FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

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

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

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

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

The McGraw-Hill Companies

ADVANCED COMPUTER ARCHITECTURE

Parallelism, Scalability, Programmability

Second Edition

Kai Hwang

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

Naresh Jotwani

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

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

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

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.

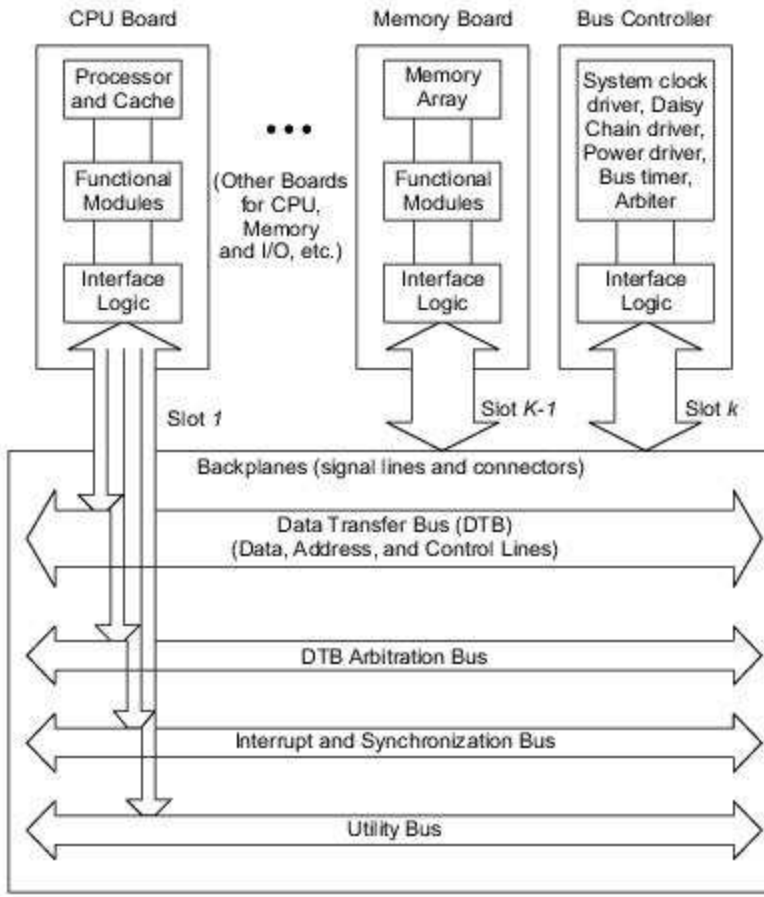


Fig. 5.1 Backplane buses, system interfaces, and slot connections to various functional boards in a multiprocessor system

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.

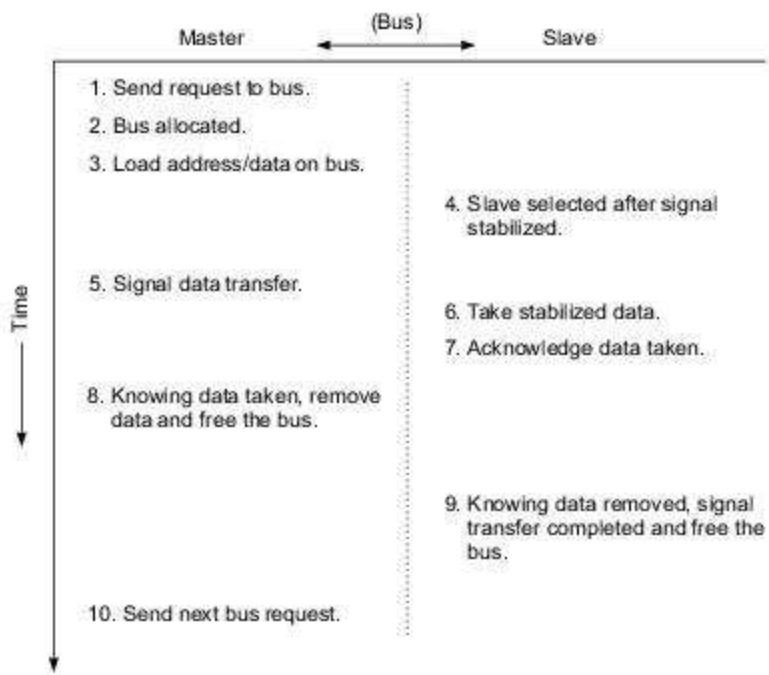


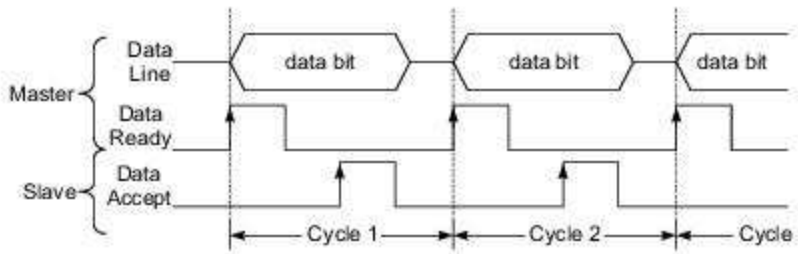
Fig. 5.2 Typical time sequence for information transfer between a master and a slave over a system bus

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.

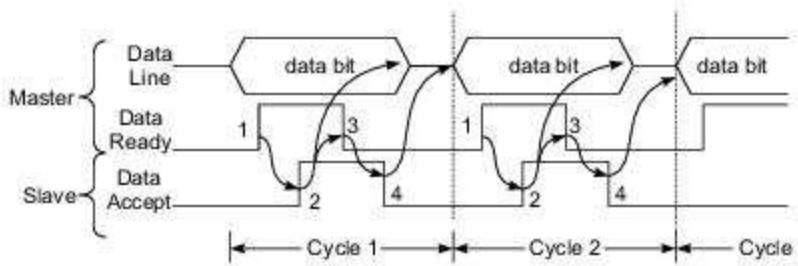
Once the data becomes stabilized on the data lines, the master uses a *data-ready* pulse to initiate the transfer. The slave uses a *data-accept* pulse to signal completion of the information transfer.

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.

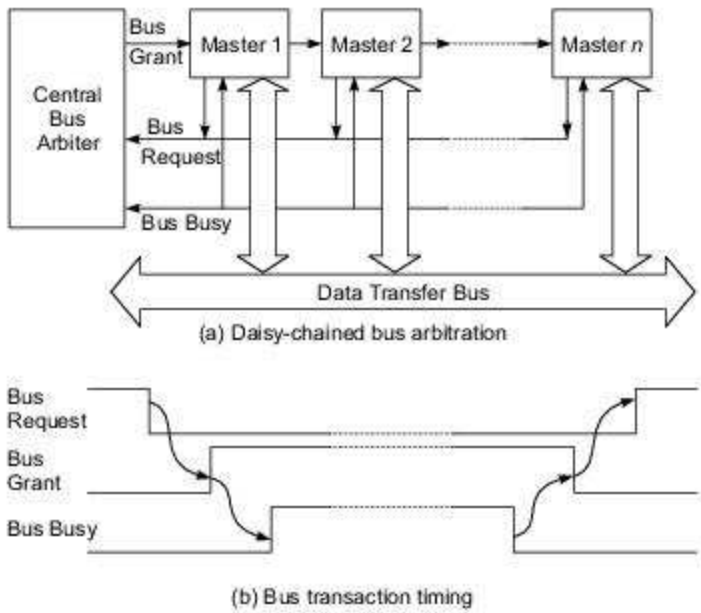


Fig. 5.4 Central bus arbitration using shared requests and daisy-chained bus grants with a fixed priority

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.

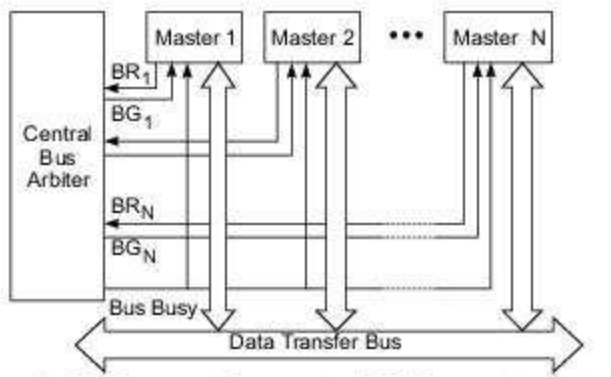
The arbitration among potential masters is still carried out by a central arbiter. However, any priority-

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.

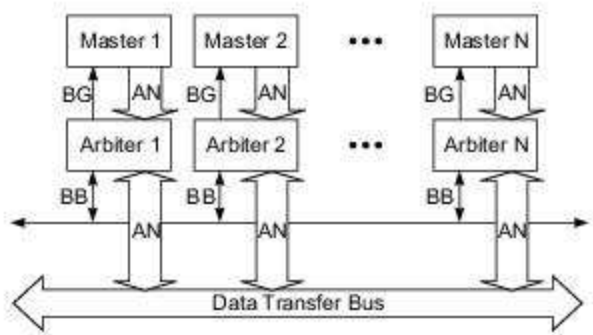
The advantage of using independent requests and grants in bus arbitration is their flexibility and faster arbitration time compared with the daisy-chained policy. The drawback is the large number of arbitration lines used.

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.



Legends: BR_i (Bus request from master i) BG_i (Bus grant to master i)

(a) Independent requests with a central arbiter



Legends: BG (Bus grant) BB (Bus busy) AN (Arbitration number)

(b) Using distributed arbiters

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.

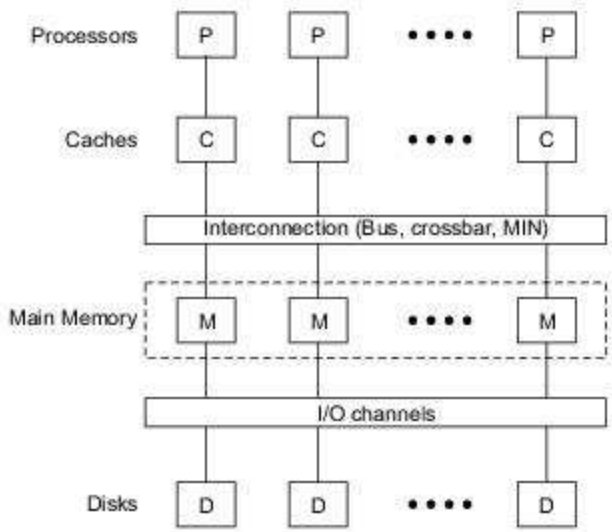


Fig. 5.6 A memory hierarchy for a shared-memory multiprocessor

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.

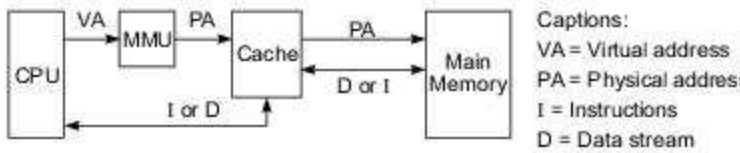
In this case, the cache is indexed and tagged with the physical address. Cache lookup must occur after address translation in the TLB or MMU.

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.

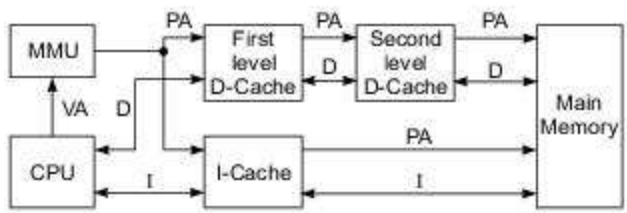
Data is written through the main memory immediately via a *write-through* (WT) cache, or delayed until block replacement by using a *write-back* (WB) cache. A WT cache requires more bus or network cycles to access the main memory, while a WB cache allows the CPU to continue without waiting for the memory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

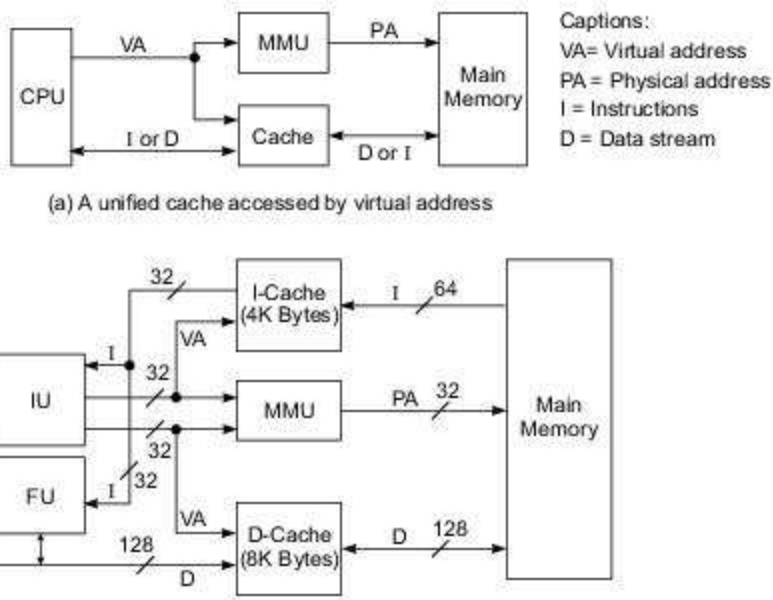


Fig. 5.8 Virtual address models for unified and split. caches (Courtesy of Intel Corporation, 1989)

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 $\overline{B}_i$ 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 $\{\overline{B}_i\}$. 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.

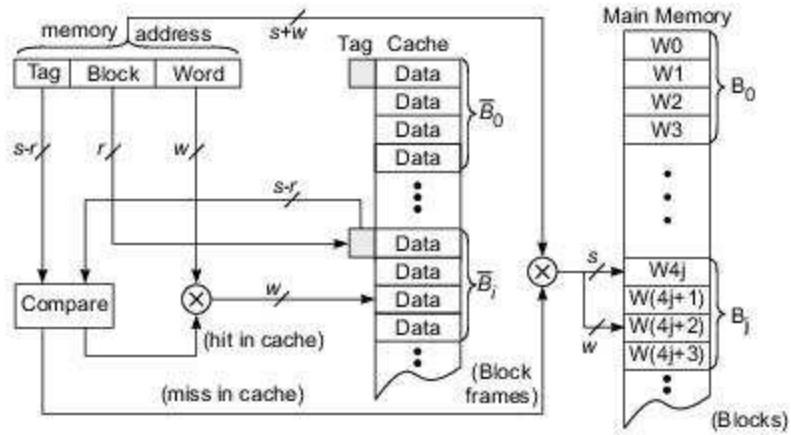
Direct-Mapping Cache This cache organization is based on a direct mapping of $n/m = 2^{s-r}$ memory blocks, separated by equal distances, to one block frame in the cache. The placement is defined below using a modulo-m function. Block B_j is mapped to block frame $\overline{B}_i$:

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

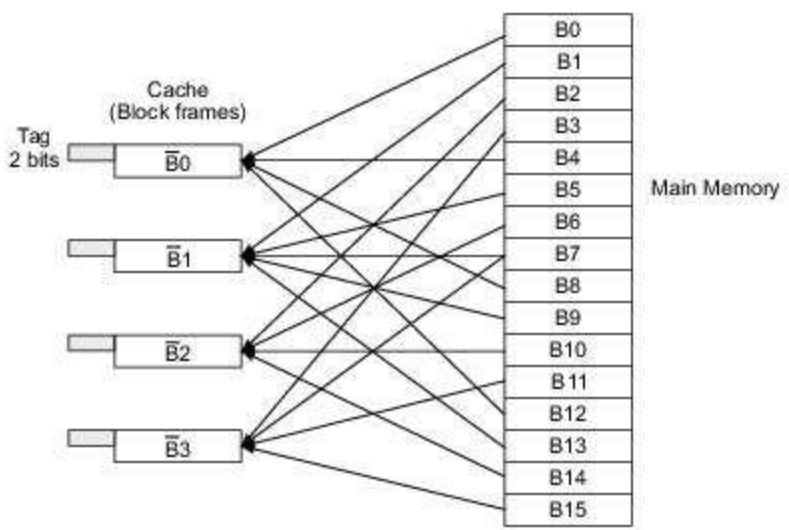
There is a *unique* block frame $\overline{B}_i$ that each B_j can load into. There is no way to implement a block replacement policy. This direct mapping is very rigid but is the simplest cache organization to implement. Direct mapping is illustrated in Fig. 5.9a for the case where each block contains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

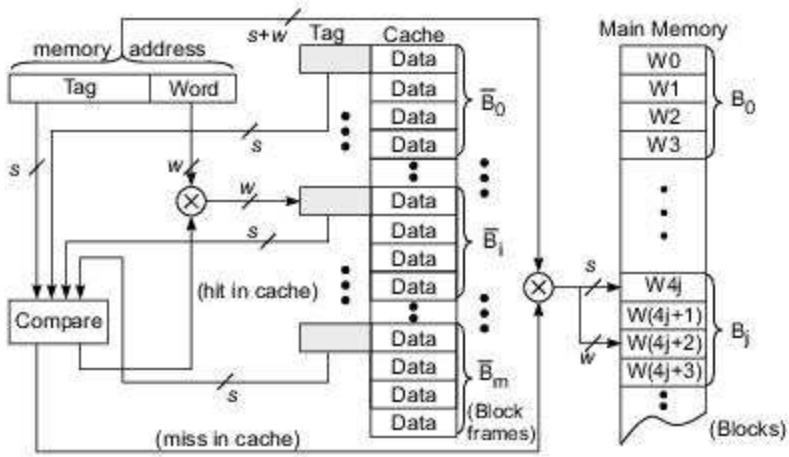
However, the rigid mapping may result in a poorer hit ratio than with the associative mappings to be introduced next. The scheme also prohibits parallel virtual address translation. The hit ratio may drop sharply if many addressed blocks have to map into the same block frame. For this reason, direct-mapped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

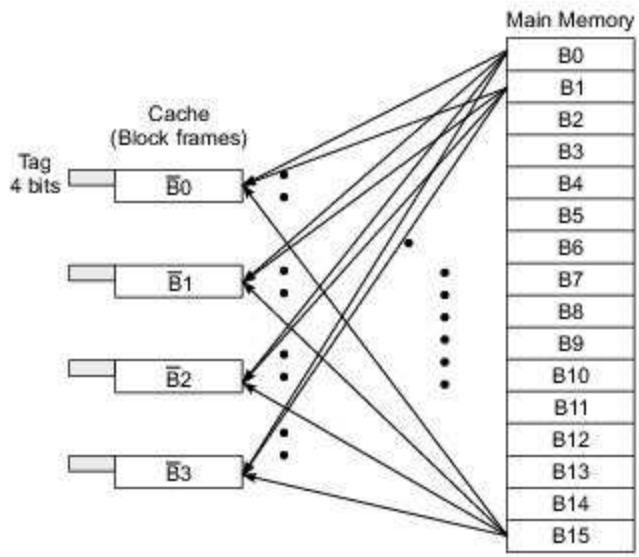
The name *fully associative cache* is derived from the fact that an m-way associative search requires the tag to be compared with all block tags in the cache. This scheme offers the greatest flexibility in implementing block replacement policies for a higher hit ratio.

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



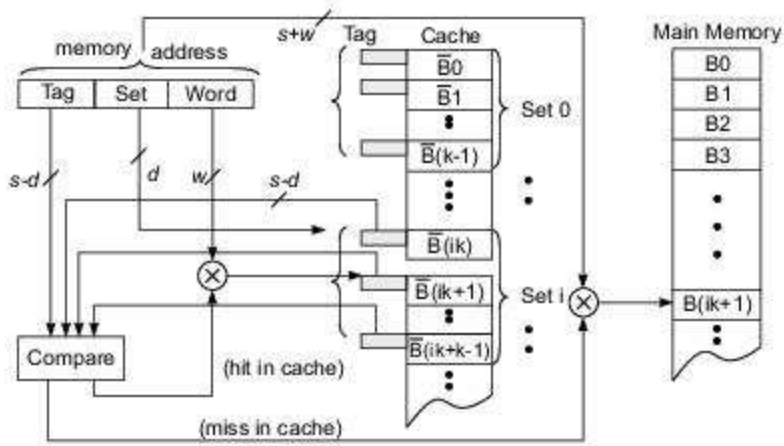
(b) Every block is mapped to any of the four block frames identified by the tag

Fig. 5.10 Fully associative cache organization and a mapping example

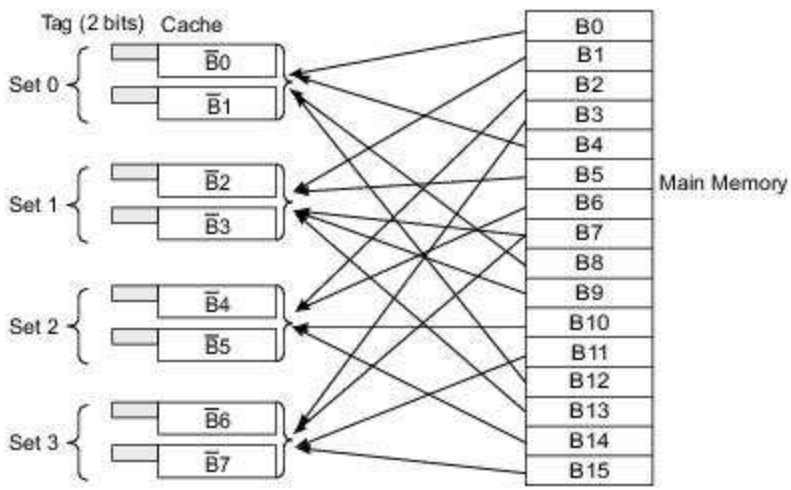
5.2.3 Set-Associative and Sector Caches

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 $\overline{B}_f$ in a set S_i defined below. The matched tag identifies the current block which resides in the frame.

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

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.

