This book is dedicated to Lucy Ann, Douglas James, and Sara Sue.
# Table of Contents

## Preface

## Level 7 Application

### 1. Computer Systems

1.1 **Levels of Abstraction**  
- Abstraction in Art  
- Abstraction in Documents  
- Abstraction in Organizations  
- Abstraction in Machines  
- Abstraction in Computer Systems

1.2 **Hardware**  
- Central Processing Unit  
- Main Memory  
- Disk

1.3 **Software**  
- Operating Systems  
- Software Analysis and Design

1.4 **Digital Information**  
- Quantifying Space  
- Quantifying Time  
- Quick Response Codes  
- Images
3.5 Floating-Point Representation 153
- Binary Fractions 154
- Excess Representations 156
- The Hidden Bit 158
- Special Values 159
- The IEEE 754 Floating-Point Standard 165

3.6 Models 168

Chapter Summary 170

Exercises 172
Problems 179

4. Computer Architecture 183

4.1 Hardware 184
- Central Processing Unit (CPU) 185
- Main Memory 186
- Input/Output Devices 188
- Data and Control 189
- Instruction Format 189

4.2 Direct Addressing 193
- The Stop Instruction 194
- The Load Word Instruction 194
- The Store Word Instruction 196
- The Add Instruction 197
- The Subtract Instruction 198
- The And and Or Instructions 199
- The Invert and Negate Instructions 200
- The Load Byte and Store Byte Instructions 202
- The Input and Output Devices 203
- Big Endian Versus Little Endian 204

4.3 von Neumann Machines 206
- The von Neumann Execution Cycle 206
- A Character Output Program 208
- von Neumann Bugs 214
- A Character Input Program 214
- Converting Decimal to ASCII 216
- A Self-Modifying Program 217
4.4 Programming at Level ISA3 221

Read-Only Memory 221
The Pep/9 Operating System 222
Using the Pep/9 System 225

Chapter Summary 226
Exercises 227
Problems 230

Level 5 Assembly 231

5. Assembly Language 233

5.1 Assemblers 234
Instruction Mnemonics 234
Pseudo-Operations 238
The .ASCII and .END Pseudo-ops 238
Assemblers 240
The .BLOCK Pseudo-op 241
The .WORD and .BYTE Pseudo-ops 242
Using the Pep/9 Assembler 244
Cross Assemblers 246

5.2 Immediate Addressing and the Trap Instructions 247
Immediate Addressing 247
The DECI, DECO, and BR Instructions 248
The STRO Instruction 251
Interpreting Bit Patterns: The HEXO Instruction 252
Disassemblers 255

5.3 Symbols 257
A Program with Symbols 258
A von Neumann Illustration 260

5.4 Translating from Level HOL6 262
The Printf() Function 263
Variables and Types 266
Global Variables and Assignment Statements 267
Type Compatibility 271
Pep/9 Symbol Tracer 273
6. Compiling to the Assembly Level 287

6.1 Stack Addressing and Local Variables 288

- Stack-Relative Addressing 289
- Accessing the Run-Time Stack 290
- Local Variables 292

6.2 Branching Instructions and Flow of Control 294

- Translating the If Statement 295
- Optimizing Compilers 297
- Translating the If/Else Statement 298
- Translating the While Loop 300
- Translating the Do Loop 302
- Translating the For Loop 303
- Spaghetti Code 305
- Flow of Control in Early Languages 307
- The Structured Programming Theorem 308
- The Goto Controversy 309

6.3 Function Calls and Parameters 310

- Translating a Function Call 310
- Translating Call-by-Value Parameters with Global Variables 313
- Translating Call-by-Value Parameters with Local Variables 317
- Translating Non-void Function Calls 320
- Translating Call-by-Reference Parameters with Global Variables 323
- Translating Call-by-Reference Parameters with Local Variables 329
- Translating Boolean Types 333
6.4 Indexed Addressing and Arrays 336
   Translating Global Arrays 337
   Translating Local Arrays 341
   Translating Arrays Passed as Parameters 344
   Translating the Switch Statement 350

6.5 Dynamic Memory Allocation 356
   Translating Global Pointers 357
   Translating Local Pointers 363
   Translating Structures 367
   Translating Linked Data Structures 372

Chapter Summary 377
Exercises 379
Problems 380

7. Language Translation Principles 391
7.1 Languages, Grammars, and Parsing 393
   Concatenation 393
   Languages 394
   Grammars 395
   A Grammar for C Identifiers 396
   A Grammar for Signed Integers 398
   A Context-Sensitive Grammar 399
   The Parsing Problem 400
   A Grammar for Expressions 401
   A C Subset Grammar 402
   Context Sensitivity of C 407

