Computer Performance:

Part 2

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

Slides by Hennessy and Patterson (modified)

ISA Design Issues

- Operations (opcode)
  - How many, what types of instructions
    - ALU, data transfer, branch operations, others
- Operands
  - How many operands (in ALU instructions)?
    - Number of memory operands
  - How to specify the locations of operands
    - Addressing modes: register, direct, immediate, …
  - Operand types (data types – more later)
- Instruction encoding
  - How to pack all in words

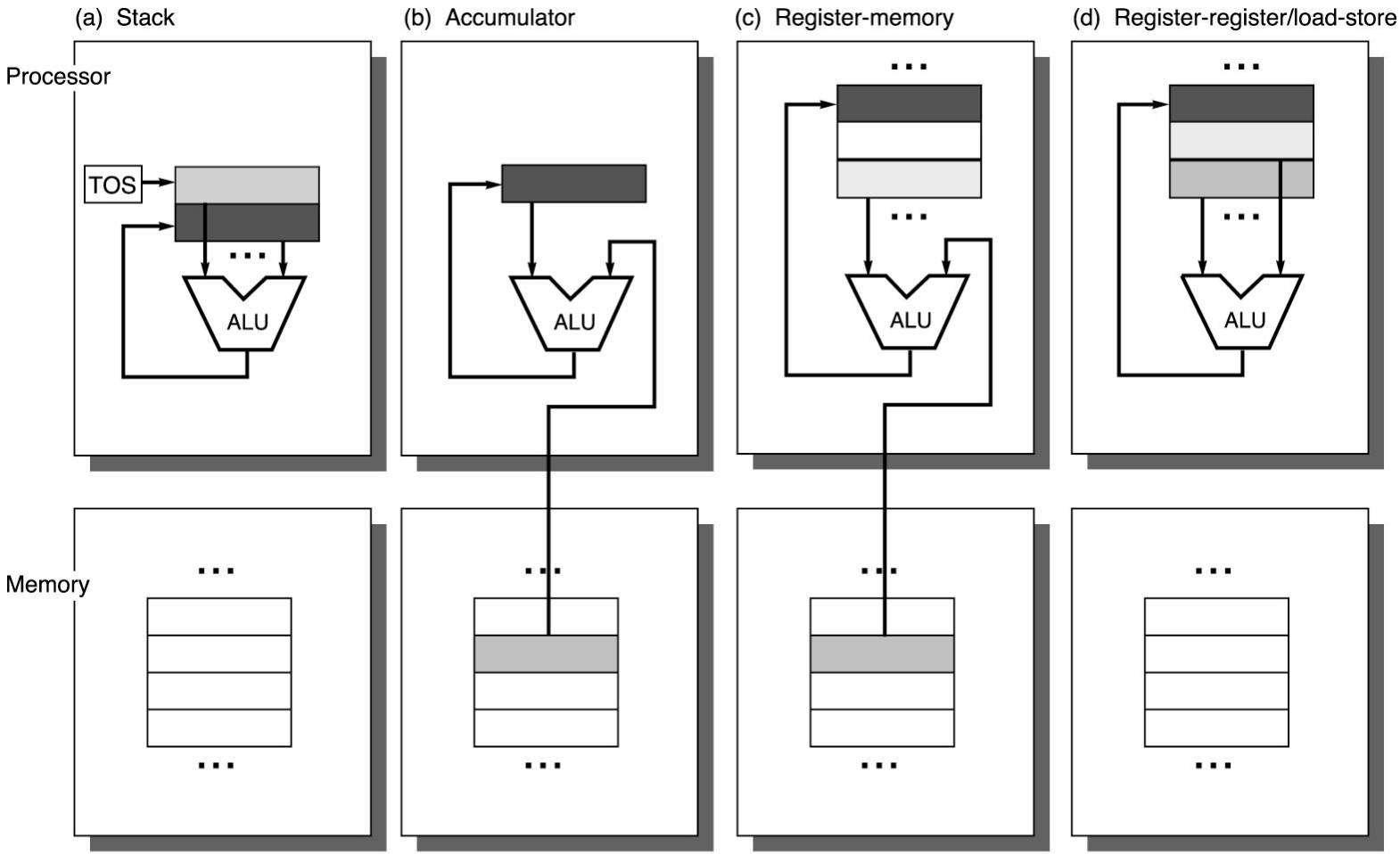
ISA Classes

- Number of operands in ALU instructions

Stack	Accumulator	Register (register-memory)	Register (load-store)
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

Figure 2.2 The code sequence for C = A + B for four classes of instruction sets. Note that the Add instruction has implicit operands for stack and accumulator architectures, and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure 2.1 shows the Add operation for each class of architecture.

Four ISA Classes



General-Purpose Register Architecture

- Advantages
  - Registers as cache
    - Faster
    - Reduce memory traffic
  - Support parallelism, reordering

General-Purpose Register Arch.

- Number of memory operands in ALU instructions
- Typical combinations (two or three operands)

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Figure 2.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers.

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 today's general-purpose computers
    - PowerPC, PA-RISC, MIPS, SPARC, Alpha
    - Only exception is Intel Pentium
      - † Internally RISC
  - For today's mobile embedded systems
    - Competitive performance, small, low-power
    - e.g., ARM

RISC versus CISC

- CISC (in contrast with RISC appeared in market in 1980s)
  - More (and complex) operations
    - e.g., vector add instruction
  - 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
  - Until around 1980, memory quite expensive
  - Size of executable file determine system performance
    - Code density (e.g. "vector add" example)

RISC and CISC

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

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

```
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, -9  // n in R11
```

* all instructions are 32-bit

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

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

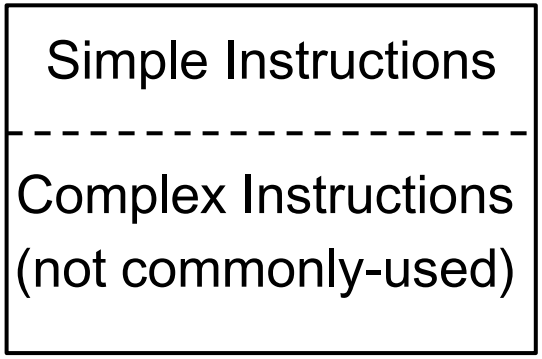
* memory address: register + offset

(A, B, C: not necessarily 32-bit)

