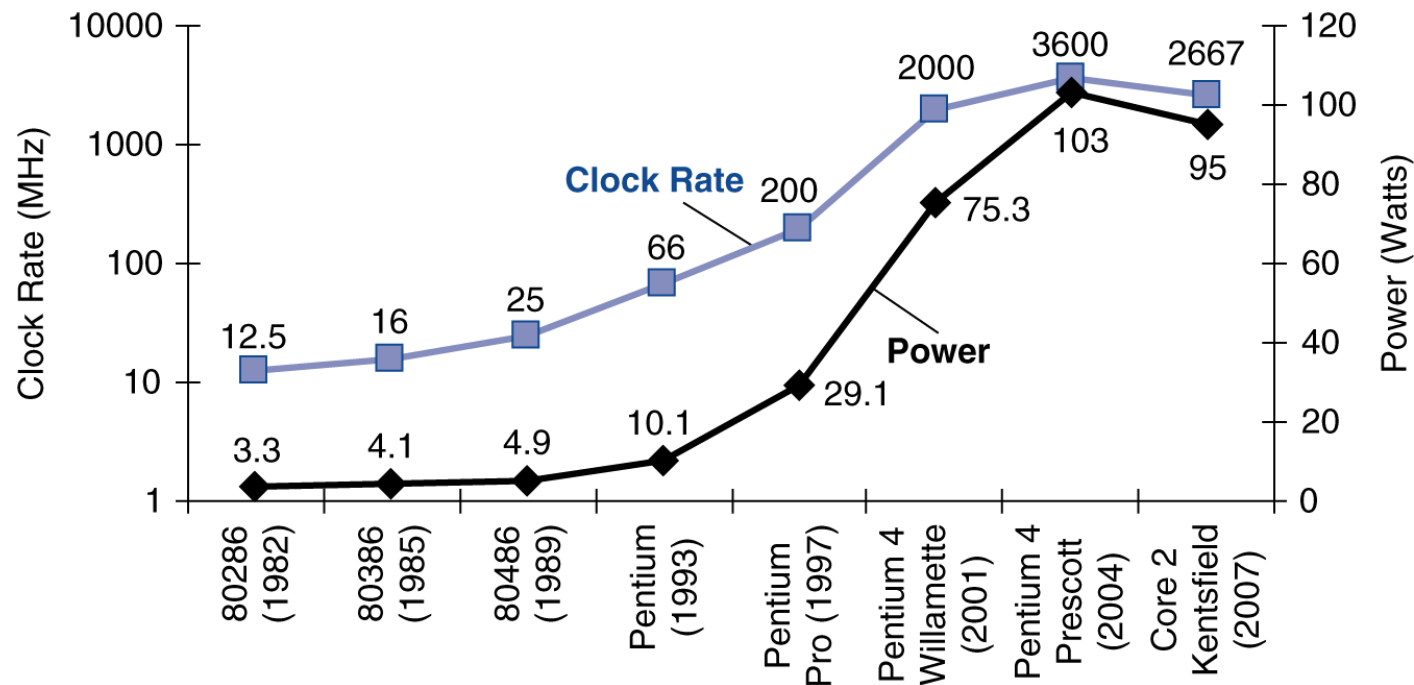


Power Wall (참고)

- ❑ 3.3 GHz Intel Core i7 consumes 130 W
- ❑ Heat must be dissipated from 1.5 x 1.5 cm chip
- ❑ This is the limit of what can be cooled by air



Power Wall (참고)



□ In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

× 30

5V → 1V

× 1000

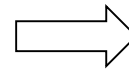
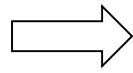
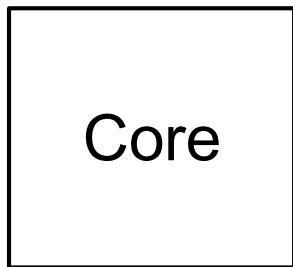
Reducing Power

- ❑ The power wall
 - We can't reduce voltage further
 - We can't remove more heat
- ❑ How else can processor designers improve performance?
- ❑ As of 2006, all desktop/server companies ship multicores
 - More than one processor per chip

Multicore Processors



Single core (same technology)



