FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

The McGraw-Hill Companies

ADVANCED COMPUTER ARCHITECTURE

Parallelism, Scalability, Programmability

Second Edition

Kai Hwang

Professor of Electrical Engineering and Computer Science
University of Southern California, USA

Naresh Jotwani

Director, School of Solar Energy
Pandit Deendayal Petroleum University
Gandhinagar, Gujarat

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Part II

Hardware Technologies

Summary

Part II contains three chapters dealing with hardware technologies underlying the development of parallel processing computers. The discussions cover advanced processors, memory hierarchy, and pipelining technologies. These hardware units must work with software, and matching hardware design with program behavior is the main theme of these chapters.

We will study RISC, CISC, scalar, superscalar, VLIW, superpipelined, vector, and symbolic processors. Digital bus, cache design, shared memory, and virtual memory technologies will be considered. Advanced pipelining principles and their applications are described for memory access, instruction execution, arithmetic computation, and vector processing. These chapters are hardware-oriented. Readers whose interest is mainly in software can skip Chapters 5 and 6 after reading Chapter 4.

The material in Chapter 4 presents the functional architectures of processors and memory hierarchy and will be of interest to both computer designers and programmers. After reading Chapter 4, one should have a clear picture of the logical structure of computers. Chapters 5 and 6 describe physical design of buses, cache operations, processor architectures, memory organizations, and their management issues.

The McGraw-Hill Companies

4

Processors and Memory Hierarchy

This chapter presents modern processor technology and the supporting memory hierarchy. We begin with a study of instruction-set architectures including CISC and RISC, and we consider typical superscalar, VLIW, superpipelined, and vector processors. The third section covers memory hierarchy and capacity planning, and the final section introduces virtual memory, address translation mechanisms, and page replacement methods.

Instruction-set processor architectures and logical addressing aspects of the memory hierarchy are emphasized at the functional level. This treatment is directed toward the programmer or computer science major. Detailed hardware designs for bus, cache, and main memory are studied in Chapter 5. Instruction and arithmetic pipelines and superscalar and superpipelined processors are further treated in Chapter 6.

4.1 ADVANCED PROCESSOR TECHNOLOGY

Architectural families of modern processors are introduced below, from processors used in workstations or multiprocessors to those designed for mainframes and supercomputers.

Major processor families to be studied include the *CISC*, *RISC*, *superscalar*, *VLIW*, *superpipelined*, *vector*, and *symbolic* processors. Scalar and vector processors are for numerical computations. Symbolic processors have been developed for AI applications.

4.1.1 Design Space of Processors

Various processor families can be mapped onto a coordinated space of clock *rate* versus *cycles per instruction* (CPI), as illustrated in Fig. 4.1. As implementation technology evolves rapidly, the clock rates of various processors have moved from low to higher speeds toward the right of the design space. Another trend is that processor manufacturers have been trying to lower the CPI rate using innovative hardware approaches.

Based on these trends, the mapping of processors in Fig. 4.1 reflects their implementation during the past decade or so.

Figure 4.1 shows the broad CPI versus clock speed characteristics of major categories of current processors. The two broad categories which we shall discuss are CISC and RISC. In the former category, at present there is the only one dominant presence—the x86 processor architecture; in the latter category, there are several examples, e.g. Power series, SPARC, MIPS, etc.

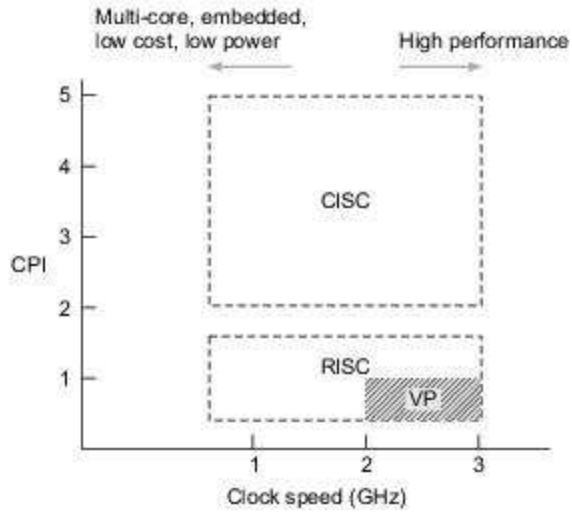


Fig. 4.1 CPI versus processor clock speed of major categories of processors

Under both CISC and RISC categories, products designed for multi-core chips, embedded applications, or for low cost and/or low power consumption, tend to have lower clock speeds. High performance processors must necessarily be designed to operate at high clock speeds. The category of vector processors has been marked VP; vector processing features may be associated with CISC or RISC main processors.

The Design Space Conventional processors like the Intel Pentium, M68040, older VAX/8600, IBM 390, etc. fall into the family known as *complex-instruction-set computing* (CISC) architecture. With advanced implementation techniques, the clock rate of today's CISC processors ranges up to a few GHz. The CPI of different CISC instructions varies from 1 to 20. Therefore, CISC processors are at the upper part of the design space.

Reduced-instruction-set computing (RISC) processors include SPARC, Power series, MIPS, Alpha, ARM, etc. With the use of efficient pipelines, the average CPI of RISC instructions has been reduced to between one and two cycles.

An important subclass of RISC processors are the *superscalar processors*, which allow multiple instructions to be issued simultaneously during each cycle. Thus the effective CPI of a superscalar processor should be lower than that of a scalar RISC processor. The clock rate of superscalar processors matches that of scalar RISC processors.

The *very long instruction word* (VLIW) architecture can in theory use even more functional units than a superscalar processor. Thus the CPI of a VLIW processor can be further lowered. Intel's i860 RISC processor had VLIW architecture.

The processors in *vector supercomputers* use multiple functional units for concurrent scalar and vector operations.

The effective CPI of a processor used in a supercomputer should be very low, positioned at the lower right corner of the design space. However, the cost and power consumption increase appreciably if processor design is restricted to the lower right corner. Some key issues impacting modern processor design will be discussed in Chapter 13.

Instruction Pipelines The execution cycle of a typical instruction includes four phases: *fetch*, *decode*, *execute*, and *write-back*. These instruction phases are often executed by an *instruction pipeline* as demonstrated in Fig. 4.2a. In other words, we can simply model an *instruction processor* by such a pipeline structure.

For the time being, we will use an abstract pipeline model for an intuitive explanation of various processor classes. The *pipeline*, like an industrial assembly line, receives successive instructions from its input end and executes them in a streamlined, overlapped fashion as they flow through.

A *pipeline cycle* is intuitively defined as the time required for each phase to complete its operation, assuming equal delay in all phases (pipeline stages). Introduced below are the basic definitions associated with instruction pipeline operations:

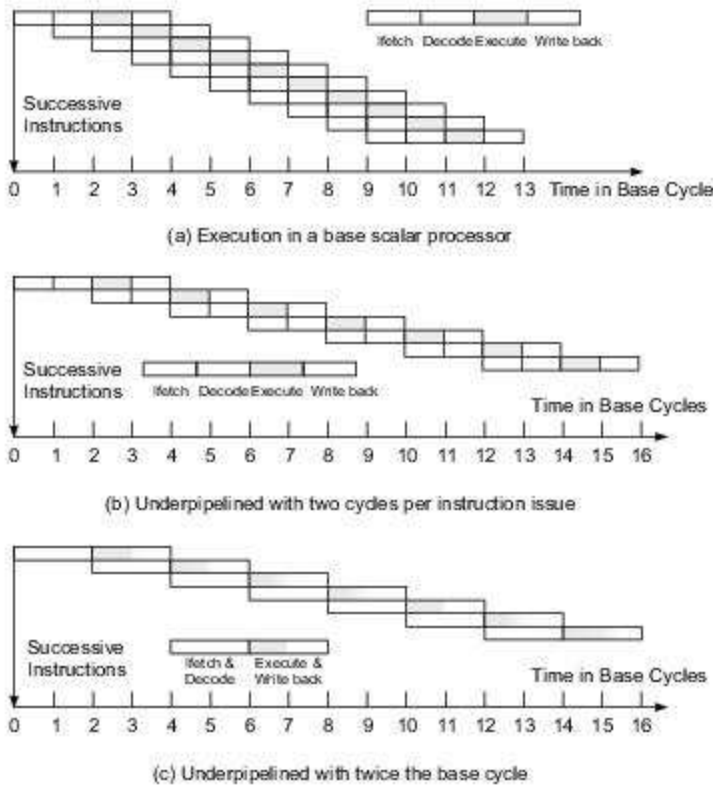


Fig. 4.2 Pipelined execution of successive instructions in a base scalar processor and in two underpipelined cases (Courtesy of Jouppi and Wall; reprinted from *Proc. ASPLOS*, ACM Press, 1989)

(1) *Instruction pipeline cycle*—the clock period of the instruction pipeline.
(2) *Instruction issue latency*—the time (in cycles) required between the issuing of two adjacent instructions.
(3) *Instruction issue rate*—the number of instructions issued per cycle, also called the *degree* of a superscalar processor.

(4) *Simple operation latency*—Simple operations make up the vast majority of instructions executed by the machine, such as *integer adds, loads, stores, branches, moves*, etc. On the contrary, complex operations are those requiring an order-of-magnitude longer latency, such as *divides, cache misses*, etc. These latencies are measured in number of cycles.

(5) *Resource conflicts*—This refers to the situation where two or more instructions demand use of the same functional unit at the same time.

A *base scalar processor* is defined as a machine with one instruction issued per cycle, a *one*-cycle latency for a simple operation, and a *one*-cycle latency between instruction issues. The instruction pipeline can be fully utilized if successive instructions can enter it continuously at the rate of one per cycle, as shown in Fig. 4.2a.

However, the instruction issue latency can be more than one cycle for various reasons (to be discussed in Chapter 6). For example, if the instruction issue latency is two cycles per instruction, the pipeline can be underutilized, as demonstrated in Fig. 4.2b.

Another underpipelined situation is shown in Fig. 4.2c, in which the pipeline cycle time is doubled by combining pipeline stages. In this case, the *fetch* and *decode* phases are combined into one pipeline stage, and *execute* and *write-back* are combined into another stage. This will also result in poor pipeline utilization.

The effective CPI rating is 1 for the ideal pipeline in Fig. 4.2a, and 2 for the case in Fig. 4.2b. In Fig. 4.2c, the clock rate of the pipeline has been lowered by one-half. According to Eq. 1.3, either the case in Fig. 4.2b or that in Fig. 4.2c will reduce the performance by one-half, compared with the ideal case (Fig. 4.2a) for the base machine.

Figure 4.3 shows the data path architecture and control unit of a typical, simple scalar processor which does not employ an instruction pipeline. Main memory, I/O controllers, etc. are connected to the external bus.

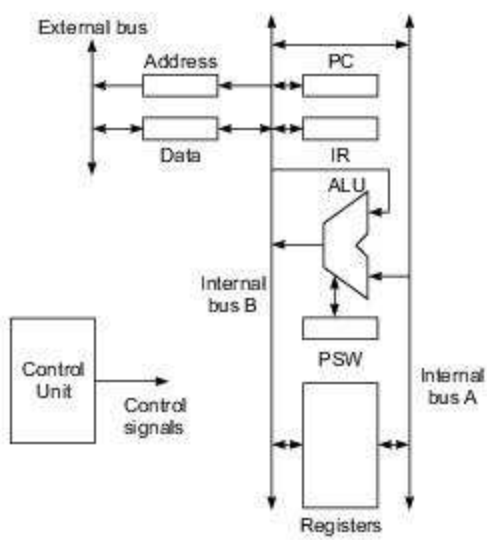


Fig. 4.3 Data path architecture and control unit of a scalar processor

The control unit generates control signals required for the *fetch, decode, ALU operation, memory access*, and *write result* phases of instruction execution. The control unit itself may employ hardwired logic, or—as

was more common in older CISC style processors—microcoded logic. Modern RISC processors employ hardwired logic, and even modern CISC processors make use of many of the techniques originally developed for high-performance RISC processors[1].

4.1.2 Instruction-Set Architectures

In this section, we characterize computer instruction sets and examine hardware features built into generic RISC and CISC scalar processors. Distinctions between them are revealed. The boundary between RISC and CISC architectures has become blurred in recent years. Quite a few processors are now built with hybrid RISC and CISC features based on the same technology. However, the distinction is still rather sharp in instruction-set architectures.

The instruction set of a computer specifies the primitive commands or machine instructions that a programmer can use in programming the machine. The complexity of an instruction set is attributed to the instruction formats, data formats, addressing modes, general-purpose registers, opcode specifications, and flow control mechanisms used. Based on past experience in processor design, two schools of thought on instruction-set architectures have evolved, namely, CISC and RISC.

Complex Instruction Sets In the early days of computer history, most computer families started with an instruction set which was rather simple. The main reason for being simple then was the high cost of hardware. The hardware cost has dropped and the software cost has gone up steadily in the past decades. Furthermore, the semantic gap between HLL features and computer architecture has widened.

The net result at one stage was that more and more functions were built into the hardware, making the instruction set large and complex. The growth of instruction sets was also encouraged by the popularity of microprogrammed control in the 1960s and 1970s. Even user-defined instruction sets were implemented using microcodes in some processors for special-purpose applications.

A typical CISC instruction set contains approximately 120 to 350 instructions using variable instruction/data formats, uses a small set of 8 to 24 *general-purpose registers* (GPRs), and executes a large number of memory reference operations based on more than a dozen addressing modes. Many HLL statements are directly implemented in hardware/firmware in a CISC architecture. This may simplify the compiler development, improve execution efficiency, and allow an extension from scalar instructions to vector and symbolic instructions.

Reduced Instruction Sets After two decades of using CISC processors, computer designers began to reevaluate the performance relationship between instruction-set architecture and available hardware/software technology.

Through many years of program tracing, computer scientists realized that only 25% of the instructions of a complex instruction set are frequently used about 95% of the time. This implies that about 75% of hardware-supported instructions often are not used at all. A natural question then popped up: Why should we waste valuable chip area for rarely used instructions?

With low-frequency elaborate instructions demanding long microcodes to execute them, it might be more advantageous to remove them completely from the hardware and rely on software to implement them. Even if the software implementation was slow, the net result would be still a plus due to their low frequency of appearance. Pushing rarely used instructions into software would vacate chip areas for building more

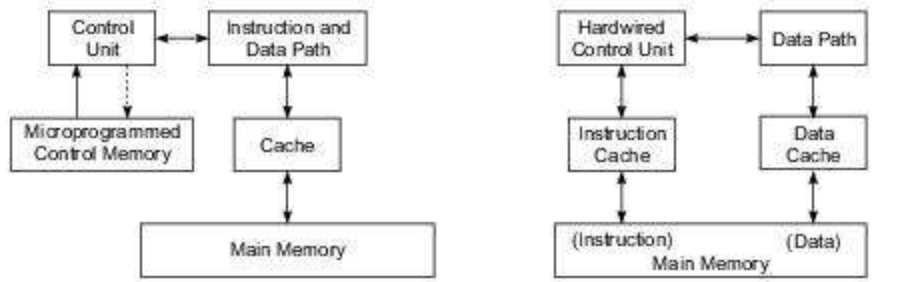
[1]Fuller discussion of these basic architectural concepts can be found in *Computer System Organisation*, by Naresh Jotwani, Tata McGraw-Hill, 2009.

powerful RISC or superscalar processors, even with on-chip caches or floating-point units, and hardwired control would allow faster clock rates.

A RISC instruction set typically contains less than 100 instructions with a fixed instruction format (32 bits). Only three to five simple addressing modes are used. Most instructions are register-based. Memory access is done by load/store instructions only . A large register file (at least 32) is used to improve fast context switching among multiple users, and most instructions execute in one cycle with hardwired control.

The resulting benefits include a higher clock rate and a lower CPI, which lead to higher processor performance.

Architectural Distinctions Hardware features built into CISC and RISC processors are compared below. Figure 4.4 shows the architectural distinctions between traditional CISC and RISC. Some of the distinctions have since disappeared, however, because processors are now designed with features from both types.



(a) The CISC architecture with microprogrammed control and unified cache

(b) The RISC architecture with hardwired control and split instruction cache and data cache

Fig. 4.4 Distinctions between typical RISC and typical CISC processor architectures (Courtesy of Gordon Bell, 1989)

Conventional CISC architecture uses a unified cache for holding both instructions and data. Therefore, they must share the same data/instruction path. In a RISC processor, separate *instruction* and *data caches* are used with different access paths. However, exceptions do exist. In other words, CISC processors may also use split cache.

The use of microprogrammed control was found in traditional CISC, and hardwired control in most RISC. Thus control memory (ROM) was needed in earlier CISC processors, which slowed down the instruction execution. However, modern CISC also uses hardwired control. Therefore, split caches and hardwired control are not today exclusive in RISC machines.

Using hardwired control reduces the CPI effectively to one instruction per cycle if pipelining is carried out perfectly. Some CISC processors also use split caches and hardwired control, such as the MC68040 and i586.

In Table 4.1, we compare the main features of typical RISC and CISC processors. The comparison involves five areas: instruction sets, *addressing modes, register file and cache design, expected CPI*, and *control mechanisms*. Clock rates of modern CISC and RISC processors are comparable.