Observations (around 1980)

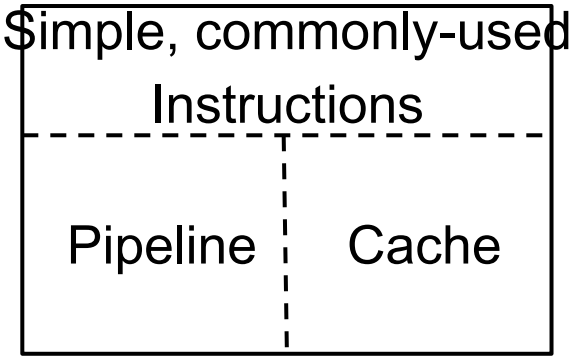
- Semiconductor technology – Moore's law
  - Memory become inexpensive
  - Not true: execution time ≈ code density
- Is CISC valid; are we using "real estate" efficiently?

CISC CPU die



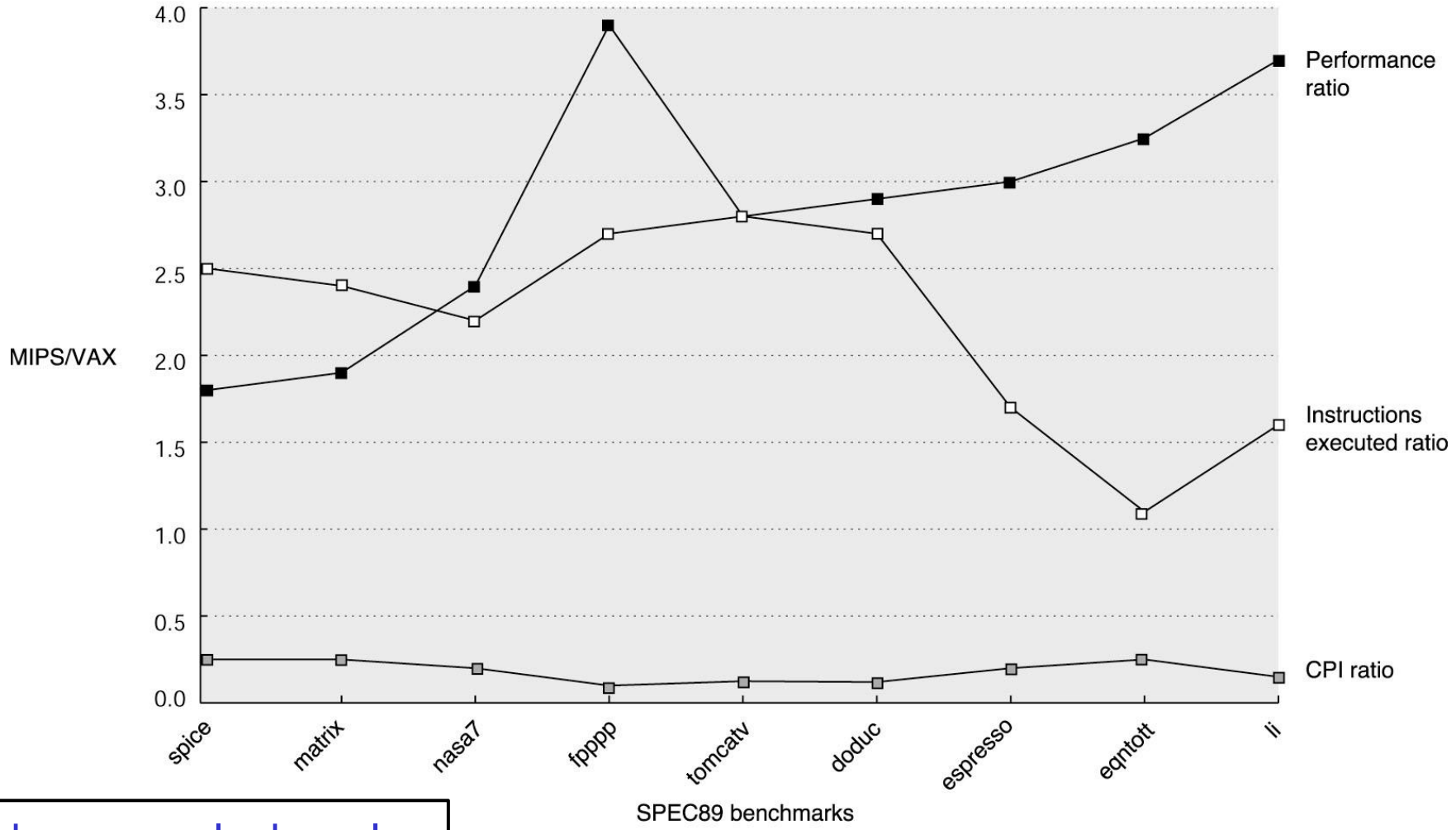
Difficult to reduce CPI, cct

RISC CPU die



Observations (around 1980)

- Is CISC valid, are we using "real estate" efficiently?
  - Additional operations, addressing modes
    - Require hardware (i.e., die size) to implement
    - Not used frequently
  - Resulting complex instruction format
    - Efficient implementation become difficult
      - † Especially not good for pipelining
- What's the best way to use "real estate"?
  - What if provide only simple, commonly-used instructions
  - Use die area for efficient speedup (pipeline, cache)
  - Rely on compiler to synthesize complex operations



Under same clock cycle

	MIPS		VAX7800
IC	2	:	1
CPI	1	:	6

RISC – Established in 1980s

- Back to fundamentals of performance
  - Execution time = IC • CPI • clock cycle time
- Characteristics of RISC (32-bit)
  - Fewer operations
  - Fewer addressing modes
  - Fixed and easy-to-decode instruction format
  - Single cycle execution (CPI ≈ 1)
    - Pipelining and cache
  - Use of optimizing compilers (to reduce IC)
  - Access memory only through Load and Store