Die size	1 : 2
Performance	1 : $\sqrt{2}$
Power	1 : 2

Die size	1 : 4
Performance	1 : 2
Power	1 : 4

Die size	1 : 4
Performance	1 : 4
Power	1 : 4



Multicore

(Shared-Memory) Multicores

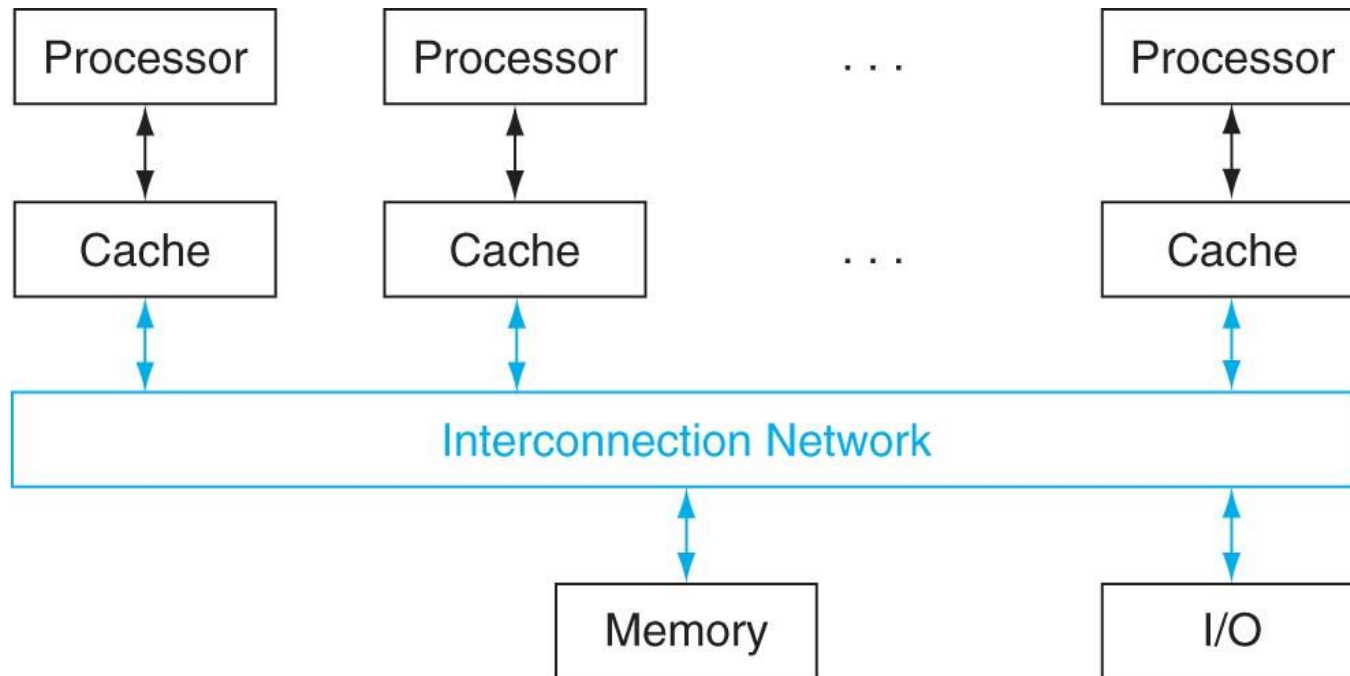


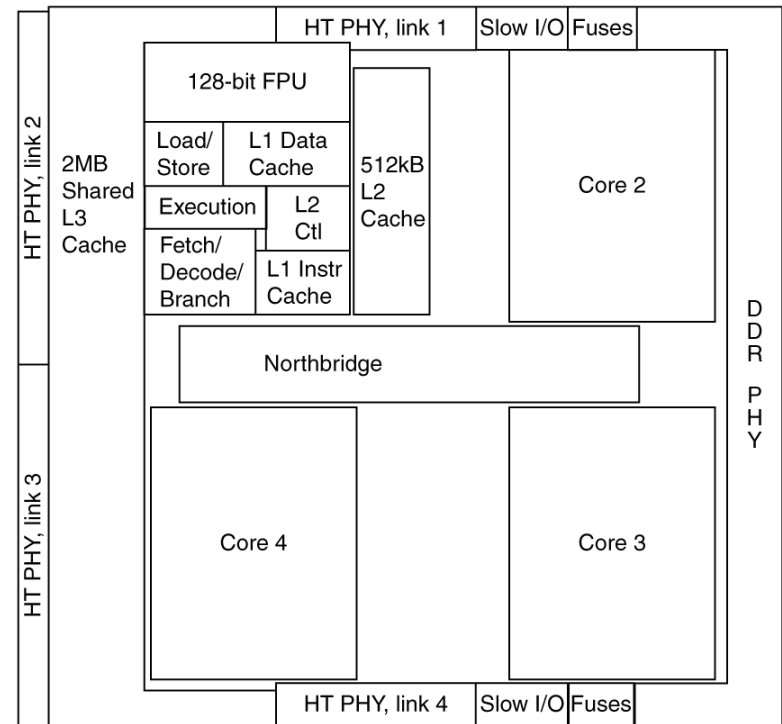
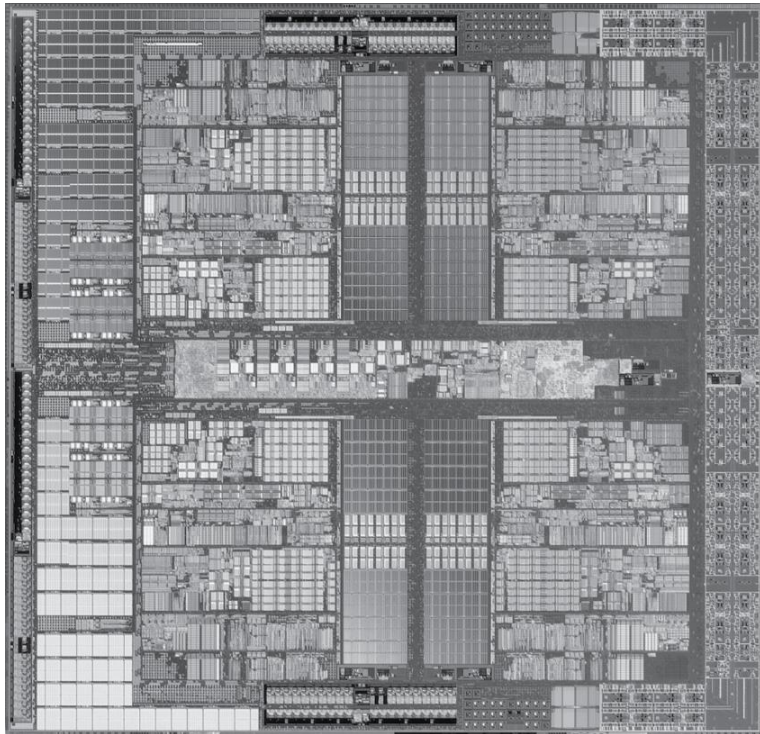
FIGURE 7.2 Classic organization of a shared memory multiprocessor. Copyright © 2009 Elsevier, Inc. All rights reserved.

- ❑ Single die
- ❑ Core = processor

Inside the Processor (반복)

(Hennessy and Patterson slide, Computer Organization and Design, Morgan Kaufmann)

❑ AMD Opteron X4 (Barcelona): 4 processor cores



Parallel Revolution

- ❑ Instead of continuing to decrease response time of single program on single processor, IT industry tied its future to parallel computing (2006)
 - In the past, program performance doubled every 18 months due to innovations in hardware, architecture, compiler
 - Today, to improve response time, programmers must rewrite programs to exploit multiple processors
- ❑ Dream: Higher performance with multiple processors
 - However, explicitly rewriting programs to be parallel has been “third rail” of computer architecture

Parallel Revolution

- ❑ Will programmers finally successfully switch to explicitly parallel programming?
 - Processor in your desktop likely to be multicore
- ❑ Why is parallel programming difficult?
 - It is performance programming
 - Partitioning, load balancing
