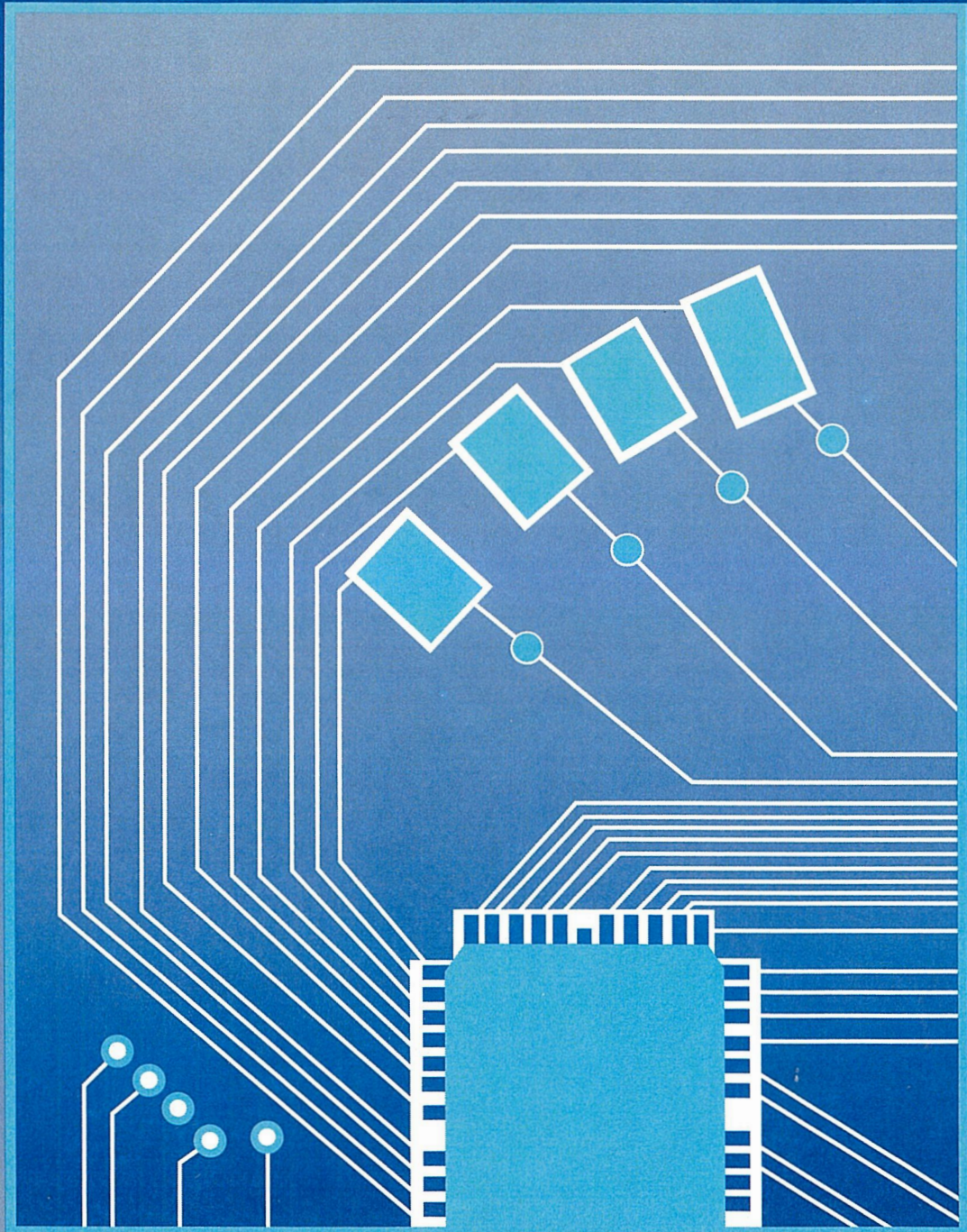


Michael J. Flynn

Computer Architecture

PIPELINED AND PARALLEL PROCESSOR DESIGN



Editorial, Sales, and Customer Service Offices

Jones and Bartlett Publishers

One Exeter Plaza

Boston, MA 02116

1-800-832-0034

617-859-3900

Jones and Bartlett Publishers International

7 Melrose Terrace

London W6 7RL

England

Copyright © 1995 by Jones and Bartlett Publishers, Inc.

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

Library of Congress Cataloging-in-Publication Data

Flynn, Michael J., 1934 -

Computer architecture : pipelined and parallel processor design /

Michael J. Flynn.

p. cm.

Includes bibliographical references and index.

