CPU Performance and Its Factors

EXAMPLE

- One program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer to build a computer B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

CPU Performance and Its Factors

One program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer to build a computer B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \ \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \ \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \ \text{cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \ \text{cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \ \text{cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \ \text{cycles}}{\text{second}} = \frac{4 \times 10^9 \ \text{cycles}}{\text{second}} = \boxed{4 \text{ GHz}}$$

To run the program in 6 seconds, B must have $\boxed{\text{twice the clock rate of A.}}$

Instruction Performance

- Since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program.
- Execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\boxed{\text{CPU clock cycles} = \text{Instructions for a program} \times \begin{array}{c}\text{Average clock cycles}\\ \text{per instruction}\end{array}}$$

Instruction Performance

- The term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI.

- Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program.

- CPI provides one way of comparing two different implementations of the same instruction set architecture, since the number of instructions executed for a program will be the same.

Instruction Performance

EXAMPLE

- Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

Instruction Performance

We know that each computer executes the same number of instructions for the program; let's call this number I. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$

$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time}\\ &= I \times 2.0 \times 250\text{ ps} = 500 \times I\text{ ps}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500\text{ ps} = 600 \times I\text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I\text{ ps}}{500 \times I\text{ ps}} = \boxed{1.2}$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

The Classic CPU Performance Equation

- Basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time

$$\boxed{\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}}$$

or, since the clock rate is the inverse of clock cycle time:

$$\boxed{\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}}$$

The Classic CPU Performance Equation

EXAMPLE

Comparing Code Segments

A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

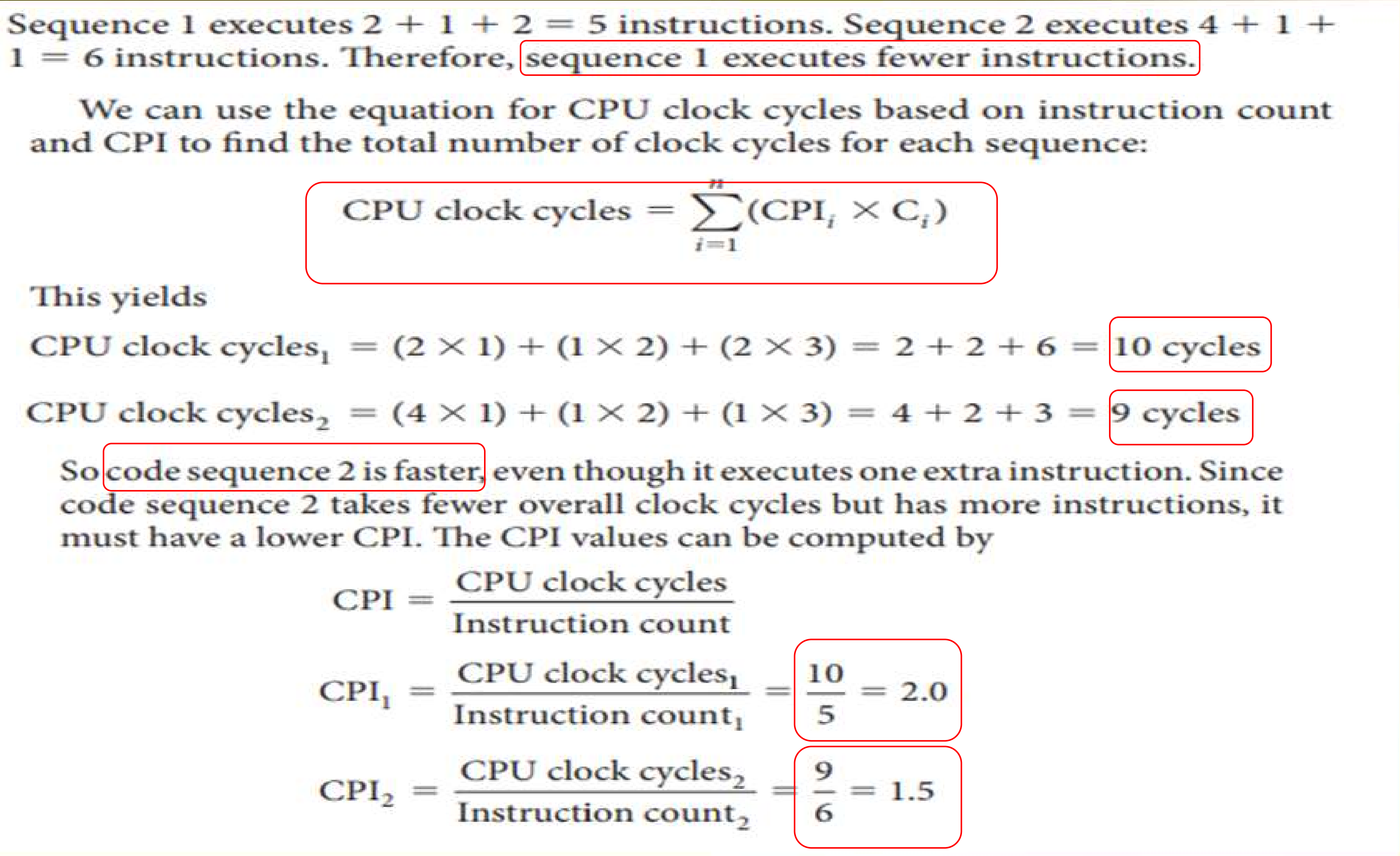
	CPI for each instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