The large number of instructions used in a CISC processor is the result of using variable-format instructions–integer, floating-point, and vector data–and of using over a dozen different addressing modes. Furthermore, with few GPRs, many more instructions access the memory for operands. The CPI is thus high as a result of the long microcodes used to control the execution of some complex instructions.

On the other hand, most RISC processors use 32-bit instructions which are predominantly register-based. With few simple addressing modes, the memory-access cycle is broken into pipelined access operations involving the use of caches and working registers. Using a large register file and separate I-and D-caches benefits internal data forwarding and eliminates unnecessary storage of intermediate results. With hardwired control, the CPI is reduced to 1 for most RISC instructions. Most recently introduced processor families have infact been based on RISC architecture.

Table 4.1 *Characteristics of Typical CISC and RISC Architectures*

Architectural Characteristic	*Complex Instruction Set Computer (CISC)*	*Reduced Instruction Set Computer (RISC)*
Instruction-set size and instruction formats	Large set of instructions with variable formats (16–64 bits per instruction).	Small set of instructions with fixed (32-bit) format and most register-based instructions.
Addressing modes	12–24.	Limited to 3–5.
General-purpose registers and cache design	8–24 GPRs, originally with a unified cache for instructions and data, recent designs also use split caches.	Large numbers (32–192) of GPRs with mostly split data cache and instruction cache.
CPI	CPI between 2 and 15.	One cycle for almost all instructions and an average CPI < 1.5.
CPU Control	Earlier microcoded using control memory (ROM), but modern CISC also uses hardwired control.	Hardwired without control memory.

4.1.3 CISC Scalar Processors

A scalar processor executes with scalar data. The simplest scalar processor executes integer instructions using fixed-point operands. More capable scalar processors execute both integer and floating-point operations. A modern scalar processor may possess both an integer unit and a floating-point unit, or even multiple such units. Based on a complex instruction set, a *CISC scalar processor* can also use pipelined design.

However, the processor is often underpipelined as in the two cases shown in Figs. 4.2b and 4.2c. Major causes of the underpipelined situations (Figs. 4.2b) include data dependence among instructions, resource conflicts, branch penalties, and logic hazards which will be studied in Chapter 6, and further in Chapter 12.

The case in Fig. 4.2c is caused by using a clock cycle which is greater than the simple operation latency. In subsequent sections, we will show how RISC and superscalar techniques can be applied to improve pipeline performance.

Representative CISC Processors In Table 4.2, three early representative CISC scalar processors are listed. The VAX 8600 processor was built on a PC board. The i486 and M68040 were single-chip microprocessors. These two processor families are still in use at present. We use these popular architectures to explain some interesting features built into CISC processors. In any processor design, the designer attempts to achieve higher throughput in the processor pipelines.

Both hardware and software mechanisms have been developed to achieve these goals. Due to the complexity involved in a CISC processor, the most difficult task for a designer is to shorten the clock cycle to match the simple operation latency. This problem is easier to overcome with a RISC architecture.

Example 4.1 The Digital Equipment VAX 8600 processor architecture

The VAX 8600 was introduced by Digital Equipment Corporation in 1985. This machine implemented a typical CISC architecture with microprogrammed control. The instruction set contained about 300 instructions with 20 different addressing modes. As shown in Fig. 4.5, the VAX 8600 executed the same instruction set, ran the same VMS operating system, and interfaced with the same I/O buses (such as SBI and Unibus) as the VAX 11/780.

The CPU in the VAX 8600 consisted of two functional units for concurrent execution of integer and floating-point instructions. The unified cache was used for holding both instructions and data. There were 16 GPRs in the instruction unit. Instruction pipelining was built with six stages in the VAX 8600, as in most else machines. The instruction unit prefetched and decoded instructions, handled branching operations, and supplied operands to the two functional units in a pipelined fashion.

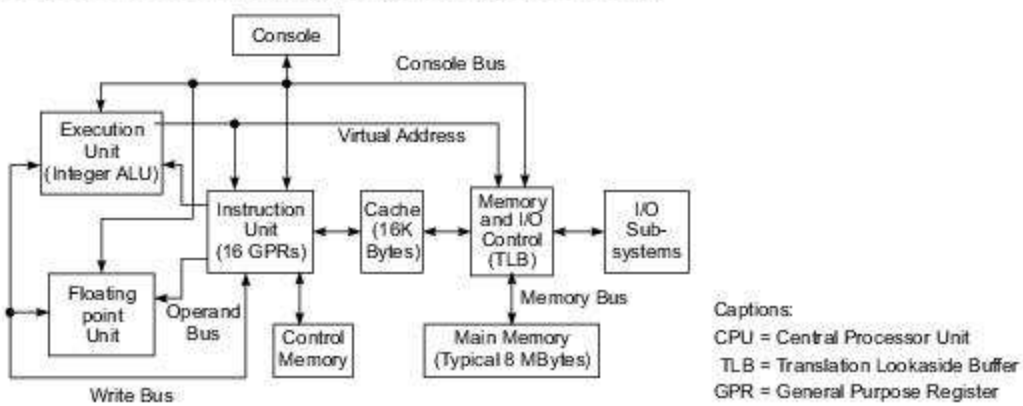


Fig. 4.5 The VAX 8600 CPU, a typical CISC processor architecture (Courtesy of Digital Equipment Corporation, 1985)

A *translation lookaside buffer* (TLB) was used in the memory control unit for fast generation of a physical address from a virtual address. Both integer and floating-point units were pipelined. The performance of the processor pipelines relied heavily on the cache hit ratio and on minimal branching damage to the pipeline flow.

The CPI of a VAX 8600 instruction varied within a wide range from 2 cycles to as high as 20 cycles. For example, both *multiply* and *divide* might tie up the execution unit for a large number of cycles. This was caused by the use of long sequences of microinstructions to control hardware operations.

The general philosophy of designing a CISC processor is to implement useful instructions in hardware/firmware which may result in a shorter program length with a lower software overhead. However, this advantage can only be obtained at the expense of a lower clock rate and a higher CPI, which may not pay off at all.

The VAX 8600 was improved from the earlier VAX/11 Series. The system was later further upgraded to the VAX 9000 Series offering both vector hardware and multiprocessor options. All the VAX Series have used a paging technique to allocate the physical memory to user programs.

CISC Microprocessor Families In 1971, the Intel 4004 appeared as the first microprocessor based on a 4-bit ALU. Since then, Intel has produced the 8-bit 8008, 8080, and 8085. Intel's 16-bit processors appeared in 1978 as the 8086, 8088, 80186, and 80286. In 1985, the 80386 appeared as a 32-bit machine. The 80486 and Pentium are the latest 32-bit processors in the Intel 80x86 family.

Motorola produced its first 8-bit microprocessor, the MC6800, in 1974, then moved to the 16-bit 68000 in 1979, and then to the 32-bit 68020 in 1984. Then came the MC68030 and MC68040 in the Motorola MC680x0 family. National Semiconductor's 32-bit microprocessor NS32532 was introduced in 1988. These CISC microprocessor families have been widely used in the *personal computer* (PC) industry, with Intel x86 family dominating.

Over the last two decades, the parallel computer industry has built systems with a large number of open-architecture microprocessors. Both CISC and RISC microprocessors have been employed in these systems. One thing worthy of mention is the compatibility of new models with the old ones in each of the families. This makes it easier to port software along the series of models.

Table 4.2 lists three typical CISC processors of the year 1990[2].

Table 4.2 *Representative CISC Scalar Processors of year 1990*

Feature	*Intel i486*	*Motorola MC68040*	*NS 32532*
Instruction-set size and word length	157 instructions, 32 bits.	113 instructions, 32 bits.	63 instructions, 32 bits.
Addressing modes	12	18	9
Integer unit and GPRs	32-bit ALU with 8 registers.	32-bit ALU with 16 registers.	32-bit ALU with 8 registers.
On-chip cache(s) and MMUs	8-KB unified cache for both code and data. with separate MMUs.	4-KB code cache 4-KB data cache	512-B code cache 1-KB data cache.
Floating-point unit, registers, and function units	On-chip with 8 FP registers adder, multiplier, shifter.	On-chip with 3 pipeline stages, 8 80-bit FP registers.	Off-chip FPU NS 32381, or WTL 3164.
Pipeline stages	5	6	4
Protection levels	4	2	2
Memory organization and TLB/ATC entries	Segmented paging with 4 KB/page and 32 entries in TLB.	Paging with 4 or 8 KB/page, 64 entries in each ATC.	Paging with 4 KB/page, 64 entries.
Technology, clock rate, packaging, and year introduced	CHMOS IV, 25 MHz, 33 MHz, 1.2M transistors, 168 pins, 1989.	0.8–μm HCMOS, 1.2 M transistors, 20 MHz, 40 MHz, 179 pins, 1990.	1.25–μm CMOS 370K transistors, 30 MHz, 175 pins, 1987.
Claimed performance	24 MIPS at 25 MHz,	20 MIPS at 25 MHz, 30 MIPS at 60 MHz.	15 MIPS at 30 MHz.

[2] Motorola microprocessors are at presently built and marked by the divested company Freescale.

Example 4.2 The Motorola MC68040 microprocessor architecture

The MC68040 is a 0.8-μm HCMOS microprocessor containing more than 1.2 million transistors, comparable to the i80486. Figure 4.6 shows the MC68040 architecture. The processor implements over 100 instructions using 16 general-purpose registers, a 4-Kbyte data cache, and a 4-Kbyte instruction cache, with separate *memory management units* (MMUs) supported by an *address translation cache* (ATC), equivalent to the TLB used in other systems. The data formats range from 8 to 80 bits, with provision for the IEEE floating-point standard.

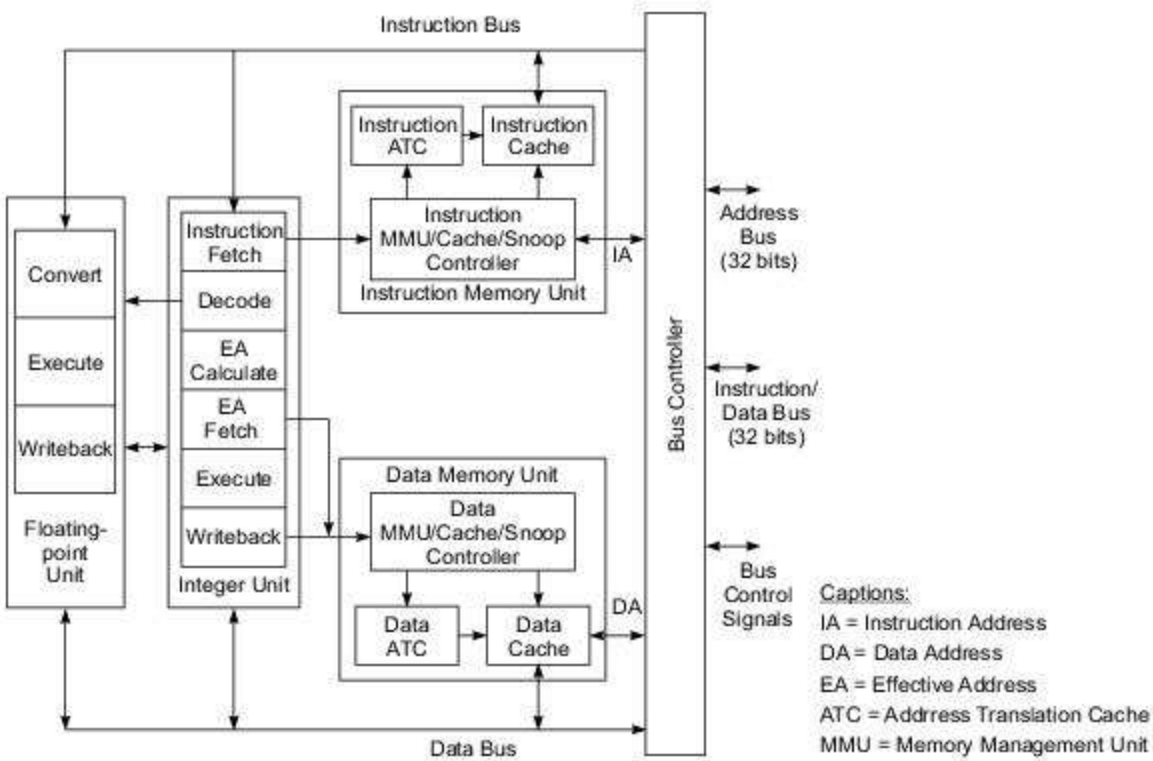


Fig. 4.6 Architecture of the MC68040 processor (Courtesy of Motorola Inc., 1991)

Eighteen addressing modes are supported, including register direct and indirect, indexing, memory indirect, program counter indirect, absolute, and immediate modes. The instruction set includes data movement, integer, BCD, and floating point arithmetic, logical, shifting, bit-field manipulation, cache maintenance, and multiprocessor communications, in addition to program and system control and memory management instructions.

The integer unit is organized in a six-stage instruction pipeline. The floating-point unit consists of three pipeline stages (details to be studied in Section 6.4.1). All instructions are decoded by the integer unit. Floating-point instructions are forwarded to the floating-point unit for execution.

Separate instruction and data buses are used to and from the instruction and data memory units, respectively. Dual MMUs allow interleaved fetch of instructions and data from the main memory. Both the address bus and the data bus are 32 bits wide.

Three simultaneous memory requests can be generated by the dual MMUs, including data operand read and write and instruction pipeline refill. Snooping logic is built into the memory units for monitoring bus events for cache invalidation.

The complete memory management is provided with support for virtual demand paged operating system. Each of the two ATCs has 64 entries providing fast translation from virtual address to physical address. With the CISC complexity involved, the M68040 does not provide delayed branch hardware support, which is often found in RISC processors like Motorola's own M88100 microprocessor.

4.1.4 RISC Scalar Processors

Generic RISC processors are called *scalar* RISC because they are designed to issue one instruction per cycle, similar to the base scalar processor shown in Fig. 4.2a. In theory, both RISC and CISC scalar processors should perform about the same if they run with the same clock rate and with equal program length. However, these two assumptions are not always valid, as the architecture affects the quality and density of code generated by compilers.

The RISC design gains its power by pushing some of the less frequently used operations into software. The reliance on a good compiler is much more demanding in a RISC processor than in a CISC processor. Instruction-level parallelism is exploited by pipelining in both processor architectures.

Without a high clock rate, a low CPI, and good compilation support, neither CISC nor RISC can perform well as designed. The simplicity introduced with a RISC processor may lead to the ideal performance of the base scalar machine modeled in Fig. 4.2a.

Representative RISC Processors Four representative RISC-based processors from the year 1990, the Sun SPARC, Intel i860, Motorola M88100, and AMD 29000, are summarized in Table 4.3. All of these processors use 32-bit instructions. The instruction sets consist of 51 to 124 basic instructions. On-chip floating-point units are built into the i860 and M88100, while the SPARC and AMD use off-chip floating-point units. We consider these four processors as generic scalar RISC, issuing essentially only one instruction per pipeline cycle.

Among the four scalar RISC processors, we choose to examine the Sun SPARC and i860 architectures below. SPARC stands for *scalable processor architecture*. The scalability of the SPARC architecture refers to the use of a different number of *register windows* in different SPARC implementations.

This is different from the M88100, where scalability refers to the number of *special function units* (SFUs) implementable on different versions of the M88000 processor. The Sun SPARC is derived from the original Berkeley RISC design.

Table 4.3 *Representative RISC Scalar Processors of year 1990*

Feature	*Sun SPARC CY7C601*	*Intel i860*	*Motorola M 88100*	*AMD 29000*
Instruction set, formats, addressing modes.	69 instructions, 32-bit format, 7 data types, 4-stage instr. pipeline.	82 instructions, 32-bit format, 4 addressing modes.	51 instructions, 7 data types, 3 instr. formats, 4 addressing modes.	112 instructions, 32-bit format, all registers indirect addressing.
Integer unit, GPRs.	32-bit RISC/IU, 136 registers divided into 8 windows.	32-bit RISC core, 32 registers.	32-bit IU with 32 GPRs and scoreboarding.	32-bit IU with 192 registers without windows.
Caches(s), MMU, and memory organization.	Off-chip cache/MMU on CY7C604 with 64-entry TLB.	4-KB code, 8-KB data, on-chip' MMU, paging with 4 KB/page.	Off-chip M88200 caches/MMUs, segmented paging, 16-KB cache.	On-chip MMU with 32-entry TLB, with 4-word prefetch buffer and 512-B branch target cache.
Floating-point unit registers and functions	Off-chip FPU on CY7C602, 32 registers, 64-bit pipeline (equiv. to TI8848).	On-chip 64-bit FP multiplier and FP adder with 32 FP registers, 3-D graphics unit.	On-chip FPU adder, multiplier with 32 FP registers and 64-bit arithmetic.	Off-chip FPU on AMD 29027, on-chip FPU withAMD 29050.
Operation modes	Concurrent IU and FPU operations.	Allow dual instructions and dual FP operations.	Concurrent IU, FPU and memory access with delayed branch.	4-stage pipeline processor.
Technology, clock rate, packaging, and year	0.8-μm CMOS IV,33 MHz, 207 pins, 1989.	1-μm CHMOS IV, over 1M transistors, 40 MHz, 168 pins, 1989	1-μm HCMOS, 1.2M transistors, 20 MHz, 180 pins, 1988.	1.2-μm CMOS, 30 MHz, 40 MHz, 169 pins, 1988.
Claimed performance	24 MIPS for 33 MHz version, 50 MIPS for 80 MHz ECL version. Up to 32 register windows can be built.	40 MIPS and 60 Mflops for 40 MHz, i860/XP announced in 1992 with 2.5M transistors.	17 MIPS and 6 Mflops at 20 MHz, up to 7 special function units could be configured.	27 MIPS at 40 MHz, new version AMD 29050 at 55 MHz in 1990.