GPR Architectures

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

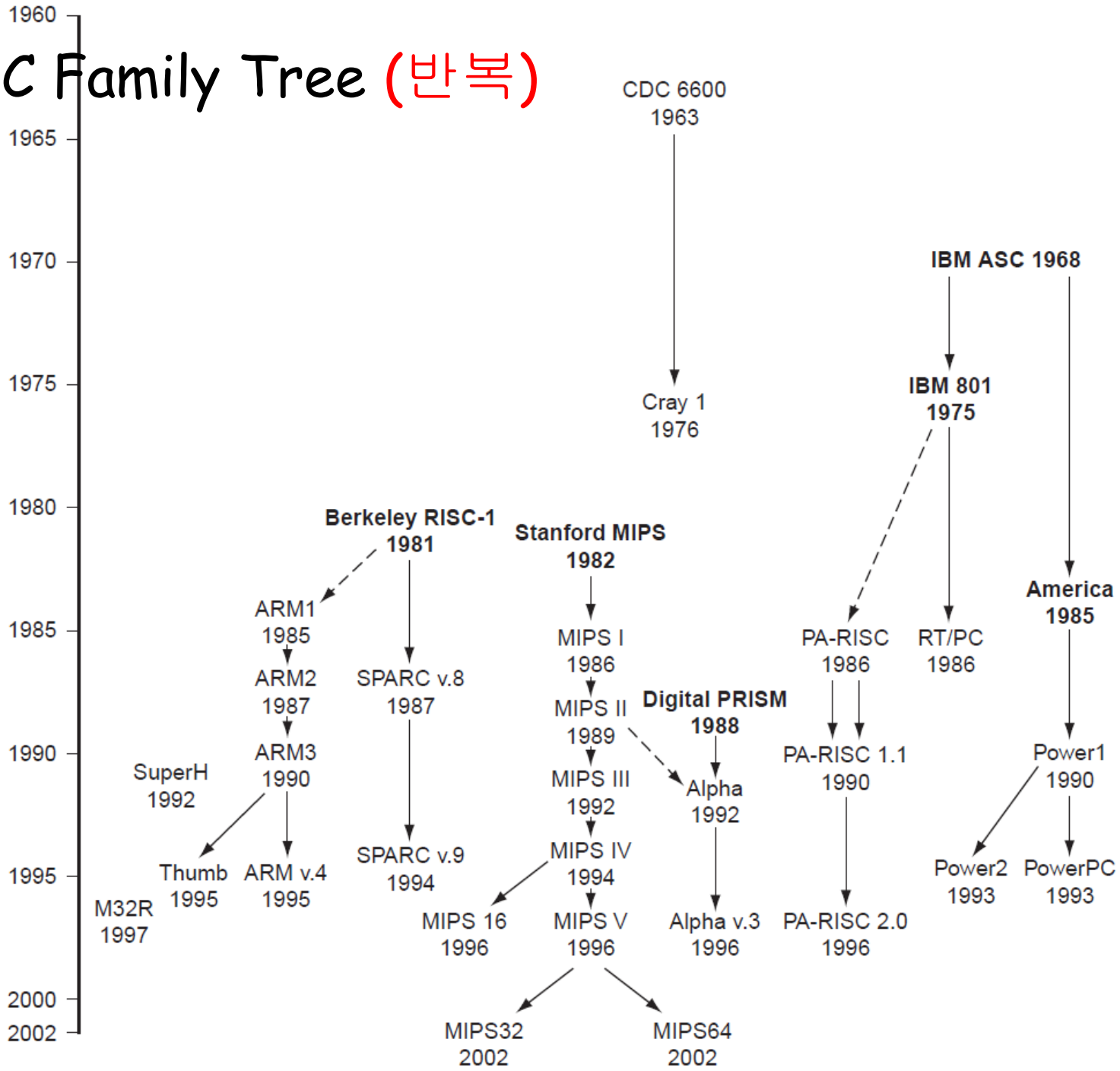
Figure 2.4 Advantages and disadvantages of the three most common types of general-purpose register computers.

Instruction Set Architecture (반복)

- Processor design in 1970s: what we call CISC
  - Constraint: memory expensive
- 1980s: renaissance of processor design (RISC style)
  - Semiconductor technology
    - Memory become cheaper
  - Open Unix operating system
  - High-level programming
- Emergence of powerful 32-bit RISC processers
  - PowerPC, PA-RISC, MIPS, SPARC, Alpha, ARM

† Exception is Intel Pentium

RISC Family Tree (반복)



Performance of x86 Architecture

- How did Intel manage to compete with RISC processors?
  - Fetch x86 instructions
  - Translate them into internal RISC-like instructions
  - These micro-operations are executed by pipeline

ISA Classes Again

- Given RISC, is stack or accumulator architecture obsolete?
  - See the "Dalvik" controversy
    - † Sort of mainstream computer project you might try

Amdahl's Law
(Law of Diminishing Returns)

* 여기서부터는 chapter 7
(multicores, multiprocessors)
내용이 포함되어 있음

Amdahl's Law

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

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

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!
- 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?
- 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?