 - Coordination (scheduling, synchronization)
 - Communication overhead
 - Existence of serial code

ISA Design (다시보기)

- ❑ Make common case faster (Amdahl, common sense)
 - Common case: simple operations
 - ❑ How do we prove the claim?
 - Analyze benchmark programs
 - Compile and count frequently used operations
 - † Use RISC compiler or CISC compiler?
 - Add necessary and justifiable instructions
 - ❑ If you do the above, you end up with RISC style ISA
 - Let's look into MIPS instruction set
- (RISC instruction set 감상할 만큼 전공지식 준비되었음)

Homework #6 (see Class Homepage)

- 1) Write a report summarizing the materials discussed in Topic 1-2
- 2) Read the textbook section 2.21 and write a summary report (itemization is good enough; about 3 pages) - you can obtain the section 2.21 by clicking "online companion materials" above and then clicking "Historical perspectives and further reading" on top-left

** 문장으로 써도 좋고 파워포인트 형태의 개조식 정리도 좋음

□ Due: see Blackboard

- Submit electronically to Blackboard

Class Topics (클래스 홈페이지 참조)

- ❑ Part 1: Fundamental concepts and principles
- ❑ Part 2: 빠른 컴퓨터를 위한 ISA design
 - Topic 1 Computer performance and ISA design (Ch. 1)
 - Topic 2 RISC (MIPS) instruction set (Chapter 2)
 - 2-1 ALU and data transfer instructions
 - 2-2 Branch instructions
 - 2-3 Supporting program execution
 - Topic 3 Computer arithmetic and ALU (Chapter 3)
- ❑ Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)