7.2 Finite-State Machines 408
   An FSM to Parse an Identifier 409
   Simplified FSMs 410
   Nondeterministic FSMs 410
   Machines with Empty Transitions 412
   Multiple Token Recognizers 415
   Grammars Versus FSMs 417

7.3 Implementing Finite-State Machines 418
   The Compilation Process 420
A Table-Lookup Parser 421
A Direct-Code Parser 423
An Input Buffer Class 426
A Multiple-Token Parser 428

7.4 Code Generation 435
A Language Translator 435
Parser Characteristics 455

Chapter Summary 456
Exercises 457
Problems 460

Level 4 Operating System 467

8. Process Management 469

8.1 Loaders 470
The Pep/9 Operating System 471
The Pep/9 Loader 472
Program Termination 474

8.2 Traps 475
The Trap Mechanism 476
The RETTR Instruction 477
The Trap Handlers 478
Trap Addressing Mode Assertion 480
Trap Operand Address Computation 483
The No-Operation Trap Handlers 487
The DECI Trap Handler 488
The DECO Trap Handler 495
The HEXO and STRO Trap Handlers and Operating System Vectors 498

8.3 Concurrent Processes 501
Asynchronous Interrupts 502
Processes in the Operating System 503
Multiprocessing 505
A Concurrent Processing Program 506
Critical Sections 508
A First Attempt at Mutual Exclusion 509
### 8.4 Deadlocks 516

- Resource Allocation Graphs 517
- Deadlock Policy 519

#### Chapter Summary 520

- Exercises 520
- Problems 527

### 9. Storage Management 531

#### 9.1 Memory Allocation 532

- Uniprogramming 532
- Fixed-Partition Multiprogramming 533
- Logical Addresses 534
- Variable-Partition Multiprogramming 537
- Paging 540

#### 9.2 Virtual Memory 543

- Large Program Behavior 543
- Virtual Memory 544
- Demand Paging 545
- Page Replacement 546
- Page-Replacement Algorithms 547

#### 9.3 File Management 549

- Disk Drives 550
- File Abstraction 551
- Allocation Techniques 552

#### 9.4 Error-Detecting and Error-Correcting Codes 555

- Error-Detecting Codes 555
- Code Requirements 556
- Single-Error-Correcting Codes 559

#### 9.5 RAID Storage Systems 562

- RAID Level 0: Nonredundant Striped 562
- RAID Level 1: Mirrored 563
- RAID Levels 01 and 10: Striped and Mirrored 564
Chapter Summary 570
Exercises 571

Level 1 Logic Gate 575

10. Combinational Circuits 577

10.1 Boolean Algebra and Logic Gates 579
   Combinational Circuits 579
   Truth Tables 580
   Boolean Algebra 580
   Boolean Algebra Theorems 582
   Proving Complements 583
   Logic Diagrams 585
   Alternate Representations 587

10.2 Combinational Analysis 588
   Boolean Expressions and Logic Diagrams 589
   Truth Tables and Boolean Expressions 591
   Two-Level Circuits 593
   The Ubiquitous NAND 595

10.3 Combinational Design 597
   Canonical Expressions 598
   Three-Variable Karnaugh Maps 599
   Four-Variable Karnaugh Maps 604
   Dual Karnaugh Maps 607
   Don't-Care Conditions 608

10.4 Combinational Devices 609
   Viewpoints 609
   Multiplexer 610
   Binary Decoder 611
   Demultiplexer 612
   Adder 613
   Adder/Subtractor 615
11. Sequential Circuits 637

11.1 Latches and Clocked Flip-Flops 638
- The SR Latch 639
- The Clocked SR Flip-Flop 641
- The Master–Slave SR Flip-Flop 643
- The Basic Flip-Flops 649
- The JK Flip-Flop 650
- The D Flip-Flop 653
- The T Flip-Flop 654
- Excitation Tables 655

11.2 Sequential Analysis and Design 656
- A Sequential Analysis Problem 657
- Preset and Clear 661
- Sequential Design 661
- A Sequential Design Problem 662

11.3 Computer Subsystems 666
- Registers 666
- Buses 667
- Memory Subsystems 669
- Address Decoding 674
- A Two-Port Register Bank 680