	Instruction counts for each instruction class		
Code sequence	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

The Classic CPU Performance Equation



Sequence 1 executes 2 + 1 + 2 = 5 instructions. Sequence 2 executes 4 + 1 + 1 = 6 instructions. Therefore, $\boxed{\text{sequence 1 executes fewer instructions.}}$

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\boxed{\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)}$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = \boxed{10 \text{ cycles}}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = \boxed{9 \text{ cycles}}$$

So $\boxed{\text{code sequence 2 is faster,}}$ even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \boxed{\frac{10}{5} = 2.0}$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \boxed{\frac{9}{6} = 1.5}$$

The Classic CPU Performance Equation

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

FIGURE 7 The basic components of performance and how each is measured.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

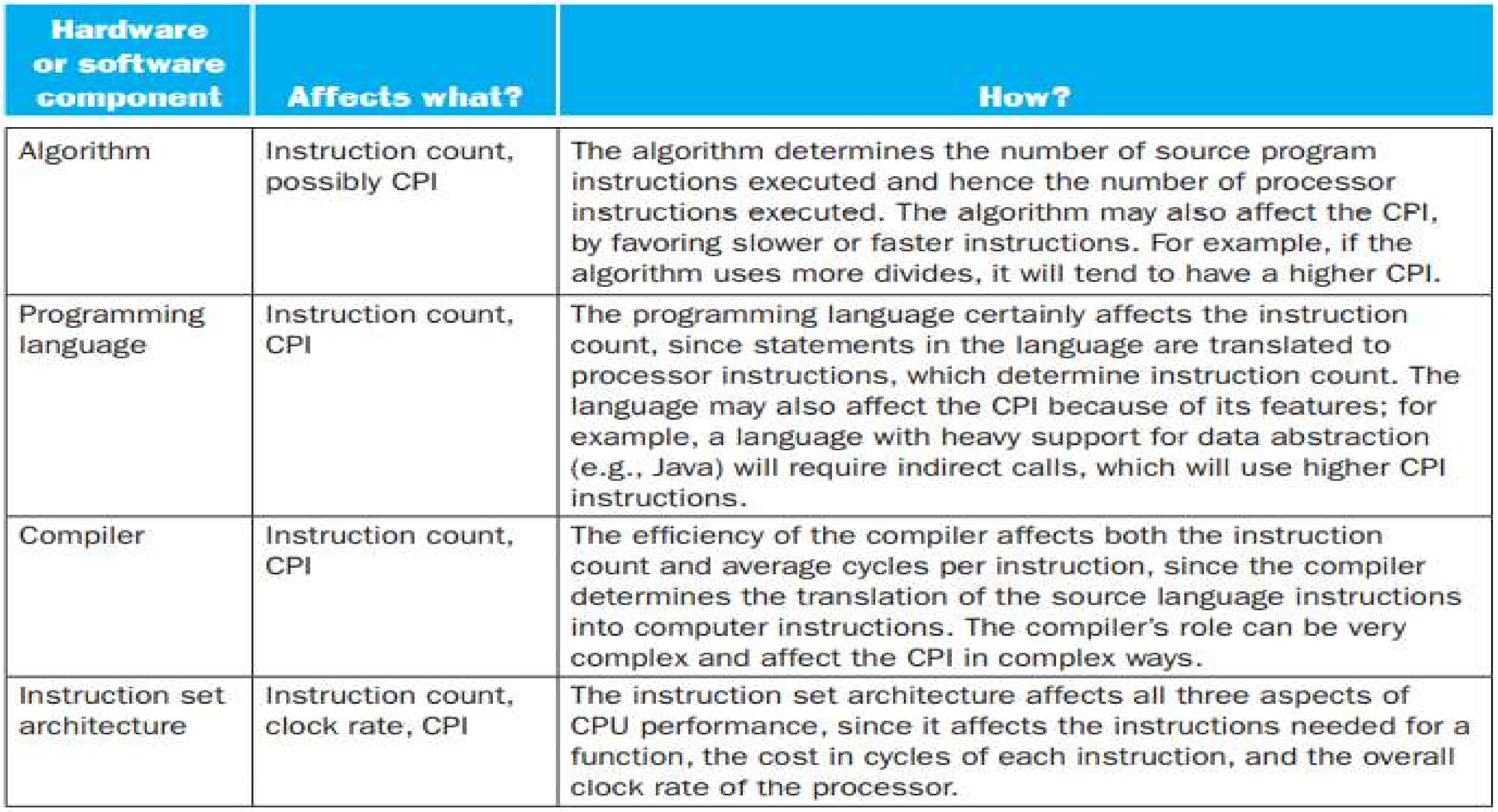
$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

The Classic CPU Performance Equation

- We can measure the CPU execution time by running the program, and the clock cycle time is usually published as part of the documentation for a computer.
- We can measure the instruction count by using software tools that profile the execution or by using a simulator of the architecture.
- Alternatively, we can use hardware counters, which are included in most processors, to record a variety of measurements, including the number of instructions executed, the average CPI, the sources of performance loss.
- When comparing two computers, you must look at all three components, which combine to form execution time. If some of the factors are identical, performance can be determined by comparing all the nonidentical factors.
- Since CPI varies by instruction mix, both instruction count and CPI must be compared, even if clock rates are identical.