Example 4.3 The Sun Microsystems SPARC architecture

The SPARC has been implemented by a number of licensed manufacturers as summarized in Table 4.4. Different technologies and window numbers are used by different SPARC manufacturers. Data presented is from around the year 1990.

Table 4.4 *SPARC Implementations by Licensed Manufacturers (1990)*

SPARC Chip	*Technology*	*Clock Rate (MHz)*	*Claimed VAX MIPS*	*Remarks*
Cypress CY7C601 IU	O.8μm CMOS IV, 207 pins.	33	24	CY7C602 FPU with 4.5 Mflops DP Linpack, CY7C604 Cache/MMC, CY7C157 Cache.
Fujitsu MB 86901 IU	1.2-μm CMOS, 179 pins.	25	15	MB 86911 FPC FPC and TI 8847 FPP, MB86920 MMU, 2.7 Mflops DP Linpack by FPU.
LSI Logic L64811	1.0-μm HCMOS, 179 pins.	33	20	L64814 FPU, L64815 MMU.
TI 8846	0.8-μm CMOS	33	24	42 Mflops DP Linpack on TI-8847 FPP.
BIT IU B-3100	ECL family.	80	50	15 Mflops DP Linpack on FPUs: B-3120 ALU, B-3611 FP Multiply/Divide.

At the time, all of these manufacturers implemented the *floating-point unit* (FPU) on a separate coprocessor chip. The SPARC processor architecture contains essentially a RISC *integer unit* (IU) implemented with 2 to 32 register windows.

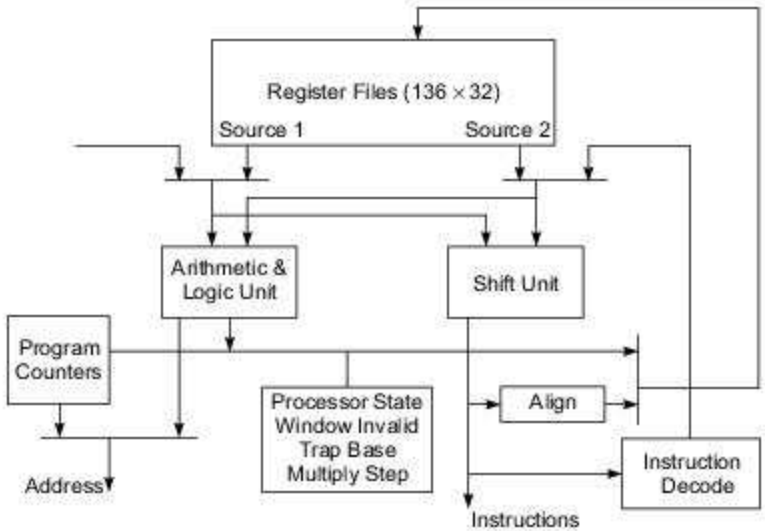
We choose to study the SPARC family chips produced by Cypress Semiconductors, Inc. Figure 4.7 shows the architecture of the Cypress CY7C601 SPARC processor and of the CY7C602 FPU. The Sun SPARC instruction set contains 69 basic instructions, a significant increase from the 39 instructions in the original Berkeley RISCII instruction set.

The SPARC runs each procedure with a set of thirty-two 32-bit IU registers. Eight of these registers are *global registers* shared by all procedures, and the remaining 24 are *window registers* associated with only each procedure. The concept of using overlapped register windows is the most important feature introduced by the Berkeley RISC architecture.

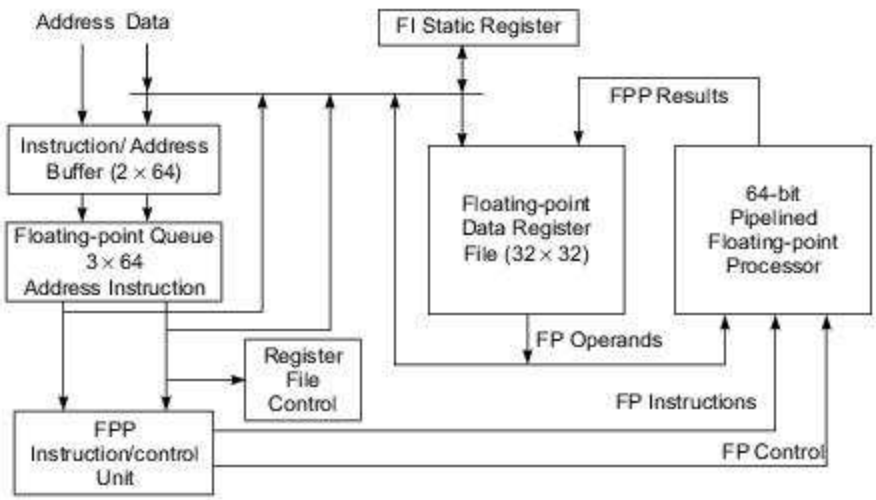
The concept is illustrated in Fig. 4.8 for eight overlapping windows (formed with 64 local registers and 64 overlapped registers) and eight globals with a total of 136 registers, as implemented in the Cypress 601.

Each register window is divided into three eight-register sections, labeled *Ins*, *Locals*, and *Outs*. The local registers are only locally addressable by each procedure. The Ins and Outs are shared among procedures.

The calling procedure passes parameters to the called procedure via its Outs (r8 to r15) registers, which are the Ins registers of the called procedure. The window of the currently running procedure is called the active window pointed to by a current window pointer. A window invalid mask is used to indicate which window is invalid. The trap base register serves as a pointer to a trap handler.



(a) The Cypress CY7C601 SPARC processor

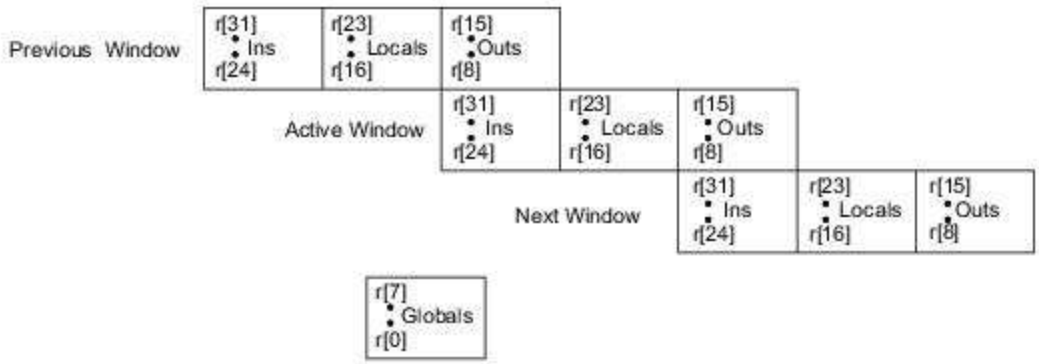


(b) The Cypress CY7C602 floating-point unit

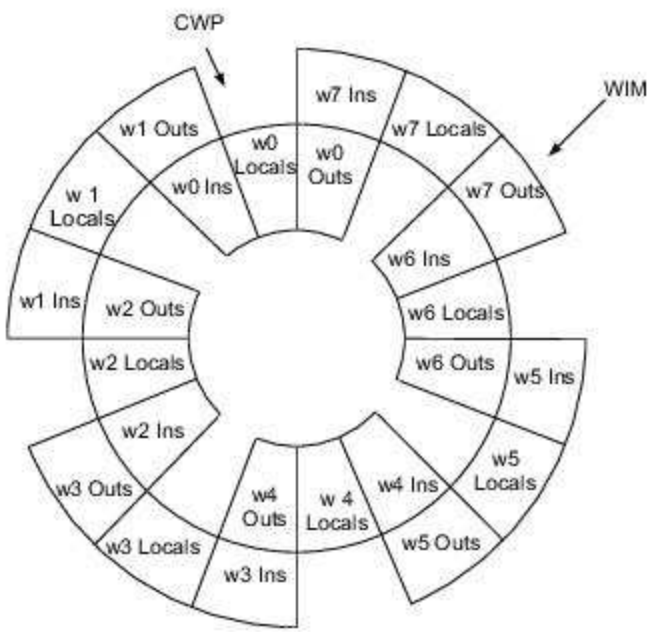
Fig. 4.7 The SPARC architecture with the processor and the floating-point unit on two separate chips (Courtesy of Cypress Semiconductor Co., 1991)

A special register is used to create a 64-bit product in multiple step instructions. Procedures can also be called without changing the window. The overlapping windows can significantly save the time required for interprocedure communications, resulting in much faster context switching among cooperative procedures.

The FPU features 32 single-precision (32-bit) or 16 double-precision (64-bit) floating-point registers (Fig. 4.7b). Fourteen of the 69 SPARC instructions are for floating-point operations. The SPARC architecture implements three basic instruction formats, all using a single word length of 32 bits.



(a) Three overlapping register windows and the globals registers



(b) Eight register windows forming a circular stack

Fig. 4.8 The concept of overlapping register windows in the SPARC architecture (Courtesy of Sun Microsystems, Inc., 1987)

Table 4.4 shows the MIPS rate relative to that of the VAX 11/780, which has been used as a reference machine with 1 MIPS. The 50-MIPS rate is the result of ECL implementation with a 80-MHz clock. A GaAs SPARC was reported to yield a 200-MIPS peak at 200-MHz clock rate.

Example 4.4 The Intel i860 processor architecture

In 1989, Intel Corporation introduced the i860 microprocessor. It was a 64-bit RISC processor fabricated on a single chip containing more than 1 million transistors. The peak performance of the i860 was designed to reach 80 Mflops single-precision or 60 Mflops double-precision, or 40 MIPS in 32-bit integer operations at a 40-MHz clock rate.

A schematic block diagram of major components in the i860 is shown in Fig. 4.9. There were nine functional units (shown in nine boxes) interconnected by multiple data paths with widths ranging from 32 to 128 bits.

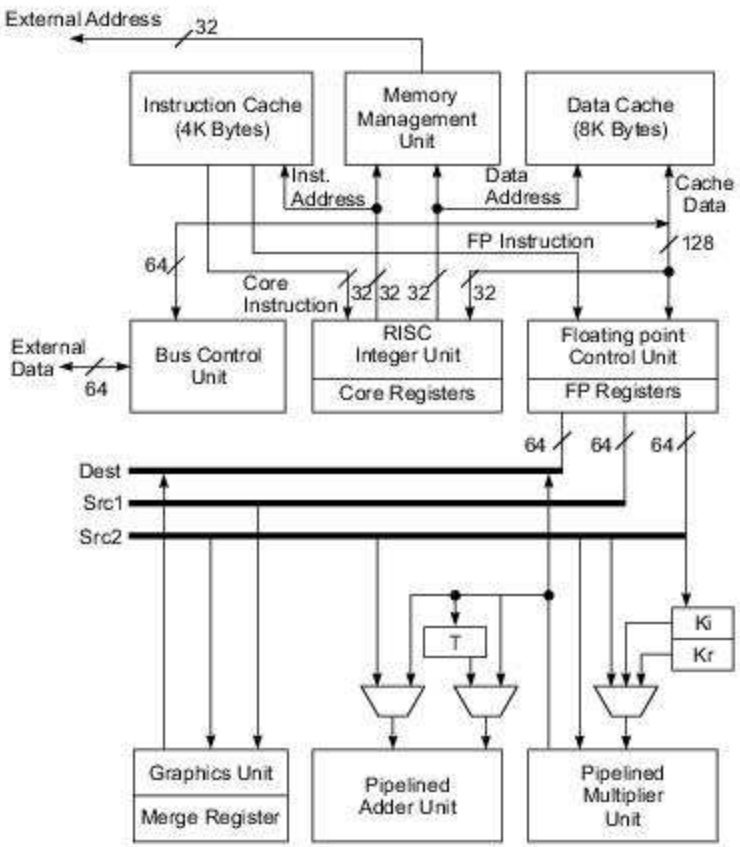


Fig. 4.9 Functional units and data paths of the Intel i860 RISC microprocessor (Courtesy of Intel Corporation, 1990)

All external or internal address buses were 32-bit wide, and the external data path or internal data bus was 64 bits wide. However, the internal RISC integer ALU was only 32 bits wide. The instruction cache had 4 Kbytes organized as a two-way set-associative memory with 32 bytes per cache block. It transferred 64 bits per clock cycle, equivalent to 320 Mbytes/s at 40 MHz.

The data cache was a two-way set-associative memory of 8 Kbytes. It transferred 128 bits per clock cycle (640 Mbytes/s) at 40 MHz. A write-back policy was used. Cacheing could be inhibited by software, if needed. The bus control unit coordinated the 64-bit data transfer between the chip and the outside world.

The MMU implemented protected 4 Kbyte paged virtual memory of 2^{32} bytes via a TLB. The paging and MMU structure of the i860 was identical to that implemented in the i486. An i860 and an i486 could be used jointly in a heterogeneous multiprocessor system, permitting the development of compatible OS kernels. The RISC integer unit executed *load, store, integer, bit,* and *control* instructions and fetched instructions for the floating-point control unit as well.

There were two floating-point units, namely, the *multiplier unit* and the *adder unit*, which could be used separately or simultaneously under the coordination of the floating-point control unit. Special dual-operation floating-point instructions such as *add-and-multiply* and *subtract-and-multiply* used both the multiplier and adder units in parallel (Fig. 4.10).

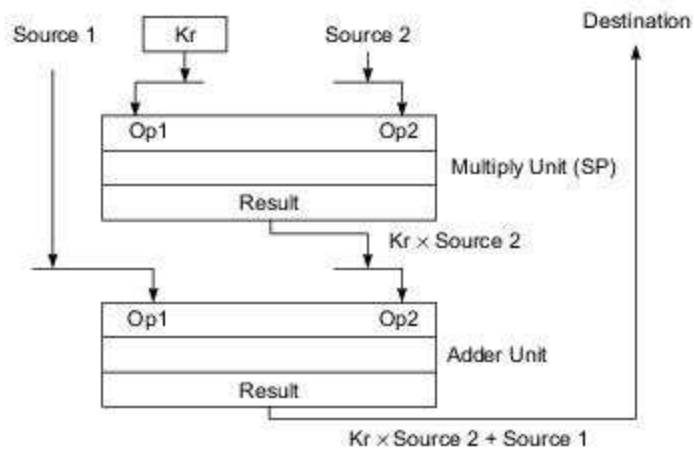


Fig. 4.10 Dual floating-point operations in the i860 processor

Furthermore, both the integer unit and the floating-point control unit could execute concurrently. In this sense, the i860 was also a superscalar RISC processor capable of executing two instructions, one integer and one floating-point, at the same time. The floating-point unit conformed to the IEEE 754 floating-point standard, operating with single-precision (32-bit) and double-precision (64-bit) operands.

The graphics unit executed integer operations corresponding to 8-, 16-, or 32-bit pixel data types. This unit supported three-dimensional drawing in a graphics frame buffer, with color intensity, shading, and hidden surface elimination. The merge register was used only by vector integer instructions. This register accumulated the results of multiple addition operations.

The i860 executed 82 instructions, including 42 RISC integer, 24 floating-point, 10 graphics, and 6 assembler pseudo-operations. All the instructions executed in one cycle, i.e. 25 ns for a 40-MHz clock rate. The i860 and its successor, the i860XP, were used in floating-point accelerators, graphics subsystems, workstations, multiprocessors, and multicomputers. However, due to the market dominance of Intel's own x86 family, the i860 was subsequently withdrawn from production.

The RISC Impacts The debate between RISC and CISC designers lasted for more than a decade. Based on Eq. 1.3, it seems that RISC will outperform CISC if the program length does not increase dramatically. Based on one reported experiment, converting from a CISC program to an equivalent RISC program increases the code length (instruction count) by only 40%.

Of course, the increase depends on program behavior, and the 40% increase may not be typical of all programs. Nevertheless, the increase in code length is much smaller than the increase in clock rate and the reduction in CPI. Thus the intuitive reasoning in Eq. 1.3 prevails in both cases, and in fact the RISC approach has proved its merit.

Further processor improvements include full 64-bit architecture, multiprocessor support such as snoopy logic for cache coherence control, faster interprocessor synchronization or hardware support for message passing, and special-function units for I/O interfaces and graphics support.

The boundary between RISC and CISC architectures has become blurred because both are now implemented with the same hardware technology. For example, starting with the VAX 9000, Motorola 88100, and Intel Pentium, CISC processors are also built with mixed features taken from both the RISC and CISC camps.

Further discussion of relevant issues in processor design will be continued in Chapter 13.