Chapter Summary 682
Exercises 683

Level 2 Microcode 689

12. Computer Organization 691

12.1 Constructing a Level-ISA3 Machine 692
- The CPU Data Section 692
- The von Neumann Cycle 696
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2 Performance</td>
<td>716</td>
</tr>
<tr>
<td>The Data Bus Width and Memory Alignment</td>
<td>717</td>
</tr>
<tr>
<td>Memory Alignment</td>
<td>722</td>
</tr>
<tr>
<td>The Definition of an $n$-Bit Computer</td>
<td>727</td>
</tr>
<tr>
<td>Cache Memories</td>
<td>729</td>
</tr>
<tr>
<td>The System Performance Equation</td>
<td>738</td>
</tr>
<tr>
<td>RISC Versus CISC</td>
<td>739</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3 The MIPS Machine</td>
<td>743</td>
</tr>
<tr>
<td>The Register Set</td>
<td>743</td>
</tr>
<tr>
<td>The Addressing Modes</td>
<td>744</td>
</tr>
<tr>
<td>The Instruction Set</td>
<td>749</td>
</tr>
<tr>
<td>MIPS Computer Organization</td>
<td>755</td>
</tr>
<tr>
<td>Pipelining</td>
<td>760</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.4 Conclusion</td>
<td>771</td>
</tr>
<tr>
<td>Simplifications in the Model</td>
<td>771</td>
</tr>
<tr>
<td>The Big Picture</td>
<td>772</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter Summary</td>
<td>774</td>
</tr>
<tr>
<td>Exercises</td>
<td>775</td>
</tr>
<tr>
<td>Problems</td>
<td>780</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix</td>
<td>783</td>
</tr>
<tr>
<td>Solutions to Selected Exercises</td>
<td>795</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>813</td>
</tr>
</tbody>
</table>
Preface

The fifth edition of Computer Systems offers a clear, detailed, step-by-step exposition of the central ideas in computer organization, assembly language, and computer architecture. The book is based in large part on a virtual computer, Pep/9, which is designed to teach the basic concepts of the classic von Neumann machine. The strength of this approach is that the central concepts of computer science are taught without getting entangled in the many irrelevant details that often accompany such courses. This approach also provides a foundation that encourages students to think about the underlying themes of computer science. Breadth is achieved by emphasizing computer science topics that are related to, but not usually included in, the treatment of hardware and its associated software.

Summary of Contents

Computers operate at several levels of abstraction; programming at a high level of abstraction is only part of the story. This book presents a unified concept of computer systems based on the level structure of Figure P.1.

The book is divided into seven parts, corresponding to the seven levels of Figure P.1:

1. Logic gate level
2. Microcode level
3. Instruction set architecture level
4. Operating system level
5. Assembly level
6. High-order language level
7. Application level

The text generally presents the levels top-down, from the highest to the lowest. Level ISA3 is discussed before Level Asmb5, and Level LG1 is discussed before Level Mc2 for pedagogical reasons. In these two instances, it is more
natural to revert temporarily to a bottom-up approach so that the building blocks of the lower level will be in hand for construction of the higher level.

**Level App7**  Level App7 is a single chapter on application programs. It presents the idea of levels of abstraction and binary information and establishes the framework for the remainder of the book. A few concepts of relational databases are presented as an example of a typical computer application.

**Level HOL6**  Level HOL6 consists of one chapter, which reviews the C programming language. The chapter assumes that the student has experience in some imperative language, such as Java or Python—not necessarily C. The instructor can readily translate the C examples to other common Level HOL6 languages if necessary.

This chapter emphasizes the C memory model, including global versus local variables, functions with parameters, and dynamically allocated variables. The topic of recursion is treated because it depends on the mechanism of memory allocation on the run-time stack. A fairly detailed explanation is given on the details of the memory allocation process for function calls, as this mechanism is revisited at a lower level of abstraction later in the book.

**Level ISA3**  Level ISA3 is the instruction set architecture level. Its two chapters describe Pep/9, a virtual computer designed to illustrate computer concepts. Pep/9 is a small complex instruction set computer (CISC); a von Neumann machine. The central processing unit (CPU) contains an accumulator, an index register, a program counter, a stack pointer, and an instruction register. It has eight addressing modes: immediate, direct, indirect, stack-relative, stack-relative deferred, indexed, stack-indexed, and stack-deferred indexed. The Pep/9 operating system, in simulated read-only memory (ROM), can load and execute programs in hexadecimal format from students’ text files. Students run short programs on the Pep/9 simulator and learn that executing a store instruction to ROM does not change the memory value.

Students learn the fundamentals of information representation and computer organization at the bit level. Because a central theme of this book is the relationship of the levels to one another, the Pep/9 chapters show the relationship between the ASCII representation (Level ISA3) and C variables of type `char` (Level HOL6). They also show the relationship between two’s complement representation (Level ISA3) and C variables of type `int` (Level HOL6).