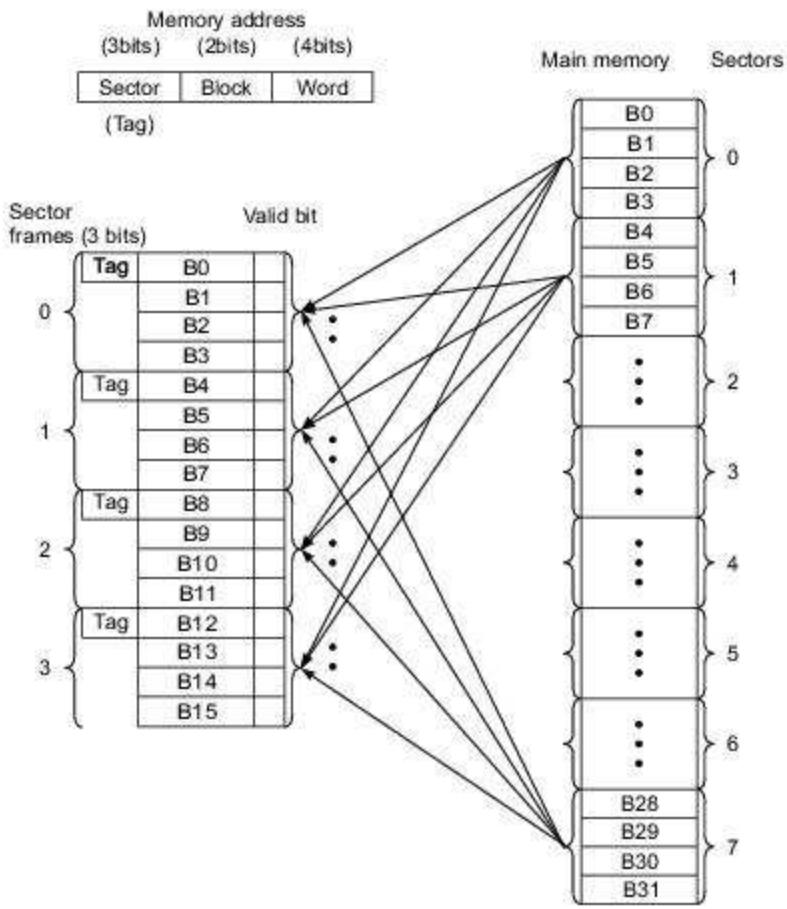


Fig. 5.12 A four-way sector mapping cache organization

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.

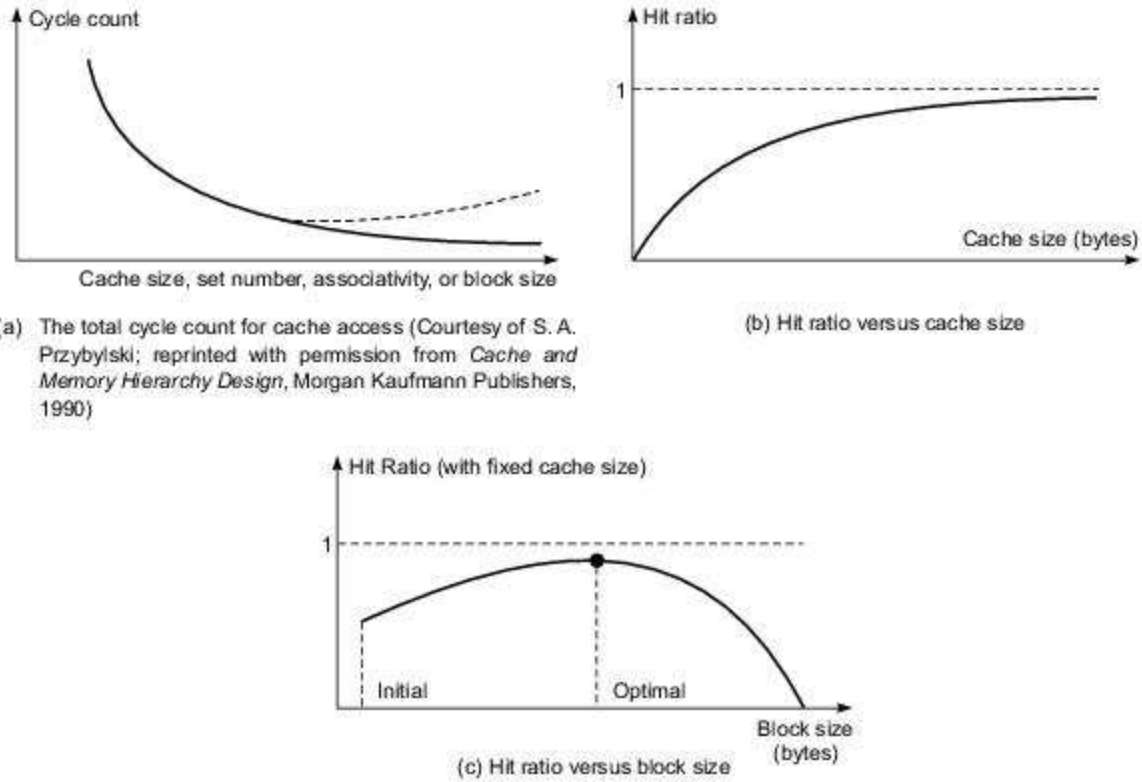
The cycle counts are not credible unless detailed simulation of all aspects of a memory hierarchy is performed. The write-through or write-back policies also affect the cycle count. Cache size, block size, set number, and associativity all affect the cycle count as illustrated in Fig. 5.13.

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.

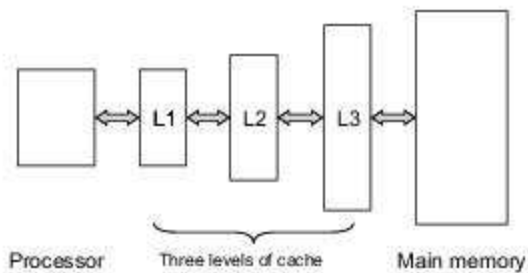


Fig. 5.14 Three levels of cache between processor and main memory

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.

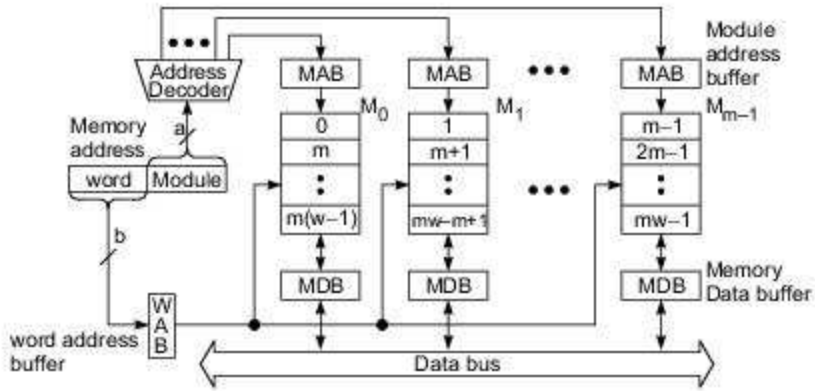
Consider a main memory formed with $m = 2^a$ memory modules, each containing $w = 2^b$ words of memory cells. The total memory capacity is $m \cdot w = 2^{a+b}$ words. These memory words are assigned linear addresses. Different ways of assigning linear addresses result in different memory organizations.

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

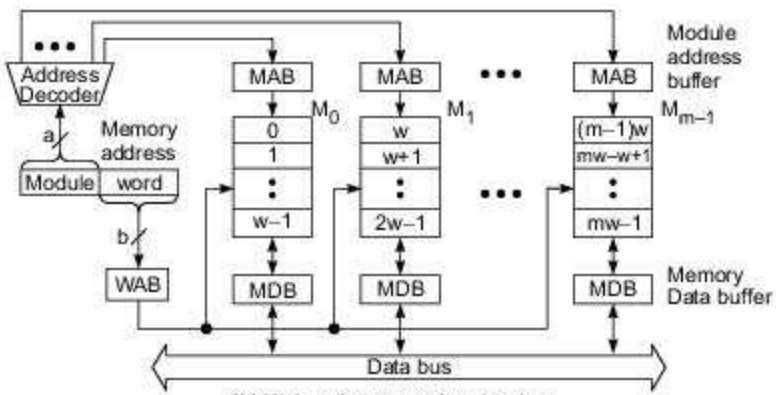
access may correspond to the size of a cache block (cache line) or to that of several cache blocks. Therefore, it is desirable to design the memory to facilitate block access of contiguous words.

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)

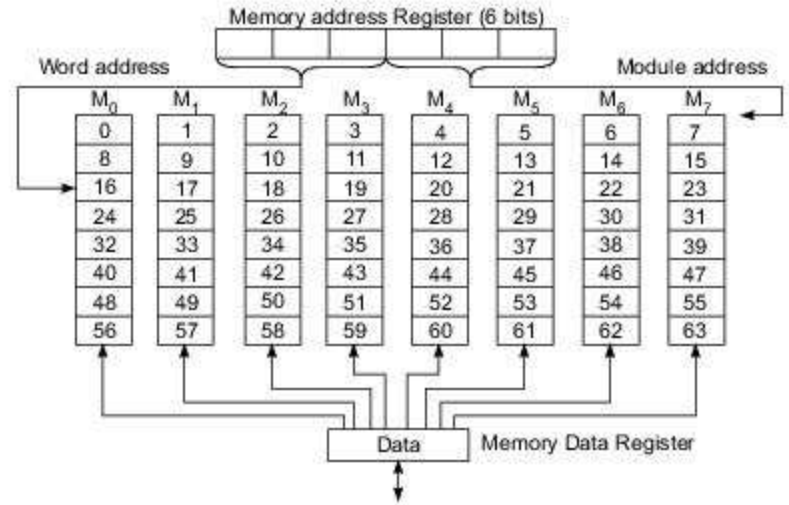


(b) High-order m-way interleaving

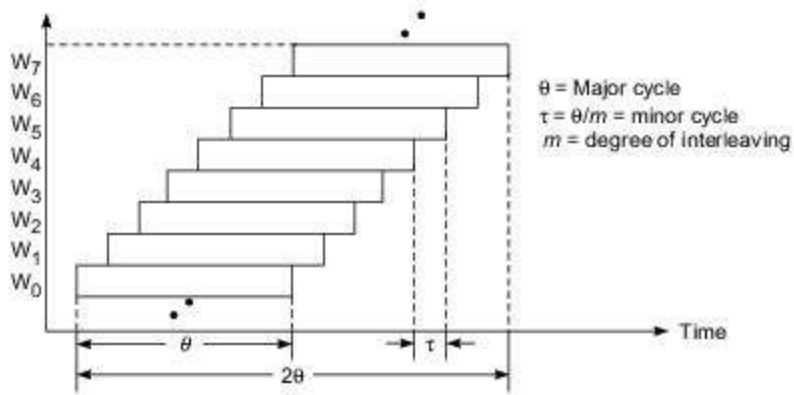
Fig. 5.15 Two interleaved memory organizations with $m = 2^a$ modules and $w = 2^b$ words per module (word addresses shown in boxes)

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

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

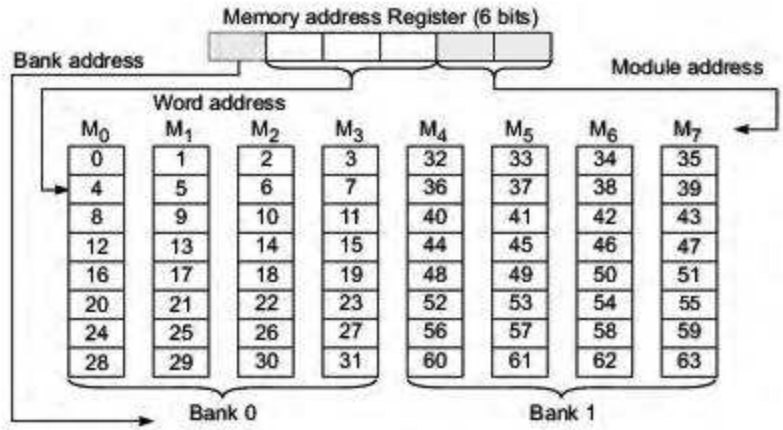
When $n \to \infty$ (very long vector), $t_1 \to \theta/m = \tau$ as derived in Eq. 5.4. As $n \to 1$ (scalar access), $t_1 \to \theta$. Equation 5.6 conveys the message that interleaved memory appeals to pipelined access of long vectors; the longer the better.

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.

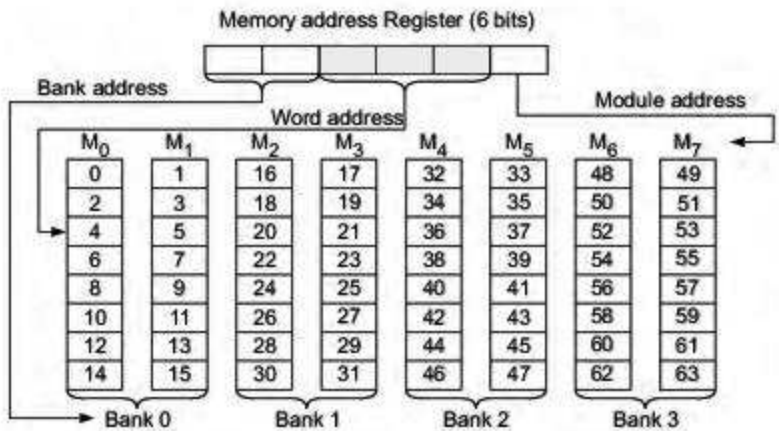
This makes it easier to isolate faulty memory modules in a *memory bank* of m memory modules. When one module failure is detected, the remaining modules can still be used by opening a window in the address space. This fault isolation cannot be carried out in a low-order interleaved memory, in which a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

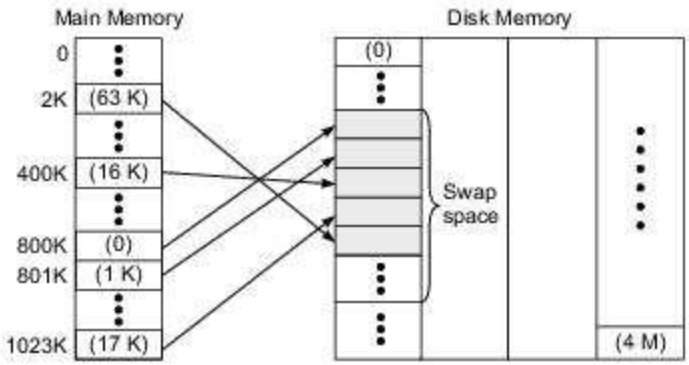
First, the swapping policy can be made either nonpreemptive or preemptive. In *nonpreemptive allocation*, the incoming block can be placed only in a free region of the main memory. A *preemptive allocation* scheme allows the placement of an incoming block in a region presently occupied by another process.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

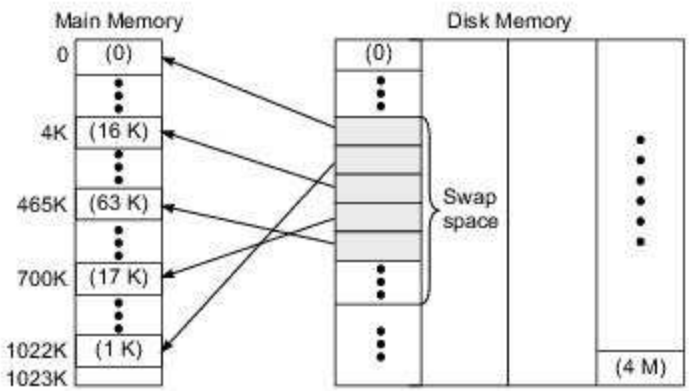
The preemptive scheme offers more flexibility, but it requires mechanisms be established to determine which pages or processes are to be swapped out and to avoid *thrashing* caused by an excessive amount of swapping between memory levels. This implies that preemptive allocation schemes are more complex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

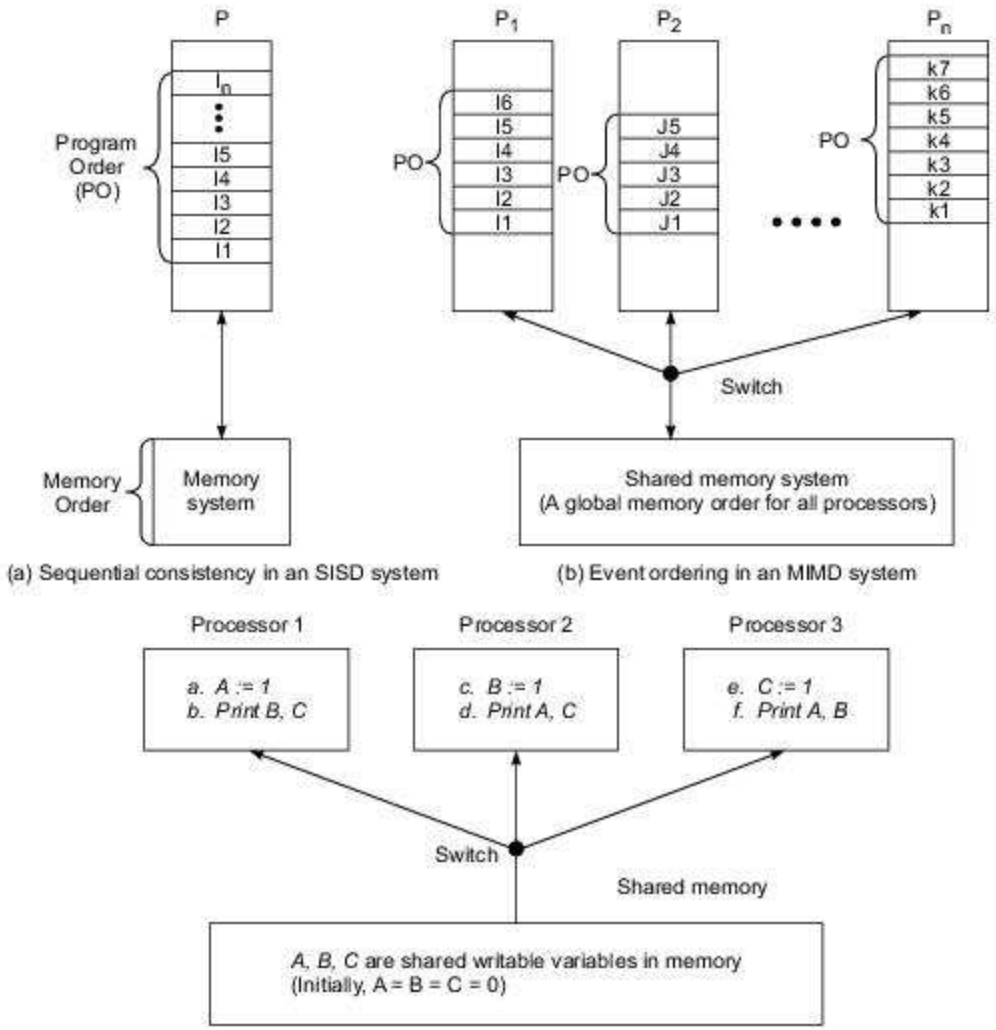


Fig. 5.19 The access ordering of memory events in a uniprocessor and in a multiprocessor, respectively (Courtesy of Dubois and Briggs, *Tutorial Notes on Shared-Memory Multiprocessors*, Int. Symp. Computer Arch., May 1990)

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

condition for an atomic memory to be sequentially consistent is that all memory accesses must be performed to preserve all individual program orders.

In a multiprocessor with nonatomic memory accesses, having individual program orders that conform is not a sufficient condition for sequential consistency. In a cache/network-based multiprocessor, the system can be nonatomic if an invalidation signal does not reach all processors at the same time. Thus a *store* is inherently nonatomic in such an architecture unless special hardware mechanisms are provided to assure atomicity. Only in atomic systems can the ordering of memory events be strongly ordered to make the program order consistent with the memory-access order.

With a nonatomic memory system, the multiprocessor cannot be strongly ordered. Thus weak ordering is very much desired in a multiprocessor with nonatomic memory accesses. The above discussions lead to the division between strong and weak consistency models to be described in the next two subsections.

5.4.2 Sequential Consistency Model

The *sequential consistency* (SC) memory model is widely understood among multiprocessor designers. In this model, the *loads*, *stores*, and *swaps* of all processors appear to execute serially in a single global memory order that conforms to the individual program orders of the processors, as illustrated in Fig. 5.20. Two definitions of SC model are given below.

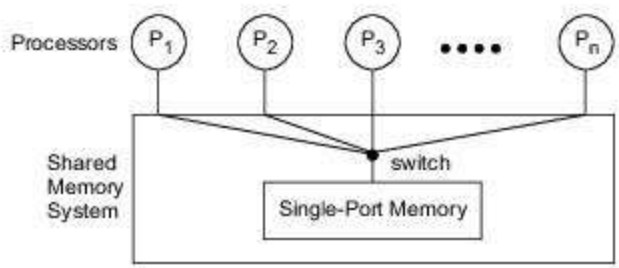


Fig. 5.20 Sequential consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, 1992)

Lamport's Definition Lamport (1979) defined *sequential consistency* as follows: A multiprocessor system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Dubois, Scheurich, and Briggs (1986) have provided the following two sufficient conditions to achieve sequential consistency in shared-memory access:

(a) Before a *load* is allowed to perform with respect to any other processor, all previous *load* accesses must be globally performed and all previous *store* accesses must be performed with respect to all processors.

(b) Before a *store* is allowed to perform with respect to any other processor, all previous *load* accesses must be globally performed and all previous *store* accesses must be performed with respect to all processors.

Sindhu, Frailong, and Cekleov (1992) have specified the sequential consistency memory model with the following five axioms:

(1) A *load* by a processor always returns the value written by the latest *store* to the same location by other processors.
(2) The memory order conforms to a total binary order in which shared memory is accessed in real time over all *loads* and *stores* with respect to all processor pairs and location pairs.
(3) If two operations appear in a particular program order, then they appear in the same memory order.
(4) The *swap* operation is atomic with respect to other *stores*. No other *store* can intervene between the *load* and *store* parts of a *swap*.
(5) All *stores* and *swaps* must eventually terminate.

