Part II

Hardware Technologies

Chapter 4

Processors and Memory Hierarchy

Chapter 5

Bus, Cache, and Shared Memory

Chapter 6

Pipelining and Superscalar Techniques

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.

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.

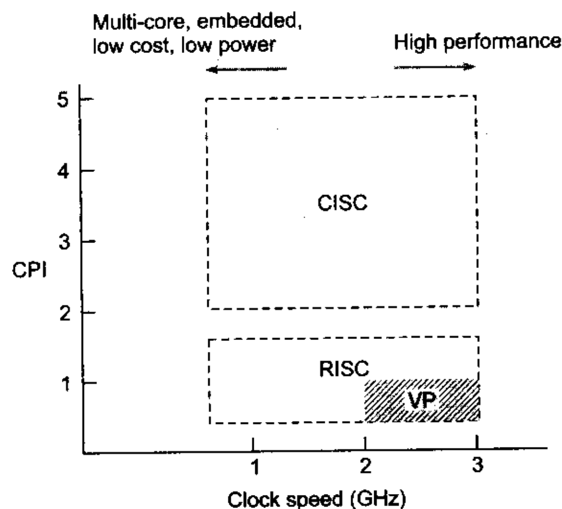


Fig. 4.1 CPI versus processor clock speed of major categories of processors

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.

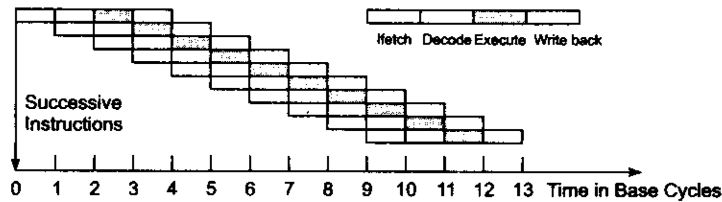
The processors in *vector supercomputers* use multiple functional units for concurrent scalar and vector operations.

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.

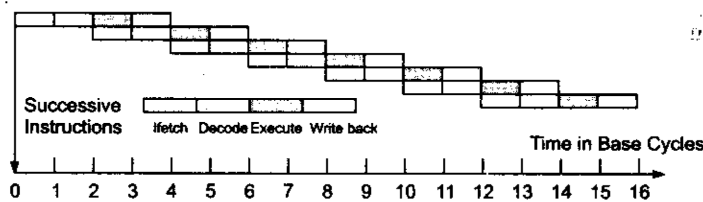
Instruction Pipelines The execution cycle of a typical instruction includes four phases: *fetch*, *decode*, *execute*, and *write-back*. These instruction phases are often executed by an *instruction pipeline* as demonstrated in Fig. 4.2a. In other words, we can simply model an *instruction processor*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

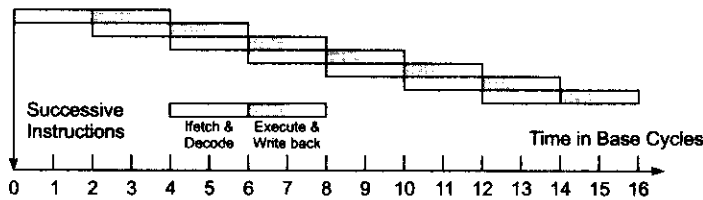
A *pipeline cycle* is intuitively defined as the time required for each phase to complete its operation, assuming equal delay in all phases (pipeline stages). Introduced below are the basic definitions associated with instruction pipeline operations:



(a) Execution in a base scalar processor



(b) Underpipelined with two cycles per instruction issue



(c) Underpipelined with twice the base cycle

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.

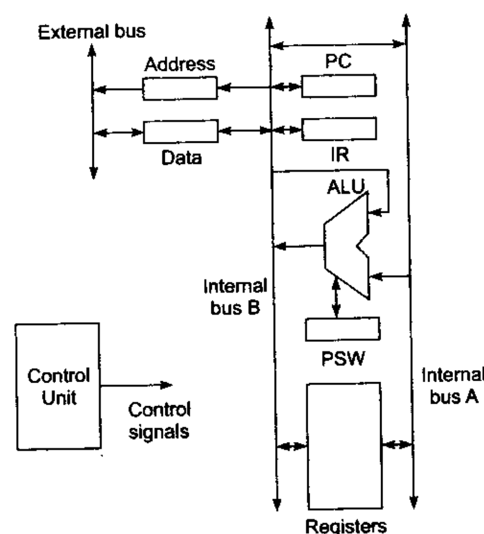


Fig. 4.3 Data path architecture and control unit of a scalar processor

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 technoiogy. 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

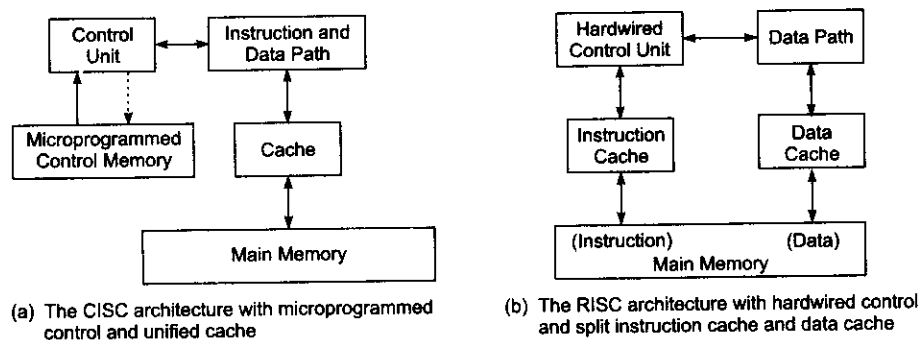
[1]Fuller discussion of these basic architectural concepts can be found in *Computer System Organisation*, by Naresh Jotwani, Tata McGraw-Hill, 2009.

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 ofTypical 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 elsc machines. The instruction unit prefetched and decoded instructions, handled branching operations, and supplied operands to the two functional units in a pipelined fashion.

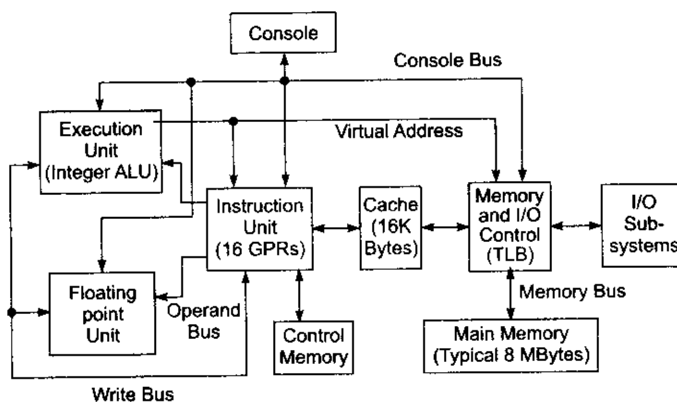


Fig. 4.5 The VAX 8600 CPU, a typical CISC processor architecture (Courtesy of Digital Equipment Corporation, 1985)

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.

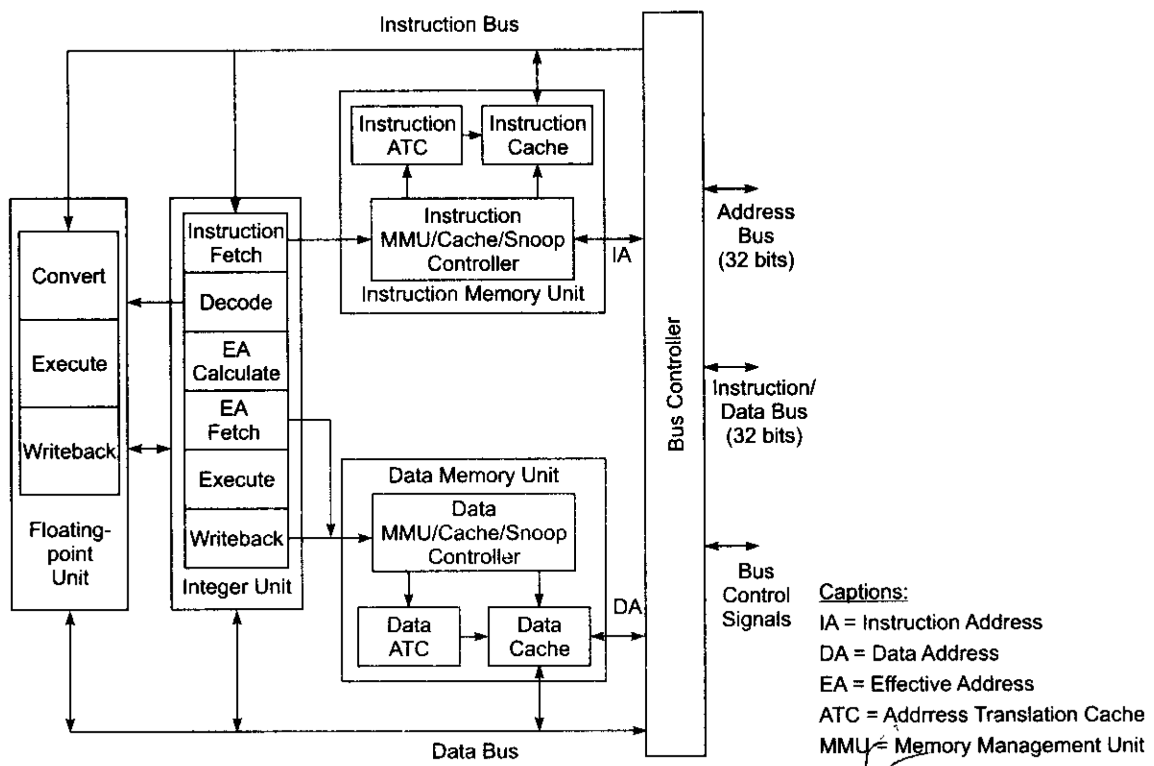


Fig. 4.6 Architecture of the MC68040 processor (Courtesy of Motorola Inc., 1991)

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 cacbe/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	0.8μm 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.

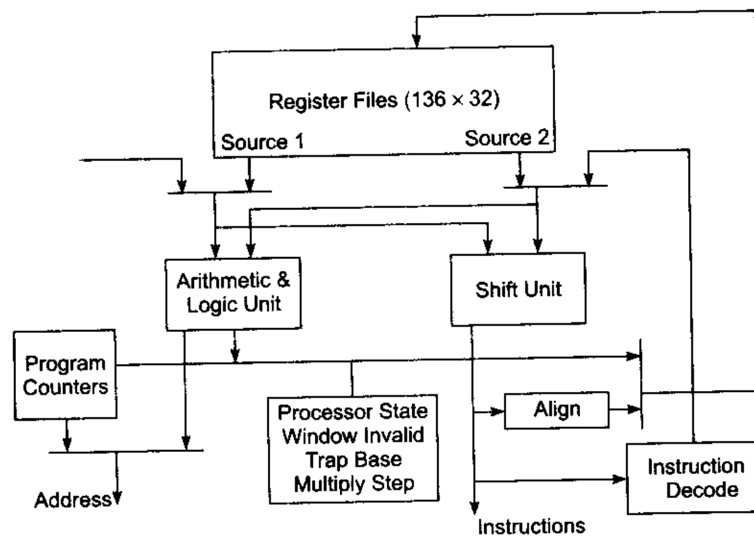
We choose to study the SPARC family chips produced by Cypress Semiconductors, Inc. Figure 4.7 shows the architecture of the Cypress CY7C601 SPARC processor and of the CY7C602 FPU. The Sun SPARC instruction set contains 69 basic instructions, a significant increase from the 39 instructions in the original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

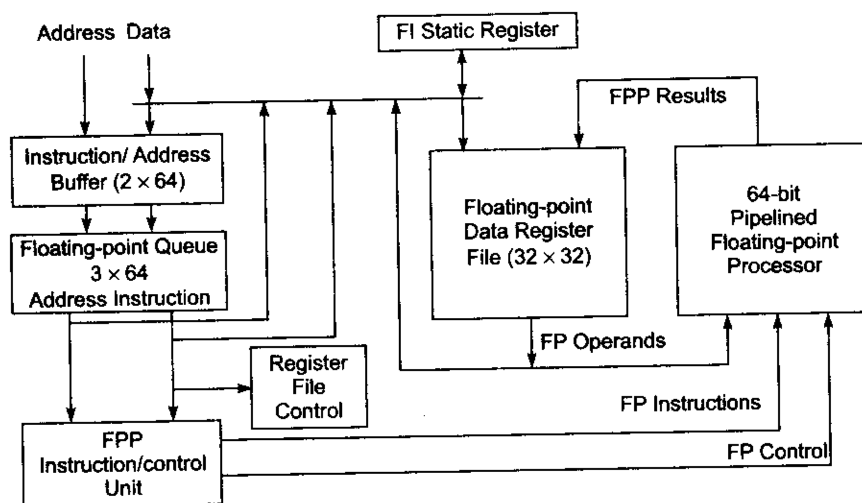
The concept is illustrated in Fig. 4.8 for eight overlapping windows (formed with 64 local registers and 64 overlapped registers) and eight globals with a total of 136 registers, as implemented in the Cypress 601.

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

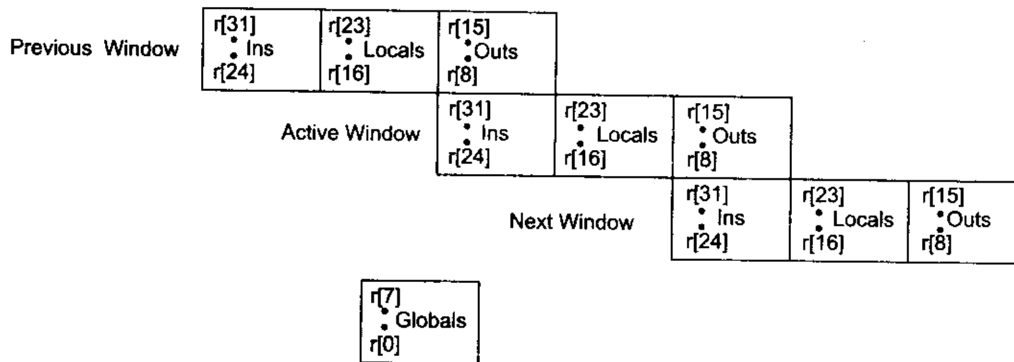


(b) The Cypress CY7C602 floating-point unit

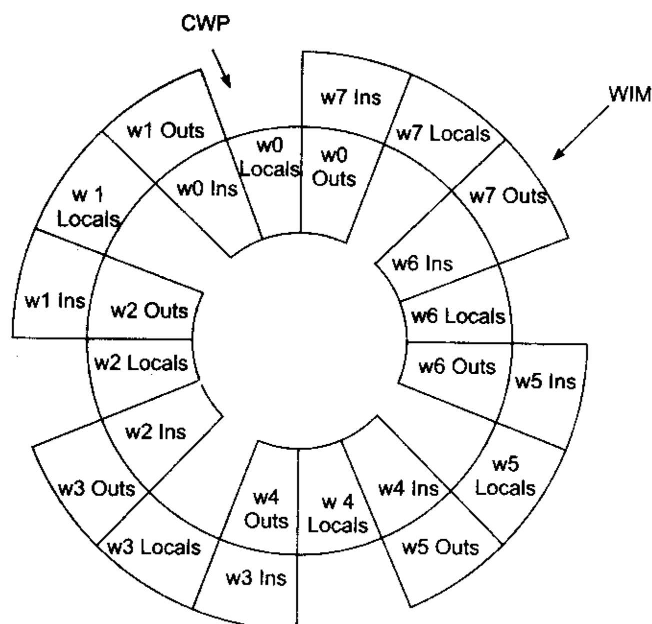
Fig. 4.7 The SPARC architecture with the processor and the floating-point unit on two separate chips (Courtesy of Cypress Semiconductor Co., 1991)

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.

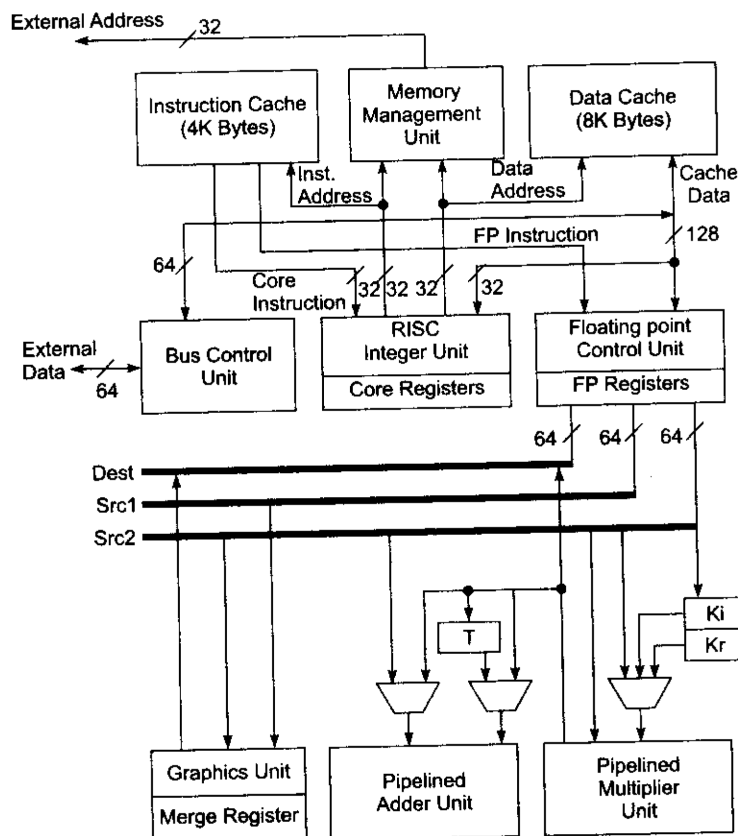


Fig. 4.9 Functional units and data paths of the Intel i860 RISC microprocessor (Courtesy of Intel Corporation, 1990)

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).