**Level Asmb5**  Level Asmb5 is the assembly level. The text presents the concept of the assembler as a translator between two levels—assembly and machine. It introduces Level Asmb5 symbols and the symbol table.
The unified approach really comes into play here. Chapters 5 and 6 present the compiler as a translator from a high-order language to assembly language. Previously, students learned a specific Level HOL6 language, C, and a specific von Neumann machine, Pep/9. These chapters continue the theme of relationships between the levels by showing the correspondence between (a) assignment statements at Level HOL6 and load/store instructions at Level Asmb5, (b) loops and if statements at Level HOL6 and branching instructions at Level Asmb5, (c) arrays at Level HOL6 and indexed addressing at Level Asmb5, (d) procedure calls at Level HOL6 and the run-time stack at Level Asmb5, (e) function and procedure parameters at Level HOL6 and stack-relative addressing at Level Asmb5, (f) switch statements at Level HOL6 and jump tables at Level Asmb5, and (g) pointers at Level HOL6 and addresses at Level Asmb5.

The beauty of the unified approach is that the text can implement the examples from the C chapter at this lower level. For example, the run-time stack illustrated in the recursive examples of Chapter 2 corresponds directly to the hardware stack in Pep/9 main memory. Students gain an understanding of the compilation process by translating manually between the two levels.

This approach provides a natural setting for the discussion of central issues in computer science. For example, the book presents structured programming at Level HOL6 versus the possibility of unstructured programming at Level Asmb5. It discusses the goto controversy and the structured programming/efficiency tradeoff, giving concrete examples from languages at the two levels.

Chapter 7, “Language Translation Principles,” introduces students to computer science theory. Now that students know intuitively how to translate from a high-level language to assembly language, we pose the fundamental question underlying all of computing: What can be automated? The theory naturally fits in here because students now know what a compiler (an automated translator) must do. They learn about parsing and finite-state machines—deterministic and nondeterministic—in the context of recognizing C and Pep/9 assembly language tokens. This chapter includes an automatic translator between two small languages, which illustrates lexical analysis, parsing, and code generation. The lexical analyzer is an implementation of a finite-state machine. What could be a more natural setting for the theory?

Level OS4 Level OS4 consists of two chapters on operating systems. Chapter 8 is a description of process management. Two sections, one on loaders and another on trap handlers, illustrate the concepts with the Pep/9 operating system. Seven instructions have unimplemented opcodes that generate software traps. The operating system stores the process control block of the user’s running process on the system stack, and the interrupt service routine interprets the instruction. The classic state transition diagram for running...
and waiting processes in an operating system is thus reinforced with a specific implementation of a suspended process. The chapter concludes with a description of concurrent processes and deadlocks. Chapter 9 describes storage management, both main memory and disk memory.

**Level LG1** Level LG1 uses two chapters to present combinational and sequential circuits. Chapter 10 emphasizes the importance of the mathematical foundation of computer science by starting with the axioms of Boolean algebra. It shows the relationship between Boolean algebra and logic gates and then describes some common logic devices, including a complete logic design of the Pep/9 arithmetic logic unit (ALU). Chapter 11 illustrates the fundamental concept of a finite-state machine through the state transition diagrams of sequential circuits. It concludes with a description of common computer subsystems such as bidirectional buses, memory chips, and two-port memory banks.

**Level Mc2** Chapter 12 describes the microprogrammed control section of the Pep/9 CPU. It gives the control sequences for a few sample instructions and addressing modes and provides a large set of exercises for the others. It also presents concepts of load/store architectures, contrasting the MIPS reduced instruction set computer (RISC) machine with the Pep/9 CISC machine. It concludes with performance issues by describing cache memories, pipelining, dynamic branch prediction, and superscalar machines.

**Use in a Course**

This book offers such broad coverage that instructors may wish to omit some of the material when designing a course. I use Chapters 1–7 in a computer systems course and Chapters 10–12 in a computer organization course.

In the book, Chapters 1–5 must be covered sequentially. Chapters 6 (“Compiling to the Assembly Level”) and 7 (“Language Translation Principles”) can be covered in either order. I often skip ahead to Chapter 7 to initiate a large software project, writing an assembler for a subset of Pep/9 assembly language, so students will have sufficient time to complete it during the semester. Chapter 11 (“Sequential Circuits”) is obviously dependent on Chapter 10 (“Combinational Circuits”), but neither depends on Chapter 9 (“Storage Management”), which may be omitted. **FIGURE P.2** a chapter dependency graph, summarizes the possible chapter omissions.

**Changes Made for the Fifth Edition**

Improvements are in every chapter of this edition, and fall into one of two categories: a change in the virtual machine from Pep/8 to Pep/9 and other
changes in content. The changes in content are too numerous to list, but major changes include the following:

- **HOL6 language**—This edition changes the HOL6 language from C++ to C. The C language is more common as a systems programming language and is more appropriate for a computer systems text. The previous edition refers to the memory model of C++ consisting of global variables allocated at a fixed location in memory, local variables and parameters allocated on the run-time stack, and dynamic variables allocated from the heap. But C++ is an OO language with a more complex memory model. The above model more accurately describes C than C++. The switch is visible in three places. Input/output (I/O) in the example programs use `scanf()` and `printf()` instead of `cin` and `cout`. The pass-by reference mechanism in C has the address operator `&` explicit in the actual parameter with a corresponding pointer type in the formal parameter. Memory allocation from the heap uses `malloc` instead of `new`.