Lamport's definition sets the basic spirit of sequential consistency. The memory access constraints imposed by Dubois et al. are refined from Lamport's definition with respect to atomicity. The conditions on sequential consistency specified by Sindhu et al. are further refined with respect to partial ordering relations. Implementation requirements of these constraints are discussed below.

Implementation Considerations Figure 5.20 shows that the shared memory consists of a single port that is able to service exactly one operation at a time, and a switch that connects this memory to one of the processors for the duration of each memory operation. The order in which the switch is thrown from one processor to another determines the global order of memory-access operations.

The sequential consistency model implies total ordering of *stores/loads* at the instruction level. This should be transparent to all processors. In other words, sequential consistency must hold for any processor in the system.

A conservative multiprocessor designer may prefer the sequential consistency model, in which consistency is enforced by hardware on-the-fly. Memory accesses are atomic and strongly ordered, and confusion can be avoided by having all processors/caches wait sufficiently long for unexpected events.

Strong ordering of all shared-memory accesses in the sequential consistency model preserves the program order in all processors. A sequentially consistent multiprocessor cannot determine whether the system is a multitasking uniprocessor or a multiprocessor. Interprocessor communication can be implemented with simple *loads/stores*, such as Dekker's algorithm for synchronized entry into a critical section by multiple processors. All memory accesses must be globally performed in program order.

A processor cannot issue another access until the most recently shared writable memory access by a processor has been globally performed. This may require the propagation of all shared-memory accesses to all processors, which is rather time-consuming and costly.

Most multiprocessors have implemented the sequential consistency model because of its simplicity. However, the model may lead to rather poor memory performance due to the imposed strong ordering of memory events. This is especially true when the system becomes very large. Thus sequential consistency reduces the scalability of a multiprocessor system.

5.4.3 Weak Consistency Models

The multiprocessor memory model may range anywhere from strong (or sequential) consistency to various degrees of weak consistency. In this section, we describe the *weak consistency* model introduced by Dubois et al. (1986) and a TSO model introduced with the SPARC architecture.

The DSB Model Dubois, Scheurich, and Briggs (1986) have derived a weak consistency memory model by relating memory request ordering to synchronization points in the program. We call this the DSB model specified by the following three conditions:

(1) All previous *synchronization* accesses must be performed, before a *load* or a *store* access is allowed to perform with respect to any other processor.
(2) All previous *load* and *store* accesses must be performed, before a *synchronization* access is allowed to perform with respect to any other processor
(3) *Synchronization* accesses are sequentially consistent with respect to one another.

These conditions provide a weak ordering of memory-access events in a multiprocessor. The dependence conditions on shared variables are weaker in such a system because they are only limited to hardware-recognized synchronizing variables. Buffering is allowed in *write buffers* except for operations on hardware-recognized synchronizing variables. Buffering memory accesses in multiprocessors can enhance the shared memory performance.

With different restrictions on the memory-access ordering, many different weak memory models can be similarly defined. The following is another weak consistency model, called the TSO (total store order), developed by the SPARC architecture group at Sun Microsystems.

Example 5.9 The TSO weak consistency model used in SPARC architecture (Sun Microsystems, Inc., 1990 and Sindhu et al., 1992)

Figure 5.21 shows the weak consistency TSO model developed by Sun Microsystems' SPARC architecture group (1990). Sindhu et al. described that the *stores* and *swaps* issued by a processor are placed in a dedicated store buffer for the processor, which is operated as first-in-first-out. Thus the order in which memory executes these operations for a given processor is the same as the order in which the processor issued them (in program order).

The memory order corresponds to the order in which the switch is thrown from one processor to another. This was described by Sindhu et al. as follows: A *load* by a processor first checks its store buffer to see if it contains a *store* to the same location. If it does, then the *load* returns the value of the most recent such *store*. Otherwise, the *load* goes directly to memory. Since not all *loads* go to memory immediately, *loads* in general do not appear in memory order. A processor is logically blocked from issuing further operations until the *load* returns a value. A *swap* behaves like a *load* and a *store*. It is placed in the store buffer like a *store*, and it blocks the processor like a *load*. In other words, the *swap* blocks until the store buffer is empty and then proceeds to the memory.

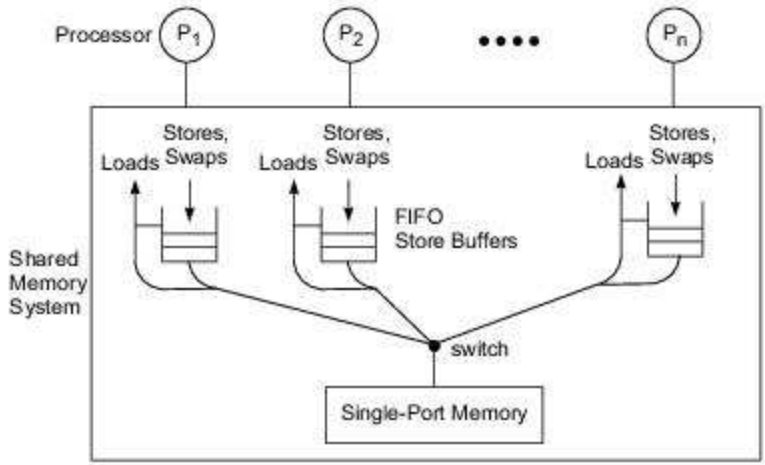


Fig. 5.21 The TSO Weak consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, 1992)

A TSO Formal Specification Sindhu, Frailong, and Cekleov (1992) have specified the TSO weak consistency model with six behavioral axioms. Only an intuitive description of their axioms is abstracted below:

(1) A *load* access is always returned with the latest *store* to the same memory location issued by any processor in the system.
(2) The memory order is a total binary relation over all pairs of *store* operations.
(3) If two *stores* appear in a particular program order, then they must also appear in the same memory order.
(4) If a memory operation follows a *load* in program order, then it must also follow the *load* in memory order.
(5) A *swap* operation is atomic with respect to other *stores*. No other *store* can interleave between the *load* and *store* parts of a *swap*.
(6) All *stores* and *swaps* must eventually terminate.

Note that the above axioms (5) and (6) are identical to axioms (4) and (5) for the sequential consistency model. Axiom (1) covers the effects of both local and remote processors. Both axioms (2) and (3) are weakened from the corresponding axioms (2) and (3) for the sequential consistency model. Axiom (4) states that the *load* operations do not have to be weakened, as far as ordering is concerned. For a formal axiomatic specification, the reader is referred to the original paper by Sindhu et al. (1992).

Comparison of Memory Models In summary, the weak consistency model may offer better performance than the sequential consistency model at the expense of more complex hardware/software support and more programmer awareness of the imposed restrictions. The relative merits of the strong and weak memory models have been subjects of debate in the multiprocessor research community.

The DSB and the TSO are two different weak-consistency memory models. The DSB model is weakened by enforcing sequential consistency at synchronization points. The TSO model is weakened by treating *reads*, *stores*, and *swaps* differently using FIFO store buffers. The TSO model has been implemented in some SPARC architectures, while the DSB model has not been implemented in real systems yet.

Sindhu et al. (1992) have identified four system-level issues which also affect the choice of memory model. First, they suggest that one should consider extending the memory model from the processor level to the process level. To do this, one must maintain a process switch sequence. The second issue is the incorporation of I/O locations into the memory model. I/O operations may introduce even more side effects in addition to the normal semantics of *loads* and *stores*.

The third issue is code modification. In the SPARC architecture, synchronization of code modification is accomplished through the use of a *flush* instruction defined in the TSO model. Finally, they feel that the memory model should address the issue of how to incorporate pipelined processors and processors with noncoherent internal caches. The interested reader is referred to their original paper and the SPARC Architecture Manual for details of these issues.

Strong memory ordering introduces unnecessary processor/cache waiting time and reduces the amount of concurrency. Weak consistency has the potential to eliminate these shortcomings. Besides sequential consistency, DSB and TSO weak memory models, other memory consistency models, such as *processor consistency* and *release consistency*, will be treated in Chapter 9.

Summary

In this chapter we studied the functions and technical requirements of the system bus and cache memory, and also discussed some basic issues related to shared main memory in a multiprocessor system. We saw that the single system bus has performance limitations as processors become faster and the number of processors in the system increases. With this background, we shall study other system interconnect strategies in latter chapters.

The earliest multiprocessor systems were built around a single system bus. We studied the basic system requirements of the bus, i.e. addressing, timing, arbitration, transaction modes, interrupts, and so on. As one specific example of a bus specification, we looked at Futurebus+. However, bus-based communication of the earlier multiprocessors, usually based on a single backplane, has limited scalability.

With rapidly increasing computing power, it became clear that communication between different sub-systems of a computer system—and also between computer systems—is as important as the storage and processing of data. As demands on the system interconnect grew rapidly, performance limitations in using a single bus such as Futurebus+ became obvious. Scalable Coherent Interconnect (SCI) and InfiniBand, which grew out of the unsuccessful Futurebus+ effort, employ point-to-point links and packet-switching, and can therefore support highly scalable systems.

Cache memories are provided between the processor and main memory to bridge the huge speed mismatch between these two sub-systems. Over the last two or three decades, this speed mismatch has grown larger, because processor speeds have risen much faster than main memory speeds. Addressing models, direct mapped *versus* set associative cache, block size, and other relevant cache performance

issues were discussed. Multiple levels of cache are often employed; based on their design goals, different models of the same processor family may employ different multi-level cache designs.

Interleaving of memory modules is a technique for achieving higher aggregate memory bandwidth in support of higher system performance. Different schemes for memory interleaving have been considered, along with related performance issues. Memory allocation schemes such as paging and swapping were also considered.

For multiprocessor systems with shared memory, apart from the strict sequential consistency model, weaker consistency models are also considered—the aim being to achieve a greater degree of parallelism, and thereby higher system performance. Basic concepts of atomicity of memory accesses and event ordering are used to define memory consistency models; two specific such models, DSB and TSO, were discussed.

Exercises

Problem 5.1 This is an illustrative example of a design specification of a backplane bus for a shared-memory multiprocessor with 4 processor boards and 16 memory boards under the following assumptions:

- Bus clock rate = 200 MHz.
- Memory word length = 64 bits; processors always request data in blocks of four words.
- Memory access time = 100 ns.
- Shared address space = 2^{40} words.
- Maximum number of signal lines available on the backplane is 96.
- Synchronous timing protocol.
- Neglect buffer and propagation delays.

Specify the following in your design of the bus system:

(a) Maximum bus bandwidth.
(b) Effective bus bandwidth (worst case).
(c) Arbitration scheme.
(d) Name and functionality of each of the signal lines.
(e) Number of slots required on the backplane.

Justify any additional assumptions you need to make.

Problem 5.2 Describe the daisy-chaining (Fig. 5.4) and the *distributed arbiter* (Fig. 5.5b) for bus arbitration in a multiprocessor system. State the advantages and shortcomings of each case from both the implementational and operational points of view.

Problem 5.3 Read the paper by Mudge et al. (1987) on multiple-bus systems and solve the following problems:

(a) Find the maximum bandwidth for a multiprocessor system using b buses, where $b > m$ and m is the number of memory modules and the system has n processors.
(b) Prove that $BW_b < np$, where $p > 0$ is the probability that an arbitrary processor will generate a request to access the shared memory at the start of a memory cycle.

Problem 5.4 Estimate the effective MIPS rate of a bus-connected multiprocessor system under the following assumptions. The system has 16 processors, each connected to an on-board private cache which is connected to a common bus. Globally shared memory is also connected to the bus. The private cache and the shared memory form a two-level access hierarchy.

Each processor is rated 500 MIPS if a 100% cache hit ratio is assumed. On the average, each instruction needs 0.2 memory access. The read access and write access are assumed equally probable.

For a crude approximation, consider only the penalty caused by shared-memory access and ignore all other overheads. The cache is targeted to maintain a hit ratio of 0.95. A cache access on a read-hit takes 2 ns; that on a write-hit takes 4 ns with a write-back scheme, and with a write-through scheme it needs 100 ns.

When a cache block is to be replaced, the probability that it is dirty is estimated as 0.1. An average block transfer time between the cache and shared memory via the bus is 100 ns.

(a) Derive the effective memory-access times per instruction for the write-through and write-back caches separately.

(b) Calculate the effective MIPS rate for each processor. Determine an upper bound on the effective MIPS rate of the 16-processor system. Discuss why the upper bound cannot be achieved by considering the memory penalty alone.

Problem 5.5 Explain the following terms associated with cache and memory architectures.

(a) Low-order memory interleaving.

(b) Physical address cache versus virtual address cache.

(c) Atomic versus nonatomic memory accesses.

(d) Memory bandwidth and fault tolerance.

Problem 5.6 Explain the following terms associated with cache design:

(a) Write-through versus write-back caches.

(b) Cacheable versus noncacheable data.

(c) Private caches versus shared caches.

(d) Cache flushing policies.

(e) Factors affecting cache hit ratios.

Problem 5.7 Consider the simultaneous execution of the three programs on the three processors shown in Fig. 5.19c. Answer the following questions with reasoning or supported by computer simulation results:

(a) List the 90 execution interleaving orders of the six instructions $\{a, b, c, d, e, f\}$ which will preserve the individual program orders. The corresponding output patterns (6-tuples) should be listed accordingly.

(b) Can all 6-tuple combinations be generated out of the 720 non-program-order interleavings? Justify the answer with reasoning and examples.

(c) We have assumed atomic memory access in this example. Explain why the output 011001 is not possible in an atomic memory multiprocessor system if individual program orders are preserved.

(d) Suppose nonatomic memory access is allowed in the above multiprocessor. For example, an invalidation does not reach all private caches at the same time. Prove that 011001 is possible even if all instructions were executed in program order but other processors did not observe them in program order.

Problem 5.8 The main memory of a computer is organized as 64 blocks, with a block size of eight words. The cache has eight block frames. In parts (a) through (d), show the mappings from the numbered blocks in main memory to the block frames in the cache. Draw all lines showing the mappings as clearly as possible.

(a) Show the direct mapping and the address bits that identify the tag field, the block number, and the word number.

(b) Show the fully associative mapping and the address bits that identify the tag field and the word number.

(c) Show the two-way set-associative mapping and the address bits that identify the tag field, the set number, and the word number.

(d) Show the sector mapping with four blocks per sector and the address bits that identify

the sector number, the block number, and the word number.

Problem 5.9 Consider a cache (M_1) and memory (M_2) hierarchy with the following characteristics:

M_1: 64K words, 5 ns access time

M_2: 4M words, 40 ns access time

Assume eight-word cache blocks and a set size of 256 words with set-associative mapping.

(a) Show the mapping between M_2 and M_1.

(b) Calculate the effective memory-access time with a cache hit ratio of $h = 0.95$.

Problem 5.10 Consider a main memory consisting of four memory modules with 256 words per module. Assume 16 words in each cache block. The cache has a total capacity of 256 words. Set-associative mapping is used to allocate cache blocks to block frames. The cache is divided into four sets.

(a) Show the address assignment for all 1024 words in a four-way low-order interleaved organization of the main memory.

(b) How many blocks are there in the main memory? How many block frames are there in the cache?

(c) Explain the bit fields needed for addressing each word in the two-level memory system.

(d) Show the mapping from the blocks in the main memory to the sets in the cache and explain how to use the tag field to locate a block frame within each set.

Problem 5.11

(a) A uniprocessor system uses separate instruction and data caches with hit ratios h_i and h_d, respectively. The access time from the processor to either cache is c clock cycles, and the block transfer time between the caches and main memory is b clock cycles.

Among all memory references made by the CPU, f_i is the percentage of references to instructions. Among blocks replaced in the data cache, f_{dir} is the percentage of *dirty* blocks. (*Dirty* means that the cache copy is different from the memory copy.)

Assuming a write-back policy, determine the effective memory-access time in terms of h_i, h_d, c, b, f_i, and f_{dir} for this memory system.

(b) The processor memory system described in part (a) is used to construct a bus-based shared-memory multiprocessor. Assume that the hit ratio and access times remain the same as in part (a). However, the effective memory-access time will be different because every processor must now handle cache invalidation in addition to reads and writes.

Let f_{inv} be the fraction of data references that cause invalidation signals to be sent to other caches. The processor sending the invalidation signal requires i clock cycles to complete the invalidation operation. Other processors are not involved in the invalidation process. Assuming a write-back policy again, determine the effective memory-access time for this multiprocessor.

Problem 5.12 A computer system has a 128-byte cache. It uses four-way set-ssociative mapping with 8 bytes in each block. The physical address size is 32 bits, and the smallest addressable unit is 1 byte.

(a) Draw a diagram showing the organization of the cache and indicating how physical addresses are related to cache addresses.

(b) To what block frames of the cache can the address $000010AF_{16}$ be assigned?

(c) If the addresses $000010AF_{16}$ and $FFFF7Axy_{16}$ can be simultaneously assigned to the same cache set, what values can the address digits x and y have?

Problem 5.13 Consider a shared-memory multiprocessor system with p processors. Let m be the average number of global memory references per instruction execution on a typical processor.

Let t be the average access time to the shared memory and x be the MIPS rate of a uniprocessor

using local memory. Consider the execution of n instructions on each processor of the multiprocessor.

(a) Determine the effective MIPS rate of the multiprocessor in terms of the parameters m, t, x, n, and p.

(b) Suppose a multiprocessor has $p = 32$ RISC processors, $m = 0.4$, and $t = 0.01$ μs. What is the MIPS rate of each processor (i.e. $x = ?$) needed to achieve a multiprocessor performance of 5600 MIPS effectively?

(c) Suppose $p = 32$ CISC processors with $x = 200$ MIPS each are used in the above multiprocessor system with $m = 1.6$ and $t = 0.01$ μs. What will be the effective MIPS rate?

Problem 5.14 Consider a RISC-based shared-memory multiprocessor with p processors, each having its own instruction cache and data cache. The peak performance rating of each processor (assuming a 100% hit ratio in both caches) is x MIPS. You are required to derive a performance formula, taking into account cache misses, shared-memory accesses, and synchronization overhead.

Assume that on the average α percent of the instructions executed are for synchronization purpose, and the penalty for each synchronization operation is an additional t_s μs. The number of memory accesses per instruction is m. Among all memory references made by the CPU, f_i is the percentage of references to instructions. Assume that the instruction cache and data cache have hit ratios h_i and h_d, respectively, after a long period of program tracing on the machine. On cache misses, instructions and data are accessed from the shared memory with an average access time t_m μs.

(a) Derive an expression for approximating the effective MIPS rate of this multiprocessor in terms of p, x, m, f_i, h_i, h_d, t_m, α, and t_s. Note that f_i, h_i, h_d, and α are all fractions and t_m and t_s are measured in μs. Ignore the cache-access time and other system overheads in your derivation.

(b) Suppose $m = 0.4$, $f_i = 0.5$, $h_i = 0.95$, $h_d = 0.8$, $\alpha = 0.02$, $x = 500$, $t_m = 0.05$ μs, and $t_s = 1$ μs. Determine the minimum number of processors needed in the above multiprocessor system in order to achieve an effective MIPS rate of 2000.

(c) Suppose the total cost of all the caches and shared-memory is upper-bounded by $25,000. The cache memory costs $1.25/Kbyte, and the shared memory costs $0.1/Kbyte. With $p = 16$ processors, each having an instruction cache of capacity $S_i = 32$ Kbytes and a data cache of capacity $S_d = 64$ Kbytes, what is the maximum shared-memory capacity C_m (in Mbytes) that can be acquired within the budget limit?

Problem 5.15 Consider the following three interleaved memory designs for a main memory system with 16 memory modules. Each module is assumed to have a capacity of 1 Mbyte. The machine is byte-addressable.

Design 1: 16-way interleaving with one memory bank.
Design 2: 8-way interleaving with two memory banks.
Design 3: 4-way interleaving with four memory banks.

(a) Specify the address formats for each of the above memory organizations.

(b) Determine the maximum memory bandwidth obtained if only one memory module fails in each of the above memory organizations.

(c) Comment on the relative merits of the three interleaved memory organizations.

Problem 5.16 Consider a memory system for the erstwhile Cray 1 computer. There are $m = 16$ interleaved modules. The access time of a module is $t_a = 50$ ns and the memory cycle time is $t_c = 12.5$ ns. We know that for this memory system the maximum memory bandwidth of 80M words per second is achieved for vector loads/stores except when the stride is a multiple of 16 (bandwidth: 20M words per second) or a multiple of 8 (but not 16) (bandwidth: 40M words per second).

(a) Find the bandwidth for all strides for similar

systems but with the following parameters: t_c = 12.5 ns, t_a = 50 ns, m = 17.

(b) Repeat part (a) for the following parameters: t_c = 12.5 ns, t_a = 50 ns, m = 8.

Problem 5.17 Consider the concurrent execution of two programs by two processors with a shared memory. Assume that *A*, *B*, *C*, *D* are initialized to 0 and that a *Print* statement prints both arguments indivisibly at the same cycle. The output forms a 4-tuple as either *ADBC* or *BCAD*.

P_0:	P_1:
a. $A = 1$	d. $C = 1$
b. $B = 1$	e. $D = 1$
c. Print *A*, *D*	f. Print *B*, *C*

(a) List all execution interleaving orders of six statements which will preserve the individual program order.

(b) Assume program orders are preserved and all memory accesses are atomic; i.e., a store by one processor is immediately seen by all the remaining processors. List all the possible 4-tuple output combinations.

(c) Assume program orders are preserved but memory accesses are nonatomic; i.e., a store by one processor may be buffered so that some other processors may not immediately observe the update. List all possible 4-tuple output combinations.

Problem 5.18 Compare the relative merits of the four cache memory organizations:

(1) Direct-mapping cache
(2) Fully associative cache
(3) Set-associative cache
(4) Sector mapping cache

Answer the following questions with reasoning:

(a) In terms of hardware complexity and implementation cost, rank the four cache organizations with justification.

(b) With respect to flexibility in implementing block replacement algorithms, rank the four cache organizations and justify the ranking order.