Understanding Program Performance

- The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.



Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more divides, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

MODULE 3

Module III (10 Hours)

Computer abstractions and technology - Introduction, Computer architecture -8 Design features, Application program - layers of abstraction, Five key components of a computer, Technologies for building processors and memory, Performance, Instruction set principles – Introduction, Classifying instruction set architectures, Memory addressing, Encoding an instruction set.

J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition. (Appendix A). [4th Edition(Appendix B)]
J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.

Instructions: Language of the Computer

Introduction

- To command a computer's hardware, we must speak its language.
- The words of a computer's language are called instructions, and its vocabulary is called an instruction set.
- The chosen instruction set comes from MIPS Technologies, and is an elegant example of the instruction sets designed since the 1980s.

Instructions: Language of the Computer

Introduction

- Three other popular instruction sets.
  1. ARMv7 is similar to MIPS. More than 9 billion chips with ARM processors were manufactured in 2011, making it the most popular instruction set in the world.
  2. The second example is the Intel x86, which powers both the PC and the cloud of the PostPC Era.
  3. The third example is ARMv8, which extends the address size of the ARMv7 from 32 bits to 64 bits. Ironically, this 2013 instruction set is closer to MIPS than it is to ARMv7 [ARM- Advanced RISC Machine]

Computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy.

Instructions: Language of the Computer

Introduction

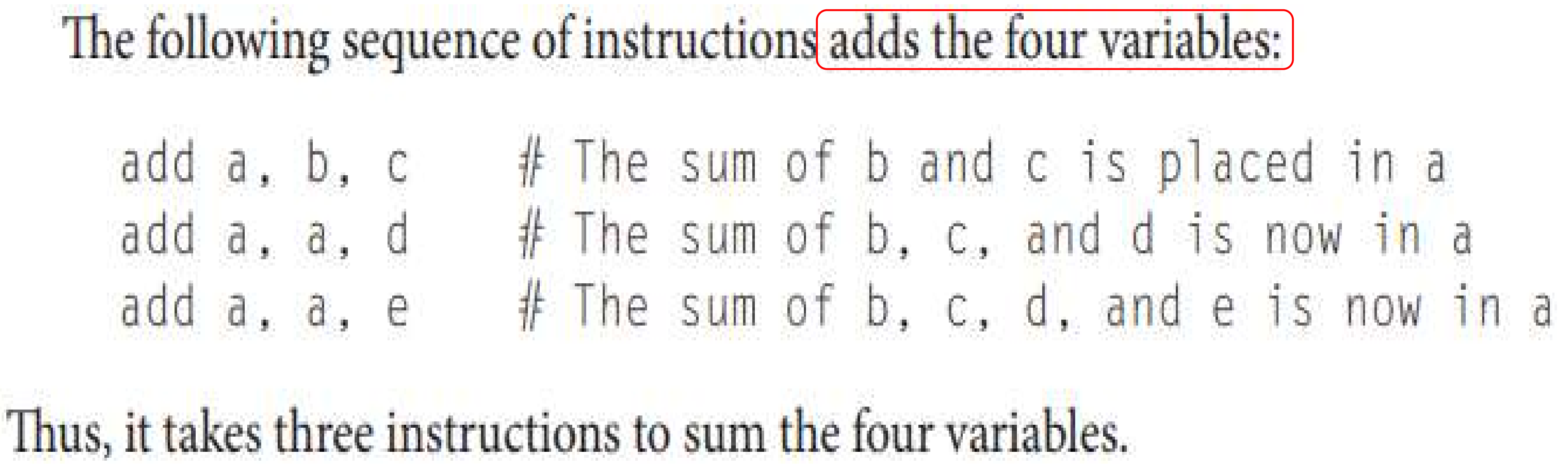
- Instruction set: The vocabulary of commands understood by a given architecture.
- Stored-program concept: The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

Operations of the Computer Hardware

- The MIPS assembly language notation

  add a, b, c

  instructs a computer to add the two variables b and c and to put their sum in a.



The following sequence of instructions adds the four variables:

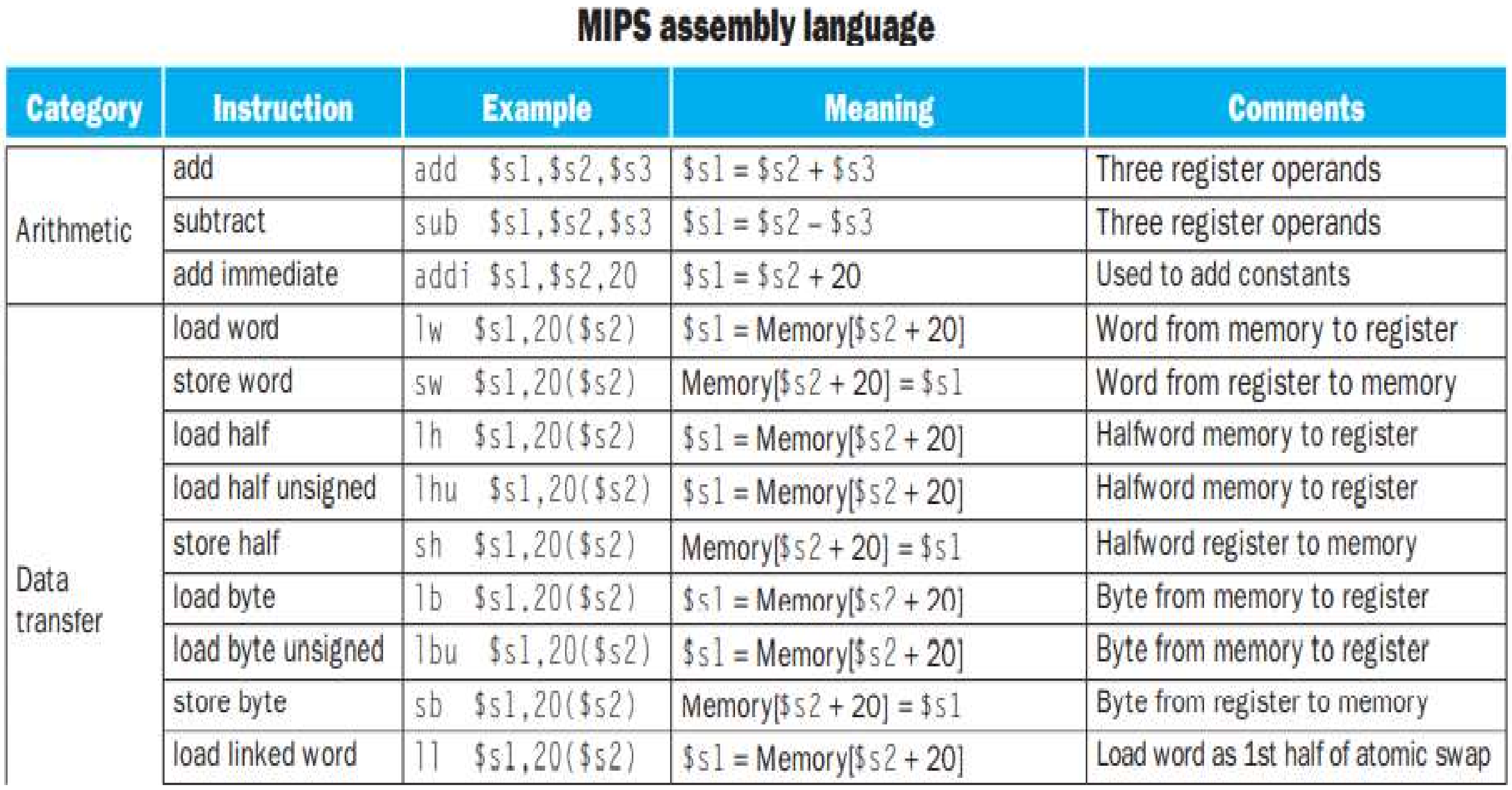
```
add a, b, c    # The sum of b and c is placed in a
add a, a, d    # The sum of b, c, and d is now in a
add a, a, e    # The sum of b, c, d, and e is now in a
```

Thus, it takes three instructions to sum the four variables.

Adding b, c, d, e

a = b + c + d + e

Operations of the Computer Hardware



MIPS assembly language

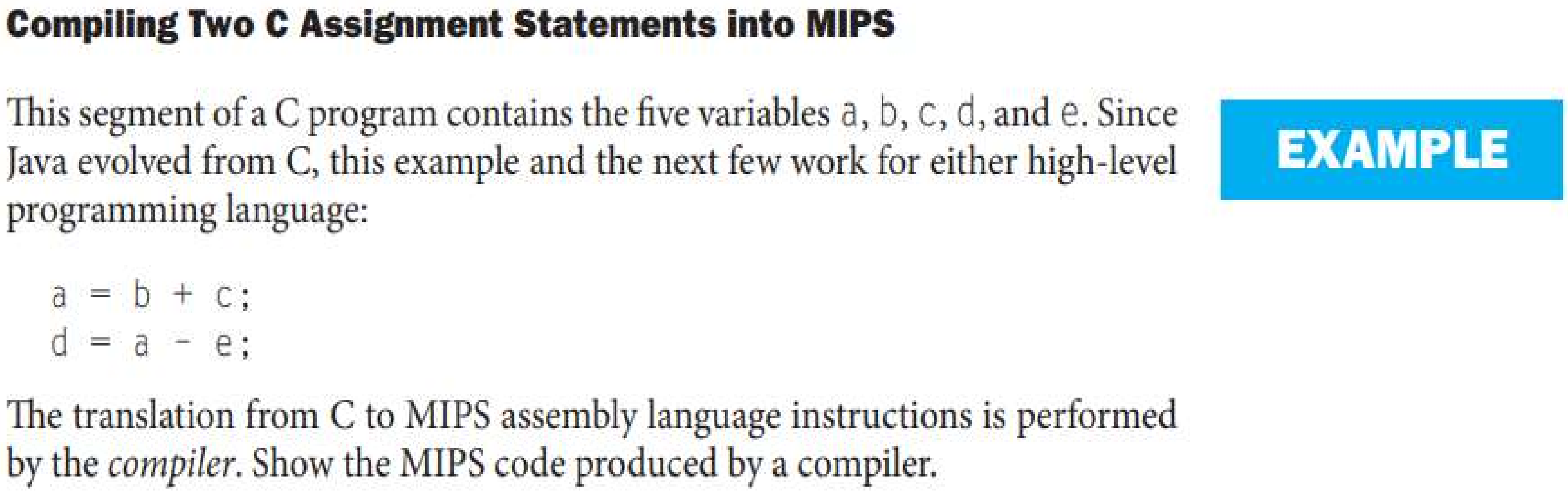
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1` = `$s2` + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap

Operations of the Computer Hardware

- The natural number of operands for an operation like addition is three: the two numbers being added together and a place to put the sum.
- Requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number.
- This situation illustrates the first of three underlying principles of hardware design:

    Design Principle 1: Simplicity favors regularity.