- **Sidebars**—In this edition, the brief biographies formerly found in each chapter have been replaced with a different set of sidebars. Each sidebar is a real-world example of the concepts described in that chapter. As most of the chapters describe the Pep/9 virtual machine, the sidebars for those chapters show some corresponding implementations for the Intel x86 architecture. The new sidebars give a consistent running example of this architecture so students have a better idea of how the concepts of the virtual machine relate to real-world implementations.

- **New and expanded topics**—Chapter 1 now emphasizes how binary information is quantified in both space and time by presenting the performance equation and the concept of bandwidth. QR codes and color displays are detailed examples of these concepts. Chapter 3 describes Unicode as well as UTF-32 and UTF-8 encoding. Chapter 4 discusses
big-endian and little-endian order. Chapter 7 uses Java instead of C++
as the implementation language for the example translator. The mi-
crocode examples in Chapter 12 use the new UnitPre and UnitPost
feature of the Pep/9 CPU simulator. The two-byte data bus now has
simulator support, and that topic leads to an extended discussion of
memory alignment issues and the new .ALIGN assembler directive.

Pep/8, the virtual machine for the two previous editions, has been su-
perseded by Pep/9. As the instruction sets of the two machines are different,
Pep/8 source and object programs are not compatible with those of Pep/9.
Only a few instructions are affected, however, so much remains the same,
including the set of eight addressing modes. The changes for Pep/9 include
the following:

Replacement of RETn with RET—In Pep/8, there are eight versions of
the return statement, RET0, RET1, ..., RET7. RETn deallocates n bytes
from the run-time stack and then executes a return from a function
call. The reasoning is that returns are always preceded by a dealloca-
tion of local variables, so programs are shorter if the assembly lan-
guage programmer is not required to write an ADDSP instruction to
explicitly deallocate the locals before the return.

While this ISA design may be justified on architectural principles, it
turned out to be deficient pedagogically. The problem is that students
must learn two different concepts: the deallocation mechanism for
data and the return mechanism for flow of control. Combining these
two different concepts into one statement can be confusing during
the learning process. Pep/9 now requires students to explicitly deal-
locate locals with the ADDSP statement. An added stylistic advantage
is that the explicit ADDSP to deallocate locals at the end of a function
corresponds directly to the SUBSP at the beginning of the function to
allocate locals.

Memory-mapped I/O—Of all the instructions in the Pep/8 instruc-
tion set, the most unrealistic are CHARI and CHARO for character in-
put and output. Most real computer systems map input and output
devices to main memory, which is now the design of Pep/9. In the
new instruction set, there are no native input and output instructions.
Instead, the Pep/8 instruction

CHARI alpha,ad

is replaced by the Pep/9 instructions

LDBA charIn,d ;Load byte to A from charIn
STBA alpha,ad ;Store byte from A to alpha
where charIn is the input device. The Pep/8 instruction

\[
\text{CHARO beta, ad}
\]

is replaced by the Pep/9 instructions

\[
\text{LDBA beta, ad ; Load byte to A from beta} \\
\text{STBA charOut, d ; Store byte to charOut}
\]

where charOut is the output device.

In the above code fragments, ad represents any valid addressing mode for the instruction. Symbols charIn and charOut are defined in the Pep/9 operating system and stored as machine vectors at the bottom of memory. Their values are included automatically in the symbol table of the assembler.

One disadvantage of memory-mapped I/O is that every CHARI and CHARO statement in a Pep/8 program must now be written as two statements, making programs longer. This disadvantage is mitigated by the fact that the trap instructions DECI, DECO, and STRO work as before, as the native I/O statements are hidden inside their trap routines.

The advantage is that students learn firsthand how memory-mapped I/O works by loading from the input device address and storing to the output device address. This requirement also illustrates the concept and the use of the memory map, a topic that students have a tendency to avoid with Pep/8. There is also a nice connection with the example in Chapter 11 on address decoding that shows how to wire an eight-port I/O chip into the memory map.

**New native instruction CPBr**—In Pep/8, byte quantities must be compared with CP, which compares two-byte quantities. Consequently, the high-order byte of the comparison must be considered, sometimes by clearing the high-order byte of the register before the comparison is made. The resulting assembler code for doing byte comparisons is convoluted.

CPBr is a new compare byte instruction that sets the status bits without regard to the high-order byte of the register. The resulting code is simpler to understand and to write.