(c) With each cache organization, explain the effects of block mapping policies on the hit ratio issues.

(d) Explain the effects of block size, set number, associativity, and cache size on the performance of a set-associative cache organization.

Problem 5.19 Explain the following terms associated with memory management:

(a) The role of a memory manager in an OS kernel.

(b) Preemptive versus nonpreemptive memory allocation policies.

(c) Swapping memory system and examples.

(d) Demand paging memory system and examples.

(e) Hybrid memory system and examples.

Problem 5.20 Compare the memory-access constraints in the following memory consistency models:

(a) Determine the similarities and subtle differences among the conditions on sequential consistency imposed by Lamport (1979), by Dubois et al. (1986), and by Sindhu et al. (1992), respectively.

(b) Repeat question (a) between the DSB model and the TSO model for weak consistency memory systems.

(c) A PSO (partial store order) model for weak consistency has been refined from the TSO model. Study the PSO specification in the paper by Sindhu et al. (1992) and compare the relative merits between the TSO and the PSO memory models.

6

Pipelining and Superscalar Techniques

This chapter deals with pipelining and superscalar design in processor development. We begin with a discussion of conventional linear pipelines and analyze their performance. A generalized pipeline model is introduced to include nonlinear interstage connections. Collision-free scheduling techniques are described for performing dynamic functions.

Specific techniques for building instruction pipelines, arithmetic pipelines, and memory-access pipelines are presented. The discussion includes instruction prefetching, internal data forwarding, software interlocking, hardware scoreboarding, hazard avoidance, branch handling, and instruction-issuing techniques. Both static and multifunctional arithmetic pipelines are designed. Superscalar design techniques are studied along with performance analysis.

6.1 LINEAR PIPELINE PROCESSORS

A *linear pipeline processor* is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to the other. In modern computers, linear pipelines are applied for instruction execution, arithmetic computation, and memory-access operations.

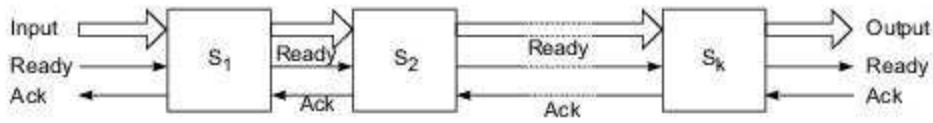
6.1.1 Asynchronous and Synchronous Models

A linear pipeline processor is constructed with k processing stages. External inputs (operands) are fed into the pipeline at the first stage S_1. The processed results are passed from stage S_i to stage S_{i+1}, for all $i = 1, 2, \ldots, k-1$. The final result emerges from the pipeline at the last stage S_k.

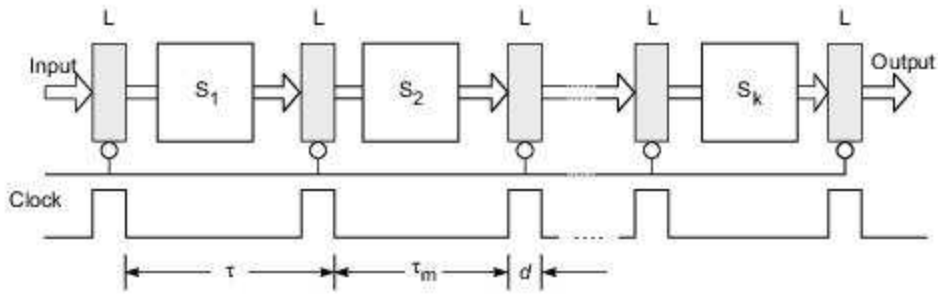
Depending on the control of data flow along the pipeline, we model linear pipelines in two categories: *asynchronous* and *synchronous*.

Asynchronous Model As shown in Fig. 6.1a, data flow between adjacent stages in an asynchronous pipeline is controlled by a handshaking protocol. When stage S_i is ready to transmit, it sends a *ready* signal to stage S_{i+1}. After stage S_{i+1} receives the incoming data, it returns an *acknowledge* signal to S_i.

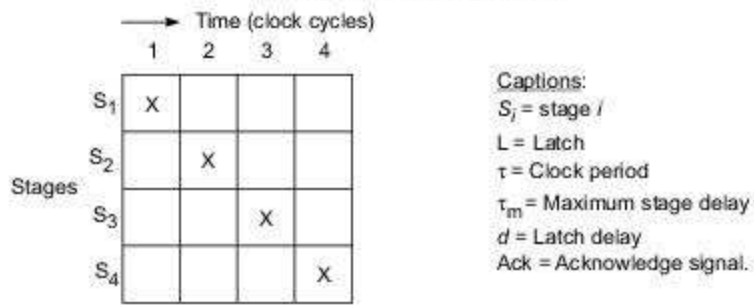
Asynchronous pipelines are useful in designing communication channels in message-passing multicomputers where pipelined wormhole routing is practiced (see Chapter 9). Asynchronous pipelines may have a variable throughput rate. Different amounts of delay may be experienced in different stages.



(a) An asynchronous pipeline model



(b) A synchronous pipeline model



(c) Reservation table of a four-stage linear pipeline

Fig. 6.1 Two models of linear pipeline units and the corresponding reservation table

Synchronous Model Synchronous pipelines are illustrated in Fig. 6.1b. Clocked latches are used to interface between stages. The latches are made with master-slave flip-flops, which can isolate inputs from outputs. Upon the arrival of a clock pulse, all latches transfer data to the next stage simultaneously.

The pipeline stages are combinational logic circuits. It is desired to have approximately equal delays in all stages. These delays determine the clock period and thus the speed of the pipeline. Unless otherwise specified, only synchronous pipelines are studied in this book.

The utilization pattern of successive stages in a synchronous pipeline is specified by a *reservation table*. For a linear pipeline, the utilization follows the diagonal streamline pattern shown in Fig. 6.1c. This table is essentially a space-time diagram depicting the precedence relationship in using the pipeline stages. For a k-stage linear pipeline, k clock cycles are needed for data to flow through the pipeline.

Successive tasks or operations are initiated one per cycle to enter the pipeline. Once the pipeline is filled up, one result emerges from the pipeline for each additional cycle. This throughput is sustained only if the successive tasks are independent of each other.

6.1.2 Clocking and Timing Control

The *clock cycle* τ of a pipeline is determined below. Let τ_i be the time delay of the circuitry in stage S_i and d the time delay of a latch, as shown in Fig. 6.1b.

Clock Cycle and Throughput Denote the *maximum stage delay* as τ_m and we can write τ as

$$\tau = \max_i \{\tau_i\}_1^k + d = \tau_m + d \tag{6.1}$$

At the rising edge of the clock pulse, the data is latched to the master flip-flops of each latch register. The clock pulse has a width equal to d. In general, $\tau_m >> d$ by one to two orders of magnitude. This implies that the maximum stage delay τ_m dominates the clock period.

The *pipeline frequency* is defined as the inverse of the clock period:

$$f = \frac{1}{\tau} \tag{6.2}$$

If one result is expected to come out of the pipeline per cycle, f represents the *maximum throughput* of the pipeline. Depending on the initiation rate of successive tasks entering the pipeline, the *actual throughput* of the pipeline may be lower than f. This is because more than one clock cycle has elapsed between successive task initiations.

Clock Skewing Ideally, we expect the clock pulses to arrive at all stages (latches) at the same time. However, due to a problem known as *clock skewing*, the same clock pulse may arrive at different stages with a time offset of s. Let t_{max} be the time delay of the longest logic path within a stage and t_{min} that of the shortest logic path within a stage.

To avoid a race in two successive stages, we must choose $\tau_m \geq t_{max} + s$ and $d \leq t_{min} - s$. These constraints translate into the following bounds on the clock period when clock skew takes effect:

$$d + t_{max} + s \leq \tau \leq \tau_m + t_{min} - s \tag{6.3}$$

In the ideal case $s = 0$, $t_{max} = \tau_m$, and $t_{min} = d$. Thus, we have $\tau = \tau_m + d$, consistent with the definition in Eq. 6.1 without the effect of clock skewing.

6.1.3 Speedup, Efficiency, and Throughput

Ideally, a linear pipeline of k stages can process n tasks in $k + (n - 1)$ clock cycles, where k cycles are needed to complete the execution of the very first task and the remaining $n - 1$ tasks require $n - 1$ cycles. Thus the total time required is

$$T_k = [k + (n - 1)]\tau \tag{6.4}$$

where τ is the clock period. Consider an equivalent-function nonpipelined processor which has a *flow-through delay* of $k\tau$. The amount of time it takes to execute n tasks on this nonpipelined processor is $T_1 = nk\tau$.

Speedup Factor The speedup factor of a k-stage pipeline over an equivalent non pipelined processor is defined as

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k + (n-1)} \tag{6.5}$$

Note 6.1 Pipelined versus non-pipelined processors

If each pipeline stage has a stage delay of τ, then clearly an instruction passing through k pipeline stages in a processor sees a total latency of $k\tau$. Now suppose we also have a non-pipelined processor for the same instruction set, using the same technology. This non-pipelined processor need not present a latency of $k\tau$ to every instruction, because it does not have k separate stages for an instruction to pass through. Since the non-pipelined processor would have a more compact hardware design, we can expect that the average latency seen by instructions on this processor will be smaller than $k\tau$.

In other words, the advantage of a pipelined processor lies in its instruction throughput; in terms of instruction latency, the non-pipelined version can in fact be expected do better. However, for the comparative analysis here, we have assumed that the instruction latency on the non-pipelined version is also $k\tau$. This is a simplification which does not change substantially the conclusion reached.

Example 6.1 Pipeline speedup versus stream length

The maximum speedup is $S_k \rightarrow k$ as $n \rightarrow \infty$. This maximum speedup is very difficult to achieve because of data dependences between successive tasks (instructions), program branches, interrupts, and other factors to be studied in subsequent sections.

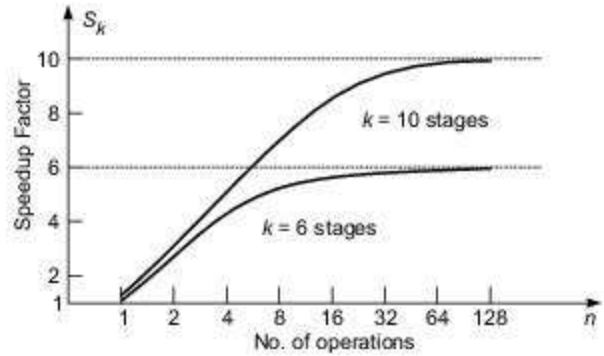
Figure 6.2a plots the speedup factor as a function of n, the number of tasks (operations or instructions) performed by the pipeline. For small values of n, the speedup can be very poor. The smallest value of S_k is 1 when $n = 1$.

The larger the number k of subdivided pipeline stages, the higher the potential speedup performance. When $n = 64$, an eight-stage pipeline has a speedup value of 7.1 and a four-stage pipeline has a speedup of 3.7. However, the number of pipeline stages cannot increase indefinitely due to practical constraints on costs, control complexity, circuit implementation, and packaging limitations. Furthermore, the stream length n also affects the speedup; the longer the better in using a pipeline.

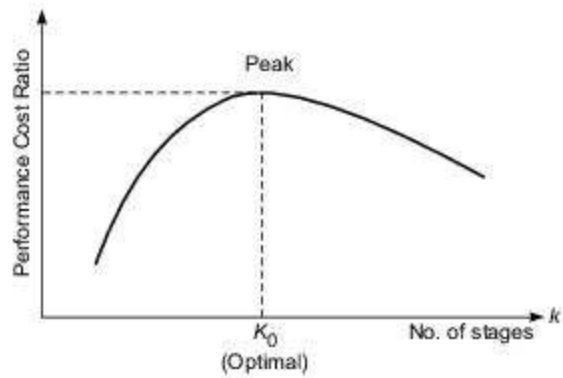
Optimal Number of Stages In practice, most pipelining is staged at the functional level with $2 \leq k \leq 15$. Very few pipelines are designed to exceed 10 stages in real computers. The optimal choice of the number of pipeline stages should be able to maximize the performance/cost ratio for the target processing load.

Let t be the total time required for a nonpipelined sequential program of a given function. To execute the same program on a k-stage pipeline with an equal flow-through delay t, one needs a clock period of $p = t/k + d$, where d is the latch delay. Thus, the pipeline has a maximum throughput of $f = 1/p = 1/(t/k + d)$. The total pipeline cost is roughly estimated by $c + kh$, where c covers the cost of all logic stages and h represents the cost of each latch. A pipeline *performance/cost ratio* (PCR) has been defined by Larson (1973):

$$PCR = \frac{f}{c + kh} = \frac{1}{(t/k + d)(c + kh)} \tag{6.6}$$



(a) Speedup factor as a function of the number of operations (Eq. 6.5)



(b) Optimal number of pipeline stages (Eqs. 6.6 and 6.7)

Fig. 6.2 Speedup factors and the optimal number of pipeline stages for a linear pipeline unit

Figure 6.2b plots the PCR as a function of k. The peak of the PCR curve corresponds to an optimal choice for the number of desired pipeline stages:

$$k_0 = \sqrt{\frac{t \cdot c}{d \cdot h}} \tag{6.7}$$

where t is the total flow-through delay of the pipeline. Thus the total stage cost c, the latch delay d, and the latch cost h must be considered to achieve the optimal value k_0.

Efficiency and Throughput The *efficiency* E_k of a linear k-stage pipeline is defined as

$$E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)} \tag{6.8}$$

Obviously, the efficiency approaches 1 when $n \to \infty$, and a lower bound on E_k is $1/k$ when $n = 1$. The *pipeline throughput* H_k is defined as the number of tasks (operations) performed per unit time:

$$H_k = \frac{n}{[k + (n-1)]\tau} = \frac{nf}{k + (n-1)} \tag{6.9}$$

The *maximum throughput f* occurs when $E_k \to 1$ as $n \to \infty$. This coincides with the speedup definition given in Chapter 3. Note that $H_k = E_k \cdot f = E_k/\tau = S_k/k\tau$. Other relevant factors of instruction pipelines will be discussed in Chapters 12 and 13.

6.2 NONLINEAR PIPELINE PROCESSORS

A *dynamic pipeline* can be reconfigured to perform variable functions at different times. The traditional linear pipelines are static pipelines because they are used to perform fixed functions.

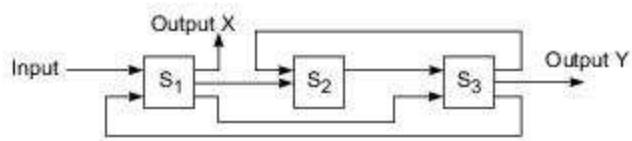
A dynamic pipeline allows feedforward and feedback connections in addition to the streamline connections. For this reason, some authors call such a structure a *nonlinear pipeline*.

6.2.1 Reservation and Latency Analysis

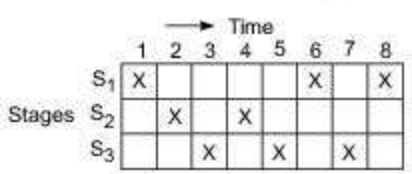
In a static pipeline, it is relatively easy to partition a given function into a sequence of linearly ordered subfunctions. However, function partitioning in a dynamic pipeline becomes quite involved because the pipeline stages are interconnected with loops in addition to streamline connections.

A multifunction dynamic pipeline is shown in Fig. 6.3a. This pipeline has three stages. Besides the *streamline connections* from S_1 to S_2 and from S_2 to S_3, there is a *feed forward connection* from S_1 to S_3 and two *feedback connections* from S_3 to S_2 and from S_3 to S_1.

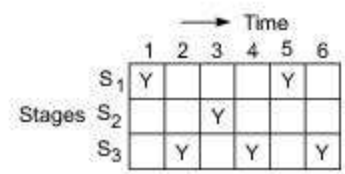
These feedforward and feedback connections make the scheduling of successive events into the pipeline a nontrivial task. With these connections, the output of the pipeline is not necessarily from the last stage. In fact, following different dataflow patterns, one can use the same pipeline to evaluate different functions.



(a) A three-stage pipeline



(b) Reservation table for function X



(c) Reservation table for function Y

Fig. 6.3 A dynamic pipeline with feed forward and feedback connections for two different functions

Reservation Tables The reservation table for a static linear pipeline is trivial in the sense that dataflow follows a linear streamline. The *reservation table* for a dynamic pipeline becomes more interesting because a nonlinear pattern is followed. Given a pipeline configuration, multiple reservation tables can be generated for the evaluation of different functions.

Two reservation tables are given in Figs. 6.3b and 6.3c, corresponding to a function X and a function Y, respectively. Each function evaluation is specified by one reservation table. A static pipeline is specified by a single reservation table. A dynamic pipeline may be specified by more than one reservation table.

Each reservation table displays the time-space flow of data through the pipeline for one function evaluation. Different functions follow different paths through the pipeline.

The number of columns in a reservation table is called the *evaluation time* of a given function. For example, the function X requires eight clock cycles to evaluate, and function Y requires six cycles, as shown in Figs. 6.3b and 6.3c, respectively.

A pipeline *initiation* table corresponds to each function evaluation. All initiations to a static pipeline use the same reservation table. On the other hand, a dynamic pipeline may allow different initiations to follow a mix of reservation tables. The checkmarks in each row of the reservation table correspond to the time instants (cycles) that a particular stage will be used.

There may be multiple checkmarks in a row, which means repeated usage of the same stage in different cycles. Contiguous checkmarks in a row simply imply the extended usage of a stage over more than one cycle. Multiple checkmarks in a column mean that multiple stages need to be used in parallel during a particular clock cycle.

Latency Analysis The number of time units (clock cycles) between two initiations of a pipeline is *the latency* between them. Latency values must be nonnegative integers. A latency of k means that two initiations are separated by k clock cycles. Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a *collision*.

A collision implies resource conflicts between two initiations in the pipeline. Therefore, all collisions must be avoided in scheduling a sequence of pipeline initiations. Some latencies will cause collisions, and some will not. Latencies that cause collisions are called *forbidden latencies*. In using the pipeline in Fig. 6.3 to evaluate the function X, latencies 2 and 5 are forbidden, as illustrated in Fig. 6.4.

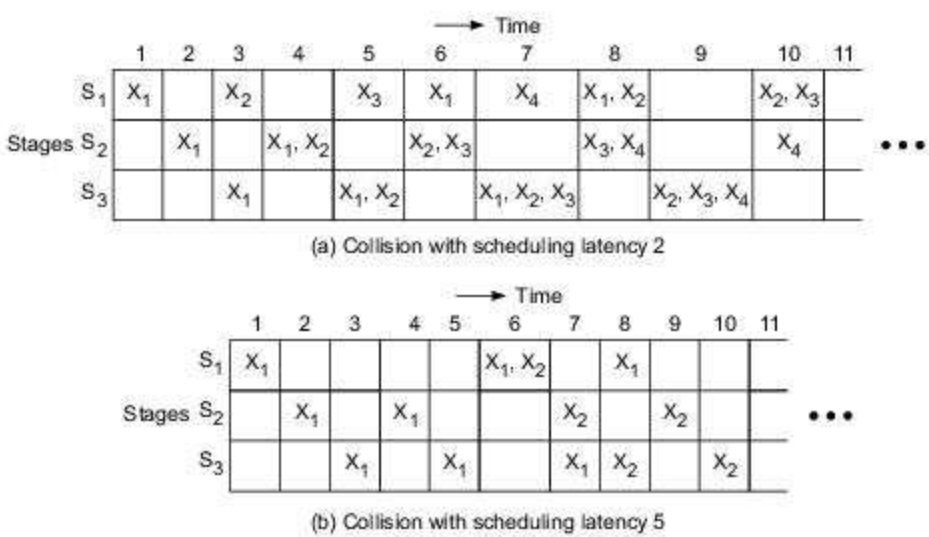


Fig. 6.4 Collisions with forbidden latencies 2 and 5 in using the pipeline in Fig. 6.3 to evaluate the function X

The ith initiation is denoted as X_i in Fig. 6.4. With latency 2, initiations X_1 and X_2 collide in stage 2 at time 4. At time 7, these initiations collide in stage 3. Similarly, other collisions are shown at times 5, 6, 8, ..., etc.

The collision patterns for latency 5 are shown in Fig. 6.4b, where X_1 and X_2 are scheduled 5 clock cycles apart. Their first collision occurs at time 6.

To detect a forbidden latency, one needs simply to check the distance between any two checkmarks in the same row of the reservation table. For example, the distance between the first mark and the second mark in row S_1 in Fig. 6.3b is 5, implying that 5 is a forbidden latency.

Similarly, latencies 2, 4, 5, and 7 are all seen to be forbidden from inspecting the same reservation table. From the reservation table in Fig. 6.3c, we discover the forbidden latencies 2 and 4 for function Y. A *latency sequence* is a sequence of permissible nonforbidden latencies between successive task initiations.

A *latency cycle* is a latency sequence which repeats the same subsequence (cycle) indefinitely. Figure 6.5 illustrates latency cycles in using the pipeline in Fig. 6.3 to evaluate the function X without causing a collision. For example, the latency cycle (1, 8) represents the infinite latency sequence 1, 8, 1, 8, 1, 8, ... This implies that successive initiations of new tasks are separated by one cycle and eight cycles alternately.