Operations of the Computer Hardware



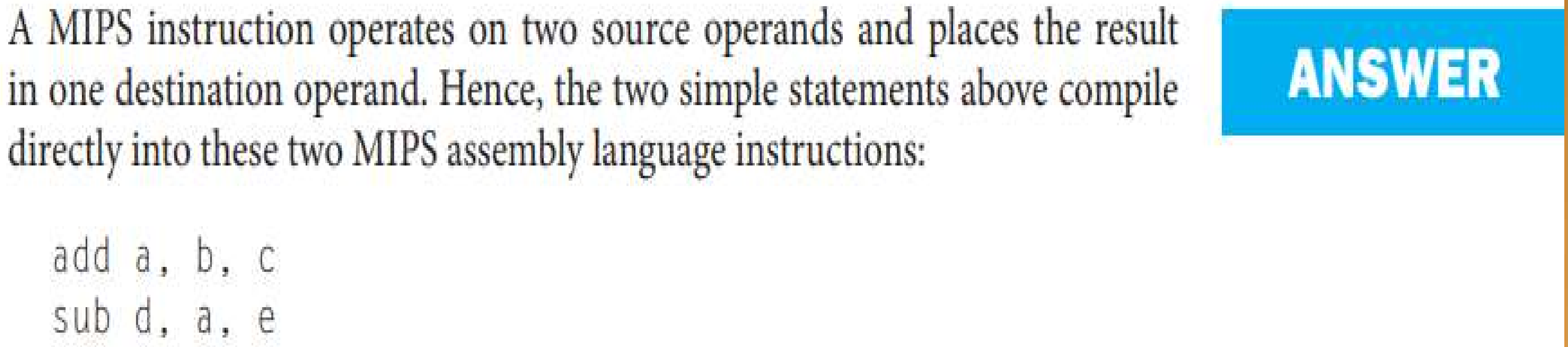
Compiling Two C Assignment Statements into MIPS

EXAMPLE

This segment of a C program contains the five variables a, b, c, d, and e. Since Java evolved from C, this example and the next few work for either high-level programming language:

```
a = b + c;
d = a - e;
```

The translation from C to MIPS assembly language instructions is performed by the *compiler*. Show the MIPS code produced by a compiler.



ANSWER

A MIPS instruction operates on two source operands and places the result in one destination operand. Hence, the two simple statements above compile directly into these two MIPS assembly language instructions:

```
add a, b, c
sub d, a, e
```

Operations of the Computer Hardware

Compiling a Complex C Assignment into MIPS

EXAMPLE

A somewhat complex statement contains the five variables f, g, h, i, and j:

```
f = (g + h) - (i + j);
```

What might a C compiler produce?

ANSWER

The compiler must break this statement into several assembly instructions, since only one operation is performed per MIPS instruction. The first MIPS instruction calculates the sum of g and h. We must place the result somewhere, so the compiler creates a temporary variable, called t0:

```
add t0,g,h # temporary variable t0 contains g + h
```

Although the next operation is subtract, we need to calculate the sum of i and j before we can subtract. Thus, the second instruction places the sum of i and j in another temporary variable created by the compiler, called t1:

```
add t1,i,j # temporary variable t1 contains i + j
```