ISBN 0-86720-204-1

1. Computer architecture. 2. Microprocessors--Design and construction. I. Title.

QA76.9.A73F58 1995

004.2'2--dc20

94-41225

CIP

Printed in the United States of America

99 98 97 96 95 10 9 8 7 6 5 4 3 2 1

Contents

Preface	xv
Acknowledgments	xvii
1 Architecture and Machines	1
1.1 Some Definitions and Terms	3
1.2 Interpretation and Microprogramming	5
1.3 The Instruction Set	9
1.4 Basic Data Types	13
1.5 Instructions	21
1.5.1 Classes of Operations	21
1.5.2 Instruction Mnemonics	25
1.5.3 General Machine Conventions	26
1.5.4 Branches	30
1.5.5 Register Sets and Addressing Modes	32
1.5.6 Instruction Code Examples	33
1.5.7 Other Instruction Set Issues	35
1.5.8 Program Size	37
1.6 Addressing and Memory	39
1.6.1 Process Addressing	39
1.6.2 System Addresses and Segmentation	41
1.6.3 Memory Space	43
1.7 Virtual to Real Mapping	45
1.8 Basic Instruction Timing	47
1.8.1 Examples of Well-mapped Machine Instruction Timing	49
1.8.2 Overlapped and Pipelined Processors	53
1.9 Conclusions	53
1.10 Historical Development of Computers	54
1.11 Annotated Bibliography	56
1.12 Problem Set	58

2	Time, Area, and Instruction Sets	63
2.1	Introduction	63
2.2	Time	64
2.2.1	The Nature of a Cycle	64
2.2.2	Partitioning Instruction Execution into Cycles	65
2.2.3	Clocking Overhead and Reliable Clocking	66
2.2.4	Pipelined Processors	70
2.2.5	Optimum Pipelining	70
2.2.6	Cycle Quantization	77
2.2.7	Wave Pipelining	79
2.3	Cost-Area	83
2.3.1	Area	84
2.3.2	Data Storage	93
2.4	Technology State of the Art	99
2.5	The Economics of a Processor Project: A Study	103
2.5.1	Phase 1: Development	106
2.5.2	Phase 2: Early Manufacturing	106
2.5.3	Phase 3: Production	107
2.5.4	Phase 4: All Good Things Must Come to an End	108
2.6	Instruction Sets: Processor Evaluation Metrics	109
2.6.1	Program Execution	110
2.6.2	Instruction Set Comparisons	112
2.6.3	Invariant Effects	117
2.6.4	Code Density	118
2.6.5	Role of Registers, Evaluation Stacks, and Data Buffers	124
2.7	Conclusions	132
2.8	Some Areas for Further Research	133
2.9	Data Notes	134
2.10	Annotated Bibliography	134
2.11	Problem Set	136
3	Data: How Programs Behave	141
3.1	Introduction	141
3.2	Instruction Usage	142
3.2.1	Data Categories	142
3.2.2	Format Distribution	144
3.2.3	Operation Set Distribution	144
3.3	Process Management	150
3.3.1	Procedure Calls: User State	150
3.3.2	Calls to the System	153

3.4	Breaks in Machine Execution	156
3.4.1	Instruction Run Length	156
3.4.2	Branches	158
3.4.3	Branch Target Distribution	161
3.4.4	Condition Code Testing	163
3.4.5	Move and Arithmetic Class Operations	164
3.4.6	Register-Based Addressing	167
3.4.7	Decimal and Character Operand Length	170
3.5	Conclusions	174
3.6	Some Areas for Further Research	174
3.7	Data Notes	175
3.8	Annotated Bibliography	176
3.9	Problem Set	177
4	Pipelined Processor Design	181
4.1	Introduction	181
4.1.1	Evolution of a Computer Processor Family	182
4.1.2	Processor Design	185
4.1.3	Organization of the Chapter	185
4.2	Approaching Pipelined Processors	186
4.2.1	Examples of Pipeline Implementations	190
4.3	Evaluating Pipelined Processor Performance	193
4.4	Design of a Pipelined Processor	213
4.4.1	Cache Access Controller	214
4.4.2	Accounting for the Effect of Buffers in a Pipelined System	215
4.4.3	Buffer Design	216
4.4.4	Designing a Buffer for a Mean Request Rate	216
4.4.5	I-Buffers Designed for Maximum Request Rates	219
4.5	Branches	222
4.5.1	Branch Elimination	225
4.5.2	Branch Speedup	225
4.5.3	Branch Prediction Strategies	226
4.5.4	Branch Target Capture: Branch Target Buffers	236
4.6	Interlocks	240
4.6.1	Decoder and Interlocks	240
4.6.2	Bypassing	241
4.6.3	Address Generation Interlocks	242
4.6.4	Execution Interlocks and Interlock Tables	243
4.7	Run-On Delay	250