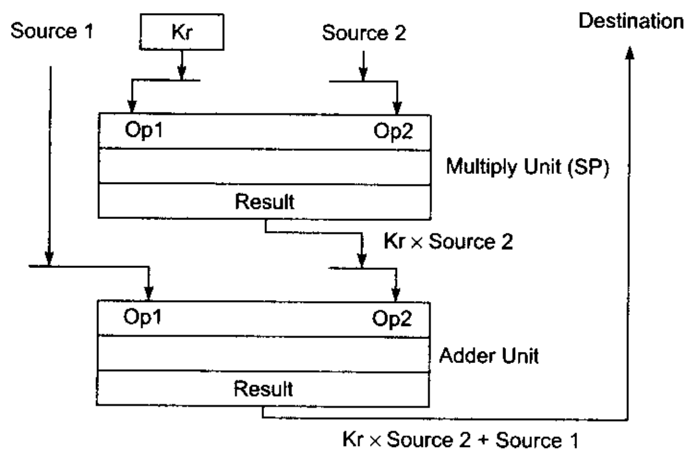


Fig. 4.10 Dual floating-point operations in the i860 processor

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.

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.

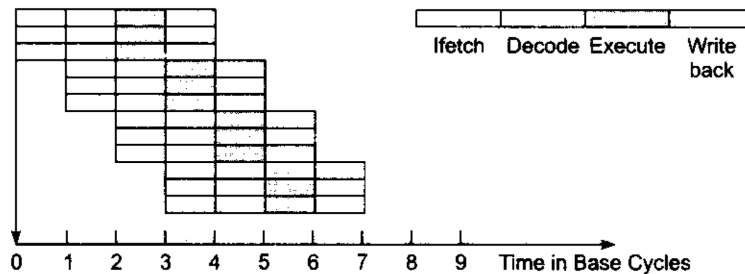


Fig. 4.11 A superscalar processor of degree m = 3

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	1-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.

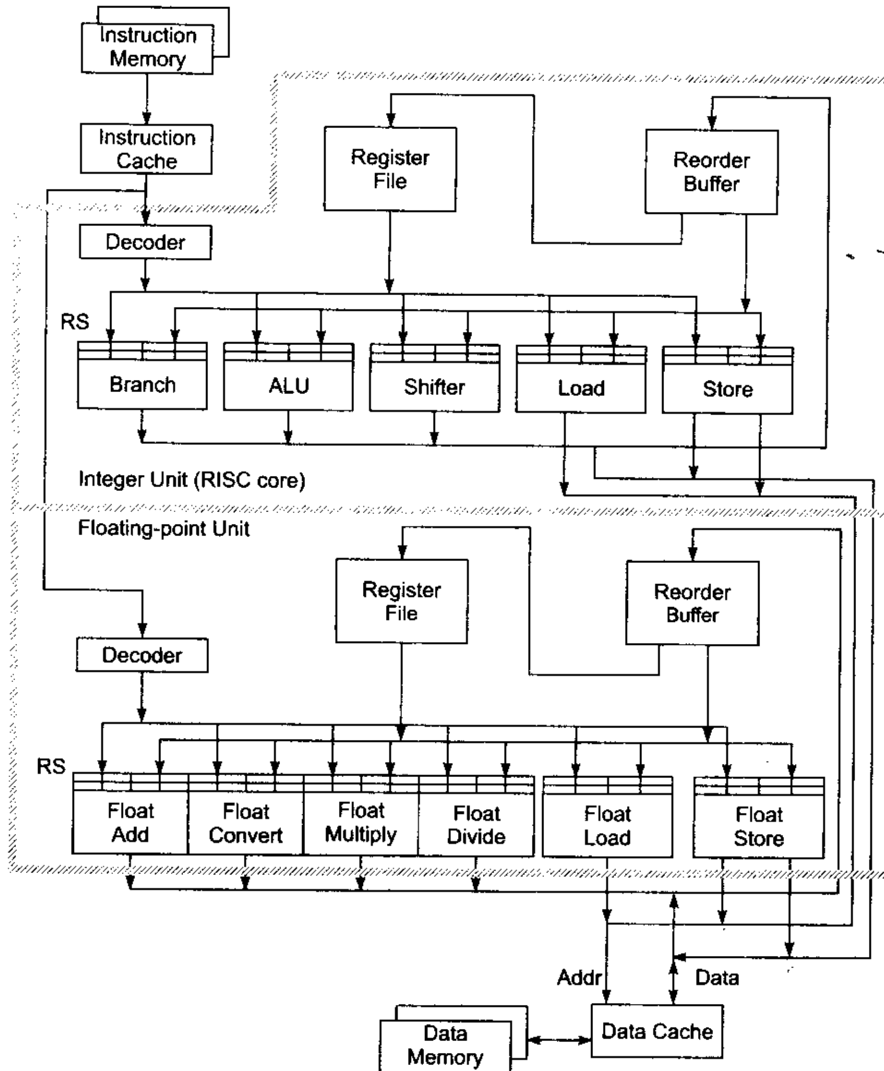


Fig. 4.12 A typical superscalar RISC processor architecture consisting of an integer unit and a floating-point unit (Courtesy of M. Johnson, 1991; reprinted with permission from Prentice-Hall, Inc.)

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.

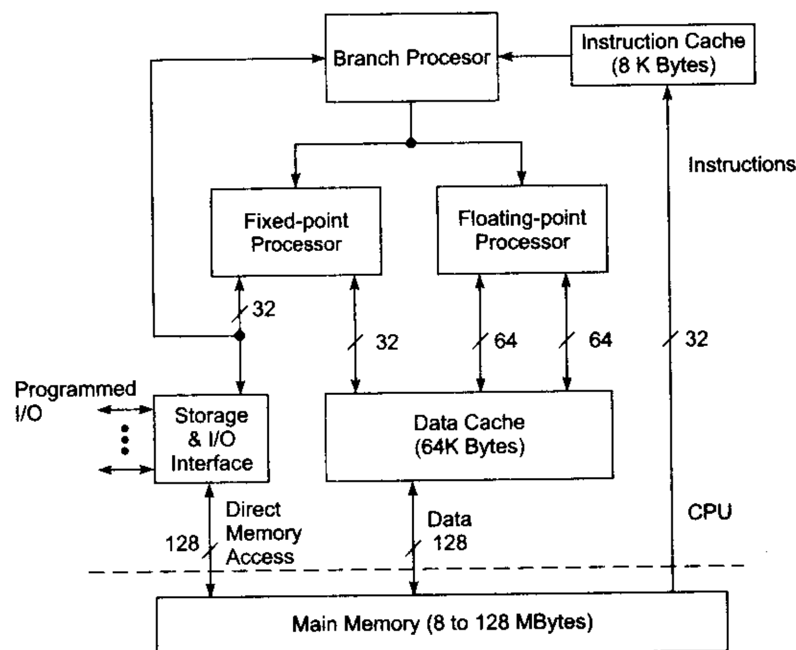


Fig. 4.13 The POWER architecture of the IBM RISC System/6000 superscalar processor (Courtesy of International Business Machines Corporation, 1990)

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.

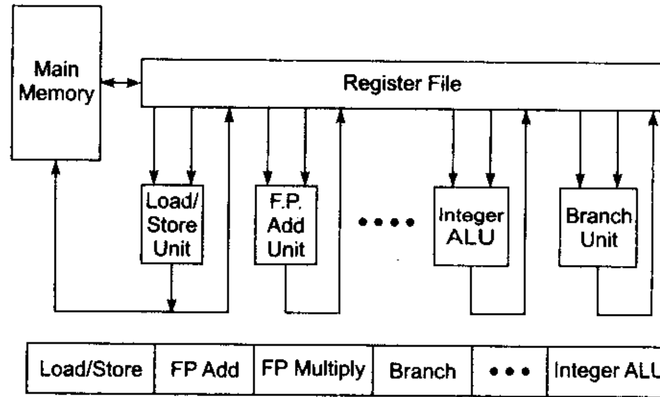
The RS/6000 design was optimized to perform well in numerically intensive scientific and engineering applications, as well as in multiuser commercial environments. A number of RS/6000-based workstations and servers were produced by IBM. For example, the POWERstation 530 had a clock rate of 25 MHz with 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

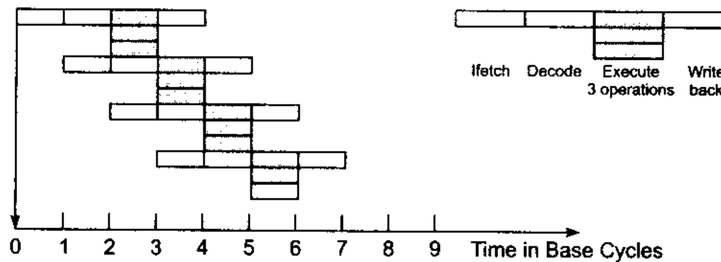
The VLIW architecture is generalized from two well-established concepts: horizontal microcoding and superscalar processing. A typical VLIW (*very long instruction word*) machine has instruction words hundreds of bits in length. As illustrated in Fig. 4.14a, multiple functional units are used concurrently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

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

$$\begin{array}{lcll} M(1:n) & \rightarrow & \mathbf{V}_1 & \text{(vector load)} \\ \mathbf{V}_1 & \rightarrow & M(1:n) & \text{(vector store)} \\ \mathrm{o}\ \mathbf{V}_1 & \rightarrow & \mathbf{V}_2 & \text{(unary vector)} \\ \mathrm{o}\ \mathbf{V}_1 & \rightarrow & s_1 & \text{(unary reduction)} \end{array} \quad (4.1)$$

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:

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

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.

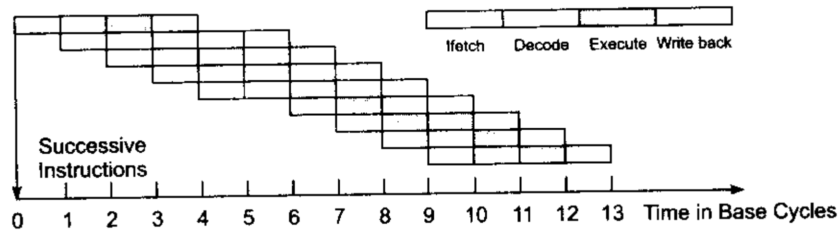
Vector Pipelines Vector processors take advantage of unrolled-loop-level parallelism. The vector pipelines can be attached to any scalar or superscalar processor.

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.

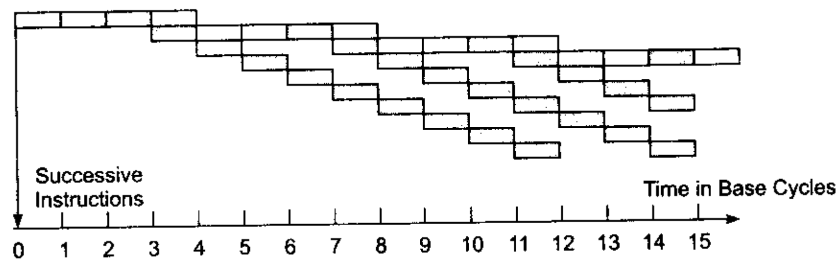
The pipelined execution in a vector processor is compared with that in a scalar processor in Fig. 4.15. Figure 4.15a is a redrawing of Fig. 4.2a in which each scalar instruction executes only one operation over one data element. For clarity, only serial issue and parallel execution of vector instructions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



(a) Scalar pipeline execution (Fig. 4.2a redrawn)



(b) Vector pipeline execution

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.

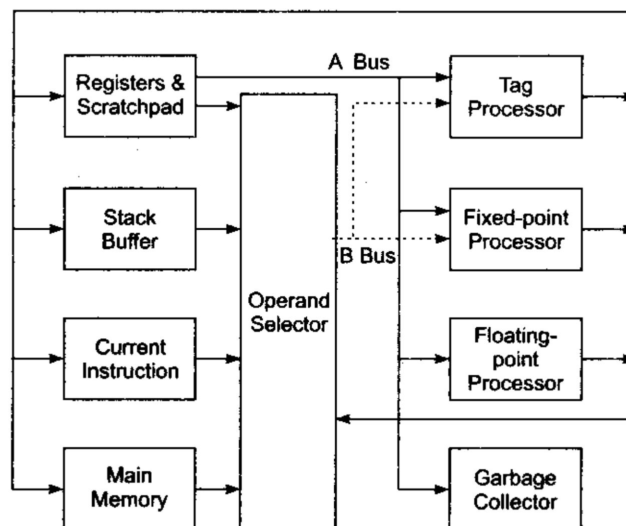


Fig. 4.16 The architecture of the Symbolics 3600 Lisp processor (Courtesy of Symbolics, Inc., 1985)

[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 $i + 1$.

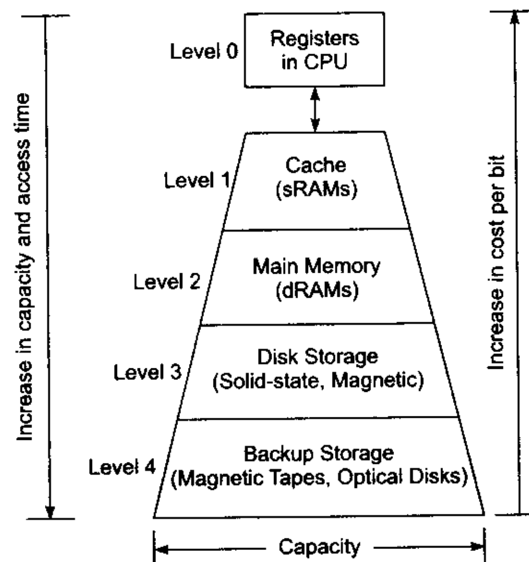


Fig. 4.17 A four-level memory hierarchy with increasing capacity and decreasing speed and cost from low to high levels

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.

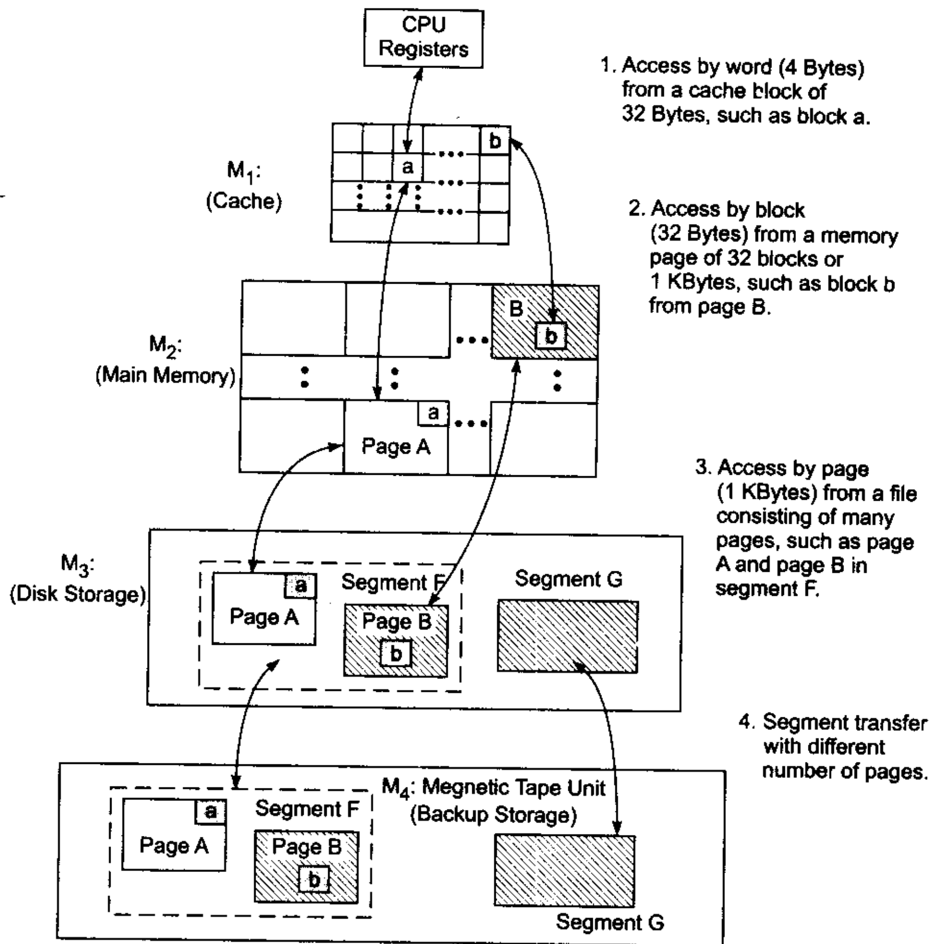


Fig. 4.18 The inclusion property and data transfers between adjacent levels of a memory hierarchy

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 $i = 1, 2, \dots, n-1$.