Finally, the subtract instruction subtracts the second sum from the first and places the difference in the variable f, completing the compiled code:

```
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```

Operands of the Computer Hardware

- The operands of arithmetic instructions are restricted;
- They must be from a limited number of special locations built directly in hardware called registers.
- Registers are primitives used in hardware design that are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction.
- The size of a register in the MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name word in the MIPS architecture.
- The reason for the limit of 32 registers may be found as one of the design principles of hardware technology:

Design Principle 2: Smaller is faster

Instruction Set Principles

- Desktop computing emphasizes performance of programs with integer and floating-point data types, with little regard for program size or processor power consumption. For example, code size has never been reported in the five generations of SPEC benchmarks (Standard Performance Evaluation Corporation)
- Servers today are used primarily for database, file server, and Web applications, plus some time-sharing applications for many users. Hence, floating-point performance is much less important for performance than integers and character strings, yet virtually every server processor still includes floating-point instructions.
- Embedded applications value cost and power, so code size is important because less memory is both cheaper and lower power, and some classes of instructions (such as floating point) may be optional to reduce chip costs.

Instruction Set Principles

- Instruction sets for all three applications are very similar.
- In fact, the MIPS architecture has been used successfully in desktops, servers, and embedded applications.
- The commercial importance of binary compatibility with PC software combined with the abundance of transistors provided by Moore's Law led Intel to use a RISC (Reduced Instruction Set Computer) instruction set internally while supporting an 80x86 instruction set externally.
- Recent 80x86 microprocessors, such as the Pentium 4, use hardware to translate from 80x86 instructions to RISC-like instructions and then execute the translated operations inside the chip.

Classifying Instruction Set Architectures

- The type of internal storage in a processor is the most basic differentiation.
- The major choices are a stack, an accumulator, or a set of registers.
- Operands may be named explicitly or implicitly.
- The operands in a stack architecture are implicitly on the top of the stack, and in an accumulator architecture one operand is implicitly the accumulator.
- The general-purpose register architectures have only explicit operands—either registers or memory locations.

Classifying Instruction Set Architectures

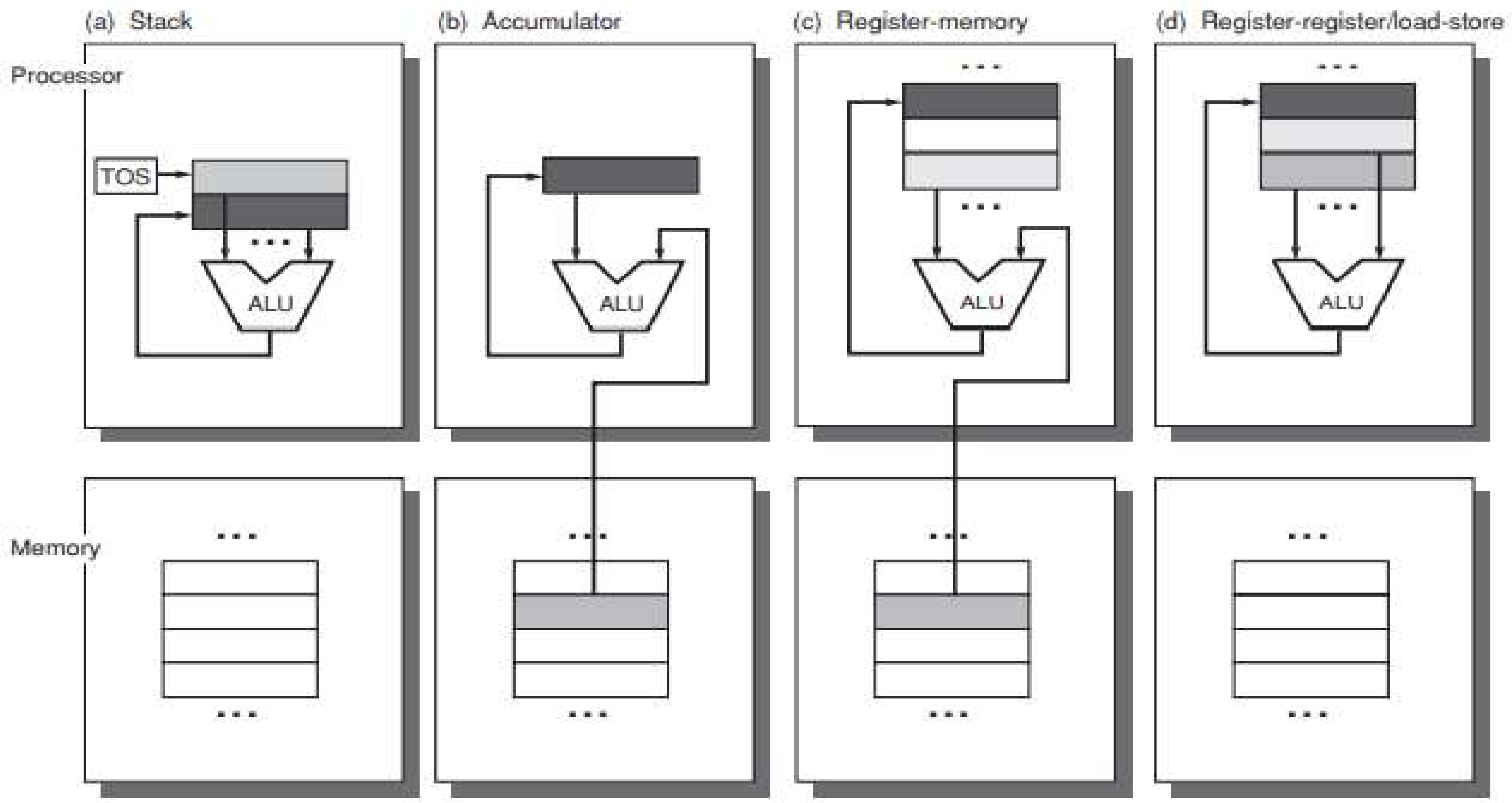
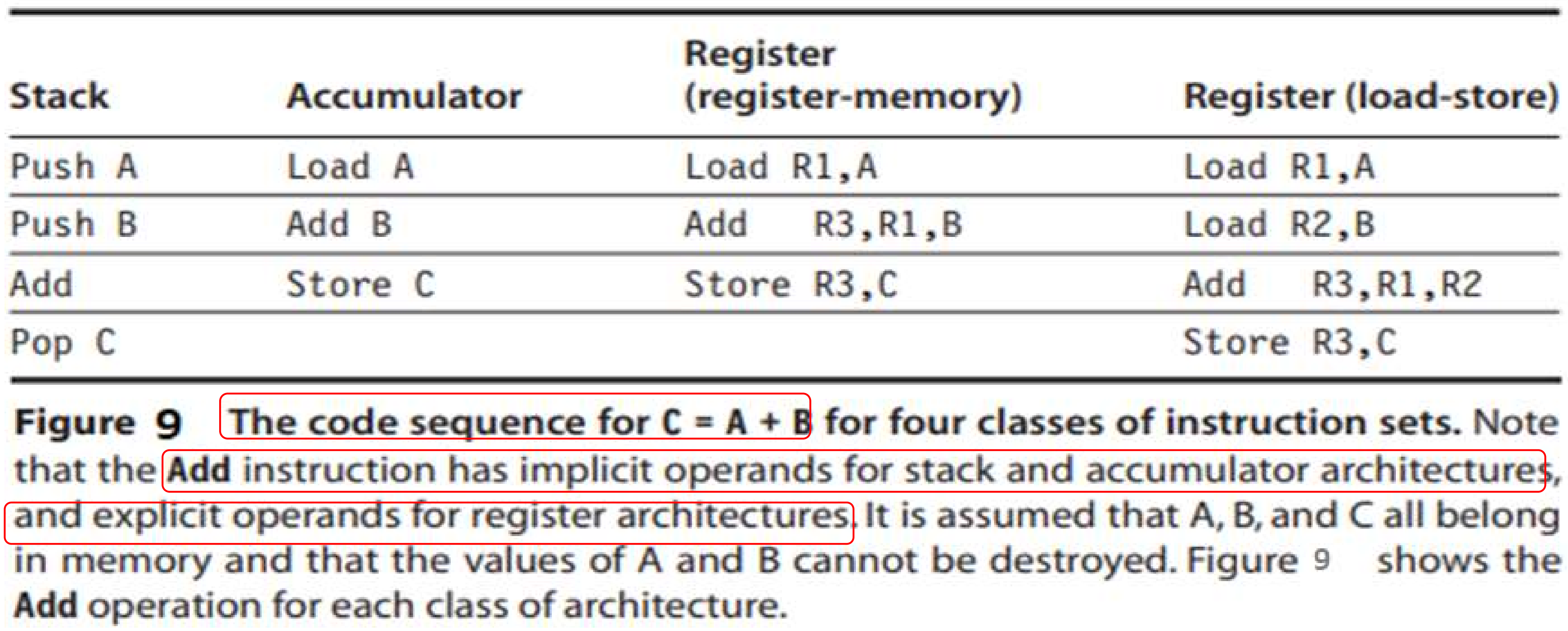