4.2 SUPERSCALAR AND VECTOR PROCESSORS

A CISC or a RISC scalar processor can be improved with a *superscalar* or *vector* architecture. Scalar processors are those executing one instruction per cycle. Only one instruction is issued per cycle, and only one completion of instruction is expected from the pipeline per cycle.

In a superscalar processor, multiple instructions are issued per cycle and multiple results are generated per cycle. A vector processor executes vector instructions on arrays of data; each vector instruction involves a string of repeated operations, which are ideal for pipelining with one result per cycle.

4.2.1 Superscalar Processors

Superscalar processors are designed to exploit more instruction-level parallelism in user programs. Only independent instructions can be executed in parallel without causing a wait state. The amount of instruction-level parallelism varies widely depending on the type of code being executed.

It has been observed that the average value is around 2 for code without loop unrolling. Therefore, for these codes there is not much benefit gained from building a machine that can issue more then three instructions per cycle. The *instruction-issue degree* in a superscalar processor has thus been limited to 2 to 5 in practice.

Pipelining in Superscalar Processors The fundamental structure of a three-issue superscalar pipeline is illustrated in Fig. 4.11. Superscalar processors were originally developed as an alternative to vector processors, with a view to exploit higher degree of instruction level parallelism.

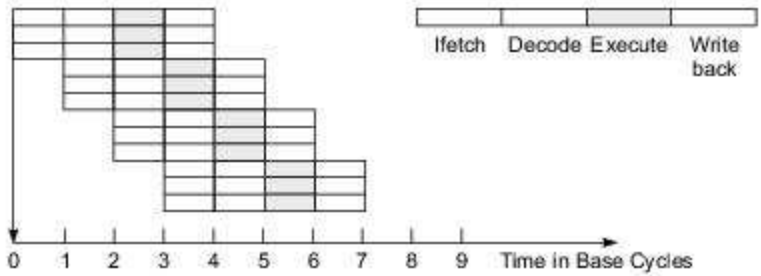


Fig. 4.11 A superscalar processor of degree m = 3

A superscalar processor of degree m can issue m instructions per cycle. In this sense, the base scalar processor, implemented either in RISC or CISC, has $m = 1$. In order to fully utilize a superscalar processor of degree m, m instructions must be executable in parallel. This situation may not be true in all clock cycles. In that case, some of the pipelines may be stalling in a wait state.

In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor. Due to the desire for a higher degree of instruction-level parallelism in programs, the superscalar processor depends more on an optimizing compiler to exploit parallelism. Table 4.5 lists some landmark examples of superscalar processors from the early 1990s.

A typical superscalar architecture for a RISC processor is shown in Fig. 4.12.

The instruction cache supplies multiple instructions per fetch. However, the actual number of instructions issued to various functional units may vary in each cycle. The number is constrained by data dependences and resource conflicts among instructions that are simultaneously decoded. Multiple functional units are built into the integer unit and into the floating-point unit.

Multiple data buses exist among the functional units. In theory, all functional units can be simultaneously used if conflicts and dependences do not exist among them during a given cycle.

Representative Superscalar Processors A number of commercially available processors have been implemented with the superscalar architecture. Notable early ones include the IBM RS/6000, DEC Alpha 21064, and Intel i960CA processors as summarized in Table 4.5. Due to the reduced CPI and higher clock rates used, generally superscalar processors outperform scalar processors.

The maximum number of instructions issued per cycle ranges from two to five in these superscalar processors. Typically, the register files in the IU and FPU each have 32 registers. Most superscalar processors implement both the IU and the FPU on the same chip. The superscalar degree is low due to limited instruction parallelism that can be exploited in ordinary programs.

Besides the register files, *reservation stations* and *reorder buffers* can be used to establish *instruction windows*. The purpose is to support instruction lookahead and internal data forwarding, which are needed

to schedule multiple instructions simultaneously. We will discuss the use of these mechanisms in Chapter 6, where advanced pipelining techniques are studied, and further in Chapter 12.

Table 4.5 *Representative Superscalar Processors* (circa 1990)

Feature	*Intel i960CA*	*IBM RS/6000*	*DEC Alpha 21064*
Technology, clock rate, year	25 MHz 1986.	1-μm CMOS technology, 30 MHz, 1990.	0.75-μm CMOS, 150 MHz, 431 pins, 1992.
Functional units and multiple instruction issues	Issue up to 3 instructions (register, memory, and control) per cycle, seven functional units available for concurrent use.	POWER architecture, issue 4 instructions (1 FXU, 1 FPU, and 2 ICU operations) per cycle.	Alpha architecture, issue 2 instructions per cycle , 64-bit IU and FPU, 128-bit data bus, and 34-bit address bus implemented in initial version.
Registers, caches, MMU, address space	l-KB I-cache, 1.5-KB RAM, 4-channel I/O with DMA, parallel decode, multiported registers.	32 32-bit GPRs, 8-KB I-cache, 64-KB D-cache with separate TLBs.	32 64-bit GPRs, 8-KB I-cache, 8-KB D-cache, 64-bit virtual space designed, 43-bit address space implemented in initial version.
Floating-point unit and functions	On-chip FPU, fast multimode interrupt, multitask control.	On-chip FPU 64-bit multiply, add, divide, subtract, IEEE 754 standard.	On-chip FPU, 32 64-bit FP registers, 10-stage pipeline, IEEE and VAX FP standards.
Claimed performance and remarks	30 VAX/MIPS peak at 25 MHz, real-time embedded system control, and multiprocessor applications.	34 MIPS and 11 Mflops at 25 MHz on POWER station 530.	300 MIPS peak and 150 Mflops peak at 150 MHz, multiprocessor and cache coherence support.

Note: KB = Kbytes, FP = floating point.

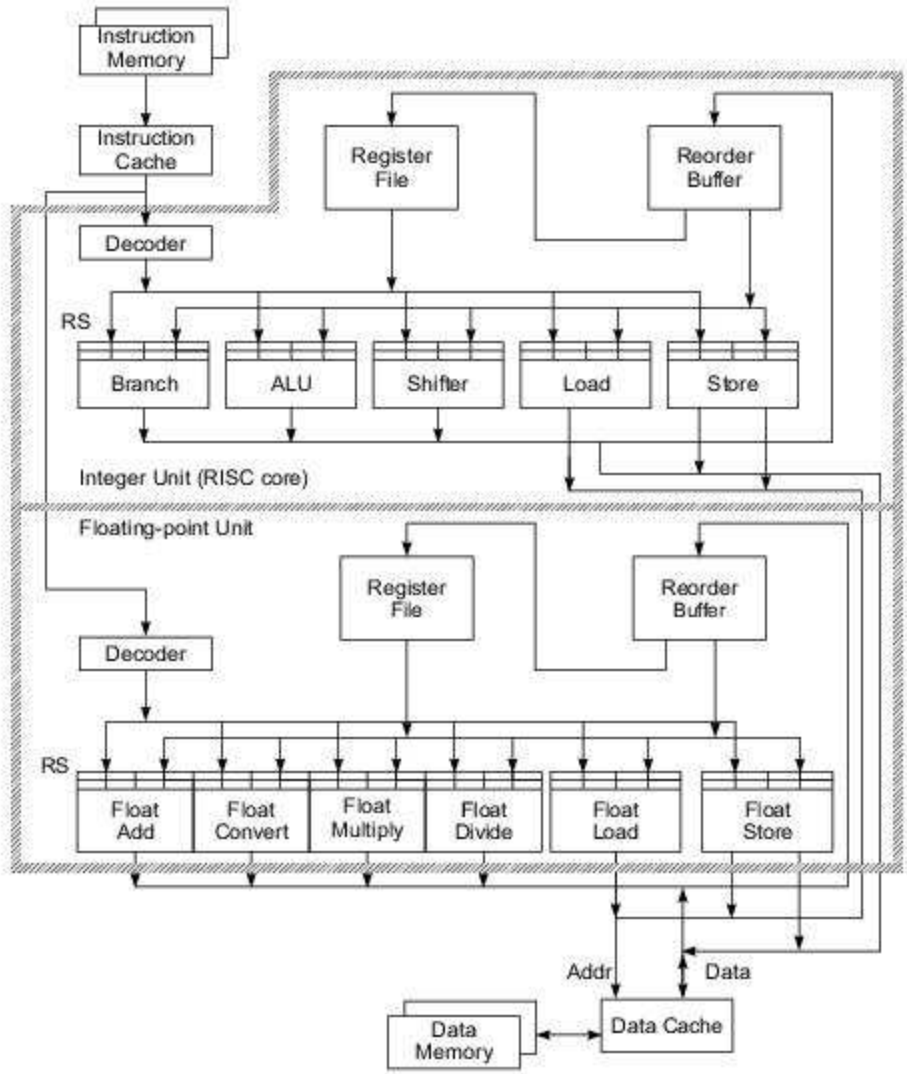


Fig. 4.12 A typical superscalar RISC processor architecture consisting of an integer unit and a floating-point unit (Courtesy of M. Johnson, 1991; reprinted with permission from Prentice-Hall, Inc.)

Example 4.5 The IBM RS/6000 architecture

In early 1990, IBM announced the RISC System 6000. It was a superscalar processor as illustrated in Fig. 4.13, with three functional units called the *branch processor*, *fixed-point unit*, and *floating-point unit*, which could operate in parallel.

The branch processor could arrange the execution of up to five instructions per cycle. These included one *branch* instruction in the branch processor, one *fixed-point* instruction in the FXU, one *condition-register* instruction in the branch processor, and one *floating-point multiply-add* instruction in the FPU, which could be counted as two floating-point operations.

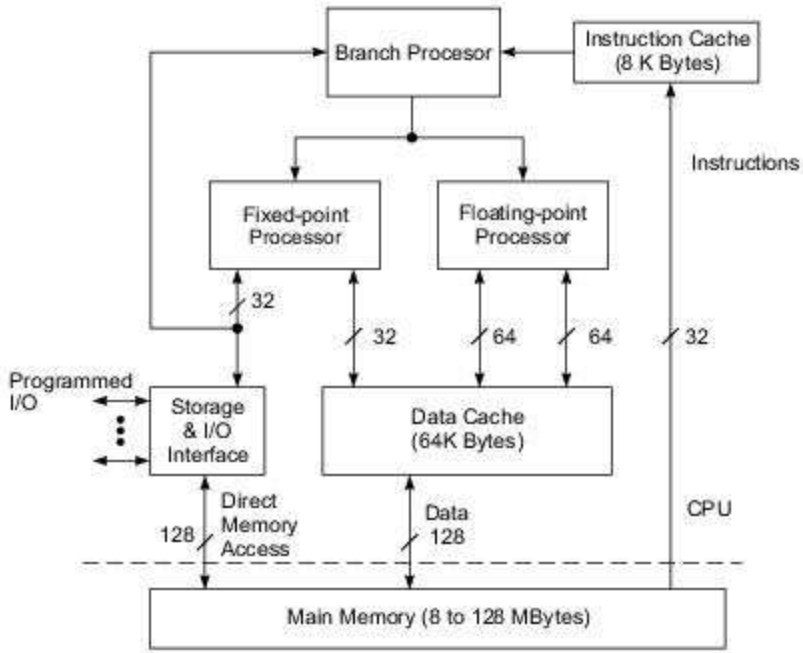


Fig. 4.13 The POWER architecture of the IBM RISC System/6000 superscalar processor (Courtesy of International Business Machines Corporation, 1990)

As any RISC processor, RS/6000 used hardwired rather than microcoded control logic. The system used a number of wide buses ranging from one word (32 bits) for the FXU to two words (64 bits) for the FPU, and four words for the I-cache and D-cache, respectively. These wide buses provided the high instruction and data bandwidths required for superscalar implementation.

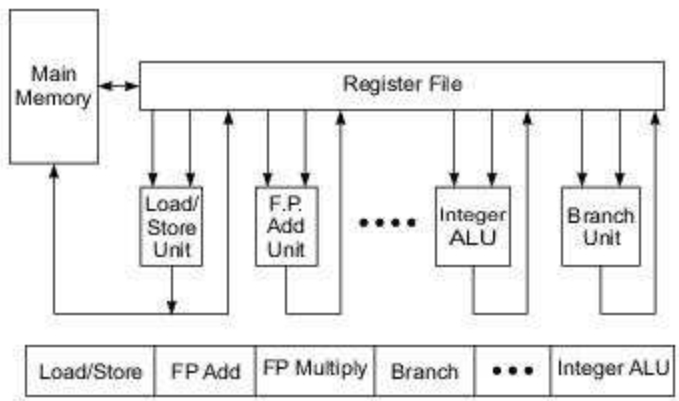
The RS/6000 design was optimized to perform well in numerically intensive scientific and engineering applications, as well as in multiuser commercial environments. A number of RS/6000-based workstations and servers were produced by IBM. For example, the POWERstation 530 had a clock rate of 25 MHz with performance benchmarks reported as 34.5 MIPS and 10.9 Mflops. In subsequent years, these systems were developed into a series of RISC-based server products. See also Chapter 13.

4.2.2 The VLIW Architecture

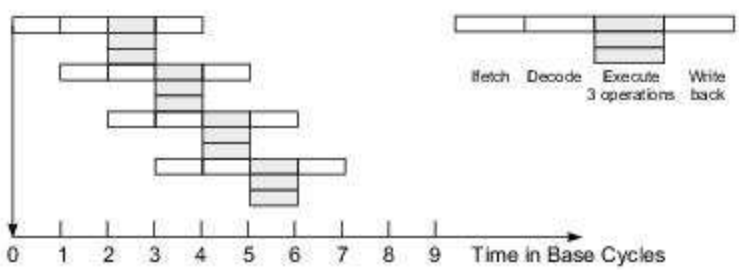
The VLIW architecture is generalized from two well-established concepts: horizontal microcoding and superscalar processing. A typical VLIW (*very long instruction word*) machine has instruction words hundreds of bits in length. As illustrated in Fig. 4.14a, multiple functional units are used concurrently in

a VLIW processor. All functional units share the use of a common large register file. The operations to be simultaneously executed by the functional units are synchronized in a VLIW instruction, say, 256 or 1024 bits per instruction word, an early example being the Multiflow computer models.

Different fields of the long instruction word carry the opcodes to be dispatched to different functional units. Programs written in conventional short instruction words (say 32 bits) must be compacted together to form the VLIW instructions. This code compaction must be done by a compiler which can predict branch outcomes using elaborate heuristics or run-time statistics.



(a) A typical VLIW processor with degree $m = 3$



(b) VLIW execution with degree $m = 3$

Fig. 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations (Courtesy of Multiflow Computer, Inc., 1987)

Pipelining in VLIW Processors The execution of instructions by an ideal VLIW processor is shown in Fig. 4.14b. Each instruction specifies multiple operations. The effective CPI becomes 0.33 in this particular example. VLIW machines behave much like superscalar machines with three differences: First, the decoding of VLIW instructions is easier than that of superscalar instructions.

Second, the code density of the superscalar machine is better when the available instruction-level parallelism is less than that exploitable by the VLIW machine. This is because the fixed VLIW format includes bits for non-executable operations, while the superscalar processor issues only executable instructions.

Third, a superscalar machine can be object-code-compatible with a large family of non-parallel machines. On the contrary, a VLIW machine exploiting different amounts of parallelism would require different instruction sets.

Instruction parallelism and data movement in a VLIW architecture are completely specified at compile time. Run-time resource scheduling and synchronization are in theory completely eliminated. One can view a VLIW processor as an extreme example of a superscalar processor in which all independent or unrelated operations are already synchronously compacted together in advance. The CPI of a VLIW processor can be even lower than that of a superscalar processor. For example, the Multiflow trace computer allows up to seven operations to be executed concurrently with 256 bits per VLIW instruction.

VLIW Opportunities In a VLIW architecture, random parallelism among scalar operations is exploited instead of regular or synchronous parallelism as in a vectorized supercomputer or in an SIMD computer. The success of a VLIW processor depends heavily on the efficiency in code compaction. The architecture is totally incompatible with that of any conventional general-purpose processor.

Furthermore, the instruction parallelism embedded in the compacted code may require a different latency to be executed by different functional units even though the instructions are issued at the same time. Therefore, different implementations of the same VLIW architecture may not be binary-compatible with each other.

By explicitly encoding parallelism in the long instruction, a VLIW processor can in theory eliminate the hardware or software needed to detect parallelism. The main advantage of VLIW architecture is its simplicity in hardware structure and instruction set. The VLIW processor can potentially perform well in scientific applications where the program behavior is more predictable.

In general-purpose applications, the architecture may not be able to perform well. Due to its lack of compatibility with conventional hardware and software, the VLIW architecture has not entered the mainstream of computers. Although the idea seems sound in theory, the dependence on trace-scheduling compiling and code compaction has prevented it from gaining acceptance in the commercial world. Further discussion of this concept will be found in Chapter 12.

4.2.3 Vector and Symbolic Processors

By definition, a *vector processor* is specially designed to perform vector computations. A vector instruction involves a large array of operands. In other words, the same operation will be performed over an array or a string of data. Specialized vector processors are generally used in supercomputers.

A vector processor can assume either a *register-to-register* architecture or a *memory-to-memory* architecture. The former uses shorter instructions and vector register files. The latter uses memory-based instructions which are longer in length, including memory addresses.