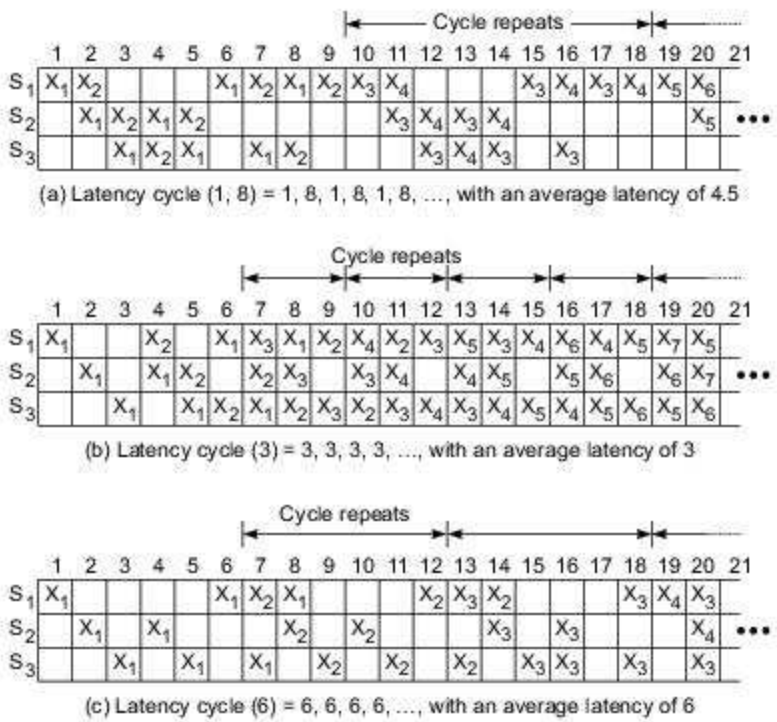


Fig. 6.5 Three valid latency cycles for the evaluation of function X

The *average latency* of a latency cycle is obtained by dividing the sum of all latencies by the number of latencies along the cycle. The latency cycle (1, 8) thus has an average latency of $(1 + 8)/2 = 4.5$. A *constant cycle* is a latency cycle which contains only one latency value. Cycles (3) and (6) in Figs. 6.5b and 6.5c are both constant cycles. The average latency of a constant cycle is simply the latency itself. In the next section, we describe how to obtain these latency cycles systematically.

6.2.2 Collision-Free Scheduling

When scheduling events in a nonlinear pipeline, the main objective is to obtain the shortest average latency between initiations without causing collisions. In what follows, we present a systematic method for achieving such collision-free scheduling.

We study below *collision vectors, state diagrams, single cycles, greedy cycles,* and *minimal average latency* (MAL). This pipeline design theory was originally developed by Davidson (1971) and his students.

Collision Vectors By examining the reservation table, one can distinguish the set of permissible latencies from the set of forbidden latencies. For a reservation table with n columns, the *maximum forbidden latency* $m \leq n-1$. The permissible latency p should be as small as possible. The choice is made in the range $1 \leq p \leq m-1$.

A permissible latency of $p = 1$ corresponds to the ideal case. In theory, a latency of 1 can always be achieved in a static pipeline which follows a linear (diagonal or streamlined) reservation table as shown in Fig. 6.1c.

The combined set of permissible and forbidden latencies can be easily displayed by a collision vector, which is an m-bit binary vector $C = (C_m C_{m-1} \ldots C_2 C_1)$. The value of $C_i = 1$ if latency i causes a collision and $C_i = 0$ if latency i is permissible. Note that it is always true that $C_m = 1$, corresponding to the maximum forbidden latency.

For the two reservation tables in Fig. 6.3, the collision vector $C_X = (1011010)$ is obtained for function X, and $C_Y = (1010)$ for function Y. From C_X, we can immediately tell that latencies 7, 5, 4, and 2 are forbidden and latencies 6, 3, and 1 are permissible. Similarly, 4 and 2 are forbidden latencies and 3 and 1 are permissible latencies for function Y.

State Diagrams From the above collision vector, one can construct a *state diagram* specifying the permissible state transitions among successive initiations. The collision vector, like C_X above, corresponds to the *initial state* of the pipeline at time 1 and thus is called an *initial collision vector*. Let p be a permissible latency within the range $1 \leq p \leq m-1$.

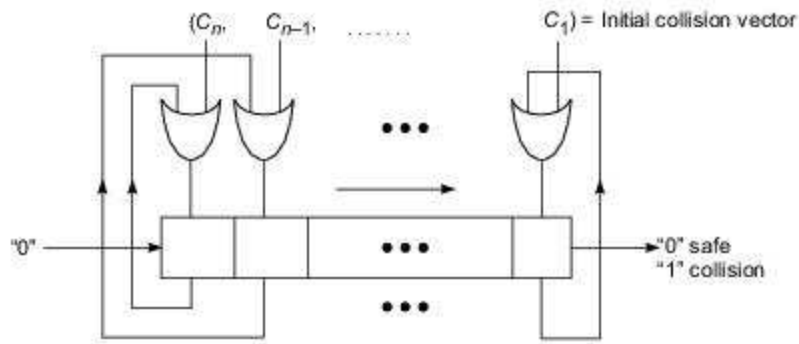
The *next state* of the pipeline at time $t + p$ is obtained with the assistance of an m-bit right shift register as in Fig. 6.6a. The initial collision vector C is initially loaded into the register. The register is then shifted to the right. Each 1-bit shift corresponds to an increase in the latency by 1. When a 0 bit emerges from the right end after p shifts, it means p is a permissible latency. Likewise, a 1 bit being shifted out means a collision, and thus the corresponding latency should be forbidden.

Logical 0 enters from the left end of the shift register. The next state after p shifts is thus obtained by bitwise-ORing the initial collision vector with the shifted register contents. For example, from the initial state $C_X = (1011010)$, the next state (1111111) is reached after one right shift of the register, and the next state (1011011) is reached after three shifts or six shifts.

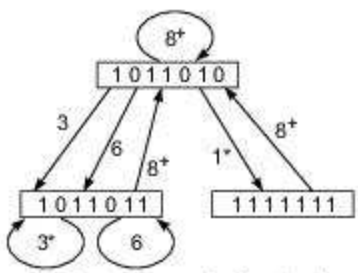
Example 6.2 The state transition diagram for a pipeline unit

A *state diagram* is obtained in Fig. 6.6b for function X. From the initial state (1011010), only three outgoing transitions are possible, corresponding to the three permissible latencies 6, 3, and 1 in the initial collision vector. Similarly, from state (1011011), one reaches the same state after either three shifts or six shifts.

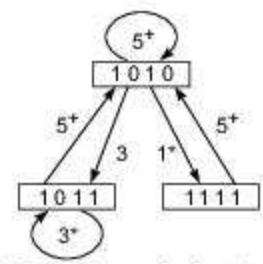
When the number of shifts is $m + 1$ or greater, all transitions are redirected back to the initial state. For example, after eight or more (denoted as 8^+) shifts, the next state must be the initial state, regardless of which state the transition starts from. In Fig. 6.6c, a state diagram is obtained for the reservation table in Fig. 6.3c using a 4-bit shift register. Once the initial collision vector is determined, the corresponding state diagram is uniquely determined.



(a) State transition using an n-bit right shift register, where n is the maximum forbidden latency



(b) State diagram for function X



(c) State diagram for function Y

Fig. 6.6 Two state diagrams obtained from the two reservation tables in Fig. 6.3, respectively

The 0's and 1's in the present state, say at time t, of a state diagram indicate the permissible and forbidden latencies, respectively, at time t. The bitwise ORing of the shifted version of the present state with the initial collision vector is meant to prevent collisions from future initiations starting at time $t + 1$ and onward.

Thus the state diagram covers all permissible state transitions that avoid collisions. All latencies equal to or greater than m are permissible. This implies that collisions can always be avoided if events are scheduled far apart (with latencies of m^+). However, such long latencies are not tolerable from the viewpoint of pipeline throughput.

Greedy Cycles From the state diagram, we can determine optimal latency cycles which result in the MAL. There are infinitely many latency cycles one can trace from the state diagram. For example, (1, 8), (1, 8, 6, 8), (3), (6), (3, 8), (3, 6, 3) ..., are legitimate cycles traced from the state diagram in Fig. 6.6b. Among these cycles, only *simple cycles* are of interest.

A simple cycle is a latency cycle in which each state appears only once. In the state diagram in Fig. 6.6b, only (3), (6), (8), (1, 8), (3, 8), and (6, 8) are simple cycles. The cycle (1, 8, 6, 8) is not simple because it travels through the state (1011010) twice. Similarly, the cycle (3, 6, 3, 8, 6) is not simple because it repeats the state (1011011) three times.

Some of the simple cycles are *greedy cycles*. A greedy cycle is one whose edges are all made with minimum latencies from their respective starting states. For example, in Fig. 6.6b the cycles (1, 8) and (3) are greedy cycles. Greedy cycles in Fig. 6.6c are (1, 5) and (3). Such cycles must first be simple, and their average latencies must be lower than those of other simple cycles. The greedy cycle (1, 8) in Fig. 6.6b has an average latency of $(1 + 8)/2 = 4.5$, which is lower than that of the simple cycle $(6, 8) = (6 + 8)/2 = 7$. The greedy cycle (3) has a constant latency which equals the MAL for evaluating function X without causing a collision.

The MAL in Fig. 6.6c is 3, corresponding to either of the two greedy cycles. The minimum-latency edges in the state diagrams are marked with asterisks.

At least one of the greedy cycles will lead to the MAL. The collision-free scheduling of pipeline events is thus reduced to finding greedy cycles from the set of simple cycles. The greedy cycle yielding the MAL is the final choice.

6.2.3 Pipeline Schedule Optimization

An optimization technique based on the MAL is given below. The idea is to insert noncompute delay stages into the original pipeline. This will modify the reservation table, resulting in a new collision vector and an improved state diagram. The purpose is to yield an optimal latency cycle, which is absolutely the shortest.

Bounds on the MAL In 1972, Shar determined the following bounds on the *minimal average latency* (MAL) achievable by any control strategy on a statically reconfigured pipeline executing a given reservation table:

(1) The MAL is lower-bounded by the maximum number of checkmarks in any row of the reservation table.
(2) The MAL is lower than or equal to the average latency of any greedy cycle in the state diagram.
(3) The average latency of any greedy cycle is upper-bounded by the number of 1's in the initial collision vector plus 1. This is also an upper bound on the MAL.

Interested readers may refer to Shar (1972) or find proofs of these bounds in Kogge (1981). These results suggest that the optimal latency cycle must be selected from one of the lowest greedy cycles. However, a greedy cycle is not sufficient to guarantee the optimality of the MAL. The lower bound guarantees the optimality. For example, the MAL = 3 for both function X and function Y and has met the lower bound of 3 from their respective reservation tables.

From Fig. 6.6b, the upper bound on the MAL for function X is equal to $4 + 1 = 5$, a rather loose bound. On the other hand, Fig. 6.6c shows a rather tight upper bound of $2 + 1 = 3$ on the MAL. Therefore, all greedy cycles for function Y lead to the optimal latency value of 3, which cannot be lowered further.

To optimize the MAL, one needs to find the lower bound by modifying the reservation table. The approach is to reduce the maximum number of checkmarks in any row. The modified reservation table must preserve the original function being evaluated. Patel and Davidson (1976) have suggested the use of noncompute delay stages to increase pipeline performance with a shorter MAL. Their technique is described below.

Delay Insertion The purpose of delay insertion is to modify the reservation table, yielding a new collision vector. This leads to a modified state diagram, which may produce greedy cycles meeting the lower bound on the MAL.

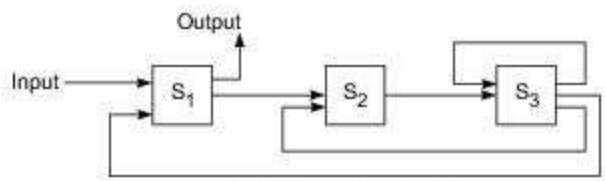
Before delay insertion, the three-stage pipeline in Fig. 6.7a is specified by the reservation table in Fig. 6.7b. This table leads to a collision vector $C = (1011)$, corresponding to forbidden latencies 1, 2, and 4. The corresponding state diagram (Fig. 6.7c) contains only one self-reflecting state with a greedy cycle of latency 3 equal to the MAL.

Based on the given reservation table, the maximum number of checkmarks in any row is 2. Therefore, the MAL = 3 so obtained in Fig. 6.7c is not optimal.

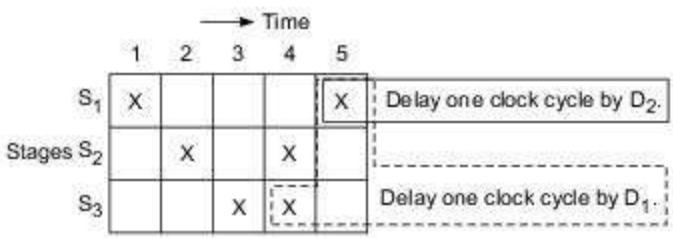
Example 6.3 Inserting noncompute delays to reduce the MAL

To insert a noncompute stage D_1 after stage S_3 will delay both X_1 and X_2 operations one cycle beyond time 4. To insert yet another noncompute stage D_2 after the second usage of S_1 will delay the operation X_2 by another cycle.

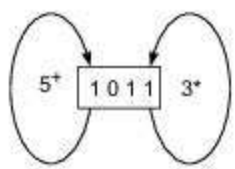
These delay operations, as grouped in Fig. 6.7b, result in a new pipeline configuration in Fig. 6.8a. Both delay elements D_1 and D_2 are inserted as extra stages, as shown in Fig. 6.8b with an enlarged reservation table having 3 + 2 = 5 rows and 5 + 2 = 7 columns.



(a) A three-stage pipeline



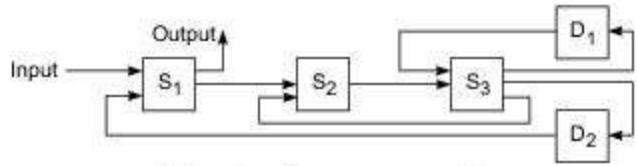
(b) Reservation table and operations being delayed



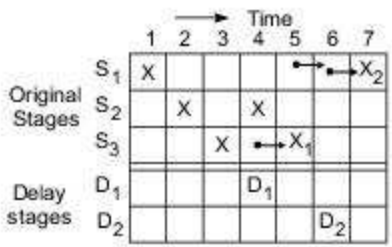
(c) State transition diagram with MAL = 3

Fig. 6.7 A pipline with a minimum average latency of 3

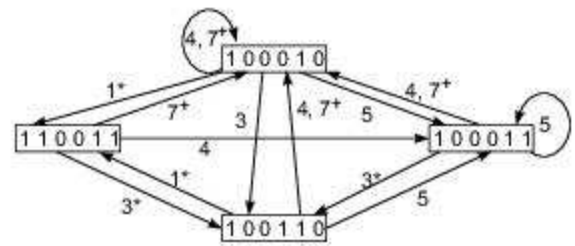
In total, the operation X_1 has been delayed one cycle from time 4 to time 5 and the operation X_2 has been delayed two cycles from time 5 to time 7. All remaining operations (marked as X in Fig. 6.8b) are unchanged. This new table leads to a new collision vector (100010) and a modified state diagram in Fig. 6.8c.



(a) Insertion of two noncompute delay stages



(b) Modified reservation table



(c) Modified state diagram with a reduced MAL = (1 + 3)/2 = 2

Fig. 6.8 Insertion of two delay stages to obtain an optimal MAL for the pipeline in Fig. 6.7

This diagram displays a greedy cycle (1, 3), resulting in a reduced MAL = (1 + 3)/2 = 2. The delay insertion thus improves the pipeline performance, yielding a lower bound for the MAL.

Pipeline Throughput This is essentially the initiation rate or the average number of task initiations per clock cycle. If N tasks are initiated within n pipeline cycles, then the *initiation rate* or *pipeline throughput* is measured as N/n. This rate is determined primarily by the inverse of the MAL adapted. Therefore, the scheduling strategy does affect the pipeline performance.

In general, the shorter the adapted MAL, the higher the throughput that can be expected. The highest achievable throughput is one task initiation per cycle, when the MAL equals 1 since 1 ≤ MAL ≤ the shortest latency of any greedy cycle. Unless the MAL is reduced to 1, the pipeline throughput becomes a fraction.

Pipeline Efficiency Another important measure is *pipeline efficiency*. The percentage of time that each pipeline stage is used over a sufficiently long series of task initiations is the *stage utilization*. The accumulated rate of all stage utilizations determines the pipeline efficiency.

Let us reexamine latency cycle (3) in Fig. 6.5b. Within each latency cycle of three clock cycles, there are two pipeline stages, S_1 and S_3, which are completely and continuously utilized after time 6. The pipeline stage S_2 is used for two cycles and is idle for one cycle.

Therefore, the entire pipeline can be considered 8/9 = 88.8% efficient for latency cycle (3). On the other hand, the pipeline is only 14/27 = 51.8% efficient for latency cycle (1, 8) and 8/16 = 50% efficient for latency cycle (6), as illustrated in Figs. 6.5a and 6.5c, respectively. Note that none of the three stages is fully utilized with respect to two initiation cycles.

The pipeline throughput and pipeline efficiency are related to each other. Higher throughput results from a shorter latency cycle. Higher efficiency implies less idle time for pipeline stages. The above example demonstrates that higher throughput also accompanies higher efficiency. Other examples however may show

a contrary conclusion. The relationship between the two measures is a function of the reservation table and of the initiation cycle adopted.

At least one stage of the pipeline should be fully (100%) utilized at the steady state in any acceptable initiation cycle; otherwise, the pipeline capability has not been fully explored. In such cases, the initiation cycle may not be optimal and another initiation cycle should be examined for improvement.

6.3 INSTRUCTION PIPELINE DESIGN

A stream of instructions can be executed by a pipeline in an overlapped manner. We describe below instruction pipelines for CISC and RISC scalar processors. Topics to be studied include instruction prefetching, data forwarding, hazard avoidance, interlocking for resolving data dependences, dynamic instruction scheduling, and branch handling techniques for improving pipelined processor performance. Further discussion on instruction level parallelism will be found in Chapter 12.

6.3.1 Instruction Execution Phases

A typical instruction execution consists of a sequence of operations, including instruction fetch, decode, operand fetch, execute, and write-back phases. These phases are ideal for overlapped execution on a linear pipeline.

Pipelined Instruction Processing A typical instruction pipeline is depicted in Fig. 6.9. The *fetch stage* (F) fetches instructions from a cache memory, ideally one per cycle. The *decode stage* (D) reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units. The *issue stage* (I) reserves resources. The operands are also read from registers during the issue stage.

The instructions are executed in one or several *execute stages* (E). Three execute stages are shown in Fig. 6.9a. The last *writeback stage* (W) is used to write results into the registers. Memory load or store operations are treated as part of execution. Figure 6.9 shows the flow of machine instructions through a typical pipeline. These eight instructions are for pipelined execution of the high-level language statements $X = Y + Z$ and $A = B \times C$. Here we have assumed that *load* and *store* instructions take four execution clock cycles, while floating-point *add* and *multiply* operations take three cycles.

The above timing assumptions represent typical values found in an older CISC processor. In many RISC processors, fewer clock cycles are needed. On the other hand, Cray 1 required 11 cycles for a load and a floating-point addition took six. With in-order instruction issuing, if an instruction is blocked from issuing due to a data or resource dependence, all instructions following it are blocked.

Figure 6.9b illustrates the issue of instructions following the original program order. The shaded boxes correspond to idle cycles when instruction issues are blocked due to resource latency or conflicts or due to data dependences. The first two *load* instructions issue on consecutive cycles. The *add* is dependent on both *loads* and must wait three cycles before the data (Y and Z) are loaded in.

Similarly, the *store* of the sum to memory location X must wait three cycles for the *add* to finish due to a flow dependence. There are similar blockages during the calculation of A. The total time required is 17 clock cycles. This time is measured beginning at cycle 4 when the first instruction starts execution until cycle 20

when the last instruction starts execution. This timing measure eliminates the undue effects of the pipeline "startup" or "draining" delays.

Figure 6.9c shows an improved timing after the instruction issuing order is changed to eliminate unnecessary delays due to dependence. The idea is to issue all four *load* operations in the beginning. Both the *add* and *multiply* instructions are blocked fewer cycles due to this data prefetching. The reordering should not change the end results. The time required is being reduced to 11 cycles, measured from cycle 4 to cycle 14.

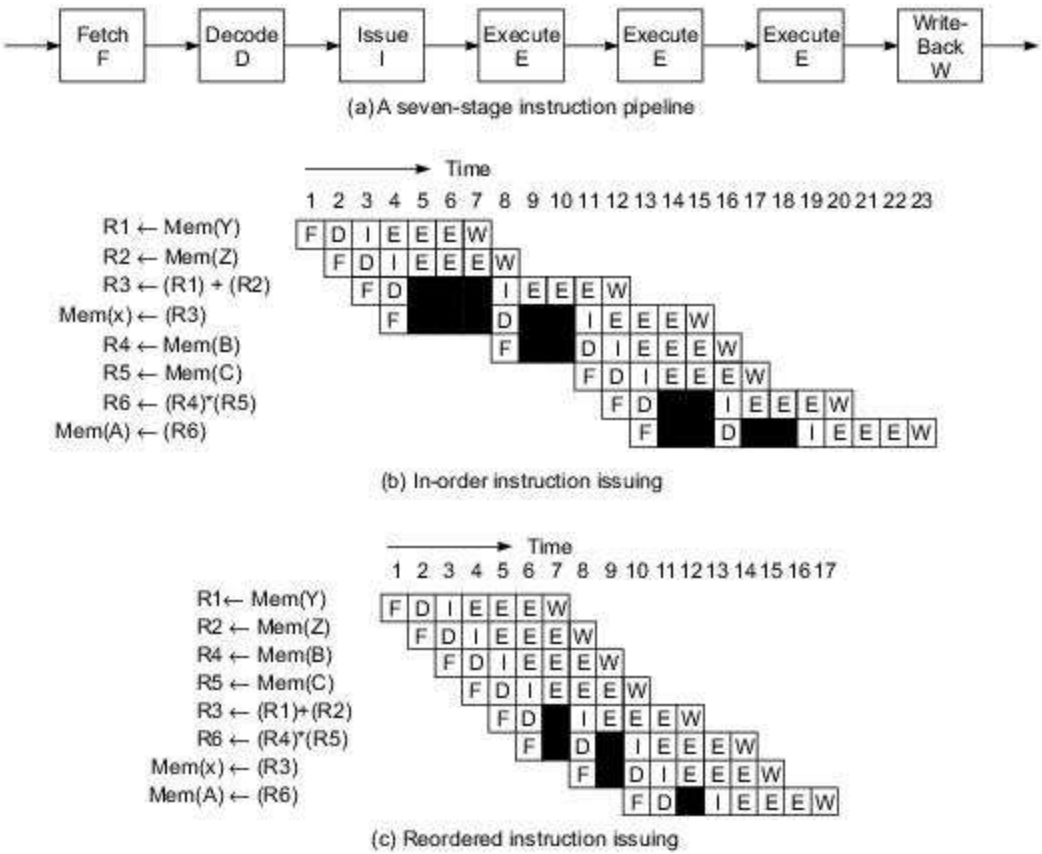


Fig. 6.9 Pipelined execution of X = Y + Z and A = B × C (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

Example 6.4 The MIPS R4000 instruction pipeline

The MIPS R4000 was a pipelined 64-bit processor using separate instruction and data caches and an eight-stage pipeline for executing register-based instructions. As illustrated in Fig. 6.10, the processor pipeline design was targeted to achieve an execution rate approaching one instruction per cycle.