**Improved mnemonics**—Pep/9 renames the mnemonics for the compare, load, and store instructions, as FIGURE P.3 shows. The new scheme retains the letters CP for compare, LD for load, and ST for store; however, the scheme is now consistent in using the letters W for word, which is now required, and B for byte with this group of instructions. Not only is this naming convention more consistent, but there is a tendency for students to forget the meaning of a word (two
bytes in the Pep computers). Including the letter \( W \) in the mnemonics for the two-byte instructions reinforces the meaning of “word.”

- **New trap instruction** \( \text{HEXD} \) —Pep/9 eliminates the \( \text{NOP2} \) and \( \text{NOP3} \) trap instructions from the instruction set, which, together with the elimination of the \( \text{RETn} \) and character I/O instructions, allows the inclusion of another nonunary trap instruction, \( \text{HEXD} \), which stands for *hexadecimal output*, was available in Pep/7 and is resurrected in Pep/9. It outputs a word as four hexadecimal characters.

- **Addressing mode nomenclature** —Pep/9 changes the name stack-indexed deferred addressing to stack-deferred indexed addressing and the corresponding assembler notation from \( \text{sxf} \) to \( \text{sfx} \). This change more accurately reflects the semantics of the addressing mode, as the stack-deferred operation happens before the index operation.

- **Expanded MIPS coverage** —The MIPS architecture continues to be the RISC model contrasted with the Pep/9 CISC model. Its coverage is expanded with a more extensive and systematic description of all the MIPS addressing modes and instruction types and their implementation at the LG1 level. This edition also includes a more extensive exposition of RISC design principles, as illustrated by the MIPS architecture.

### Unique Features

*Computer Systems* has several unique features that differentiate it from other texts on computer systems, assembly language, and computer organization.

- **Conceptual approach** —Many textbooks attempt to stay abreast of the field by including the latest technological developments, such as communication protocol specifications for the newest peripheral devices. They typically have how-this-device-works narrative explanations throughout. This text eschews such material in favor of selecting only those computing concepts that are fundamental, the mastery of which provides a basis for understanding both current and future technology. For instance, in mastering the concept of the space/time tradeoff, it is more important for students to experience the ideas with digital circuit design problems than to simply read about them in general. As another example, the concept of hardware parallelism is mastered best by learning how to combine cycles in a microcode implementation of an ISA instruction.

- **Problem-solving emphasis** —Students retain less when they only hear about or read about a subject. They retain more when they experience the subject. *Computer Systems* reflects this emphasis through the nearly 400 problem-solving exercises at the end of the chapters, many
with multiple parts. Rather than ask the student to repeat verbiage from the text, the exercises require quantitative answers, or the analysis or design of a program or digital circuit at one of the levels of abstraction of the system.

- **Consistent machine model**—The Pep/9 machine, a small CISC computer, is the vehicle for describing all the levels of the system. Students clearly see the relation between the levels of abstraction because they either program or design digital circuits for that machine at all the levels. For example, when they design an ALU component at the LG1 level, they know where the ALU fits in the implementation of the ISA3 level. They learn the difference between an optimizing and nonoptimizing compiler by translating C programs to assembly language as a compiler would. Using the same machine model for these learning activities at the different levels is a huge productivity advantage because the model is consistent from top to bottom. However, *Computer Systems* also presents the MIPS machine to contrast RISC design principles with microprogrammed CISC designs.

- **Complete program examples**—Many computer organization and assembly language texts suffer from the code fragment syndrome. The memory model, addressing modes, and I/O features of Pep/9 enable students to write complete programs that can be easily executed and tested without resorting just to code fragments. Real machines, and especially RISC machines, have complex function-calling protocols involving issues like register allocation, register spillover, and memory alignment constraints. Pep/9 is one of the few pedagogic machines—perhaps the only one—that permits students to write complete programs with input and output using the following: global and local variables, global and local arrays, call by value and by reference, array parameters, switch statements with jump tables, recursion, linked structures with pointers, and the heap. Assignments to write complete programs further the goal of learning by doing, as opposed to learning by reading code fragments.

- **Integration of theory and practice**—Some readers observe that Chapter 7 on language translation principles is unusual in a computer systems book. This observation is a sad commentary on the gulf between theory and practice in computer science curricula and perhaps in the field of computer science itself. Because the text presents the C language at Level HOL6, assembly language at Level Asmb5, and machine language at Level ISA3, and has as one goal understanding the relationship between the levels, a better question is: How could a chapter on language translation principles not be included? *Computer Systems* incorporates theory whenever possible to bolster practice.
For example, it presents Boolean algebra as an axiomatic system with exercises for proving theorems.