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:

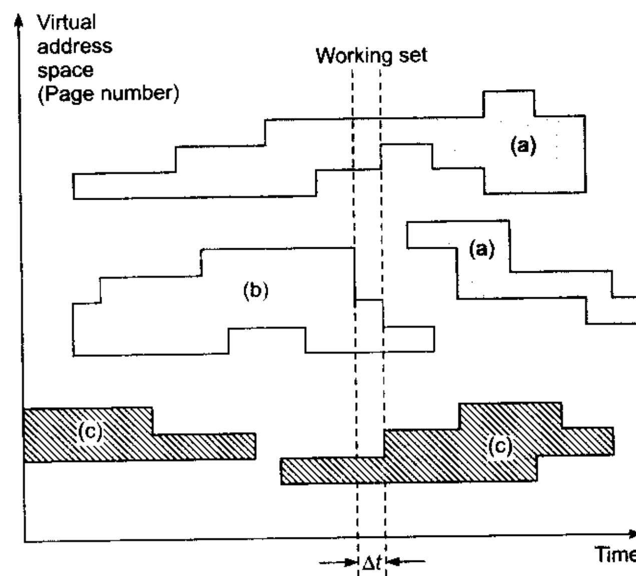


Fig. 4.19 Memory reference patterns in typical program trace experiments, where regions (a), (b), and (c) are generated with the execution of three software processes

(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:

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

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:

$$C_{\text{total}} = \sum_{i=1}^{n} c_i \cdot s_i \tag{4.4}$$

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

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

subject to the following constraints:

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

$$C_{\text{total}} = \sum_{i=1}^{n} c_i \cdot s_i < C_0 \tag{4.6}$$

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 Main memory Disk array	$t_1 = 25$ ns t_2 = unknown $t_3 = 4$ ms	$s_1 = 512$ Kbytes $s_2 = 32$ Mbytes s_3 = unknown	$c_1 = \$0.12$ $c_2 = \$0.02$ $c_3 = \$0.00002$

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:

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

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.

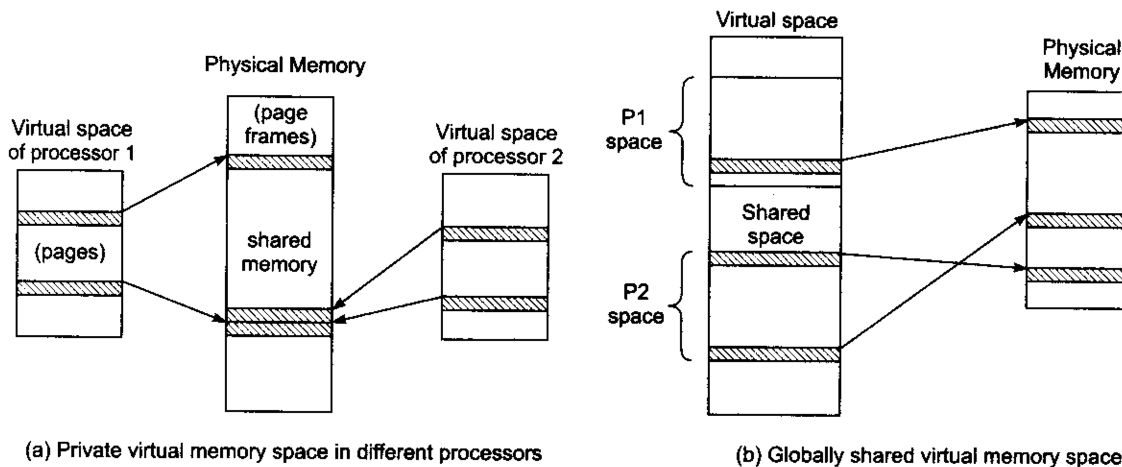


Fig. 4.20 Two virtual memory models for multiprocessor systems (Courtesy of Dubois and Briggs, tutorial, *Annual Symposium on Computer Architecture*, 1990)

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.

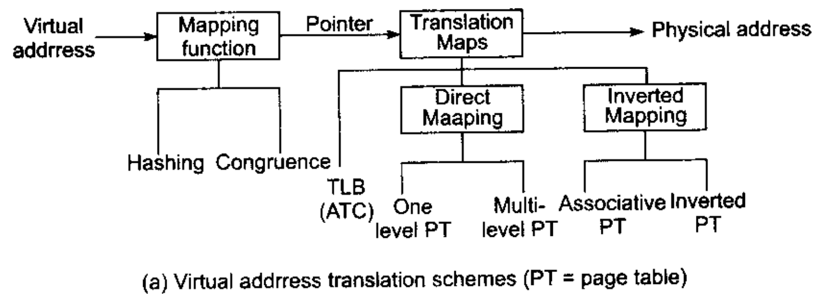
4.4.2 TLB, Paging, and Segmentation

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.

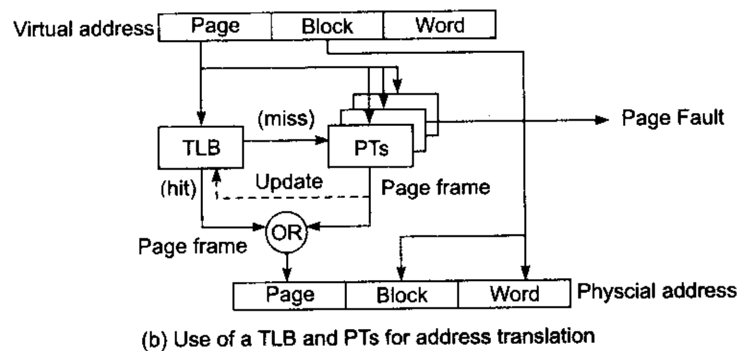
Address Translation Mechanisms The process demands the translation of virtual addresses into physical addresses. Various schemes for virtual address translation are summarized in Fig. 4.21a. The translation demands the use of *translction maps* which can be implemented in various ways.

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.

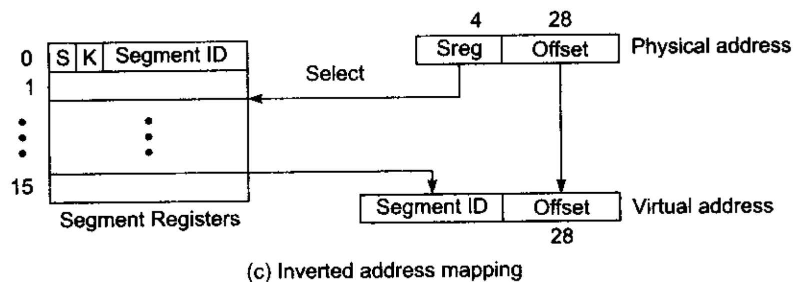
Hashing is a simple computer technique for converting a long page number into a short one with fewer bits. The hashing function should randomize the virtual page number and produce a unique hashed number to be used as the pointer.



(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.

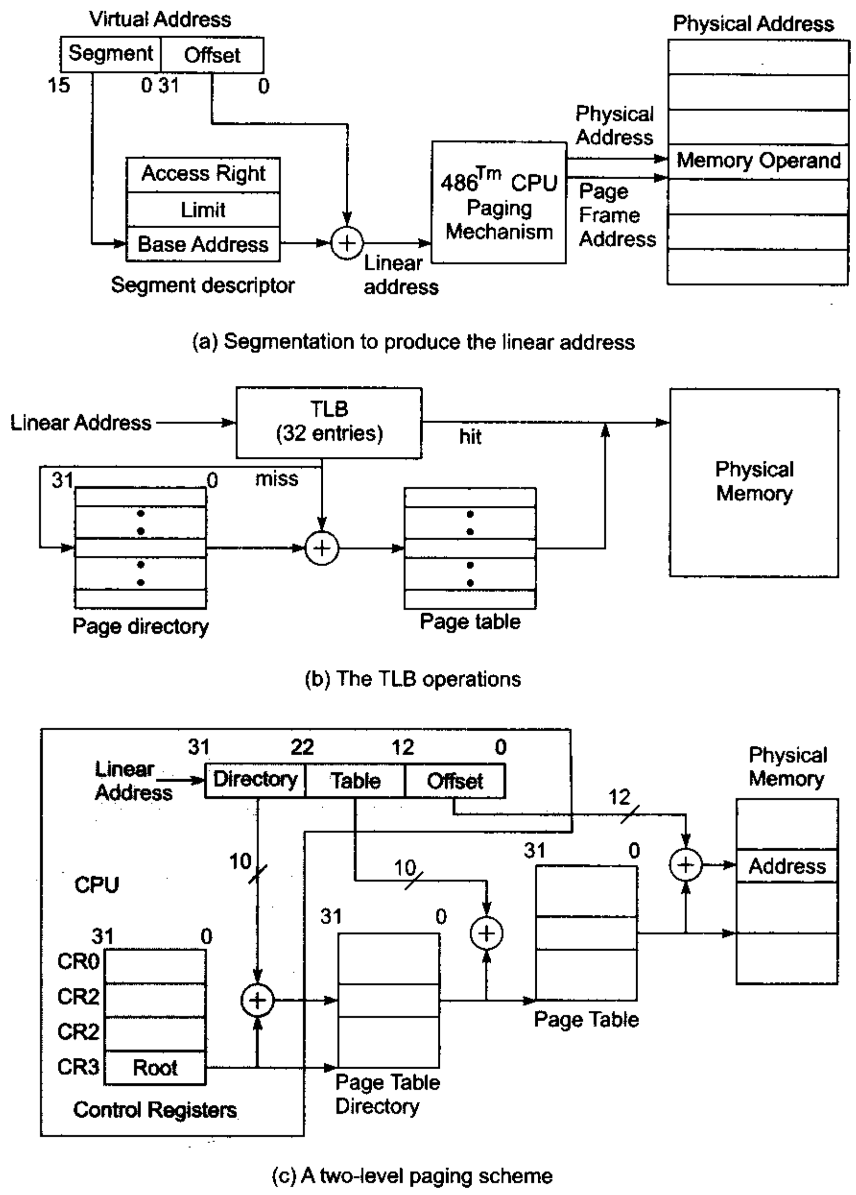


Fig. 4.22 **Paging and segmentation mechanisms built into the Intel i486 CPU (Courtesy of Intel Corporation, 1990)**

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	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	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	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.

5
Bus, Cache, and Shared Memory

This chapter describes the design and operational principles of bus, cache, and shared-memory organization. Backplane bus systems are studied, including features of VME, Futurebus+ and other bus specifications. Cache addressing models and implementation schemes are described. We study memory interleaving, allocation schemes, and the sequential and weak consistency models for shared-memory systems. Other relaxed memory consistency models are given in Chapter 9.

5.1 BUS SYSTEMS

The system bus of a computer system operates on a contention basis. Several active devices such as processors may request use of the bus at the same time. However, only one of them can be granted access at a time. The *effective bandwidth* available to each processor is inversely proportional to the number of processors contending for the bus.

For this reason, most bus-based commercial multiprocessors have been small in size. The simplicity and low cost of a bus system made it attractive in building small multiprocessors ranging from 4 to 16 processors. We shall see in Chapter 13 that advances in interconnect technologies have had a major impact on multiprocessor architecture.

In this section, we specify system buses which are confined to a single computer system. We concentrate on logical specification instead of physical implementation. Standard bus specifications should be both technology-independent and architecture independent.

5.1.1 Backplane Bus Specification

A backplane bus interconnects processors, data storage, and peripheral devices in a tightly coupled hardware configuration. The system bus must be designed to allow communication between devices on the bus without disturbing the internal activities of all the devices attached to the bus. Timing protocols must be established to arbitrate among multiple requests. Operational rules must be set to ensure orderly data transfers on the bus.

Signal lines on the backplane are often functionally grouped into several buses as depicted in Fig. 5.1. The four groups shown here are very similar to those proposed in the 64-bit VME bus specification (VITA, 1990).

Various functional boards are plugged into slots on the backplane. Each slot is provided with one or more connectors for inserting the boards as demonstrated by the vertical arrows in Fig. 5.1. For example, one or two 96-pin connectors are used per slot on the VME backplane.

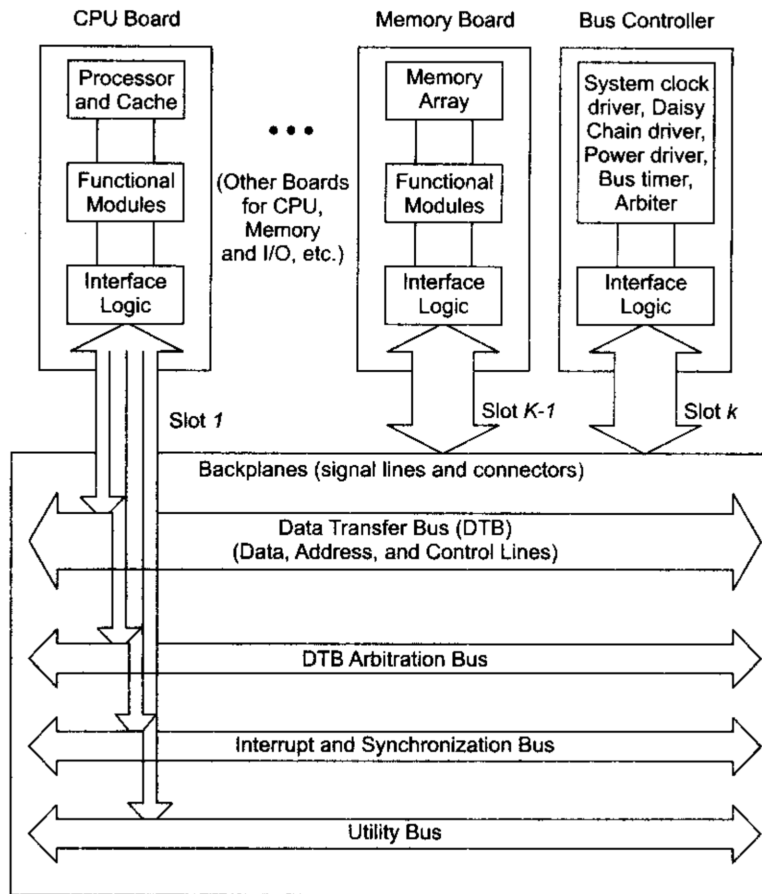


Fig. 5.1 **Backplane buses, system interfaces, and slot connections to various functional boards in a multiprocessor system**

Data Transfer Bus Data, address, and control lines form the *data transfer bus* (DTB) in a VME bus. The addressing lines are used to broadcast the data and device address. The number of address lines is proportional to the logarithm of the size of the address space. Address modifier lines can be used to define special addressing modes. The data lines are often proportional to the memory word length.

For example, the revised VME bus specification has 32 address lines and 32 (or 64) data lines. Besides being used in addressing, the 32 address lines can be multiplexed to serve as the lower half of the 64-bit data during data transfer cycles. The DTB control lines are used to indicate read/write, timing control, and bus error conditions.

Bus Arbitration and Control The process of assigning control of the DTB to a requester is called *arbitration*. Dedicated lines are reserved to coordinate the arbitration process among several requesters. The requester is called a *master*, and the receiving end is called a *slave*.

Interrupt lines are used to handle interrupts, which are often prioritized. Dedicated lines may be used to synchronize parallel activities among the processor modules. Utility lines include signals that provide periodic timing (clocking) and coordinate the power-up and power-down sequences of the system.

The backplane is made of signal lines, power lines, and connectors. A special bus controller board is often used to house the backplane control logic, such as the system clock driver, arbiter, bus timer, and power driver.

Functional Modules A *functional module* is a collection of electronic circuitry that resides on one functional board (Fig. 5.1) and works to achieve special bus control functions. Special functional modules are introduced below:

An *arbiter* is a functional module that accepts bus requests from the requester module and grants control of the DTB to one requester at a time.

A *bus timer* measures the time each data transfer takes on the DTB and terminates the DTB cycle if a transfer takes too long.

An *interrupter* module generates an interrupt request and provides status /ID information when an *interrupt handler* module requests it.

A *location monitor* is a functional module that monitors data transfers over the DTB. A *power monitor* watches the status of the power source and signals when power becomes unstable.

A *system clock driver* is a module that provides a clock timing signal on the utility bus. In addition, *board interface logic* is needed to match the signal line impedance, the propagation time, and termination values between the backplane and the plug-in boards.

Physical Limitations Due to electrical, mechanical, and packaging limitations, only a limited number of boards can be plugged into a single backplane. Multiple backplane buses can be mounted on the same backplane chassis.

For example, the VME chassis can house one to three backplane buses. Two can be used as a shared bus among all processors and memory boards, and the third as a local bus connecting a host processor to additional memory and I/O boards. Means of extending a single-bus system to build larger multiprocessors will be studied in Section 7.1.1. The bus system is difficult to scale, mainly limited by contention and packaging constraints.

5.1.2 Addressing and Timing Protocols

There are two types of IC chips or printed-circuit boards connected to a bus: *active* and *passive*. Active devices like processors can act as bus masters or as slaves at different times. Passive devices like memories can act only as slaves.

The master can initiate a bus cycle, and the slaves respond to requests by a master. Only one master can control the bus at a time. However, one or more slaves can respond to the master's request at the same time.

Bus Addressing The backplane bus is driven by a digital clock with a fixed cycle time called the *bus cycle*. The bus cycle is determined by the electrical, mechanical, and packaging characteristics of the backplane.

The backplane is designed to have a limited physical size which will not skew information with respect to the associated strobe signals. To speed up the operations, cycles on parallel lines in different buses may overlap in time. Factors affecting the bus delay include the source's line drivers, the destination's receivers, slot capacitance, line length, and the bus loading effects (the number of boards attached).

Not all the bus cycles are used for data transfers. To optimize performance, the bus should be designed to minimize the time required for request handling, arbitration, addressing, and interrupts so that most bus cycles are used for useful data transfer operations.

Each device can be identified with a *device number*. When the device number matches the contents of high-order address lines, the device is selected as a slave. This addressing allows the allocation of a logical device address under software control, which increases the application flexibility.

Broadcall and Broadcast Most bus transactions involve only one master and one slave. However, a *broadcall* is a read operation involving multiple slaves placing their data on the bus lines. Special AND or OR operations over these data are performed on the bus from the selected slaves.