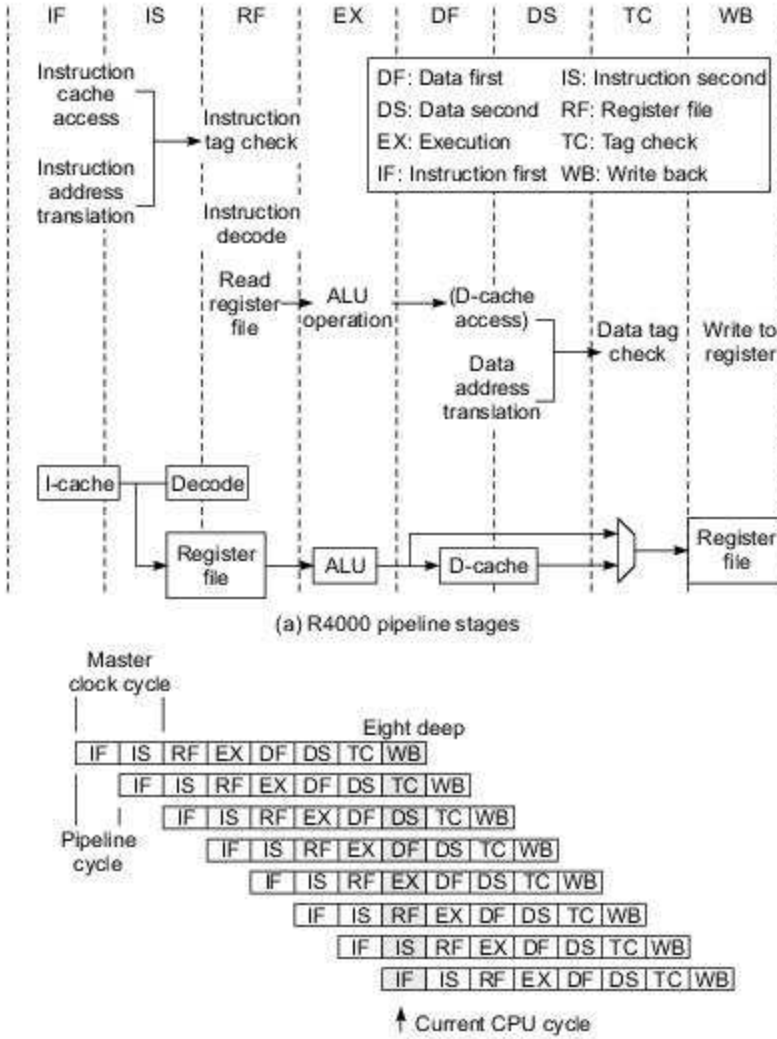


Fig. 6.10 The architecture of the MIPS R4000 instruction pipeline (Courtesy of MIPS Computer Systems)

The execution of each R4000 instruction consisted of eight major steps as summarized in Fig. 6.10a. Each of these steps required approximately one clock cycle. The instruction and data memory references are split across two stages. The single-cycle ALU stage took slightly more time than each of the cache access stages.

The overlapped execution of successive instructions is shown in Fig. 6.10b. This pipeline operated efficiently because different CPU resources, such as address and bus access, ALU operations, register accesses, and so on, were utilized simultaneously on a noninterfering basis.

The internal pipeline clock rate (100 MHz) of the R4000 was twice the external input or master clock

frequency. Figure 6.10b shows the optimal pipeline movement, completing one instruction every internal clock cycle. Load and branch instructions introduce extra delays.

6.3.2 Mechanisms for Instruction Pipelining

We introduce instruction buffers and describe the use of cacheing, collision avoidance, multiple functional units, register tagging, and internal forwarding to smooth pipeline flow and to remove bottlenecks and unnecessary memory access operations.

Prefetch Buffers Three types of buffers can be used to match the instruction fetch rate to the pipeline consumption rate. In one memory-access time, a block of consecutive instructions are fetched into a prefetch buffer as illustrated in Fig. 6.11. The block access can be achieved using interleaved memory modules or using a cache to shorten the effective memory-access time as demonstrated in the MIPS R4000.

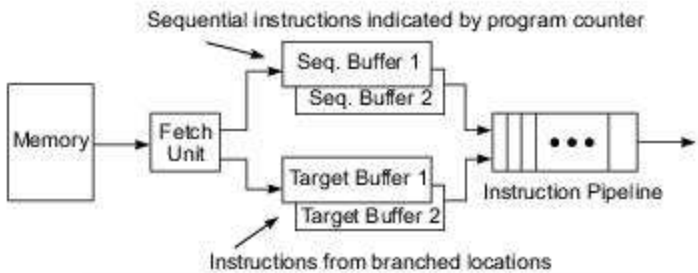


Fig. 6.11 The use of sequential and target buffers

Sequential instructions are loaded into a pair of *sequential buffers* for in-sequence pipelining. Instructions from a branch target are loaded into a pair of *target buffers* for out-of-sequence pipelining. Both buffers operate in a first-in-first-out fashion. These buffers become part of the pipeline as additional stages.

A conditional branch instruction causes both sequential buffers and target buffers to fill with instructions. After the branch condition is checked, appropriate instructions are taken from one of the two buffers, and instructions in the other buffer are discarded. Within each pair, one can use one buffer to load instructions from memory and use another buffer to feed instructions into the pipeline. The two buffers in each pair alternate to prevent a collision between instructions flowing into and out of the pipeline.

A third type of prefetch buffer is known as a *loop buffer*. This buffer holds sequential instructions contained in a small loop. The loop buffers are maintained by the fetch stage of the pipeline. Prefetched instructions in the loop body will be executed repeatedly until all iterations complete execution. The loop buffer operates in two steps. First, it contains instructions sequentially ahead of the current instruction. This saves the instruction fetch time from memory. Second, it recognizes when the target of a branch falls within the loop boundary. In this case, unnecessary memory accesses can be avoided if the target instruction is already in the loop buffer. The CDC 6600 and Cray 1 made use of loop buffers.

Multiple Functional Units Sometimes a certain pipeline stage becomes the bottleneck. This stage corresponds to the row with the maximum number of checkmarks in the reservation table. This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously. This leads to the use of multiple execution units in a pipelined processor design (Fig. 6.12).

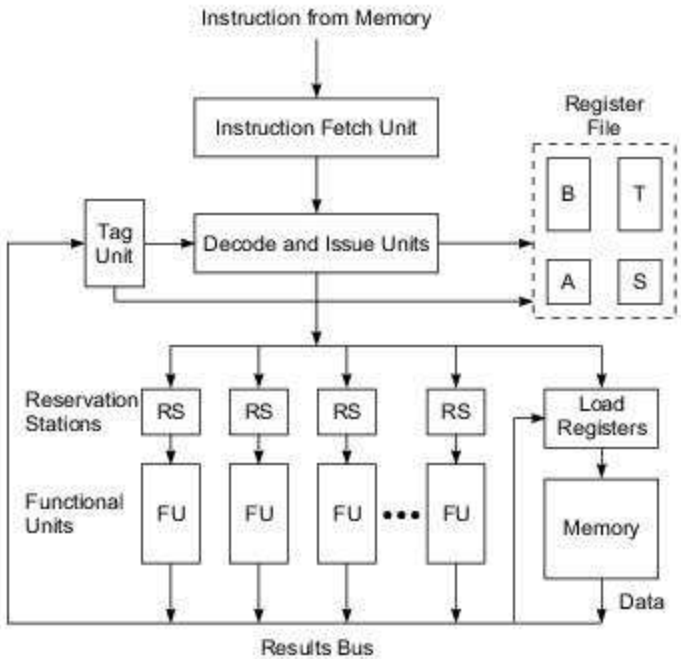


Fig. 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

Sohi (1990) used a model architecture for a pipelined scalar processor containing multiple functional units (Fig. 6.12). In order to resolve data or resource dependences among the successive instructions entering the pipeline, the *reservation stations* (RS) are used with each functional unit. Operations wait in the RS until their data dependences have been resolved. Each RS is uniquely identified by a *tag*, which is monitored by a *tag unit*.

The tag unit keeps checking the tags from all currently used registers or RSs. This register tagging technique allows the hardware to resolve conflicts between source and destination registers assigned for multiple instructions. Besides resolving conflicts, the RSs also serve as buffers to interface the pipelined functional units with the decode and issue units. The multiple functional units operate in parallel, once the dependences are resolved. This alleviates the bottleneck in the execution stages of the instruction pipeline.

Internal Data Forwarding The throughput of a pipelined processor can be further improved with internal data forwarding among multiple functional units. In some cases, some memory-access operations can be replaced by register transfer operations. The idea is described in Fig. 6.13.

A *store-load forwarding* is shown in Fig. 6.13a in which the *load operation* (LD R2, M) from memory to register R2 can be replaced by the *move* operation (MOVE R2, R1) from register R1 to register R2. Since register transfer is faster than memory access, this data forwarding will reduce memory traffic and thus results in a shorter execution time. Similarly, *load-load forwarding* (Fig. 6.13b) eliminates the second

load operation (LD R2, M) and replaces it with the *move* operation (MOVE R2, R1). Further discussion on operand forwarding will be continued in Chapter 12.

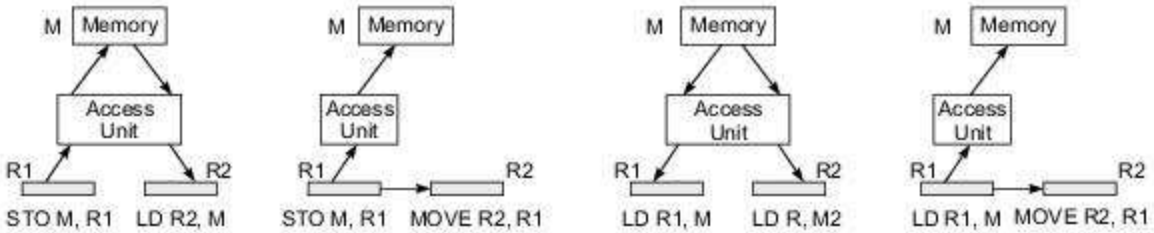


Fig. 6.13 Internal data forwarding by replacing memory-access operations with register transfer operations

Example 6.5 Implementing the dot-product operation with internal data forwarding between a multiply unit and an add unit

One can feed the output of a multiplier directly to the input of an adder (Fig. 6.14) for implementing the following dot-product operation:

$$s = \sum_{i=1}^{n} a_i \times b_i \tag{6.10}$$

Without internal data forwarding between the two functional units, the three instructions must be sequentially executed in a looping structure (Fig. 6.14a). With data forwarding, the output of the multiplier is fed directly into the input register R4 of the adder (Fig. 6.14b). At the same time, the output of the multiplier is also routed to register R3. Internal data forwarding between the two functional units thus reduces the total execution time through the pipelined processor.

Hazard Avoidance The *read* and *write* of shared variables by different instructions in a pipeline may lead to different results if these instructions are executed out of order. As illustrated in Fig. 6.15, three types of logic *hazards* are possible.

Consider two instructions I and J. Instruction J is assumed to logically follow instruction I according to program order. If the actual execution order of these two instructions violates the program order, incorrect results may be read or written, thereby producing hazards.

Hazards should be prevented before these instructions enter the pipeline, such as by holding instruction J until the dependence on instruction I is resolved. We use the notation D(I) and R(I) for the *domain* and *range* of an instruction I.

The domain contains the *input set* (such as operands in registers or in memory) to be used by instruction I. The range corresponds to the *output set* of instruction I. Listed below are conditions under which possible hazards can occur:

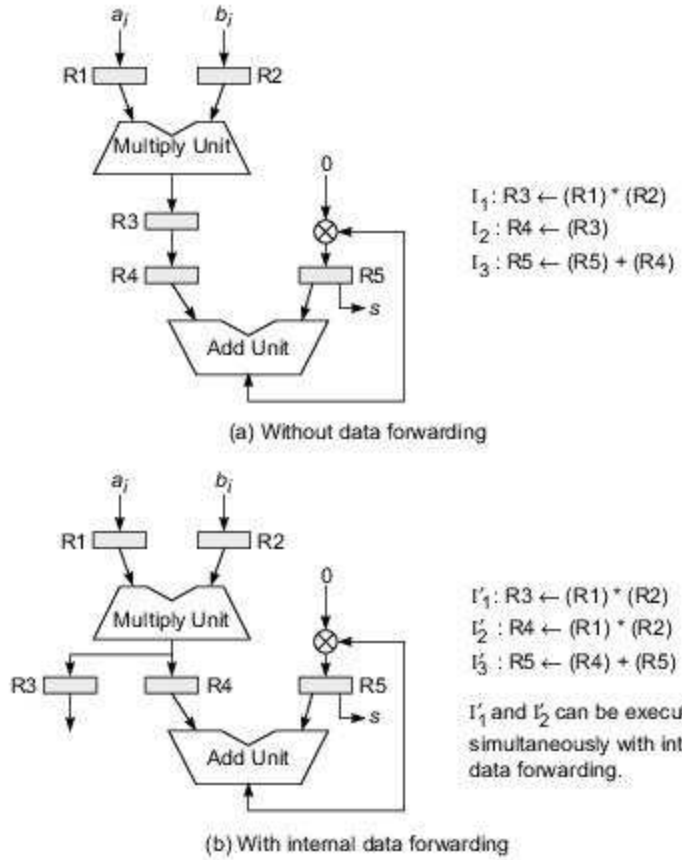


Fig. 6.14 Internal data forwarding for implementing the dot-product operation

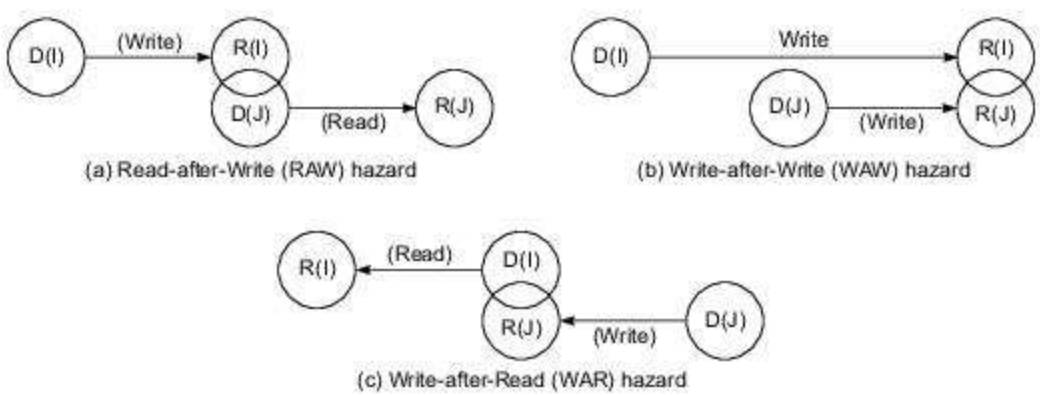


Fig. 6.15 Possible hazards between read and write operations in an instruction pipeline (instruction I is ahead of instruction J in program order)

$$R(I) \cap D(J) \neq \phi \text{ for RAW hazard}$$
$$R(I) \cap R(J) \neq \phi \text{ for WAW hazard} \quad (6.11)$$
$$D(I) \cap R(J) \neq \phi \text{ for WAR hazard}$$

These conditions are necessary but not sufficient. This means the hazard may not appear even if one or more of the conditions exist. The RAW hazard corresponds to the flow dependence, WAR to the antidependence, and WAW to the output dependence introduced in Section 2.1. The occurrence of a logic hazard depends on the order in which the two instructions are executed. Chapter 12 discusses techniques to handle such hazards.

6.3.3 Dynamic Instruction Scheduling

In this section, we describe three methods for scheduling instructions through an instruction pipeline. The *static scheduling* scheme is supported by an optimizing compiler. *Dynamic scheduling* is achieved using a technique such as Tomasulo's *register-tagging* scheme built in the IBM 360/91, or the *scoreboarding* scheme built in the CDC 6600 processor.

Static Scheduling Data dependences in a sequence of instructions create interlocked relationships among them. Interlocking can be resolved through a compiler-based static scheduling approach. A compiler or a postprocessor can be used to increase the separation between interlocked instructions.

Consider the execution of the following code fragment. The *multiply* instruction cannot be initiated until the preceding *load* is complete. This data dependence will stall the pipeline for three clock cycles since the two *loads* overlap by one cycle.

Stage delay:	Instruction:		
2 cycles	Add	R0, R1	/R0 ← (R0) + (R1)/
1 cycle	Move	R1, R5	/R1 ← (R5)/
2 cycles	Load	R2, M(α)	/R2 ← (Memory (α))/
2 cycles	Load	R3, M(β)	/R3 ← (Memory (β))/
3 cycles	Multiply	R2, R3	/R2 ← (R2) × (R3)/

The two *loads*, since they are independent of the *add* and *move*, can be moved ahead to increase the spacing between them and the *multiply* instruction. The following program is obtained after this modification:

Load	R2, M(α)	2 to 3 cycles
Load	R3, M (β)	2 cycles due to overlapping
Add	R0, R1	2 cycles
Move	R1, R5	1 cycle
Multiply	R2, R3	3 cycles

Through this code rearrangement, the data dependences and program semantics are preserved, and the *multiply* can be initiated without delay. While the operands are being loaded from memory cells α and β into registers R2 and R3, the two instructions *add* and *move* consume three cycles and thus pipeline stalling is avoided.

Tomasulo's Algorithm This hardware dependence-resolution scheme was first implemented with multiple floating-point units of the IBM 360/91 processor. The hardware platform is abstracted in Fig. 6.12. For the Model 91 processor, three RSs were used in a floating-point adder and two pairs in a floating-point multiplier. The scheme resolved resource conflicts as well as data dependences using register tagging to allocate or deallocate the source and destination registers.

An issued instruction whose operands are not available is forwarded to an RS associated with the functional unit it will use. It waits until its data dependences have been resolved and its operands become available. The dependence is resolved by monitoring the result bus (called *common data bus* in Model 91). When all operands for an instruction are available, it is dispatched to the functional unit for execution. All working registers are tagged. If a source register is busy when an instruction reaches the issue stage, the tag for the source register is forwarded to an RS. When the register data becomes available, it also reaches the RS which has the same tag.

Example 6.6 Tomasulo's algorithm for dynamic instruction scheduling

Tomasulo's algorithm was applied to work with processors having a few floating-point registers. In the case of Model 91, only four registers were available. Figure 6.16a shows a minimum-register machine code for computing X = Y + Z and A = B × C. The pipeline timing with Tomasulo's algorithm appears in Fig. 6.16b. Here, the total execution time is 13 cycles, counting from cycle 4 to cycle 15 by ignoring the pipeline startup and draining times.

Memory is treated as a special functional unit. When an instruction has completed execution, the result (along with its tag) appears on the result bus. The registers as well as the RSs monitor the result bus and update their contents (and ready/busy bits) when a matching tag is found. Details of the algorithm can be found in the original paper by Tomasulo (1967).

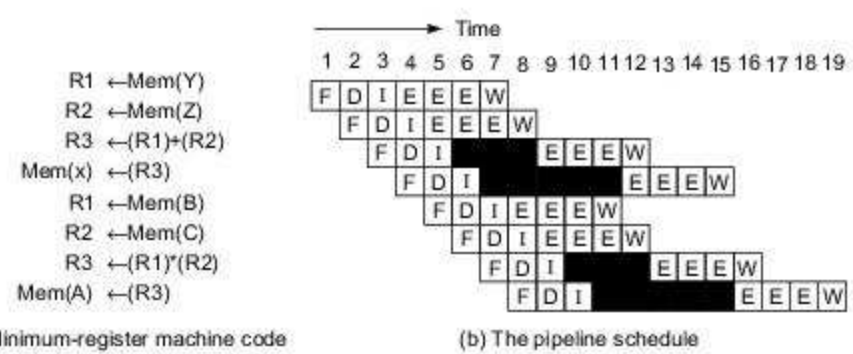
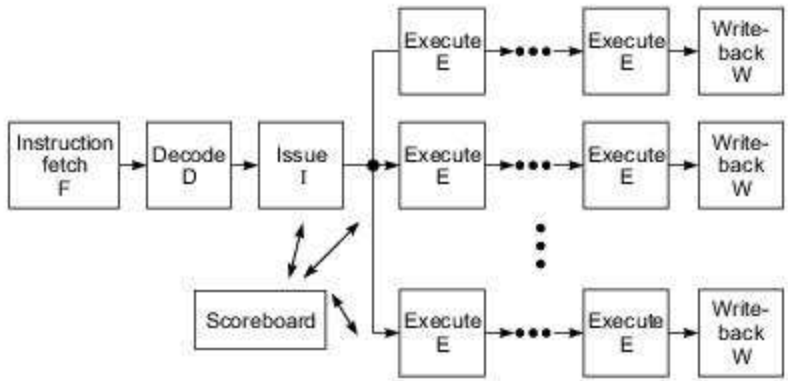


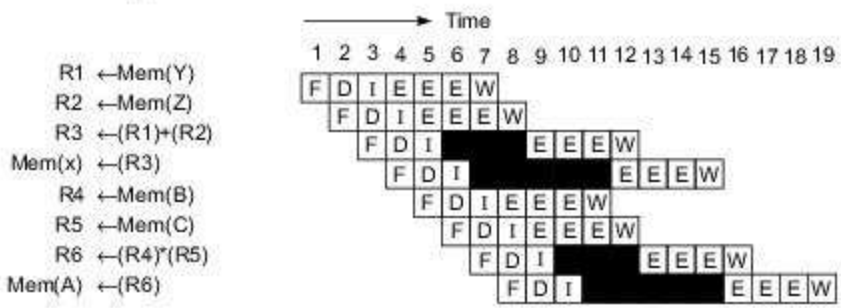
Fig. 6.16 Dynamic instruction scheduling using Tomasulo's algorithm on the processor in Fig. 6.12 (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

CDC Scoreboarding The CDC 6600 was an early high-performance computer that used dynamic instruction scheduling hardware. Figure 6.17a shows a CDC 6600-like processor, in which multiple functional units

appeared as multiple execution pipelines. Parallel units allowed instructions to complete out of the original program order. The processor had instruction buffers for each execution unit. Instructions were issued to available functional units regardless of whether register input data was available.