Amdahl's Law

- Parallelism and speed up
  - Curse to parallel programs
  - † Multicore processor: different task for each core
  - † Natural parallelism in TP, web server
    - ⇒ Multiprocessor systems, multithreading
    - ⇒ Throughput rather than response time
  - † Today's supercomputers: extensive parallel processing
- RISC common-sense strategy: "make common case faster"
  - Common case: simple operations
  - CPU time = IC * CPI * clock cycle time
    - † IC: smart compiler
    - † CPI: pipeline, cache
    - † Clock cycle time

ISA Design

- What is a good ISA?
  - Is level of abstraction adequate?
    - † Complex question
- Heuristic: make common case faster (Amdahl)
    - † Common case: simple operations
  - RISC: common sense approach
  - Analyze benchmark programs
    - † Frequently used operations
    - † Absolutely necessary operations
- Ultimate measure

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

Measuring and Modeling IC, CPI

(참고자료)

- 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

RISC around 2000

- All newly-designed processors since 1980
  - Proven technology
  - Code density low
    - e.g., IBM, Hitachi
  - Recent challenges
    - Multimedia applications
    - Return on investment
    - Power consumption
- Parallel revolution (around 2006)
  - All desktop/server companies ship multicore processors

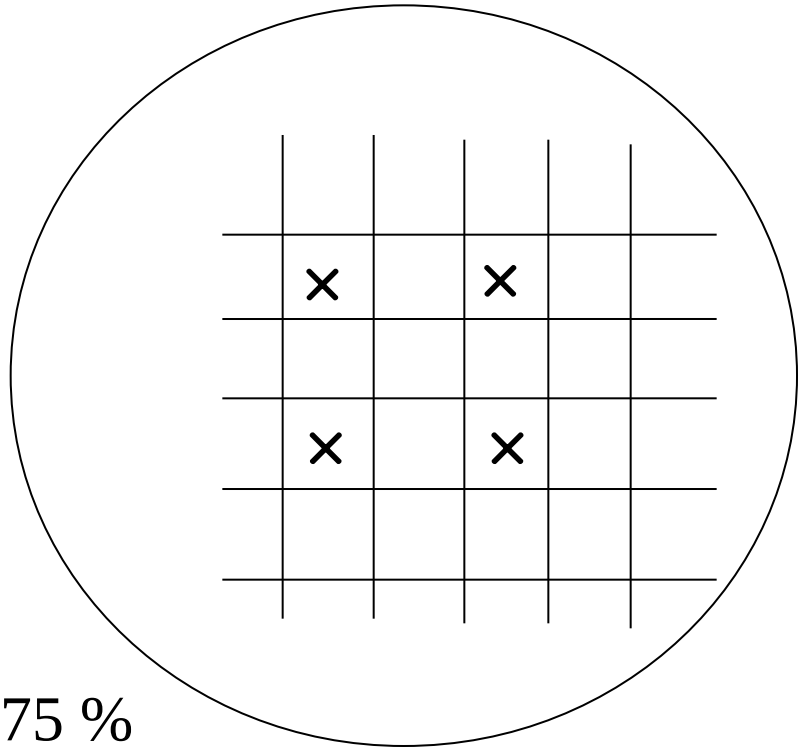
Power Limit and Multicores

Trends in Technology (반복)

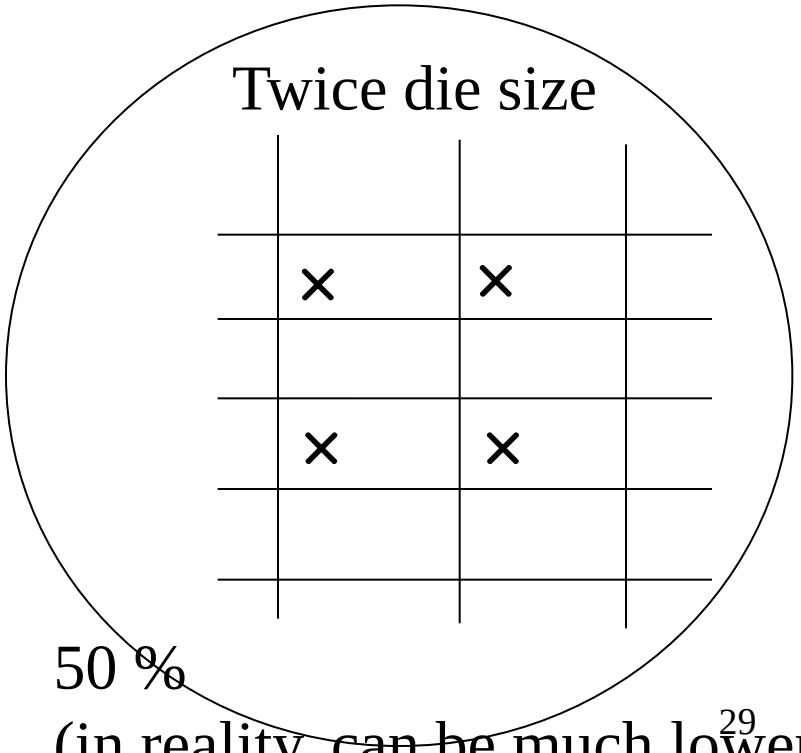
- Integrated circuit technology
  - Transistor density: 35%/year
  - Die size: 10-20%/year
  - Integration overall: 40-55%/year
- DRAM capacity: 25-40%/year (slowing)
- Flash capacity: 50-60%/year
  - 15-20X cheaper/bit than DRAM
- Magnetic disk technology: 40%/year
  - 15-25X cheaper/bit then Flash
  - 300-500X cheaper/bit than DRAM

Yield (수율): Proportion of Good Die

- Pushing semiconductor technology
  - Smaller transistor, larger die size
    - † Increase power consumption in processor



75 %



50 %
(in reality, can be much lower)

Integrated Circuit Cost (skip)