4.8	Miscellaneous Effects	251
4.8.1	Store in Instruction Stream Delay	251
4.9	Conclusions	259
4.10	Some Areas for Further Research	260
4.11	Data Notes	260
4.12	Annotated Bibliography	261
4.13	Problem Set	262
5	Cache Memory	265
5.1	Introduction	265
5.2	Basic Notions	266
5.3	Cache Organization	269
5.4	Cache Data	274
5.5	Adjusting the Data for Cache Organization	277
5.6	Write Policies	280
5.7	Strategies for Line Replacement at Miss Time	283
5.7.1	Fetching a Line	284
5.7.2	Line Replacement	286
5.8	Cache Environment	287
5.9	Other Types of Cache	291
5.10	Split I- and D-Caches	294
5.10.1	I- and D-Caches	294
5.10.2	Code Density Effects	297
5.11	On-Chip Caches	299
5.12	Two-Level Caches	302
5.12.1	Logical Inclusion	308
5.13	Write Assembly Cache	309
5.14	Cache References per Instruction	311
5.14.1	Instruction Traffic	311
5.14.2	Data Traffic	314
5.15	Technology-Dependent Cache Considerations	317
5.16	Virtual-to-Real Translation	322
5.16.1	Translation Lookaside Buffer (TLB)	323
5.17	Overlapping the T cycle in $V \rightarrow R$ Translation	325
5.17.1	Set Associative Caches	326
5.17.2	Virtual Caches	327
5.17.3	Real Caches Using Colored Pages	328
5.18	Studies	329
5.18.1	Actual Reference Traffic	331
5.19	Design Summary	336

5.19.1	Cache Evaluation Design Rules	336
5.19.2	Cache/TLB Excess CPI Design Rules	337
5.20	Conclusions	337
5.21	Some Areas for Further Research	338
5.22	Data Notes	338
5.23	Bibliography	340
5.24	Problem Set	341
6	Memory System Design	345
6.1	Introduction	345
6.2	The Physical Memory	349
6.2.1	The Memory Module	350
6.2.2	Error Detection and Correction	354
6.2.3	Memory Buffers	358
6.2.4	Partitioning of the Address Space	358
6.3	Models of Simple Processor-Memory Interaction	360
6.3.1	Memory Systems Design	361
6.3.2	Multiple Simple Processors	362
6.3.3	Hellerman's Model	363
6.3.4	Strecker's Model	363
6.3.5	Rau's Model	365
6.4	Processor-Memory Modeling Using Queueing Theory	365
6.4.1	Performance Models of Processor Memory Interactions	368
6.4.2	Arrival Distribution	369
6.4.3	Service Distribution	370
6.4.4	Terminology	371
6.4.5	Queue Properties	372
6.5	Open-, Closed-, and Mixed-Queue Models	374
6.5.1	The Open-Queue (Flores) Memory Model	376
6.5.2	Closed Queues	378
6.5.3	Mixed Queues	382
6.6	Waiting Time, Performance, and Buffer Size	384
6.6.1	Pipelined Processors	385
6.6.2	Designing a $M/M/1$ Buffer Given a Mean Queue Size .	389
6.6.3	Comparison of Memory Models	391
6.7	Review and Selection of Queueing Models	394
6.8	Processors with Cache	396
6.8.1	Fully and Partially Blocking Caches	397
6.8.2	Accessing a Line ($T_{\text{line access}}$)	397
6.8.3	Contention Time (T_{busy}) and Copyback Caches	400

10.1.2	Design Alternatives	665
10.1.3	Pipeline Timing Analysis	665
10.1.4	Pipeline Penalty Analysis	671
10.1.5	Cache and Memory Analysis	676
10.1.6	Cost-Performance Analysis	677
10.2	Area Performance Analysis of Processors	683
10.2.1	The Problem	683
10.2.2	Specifications	685
10.2.3	Assumptions	688
10.2.4	The Design	688
10.2.5	Analysis	704
10.3	Study Results	717
10.4	Conclusions	718
Appendix A DTMR Cache Miss Rates		719
A.1	Basic DTMR	719
A.2	Associativity Adjustments	719
A.3	User + System	721
A.4	Transaction-Based Systems	722
A.5	Multiprogrammed (Warm Cache) Environment	728
Appendix B SPECmark vs. DTMR Cache Performance		741
Appendix C Modeling System Effects in Caches		743
C.1	Cold Start Cache	743
C.2	Cache Misses in Multiprogramming Environment	744
Appendix D New DRAM Technologies		747
D.1	Typical Performance Enhancements	747
D.2	Enhanced DRAM	748
D.3	Synchronous DRAM	748
D.4	Cache DRAM	748
D.5	Rambus DRAM	748
D.6	Ramlink DRAM	749
D.7	Chip Level Summary	749
Appendix E M/G/1 Queues		751
Appendix F Some Details on Bus-Based Protocols		755
Bibliography		765
Index		782