(a) A CDC 6600-like processor



(b) The improved schedule from Fig. 6.9b

Fig. 6.17 Hardware scoreboarding for dynamic instruction scheduling (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

The instruction would then wait in a buffer for its data to be produced by other instructions. To control the correct routing of data between execution units and registers, the CDC 6600 used a centralized control unit known as the *scoreboard*. This unit kept track of the registers needed by instructions waiting for the various functional units. When all registers had valid data, the scoreboard enabled the instruction execution. Similarly, when a functional unit finished, it signaled the scoreboard to release the resources.

Example 6.7 Pipelined operations using hardware scoreboarding on the CDC 6600-like processor (James Smith, 1989)

Figure 6.17b shows the pipeline schedule based on scoreboard issue logic. The schedule corresponds to the

execution of the same machine code for X = Y + Z and A = B × C. The pipeline latencies are the same as those resulting from Tomasulo's algorithm. The *add* instruction is issued to its functional unit before its registers are ready. It then waits for its input register operands.

The scoreboard routes the register values to the adder unit when they become available. In the meantime, the issue stage is not blocked, so other instructions can bypass the blocked add. It takes 13 clock cycles to perform the operations. Details of the CDC scoreboarding can be found in the book by Thornton (1970).

The scoreboard is a centralized control logic which keeps track of the status of registers and multiple functional units. When functional units generate new results, some data dependences can be resolved and thus a higher degree of parallelism can be explored with scoreboarding. Scoreboarding in latter microprocessors like MC88000 used forwarding logic and register tagging. In a way, scoreboarding implements a kind of *data-driven* mechanism to achieve efficient computations.

Dynamic instruction scheduling was implemented only in high-end mainframes or supercomputers in the past. Most microprocessors used static scheduling. But the trend has changed over the last two decades. RISC and superscalar processors are today built with hardware support of dynamic scheduling at runtime. Significant trace-driven data are needed to optimize the pipelined processor design. Toward this goal, processor and compiler designers have to work together to achieve an efficient design. Multiple-issue instruction pipelines, which are much more complicated than single-issue instruction pipelines, will be studied in Section 6.5.

6.3.4 Branch Handling Techniques

The performance of pipelined processors is limited by data dependences and branch instructions. In previous sections, we have studied the effects of data dependence. In this subsection, we study the effects of branching. Various branching strategies are reviewed. The evaluation of branching strategies can be performed either on specific pipeline architecture using trace data, or by applying analytic models. We provide below a simple performance analysis. For a more detailed treatment of the subject, readers are referred to the book *Branch Strategy Taxonomy and Performance Models* by Harvey Cragon (1992).

Effect of Branching Three basic terms are introduced below for the analysis of branching effects: The action of fetching a nonsequential or remote instruction after a branch instruction is called *branch taken*. The instruction to be executed after a branch taken is called a *branch target*. The number of pipeline cycles wasted between a branch taken and the fetching of its branch target is called the *delay slot*, denoted by b. In general, $0 \le b \le k-1$, where k is the number of pipeline stages.

When a branch is taken, all the instructions following the branch in the pipeline become useless and will be drained from the pipeline. This implies that a branch taken causes the pipeline to be flushed, losing a number of useful cycles.

These terms are illustrated in Fig. 6.18, where a branch taken causes I_{b+1} through I_{b+k-1} to be drained from the pipeline. Let p be the probability of a conditional branch instruction in a typical instruction stream and q the probability of a successfully executed conditional branch instruction (a branch taken). Typical values of $p = 20\%$ and $q = 60\%$ have been observed in some programs.

The penalty paid by branching is equal to $pqnb\tau$ because each branch taken costs $b\tau$ extra pipeline cycles. Based on Eq. 6.4, we thus obtain the total execution time of n instructions, including the effect of branching, as follows:

$$T_{eff} = k\tau + (n-1)\,\tau + pqnb\tau$$

Modifying Eq. 6.9, we define the following *effective pipeline throughput* with the influence of branching:

$$H_{eff} = \frac{n}{T_{eff}} = \frac{nf}{k + n - 1 + pqnb} \tag{6.12}$$

When $n \to \infty$, the tightest upper bound on the effective pipeline throughput is obtained when $b = k - 1$:

$$H^*_{eff} = \frac{f}{pq(k-1)+1} \tag{6.13}$$

When $p = q = 0$ (no branching), the above bound approaches the maximum throughput $f = 1/\tau$, same as in Eq, 6.2. Suppose $p = 0.2$, $q = 0.6$, and $b = k - 1 = 7$. We define the following *performance degradation factor*:

$$D = \frac{f - H^*_{eff}}{f} = 1 - \frac{1}{pq(k-1)+1} = \frac{pq(k-1)}{pq(k-1)+1} = \frac{0.84}{1.84} = 0.46 \tag{6.14}$$

The above analysis implies that the pipeline performance can be degraded by 46% with branching when the instruction stream is sufficiently long. This analysis demonstrates the high degree of performance degradation caused by branching in an instruction pipeline.

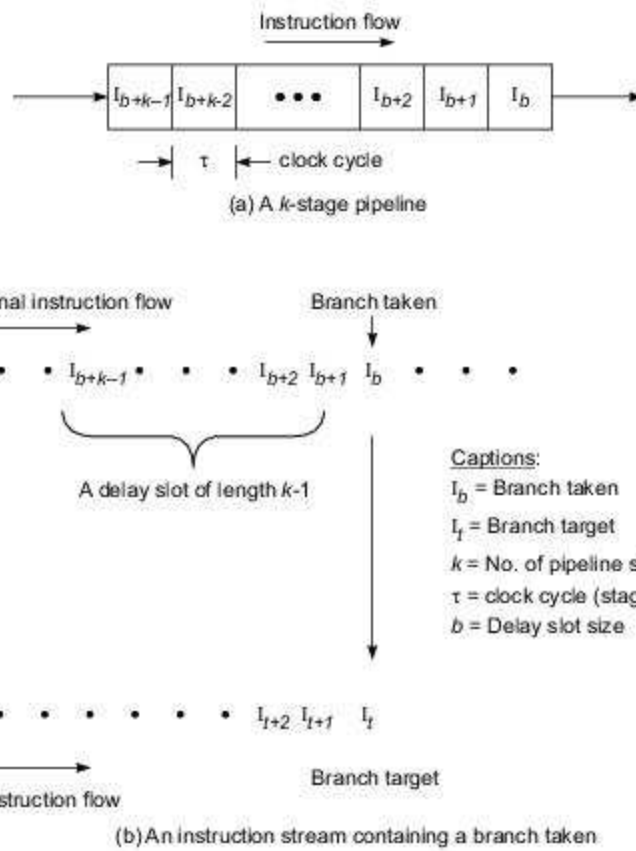


Fig. 6.18 The decision of a branch taken at the last stage of an instruction pipeline causes $b \leq k - 1$ previously loaded instructions to be drained from the pipeline

Branch Prediction Branch can be predicted either based on branch code types statically or based on branch history during program execution. The probability of branch with respect to a particular branch instruction type can be used to predict branch. This requires collecting the frequency and probabilities of branch taken and branch types across a large number of program traces. Such a *static branch strategy* may not be very accurate.

The static prediction direction (*taken* or *not taken*) can even be wired into the processor. According to past experience, the best performance is given by predicting *taken*. This results from the fact that most conditional branch instructions are taken in program execution. The wired-in static prediction cannot be changed once committed to the hardware. However, the scheme can be modified to allow the compiler to select the direction of each branch on a *semi-static* prediction basis.

A *dynamic branch strategy* works better because it uses recent branch history to predict whether or not the branch will be taken next time when it occurs. To be accurate, one may need to use the entire history of the branch to predict the future choice. This is infeasible to implement. Therefore, most dynamic prediction is determined with limited recent history, as illustrated in Fig. 6.19.

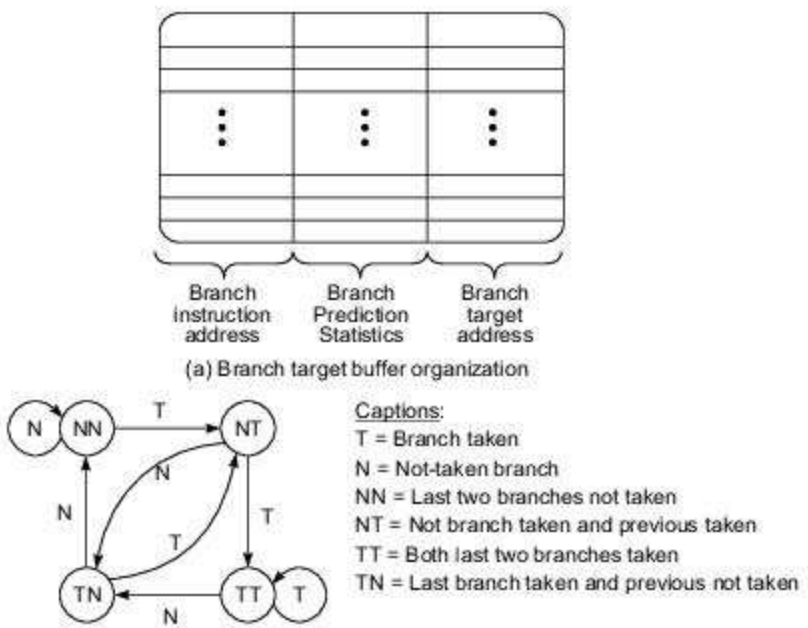


Fig. 6.19 Branch history buffer and a state transition diagram used in dynamic branch prediction (Courtesy of Lee and Smith, *IEEE Computer*, 1984)

Cragon (1992) classified dynamic branch strategies into three major classes: One class predicts the branch direction based upon information found at the decode stage. The second class uses a cache to store target addresses at the stage the effective address of the branch target is computed. The third scheme uses a cache to store target instructions at the fetch stage.

Dynamic prediction demands additional hardware to keep track of the past behavior of the branch instructions at run time. The amount of history recorded should be relatively small. Otherwise, the prediction logic becomes too costly to implement.

Lee and Smith (1984) suggested the use of a *branch target buffer* (BTB) to implement branch prediction (Fig. 6.19a). The BTB is used to hold recent branch information including the address of the branch target used. The address of the branch instruction locates its entry in the BTB.

A state transition diagram (Fig. 6.19b) has been used by Lee and Smith for tracking the last two outcomes at each branch instruction in a given program. The BTB entry contains the information which will guide the prediction. Prediction information is updated upon completion of the current branch.

The BTB can be extended to store not only the branch target address but also the target instruction itself, in order to allow zero delay in converting conditional branches to unconditional branches. The *taken* (T) and *not-taken* (N) labels in the state diagram correspond to actual program behavior. Further discussion on this topic will be found in Chapter 12.

Delayed Branches Examining the branch penalty, we realize that the branch penalty would be reduced significantly if the delay slot could be shortened or minimized to a zero penalty. The purpose of delayed branches is to make this possible, as illustrated in Fig. 6.20.

The idea was originally used to reduce the branching penalty in coding microinstructions. A *delayed branch* of d cycles allows at most $d - 1$ useful instructions to be executed following the branch taken. The execution of these instructions should be independent of the outcome of the branch instruction. Otherwise, a zero branching penalty cannot be achieved.

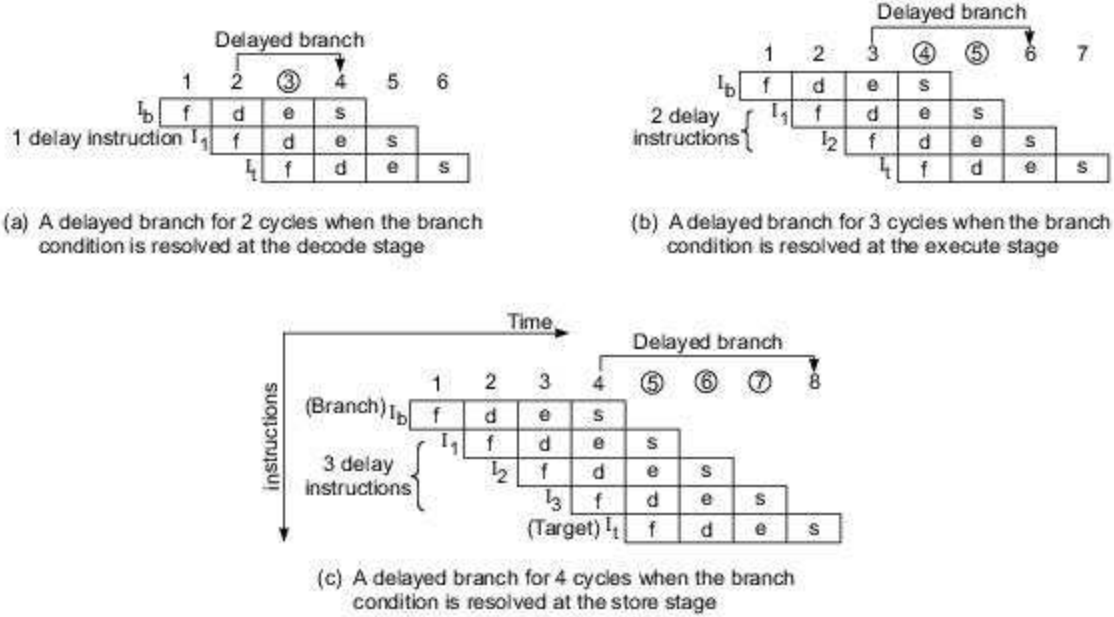
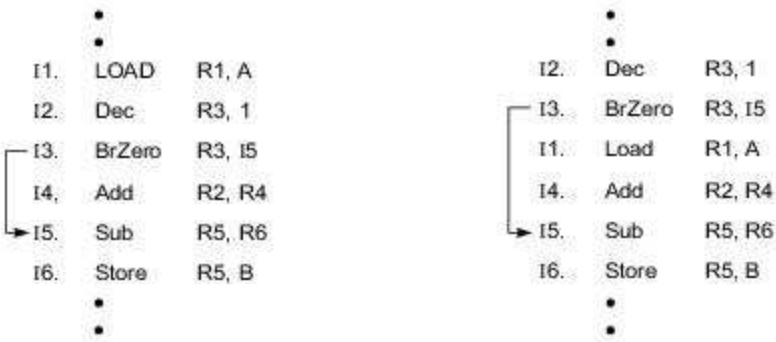


Fig. 6.20 The concept of delayed branch by moving independent instructions or NOP fillers into the delay slot of a four-stage pipeline

The technique is similar to that used for software interlocking. NOPs can be used as fillers if needed. The probability of moving one instruction ($d = 2$ in Fig. 6.20a) into the delay slot is greater than 0.6, that of moving two instructions ($d = 3$ in Fig. 6.20b) is about 0.2, and that of moving three instructions ($d = 4$ in Fig. 6.20c) is less than 0.1, according to some program trace results.

Example 6.8 A delayed branch with code motion into a delay slot

Code motion across branches can be used to achieve a delayed branch, as illustrated in Fig. 6.21. Consider the execution of a code fragment in Fig. 6.21a. The original program is modified by moving the useful instruction I1 into the delay slot after the branch instruction I3.



(a) Original program (b) Moving useful instructions into the delay slot

Fig. 6.21 Code motion across a branch to achieve a delayed branch with a reduced penalty to pipeline performance

In case the branch is not taken, the execution of the modified program produces the same results as the original program. In case the branch is taken in the modified program, execution of the delayed instructions I1 and I5 is needed anyway.

In general, data dependence between instructions moving across the branch and the remaining instructions being scheduled must be analyzed. Since instruction I1 is independent of the remaining instructions, leaving it in the delay slot will not create logic hazards or data dependences.

Sometimes NOP fillers can be inserted in the delay slot if no useful instructions can be found. However, inserting NOP fillers does not save any cycles in the delayed branch operation. From the above analysis, one can conclude that delayed branching may be more effective in short instruction pipelines with about four stages. Delayed branching has been built into some RISC processors, including the MIPS R4000 and Motorola MC88110.

6.4 ARITHMETIC PIPELINE DESIGN

Pipelining techniques can be applied to speed up numerical arithmetic computations. We start with a review of arithmetic principles and standards. Then we consider arithmetic pipelines with fixed functions.

A fixed-point multiply pipeline design and the MC68040 floating-point unit are used as examples to illustrate the design techniques involved. A multifunction arithmetic pipeline is studied with the TI-ASC arithmetic processor as an example.

6.4.1 Computer Arithmetic Principles

In a digital computer, arithmetic is performed with *finite precision* due to the use of fixed-size memory words or registers. Fixed-point or integer arithmetic offers a fixed range of numbers that can be operated upon. Floating-point arithmetic operates over a much increased dynamic range of numbers.

In modern processors, fixed-point and floating-point arithmetic operations are very often performed by separate hardware on the same processor chip.

Finite precision implies that numbers exceeding the limit must be truncated or rounded to provide a precision within the number of significant bits allowed. In the case of floating-point numbers, exceeding the exponent range means error conditions, called *overflow* or *underflow*. The Institute of Electrical and Electronics Engineers (IEEE) has developed standard formats for 32- and 64-bit floating numbers known as the *IEEE 754 Standard*. This standard has been adopted for most of today's computers.

Fixed-Point Operations Fixed-point numbers are represented internally in machines in *sign-magnitude*, *one's complement*, or *two's complement* notation. Most computers use the two's complement notation because of its unique representation of all numbers (including zero). One's complement notation introduces a second zero representation called *dirty zero*.

Add, *subtract*, *multiply*, and *divide* are the four primitive arithmetic operations. For fixed-point numbers, the add or subtract of two n-bit integers (or fractions) produces an n-bit result with at most one carry-out.

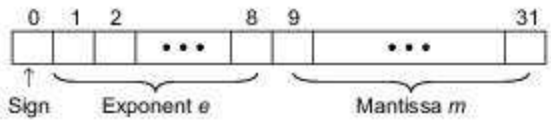
The multiplication of two n-bit numbers produces a $2n$-bit result which requires the use of two memory words or two registers to hold the full-precision result.

The division of an n-bit number by another may create an arbitrarily long quotient and a remainder. Only an approximate result is expected in fixed-point division with rounding or truncation. However, one can expand the precision by using a $2n$-bit dividend and an n-bit divisor to yield an n-bit quotient.

Floating-Point Numbers A floating-point number X is represented by a pair (m, e), where m is the *mantissa* (or *fraction*) and e is the *exponent* with an implied *base* (or *radix*). The algebraic value is represented as $X = m \times r^e$. The sign of X can be embedded in the mantissa.

Example 6.9 The IEEE 754 floating-point standard

A 32-bit floating-point number is specified in the IEEE 754 Standard as follows:



A binary base is assumed with $r = 2$. The 8-bit exponent e field uses an *excess-127* code. The dynamic range of e is (–127, 128), internally represented as (0, 255). The sign s and the 23-bit mantissa field m form a 25-bit sign-magnitude fraction, including an implicit or "hidden" 1 bit to the left of the binary point. Thus the complete mantissa actually represents the value $1.m$ in binary.

This hidden bit is not stored with the number. If $0 < e < 255$, then a nonzero normalized number represents the following algebraic value:

$$X = (-1)^s \times 2^{e-127} \times (1.m) \tag{6.15}$$

When $e = 255$ and $m \neq 0$, a *not-a-number* (NaN) is represented. NaNs can be caused by dividing a zero by a zero or taking the square root of a negative number, among many other nondeterminate cases. When $e = 255$ and $m = 0$, an infinite number $X = (-1)^s \infty$ is represented. Note that $+\infty$ and $-\infty$ are represented differently.

When $e = 0$ and $m \neq 0$, the number represented is $X = (-1)^s 2^{-126}(0.m)$. When $e = 0$ and $m = 0$, a zero is represented as $X = (-1)^s 0$. Again, +0 and – 0 are possible.

The 64-bit (double-precision) floating point can be defined similarly using an excess-l023 code in the exponent field and a 52-bit mantissa field. A number which is nonzero, finite, non-NaN, and normalized, has the following value:

$$X = (-1)^s \times 2^{e-1023} \times (1.m) \tag{6.16}$$

Special rules are given in the standard to handle overflow or underflow conditions. Interested readers may check the published IEEE standards for details.

Floating-Point Operations The four primitive arithmetic operations are defined below for a pair of floating-point numbers represented by $X = (m_x, e_x)$ and $Y = (m_y, e_y)$. For clarity, we assume $e_x \leq e_y$ and base $r = 2$.

$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y} \tag{6.17}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y} \tag{6.18}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y} \tag{6.19}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y} \tag{6.20}$$

The above equations clearly identify the number of arithmetic operations involved in each floating-point function. These operations can be divided into two halves: One half is for exponent operations such as comparing their relative magnitudes or adding/subtracting them; the other half is for mantissa operations, including four types of fixed-point operations.