Broadcall operations are used to detect multiple interrupt sources. A *broadcast* is a write operation involving multiple slaves. This operation is essential in implementing multicache coherence on the bus.

Timing protocols are needed to synchronize master and slave operations. Figure 5.2 shows a typical timing sequence when information is transferred over a bus from a source to a destination. Most bus timing protocols implement such a sequence.

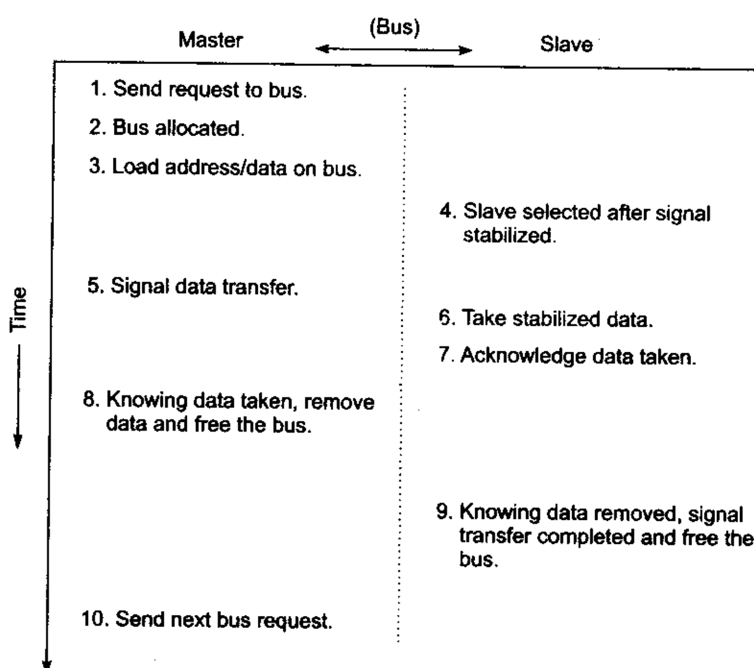


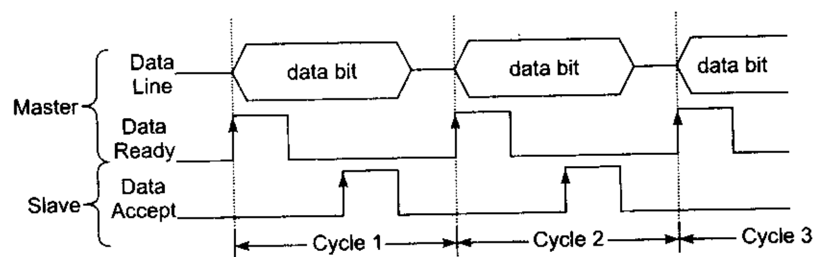
Fig. 5.2 Typical time sequence for information transfer between a master and a slave over a system bus

Synchronous Timing All bus transaction steps take place at fixed clock edges as shown in Fig. 5.3a. The clock signals are broadcast to all potential masters and slaves.

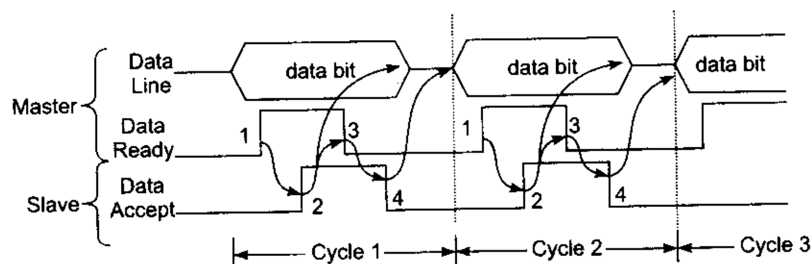
Once the data becomes stabilized on the data lines, the master uses a *data-ready* pulse to initiate the transfer. The slave uses a *data-accept* pulse to signal completion of the information transfer.

A synchronous bus is simple to control, requires less control circuitry, and thus costs less. It is suitable for connecting devices having relatively the same speed. Otherwise, the slowest device will slow down the entire bus operation.

Asynchronous Timing Asynchronous timing is based on a handshaking or interlocking mechanism as illustrated in Fig. 5.3b. No fixed clock cycle is needed. The rising edge (1) of the *data-ready* signal from the master triggers the rising (2) of the *data-accept* signal from the slave. The second signal triggers the falling (3) of the *data-ready* clock and the removal of data from the bus. The third signal triggers the trailing edge (4) of the *data-accept* clock. This four-edge handshaking (interlocking) process is repeated until all the data are transferred.



(a) Synchronous bus timing with fixed-length clock signals for all devices



(b) Asynchronous bus timing using a four-edge handshaking (interlocking with variable length signals for different speed devices.

Fig. 5.3 Synchronous versus asynchronous bus timing protocols

The advantage of using an asynchronous bus lies in the freedom of using variable length clock signals for different speed devices. This does not impose any response-time restrictions on the source and destination. It allows fast and slow devices to be connected on the same bus, and it is less prone to noise. Overall, an asynchronous bus offers better application flexibility at the expense of increased complexity and costs.

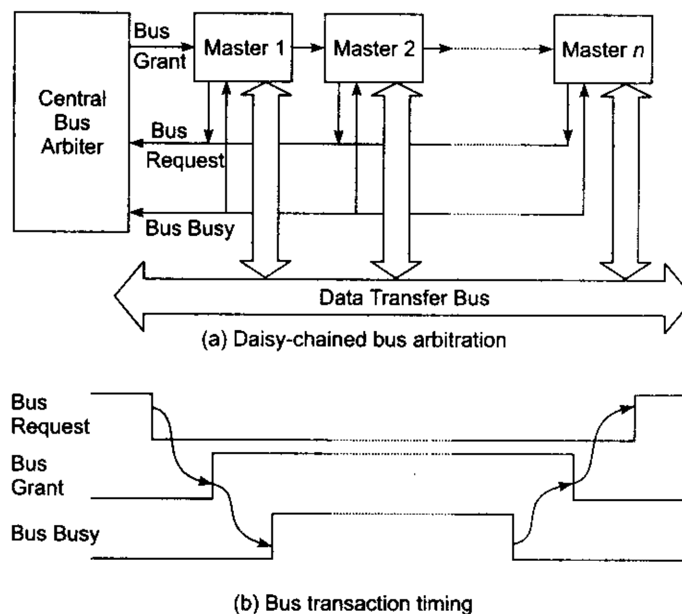
5.1.3 Arbitration, Transaction, and Interrupt

The process of selecting the next bus master is called *arbitration*. The duration of a master's control of the bus is called *bus tenure*. This arbitration process is designed to restrict tenure of the bus to one master at a time. Competing requests must be arbitrated on a fairness or priority basis.

Arbitration competition and bus transactions may take place concurrently on a parallel bus with separate lines for both purposes.

Central Arbitration As illustrated in Fig. 5.4a, a central arbitration scheme uses a central arbiter. Potential masters are daisy-chained in a cascade. A special signal line is used to propagate a *bus-grant* signal level from the first master (at slot 1) to the last master (at slot *n*).

Each potential master can send a bus request. However, all requests share the same *bus-request* line. As Shown in Fig. 5. 4b, the *bus-request* signals the rise of the *bus-grant* level, which in turn raises the *bus-busy* level.



(b) Bus transaction timing

Fig. 5.4 Central bus arbitration using shared requests and daisy-chained bus grants with a fixed priority

A fixed priority is set in a daisy chain from left to right. Only when the devices on the left do not request bus control can a device be granted bus tenure. When the bus transaction is complete, the *bus-busy* level is lowered, which triggers the falling of the *bus grant* signal and the subsequent rising of the *bus-request* signal.

The advantage of this arbitration scheme is its simplicity. Additional devices can be added anywhere in the daisy chain by sharing the same set of arbitration lines. The disadvantage is a fixed-priority sequence violating the fairness practice. Another drawback is its slowness in propagating the *bus-grant* signal along the daisy chain.

Whenever a higher-priority device fails, all the lower-priority devices on the right of the daisy chain cannot use the bus. Bypassing a failing device or a removed device on the daisy chain is desirable. Some new bus standards are specified with such a capability.

Independent Requests and Grants Instead of using shared request and grant lines as in Fig. 5.4, multiple *bus-request* and *bus-grant* signal lines can be independently provided for each potential master as in Fig. 5.5a. No daisy-chaining is used in this scheme, and the total number of signal lines required is larger.

The arbitration among potential masters is still carried out by a central arbiter. However, any priority-

based or fairness-based bus allocation policy can be implemented. A multiprocessor system usually uses a priority-based policy for I/O transactions and a fairness-based policy among the processors.

In some asymmetric multiprocessor architectures, the processors may be assigned different functions, such as serving as a front-end host, an executive processor, or a back-end slave processor. In such cases, a priority policy can also be used among the processors.

The advantage of using independent requests and grants in bus arbitration is their flexibility and faster arbitration time compared with the daisy-chained policy. The drawback is the large number of arbitration lines used.

Distributed Arbitration The idea of using distributed arbiters is depicted in Fig. 5.5b. Each potential master is equipped with its own arbiter and a unique *arbitration number*. The arbitration number is used to resolve the arbitration competition. When two or more devices compete for the bus, the winner is the one whose arbitration number is the largest.

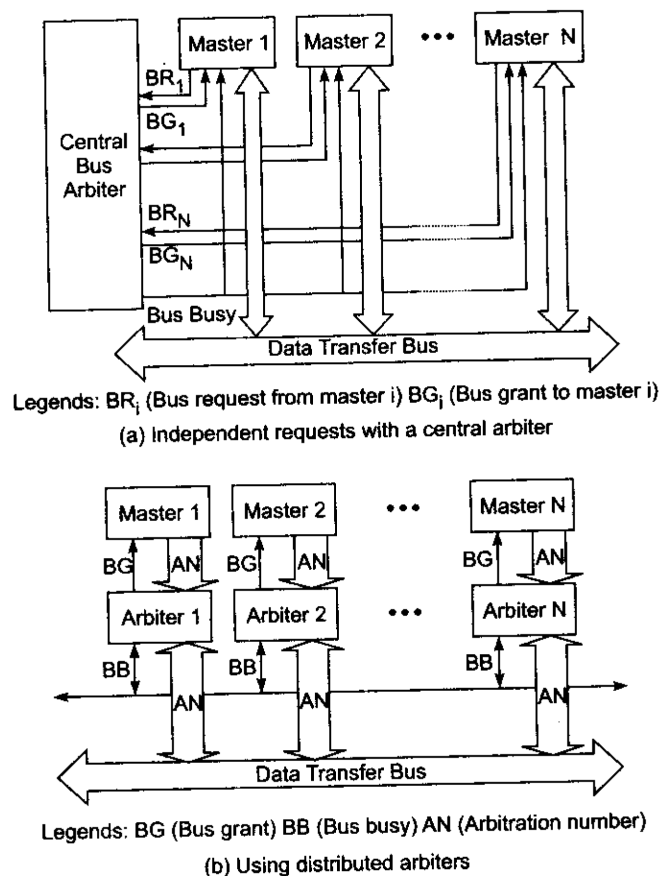


Fig. 5.5 Two bus arbitration schemes using independent requests and distributed arbiters, respectively

Parallel contention arbitration is used to determine which device has the highest arbitration number. All potential masters can send their arbitration numbers to the shared-bus request/grant (SBRG) lines on the arbitration bus via their respective arbiters.

Each arbiter compares the resulting number on the SBRG lines with its own arbitration number. If the SBRG number is greater, the requester is dismissed. At the end, the winner's arbitration number remains on the arbitration bus. After the current bus transaction is completed, the winner seizes control of the bus.

Clearly, the distributed arbitration policy is priority-based. Multibus II and Futurebus+ adopted such a distributed arbitration scheme. Besides distributed arbiters, Futurebus+ standard also provided options for a separate central arbiter.

Transaction Modes An *address-only transfer* consists of an address transfer followed by no data. A *compelled data transfer* consists of an address transfer followed by a block of one or more data transfers to one or more contiguous addresses. A *packet data transfer* consists of an address transfer followed by a fixed-length block of data transfers (*packet*) from a set of contiguous addresses.

Data transfers and priority interrupts handling are two classes of operations regularly performed on a bus. A bus transaction consists of a request followed by a response. A *connected transaction* is used to carry out a master's request and a slave's response in a single bus transaction. A *split transaction* splits the request and response into separate bus transactions. Three data transfer modes are specified below.

Split transactions allow devices with a long data latency or access time to use the bus resources in a more efficient way. A complete split transaction may require two or more connected bus transactions. Split transactions across multiple bus sequences are performed to achieve cache coherence in a large multiprocessor system.

Interrupt Mechanisms An *interrupt* is a request from I/O or other devices to a processor for service or attention. A priority interrupt bus is used to pass the interrupt signals. The interrupter must provide status and identification information. A functional module can be used to serve as an interrupt handler.

Priority interrupts are handled at many levels. For example, the VME bus uses seven *interrupt-request* lines. Up to seven interrupt handlers can be used to handle multiple interrupts.

Interrupts can also be handled by message-passing using the data bus lines on a time-sharing basis. The saving of dedicated interrupt lines is obtained at the expense of requiring some bus cycles for handling message-based interrupts.

The use of time-shared data bus lines to implement interrupts is called *virtual interrupt*. Futurebus+ was proposed without dedicated interrupt lines because virtual interrupts can be effectively implemented with the data transfer bus.

5.1.4 IEEE Futurebus+ and Other Standards

By the early 1990s, a large number of backplane bus standards were developed by various computer manufacturers in cooperation with the relevant IEEE standards committees. Among the well-known ones have been those for the VME bus (VITA and IEEE Standard 1014-1987), the Multibus II (Intel and IEEE Standard 1296-1987), the Nubus (Texas Instruments, 1983), the Fastbus (Gustavason, 1986), and the Nanobus by Encore Computer Systems.

Some of these buses have been used in building multiprocessors; however, each has had its own limitations. Most of them support only a data path of 32 bits, and none of them support an efficient cache coherence protocol or fast interprocessor synchronization.

The Futurebus+ standard was being developed under the cooperative effort of the VME International Trade Association, Multibus Manufacturers Group, U.S. Navy Next Generation Computer Resources Program, IEEE Microcomputer Standards Committee, and experts from companies and universities.

The objective was to develop a truly open bus standard that could support a 64-bit address space and the throughput required by multi-RISC or future generations of multiprocessor architectures.

The standards must be expandable upward or scalable and be independent of particular architecture and processor technologies. The key features of the IEEE Futurebus+ Standard 896.1-1991 are presented below.

Standard Requirements The major objectives of the Futurebus+ standards committee were to create a bus standard that would provide a significant step forward in improving the facilities and performance available to the designers of multiprocessor systems. The aim was to provide a stable platform on which several generations of computer systems could be based. Summarized below are design requirements set by the IEEE 896.1-1991 Standards Committee:

(1) Architecture-, processor-, and technology-independent open standard available to all designers.
(2) A fully asynchronous (compelled) timing protocol for data transfer with hand-shaking flow control.
(3) An optional source-synchronized (packet) protocol for high-speed block data transfers.
(4) Fully distributed parallel arbitration protocols to support a rich variety of bus transactions including broadcast, broadcall, and three-party transactions.
(5) Support of high reliability and fault-tolerant applications with provisions for live card insertion/removal, parity checks on all lines and feedback checking, and no daisy-chained signals to facilitate dynamic system reconfiguration in the event of module failure.
(6) Use of multilevel mechanisms for the locking of modules and avoidance of deadlock or livelock.
(7) Circuit-switched and split transaction protocols plus support for memory commands for implementing remote lock and SIMD-like operations.
(8) Support of real-time mission-critical computations with multiple priority levels and consistent priority treatment, plus support of a distributed clock synchronization protocol.
(9) Support of 32- or 64-bit addressing with dynamically sized data buses from 32 to 64, 128, and 256 bits to satisfy different bandwidth demands.
(10) Direct support of snoopy cache-based multiprocessors with recursive protocols to support large systems interconnected by multiple buses.
(11) Compatible message-passing protocols with multicomputer connections and special application profiles and interface design guides provided.

Over the last three decades, clock speeds and device densities of processor chips have increased exponentially. Parallel and cluster computing systems today employ a much larger number of processors than the systems of two decades ago. The net result of these factors has been a huge increase in bandwidth demands both within the computer system and in terms of communication with external devices and networks.

The complex Futurebus+ architecture described above could not satisfy the rapidly increasing bandwidth and efficiency demands of newer systems, and thereby the basic performance limitations of bus-based systems also became clear. The need was to support high performance distributed and cluster computing with high bandwidth, low latency, and a scalable architecture to allow building large systems using inexpensive building blocks. To meet these requirements, Scalable Coherent Interface (SCI) and InfiniBand came up as simpler and more efficient offshoots of the Futurebus+ standard.

Low latency is important for efficient distributed computing, and therefore protocols must work with low overheads. Every SCI and InfiniBand node comes with its own links, so that aggregate bandwidth increases with the number of nodes, and thus the system remains scalable. Single link bandwidths are in GBytes/sec; with switched interconnects, a variety of topologies and speeds is supported, and media-independent protocols support a mix of copper and optical fiber links.

SCI was developed to support the requirements of both internal system bus (between processor, memory and I/O subsystem), and the external network. The aim behind this initiative was to avoid the bottlenecks of physical buses, scale up to supercomputer performance, and support efficient parallel processing software.

With the use of point-to-point links and packet switching, SCI protocols were kept simple, so that interface chips could run fast, allowing scalable and distance-independent protocols. Physical packaging is not restricted to a bus backplane, and performance degrades only slowly as distance increases.