6.8.4	I/O Effects	402
6.8.5	Performance Effects	403
6.8.6	Copyback Cache Study	404
6.8.7	Simple Write-Through Caches	407
6.8.8	Write-Through Cache Example	410
6.8.9	Shared Bus	411
6.8.10	Nonblocking Caches	415
6.8.11	Interleaved Caches	416
6.9	Conclusions	417
6.10	Some Areas for Further Research	418
6.11	Data Notes	419
6.12	Annotated Bibliography	419
6.13	Problem Set	420
7	Concurrent Processors	425
7.1	Introduction	425
7.2	Vector Processors	427
7.2.1	Vector Functional Units	428
7.2.2	Vector Instructions/Operations	432
7.2.3	Vector Processor Implementation	435
7.2.4	A Generic Vector Processor	437
7.3	Vector Memory	438
7.3.1	The Special Case of Vector Memory	438
7.3.2	Modeling Vector Memory Performance	444
7.3.3	Gamma (γ)-Binomial Model	446
7.3.4	Bypassing between Vector Instructions	450
7.4	Vector Processor Speedup	452
7.4.1	Basic Issues	452
7.4.2	Measures	456
7.5	Multiple-Issue Machines	457
7.6	Out-of-Order and Multiple-Instruction Execution	459
7.6.1	Data Dependencies	460
7.6.2	Representing Data Dependencies	462
7.6.3	Other Types of Dependencies	464
7.6.4	When and How to Detect Instruction Concurrency	465
7.6.5	Two Scheduling Implementations	468
7.6.6	An Improved Scoreboard	476
7.6.7	Dealing with Out-of-Order Execution	487
7.6.8	Interleaved Caches	491
7.6.9	Branches and Speculative Execution	493

7.6.10 Adaptive Speculation	494
7.6.11 Results	495
7.7 Comparing Vector and Multiple-Issue Processors	500
7.7.1 Cost Comparison	500
7.7.2 Performance Comparison	503
7.7.3 Alternative Organizations	506
7.8 Conclusions	506
7.9 Some Areas for Further Research	507
7.10 Data Notes	508
7.11 Annotated Bibliography	508
7.12 Problem Set	509
8 Shared Memory Multiprocessors	511
8.1 Basic Issues	511
8.2 Partitioning	513
8.3 Scheduling	519
8.3.1 Run-Time Scheduling Techniques	520
8.4 Synchronization and Coherency	523
8.5 The Effects of Partitioning and Scheduling Overhead	526
8.5.1 Grain Size and Overhead	531
8.6 Types of Shared Memory Multiprocessors	532
8.7 Multithreaded or Shared Resource Multiprocessing	533
8.8 Memory Coherence in Shared Memory Multiprocessors	538
8.9 Shared-Bus Multiprocessors	541
8.9.1 Snoopy Protocols	541
8.9.2 Bus-Based Models	551
8.10 Scalable Multiprocessors	555
8.11 Directory-Based Protocols	556
8.11.1 Directory Structure	557
8.11.2 Invalidate Protocols	559
8.11.3 Update Protocols	562
8.12 Evaluating Some Systems Alternatives	563
8.13 Interconnections	569
8.14 Static Networks	571
8.14.1 Links and Nodes	572
8.15 Dynamic Networks	574
8.16 Evaluating Interconnect Networks	578
8.16.1 Direct Static vs. Indirect Dynamic	578
8.16.2 Network Dimensionality and Link-Limited Network	585
8.17 Hotspots and Combining	588

8.18	Other Characterizations of Multiprocessors	590
8.19	Conclusions	592
8.20	Some Areas for Further Research	594
8.21	Annotated Bibliography	594
8.22	Problem Set	595
9	I/O and the Storage Hierarchy	599
9.1	The Role of I/O	599
9.2	Evolution of I/O Systems Organization	601
9.2.1	I/O Processors/Channels	603
9.2.2	I/O System Support for Multiprocessors	604
9.3	Design of Storage Systems	605
9.3.1	Disk Technology	605
9.3.2	The Disk Device	606
9.4	Simple I/O Transactions	609
9.4.1	Multiple Servers	614
9.4.2	Single-Server Low Population (n)	615
9.4.3	Disk Modeling	618
9.4.4	Multiprogramming Models and Inverted Servers	621
9.4.5	Improving I/O Response and Capacity	630
9.5	I/O Traffic and Virtual Memory Effects	632
9.5.1	Basic I/O Request Rate	633
9.5.2	Virtual Memory I/O Traffic	635
9.5.3	Disk Cache Buffers	636
9.5.4	Concurrent Disks	638
9.5.5	Clusters of Independent Disks	643
9.5.6	Striping	644
9.5.7	Disk Arrays	646
9.5.8	Composite Configurations	648
9.6	Some Practical Considerations	652
9.7	Redundancy in Disk Arrays	654
9.8	Conclusions	657
9.9	Some Areas for Further Research	658
9.10	Data Notes	658
9.11	Annotated Bibliography	659
9.12	Problem Set	660
10	Processor Studies	663
10.1	The Baseline Mark II	663
10.1.1	Design Assumptions	664

Chapter 1

An Introduction to Architecture and Machines

The most important influences on the design of a computing machine or processor are the instruction set (or sets) executed and the technology used in its implementation. In order to understand the machine design process, one must first understand the instruction set and the function of each instruction.

The term *computer architecture* [16] is usually used to describe the instruction set together with major implementation details such as bus structure, storage type, register type, etc. While the instruction set implies many of these implementation details, the *implementation* is separate from the architecture, since the implementation is the mechanism that executes or interprets the instruction (Figure 1.1). If an instruction specifies 16 general-purpose registers, most implementations have at least 16 registers. Implementations that have significantly more or fewer than this number are interesting exceptions, which we consider later in this book. Our primary interest in this chapter is to review the concepts and requirements implied by instruction sets, since they form the basis for intelligent machine design.