Vector Instructions Register-based vector instructions appear in most register-to-register vector processors like Cray supercomputers. Denote a vector register of length n as V_I, a *scalar register* as s_i, and a *memory array* of length n as $M(1 : n)$. Typical register-based vector operations are listed below, where a vector operator is denoted by a small circle "o":

$\mathbf{V_1}$	o	$\mathbf{V_2}$	$\rightarrow$	$\mathbf{V_3}$	(binary vector)
s_1	o	$\mathbf{V_1}$	$\rightarrow$	$\mathbf{V_2}$	(scaling)
$\mathbf{V_1}$	o	$\mathbf{V_2}$	$\rightarrow$	s_1	(binary reduction)

$$
\begin{array}{lccll}
 & M(1:n) & \rightarrow & \mathbf{V_1} & \text{(vector load)} \\
 & \mathbf{V_1} & \rightarrow & M(1:n) & \text{(vector store)} \\
o & \mathbf{V_1} & \rightarrow & \mathbf{V_2} & \text{(unary vector)} \\
o & \mathbf{V_1} & \rightarrow & s_1 & \text{(unary reduction)}
\end{array} \tag{4.1}
$$

It should be noted that the vector length should be equal in the two operands used in a binary vector instruction. The reduction is an operation on one or two vector operands, and the result is a scalar—such as the *dot product* between two vectors and the *maximum* of all components in a vector.

In all cases, these vector operations are performed by dedicated pipeline units, including *functional pipelines* and *memory-access pipelines*. Long vectors exceeding the register length n must be segmented to fit the vector registers n elements at a time.

Memory-based vector operations are found in memory-to-memory vector processors such as those in the early supercomputer CDC Cyber 205. Listed below are a few examples:

$$
\begin{array}{rcccl}
M_1(1:n) & o & M_2(1:n) & \rightarrow & M(1:n) \\
s_1 & o & M_1(1:n) & \rightarrow & M_2(1:n) \\
 & o & M_1(1:n) & \rightarrow & M_2(1:n) \\
M_1(1:n) & o & M_2(1:n) & \rightarrow & M(k)
\end{array} \tag{4.2}
$$

where $M_1(1:n)$ and $M_2(1:n)$ are two vectors of length n and $M(k)$ denotes a scalar quantity stored in memory location k. Note that the vector length is not restricted by register length. Long vectors are handled in a streaming fashion using *super words* cascaded from many shorter memory words.

Vector Pipelines Vector processors take advantage of unrolled-loop-level parallelism. The vector pipelines can be attached to any scalar or superscalar processor.

Dedicated vector pipelines eliminate some software overhead in looping control. Of course, the effectiveness of a vector processor relies on the capability of an optimizing compiler that vectorizes sequential code for vector pipelining. Typically, applications in science and engineering can make good use of vector processing capabilities.

The pipelined execution in a vector processor is compared with that in a scalar processor in Fig. 4.15. Figure 4.15a is a redrawing of Fig. 4.2a in which each scalar instruction executes only one operation over one data element. For clarity, only serial issue and parallel execution of vector instructions are illustrated in Fig. 4.2b. Each vector instruction executes a string of operations, one for each element in the vector.

We will study vector processors and SIMD architectures in Chapter 8. Various functional pipelines and their chaining or networking schemes will be introduced for the execution of compound vector functions. Many of the above vector instructions also have equivalent counterparts in an SIMD computer. Vector processing is achieved through efficient pipelining in vector supercomputers and through spatial or data parallelism in an SIMD computer.

Symbolic Processors Symbolic processing has been applied in many areas, including theorem proving, pattern recognition, expert systems, knowledge engineering, text retrieval, cognitive science, and machine intelligence. In these applications, data and knowledge representations, primitive operations, algorithmic behavior, memory, I/O and communications, and special architectural features are different than in numerical computing. Symbolic processors have also been called *prolog processors*, *Lisp processors*, or *symbolic manipulators*. Table 4.6 summarizes these characteristics.

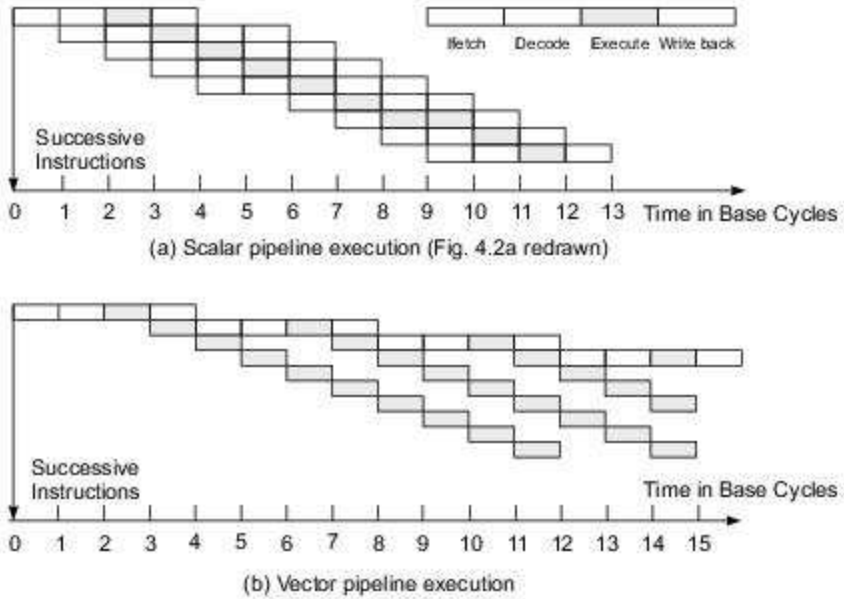


Fig. 4.15 Pipelined execution in a base scalar processor and in a vector processor, respectively (Courtesy of Jouppi and Wall; reprinted from Proc. ASPLOS, ACM Press, 1989)

Table 4.6 *Characteristics of Symbolic Processing*

Attributes	*Characteristics*
Knowledge Representations	Lists, relational databases, scripts, semantic nets, frames, blackboards, objects, production systems.
Common Operations	Search, sort, pattern matching, filtering, contexts, partitions, transitive closures, unification, text retrieval, set operations, reasoning.
Memory Requirements	Large memory with intensive access pattern. Addressing is often content -based. Locality of reference may not hold.
Communication Patterns	Message traffic varies in size and destination; granularity and format of message units change with applications.
Properties of Algorithms	Nondeterministic, possibly parallel and distributed computations. Data dependences may be global and irregular in pattern and granularity.
Input/Output requirements	User-guided programs; intelligent person-machine interfaces; inputs can be graphical and audio as well as from keyboard; access to very large on-line databases.
Architecture Features	Parallel update of large knowledge bases, dynamic load balancing; dynamic memory allocation; hardware-supported garbage collection; stack processor architecture; symbolic processors.

For example, a Lisp program can be viewed as a set of functions in which data are passed from function to function. The concurrent execution of these functions forms the basis for parallelism. The applicative and recursive nature of Lisp requires an environment that efficiently supports stack computations and function calling. The use of linked lists as the basic data structure makes it possible to implement an automatic garbage collection mechanism.

Instead of dealing with numerical data, symbolic processing deals with logic programs, symbolic lists, objects, scripts, blackboards, production systems, semantic networks, frames, and artificial neural networks.

Primitive operations for artificial intelligence include *search, compare, logic inference, pattern matching, unification, filtering, context, retrieval, set operations, transitive closure,* and *reasoning operations.* These operations demand a special instruction set containing *compare, matching, logic,* and *symbolic manipulation* operations. Floating point operations are not often used in these machines.

Example 4.6 The Symbolics 3600 Lisp processor[3]

The processor architecture of the Symbolics 3600 is shown in Fig. 4.16. This was a stack-oriented machine. The division of the overall machine architecture into layers allowed the use of a simplified instruction-set design, while implementation was carried out with a stack-oriented machine. Since most operands were fetched from the stack, the stack buffer and scratch-pad memories were implemented as fast caches to main memory.

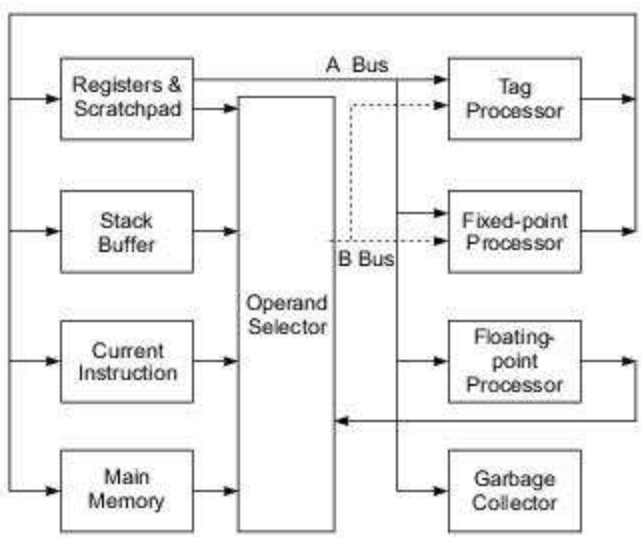


Fig. 4.16 The architecture of the Symbolics 3600 Lisp processor (Courtesy of Symbolics, Inc., 1985)

[3] The company Symbolics has since gone out of business, but the AI concepts it employed and developed are still valid. On a general-purpose computer, these concepts would be implemented in software.

The Symbolics 3600 executed most Lisp instructions in one machine cycle. Integer instructions fetched operands form the stack buffer and the duplicate top of the stack in the scratch-pad memory. Floating-point addition, garbage collection, data type checking by the tag processor, and fixed-point addition could be carried out in parallel.

4.3 MEMORY HIERARCHY TECHNOLOGY

In a typical computer configuration, the cost of memory, disks, printers, and other peripherals often exceeds that of the processors. We briefly introduce below the memory hierarchy and peripheral technology.

4.3.1 Hierarchical Memory Technology

Storage devices such as *registers*, *caches*, *main memory*, *disk devices*, and *backup storage* are often organized as a hierarchy as depicted in Fig. 4.17. The memory technology and storage organization at each level are characterized by five parameters: the *access time* (t_i), *memory size* (s_i), *cost per byte* (c_i), *transfer bandwidth* (b_i), and *unit of transfer* (x_i).

The access time t_i refers to the round-trip time from the CPU to the ith-level memory. The memory size s_i is the number of bytes or words in level i. The cost of the ith-level memory is estimated by the product $c_i s_i$. The bandwidth b_i refers to the rate at which information is transferred between adjacent levels. The unit of transfer x_i refers to the grain size for data transfer between levels i and $i+1$.

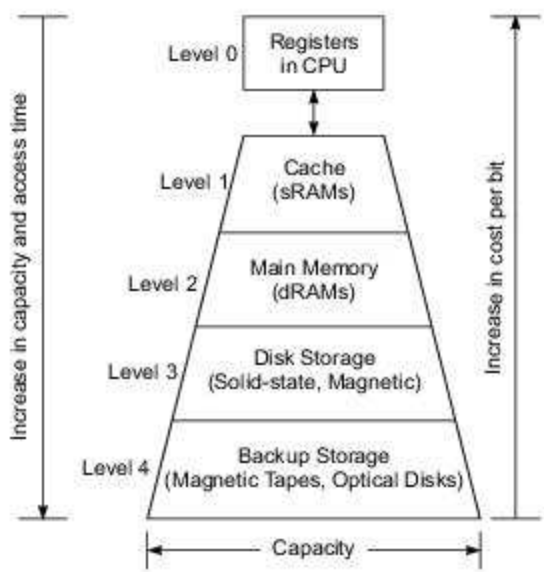


Fig. 4.17 A four-level memory hierarchy with increasing capacity and decreasing speed and cost from low to high levels

Memory devices at a lower level are faster to access, smaller in size, and more expensive per byte, having a higher bandwidth and using a smaller unit of transfer as compared with those at a higher level. In other words, we have $t_{i-1} < t_i$, $s_{i-1} < s_i$, $c_{i-1} > c_i$, $b_{i-1} > b_i$, and $x_{i-1} < x_i$, for $i = 1, 2, 3$, and 4, in the hierarchy where $i = 0$ corresponds to the CPU register level. The cache is at level 1, main memory at level 2, the disks at level 3, and backup storage at level 4. The physical memory design and operations of these levels are studied in subsequent sections and in Chapter 5.

Registers and Caches The registers are parts of the processor; multi-level caches are built either on the processor chip or on the processor board. Register assignment is made by the compiler. Register transfer operations are directly controlled by the processor after instructions are decoded. Register transfer is conducted at processor speed, in one clock cycle.

Therefore, many designers would not consider registers a level of memory. We list them here for comparison purposes. The cache is controlled by the MMU and is programmer-transparent. The cache can also be implemented at one or multiple levels, depending on the speed and application requirements. Over the last two or three decades, processor speeds have increased at a much faster rate than memory speeds. Therefore multi-level cache systems have become essential to deal with memory access latency.

Main Memory The main memory is sometimes called the primary memory of a computer system. It is usually much larger than the cache and often implemented by the most cost-effective RAM chips, such as DDR SDRAMs, i.e. dual data rate synchronous dynamic RAMs. The main memory is managed by a MMU in cooperation with the operating system.

Disk Drives and Backup Storage The disk storage is considered the highest level of on-line memory. It holds the system programs such as the OS and compilers, and user programs and their data sets. Optical disks and magnetic tape units are off-line memory for use as archival and backup storage. They hold copies of present and past user programs and processed results and files. Disk drives are also available in the form of RAID arrays.

A typical workstation computer has the cache and main memory on a processor board and hard disks in an attached disk drive. Table 4.7 presents representative values of memory parameters for a typical 32-bit mainframe computer built in 1993. Since the time, there has been one or two orders of magnitude improvement in most parameters, as we shall see in Chapter 13.

Peripheral Technology Besides disk drives and backup storage, peripheral devices include printers, plotters, terminals, monitors, graphics displays, optical scanners, image digitizers, output microfilm devices, etc. Some I/O devices are tied to special-purpose or multimedia applications.

The technology of peripheral devices has improved rapidly in recent years. For example, we used dot-matrix printers in the past. Now, as laser printers become affordable and popular, in-house publishing becomes a reality. The high demand for multimedia I/O such as image, speech, video, and music has resulted in further advances in I/O technology.

4.3.2 Inclusion, Coherence, and Locality

Information stored in a memory hierarchy (M_1, M_2,..., M_n) satisfies three important properties: *inclusion*, *coherence*, and *locality* as illustrated in Fig. 4.18. We consider cache memory the innermost level M_1, which directly communicates with the CPU registers. The outermost level M_n contains all the information words stored. In fact, the collection of all addressable words in M_n forms the virtual address space of a computer. Program and data locality is characterized below as the foundation for using a memory hierarchy effectively.

Table 4.7 *Memory Characteristics of a Typical Mainframe Computer in 1993*

Memory level Characteristics	*Level 0 CPU Registers*	*Level 1 Cache*	*Leve 2 Main Memory*	*Level 3 Disk Storage*	*Level 4 Tape Storage*
Device technology	ECL	256K-bit SRAM	4M-bit DRAM	1-Gbyte magnetic disk unit	5-Gbyte magnetic tape unit
Access time, t_i	10 ns	25–40 ns	60–100 ns	12–20 ms	2–20 min (search time)
Capacity, s_i (in bytes)	512 bytes	128 Kbytes	512 Mbytes	60–228 Gbytes	512 Gbytes–2 Tbytes
Cost, c_i (in cents/KB)	18,000	72	5.6	0.23	0.01
Bandwidth, b_i (in MB/s)	400–800	250–400	80–133	3–5	0.18–0.23
Unit of transfer, x_i	4–8 bytes per word	32 bytes per block	0.5–1 Kbytes per page	5–512 Kbytes per file	Backup storage
Allocation management	Compiler assignment	Hardware control	Operating system	Operating system/user	Operating system/user

Inclusion Property The *inclusion property* is stated as $M_1 \subset M_2 \subset M_3 \subset \ldots \subset M_n$. The set inclusion relationship implies that all information items are originally stored in the outermost level M_n. During the processing, subsets of M_n are copied into M_{n-1}. Similarly, subsets of M_{n-1} are copied into M_{n-2}, and so on.

In other words, if an information word is found in M_i, then copies of the same word can also be found in all upper levels $M_{i+1}, M_{i+2}, \ldots, M_n$. However, a word stored in M_{i+1} may not be found in M_i. A word miss in M_i implies that it is also missing from all lower levels $M_{i-1}, M_{i-2}, \ldots, M_1$. The highest level is the backup storage, where everything can be found.

Information transfer between the CPU and cache is in terms of *words* (4 or 8 bytes each depending on the word length of a machine). The cache (M_1) is divided into *cache blocks*, also called *cache lines* by some authors. Each block may be typically 32 bytes (8 words). Blocks (such as "a" and "b" in Fig. 4.18) are the units of data transfer between the cache and main memory, or between L_1 and L_2 cache, etc.

The main memory (M_2) is divided into *pages*, say, 4 Kbytes each. Each page contains 128 blocks for the example in Fig. 4.18. Pages are the units of information transferred between disk and main memory.

Scattered pages are organized as a segment in the disk memory, for example, segment F contains page A, page B, and other pages. The size of a segment varies depending on the user's needs. Data transfer between the disk and backup storage is handled at the file level, such as segments F and G illustrated in Fig. 4.18.