SCI provides distributed directory based cache coherence for a global shared memory model, and provides a degree of fault tolerance. For high-performance computing, it is employed to build NUMA computer clusters and other parallel architectures. Sun Microsystems has used SCI for all of their high-performance systems.

InfiniBand is another switched interconnect architecture which emerged from the Futurebus+ standard. Serial point-to-point links are used, with simpler, less expensive, more reliable and scalable architecture. Links can carry multiple channels of data at the same time in a packet-multiplexed manner, with throughputs of up to 2.5 GBytes/sec.

Packet switching implies that control information determines the route a given packet or message follows from source to destination. InfiniBand uses Internet Protocol Version 6 (IPv6), allowing a vast range of system expansion. One or more packets are combined to form a message; a message can be a simple send or receive operation, remote direct memory access operation, a transaction, or a multicast transmission.

Technology/Architecture Independence Any bus standard should aim to achieve *technology independence* through basing the protocols on fundamental principles and optimizing them for maximum communication efficiency rather than a particular generation or type of processor. Timing and handshake protocols should be governed by operational constraints rather than limitations of technology such as device delays and capture windows.

The standard specification may be implemented with any logic family, provided that physical implementation meets the signaling requirements.

Architecture independence should provide a flexible general-purpose solution to cache consistency within which other cache protocols operate compatibly while at the same time providing an elegant unification with the message-passing protocols used in a multicomputer environment. Such architecture independence increases the application flexibility of a multiprocessor system built around the bus standard. Other bus standards PCI, PCI Express and HyperTransport are described in brief in Chapter 13.

5.2 CACHE MEMORY ORGANIZATIONS

This section deals with physical address caches, virtual address caches, cache implementation using direct, fully associative, set-associative, and sector mapping. Finally, cache performance issues are analyzed based on some reported trace results. Multicache coherence protocols will be studied in Chapter 7.

5.2.1 Cache Addressing Models

Most multiprocessor systems use private caches associated with different processors as depicted in Fig. 5.6. Caches can be addressed using either a physical address or a virtual address. This leads to the two different cache design models presented below.

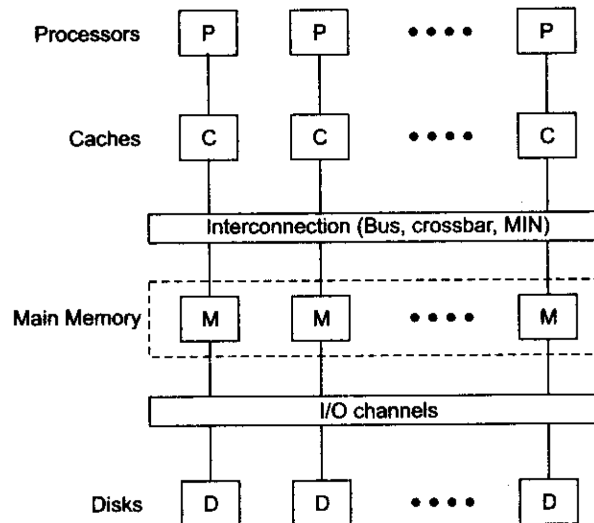


Fig. 5.6 A memory hierarchy for a shared-memory multiprocessor

Physical Address Caches When a cache is accessed with a physical memory address, it is called a *physical address cache*. Physical address cache models are illustrated in Fig. 5.7. In Fig. 5.7a, the model is based on the experience of using a unified cache as in the VAX 8600 and the Intel i486.

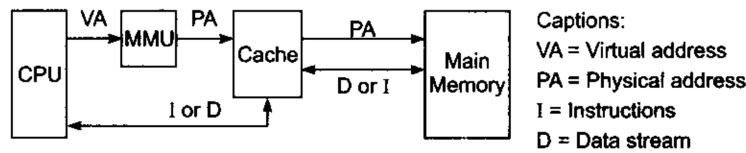
In this case, the cache is indexed and tagged with the physical address. Cache lookup must occur after address translation in the TLB or MMU.

A *cache hit* occurs when the addressed data/instruction is found in the cache. Otherwise, a *cache miss* occurs. After a miss, the cache is loaded with the data from the memory. Since a whole cache block is loaded at one time, unwanted data may also be loaded. Locality of references will find most of the loaded data useful in subsequent instruction cycles.

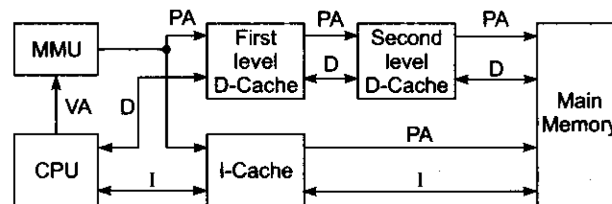
Data is written through the main memory immediately via a *write-through* (WT) cache, or delayed until block replacement by using a *write-back* (WB) cache. A WT cache requires more bus or network cycles to access the main memory, while a WB cache allows the CPU to continue without waiting for the memory to cycle.

Example 5.1 Cache design in a Silicon Graphics workstation

Figure 5.7b demonstrates the split cache design using the MIPS R3000 CPU in the Silicon Graphics 4-D Series workstation. Both data cache and instruction cache are accessed with a physical address issued from the on-chip MMU. A two-level data cache is implemented in this design.



(a) A unified cache accessed by physical address



(b) Split caches accessed by physical address in the Silicon Graphics workstation

Fig. 5.7 Physical address models for unified and split caches

The first level uses 64 Kbytes of WT D-cache. The second level uses 256 Kbytes of WB D-cache. The single-level I-cache is 64 Kbytes. By the inclusion property, the first-level cache is always a subset of the second-level cache.

The major advantages of physical address caches include no need to perform cache flushing, no aliasing problems, and thus fewer cache bugs in the OS kernels. The shortcoming is the slowdown in accessing the cache until the MMU /TLB finishes translating the address. This motivates the use of a virtual address cache. Integration of the MMU and caches on the same VLSI chip can alleviate some of these problems.

Most conventional system designs use a physical address cache because of its simplicity and because it requires little intervention from the OS kernel. When physical address caches are used in a UNIX environment, no flushing of data caches is needed if bus watching is provided to monitor the system bus for DMA requests from I/O devices or from other CPUs. Otherwise, the cache must be flushed for every I/O without proper bus watching.

Virtual Address Caches When a cache is indexed or tagged with a virtual address as shown in Fig. 5.8, it is called a *virtual address* cache. In this model, both cache and MMU translation or validation are done in parallel. The physical address generated by the MMU can be saved in tags for later write-back but is not used

during the cache lookup operations. The virtual address cache is motivated with its enhanced efficiency to access the cache faster, overlapping with the MMU translation as exemplified below.

Example 5.2 The virtual addressed split cache design in Intel i860

Figure 5.8b shows the virtual address design in the Intel i860 using split caches for data and instructions. Instructions are 32 bits wide. Virtual addresses generated by the integer unit (IU) are 32 bits wide, and so are the physical addresses generated by the MMU. The data cache is 8 Kbytes with a block size of 32 bytes. A two-way set-associative cache organization (Sec. 5.2.3) is implemented with 128 sets in the D-cache and 64 sets in the I-cache.

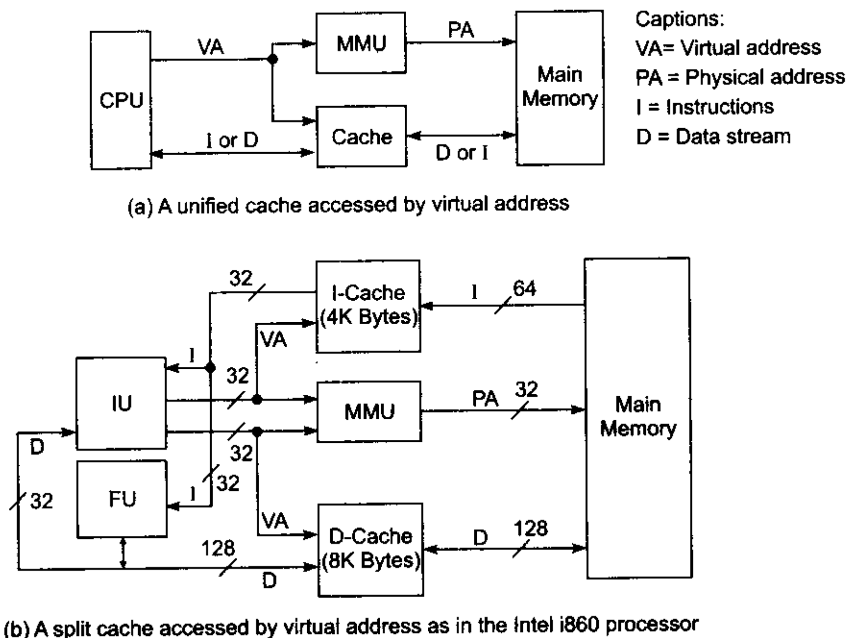


Fig. 5.8 Virtual address models for unified and split caches (Courtesy of Intel Corporation, 1989)

The Aliasing Problem The major problem associated with a virtual address cache is *aliasing*, when different logically addressed data have the same index/tag in the cache. Multiple processes will in general use the same range of virtual addresses. This aliasing problem may create confusion if two or more processes access the same physical cache location. One way to solve the aliasing problem is to flush the entire cache whenever a context switch occurs.

Large amounts of *flushing* may result in a poor cache performance, with a low hit ratio and too much time wasted in flushing. When a virtual address cache is used with UNIX, flushing is needed after each context

switching. Before I/O writes or after I/O reads, the cache must be flushed. Furthermore, aliasing between the UNIX kernel and user programs and data is a serious problem. All of these problems introduce additional system overhead.

Flushing the cache does not overcome the aliasing problem completely when using a shared memory with mapped files and copy-on-write as in UNIX systems. These UNIX operations may not benefit from virtual caches. In each entry/exit to or from the UNIX kernel, the cache must be flushed upon every system call and interrupt.

The virtual space must be divided between kernel and user modes by tagging kernel and user data separately. This implies that a virtual address cache may lead to a lower performance unless most processes are compute-bound.

With a frequently flushed cache, the debugging of programs is almost impossible to perform. Two commonly used methods to get around the virtual address cache problems are to apply special tagging with a *process key* or with a *physical address*. For example, the SUN 3/200 Series has used a virtual address, write-back cache with the capability of being noncacheable. Three-bit keys are used in the cache to distinguish among eight simultaneous contexts.

The flushing can be done at the page, segment, or context level. In this case, context switching does not need to flush the cache but needs to change the current process key. Thus cached shared memory must be shared on fixed-size boundaries. Other memory can be shared with noncacheability. Flushing and special tagging may be traded for performance/cost reasons.

5.2.2 Direct Mapping and Associative Caches

The transfer of information from main memory to cache memory is conducted in units of cache blocks or cache lines. Four block placement schemes are presented below. Each placement scheme has its own merits and shortcomings. The ultimate performance depends on the cache-access patterns, cache organization, and management policy used.

Blocks in caches are called *block frames* in order to distinguish them from the corresponding *blocks* in main memory. Block frames are denoted as $\bar{B}_i$ for $i = 0, 1, 2, \ldots, m$. Blocks are denoted as B_j for $j = 0, 1, 2, \ldots, n$. Various mappings can be defined from set $\{B_j\}$ to set $\{\bar{B}_i\}$. It is also assumed that $n >> m$, $n = 2^s$, and $m = 2^r$.

Each block (or block frame) is assumed to have b words, where $b = 2^w$. Thus the cache consists of $m \cdot b = 2^{r+w}$ words. The main memory has $n \cdot b = 2^{s+w}$ words addressed by $(s + w)$ bits. When the block frames are divided into $v = 2^t$ sets, $k = m/v = 2^{r-t}$ blocks are in each set.

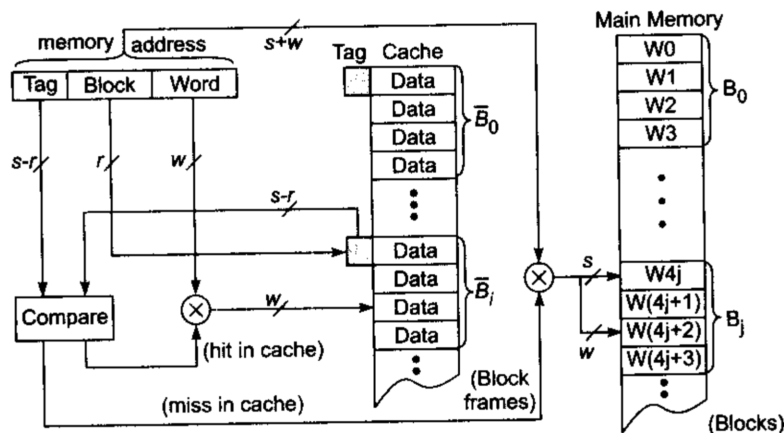
Direct-Mapping Cache This cache organization is based on a direct mapping of $n/m = 2^{s-r}$ memory blocks, separated by equal distances, to one block frame in the cache. The placement is defined below using a modulo-m function. Block B_j is mapped to block frame $\bar{B}_i$:

$$B_j \rightarrow \bar{B}_i, \qquad \text{if } i = j \text{ (modulo } m) \tag{5.1}$$

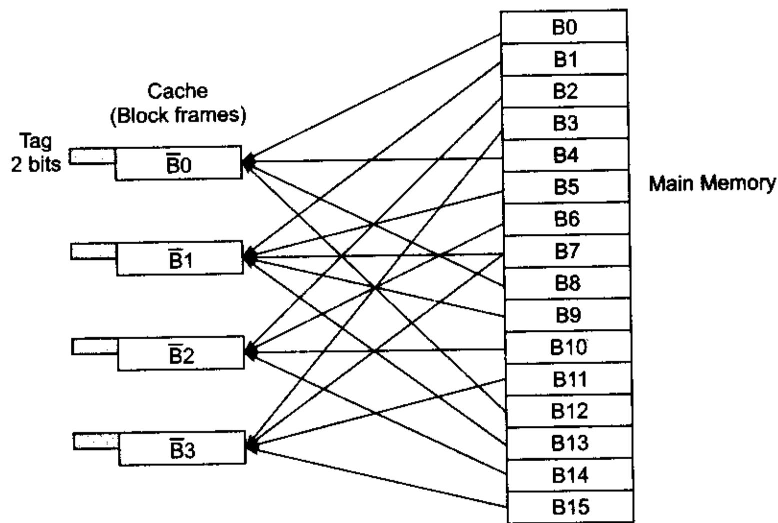
There is a *unique* block frame $\bar{B}_i$ that each B_j can load into. There is no way to implement a block replacement policy. This direct mapping is very rigid but is the simplest cache organization to implement. Direct mapping is illustrated in Fig. 5.9a for the case where each block contains four words ($w = 2$ bits).

The memory address is divided into three fields: The lower w bits specify the *word offset* within each block. The upper s bits specify the *block address* in main memory, while the leftmost $(s - r)$ bits specify the

tag to be matched. The *block* field (r bits) is used to implement the (modulo-m) placement, where $m = 2^r$. Once the block $\bar{B}_i$ is uniquely identified by this field, the tag associated with the addressed block is compared with the tag in the memory address.



(a) The cache/memory addressing



(b) Block B_j can be mapped to block frame $\bar{B}_i$ if $i = j$ (modulo 4)

Fig. 5.9 Direct-mapping cache organization and a mapping example

A cache hit occurs when the two tags match. Otherwise a cache miss occurs. In case of a cache hit, the word offset is used to identify the desired data word within the addressed block. When a miss occurs, the entire memory address ($s + w$ bits) is used to access the main memory. The first s bits locate the addressed block, and the lower w bits locate the word within the block.

Example 5.3 Direct-mapping cache design and block mapping

An example mapping is given in Fig. 5.9b, where $n = 16$ blocks are mapped to $m = 4$ block frames, with four possible sources mapping into one destination using modulo-4 mapping.

Cache Design Parameters

In practice, the two parameters n and m differ by at least two to three orders of magnitude. A typical cache block has 32 bytes corresponding to eight 32-bit words. Thus $w = 3$ bits if the machine is word-addressable. If the machine is byte-addressable, then $w = 5$ bits.

Consider a cache with 64 Kbytes. This implies $m = 2^{11} = 2048$ block frames with $r = 11$ bits. Consider a main memory with 32 Mbytes. Thus $n = 2^{20}$ blocks with $s = 20$ bits, and the memory address needs $s + w = 20 + 3 = 23$ bits for word addressing and 25 bits for byte addressing. In this case, $2^{s-r} = 2^9 = 512$ blocks are possible candidates to be mapped into a single block frame in a direct-mapping cache.

Advantages of a direct-mapping cache include simplicity in hardware, no associative search needed, no page replacement algorithm needed, and thus lower cost and higher speed.

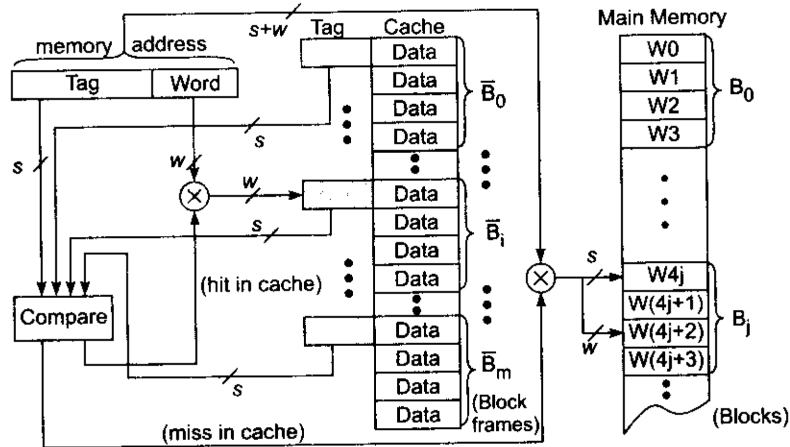
However, the rigid mapping may result in a poorer hit ratio than with the associative mappings to be introduced next. The scheme also prohibits parallel virtual address translation. The hit ratio may drop sharply if many addressed blocks have to map into the same block frame. For this reason, direct-mapped caches tend to use a larger cache size with more block frames to avoid the contention.