Figure 8 Operand locations for four instruction set architecture classes. The arrows indicate whether the operand is an input or the result of the ALU operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (a), a Top Of Stack register (TOS), points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (b), the Accumulator is both an implicit input operand and a result. In (c), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (d) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (a) and load or store for (d).

[J. Hennessy and D. Patterson, “Computer Architecture, A quantitative approach”, 5th Edition.]

Classifying Instruction Set Architectures



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 9 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 9 shows the **Add** operation for each class of architecture.

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]

Figure 9 shows how the code sequence C = A + B would typically appear in these three classes of instruction sets. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

Classifying Instruction Set Architectures

- There are really two classes of register computers.
    - One class can access memory as part of any instruction, called register-memory architecture and
    - The other can access memory only with load and store instructions, called load-store architecture.
- A third class, not found in computers today, keeps all operands in memory and is called a memory-memory architecture.
- Some instruction set architectures have more registers than a single accumulator, but place restrictions on uses of these special registers. Such an architecture is sometimes called an extended accumulator or special-purpose register computer.

Classifying Instruction Set Architectures

- Although most early computers used stack or accumulator-style architectures, virtually every architecture designed after 1980 uses a load-store register architecture.
- The major reasons for the emergence of general-purpose register (GPR) computers are twofold.
- First, registers—like other forms of storage internal to the processor—are faster than memory.
- Second, registers are more efficient for a compiler to use than other forms of internal storage.
- For example, on a register computer the expression (A*B) – (B*C) – (A*D) may be evaluated by doing the multiplications in any order, which may be more efficient because of the location of the operands or because of pipelining concerns. On a stack computer the hardware must evaluate the expression in only one order, since operands are hidden on the stack, and it may have to load an operand multiple times.

Classifying Instruction Set Architectures

- Registers can be used to hold variables.
- When variables are allocated to registers, the memory traffic reduces, the program speeds up (since registers are faster than memory),
- And the code density improves (since a register can be named with fewer bits than can a memory location).

Classifying Instruction Set Architectures

- Compiler writers would prefer that all registers be equivalent and unreserved.
- Older computers compromise this desire by dedicating registers to special uses, effectively decreasing the number of general purpose registers.