*Breadth and depth*—The material in Chapters 1–6 is typical for books on computer systems or assembly language programming, and that in Chapters 8–12 for computer organization. Combining this breadth of material into one volume is unique and permits a consistent machine model to be used throughout the levels of abstraction of the complete system. Also unique is the depth of coverage at the digital circuit level, which takes the mystery out of the component parts of the CPU. For example, *Computer Systems* describes the implementations of the multiplexers, adders, ALUs, registers, memory subsystems, and bidirectional buses for the Pep/9 CPU. Students learn the implementation down to the logic gate level, with no conceptual holes in the grand narrative that would otherwise have to be taken on faith without complete understanding.

*Computer Systems* answers the question: What is the place of assembly language programming and computer organization in the computer science curriculum? It is to provide a depth of understanding about the architecture of the ubiquitous von Neumann machine. This text retains its unique goal to provide a balanced overview of all the main areas of the field, including the integration of software and hardware and the integration of theory and practice.

**Computer Science Curricula 2013**

The first edition of *Computer Systems* was published in 1999. From the beginning, its goal was to unify the presentation of computer systems using a single virtual machine, Pep/6 in the first edition, to illustrate the relationships between all seven levels of abstraction in a typical computer system. The text cut across many of the standard courses at the time (and even now), taking a breadth approach and, in so doing, sacrificing the depth of traditional details taught in standard assembly language, operating systems, computer architecture, computer organization, and digital circuit design courses.

The latest joint ACM/IEEE curriculum guidelines, *Computer Science Curricula 2013*¹ (CS2013), note the curriculum challenges arising from the rapid expansion of the field:

The growing diversity of topics potentially relevant to an education in Computer Science and the increasing integration of computing with other...
disciplines create particular challenges for this effort. Balancing topical growth with the need to keep recommendations realistic and implementable in the context of undergraduate education is particularly difficult.

One response to the challenge is the evolution of the guidelines in precisely the direction that Computer Systems has occupied since its first edition. CS2013 restructures previous “bodies of knowledge,” deleting some old areas and creating some new ones. One of the new knowledge areas (KAs) of the guidelines is Systems Fundamentals:

In previous curricular volumes, the interacting layers of a typical computing system, from hardware building blocks, to architectural organization, to operating system services, to application execution environments . . . were presented in independent knowledge areas. The new Systems Fundamentals KA presents a unified systems perspective and common conceptual foundation for other KAs.

The goal of the new Systems Fundamentals KA in CS2013 is identical to the goal of Computer Systems, namely to present a “unified systems perspective” and a common conceptual foundation for other topics of computer science. Computer Systems is a mature text that uniquely satisfies this important new goal of the latest computer science curriculum guidelines.

Support Materials

The support material listed below is available from the publisher’s website (go.jblearning.com/warford5e).

Pep/9 Assembler and Simulator The Pep/9 machine is available for MS Windows, Mac OS X, and Unix/Linux systems. The assembler features the following:

- An integrated text editor
- Error messages that are inserted within the source code at the place where the error is detected
- Student-friendly machine language object code in hexadecimal format
- The ability to code directly in machine language, bypassing the assembler
- The ability to redefine the mnemonics for the unimplemented opcodes that trigger synchronous traps

The simulator features the following:

- Simulated ROM that is not altered by store instructions
- A small operating system burned into simulated ROM that includes a loader and a trap handler system
An integrated debugger that allows for break points, single-step execution, CPU tracing, and memory tracing

The option to trace an application, the loader, or the operating system in any combination

The ability to recover from endless loops

The ability to modify the operating system by designing new trap handlers for the unimplemented opcodes

Every example from the text built into the application, making it a useful tool for class demonstrations

Pep/9 CPU Simulator  A CPU simulator, also available for MS Windows, Mac OS X, and Unix/Linux systems, is available for use in the computer organization course. The CPU simulator features the following:

- Color-coded display paths that trace the data flow, depending on control signals to the multiplexers
- A single-cycle mode of operation with graphical user interface inputs for each control signal and instant visual display of the effects of the signal
- A multicycle mode of operation with an integrated text editor for the student to write Mc2 microcode sequences and execute them to implement ISA3 instructions
- New to this edition, a set of unit tests for every example and every microcode problem built into the application

Lecture Slides  A complete set of about 50 to 125 lecture slides per chapter is available in PDF format. The slides include every figure from the text as well as summary information, often in bullet-point format. They do not, however, include many examples, so they leave room for instructor presentation of examples and instructor-led discussions.

Exam Handouts  A set of exam handouts, including reference information such as the ASCII table, instruction set tables, etc., are provided for reference during exams and study sessions.

Digital Circuit Labs  A set of six digital circuit labs provides hands-on experience with physical breadboards. The labs illustrate the combinational and sequential devices from Chapters 10 and 11 with many circuits that are not in the book. Students learn practical digital design and implementation
concepts that are beyond the scope of the text itself. They follow the sequence of topics from the text, beginning with combinational circuits and progressing through sequential circuits and ALUs.