Coherence Property The *coherence property* requires that copies of the same information item at successive memory levels be consistent. If a word is modified in the cache, copies of that word must be updated immediately or eventually at all higher levels. The hierarchy should be maintained as such. Frequently used information is often found in the lower levels in order to minimize the effective access time of the memory hierarchy. In general, there are two strategies for maintaining the coherence in a memory hierarchy.

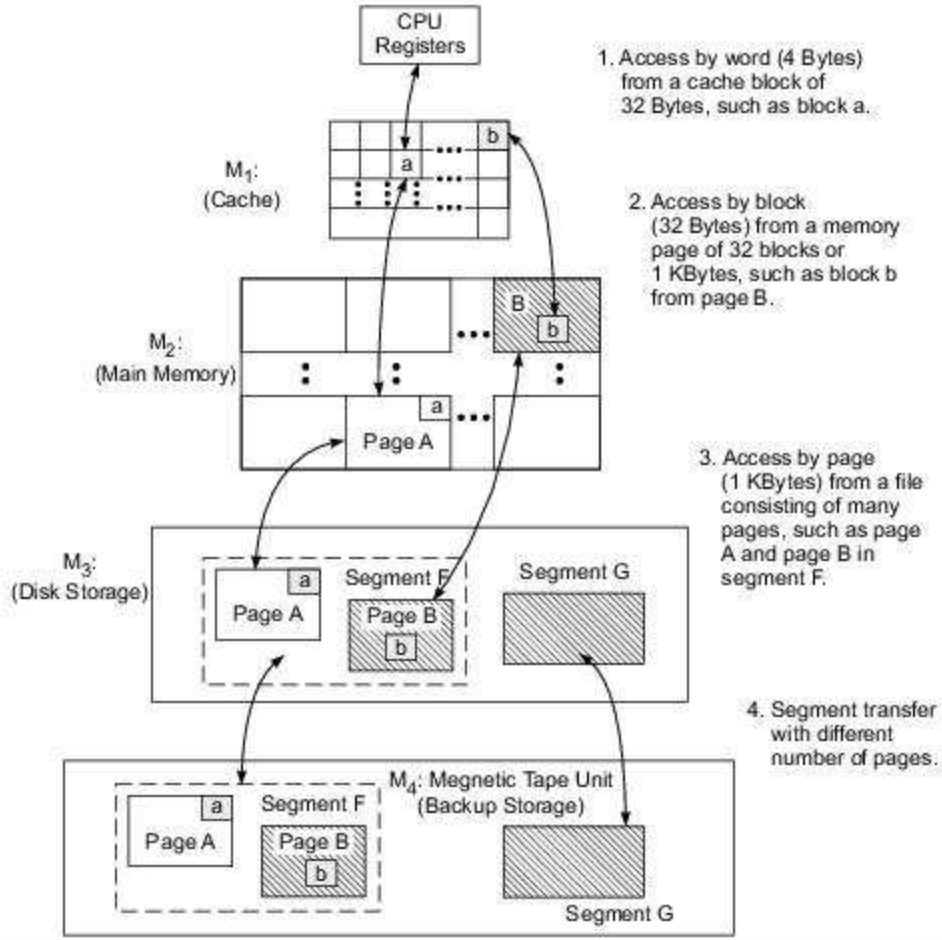


Fig. 4.18 The inclusion property and data transfers between adjacent levels of a memory hierarchy

The first method is called *write-through* (WT), which demands immediate update in M_{i+1} if a word is modified in M_i, for $i = 1, 2, \ldots, n-1$.

The second method is *write-back* (WB), which delays the update in M_{i+1} until the word being modified in M_i is replaced or removed from M_i. Memory replacement policies are studied in Section 4.4.3.

Locality of References The memory hierarchy was developed based on a program behavior known *as locality of references*. Memory references are generated by the CPU for either instruction or data access. These accesses tend to be clustered in certain regions in time, space, and ordering.

In other words, most programs act in favor of a certain portion of their address space during any time window. Hennessy and Patterson (1990) have pointed out a 90-10 rule which states that a typical program may spend 90% of its execution time on only 10% of the code such as the innermost loop of a nested looping operation.

There are three dimensions of the locality property: *temporal*, *spatial*, and *sequential*. During the lifetime of a software process, a number of pages are used dynamically. The references to these pages vary from time to time; however, they follow certain access patterns as illustrated in Fig. 4.19. These memory reference patterns are caused by the following locality properties:

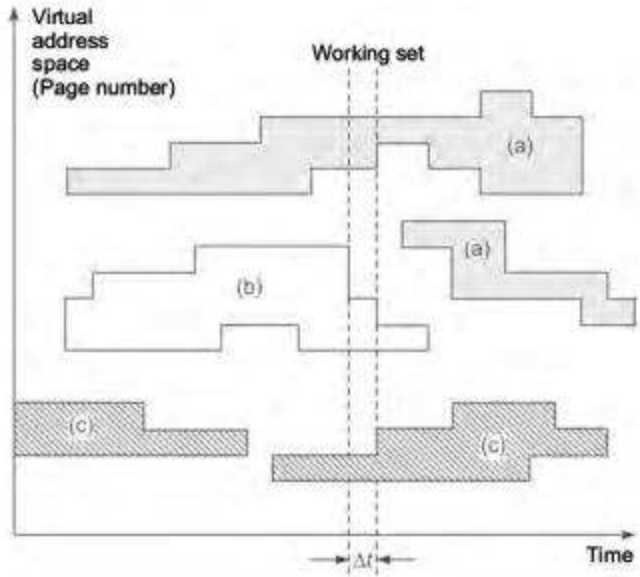


Fig. 4.19 Memory reference patterns in typical program trace experiments, where regions (a), (b), and (c) are generated with the execution of three software processes

(1) *Temporal locality*—Recently referenced items (instructions or data) are likely to be referenced again in the near future. This is often caused by special program constructs such as iterative loops, process stacks, temporary variables, or subroutines. Once a loop is entered or a subroutine is called, a small code segment will be referenced repeatedly many times. Thus temporal locality tends to cluster the access in the recently used areas.

(2) *Spatial locality*—This refers to the tendency for a process to access items whose addresses are near one another. For example, operations on tables or arrays involve accesses of a certain clustered area in the address space. Program segments, such as routines and macros, tend to be stored in the same neighborhood of the memory space.

(3) *Sequential locality*—In typical programs, the execution of instructions follows a sequential order (or the program order) unless branch instructions create out-of-order executions. The ratio of in-order execution to out-of-order execution is roughly 5 to 1 in ordinary programs. Besides, the access of a large data array also follows a sequential order.

Memory Design Implications The sequentiality in program behavior also contributes to the spatial locality because sequentially coded instructions and array elements are often stored in adjacent locations. Each type of locality affects the design of the memory hierarchy.

The temporal locality leads to the popularity of the *least recently used* (LRU) replacement algorithm, to be defined in Section 4.4.3. The spatial locality assists us in determining the size of unit data transfers between adjacent memory levels. The temporal locality also helps determine the size of memory at successive levels.

The sequential locality affects the determination of grain size for optimal scheduling (grain packing). Prefetch techniques are heavily affected by the locality properties. The principle of localities guides the design of cache, main memory, and even virtual memory organization.

The Working Sets Figure 4.19 shows the memory reference patterns of three running programs or three software processes. As a function of time, the virtual address space (identified by page numbers) is clustered into regions due to the locality of references. The subset of addresses (or pages) referenced within a given time window $(t, t+\Delta t)$ is called the *working set* by Denning (1968).

During the execution of a program, the working set changes slowly and maintains a certain degree of continuity as demonstrated in Fig. 4.19. This implies that the working set is often accumulated at the innermost (lowest) level such as the cache in the memory hierarchy. This will reduce the effective memory-access time with a higher hit ratio at the lowest memory level. The time window Δt is a critical parameter set by the OS kernel which affects the size of the working set and thus the desired cache size.

4.3.3 Memory Capacity Planning

The performance of a memory hierarchy is determined by the *effective access time* T_{eff} to any level in the hierarchy. It depends on the *hit ratios* and *access frequencies* at successive levels. We formally define these terms below. Then we discuss the issue of how to optimize the capacity of a memory hierarchy subject to a cost constraint.

Hit Ratios Hit ratio is a concept defined for any two adjacent levels of a memory hierarchy. When an information item is found in M_i, we call it a *hit*, otherwise, a *miss*. Consider memory levels M_i and M_{i-1} in a hierarchy, $i = 1, 2, \ldots, n$. The *hit ratio* h_i at M_i is the probability that an information item will be found in M_i. It is a function of the characteristics of the two adjacent levels M_{i-1} and M_i. The *miss ratio* at M_i is defined as $1 - h_i$.

The hit ratios at successive levels are a function of memory capacities, management policies, and program behavior. Successive hit ratios are independent random variables with values between 0 and 1. To simplify the future derivation, we assume $h_0 = 0$ and $h_n = 1$, which means the CPU always accesses M_1 first and the access to the outermost memory M_n is always a hit.

The *access frequency* to M_i is defined as $f_i = (1-h_1)(1-h_2)\ldots(1-h_{i-1})h_i$. This is indeed the probability of successfully accessing M_i when there are $i-1$ misses at the lower levels and a hit at M_i. Note that $\sum_{i=1}^{n} f_i = 1$ and $f_1 = h_1$.

Due to the locality property, the access frequencies decrease very rapidly from low to high levels; that is, $f_1 \gg f_2 \gg f_3 \gg \ldots \gg f_n$. This implies that the inner levels of memory are accessed more often than the outer levels.

Effective Access Time In practice, we wish to achieve as high a hit ratio as possible at M_1. Every time a miss occurs, a penalty must be paid to access the next higher level of memory. The misses have been called *block misses* in the cache and *page faults* in the main memory because blocks and pages are the units of transfer between these levels.

The time penalty for a page fault is much longer than that for a block miss due to the fact that $t_1 < t_2 < t_3$. Stone (1990) pointed out that a cache miss is 2 to 4 times as costly as a cache hit, but a page fault is 1000 to 10,000 times as costly as a page hit; but in modern systems a cache miss has a greater cost relative to a cache hit, because main memory speeds have not increased as fast as processor speeds.

Using the access frequencies f_i for $i = 1, 2, \ldots, n$, we can formally define the *effective access time* of a memory hierarchy as follows:

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$
$$= h_1 t_1 + (1 - h_1)h_2 t_2 + (1 - h_1)(1 - h_2)h_3 t_3 + \ldots +$$
$$(1 - h_1)(1 - h_2) \ldots (1 - h_{n-1})t_n \quad (4.3)$$

The first several terms in Eq. 4.3 dominate. Still, the effective access time depends on the program behavior and memory design choices. Only after extensive program trace studies can one estimate the hit ratios and the value of T_{eff} more accurately.

Hierarchy Optimization The total cost of a memory hierarchy is estimated as follows:

$$C_{total} = \sum_{i=1}^{n} c_i \cdot s_i \quad (4.4)$$

This implies that the cost is distributed over n levels. Since $c_1 > c_2 > c_3 > \ldots c_n$, we have to choose $s_1 < s_2 < s_3 < \ldots s_n$. The optimal design of a memory hierarchy should result in a T_{eff} close to the t_1 of M_1 and a total cost close to the cost of M_n. In reality, this is difficult to achieve due to the tradeoffs among n levels.

The optimization process can be formulated as a linear programming problem, given a ceiling C_0 on the total cost— that is, a problem to minimize

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i \quad (4.5)$$

subject to the following constraints:

$$s_i > 0, t_i > 0 \quad \text{for } i = 1, 2, \ldots, n$$

$$C_{total} = \sum_{i=1}^{n} c_i \cdot s_i < C_0 \quad (4.6)$$

As shown in Table 4.7, the unit cost c_i and capacity s_i at each level M_i depend on the speed t_i required. Therefore, the above optimization involves tradeoffs among t_i, c_i, s_i, and f_i or h_i at all levels $i = 1, 2, \ldots, n$. The following illustrative example shows a typical such tradeoff design.

Example 4.7 The design of a memory hierarchy

Consider the design of a three-level memory hierarchy with the following specifications for memory characteristics:

Memory level	Access time	Capacity	Cost/Kbyte
Cache	$t_1 = 25$ ns	$s_1 = 512$ Kbytes	$c_1 = \$0.12$
Main memory	$t_2 =$ unknown	$s_2 = 32$ Mbytes	$c_2 = \$0.02$
Disk array	$t_3 = 4$ ms	$s_3 =$ unknown	$c_3 = \$0.00002$

The design goal is to achieve an effective memory-access time $t = 850$ ns with a cache hit ratio $h_1 = 0.98$ and a hit ratio $h_2 = 0.99$ in main memory. Also, the total cost of the memory hierarchy is upper-bounded by \$1,500. The memory hierarchy cost is calculated as

$$C = c_1 s_1 + c_2 s_2 + c_3 s_3 \leq 1{,}500 \tag{4.7}$$

The maximum capacity of the disk is thus obtained as $s_3 = 40$ Gbytes without exceeding the budget.

Next, we want to choose the access time (t_2) of the RAM to build the main memory. The effective memory-access time is calculated as

$$t = h_1 t_1 + (1 - h_1) h_2 t_2 + (1 - h_1)(1 - h_2) h_3 t_3 \leq 850 \tag{4.8}$$

Substituting all known parameters, we have $850 \times 10^{-9} = 0.98 \times 25 \times 10^{-9} + 0.02 \times 0.99 \times t_2 + 0.02 \times 0.01 \times 1 \times 4 \times 10^{-3}$. Thus $t_2 = 1250$ ns.

Suppose one wants to double the main memory to 64 Mbytes at the expense of reducing the disk capacity under the same budget limit. This change will not affect the cache hit ratio. But it may increase the hit ratio in the main memory, and thereby, the effective memory-access time will be reduced.

4.4 VIRTUAL MEMORY TECHNOLOGY

In this section, we introduce two models of virtual memory. We study address translation mechanisms and page replacement policies for memory management. Physical memory such as caches and main memory will be studied in Chapter 5.

4.4.1 Virtual Memory Models

The main memory is considered the *physical memory* in which multiple running programs may reside. However, the limited-size physical memory cannot load in all programs fully and simultaneously. The *virtual memory* concept was introduced to alleviate this problem. The idea is to expand the use of the physical memory among many programs with the help of an auxiliary (backup) memory such as disk arrays.

Only active programs or portions of them become residents of the physical memory at one time. Active portions of programs can be loaded in and out from disk to physical memory dynamically under the coordination of the operating system. To the users, virtual memory provides almost unbounded memory space to work with. Without virtual memory, it would have been impossible to develop the multiprogrammed cr time-sharing computer systems that are in use today.

Address Spaces Each word in the physical memory is identified by a unique *physical address*. All memory words in the main memory form a *physical address space*. *Virtual addresses* are those used by machine instructions making up an executable program.

The virtual addresses must be translated into physical addresses at run time. A system of translation tables and mapping functions are used in this process. The address translation and memory management policies are affected by the virtual memory model used and by the organization of the disk and of the main memory.

The use of virtual memory facilitates sharing of the main memory by many software processes on a dynamic basis. It also facilitates software portability and allows users to execute programs requiring much more memory than the available physical memory.

Only the active portions of running programs are brought into the main memory. This permits the relocation of code and data, makes it possible to implement protection in the OS kernel, and allows high-level optimization of memory allocation and management.

Address Mapping Let V be the set of virtual addresses generated by a program running on a processor. Let M be the set of physical addresses allocated to run this program. A virtual memory system demands an automatic mechanism to implement the following mapping:

$$f_t : V \rightarrow M \cup \{\phi\} \tag{4.9}$$

This mapping is a time function which varies from time to time because the physical memory is dynamically allocated and deallocated. Consider any virtual address $v \in V$. The mapping f_t is formally defined as follows:

$$f_t(v) = \begin{cases} m, & \text{if } m \in M \text{ has been allocated to store the} \\ & \quad \text{data identified by virtual address } v \\ \phi, & \text{if data } v \text{ is missing in } M \end{cases} \tag{4.10}$$

In other words, the mapping $f_t(v)$ uniquely translates the virtual address v into a physical address m if there is a *memory hit* in M. When there is a *memory miss*, the value returned, $f_t(v) = \phi$, signals that the referenced item (instruction or data) has not been brought into the main memory at the time of reference.

The efficiency of the address translation process affects the performance of the virtual memory. Virtual memory is more difficult to implement in a multiprocessor, where additional problems such as coherence, protection, and consistency become more challenging. Two virtual memory models are discussed below.

Private Virtual Memory The first model uses a *private virtual memory space* associated with each processor, as was seen in the VAX/11 and in most UNIX systems (Fig. 4.20a). Each private virtual space is divided into pages. Virtual pages from different virtual spaces are mapped into the same physical memory shared by all processors.

The advantages of using private virtual memory include the use of a small processor address space (32 bits), protection on each page or on a per-process basis, and the use of private memory maps, which require no locking.

The shortcoming lies in the *synonym problem*, in which different virtual addresses in different virtual spaces point to the same physical page.

Shared Virtual Memory This model combines all the virtual address spaces into a single globally *shared virtual space* (Fig. 4.20b). Each processor is given a portion of the shared virtual memory to declare their addresses. Different processors may use disjoint spaces. Some areas of virtual space can be also shared by multiple processors.