The term *architecture* includes the instruction set, which yields an interpretable program representation. Multiple implementations of the System 370 instruction set, the VAX [32, 31], the Motorola M680x0, the Intel iAP x86 [58], and the RISC instruction sets (such as SPARC) illustrate the various cost-performance technology tradeoffs possible in creating a machine.

One of the longest-lived instruction sets is the IBM mainframe System/360 (S/360), introduced in 1963 [140, 141, 142, 143]. Although over the years it has been relabeled (S/370, S/370 ESA, Enterprise system, and System/390) and extended, the S/360 core instruction set remains unchanged and probably will continue for some time. Over the years, perhaps 100 different machines have implemented this core instruction set, including manufacturers such as IBM (S/370, 30X0 models, Enterprise system, etc.), Amdahl (V series), Hitachi (H series), Fujitsu, and others. For general-purpose usage, the hallmark of a good instruction set is its ability to provide efficient (cost-performance) machine implementations across wide variances in technology (both hardware and software). As well as being long-lived,

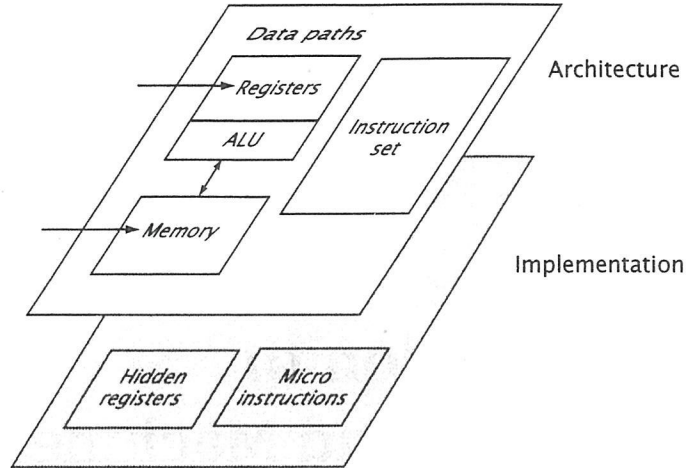


Figure 1.1 A two-level view of a computer architecture and its implementation.

a well-designed instruction set ought to allow for multiple machine implementations at various cost-performance points, to accommodate different user budgets and needs.

Occasionally, a seemingly well-designed instruction set may become difficult to implement because of unforeseen shifts in technology. The DEC VAX series (late 1970s) was an impressive extension of their earlier PDP-11 series instruction set introduced in 1968. Its instruction set provided an elaborate operational vocabulary that minimized the number of bits needed to represent or execute a program. For the technology of the '70s, this was of significant advantage; memory was an expensive part of the machine. Unfortunately, in order to achieve this code density, the instruction size and interpretation were highly variable. This was not a problem for the simple unpipelined machine implementations of the '70s, but became a significant hurdle for complex pipelined implementations 10 or 15 years later. Moreover, the cost of memory and cache has significantly decreased over the years, decreasing the value of code density. The VAX instruction set (and PDP-11) paradigm has also significantly influenced instruction set designers of contemporary (circa 1980) machines such as the Motorola M680x0 series and (to a lesser extent) the Intel 8086.

RISC instruction sets of the '80s were developed for microprocessors in part to overcome the complex implementations required by VAX-type instruction sets. RISC instruction sets used simple fixed-size instructions that facilitated implementation for complex pipelined machines. Code density was regarded as unimportant, since it complicated instruction decoding. With the advent of on-chip caches for microprocessors, optimization of memory usage is now more important, so code density may be the next hurdle for RISC implementations.

There are *always* tradeoffs in instruction set design. A well-designed instruction set allows variability in implementation technology and is less sensitive to technological changes. As time goes by, even a well-designed set must undergo changes—additions to accommodate new functionality

and perhaps a de-emphasis of older features. Thus, at any moment, a successful architecture includes an instruction set consisting of:

- A core of frequently used instructions.
- Some features extending or correcting limitations in the original design.
- Some instructions no longer expected to be used (either superseded or “out of vogue”), which remain for reasons of compatibility.

This book is not intended as a historical retrospective of instruction sets. Rather, our approach is to stress the underlying principles in computer architecture: recognizing various generic instruction set types, similarities, and (essential) differences. Our primary purpose is to study *processor design*: the realization of efficient implementations *given* an instruction set and architectural limitations (cache size, etc.); but of necessity we will also study the effect of variations in architecture on the implementation efficiency (cost and performance). There are a number of books which provide a more introductory treatment to processor design—see [271, 127, 123] and [296].