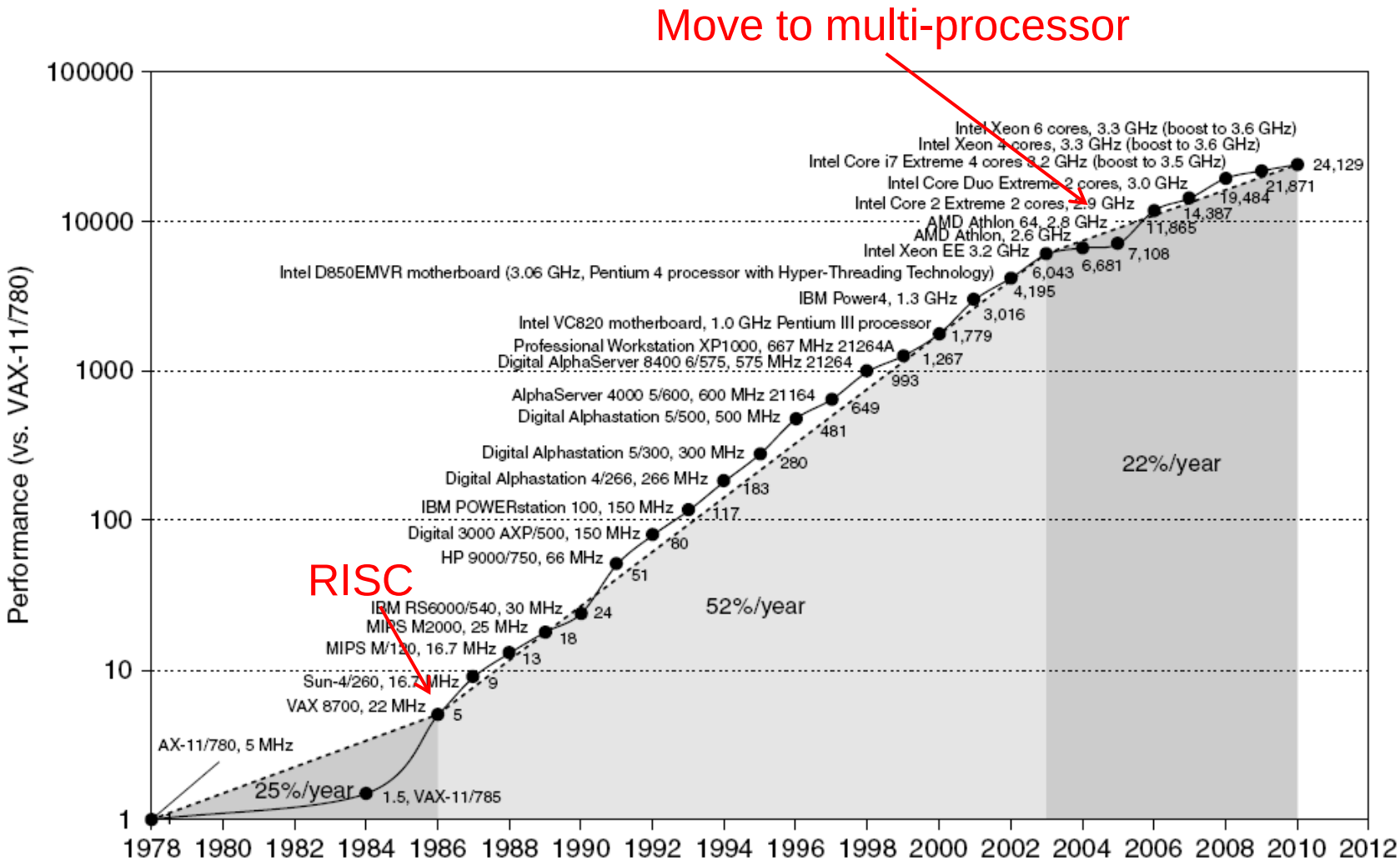
$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

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

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

- 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)

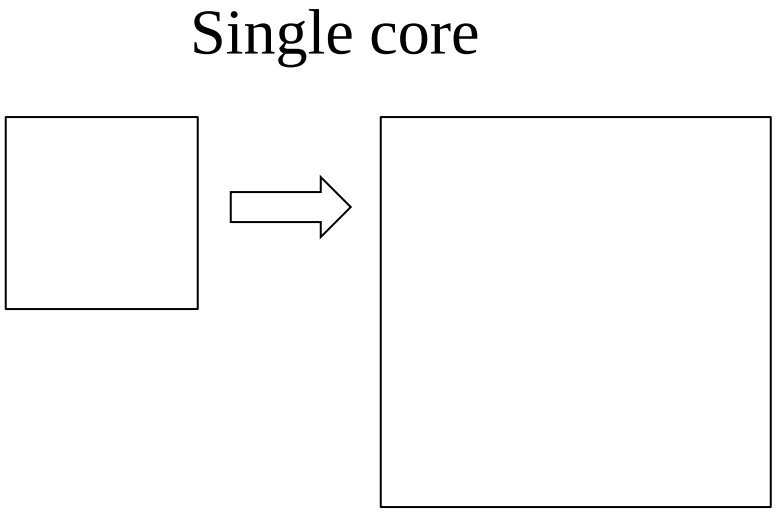


Game of Response Time

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

Diminishing Return

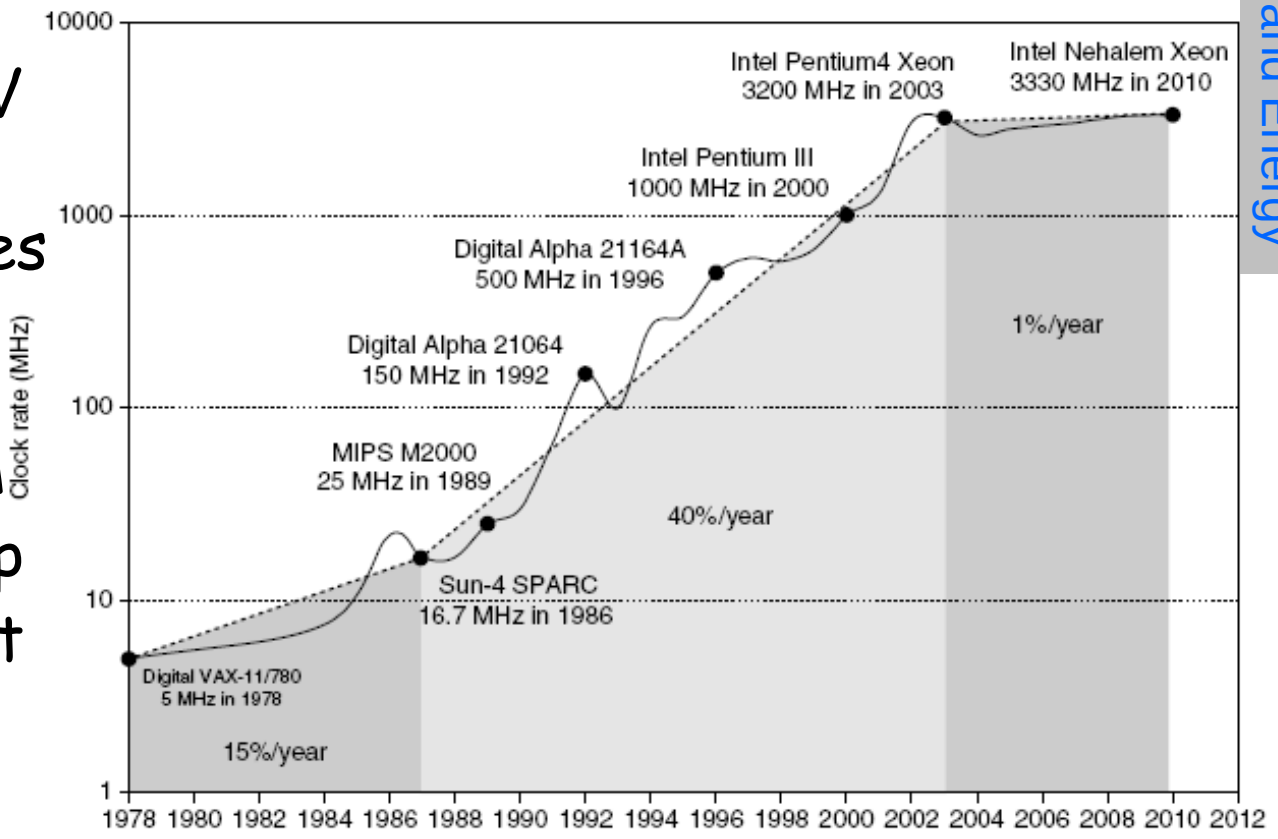
- Turn of 21C: diminishing return on investment