Examples of machines using shared virtual memory include the IBM801, RT, RP3, System 38, the HP Spectrum, the Stanford Dash, MIT Alewife, Tera, etc. We will further study shared virtual memory in Chapter 9. Until then, all virtual memory systems discussed are assumed private unless otherwise specified.

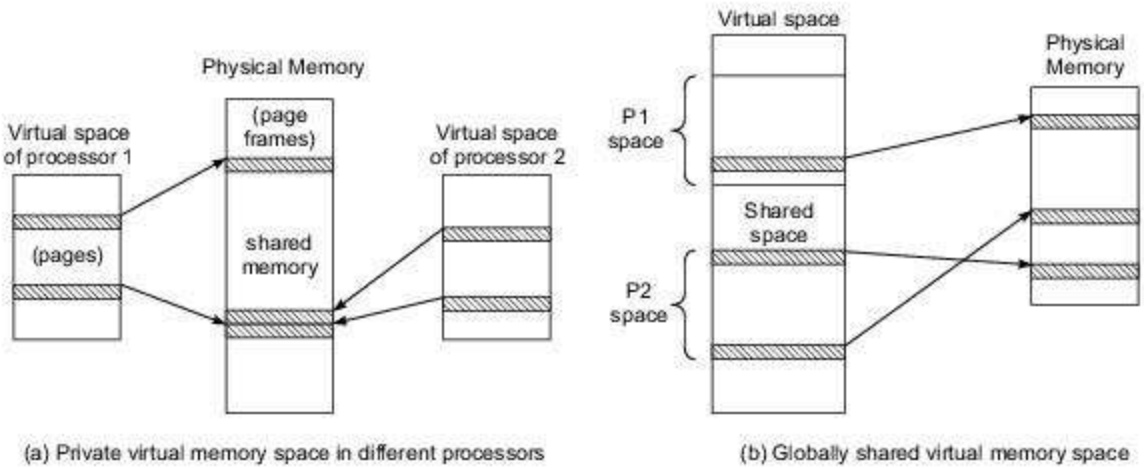


Fig. 4.20 Two virtual memory models for multiprocessor systems (Courtesy of Dubois and Briggs, tutorial, *Annual Symposium on Computer Architecture*, 1990)

The advantages in using shared virtual memory include the fact that all addresses are unique. However, each processor must be allowed to generate addresses larger than 32 bits, such as 46 bits for a 64 Tbyte (2^{46} byte) address space. Synonyms are not allowed in a globally shared virtual memory.

The page table must allow shared accesses. Therefore, *mutual exclusion* (locking) is needed to enforce protected access. Segmentation is built on top of the paging system to confine each process to its own address space (segments). Global virtual memory make may the address translation process longer.

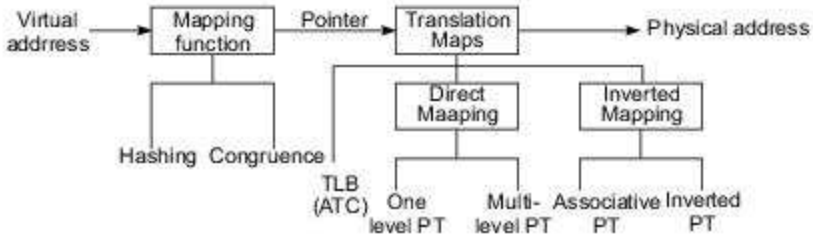
4.4.2 TLB, Paging, and Segmentation

Both the virtual memory and physical memory are partitioned into fixed-length pages as illustrated in Fig. 4.18. The purpose of memory allocation is to allocate *pages* of virtual memory to the *page frames* of the physical memory.

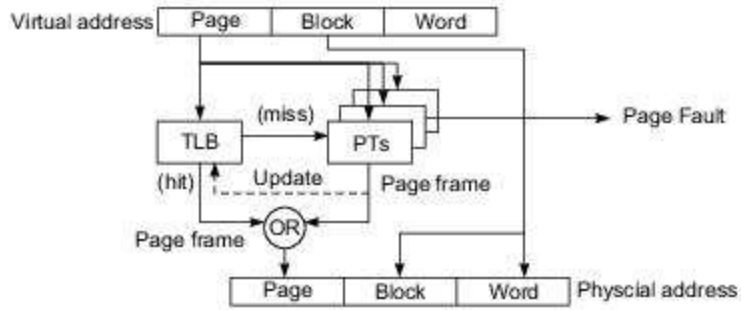
Address Translation Mechanisms The process demands the translation of virtual addresses into physical addresses. Various schemes for virtual address translation are summarized in Fig. 4.21a. The translation demands the use of *translction maps* which can be implemented in various ways.

Translation maps are stored in the cache, in associative memory, or in the main memory. To access these maps, a mapping function is applied to the virtual address. This function generates a pointer to the desired translation map. This mapping can be implemented with a *hashing* or *congruence* function.

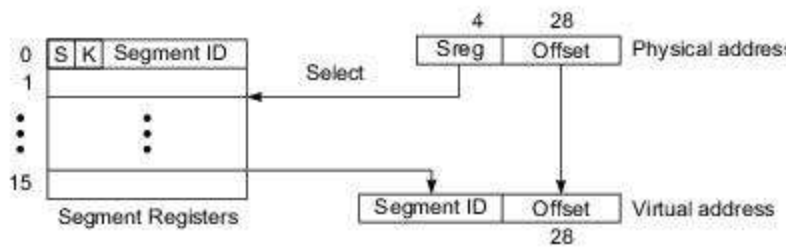
Hashing is a simple computer technique for converting a long page number into a short one with fewer bits. The hashing function should randomize the virtual page number and produce a unique hashed number to be used as the pointer.



(a) Virtual addrress translation schemes (PT = page table)



(b) Use of a TLB and PTs for address translation



(c) Inverted address mapping

Fig. 4.21 Address translation mechanisms using a TLB and various forms of page tables

Translation Lookaside Buffer Translation maps appear in the form of a *translation lookaside buffer* (TLB) and page tables (PTs). Based on the principle of locality in memory references, a particular *working set* of pages is referenced within a given context or time window.

The TLB is a high-speed lookup table which stores the most recently or likely referenced page entries. A *page entry* consists of essentially a (virtual page number, page frame number) pair. It is hoped that pages belonging to the same working set will be directly translated using the TLB entries.

The use of a TLB and PTs for address translation is shown in Fig 4.21b. Each virtual address is divided into three fields: The leftmost field holds the *virtual page number*, the middle field identifies the *cache block number*, and the rightmost field is the *word address* within the block.

Our purpose is to produce the physical address consisting of the page frame number, the block number, and the word address. The first step of the translation is to use the virtual page number as a key to search

through the TLB for a match. The TLB can be implemented with a special associative memory (content-addressable memory) or use part of the cache memory.

In case of a match (a *hit*) in the TLB, the page frame number is retrieved from the matched page entry. The cache block and word address are copied directly. In case the match cannot be found (a *miss*) in the TLB, a hashed pointer is used to identify one of the page tables where the desired page frame number can be retrieved.

Paged Memory Paging is a technique for partitioning both the physical memory and virtual memory into fixed-size pages. Exchange of information between them is conducted at the page level as described before. Page tables are used to map between pages and page frames. These tables are implemented in the main memory upon creation of user processes. Since many user processes may be created dynamically, the number of PTs maintained in the main memory can be very large. The *page table entries* (PTEs) are similar to the TLB entries, containing essentially (virtual page, page frame) address pairs.

Note that both TLB entries and PTEs need to be dynamically updated to reflect the latest memory reference history. Only "snapshots" of the history are maintained in these translation maps.

If the demanded page cannot be found in the PT, a *page fault* is declared. A page fault implies that the referenced page is not resident in the main memory. When a page fault occurs, the running process is suspended. A *context switch* is made to another ready-to-run process while the missing page is transferred from the disk or tape unit to the physical memory.

With advances in processor design and VLSI technology, very sophisticated memory management schemes can be provided on the processor chip, and even full 64 bit address space can be provided. We shall review some of these recent advances in Chapter 13.

Segmented Memory A large number of pages can be shared by segmenting the virtual address space among multiple user programs simultaneously. A *segment* of scattered pages is formed logically in the virtual memory space. Segments are defined by users in order to declare a portion of the virtual address space.

In a *segmented memory system*, user programs can be logically structured as *segments*. Segments can invoke each other. Unlike pages, segments can have variable lengths. The management of a segmented memory system is much more complex due to the nonuniform segment size.

Segments are a user-oriented concept, providing logical structures of programs and data in the virtual address space. On the other hand, paging facilitates the management of physical memory. In a paged system, all page addresses form a linear address space within the virtual space.

The segmented memory is arranged as a two-dimensional address space. Each virtual address in this space has a prefix field called the *segment number* and a postfix field called the offset within the segment. The *offset* addresses within each segment form one dimension of the contiguous addresses. The segment numbers, not necessarily contiguous to each other, form the second dimension of the address space.

Paged Segments The above two concepts of paging and segmentation can be combined to implement a type of virtual memory with *paged segments*. Within each segment, the addresses are divided into fixed-size pages. Each virtual address is thus divided into three fields. The upper field is the *segment number*, the middle one is the *page number*, and the lower one is the *offset* within each page.

Paged segments offer the advantages of both paged memory and segmented memory. For users, program files can be better logically structured. For the OS, the virtual memory can be systematically managed with

fixed-size pages within each segment. Tradeoffs do exist among the sizes of the segment field, the page field, and the offset field. This sets limits on the number of segments that can be declared by users, the segment size (the number of pages within each segment), and the page size.

Inverted Paging The direct paging described above works well with a small virtual address space such as 32 bits. In modern computers, the virtual address is large, such as 52 bits in the IBM RS/6000 or even 64 bits in some processors. A large virtual address space demands either large PTs or multilevel direct paging which will slow down the address translation process and thus lower the performance.

Besides direct mapping, address translation maps can also be implemented with inverted mapping (Fig. 4.21c). An *inverted page table* is created for each page frame that has been allocated to users. Any virtual page number can be paired with a given physical page number.

Inverted page tables are accessed either by an associative search or by the use of a hashing function. The IBM 801 prototype and subsequently the IBM RT/PC have implemented inverted mapping for page address translation. In using an inverted PT, only virtual pages that are currently resident in physical memory are included. This provides a significant reduction in the size of the page tables.

The generation of a long virtual address from a short physical address is done with the help of segment registers, as demonstrated in Fig. 4.21c. The leading 4 bits (denoted *sreg*) of a 32-bit address name a segment register. The register provides a *segment id* that replaces the 4-bit sreg to form a long virtual address.

This effectively creates a single long virtual address space with segment boundaries at multiples of 256 Mbytes (2^{28} bytes). The IBM RT/PC had a 12-bit segment id (4096 segments) and a 40-bit virtual address space.

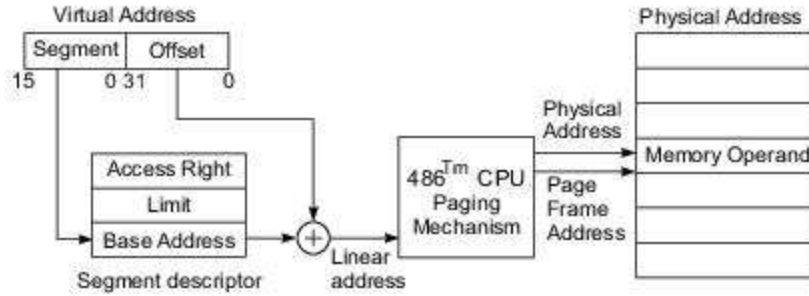
Either associative page tables or inverted page tables can be used to implement inverted mapping. The inverted page table can also be assisted with the use of a TLB. An inverted PT avoids the use of a large page table or a sequence of page tables.

Given a virtual address to be translated, the hardware searches the inverted PT for that address and, if it is found, uses the table index of the matching entry as the address of the desired page frame. A hashing table is used to search through the inverted PT. The size of an inverted PT is governed by the size of the physical space, while that of traditional PTs is determined by the size of the virtual space. Because of limited physical space, no multiple levels are needed for the inverted page table.

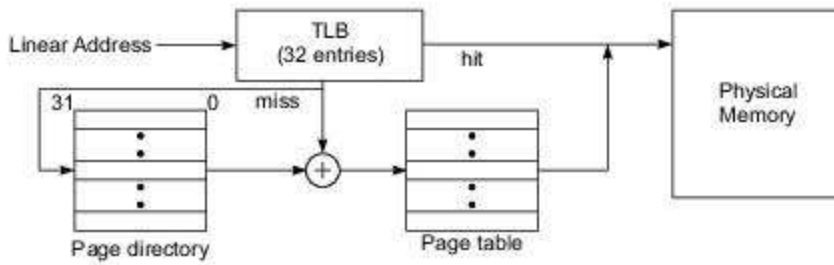
Example 4.8 Paging and segmentation in the Intel i486 processor

As with its predecessor in the x86 family, the i486 features both segmentation and paging capabilities. Protected mode increases the linear address from 4 Gbytes (2^{32} bytes) to 64 Tbytes (2^{46} bytes) with four levels of protection. The maximal memory size in real mode is 1 Mbyte (2^{20} bytes). Protected mode allows the i486 to run all software from existing 8086, 80286, and 80386 processors. A segment can have any length from 1 byte to 4 Gbytes, the maximum physical memory size.

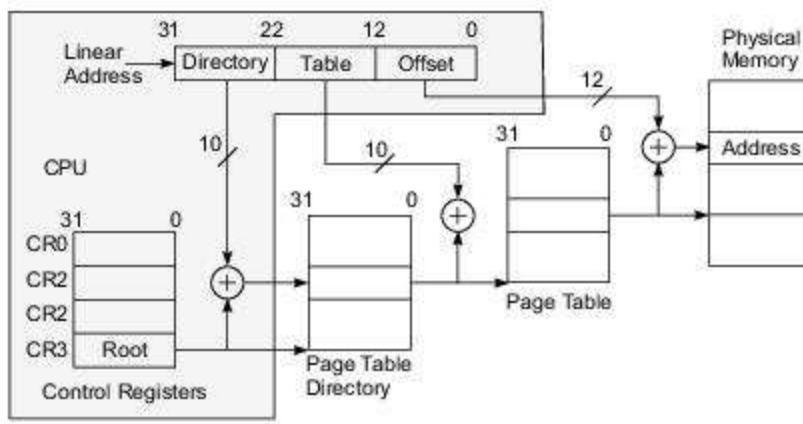
A segment can start at any base address, and storage overlapping between segments is allowed. The virtual address (Fig. 4.22a) has a 16-bit segment selector to determine the base address of the *linear address space* to be used with the i486 paging system.



(a) Segmentation to produce the linear address



(b) The TLB operations



(c) A two-level paging scheme

Fig. 4.22 Paging and segmentation mechanisms built into the Intel i486 CPU (Courtesy of Intel Corporation, 1990)

The 32-bit offset specifies the internal address within a segment. The segment descriptor is used to specify access rights and segment size besides selection of the address of the first byte of the segment.

The paging feature is optional on the i486. It can be enabled or disabled by software control. When paging is enabled, the virtual address is first translated into a linear address and then into the physical address. When paging is disabled, the linear address and physical address are identical. When a 4-Gbyte segment is selected, the entire physical memory becomes one large segment, which means the segmentation mechanism is essentially disabled.

In this sense, the i486 can be used with four different memory organizations, *pure paging*, *pure segmentation*, *segmented paging*, or *pure physical addressing* without paging and segmentation.

A 32-entry TLB (Fig 4.22b) is used to convert the linear address directly into the physical address without resorting to the two-level paging scheme (Fig 4.22c). The standard page size on the i486 is 4 Kbytes = 2^{12} bytes. Four control registers are used to select between regular paging and page fault handling.

The page table directory (4 Kbytes) allows 1024 page directory entries. Each page table at the second level is 4 Kbytes and holds up to 1024 PTEs. The upper 20 linear address bits are compared to determine if there is a hit. The hit ratios of the TLB and of the page tables depend on program behavior and the efficiency of the update (page replacement) policies. A 98% hit ratio has been observed in TLB operations.

Advanced memory management functions, to support virtual memory implementation, were first introduced in Intel's x86 processor family with the 80386 processor. Key features of the 80486 memory management scheme described here were carried forward in the Pentium family of processors.

4.4.3 Memory Replacement Policies

Memory management policies include the allocation and deallocation of memory pages to active processes and the replacement of memory pages. We will study allocation and deallocation problems in Section 5.3.3 after we discuss main memory organization in Section 5.3.1.

In this section, we study page replacement schemes which are implemented with demand paging memory systems. *Page replacement* refers to the process in which a resident page in main memory is replaced by a new page transferred from the disk.

Since the number of available page frames is much smaller than the number of pages, the frames will eventually be fully occupied. In order to accommodate a new page, one of the resident pages must be replaced. Different policies have been suggested for page replacement. These policies are specified and compared below.

The goal of a page replacement policy is to minimize the number of possible page faults so that the effective memory-access time can be reduced. The effectiveness of a replacement algorithm depends on the program behavior and memory traffic patterns encountered. A good policy should match the program locality property. The policy is also affected by page size and by the number of available frames.