1.1 Some Definitions and Terms

- A *state* is a particular configuration of storage (i.e., registers or memory), and a *state transition* is a change in that configuration.
- A *cycle* is the time between state transitions. If the storage being reconfigured is registers, we have an *internal* or *machine cycle*. If the storage is memory, we have a *memory cycle*.
- A *command* is a function that, when applied to a particular state, generates the next state. *Command* is a generic term used to describe various types of instructions.
- A *process* is a sequence of commands and an initial state. In a sense, a process is a *macro command*, since it applies to an initial state and generates a final state.
- A *machine*¹ is a set of commands and storage together with an implementation that causes the state transitions determined by the command (Figure 1.1).
- The *storage* is the range and domain of the commands. A machine's storage is only that storage referred to by its instruction set. If the instruction set cannot refer to a hidden register, it is not part of the storage, but rather part of the implementation.

The implementation that interprets the instruction can itself be a machine (that is, have instruction set, storage, and implementation); if so, the original (outermost) machine is called the *image* (or the micro-) *machine* and

¹A *computer* then is a machine that can modify the sequence of commands being executed, i.e., its control flow. We use the terms machine and computer interchangeably.

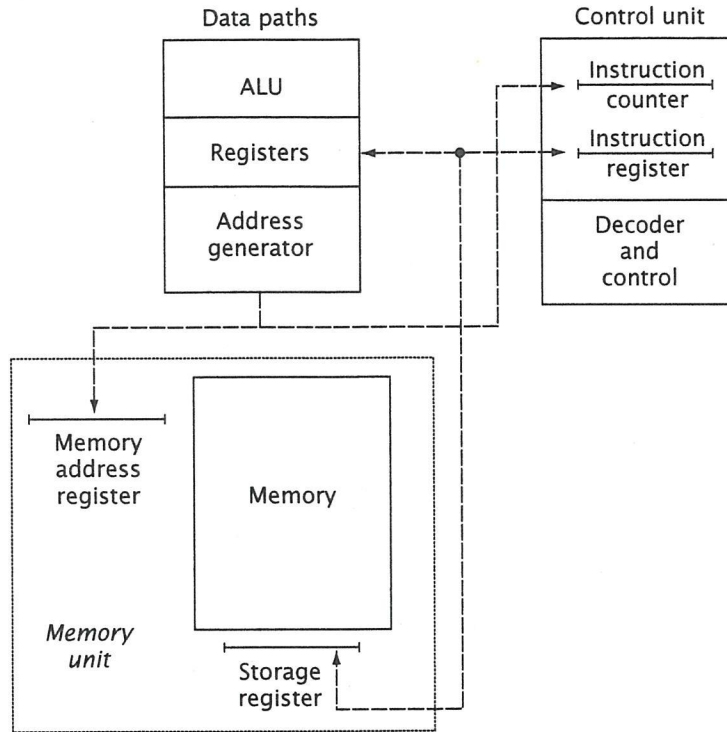


Figure 1.2 Basic units of a processor and (some) paths.

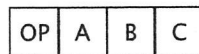


Figure 1.3 The instruction.

the implementation that interprets the commands is called the *host machine* (or the macro-) *machine*.

The set of all image commands and storage is defined as the *architecture* of the machine. By convention, image commands are called *instructions* or, if the distinction is important, *image instructions*, and the commands comprising the host are called *microinstructions* or *host instructions*.

The basic functional units of a processor are shown in Figure 1.2. The control unit contains all the registers and decoding hardware required to interpret the current instructions (in the instruction register). This hardware controls the sequence of actions in the data paths to provide correct instruction execution. The *data paths* consist of the ALU (arithmetic logical unit), any other specialized execution units (floating-point, etc.), address generation hardware, data and address registers, and the interconnections between these subunits. The memory unit is the last major processor unit. It includes an input address register, an output storage register, and the memory itself.

Table 1.1 Notation for microprograms and state transitions.

$R_1 \leftarrow R_2$	Copy contents of R_2 into R_1 .
$SR \leftarrow \text{Mem} [\text{MAR}]$ or Memory read	Memory read operation: the contents of the address in memory address register (MAR) fetched into the memory output storage register (SR).
$\text{Mem} [\text{MAR}] \leftarrow SR$ or Memory write	Memory write operation: data in the storage register (SR) copied into the location specified by the address in the memory address register (MAR).
$PC \leftarrow * + 1$ or $PC \leftarrow PC + 1$	The current value $*$ in the program counter is incremented by 1.
$\text{MAR} \leftarrow [R_X] + [R_B] + D$	Contents of register specified by R_X plus contents of register specified by R_B plus D are transferred to the MAR. $[R_X]$ indicates that R_X is a register address.

The instruction (Figure 1.3) is a vector of bits partitioned into fields. These fields identify properties of, or actions to be taken by, an instruction, including:

1. Format.
2. Object address (identifiers)—the “A,” “B,” and “C” fields in Figure 1.3.
3. Operation—the “OP” in Figure 1.3.
4. Sequence control.

Some fields (or actions) may be missing from an instruction either because they are implied or are unnecessary. Notation for describing actions within instruction execution is shown in Table 1.1.