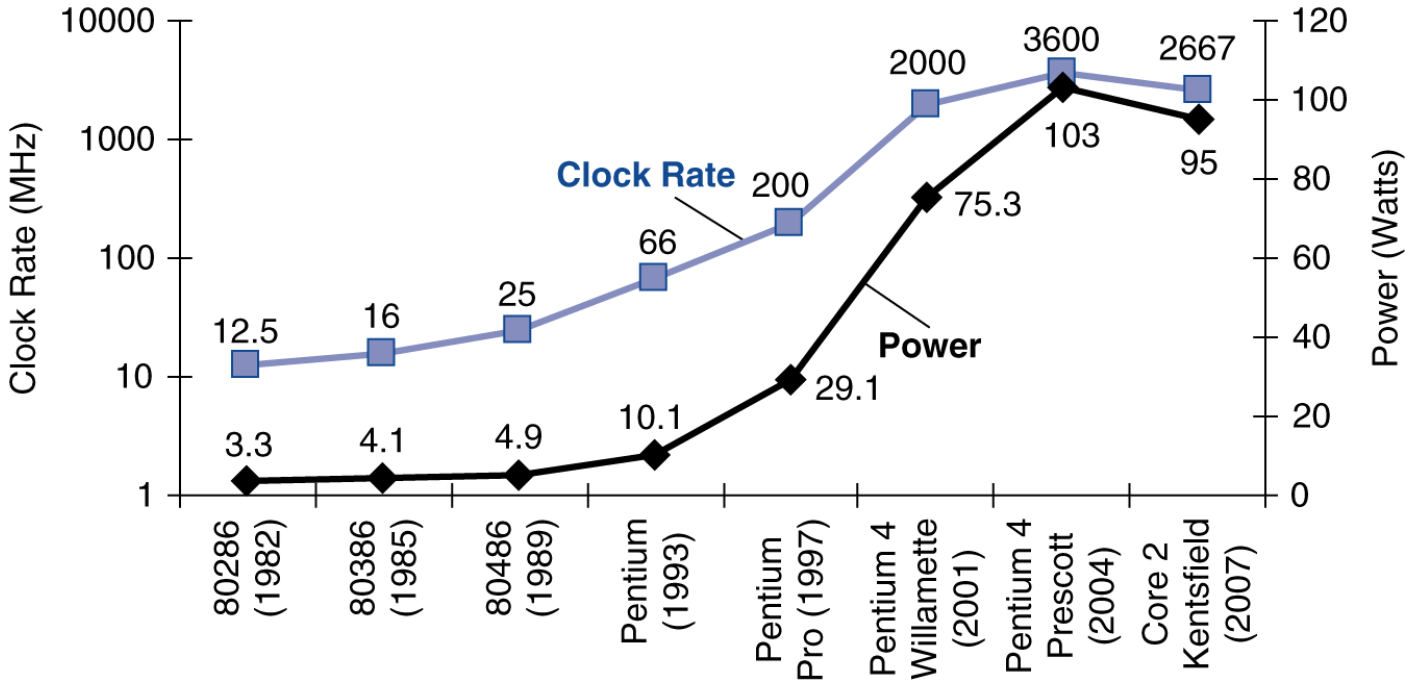
Die size	1 : 4
Performance	1 : 2
Power	1 : 4

Power Wall (참고자료)

- Intel 80386 consumed ~ 2 W
- 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 (참고자료)



- In CMOS IC technology

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

×30 | 5V → 1V | ×1000

Reducing Power (skip)