Page Traces To analyze the performance of a paging memory system, page trace experiments are often performed. A *page trace* is a sequence of *page frame numbers* (PFNs) generated during the execution of a given program. To simplify the analysis, we ignore the cache effect.

Each PFN corresponds to the prefix portion of a physical memory address. By tracing the successive PFNs in a page trace against the resident page numbers in the page frames, one can determine the occurrence of

page hits or of page faults. Of course, when all the page frames are taken, a certain replacement policy must be applied to swap the pages. A page trace experiment can be performed to determine the *hit ratio* of the paging memory system. A similar idea can also be applied to perform *block traces* on cache behavior.

Consider a page trace $P(n) = r(1)r(2) \ldots r(n)$ consisting of n PFNs requested in discrete time from 1 to n, where $r(t)$ is the PFN requested at time t. We define two reference distances between the repeated occurrences of the same page in $P(n)$.

The *forward distance* $f_t(x)$ for page x is the number of time slots from time t to the first repeated reference of page x in the future:

$$f_t(x) = \begin{cases} k, & \text{if } k \text{ is the smallest integer such that} \\ & r(t+k) = r(t) = x \text{ in } P(n) \\ \infty, & \text{if } x \text{ does not reappear in } P(n) \text{ beyond time } t \end{cases} \tag{4.11}$$

Similarly, we define a *backward distance* $b_t(x)$ as the number of time slots from time t to the most recent reference of page x in the past:

$$b_t(x) = \begin{cases} k, & \text{if } k \text{ is the smallest integer such that} \\ & r(t-k) = r(t) = x \text{ in } P(n) \\ \infty, & \text{if } x \text{ never appeared in } P(n) \text{ in the past} \end{cases} \tag{4.12}$$

Let $R(t)$ be the *resident set* of all pages residing in main memory at time t. Let $q(t)$ be the page to be replaced from $R(t)$ when a page fault occurs at time t.

Page Replacement Policies The following page replacement policies are specified in a demand paging memory system for a page fault at time t.

(1) *Least recently used* (LRU)—This policy replaces the page in $R(t)$ which has the longest backward distance:

$$q(t) = y, \quad \text{iff} \quad b_t(y) = \max_{x \in R(t)} \{b_t(x)\} \tag{4.13}$$

(2) *Optimal* (OPT) *algorithm*—This policy replaces the page in $R(t)$ with the longest forward distance:

$$q(t) = y, \quad \text{iff} \quad f_t(y) = \max_{x \in R(t)} \{f_t(x)\} \tag{4.14}$$

(3) *First-in-first-out* (FIFO)—This policy replaces the page in $R(t)$ which has been in memory for the longest time.

(4) *Least frequently used* (LFU)—This policy replaces the page in $R(t)$ which has been least referenced in the past.

(5) *Circular FIFO*—This policy joins all the page frame entries into a circular FIFO queue using a pointer to indicate the front of the queue. An *allocation bit* is associated with each page frame. This bit is set upon initial allocation of a page to the frame.

When a page fault occurs, the queue is circularly scanned from the pointer position. The pointer skips the allocated page frames and replaces the very first unallocated page frame. When all frames are allocated, the front of the queue is replaced, as in the FIFO policy.

(6) *Random replacement*—This is a trivial algorithm which chooses any page for replacement randomly.

Example 4.9 Page tracing experiments and interpretation of results

Consider a paged virtual memory system with a two-level hierarchy: main memory M_1 and disk memory M_2. For clarity of illustration, assume a page size of four words. The number of page frames in M_1 is 3, labeled a, b and c; and the number of pages in M_2 is 10, identified by 0, 1, 2, …, 9. The ith page in M_2 consists of word addresses $4i$ to $4i + 3$ for all $i = 0, 1, 2, \ldots, 9$.

A certain program generates the following sequence of word addresses which are grouped (underlined) together if they belong to the same page. The sequence of page numbers so formed is the *page trace*:

Word trace:	0,1,2,3,	4,5,6,7,	8,	16,17,	9,10,11,	12,	28,29,30,	8,9,10,	4,5,	12,	4,5
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Page trace:	0	1	2	4	2	3	7	2	1	3	1

Page tracing experiments are described below for three page replacement policies: LRU, OPT, and FIFO, respectively. The successive pages loaded in the page frames (PFs) form the trace entries. Initially, all PFs are empty.

	PF	0	1	2	4	2	3	7	2	1	3	1	Hit Ratio
LRU	*a*	0	0	0	4	4	4	7	7	7	3	3	
	b		1	1	1	1	3	3	3	1	1	1	$\frac{3}{11}$
	c			2	2	2	2	2	2	2	2	2	
	Faults	*	*	*	*		*	*		*	*		
OPT	*a*	0	0	0	4	4	3	7	7	7	3	3	
	b		1	1	1	1	1	1	1	1	1	1	$\frac{4}{11}$
	c			2	2	2	2	2	2	2	2	2	
	Fault	*	*	*	*		*	*			*		
FIFO	*a*	0	0	0	4	4	4	4	2	2	2	2	
	b		1	1	1	1	3	3	1	1	1	1	$\frac{2}{11}$
	c			2	2	2	2	7	7	7	3	3	
	Faults	*	*	*	*		*	*	*	*	*		

The above results indicate the superiority of the OPT policy over the others. However, the OPT cannot be implemented in practice. The LRU policy performs better than the FIFO due to the locality of references. From these results, we realize that the LRU is generally better than the FIFO. However, exceptions still exist due to the dependence on program behavior.

Relative Performance The performance of a page replacement algorithm depends on the page trace (program behavior) encountered. The best policy is the OPT algorithm. However, the OPT replacement is not realizable because no one can predict the future page demand in a program.

The LRU algorithm is a popular policy and often results in a high hit ratio. The FIFO and random policies may perform badly because of violation of the program locality.

The circular FIFO policy attempts to approximate the LRU with a simple circular queue implementation. The LFU policy may perform between the LRU and the FIFO policies. However, there is no fixed superiority of any policy over the others because of the dependence on program behavior and run-time status of the page frames.

In general, the page fault rate is a monotonic decreasing function of the size of the resident set $R(t)$ at time t because more resident pages result in a higher hit ratio in the main memory.

Block Replacement policies The relationship between the cache block frames and cache blocks is similar to that between page frames and pages on a disk. Therefore, those page replacement policies can be modified for *block replacement* when a cache miss occurs.

Different cache organizations (Section 5.1) may offer different flexibilities in implementing some of the block replacement algorithms. The cache memory is often associatively searched, while the main memory is randomly addressed.

Due to the difference between page allocation in main memory and block allocation in the cache, the cache hit ratio and memory page hit ratio are affected by the replacement policies differently. *Cache traces* are often needed to evaluate the cache performance. These considerations will be further discussed in Chapter 5.

Summary

One way to define the design space of processors is in terms of the processor clock rate and the average cycles per instruction (CPI). Depending on the intended applications, different processors—which may even be of the same processor family—may occupy different positions within this design space. The processor instruction set may be complex or reduced—and accordingly these two types of processors occupy different regions of the design space of clock rate *versus* CPI.

For higher performance, processor designs have evolved to superscalar processors in one direction, and vector processors in the other. A superscalar processor can schedule two or more machine instructions through the instruction pipeline in a single clock cycle. Most sequential programs, when translated into machine language, do contain some level of instruction level parallelism. Superscalar processors aim to exploit this parallelism through hardware techniques built into the processor.

Vector processors aim to exploit a common characteristic of most scientific and engineering applications—processing of large amounts of numeric data in the form of vectors or arrays. The earliest supercomputers—CDC and Cray—emphasized vector processing, whereas modern applications requirements span a much broader range, and as a result the scope of computer architecture is also broader today.

Very large instruction word (VLIW) processors were proposed on the premise that the compiler can schedule multiple independent operations per cycle and pack them into long machine instructions—relieving the hardware from the task of discovering instruction level parallelism. Symbolic processors address the needs of artificial intelligence, which may be contrasted with the number-crunching which was the focus of earlier generations of supercomputers.

Memory elements provided within the processor operate at processor speed, but they are small in size, limited by cost and power consumption. Farther away from the processor, memory elements commonly provided are (one or more levels of) cache memory, main memory, and secondary storage. The memory at each level is slower than the one at the previous level, but also much larger and less expensive per bit. The aim behind providing a memory hierarchy is to achieve, as far as possible, the speed of fast memory at the cost of the slower memory. The properties of inclusion, coherence and locality make it possible to achieve this complex objective in a computer system.

Virtual memory systems aim to free program size from the size limitations of main memory. Working set, paging, segmentation, TLBs, and memory replacement policies make up the essential elements of a virtual memory system, with locality of program references once again playing an important role.

Exercises

Problem 4.1 Define the following basic terms related to modern processor technology:

(a) Processor design space.
(b) Instruction issue latency.
(c) Instruction issue rate.
(d) Simple operation latency.
(e) Resource conflicts.
(f) General-purpose registers.
(g) Addressing modes.
(h) Unified versus split caches.
(i) Hardwired versus microcoded control.

Problem 4.2 Define the following basic terms associated with memory hierarchy design:

(a) Virtual address space.
(b) Physical address space.
(c) Address mapping.
(d) Cache blocks.
(e) Multilevel page tables.
(f) Hit ratio.
(g) Page fault.
(h) Hashing function.
(i) Inverted page table.
(j) Memory replacement policies.

Problem 4.3 Answer the following questions on designing scalar RISC or superscalar RISC processors:

(a) Why do most RISC integer units use 32 general-purpose registers? Explain the concept of register windows implemented in the SPARC architecture.
(b) What are the design tradeoffs between a large register file and a large D-cache? Why are reservation stations or reorder buffers needed in a superscalar processor?
(c) Explain the relationship between the integer unit and the floating-point unit in most RISC processors with scalar or superscalar organization.

Problem 4.4 Based on the discussion of advanced processors in Section 4.1, answer the following questions on RISC, CISC, superscalar, and VLIW architectures.

(a) Compare the instruction-set architecture in RISC and CISC processors in terms of instruction formats, addressing modes, and cycles per instruction (CPI).
(b) Discuss the advantages and disadvantages in

using a common cache or separate caches for instructions and data. Explain the support from data paths, MMU and TLB, and memory bandwidth in the two cache architectures.

(c) Distinguish between scalar RISC and superscalar RISC in terms of instruction issue, pipeline architecture, and processor performance.

(d) Explain the difference between superscalar and VLIW architectures in terms of hardware and software requirements.

Problem 4.5 Explain the structures and operational requirements of the instruction pipelines used in CISC, scalar RISC, superscalar RISC, and VLIW processors. Comment on the cycles per instruction expected from these processor architectures.

Problem 4.6 Study the Intel i486 instruction set and the CPU architecture, and answer the following questions:

(a) What are the instruction formats and data formats?

(b) What are the addressing modes?

(c) What are the instruction categories? Describe one example instruction in each category.

(d) What are the HLL support instructions and assembly directives?

(e) What are the intenupt, testing, and debug features?

(f) Explain the difference between real and virtual mode execution.

(g) Explain how to disable paging in the i486 and what kind of application may benefit from this option.

(h) Explain how to disable segmentation in the i486 and what kind of application may use this option.

(i) What kind of protection mechanisms are built into the i486?

(j) Search for information on the Pentium and explain the impovements made, compared with the i486.

Problem 4.7 Answer the following questions after studying Example 4.4, the i860 instuction set, and the architecture of the i860 and its successor the i860XP:

(a) Repeat parts (a), (b), and (c) in Problem 4.6 for the i860/i860XP.

(b) What multiprocessor, support instructions are added in the i860XP?

(c) Explain the dual-instrution mode and the dual-operation instructions in i860 processors.

(d) Explain the address translation and paged memory organization of the i860.

Problem 4.8 The SPARC arhitecture can be implemented with two to eight register windows, for a total of 40 to 132 GPRs in the integer unit. Explain how the GPRs are organized into overlapping windows in each of the following designs:

(a) Use 40 GPRs to construct two windows.

(b) Use 72 GPRs to construct four windows.

(c) In what sense is the SPARC considered a scalable architecture?

(d) Explain how to use the overlapped windows for parameter passing between the calling procedure and the called procedure.

Problem 4.9 Study Section 4.2 and also the paper by Jouppi and Wall (1989) and answer the following questions:

(a) What causes a processor pipeline to be underpipelined?

(b) What are the factors limiting the degree of superscalar design?

Problem 4.10 Answer the following questions related to vector processing:

(a) What are the differences between scalar instructions and vector instructions?

(b) Compare the pipelined execution style in a vector processor with that in a base scalar

processor (Fig. 4.15). Analyze the speedup gain of the vector pipeline over the scalar pipeline for long vectors.

(c) Suppose parallel issue is added to vector pipeline execution. What would be the further improvement in throughput, compared with parallel issue in a superscalar pipeline of the same degree?

Problem 4.11 Consider a two-level memory hierarchy, M_1 and M_2. Denote the hit ratio of M_1, as h. Let c_1 and c_2 be the costs per kilobyte, s_1 and s_2 the memory capacities, and t_1 and t_2 the access times, respectively.

(a) Under what conditions will the average cost of the entire memory system approach c_2?

(b) What is the effective memory-access time t_a of this hierarchy?

(c) Let $r = t_2/t_1$ be the speed ratio of the two memories. Let $E = t_1/t_a$ be the access efficiency of the memory system. Express E in terms of r and h.

(d) Plot E against h for $r = 5$, 20, and 100, respectively, on grid paper.

(e) What is the required hit ratio h to make $E > 0.95$ if $r = 100$?

Problem 4.12 You are asked to perform capacity planning for a two-level memory system. The first level, M_1, is a cache with three capacity choices of 64 Kbytes, 128 Kbytes, and 256 Kbytes. The second level, M_2, is a main memory with a 4-Mbyte capacity. Let c_1 and c_2 be the costs per byte and t_1 and t_2 the access times for M_1 and M_2, respectively. Assume $c_1 = 20c_2$ and $t_2 = 10t_1$. The cache hit ratios for the three capacities are assumed to be 0.7, 0.9, and 0.98, respectively.

(a) What is the average access time t_a in terms of $t_1 = 20$ ns in the three cache designs? (Note that t_1 is the time from CPU to M_1 and t_2 is that from CPU to M_2, not from M_1 to M_2).

(b) Express the average byte cost of the entire memory hierarchy if $c_2 = \$0.2$/Kbyte.

(c) Compare the three memory designs and indicate the order of merit in terms of average costs and average access times, respectively. Choose the optimal design based on the product of average cost and average access time.

Problem 4.13 Compare the advantages and shortcomings in implementing private virtual memories and a globally shared virtual memory in a multicomputer system. This comparative study should consider the latency, coherence, page migration, protection, implementation, and application problems in the context of building a scalable multicomputer system with distributed shared memories.

Problem 4.14 Explain the *inclusion property* and *memory coherence* requirements in a multilevel memory hierarchy. Distinguish between *write-through* and *write-back* policies in maintaining the coherence in adjacent levels. Also explain the basic concepts of *paging* and *segmentation* in managing the physical and virtual memories in a hierarchy.

Problem 4.15 A two-level memory system has eight virtual pages on a disk to be mapped into four page frames (PFs) in the main memory. A certain program generated the following page trace:

1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5, 1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3

(a) Show the successive virtual pages residing in the four page frames with respect to the above page trace using the LRU replacement policy. Compute the hit ratio in the main memory. Assume the PFs are initially empty.

(b) Repeat part (a) for the circular FIFO page replacement policy. Compute the hit ratio in the main memory.

(c) Compare the hit ratio in parts (a) and (b) and comment on the effectiveness of using the circular FIFO policy to approximate the LRU policy with respect to this particular page trace.

Problem 4.16

(a) Explain the temporal locality, spatial locality, and sequential locality associated with program/data access in a memory hierarchy.

(b) What is the working set? Comment on the sensitivity of the observation window size to the size of the working set. How will this affect the main memory hit ratio?

(c) What is the 90–10 rule and its relationship to the locality of references?

Problem 4.17 Consider a two-level memory hierarchy, M_1 and M_2, with access times t_1 and t_2, costs per byte c_1 and c_2, and capacities s_1 and s_2, respectively. The cache hit ratio $h_1 = 0.95$ at the first level. (Note that t_2 is the access time between the CPU and M_2, not between M_1 and M_2).

(a) Derive a formula showing the effective access time t_{eff} of this memory system.

(b) Derive a formula showing the total cost of this memory system.

(c) Suppose $t_1 = 20$ ns, t_2 is unknown, $s_1 = 512$ Kbytes, s_2 is unknown, $c_1 = \$0.01$/byte, and $c_2 = \$0.0005$/byte. The total cost of the cache and main memory is upper-bounded by \$15,000.

(i) How large a capacity of M_2 ($s_2 = ?$) can you acquire without exceeding the budget limit?

(ii) How fast a main memory ($t_2 = ?$) do you need to achieve an effective access time of $t_{eff} = 40$ ns in the entire memory system under the above hit ratio assumptions?

Problem 4.18 Distinguish between numeric processing and symbolic processing computers in terms of data objects, common operations, memory requirements, communication patterns, algorithmic properties, I/O requirements, and processor architectures.