1.2 The Machine: Interpretation and Microprogramming

Management of the interpretation process is the responsibility of the decoder (a part of the implementation mechanism). The process of interpreting or executing an instruction begins with the decoding of the opcode field from the instruction. The decoder activates storage and registers for a series of state transitions that correspond to the action of the opcode. The image machine storage consists of registers and memory, which are both explicitly stated in the instruction and implicitly defined by the instruction. Explicit registers include:

1. General purpose registers (GPR).
2. Accumulators (ACC).
3. Address registers (index or base registers).

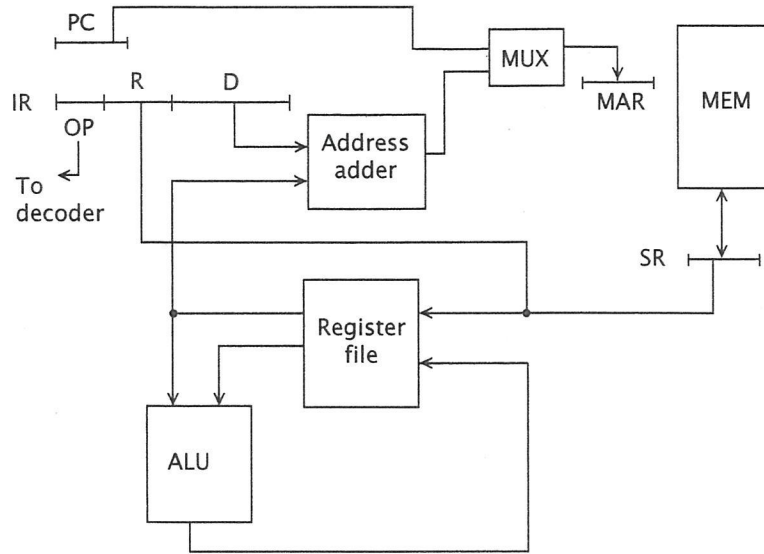


Figure 1.4 Some processor registers and data paths. *R* indicates a register in the register file, *D* an address displacement.

Implicit registers consist of the following (Figure 1.4):

1. Program or instruction counter (PC). Most instruction formats imply the next instruction in sequence as the current location plus the length of the current instruction.
2. Instruction register (IR). This register holds the instruction being interpreted or executed. Decoding is performed on the opcode held in this register.
3. Memory address register (MAR). This is the address register for a memory operation.
4. Storage register (SR). This is sometimes referred to as the *memory buffer register*, and contains the data used in the memory (to or from) operation.
5. Special use registers—usage depending on instruction.

Data paths connect the output of one register to the input of another, and may include combinatorial logic. The opcode generally defines which of the many data paths are used in execution. The collection of all opcodes defines the data paths required by a specific architecture. A register may be connected to multiple output destination registers and accepts input from one of several source registers in any one cycle. A register output (Figure 1.5) is gated to various destinations with which it can communicate in a single cycle. The activation of a particular data path is done through a *control point*. The activation and definition of every control point in the processor for every cycle of operation is the responsibility of the instruction decoder, which may be implemented directly or as a microprogram storage (Figure 1.6).

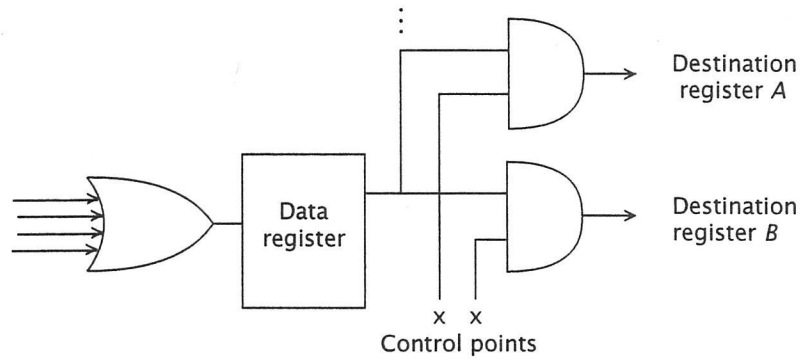


Figure 1.5 Register output gating.

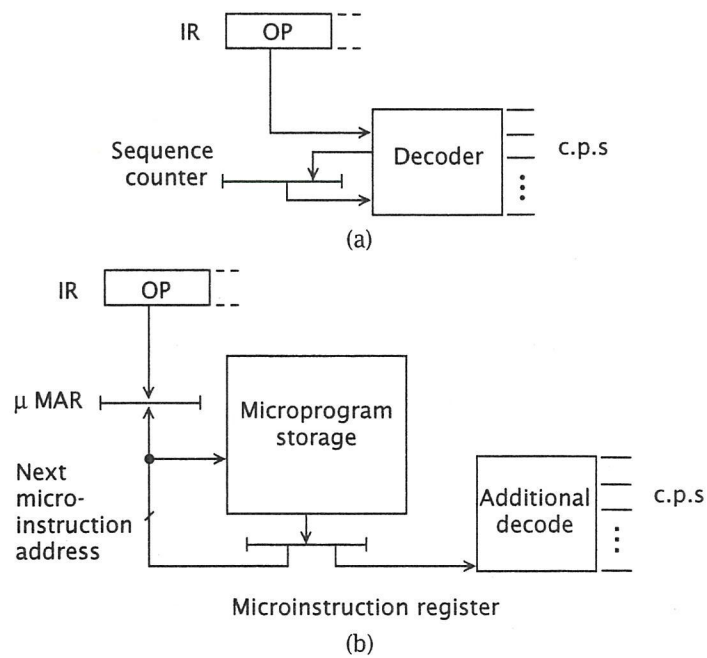


Figure 1.6 (a) Direct decoder. (b) Microprogram decoder.