**Solutions Manual** Solutions to selected exercises are provided in an appendix. Solutions to all the exercises and problems are available to instructors who adopt the book.

For information on how qualified instructors can access these resources, please contact your Jones & Bartlett Learning representative.

**Acknowledgments**

Pep/1 had 16 instructions, one accumulator, and one addressing mode. Pep/2 added indexed addressing. John Vannoy wrote both simulators in Algol W. Pep/3 had 32 instructions and was written in Pascal as a student software project by Steve Dimse, Russ Hughes, Kazuo Ishikawa, Nancy Brunet, and Yvonne Smith. In an early review, Harold Stone suggested many improvements to the Pep/3 architecture that were incorporated into Pep/4 and carried into later machines. Pep/4 had special stack instructions, simulated ROM, and software traps. Pep/5 was a more orthogonal design, allowing any instruction to use any addressing mode. John Rooker wrote the Pep/4 system and an early version of Pep/5. Gerry St. Romain implemented a Mac OS version and an MS-DOS version. Pep/6 simplified indexed addressing and included the complete set of conditional branch instructions. John Webb wrote the trace facility using the BlackBox development system. Pep/7 increased the installed memory from 4 KiB to 32 KiB. Pep/8 increased the number of addressing modes from 4 to 8 and the installed memory to 64 KiB. The GUI version of the Pep/8 assembler and simulator was implemented in C++ and maintained by teams of students using the Qt development system. The teams included Deacon Bradley, Jeff Cook, Nathan Counts, Stuart Fox, Dave Grue, Justin Haight, Paul Harvey, Hermi Heimgartner, Matt Highfield, Trent Kyono, Malcolm Lipscomb, Brady Lockhart, Adrian Lomas, Scott Mace, Ryan Okelberry, Thomas Rampelberg, Mike Spandrio, Jack Thomason, Daniel Walton, Di Wang, Peter Warford, and Matt Wells. Ryan Okelberry also wrote the Pep/8 CPU simulator. Luciano d’Ilori wrote the command line version of the assembler. The latest versions of Pep/8 and Pep/8 CPU, and the current version of Pep/9 and Pep/9 CPU, are complete rewrites in Qt by Emily Dimpfl and myself.

More than any other book, Tanenbaum’s *Structured Computer Organization* has influenced this text. This text extends the level structure of Tanenbaum’s book by adding the high-order programming level and the applications level at the top.
The following reviewers of the manuscript, students, and users of the previous edition shaped the final product significantly: Kenneth Araujo, Ziya Arnavut, Wayne P. Bailey, Leo Benegas, Jim Bilitski, Noni Bohonak, Dan Brennan, Michael Yonshik Choi, Christopher Cischke, Collin Cowart, Lionel Craddock, William Decker, Fadi Deek, Peter Drexel, Gerald S. Eisman, Victoria Evans, Mark Fienup, Paula Ford, Brooke Fugate, Robert Gann, David Garnick, Ephraim P. Glinert, John Goulden, Dave Hanscom, Michael Hennessy, Paul Jackowitz, Mark Johnson, Michael Johnson, Amitava Karmaker, Michael Kirkpatrick, Peter MacPherson, Andrew Malton, Robert Martin, John Maxfield, John McCormick, Richard H. Mercer, Jonathan Mohr, Randy Molmen, Hadi Moradi, John Motil, Mohammad Muztaba Fuad, Peter Ng, Bernard Nudel, Carolyn Oberlink, Nelson Passos, Wolfgang Pelz, James F. Peters III, James C. Pleasant, Eleanor Quinlan, Glenn A. Richard, Gerry St. Romain, David Rosser, Sally Schaffner, Peter Smith, Harold S. Stone, Robert Tureman, J. Peter Weston, and Norman E. Wright. Joe Piasentin provided artistic consultation. Two people who influenced the design of Pep/8 significantly are Myers Foreman, who was a source of many ideas for the instruction set, and Douglas Harms, who suggested (among other improvements) the MOVSPA instruction that makes possible the passing of local variables by reference.

At Jones & Bartlett Learning, Acquisitions Editor Laura Pagluica, Director of Vendor Management Amy Rose, Production Editor Vanessa Richards, and Editorial Assistant Taylor Ferracane provided valuable support and with whom were a true joy to work. Kristin Parker captured the flavor of the book with her striking cover design.

I am fortunate to be at an institution that is committed to excellence in undergraduate education. Pepperdine University, in the person of Ken Perrin, provided the creative environment and the professional support in which the idea behind this project was able to evolve. My wife, Ann, provided endless personal support. To her I owe an apology for the time this project has taken, and my greatest thanks.

Stan Warford
Malibu, California