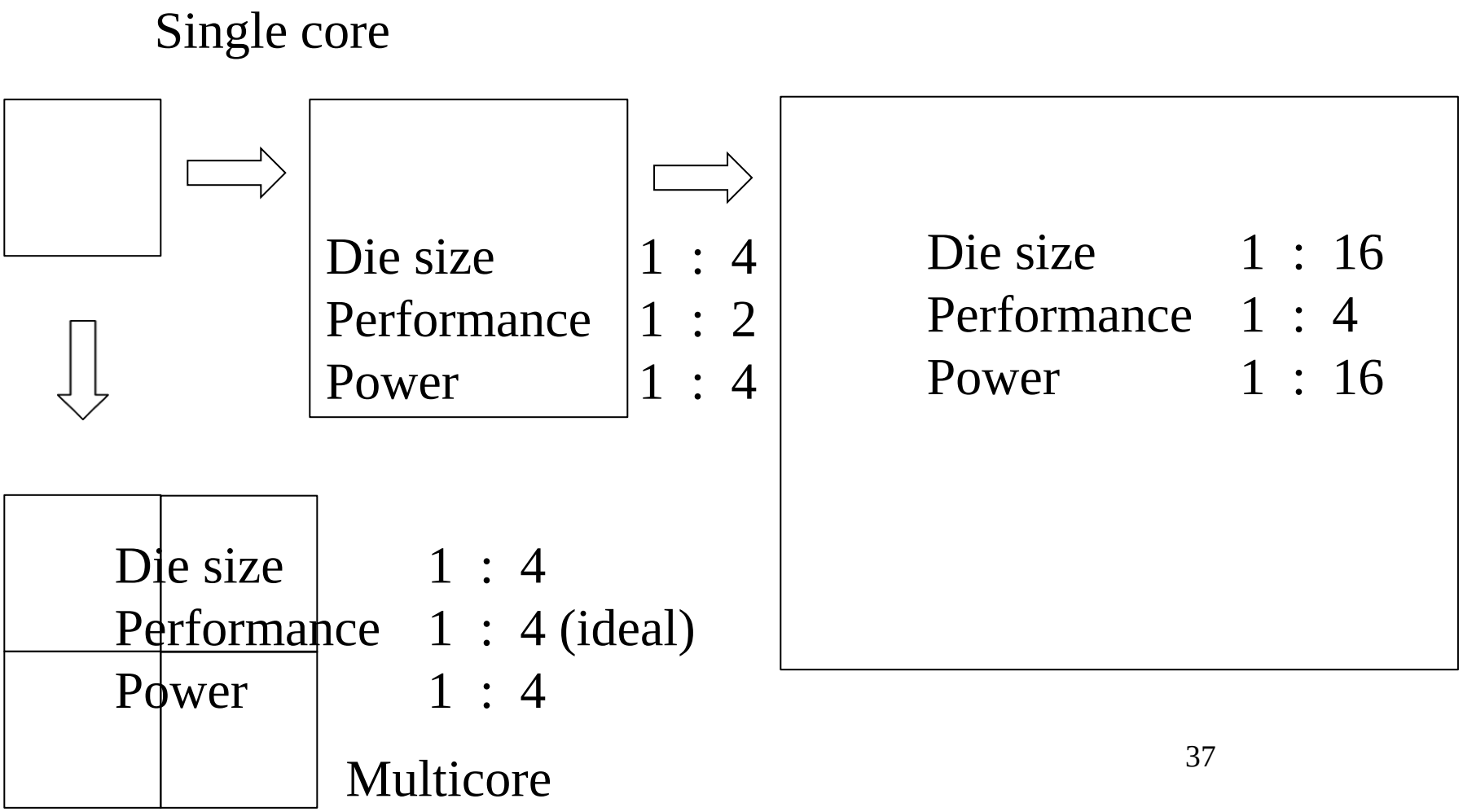
- Suppose a new CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

- The power wall
  - We can't reduce voltage further
  - We can't remove more heat
- How else can we improve performance?

Multicores

- Turn of 21C: diminishing returns again (single core)