Fully Associative Cache Unlike direct mapping, this cache organization offers the most flexibility in mapping cache blocks. As illustrated in Fig. 5.10a, each block in main memory can be placed in any of the available block frames. Because of this flexibility, an s-bit tag is needed in each cache block. As $s > r$, this represents a significant increase in tag length.

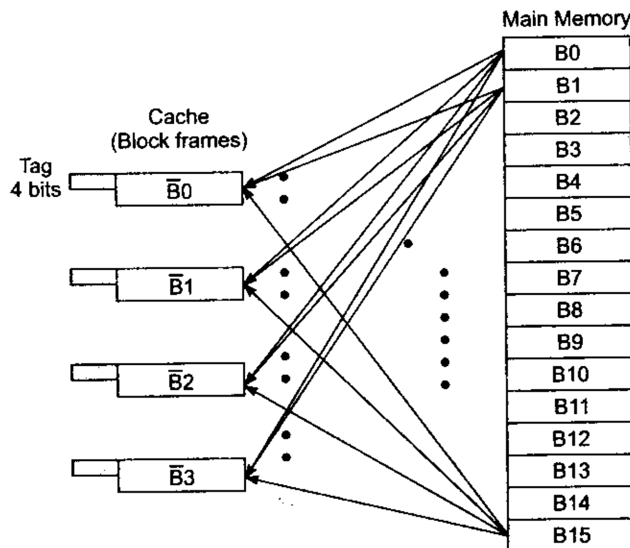
The name *fully associative cache* is derived from the fact that an m-way associative search requires the tag to be compared with all block tags in the cache. This scheme offers the greatest flexibility in implementing block replacement policies for a higher hit ratio.

The m-way comparison of all tags is very time-consuming if the tags are compared sequentially using RAMs. Thus an *associative memory* (content-addressable memory, CAM) is needed to achieve a parallel comparison with all tags simultaneously. This demands a higher implementation cost for the cache. Therefore, a fully associative cache has been implemented only in moderate size.

Figure 5.10b shows a four-way mapping example using a fully associative search. The tag is 4 bits long because 16 possible cache blocks can be destined for the same block frame. The major advantage of using full associativity is to allow the implementation of a better block replacement policy with reduced block contention. The major drawback lies in the expensive search process requiring a higher hardware cost.



(a) Associative search with all block tags



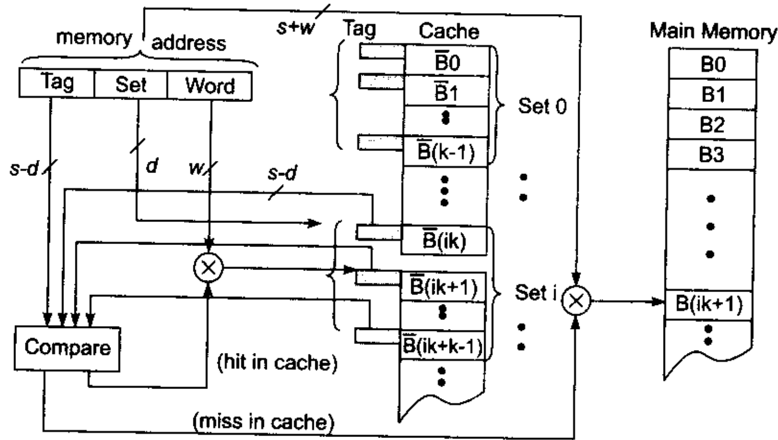
(b) Every block is mapped to any of the four block frames identified by the tag

Fig. 5.10 Fully associative cache organization and a mapping example

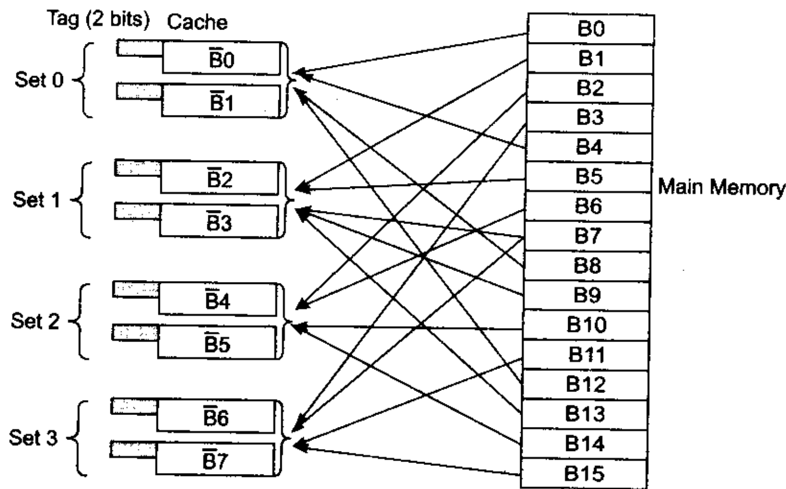
5.2.3 Set-Associative and Sector Caches

Set-associative caches are the most popular cache designs built into commercial computers. Sector mapping caches offer a design alternative to set-associative caches. These two types of cache design are described below.

Set-Associative Cache This design offers a compromise between the two extreme cache designs based on direct mapping and full associativity. If properly designed, this cache may offer the best performance-cost ratio. Most high-performance computer systems are based on this approach. The idea is illustrated in Fig. 5.11.



(a) A k-way associative search within each set of k each blocks



(b) Mapping cache blocks in a two-way associative cache wit four sets

Fig. 5.11 Set-associative cache organization and a two-way associative mapping example

In a *k-way associative* cache, the m cache block frames are divided into $v = m/k$ sets, with k blocks per set. Each set is identified by a d-bit *set number*, where $2^d = v$. The cache block tags are now reduced to $s - d$ bits. In practice, the *set size k*, or *associativity*, is chosen as 2, 4, 8, 16, or 64, depending on a tradeoff among block size w, cache size m, and other performance/cost factors.

Fully associative mapping can be visualized as having a single set (i.e. $v = 1$) or an m-way associativity. In a k-way associative search, the *tag* needs to be compared only with the k tags within the identified set, as shown in Fig. 5.11a. Since k is rather small in practice, the k-way associative search is much more economical than the full associativity.

In general, a block B_j can be mapped into any one of the available frames $\bar{B}_f$ in a set S_i defined below. The matched tag identifies the current block which resides in the frame.

$$B_j \rightarrow \bar{B}_f \in S_i \quad \text{if} \quad j(\text{modulo } v) = i \tag{5.2}$$

Design Tradeoffs The set size (associativity) k and the number of sets v are inversely related by

$$m = v \times k \tag{5.3}$$

For a fixed cache size there exists a tradeoff between the set size and the number of sets.

The advantages of the set-associative cache include the following:

First, the block replacement algorithm needs to consider only a few blocks in the same set. Thus the replacement policy can be more economically implemented with limited choices, as compared with the fully associative cache.

Second, the k-way associative search is easier to implement, as mentioned earlier. Third, many design tradeoffs can be considered (Eq. 5.3) to yield a higher hit ratio in the cache. The cache operation is often used together with TLB.

Example 5.4 Set-associative cache design and block mapping

An example is shown in Fig. 5.11b for the mapping of $n = 16$ blocks from main memory into a two-way associative cache ($k = 2$) with $v = 4$ sets over $m = 8$ block frames. For the i860 example in Fig. 5.8b, both the D-cache and I-cache are two-way associative ($k = 2$). There are 128 sets in the D-cache and 64 sets in the I-cache, with 256 and 128 block frames, respectively.

Sector Mapping Cache This block placement scheme is generalized from the above schemes. The idea is to partition both the cache and main memory into fixed-size *sectors*. Then a fully associative search is applied. That is, each sector can be placed in any of the available sector frames.

The memory requests are destined for blocks, not for sectors. This can be filtered out by comparing the sector tag in the memory address with all sector tags using a fully associative search. If a matched sector frame is found (a cache hit), the block field is used to locate the desired block within the sector frame.

If a cache miss occurs, the missing block is fetched from the main memory and brought into a congruent block frame in an available sector. That is, the ith block in a sector must be placed into the ith block frame in a destined sector frame. A *valid bit* is attached to each block frame to indicate whether the block is *valid* or *invalid*.

When the contents of a block frame are replaced from a new sector, the remaining block frames in the same sector are marked invalid. Only the block frames from the most recently referenced sector are marked valid for reference. However, multiple valid bits can be used to record other block states. The sector mapping

just described can be modified to yield other designs, depending on the block replacement policy being implemented.

Compared with fully associative or set-associative caches, the sector mapping cache offers the advantages of being flexible to implement various block replacement algorithms and being economical to perform a fully associative search across a limited number of sector tags.

The sector partitioning offers more freedom in grouping cache lines at both ends of the mapping. Making design choice between set-associative and sector mapping caches requires more trace and simulation evidence.

Example 5.5 Sector mapping cache design

Figure 5.12 shows an example of sector mapping with a sector size of four blocks. Note that each sector can be mapped to any of the sector frames with full associativity at the sector level.

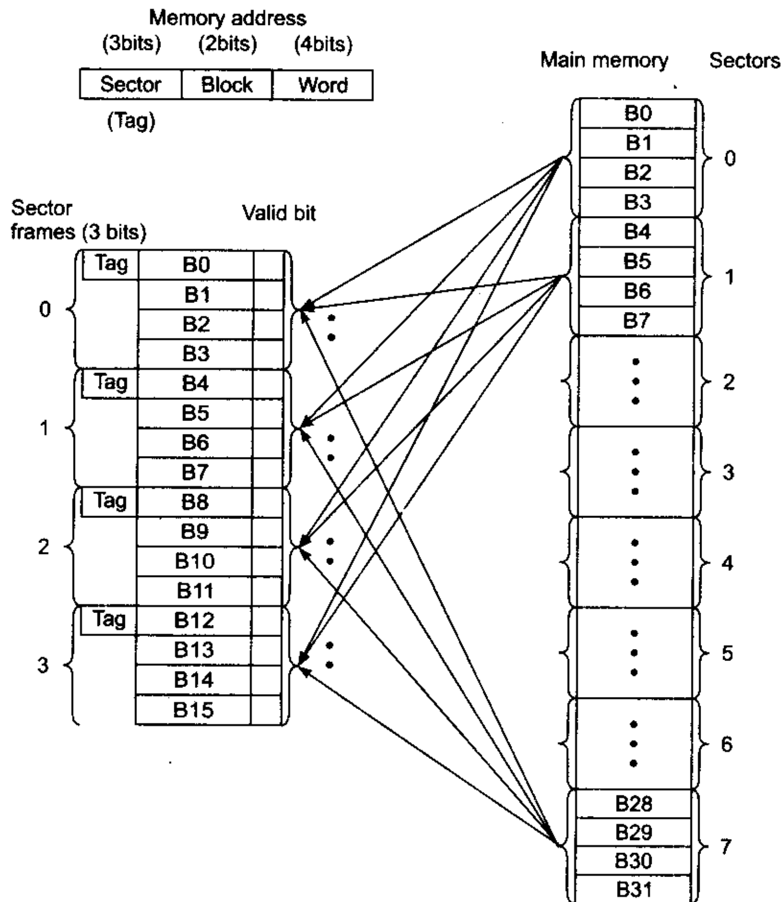


Fig. 5.12 A four-way sector mapping cache organization

This scheme was first implemented in the IBM System/360 Model 85. In the Model 85, there were 16 sectors, each having 16 blocks. Each block had 64 bytes, giving a total of 1024 bytes in each sector and a total cache capacity of 16 Kbytes using a LRU block replacement policy.

5.2.4 Cache Performance Issues

The performance of a cache design concerns two related aspects: the *cycle count* and the *hit ratio*. The cycle count refers to the number of basic machine cycles needed for cache access, update, and coherence control. The hit ratio determines how effectively the cache can reduce the overall memory-access time. Tradeoffs do exist between these two aspects. Key factors affecting cache speed and hit ratio are discussed below.

Program trace-driven simulation and *analytical modeling* are two complementary approaches to studying cache performance. Ideally, both should be applied together in order to provide a credible performance assessment.

Simulation studies present snapshots of program behavior and cache responses but they suffer from having a microscopic perspective.

Analytical models may deviate from reality under simplification. However, they provide some macroscopic and intuitive insight into the underlying processes.

Agreement between results generated from the two approaches allows one to draw a more credible conclusion. However, the generalization of any conclusion is limited by the finite-sized address traces and by the assumptions about address trace patterns. Simulation results can be used to verify the theoretical results, and analytical formulation can guide simulation experiments on a wider range of parameters.

Cycle Counts The cache speed is affected by the underlying static or dynamic RAM technology, the cache organization, and the cache hit ratios. The total cycle count should be predicated with appropriate cache hit ratios. This affects various cache design decisions, as already seen in previous sections.

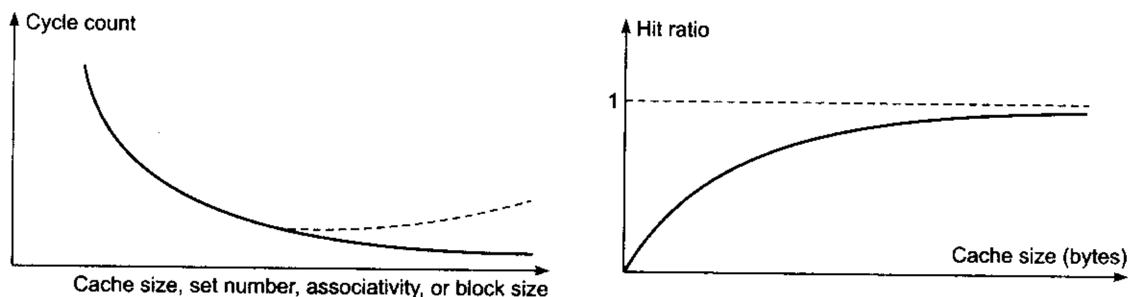
The cycle counts are not credible unless detailed simulation of all aspects of a memory hierarchy is performed. The write-through or write-back policies also affect the cycle count. Cache size, block size, set number, and associativity all affect the cycle count as illustrated in Fig. 5.13.

The cycle count is directly related to the hit ratio, which decreases almost linearly with increasing values of the above cache parameters. But the decreasing trend becomes flat and after a certain point turns into an increasing trend (the dashed line in Fig. 5.13a). This is caused primarily by the effect of the block size on the hit ratio, which will be discussed below.

Hit Ratios The cache hit ratio is affected by the cache size and by the block size in different ways. These effects are illustrated in Figs. 5.13b and 5.13c, respectively. Generally, the hit ratio increases with respect to increasing cache size (Fig. 5.13b).

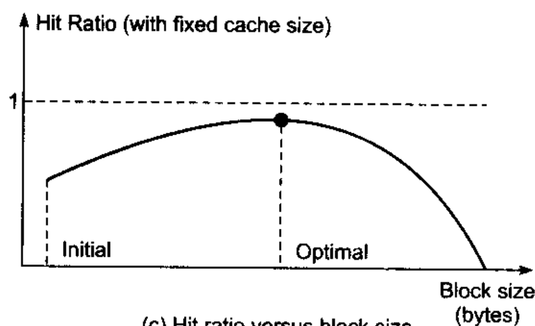
When the cache size approaches infinity, a 100% hit ratio should be expected. However, this will never happen because the cache size is always bounded by a limited budget. The initial cache loading and changes in locality also prevent such an ideal performance. The curves in Fig. 5.13b can be approximated by $1 - C^{-0.5}$, where C is the total cache size.

Effect of Block Size With a fixed cache size, cache performance is rather sensitive to block size. Figure 5.13c illustrates the rise and fall of the hit ratio as the cache block varies from small to large. Initially, we assume a block size (such as 32 bytes per block). This block size is determined mainly by the temporal locality in typical programs.



(a) The total cycle count for cache access (Courtesy of S. A. Przybylski; reprinted with permission from *Cache and Memory Hierarchy Design*, Morgan Kaufmann Publishers, 1990)

(b) Hit ratio versus cache size



(c) Hit ratio versus block size

Fig. 5.13 **Cache performance versus design parameters used**

As the block size increases, the hit ratio improves because of spatial locality in referencing larger instruction/data blocks. The increase reaches its peak at a certain *optimum block size*. After this point, the hit ratio decreases with increasing block size. This is caused by the mismatch between program behavior and block size.

As a matter of fact, as the block size becomes very large, many words fetched into the cache may never be used. Also, the temporal locality effects are gradually lost with larger block size. Finally, the hit ratio approaches zero when the block size equals the entire cache size.

For a bus-based system, Smith (1987) determined that the optimum block size should be chosen to minimize the effective memory-access time. This optimum size depends on the ratio of the access latency and the bus cycle time (data transfer rate). He identified design targets for the hit ratio, bus traffic, and average delay per reference based on an empirical model derived from a wide variety of benchmark simulations.

Effects of Set Number In a set-associative cache, the effects of set number are obvious. For a fixed cache capacity, the hit ratio may decrease as the number of sets increases. As the set number increases from 32 to 64, 128, and 256, the decrease in the hit ratio is rather small based on Smith's 1982 report. When the set number increases to 512 and beyond, the hit ratio decreases faster. Also, the tradeoffs between block size and set number should not be ignored (Eq. 5.3).