- If the number of truly general-purpose registers is too small, trying to allocate variables to registers will not be profitable.
- Instead, the compiler will reserve all the uncommitted registers for use in expression evaluation.

Classifying Instruction Set Architectures

How many registers are sufficient?

- The answer depends on the effectiveness of the compiler.
- Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables.
- Modern compiler technology and its ability to effectively use larger number of registers has led to an increase in register counts in more recent architectures.

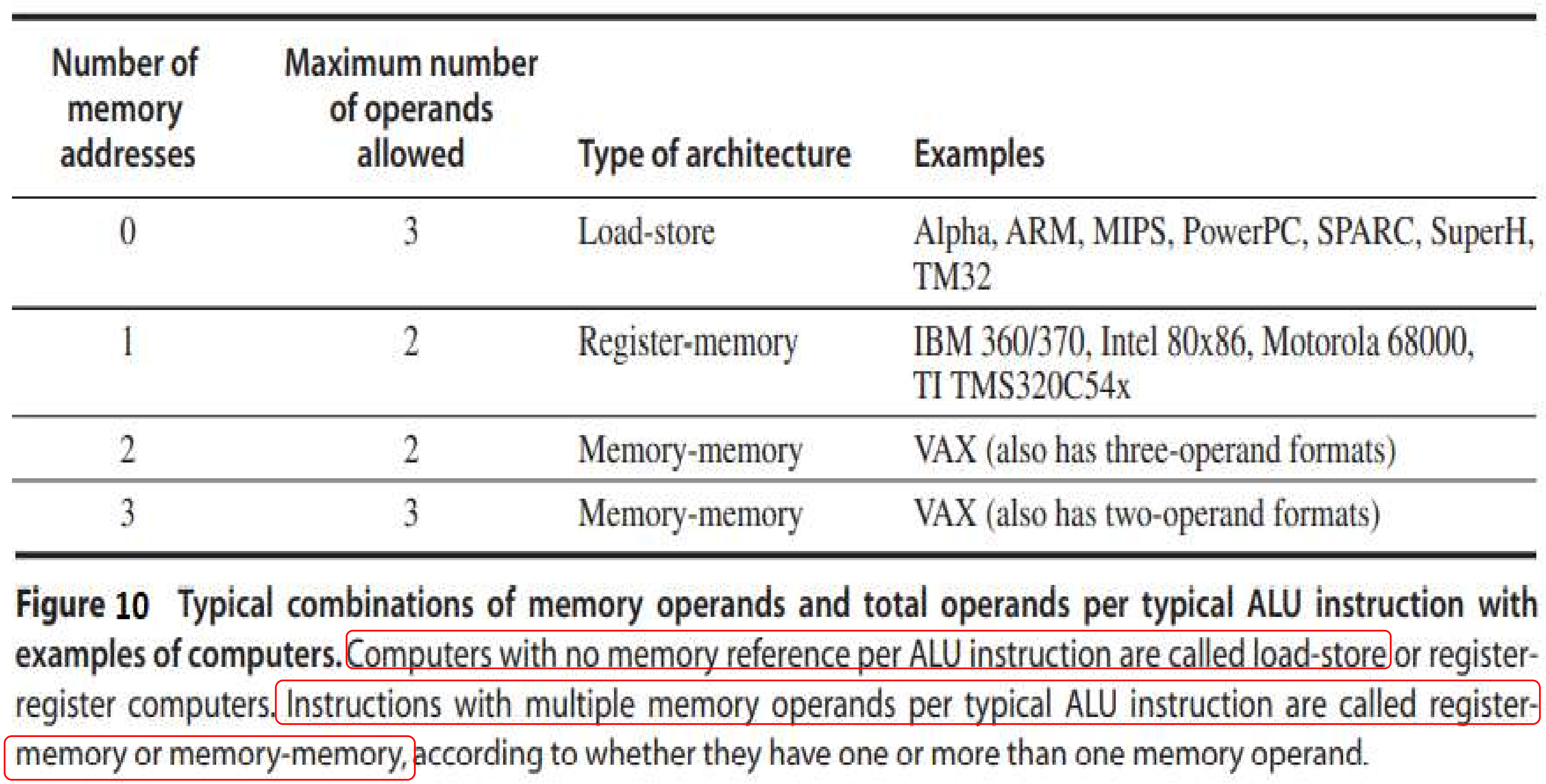
Classifying Instruction Set Architectures

- Two major instruction set characteristics divide GPR(General Purpose Registers) architectures.
- Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction).
- The first concerns whether an ALU instruction has two or three operands.
- In the three-operand format, the instruction contains one result operand and two source operands.
- In the two-operand format, one of the operands is both a source and a result for the operation.

Classifying Instruction Set Architectures

- The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions.
- The number of memory operands supported by a typical ALU instruction may vary from none to three.
- Figure 10 shows combinations of these two attributes with examples of computers.
- Although there are different possible combinations, three serve to classify nearly all existing computers.
    - Load-store (also called register-register)
    - Register-memory
    - Memory-memory

Classifying Instruction Set Architectures



Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
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 10 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.

[J. Hennessy and D. Patterson, “Computer Architecture, A quantitative approach”, 5th Edition.]

Back

Classifying Instruction Set Architectures

- Figure 11 shows the advantages and disadvantages of each of these alternatives.
- These advantages and disadvantages are not absolutes: They are qualitative and their actual impact depends on the compiler and implementation strategy.
- A GPR computer with memory-memory operations could easily be ignored by the compiler and used as a load-store computer.
- One of the most pervasive architectural impacts is on instruction encoding and the number of instructions needed to perform a task.

Classifying Instruction Set 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	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 11 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (*m*, *n*) means *m* memory operands and *n* total operands. In general, computers with fewer alternatives simplify the compiler's task since there are fewer decisions for the compiler to make
Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size since you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes 3 extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

Back

[J. Hennessy and D. Patterson, "Computer Architecture, A quantitative approach", 5th Edition.]