(Shared-Memory) Multicores

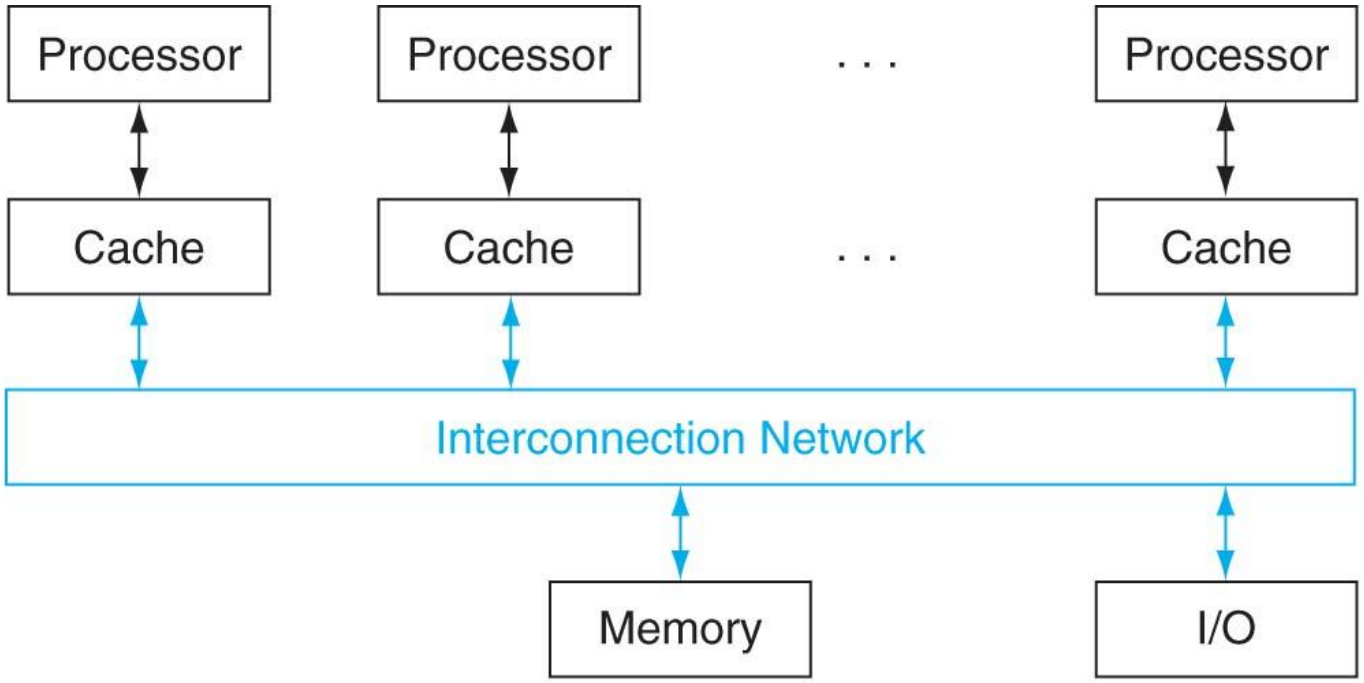
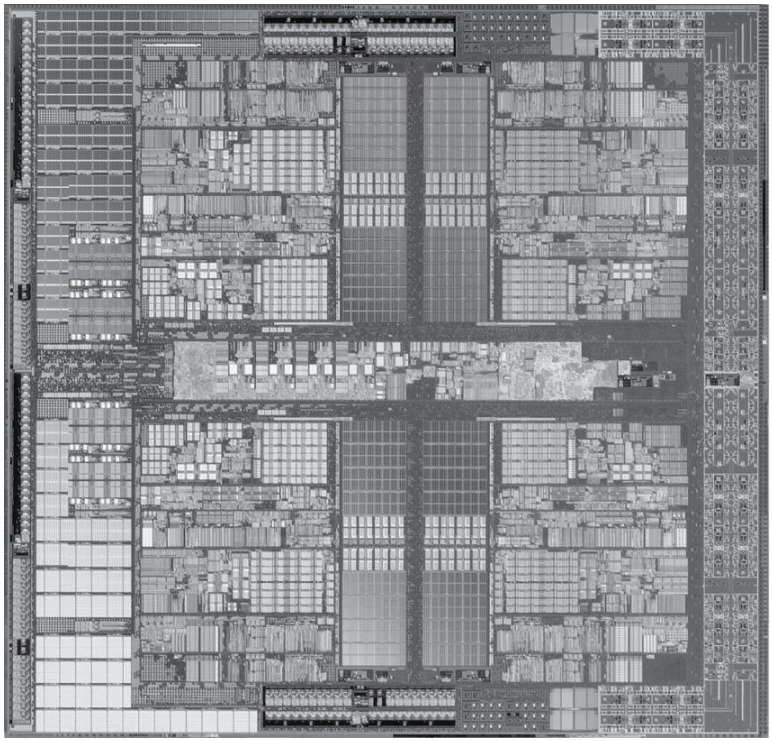
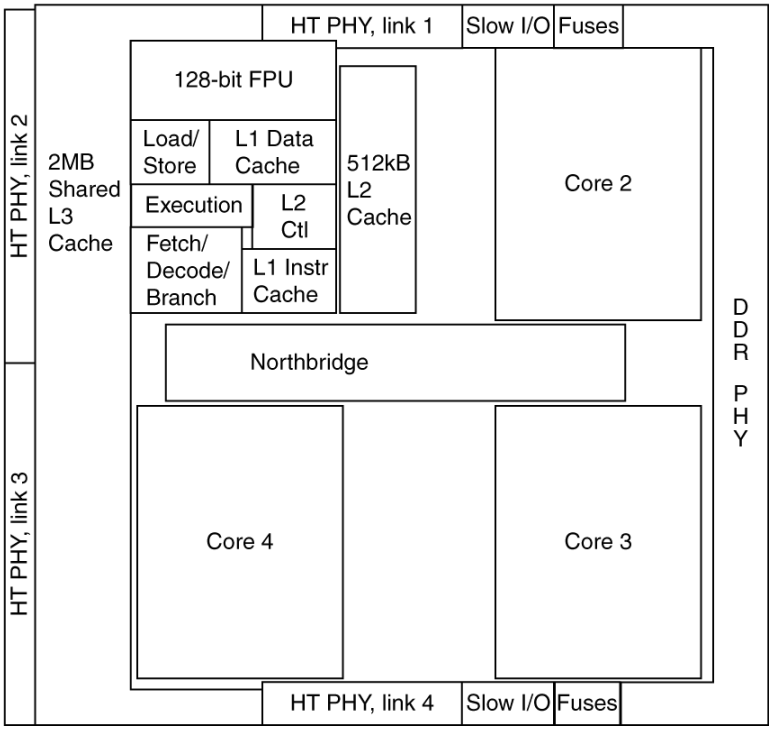


FIGURE 7.2 Classic organization of a shared memory multiprocessor. Copyright © 2009 Elsevier, Inc. All rights reserved.

Inside the Processor (반복)

- AMD Barcelona: 4 processor cores





Multicores

Product	AMD Opteron X4 (Barcelona)	Intel Nehalem	IBM Power 6	Sun Ultra SPARC T2 (Niagara 2)
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~2.5 GHz?	4.7 GHz	1.4 GHz
Microprocessor power	120 W	~100 W ?	~100 W ?	94 W

- Multicore microprocessors
  - More than one processor per chip
- As of 2006: all desktop/server companies ship multicores
  - Plan to double #cores per semiconductor generation
- Throughput rather than response time
  - Is parallel processing easy?

Parallel Revolution

- Performance of microprocessors
  - Improved 50%/year until 2002, then 20%/year
    - Limits in power, ILP, memory performance
- 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

Parallel Revolution

- Dream: Higher performance with multiple processors
  - However, explicitly rewriting programs to be parallel has been “third rail” of computer architecture
- Will programmers finally successfully switch to explicitly parallel programming?
  - Processor in your desktop likely to be multicore
- Need to understand multiprocessor systems to develop efficient parallel programs
  - Need to understand single processor system to write efficient sequential programs

Parallel Revolution

- Why is parallel programming difficult?
  - It is performance programming
  - Partitioning, Load balancing
  - Coordination (scheduling, synchronization)
  - Communication overhead

Parallelism in Textbook

- Chapter 6: parallel processors from client to cloud
- Demonstration (performance of a C program that multiply a matrix times a vector)
  - Chapter 3: subword parallelism via C intrinsics (increase performance by a factor of 3.8)
    - Data level parallelism
  - Chapter 4: loop unrolling & out-of-order execution (x2.3)
    - Instruction level parallelism
  - Chapter 5: cache blocking (x2.5)
  - Chapter 6 : parallel for loops in OpenMP (x14)
    - Thread level parallelism

Concluding Remarks

- Cost/performance is improving
  - Due to underlying technology development
- Hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure
- Power is a limiting factor
  - Use parallelism to improve performance