Other Performance Factors In a performance-directed design, tradeoffs exist among the cache size, set number, block size, and memory speed. Independent blocks, fetch sizes, and fetch strategies also affect the performance in various ways.

Multilevel cache hierarchies offer options for expanding the cache effects. Very often, a write-through policy is used in the first-level cache, and a write-back policy in the second-level cache. As in the memory hierarchy, an optimal cache hierarchy design must match special program behavior in the target application domain.

The distribution of cache references for instruction, loads, and writes of data will affect the hierarchy design. Based on some previous program traces, 63% instruction fetches, 25% loads, and 12% writes were reported. This affects the decision to split the instruction cache from the data cache.

Note 5.1 Multi-level cache memories

Over the last two decades, processor speeds have risen much faster than memory speeds. In fact, in terms of number of processor cycles—i.e. relative to processor clock speeds—main memory is much slower today than it was twenty or thirty years ago, albeit it is also much less expensive and storage densities are much greater.

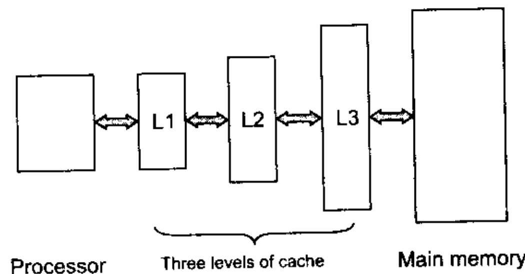


Fig. 5.14 Three levels of cache between processor and main memory

To 'bridge the divide' between processor speeds and main memory speeds—a divide which has grown over the years—multiple levels of cache memories are employed between processor(s) and main memory, as shown schematically in Fig. 5.14.

To avoid pipeline stalls, level one cache L1, closest to the processor, is divided between instruction and data cache (I-cache and D-cache). It is the fastest and smallest of the three caches, and uses faster SRAM; ideally, its access time should equal one processor clock cycle. On multi-core chips, separate L1 cache is provided with each processor core.

L2 cache may also be on the same chip but, on multi-core chips, typically it is shared between processor cores; size goes up to a few megabytes, using slower and less expensive dynamic RAM. L3 cache, if provided, may be on-chip, off-chip or may even be integrated with the main memory.

In a system with multilevel cache, to find a required byte or word in memory, the processor can initiate the access in parallel across two (or more) levels of cache; when a cache hit occurs at a particular level, access in the lower levels can be terminated.

Over the last two decades, there have been huge advances in VLSI technology in terms of both device densities on a chip and clock speeds. Over this same period, due to lower system costs, the total number of processors manufactured and sold around the world has also risen steadily. As a result, many different processors are produced in each processor family; as examples, we need only to cite Intel Pentium, Sun SPARC, MIPS, and the Power series of processors. More than one manufacturer usually produces processors in each of these families.

Different members of each processor family are targeted at different applications (recall Fig. 4.1), and are built to different cost *versus* performance criteria. To bridge today's larger processor-memory speed gap, multilevel cache systems are employed. Specific cache systems are designed based on specific processor specifications. For each level of the cache, processor designers must select the cache size, cache block size, mapping scheme (direct or set associative), write back/write through policy, etc.

As mentioned above, simulation studies and analytical models can be used for such designs. Also important in this context are past experience with earlier processor models, chip area requirements for the cache, power consumption, and often the intuitive decisions made by processor designers.

5.3 SHARED-MEMORY ORGANIZATIONS

Memory interleaving provides a higher bandwidth for pipelined access of contiguous memory locations. Methods for allocating and deallocating main memory to multiple user programs are considered for optimizing memory utilization. Memory bandwidth analysis and fault tolerance issues are also discussed below.

5.3.1 Interleaved Memory Organization

Various organizations of the physical memory are studied in this section. In order to close up the speed gap between the CPU/cache and main memory built with RAM modules, an *interleaving* technique is presented below which allows pipelined access of the parallel memory modules.

The memory design goal is to broaden the *effective memory bandwidth* so that more memory words can be accessed per unit time. The ultimate purpose is to match the memory bandwidth with the bus bandwidth and with the processor bandwidth.

Memory Interleaving The main memory is built with multiple modules. These memory modules are connected to a system bus or a switching network to which other resources such as processors or I/O devices are also connected.

Once presented with a memory address, each memory module returns with one word per cycle. It is possible to present different addresses to different memory modules so that parallel access of multiple words can be done simultaneously or in a pipelined fashion. Both parallel access and pipelined access are forms of parallelism practiced in a parallel memory organization.

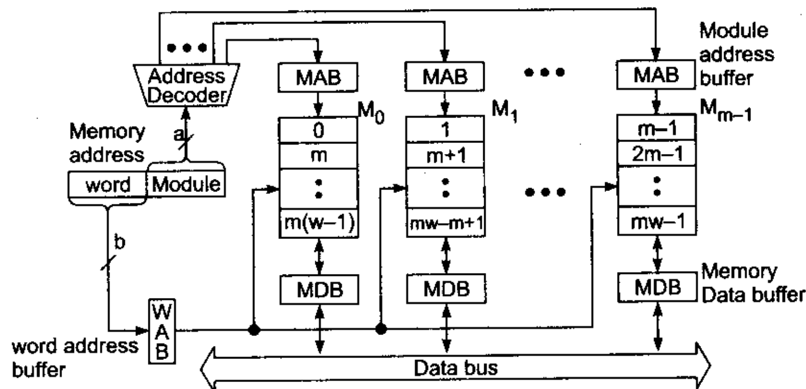
Consider a main memory formed with $m = 2^a$ memory modules, each containing $w = 2^b$ words of memory cells. The total memory capacity is $m \cdot w = 2^{a+b}$ words. These memory words are assigned linear addresses. Different ways of assigning linear addresses result in different memory organizations.

Besides random access, the main memory is often block-accessed at consecutive addresses. Block access is needed for fetching a sequence of instructions or for accessing a linearly ordered data structure. Each block

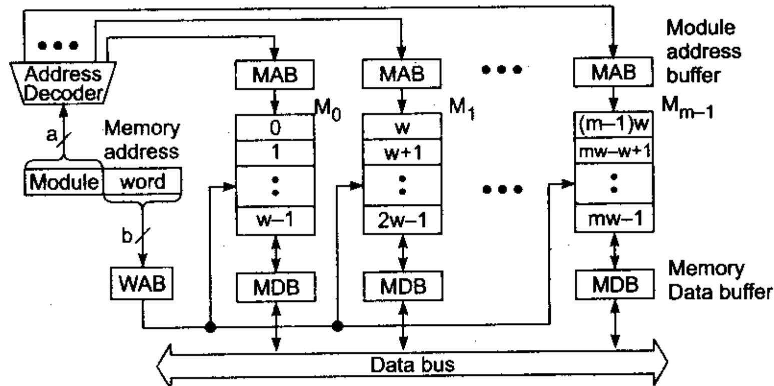
access may correspond to the size of a cache block (cache line) or to that of several cache blocks. Therefore, it is desirable to design the memory to facilitate block access of contiguous words.

Figure 5.15a shows two address formats for memory interleaving. *Low-order interleaving* spreads contiguous memory locations across the m modules horizontally (Fig. 5.15a). This implies that the low-order a bits of the memory address are used to identify the memory module. The high-order b bits are the word addresses (displacement) within each module. Note that the same word address is applied to all memory modules simultaneously. A module address decoder is used to distribute module addresses.

High-order interleaving (Fig. 5.15b) uses the high-order a bits as the module address and the low-order b bits as the word address within each module. Contiguous memory locations are thus assigned to the same memory module. In each memory cycle, only one word is accessed from each module. Thus the high-order interleaving cannot support block access of contiguous locations.



(a) Low-order m-way interleaving (the C-access memory scheme)

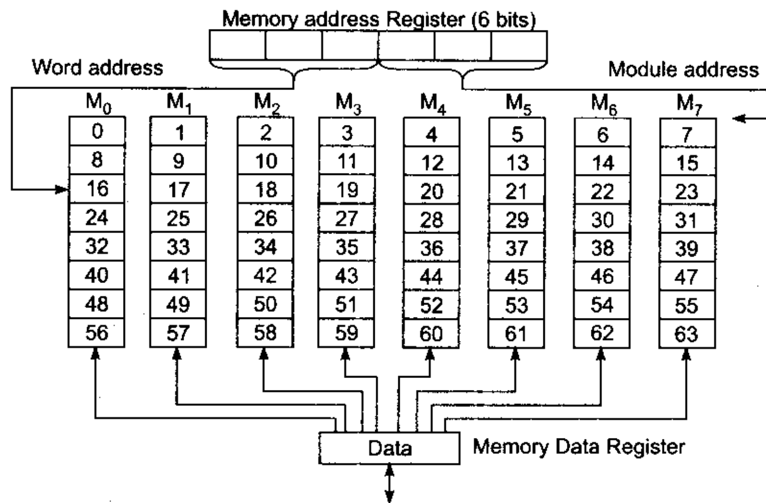


(b) High-order m-way interleaving

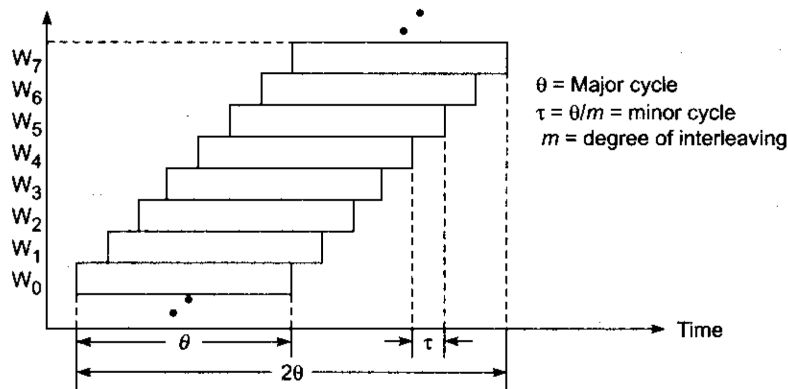
Fig. 5.15 Two interleaved memory organizations with $m = 2^a$ modules and $w = 2^b$ words per module (word addresses shown in boxes)

On the other hand, the low-order m-way interleaving does support block access in a pipelined fashion. Unless otherwise specified, we consider only low-order memory interleaving in subsequent discussions.

Pipelined Memory Access Access of the m memory modules can be overlapped in a pipelined fashion. For this purpose, the memory cycle (called the *major cycle*) is subdivided into m *minor cycles*.



(a) Eight-way low-order interleaving (absolute address shown in each memory word)



(b) Pipelined access of eight consecutive words in a C-access memory

Fig. 5.16 Multiway interleaved memory organization and the C-access timing chart

An eight-way interleaved memory (with $m = 8$ and $w = 8$ and thus $a = b = 3$) is shown in Fig. 5.16a. Let θ be the major cycle and τ the minor cycle. These two cycle times are related as follows:

$$\tau = \frac{\theta}{m} \tag{5.4}$$

where m is the *degree of interleaving*. The timing of the pipelined access of the eight contiguous memory words is shown in Fig. 5.16b. This type of *concurrent access* of contiguous words has been called a *C-access* memory scheme. The major cycle θ is the total time required to complete the access of a single word from

a module. The minor cycle τ is the actual time needed to produce one word, assuming overlapped access of successive memory modules separated in every minor cycle τ.

Note that the pipelined access of the block of eight contiguous words is sandwiched between other pipelined block accesses before and after the present block. Even though the total block access time is 2θ, the *effective access time* of each word is reduced to τ as the memory is contiguously accessed in a pipelined fashion.

5.3.2 Bandwidth and Fault Tolerance

Hellerman (1967) has derived an equation to estimate the effective increase in memory bandwidth through multiway interleaving. A single memory module is assumed to deliver one word per memory cycle and thus has a bandwidth of 1.

Memory Bandwidth The *memory bandwidth B* of an m-way interleaved memory is upper-bounded by m and lower-bounded by 1. The Hellerman estimate of B is

$$B = m^{0.56} \simeq \sqrt{m} \tag{5.5}$$

where m is the number of interleaved memory modules. This equation implies that if 16 memory modules are used, then the effective memory bandwidth is approximately four times that of a single module.

This pessimistic estimate is due to the fact that block access of various lengths and access of single words are randomly mixed in user programs. Hellerman's estimate was based on a single-processor system. If memory-access conflicts from multiple processors (such as the hot spot problem) are considered, the effective memory bandwidth will be further reduced.

In a vector processing computer, the access time of a long vector with n elements and stride distance 1 has been estimated by Cragon (1992) as follows: It is assumed that the n elements are stored in contiguous memory locations in an m-way interleaved memory system. The average time t_1 required to access one element in a vector is estimated by

$$t_1 = \frac{\theta}{m}\left(1 + \frac{m-1}{n}\right) \tag{5.6}$$

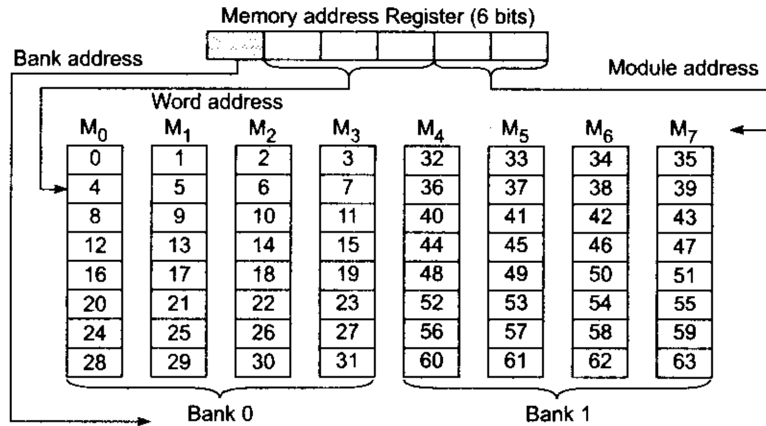
When $n \to \infty$ (very long vector), $t_1 \to \theta/m = \tau$ as derived in Eq. 5.4. As $n \to 1$ (scalar access), $t_1 \to \theta$. Equation 5.6 conveys the message that interleaved memory appeals to pipelined access of long vectors; the longer the better.

Fault Tolerance High- and low-order interleaving can be combined to yield many different interleaved memory organizations. Sequential addresses are assigned in the high-order interleaved memory in each memory module.

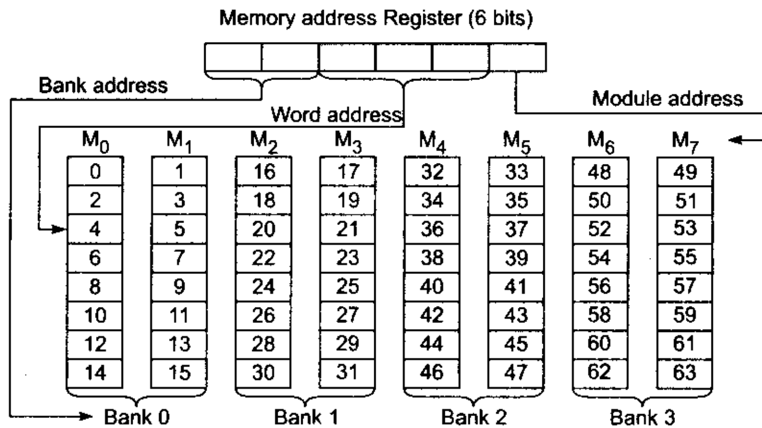
This makes it easier to isolate faulty memory modules in a *memory bank* of m memory modules. When one module failure is detected, the remaining modules can still be used by opening a window in the address space. This fault isolation cannot be carried out in a low-order interleaved memory, in which a module failure may paralyze the entire memory bank. Thus low-order interleaving memory is not fault-tolerant.

Example 5.6 Memory banks, fault tolerance, and bandwidth tradeoffs

In Fig. 5.17, two alternative memory addressing schemes are shown which combine the high- and low-order interleaving concepts. These alternatives offer a better bandwidth in case of module failure. A four-way low-order interleaving is organized in each of two memory banks in Fig. 5.17a.



(a) Four-way interleaving within each memory bank



(b) Two-way interleaving within each memory bank

Fig. 5.17 Bandwidth analysis of two alternative interleaved memory organizations over eight memory modules (Absolute address shown in each memory bank.)

On the other hand, two-way low-order interleaving is depicted in Fig. 5.17b with the memory system divided into four memory banks. The high-order bits are used to identify the memory banks. The low-order bits are used to address the modules for memory interleaving.

In case of single module failure, the maximum memory bandwidth of the eightway interleaved memory (Fig. 5.16a) is reduced to zero because the entire memory bank must be abandoned. For the four-way two-bank design (Fig. 5.17a), the maximum bandwidth is reduced to four words per memory cycle because only one of the two faulty banks is abandoned.

In the two-way design in Fig. 5.17b, the gracefully degraded memory system may still have three working memory banks; thus a maximum bandwidth of six words is expected. The higher the degree of interleaving, the higher the potential memory bandwidth if the system is fault-free.

If fault tolerance is an issue which cannot be ignored, then tradeoffs do exist between the degree of interleaving and the number of memory banks used. Each memory bank is essentially self-enclosed, is independent of the conditions of other banks, and thus offers better fault isolation in case of failure.

5.3.3 Memory Allocation Schemes

The idea of virtual memory is to allow many software processes time-shared use of the main memory, which is a precious resource with limited capacity. The portion of the OS kernel which handles the allocation and deallocation of main memory to executing processes is called the *memory manager.* The memory manager monitors the amount of available main memory and decides which processes should reside in main memory and which should be put back to disk if the main memory reaches its limit.