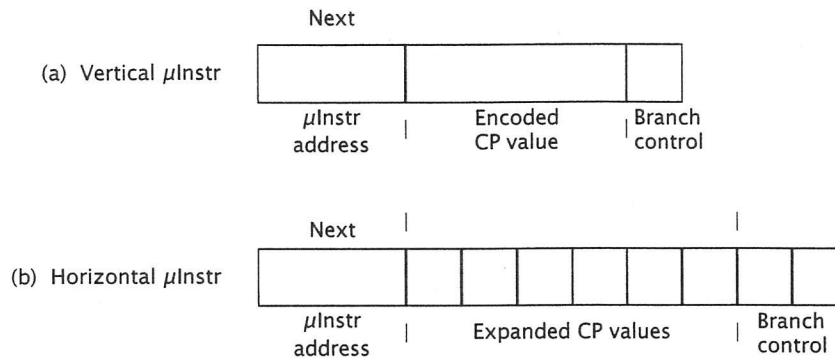


Figure 1.7 Examples of microinstructions.

Whether the decoder that manages interpretation is microprogrammed or directly decoded involves subtle tradeoffs in the host machine design. Direct decoders (Figure 1.6a) are designed by using combinatorial logic (usually PLAs) to represent the various desired control point actions. The logical inputs come from the opcode (the type of instructions to be performed), the sequence counter (a small counter to keep track of which cycle within an instruction's execution is being activated), and some test information from the data registers (e.g., sign of a value) to correctly set the next control action.

Microprogrammed decoders (Figure 1.6b) are designed using ROM (read-only memory), sometimes complemented with RAM (read/write random access memory). The opcode provides an initial address to an entry (microinstruction), which specifies the control point values as well as the expected address of the next microinstruction. During the process of instruction execution, the next microinstruction address may be modified by test information.

In microprogrammed machines [291, 305], the microinstruction defines the control point values required throughout the system as well as controls the sequencing of the interpretation of an operation. Sequencing can be performed either in-line or with the explicit address of the next microinstruction in the sequence. Microinstruction formats typically explicitly include the next microinstruction address. In most machines, the control points are encoded in some fashion in microinstruction representation, making more efficient use of the microprogram storage. Some designs highly encode the control point actions so that only a few actions may be activated in a particular cycle. This type of microinstruction is referred to as a *vertical* microinstruction (Figure 1.7a) and requires an additional decoder to process the encoded control point information. In contrast, other designs partition control point actions into many different classes, any of which can be activated within any given cycle. This type of microinstruction is referred to as a *horizontal* microinstruction (Figure 1.7b).

Table 1.2 provides a qualitative comparison between the two approaches. Direct decoders are usually fast and small, but may complicate the design process, especially for certain operating system functions (security, con-

Table 1.2 Direct or hardwired control vs. microprogrammed control: comparative decoder features.

Attribute	Hardwired Decoders	Microprogrammed Decoders
Speed	Fast	Slower
Chip area efficiency	Uses least area	Uses more area
Ease of change, additions	Somewhat difficult	Easier
Ability to handle large/complex instruction sets	Somewhat difficult	Easier
Ability to support operating systems and diagnostic features	Very difficult (unless anticipated during design)	Easy
Where used	Mostly RISC microprocessors	Mainframes, some microprocessors
Instruction set size	Usually under 100 instructions	Usually over 100 instructions
ROM size	—	2^K – 10^K by 40–200 bit microinstructions

figuration management, etc.) or machine diagnostics. Microprogrammed decoders are usually used in these environments. Note that a slower decoder does not necessarily mean a slower machine, as other paths in the machine may be the determining speed factor.

The cycle time of the machine is determined by the time it takes to move data from one register to another through intervening combinatorial logic. For machines with particularly long or elaborate combinatorial logic in the data paths, it is possible to distinguish short paths from long and to define multiples of the original clock to manage these long paths, as discussed later.

1.3 The Instruction Set

Computers deal with many different kinds of data in many different ways. There are probably as many variations of data representations as there are processor implementations. Additionally, the operations available to perform the requisite data manipulations are often subtly different on different processors and frequently use different assembly syntax. All of these factors make choosing a data and instruction representation system that is generally acceptable difficult at best.

This section aims at providing a basic understanding of the key concepts involved. Both data types and instruction set designs are presented from a “birds-eye view,” and comparisons between some of the major alternatives currently in use are considered to place the choices in perspective.