In this section, we study the basic concepts of memory swapping, either at the process level or at the individual page level. Both swapping systems and demand paging systems are introduced, based on the development of the memory management subsystem in UNIX. Possible extensions of these memory allocation schemes are discussed along with some performance issues.

Allocation Policies *Memory swapping* is the process of moving blocks of information between the levels of a memory hierarchy. For simplicity, we concentrate on swapping between the main memory and the disk memory. Several key concepts or design alternatives in implementing memory swapping are introduced below.

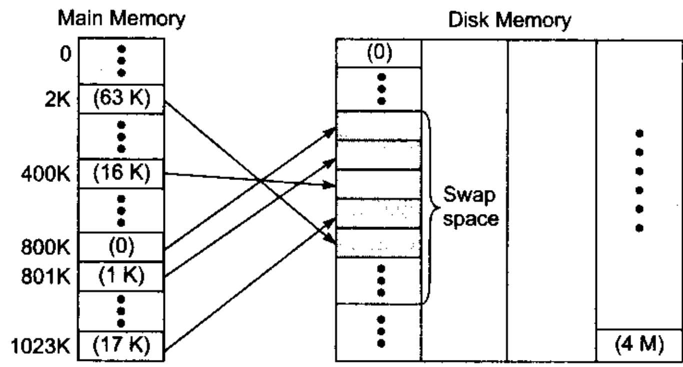
First, the swapping policy can be made either nonpreemptive or preemptive. In *nonpreemptive allocation,* the incoming block can be placed only in a free region of the main memory. A *preemptive allocation* scheme allows the placement of an incoming block in a region presently occupied by another process. In either case, the memory manager should try to allocate the free space first.

When the main memory space is fully allocated, the nonpreemptive scheme swaps out some of the allocated processes (or pages) to vacate space for the incoming block. On the other hand, a preemptive scheme has the freedom to preempt an executing process. The nonpreemptive scheme is easier to implement, but it may not yield the best memory utilization.

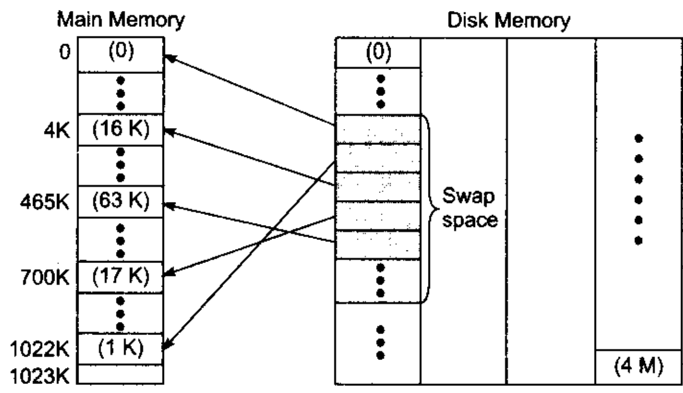
The preemptive scheme offers more flexibility, but it requires mechanisms be established to determine which pages or processes are to be swapped out and to avoid *thrashing* caused by an excessive amount of swapping between memory levels. This implies that preemptive allocation schemes are more complex and more expensive to implement.

In addition, an allocation policy can be made either local or global. A *local allocation* policy involves only the resident working set of the faulty process. A *global allocation* policy considers the history of the working sets of all resident processes in making a swapping decision. Most computers use the local policy.

Swapping Systems This refers to memory systems which allow swapping to occur only at the entire process level. A *swap device* is a configurable section of a disk which is set aside for temporary storage of information being swapped out of the main memory. The portion of the disk memory space set aside for a swap device is called the *swap space*, as depicted in Fig. 5.18.



(a) Moving a process (or pages) onto the swap space on a disk



(b) Swapping in a process (or pages) to the memory

Fig. 5.18 The concept of memory swapping in a virtual memory hierarchy (virtual page addresses are identified by numbers within parentheses, assuming a page size of 1 K words)

The memory manager allocates disk space for program files one block at a time, but it allocates space on the swap device in groups of contiguous blocks. For simplicity, we consider blocks as fixed-size pages. The virtual address space of a process may occupy a number of pages. The size of a process address space is limited by the amount of physical memory available on a swapping system.

The swapping system was used in the PDP-11 and in early UNIX systems. It transfers the entire process between main memory and the swap device. It does not transfer parts (pages) of a process separately. For example, the PDP-11 allowed a maximum process size of only 64 Kbytes. The entire process must reside in main memory to execute.

A simple example is shown in Fig. 5.18 to illustrate the concepts of *swapping out* and *swapping in* a process consisting of five resident pages identified by virtual page addresses 0, 1K, 16K, 17K, and 63K with an assumed page size of lK words (or 4 Kbytes for a 32-bit word length).

Figure 5.18a shows the allocation of physical memory before the swapping. The main memory is assumed to have 1024 page frames, and the disk can accommodate 4M pages. The five resident pages, scattered around the main memory, are swapped out to the swap device in contiguous pages as shown by the shaded boxes.

Later on, the entire process may be required to swap back into the main memory, as depicted in Fig. 5.18b. Different page frames may be allocated to accommodate the returning pages. The reason why contiguous blocks are mapped in the swap device is to enable faster I/O in one multiblock data transfer rather than in several single-block transfer operations.

It should be noted that only the assigned pages are swapped out and in, not the entire process address space. In the example process, the entire process address space is assumed to be 64K. The unassigned pages include the two gaps of virtual addresses between 2K and 16K and between 18K and 63K, respectively.

These *empty* spaces are not involved in the swapping process. When the memory manager swaps the process back into memory, the virtual address map should be able to identify the virtual addresses required for the returning process.

Swapping in UNIX In the early UNIX/OS, the kernel swaps out a process to create free memory space under the following system calls:

(1) The allocation of space for a child process being created.
(2) The increase in the size of a process address space.
(3) The increased space demand by the stack for a process.
(4) The demand for space by a returning process swapped out previously.

A special process 0 is reserved as a *swapper*. The swapper must swap the process into main memory before the kernel can schedule it for execution. In fact, process 0 is the only process which can do so. Only when there are processes to swap in and eligible processes to swap out can the swapper do its work. Otherwise, the swapper goes to sleep.

However, the kernel periodically wakes the swapper up when the situation demands. The swapper should be designed to avoid thrashing, especially in swapping out a process which has not been executed yet.

Demand Paging Systems A paged memory system often uses a *demand paging* memory allocation policy. This policy allows only pages (instead of processes) to be transferred between the main memory and the swap device. In Fig. 5.18, individual pages of a process are allowed to be independently swapped out and in, and we have a demand paging system.

The UNIX BSD 4.0 release was the first implementation of the demand paging policy. UNIX System V also supported demand paging. In a demand paging system, the entire process does not have to move into main memory to execute. The pages are brought into main memory only upon demand.

This allows the process address space to be larger than the physical address space. The major advantage of demand paging is that it offers the flexibility to dynamically accommodate a large number of processes in the physical memory on a time-sharing or multiprogrammed basis with significantly enlarged address spaces.

The idea of demand paging matches nicely with the working-set concept. Only the working sets of active processes are resident in memory. Back (1986) has defined the *working set* of a process as the set of pages referenced by the process during the last n memory references, where n is the *window size* of the working set.

Example 5.7 Working sets generated with a page trace

In the following page trace, the successive contents of the working set of a process are shown for a window of size $n = 3$:

Page trace	7	24	7	15	24	24	8	1	1	8	9	24	8	1
Working set	7	7	7	7	7	7	8	8	8	8	8	8	8	8
		24	24	24	24	24	24	24	24	24	9	9	9	1
				15	15	15	15	1	1	1	1	24	24	24

If the kernel keeps only the working sets with a sufficiently large window in the main memory, many more active processes can concurrently reside in the memory than the swapping system can provide. This potentially increases the system throughput and reduces the swapping traffic. In other words, undemanded pages are not involved in the swapping process.

Hybrid Memory Systems The VAX/VMS and UNIX System V had implemented *hybrid memory systems* combining the advantages of both swapping and demand paging. When several processes simultaneously are in the ready-to-run-but-swapped state, the swapper may choose to swap out several processes entirely to vacate the needed space. This scheme may lower the page fault rate and reduce thrashing.

Other virtual memory systems may use *anticipatory paging*, which prefetches pages based on anticipation. This scheme is rather difficult to implement. Unless memory reference patterns can be predicted at the time when the compiler generates the addresses, this scheme cannot demonstrate its power. A short-range memory reference pattern is a lot easier to predict due to the locality properties.

SEQUENTIAL AND WEAK CONSISTENCY MODELS

This section studies shared-memory behavior in relation to program execution order and memory-access order. The sequential consistency and weak consistency memory models are characterized and their potential for improving performance is assessed. In Chapter 9, we will introduce the processor consistency and release consistency models for building scalable multiprocessor systems.

5.4.1 Atomicity and Event Ordering

The problem of memory inconsistency arises when the memory-access order differs from the program execution order. As illustrated in Fig. 5.19a, a uniprocessor system maps an SISD sequence into a similar

execution sequence. Thus memory accesses (for instructions and data) are consistent with the program execution order. This property has been called *sequential consistency* (Lamport, 1979).

In a shared-memory multiprocessor, there are multiple instruction sequences in different processors as shown in Fig. 5.19b. Different ways of interleaving the MIMD instruction sequences into a global memory-access sequence lead to different shared memory behaviors.

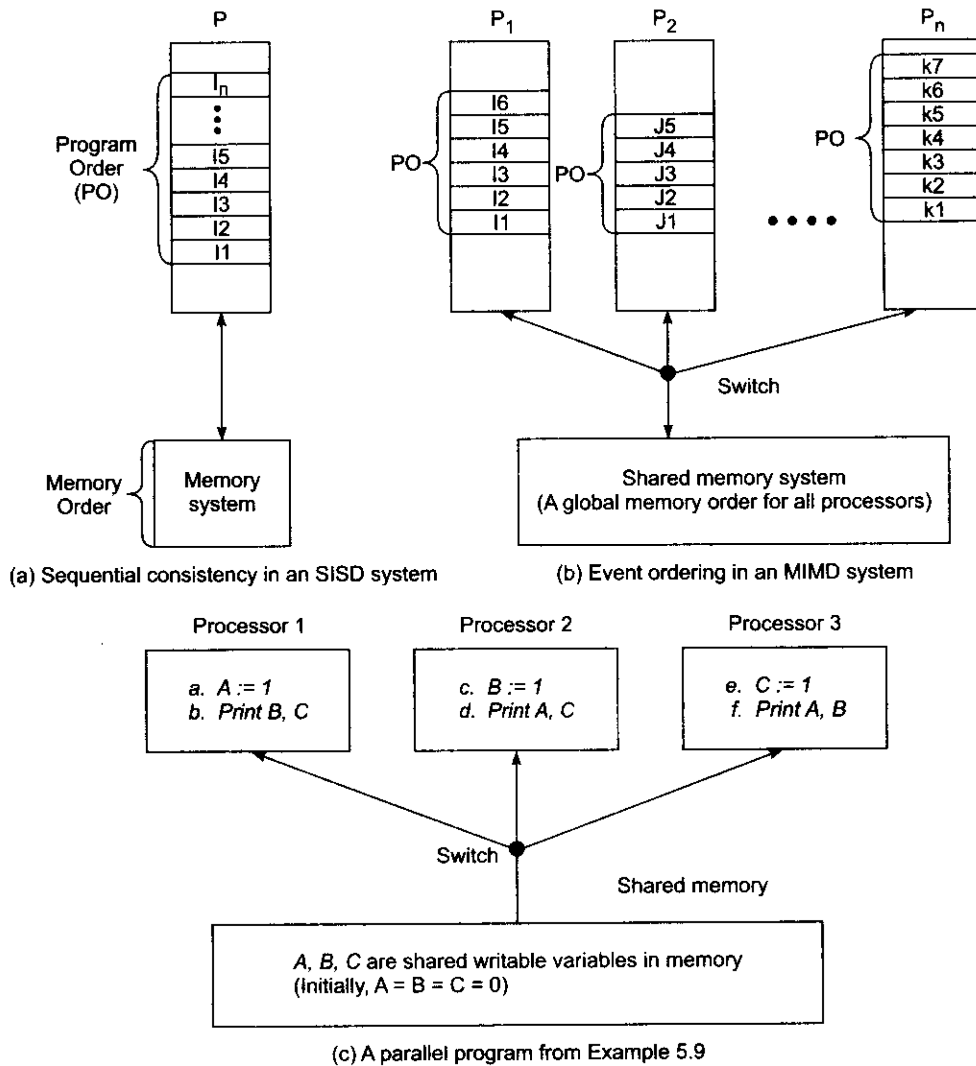


Fig. 5.19 The access ordering of memory events in a uniprocessor and in a multiprocessor, respectively (Courtesy of Dubois and Briggs, *Tutorial Notes on Shared-Memory Multiprocessors*, Int. Symp. Computer Arch., May 1990)

How these two sequences are made consistent distinguishes the memory behavior in strong and weak models. The quality of a memory model is indicated by hardware/software efficiency, simplicity, usefulness, and bandwidth performance.

Memory Consistency Issues The behavior of a shared-memory system as observed by processors is called a *memory model*. Specification of the memory model answers three fundamental questions: (1) What behavior should a programmer/compiler expect from a shared-memory multiprocessor? (2) How can a definition of the expected behavior guarantee coverage of all contingencies? (3) How must processors and the memory system behave to ensure consistent adherence to the expected behavior of the multiprocessor?

In general, choosing a memory model involves making a compromise between a strong model minimally restricting software and a weak model offering efficient implementation. The use of *partial order* in specifying memory events gives a formal description of special memory behavior.

Primitive memory operations for multiprocessors include *load* (*read*), *store* (*write*), and one or more synchronization operations such as *swap* (atomic *load-store*) or *conditional store*. For simplicity, we consider one representative synchronization operation *swap*, besides the *load* and *store* operations.

Event Orderings On a multiprocessor, concurrent instruction streams (or threads) executing on different processors are *processes*. Each process executes a code segment. The order in which shared memory operations are performed by one process may be used by other processes. *Memory events* correspond to shared-memory accesses. Consistency models specify the order by which the events from one process should be observed by other processes in the machine.

The *event ordering* can be used to declare whether a memory event is legal or illegal, when several processes are accessing a common set of memory locations. A *program order* is the order by which memory accesses occur for the execution of a single process, provided that no program reordering has taken place. Dubois et al. (1986) have defined three primitive memory operations for the purpose of specifying memory consistency models:

(1) A *load* by processor P_i is considered *performed* with respect to processor P_k at a point of time when the issuing of a *store* to the same location by P_k cannot affect the value returned by the *load*.

(2) A *store* by P_i is considered *performed* with respect to P_k at one time when an issued *load* to the same address by P_k returns the value by this *store*.

(3) A *load* is *globally performed* if it is performed with respect to all processors and if the *store* that is the source of the returned value has been performed with respect to all processors.

As illustrated in Fig. 5.19a, a processor can execute instructions out of program order using a compiler to resequence instructions in order to boost performance. A uniprocessor system allows these out-of-sequence executions provided that hardware interlock mechanisms exist to check data and control dependences between instructions.

When a processor in a multiprocessor system executes a concurrent program as illustrated in Fig. 5.19b, local dependence checking is necessary but may not be sufficient to preserve the intended outcome of a concurrent execution.

Maintaining the correctness and predictability of the execution results is rather complex on an MIMD system for the following reasons:

(a) The order in which instructions belonging to different streams are executed is not fixed in a parallel program. If no synchronization among the instruction streams exists, then a large number of different instruction interleavings is possible.
(b) If for performance reasons the order of execution of instructions belonging to the same stream is different from the program order, then an even larger number of instruction interleavings is possible.
(c) If accesses are not atomic with multiple copies of the same data coexisting as in a cache-based system, then different processors can individually observe different interleavings during the same execution. In this case, the total number of possible execution instantiations of a program becomes even larger.

Example 5.8 Event ordering in a three-processor system (Dubois, Scheurich, and Briggs, 1988)

To illustrate the possible ways of interleaving concurrent program executions among multiple processors updating the same memory, we examine the simultaneous and asynchronous executions of three program segments on the three processors in Fig. 5.19c.

The shared variables are initially set as zeros, and we assume a *Print* statement reads both variables indivisibly during the same cycle to avoid confusion. If the outputs of all three processors are concatenated in the order P_1, P_2, and P_3, then the output forms a 6-tuple of binary vectors.

There are $2^6 = 64$ possible output combinations. If all processors execute instructions in their own program orders, then the execution interleaving a, b, c, d, e, f is possible, yielding the output 001011. Another interleaving, a, c, e, b, d, f, also preserves the program orders and yields the output 111111.

If processors are allowed to execute instructions out of program order, assuming that no data dependences exist among reordered instructions, then the interleaving b, d, f, e, a, c is possible, yielding the output 000000.

Out of $6! = 720$ possible execution interleavings, 90 preserve the individual program order. From these 90 interleavings not all 6-tuple combinations can result. For example, the outcome 000000 is not possible if processors execute instructions in program order only. As another example, the outcome 011001 is possible if different processors can observe events in different orders, as can be the case with replicated memories.

Atomicity From the above example, multiprocessor memory behavior can be described in three categories:

(1) Program order preserved and uniform observation sequence by all processors.
(2) Out-of-program-order allowed and uniform observation sequence by all processors.
(3) Out-of-program-order allowed and nonuniform sequences observed by different processors.

This behavioral categorization leads to two classes of shared-memory systems for multiprocessors: The first allows *atomic memory accesses*, and the second allows *nonatomic memory accesses*. A shared-memory access is atomic if the memory updates are known to all processors at the same time. Thus a *store* is atomic if the value stored becomes readable to all processors at the same time. Thus a necessary and sufficient