Floating-point units are ideal for pipelined implementation. The two halves of the operations demand almost twice as much hardware as that required in a fixed-point unit. Arithmetic shifting operations are needed for equalizing the two exponents before their mantissas can be added or subtracted.

Shifting a binary fraction m to the right k places corresponds to the weighting $m \times 2^{-k}$, and shifting k places to the left corresponds to $m \times 2^k$. In addition, normalization of a floating-point number also requires left shifts to be performed.

Elementary Functions Elementary functions include trigonometric, exponential, logarithmic, and other transcendental functions. Truncated polynomials or power series can be used to evaluate the elementary functions, such as $\sin x$, $\ln x$, e^x, $\cosh x$, $\tan^{-1} y$, $\sqrt{x}$, x^3, etc. Interested readers may refer to the book by Hwang (1979) for details of computer arithmetic functions and their hardware implementation.

It should be noted that computer arithmetic can be implemented by hardwired logic circuitry as well as by table lookup using fast memory. Frequently used constants and special function values can also be generated by table lookup.

6.4.2 Static Arithmetic Pipelines

Most of today's arithmetic pipelines are designed to perform fixed functions. These *arithmetic/logic units* (ALUs) perform fixed-point and floating-point operations separately. The fixed-point unit is also called the integer unit. The floating-point unit can be built either as part of the central processor or on a separate coprocessor.

These arithmetic units perform scalar operations involving one pair of operands at a time. The pipelining in scalar arithmetic pipelines is controlled by software loops. Vector arithmetic units can be designed with pipeline hardware directly under firmware or hardwired control.

Scalar and vector arithmetic pipelines differ mainly in the areas of register files and control mechanisms involved. Vector hardware pipelines are often built as add-on options to a scalar processor or as an attached processor driven by a control processor. Both scalar and vector processors are used in modern supercomputers.

Arithmetic Pipeline Stages Depending on the function to be implemented, different pipeline stages in an arithmetic unit require different hardware logic. Since all arithmetic operations (such as *add, subtract, multiply, divide, squaring, square rooting, logarithm,* etc.) can be implemented with the basic add and shifting operations, the core arithmetic stages require some form of hardware to add and to shift.

For example, a typical three-stage floating-point adder includes a first stage for exponent comparison and equalization which is implemented with an integer adder and some shifting logic; a second stage for fraction addition using a high-speed carry lookahead adder; and a third stage for fraction normalization and exponent readjustment using a shifter and another addition logic.

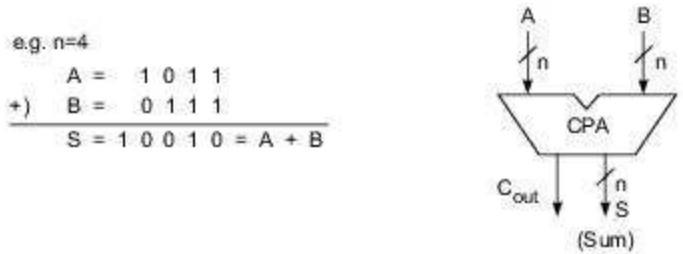
Arithmetic or logical shifts can be easily implemented with *shift registers.* High-speed addition requires either the use of a *carry-propagation adder* (CPA) which adds two numbers and produces an arithmetic sum as shown in Fig. 6.22a, or the use of a *carry-save adder* (CSA) to "add" three input numbers and produce one sum output and a carry output as exemplified in Fig. 6.22b.

In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using either ripple carry propagation or some carry looka-head technique.

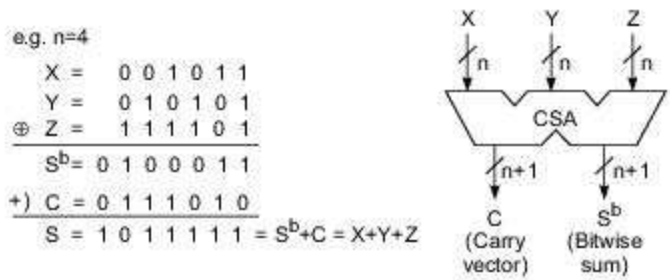
In a CSA, the carries are not allowed to propagate but instead are saved in a carry vector. In general, an n-bit CSA is specified as follows: Let X, Y, and Z be three n-bit input numbers, expressed as $X = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ and so on. The CSA performs bitwise operations simultaneously on all columns of digits to produce two n-bit output numbers, denoted as $S^b = (0, S_{n-1}, S_{n-2}, \ldots, S_1, S_0)$ and $C = (C_n, C_{n-1}, \ldots, C_1, 0)$.

Note that the leading bit of the *bitwise sum* S^b is always a 0, and the tail bit of the *carry vector* C is always a 0. The input-output relationships are expressed below:

$$S_i = x_i \oplus y_i \oplus z_i$$
$$C_{i+1} = x_i y_i \vee y_i z_i \vee z_i x_i \qquad (6.21)$$



(a) An *n*-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique



(b) An *n*-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y, and Z, and C is a carry vector generated without carry propagation between digits

Fig. 6.22 Distinction between a carry-propagate adder (CPA) and a carry-save adder (CSA)

for $i = 0,1,2,\ldots,n-1$, where $\oplus$ is the exclusive OR and $\vee$ is the logical OR operation. Note that the arithmetic sum of three input numbers, i.e., $S = X + Y + Z$, is obtained by adding the two output numbers, i.e., $S = S^b + C$, using a CPA. We use the CPA and CSAs to implement the pipeline stages of a fixed-point multiply unit as follows.

Multiply Pipeline Design Consider as an example the multiplication of two 8-bit integers $A \times B = P$, where P is the 16-bit product. This fixed-point multiplication can be written as the summation of eight partial products as shown below: $P = A \times B = P_0 + P_1 + P_2 + \cdots + P_7$, where $\times$ and $+$ are arithmetic multiply and add operations, respectively.

								1	0	1	1	0	1	0	1	=	A
							×)	1	0	0	1	0	0	1	1	=	B
								1	0	1	1	0	1	0	1	=	P_0
							1	0	1	1	0	1	0	1	0	=	P_1
						0	0	0	0	0	0	0	0	0	0	=	P_2
					0	0	0	0	0	0	0	0	0	0	0	=	P_3
				1	0	1	1	0	1	0	1	0	0	0	0	=	P_4
			0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_5
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_6
+)	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	=	P_7
0	1	1	0	0	1	1	1	1	1	1	0	1	1	1	1	=	P

Note that the partial product P_j is obtained by multiplying the multiplicand A by the jth bit of B and then shifting the result j bits to the left for $j = 0, 1, 2, \ldots, 7$. Thus P_j is $(8 + j)$ bits long with j trailing zeros. The summation of the eight partial products is done with a *Wallace tree* of CSAs plus a CPA at the final stage, as shown in Fig. 6.23.

The first stage (S_1) generates all eight partial products, ranging from 8 bits to 15 bits, simultaneously. The second stage (S_2) is made up of two levels of four CSAs, and it essentially merges eight numbers into four numbers ranging from 13 to 15 bits. The third stage (S_3) consists of two CSAs, and it merges four numbers from S_2 into two 16-bit numbers. The final stage (S_4) is a CPA, which adds up the last two numbers to produce the final product P.

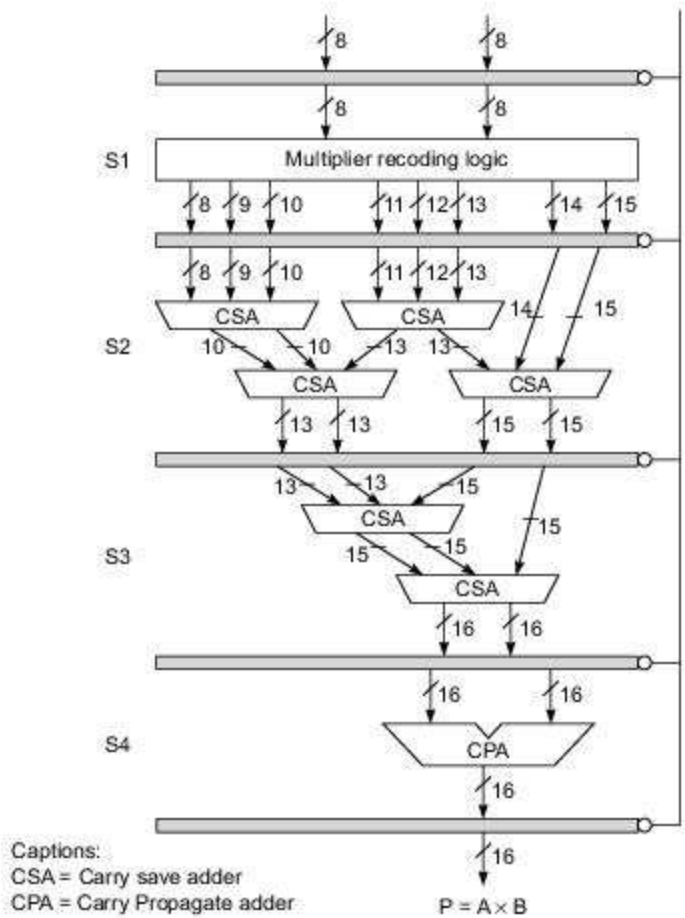


Fig. 6.23 A pipeline unit for fixed-point multiplication of 8-bit integers (The number along each line indicates the line width.)

For a maximum width of 16 bits, the CPA is estimated to need four gate levels of delay. Each level of the CSA can be implemented with a two-gate-level logic. The delay of the first stage (S_1) also involves two gate levels. Thus all the pipeline stages have an approximately equal amount of delay.

The matching of stage delays is crucial to the determination of the number of pipeline stages, as well as the clock period (Eq. 6.1). If the delay of the CPA stage can be further reduced to match that of a single CSA level, then the pipeline can be divided into six stages with a clock rate twice as fast. The basic concepts can be extended to operands with a larger number of bits, as we see in the example below.

Example 6.10 The floating-point unit in the Motorola MC68040

Figure 6.24 shows the design of a pipelined floating-point unit built as an on-chip feature in the Motorola M68040 processor.

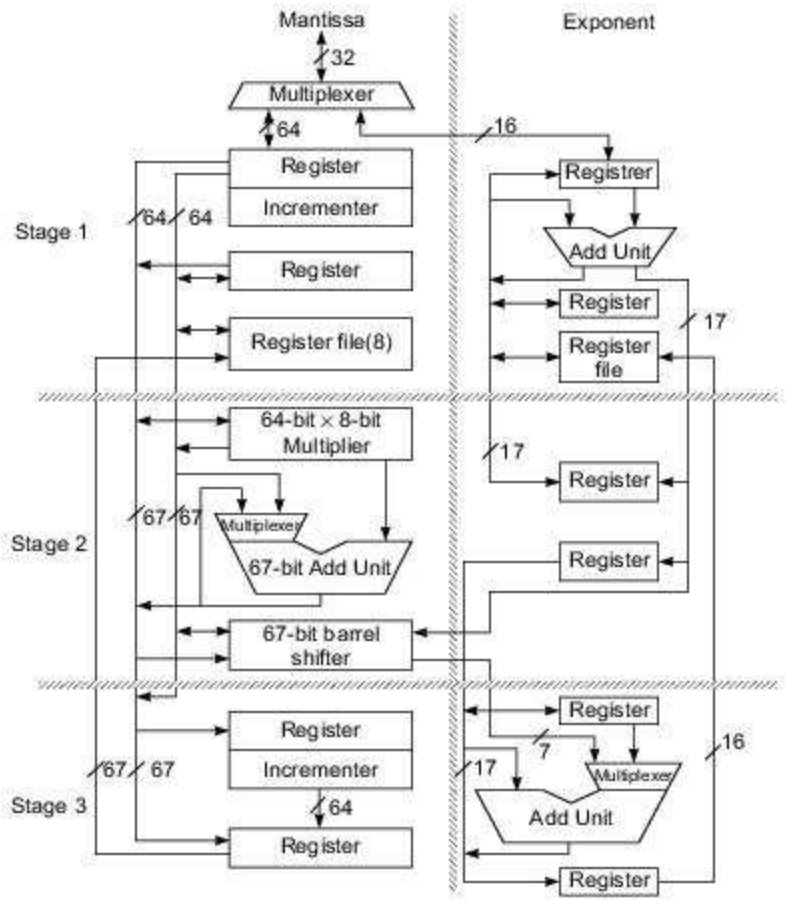


Fig. 6.24 Pipelined floating-point unit of the Motorola MC68040 processor (Courtesy of Motorola, Inc., 1992)

This arithmetic pipeline has three stages. The mantissa section and exponent section are essentially two

separate pipelines. The mantissa section can perform floating-point add or multiply operations, either single-precision (32 bits) or double-precision (64 bits).

In the mantissa section, stage 1 receives input operands and returns with computation results; 64-bit registers are used in this stage. Note that all three stages are connected to two 64-bit data buses. Stage 2 contains the array multiplier (64 × 8) which must be repeatedly used to carry out a long multiplication of the two mantissas.

The 67-bit adder performs the addition/subtraction of two mantissas, the barrel shifter is used for normalization. Stage 3 contains registers for holding results before they are loaded into the register file in stage 1 for subsequent use by other instructions.

On the exponent side, a 16-bit bus is used between stages. Stage 1 has an exponent adder for comparing the relative magnitude of two exponents. The result of stage 1 is used to equalize the exponents before mantissa addition can be performed. Therefore, a shift count (from the output of the exponent adder) is sent to the barrel shifter for mantissa alignment.

After normalization of the final result (getting rid of leading zeros), the exponent needs to be readjusted in stage 3 using another adder. The final value of the resulting exponent is fed from the register in stage 3 to the register file in stage 1, ready for subsequent usage.

Convergence Division One technique for division involves repeated multiplications. Mantissa division is carried out by a *convergence method*. This convergence division obtains the quotient $Q = M/D$ of two normalized fractions $0.5 \le M < D < 1$ in two's complement notation by performing two sequences of chain multiplications as follows:

$$Q = \frac{M \times R_1 \times R_2 \times \cdots \times R_k}{D \times R_1 \times R_2 \times \cdots \times R_k} \tag{6.22}$$

where the successive multipliers

$$R_i = 1 + \delta^{2^{i-1}} = 2 - D^{(i)} \quad \text{for } i = 1, 2, \ldots, k \quad \text{and} \quad D = 1 - \delta$$

The purpose is to choose R_i such that the denominator $D^{(k)} = D \times R_1 \times R_2 \times \cdots \times R_k \to 1$ for a sufficient number of k iterations, and then the resulting numerator $M \times R_1 \times R_2 \times \cdots \times R_k \to Q$.

Note that the multiplier R_i can be obtained by finding the two's complement of the previous chain product $D^{(i)} = D \times R_1 \times \cdots \times R_{i-1} = 1 - \delta^{2^{i-1}}$ because $2 - D^{(i)} = R_i$. The reason why $D^{(k)} \to 1$ for large k is that

$$\begin{aligned} D^{(i)} &= (1 - \delta)(1 + \delta)(1 + \delta^2)(1 + \delta^4) \cdots (1 + \delta^{2^{i-1}}) \\ &= (1 - \delta^2)(1 + \delta^2)(1 + \delta^4) \cdots (1 + \delta^{2^{i-1}}) \\ &= (1 - \delta^{2^i}) \quad \text{for } i = 1, 2, \ldots, k \end{aligned} \tag{6.23}$$

Since $0 < \delta = 1 - D \le 0.5$, $\delta^{2^i} \to 0$ as i becomes sufficiently large, say, $i = k$ for some k; thus $D^{(k)} = 1 - \delta^{2^k} = 1$ for large k. The end result is

$$Q = M \times (1 + \delta) \times (1 + \delta^2) \times \cdots \times (1 + \delta^{2^{k-1}}) \tag{6.24}$$

The above two sequences of chain multiplications are carried out alternately between the numerator and denominator through the pipeline stages. To summarize, in this technique division is carried out by repeated multiplications. Thus divide and multiply can share the same hardware pipeline.

Example 6.11 The IBM 360/Model 91 floating-point unit design

In the history of building scientific computers, IBM 360 Model 91 was certainly a milestone. Many of the pipeline design features introduced in previous sections were implemented in this machine. Therefore, it is worth the effort to examine the architecture of Model 91. In particular, we describe how floating-point add and multiply/divide operations were implemented in this machine.

As shown in Fig. 6.25, the floating-point execution unit in Model 91 consisted of two separate functional pipelines: the *add unit* and the *multiply/divide unit*, which could be used concurrently. The former was a two-stage pipeline, and the latter was a six-stage pipeline.

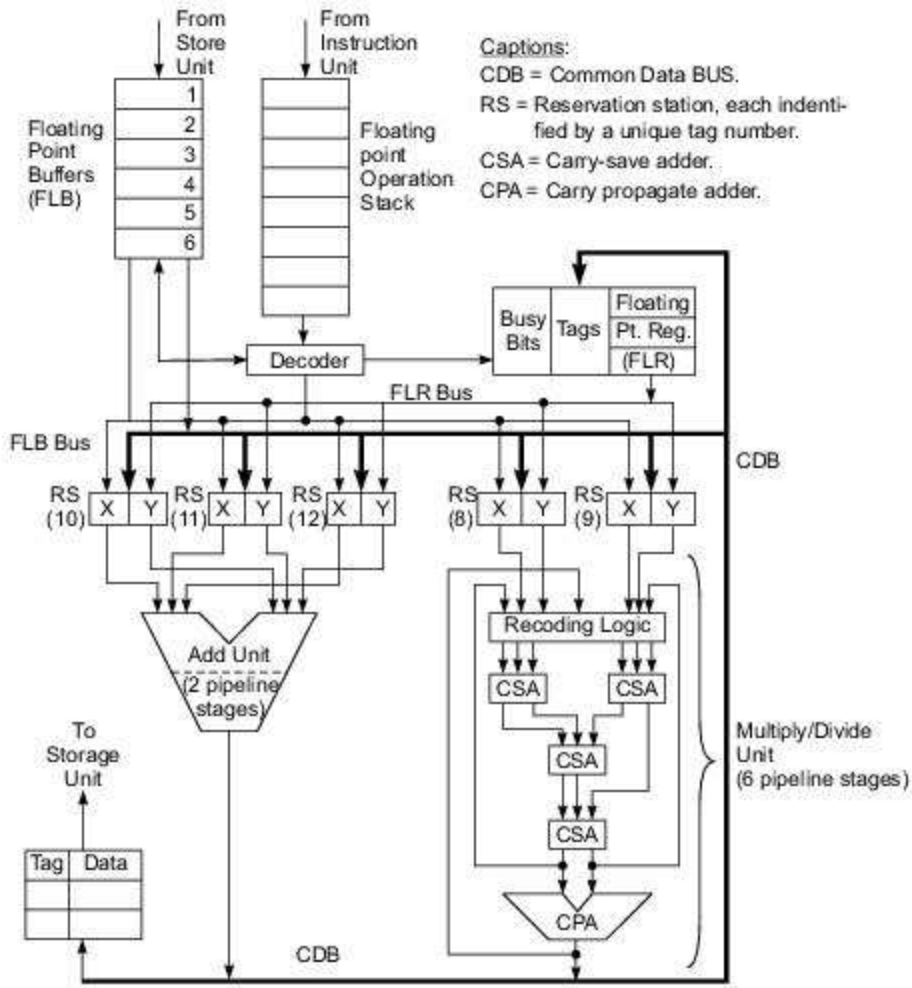


Fig. 6.25 The IBM 360 Model 91 floating-point unit (Courtesy of IBM Corporation, 1967)

The floating-point operation stack was a kind of prefetch buffer holding eight floating-point instructions for subsequent execution through the two functional pipelines. The floating-point buffers were used to input operands.

Operands may also come from the floating-point registers which were connected via the common data bus to the output bus. Results from the two functional units could be sent back to the memory via the store data buffers, or they could be routed back to the FLR or to the reservation stations at the input ends.

The add unit allowed three pairs of operands to be loaded into three reservation stations. Only one pair could be used at a time. The other two pairs held operands for subsequent use. The use of these reservation stations made the add unit behave like three virtual functional units.

Similarly, the two pairs at the input end of the multiply/divide unit made it behave like two virtual units. Internal data forwarding in Model 91 was accomplished using source tags on all registers and reservation stations. Divide was implemented in Model 91 based on the convergence method.

Every source of an input operand was uniquely identified with a 4-bit tag. Every destination of an input operand had an associated tag register that held the tag naming the source of data if the destination was busy. Through this *register tagging* technique, operands/results could be directly passed among the virtual functional units. This forwarding significantly cut down the data flow time between them.

Dynamic scheduling logic was built into Model 91 using Tomasulo's algorithm to resolve the data dependence problem. Either the add unit or the multiply/divide unit could execute an operation using operands from one of the reservation stations.

Under Tomasulo's algorithm, data dependences are preserved by copying source tags when the sources are busy. When data is generated by a source, it passes its identification and the data onto the common data bus. Awaiting destinations continuously monitor the bus in a tag watch.

When the source tag matches, the destination takes in the data from the bus. Other variations of Tomasulo's algorithm can be made to store the source tags within the destinations, to use a special tag (such as 0000) to indicate nonbusy register/buffers, or to use direct-mapped tags to avoid associative hardware.

Besides the IBM 360/370, the CDC 6600/7600 also implemented convergence division. It took two pipeline cycles to perform the floating-point add, six cycles to multiply, and 18 cycles to divide in the IBM System/360 Model 91 due to five iterations involved in the convergence division process.

6.4.3 Multifunctional Arithmetic Pipelines

Static arithmetic pipelines are designed to perform a fixed function and are thus called *unifunctional*. When a pipeline can perform more than one function, it is called *multifunctional*. A multifunctional pipeline can be either *static* or *dynamic*. Static pipelines perform *one* function at a time, but different functions can be performed at different times. A dynamic pipeline allows several functions to be performed simultaneously through the pipeline, as long as there are no conflicts in the shared usage of pipeline stages. In this section, we study a static multifunctional pipeline which was designed into the TI Advanced Scientific Computer (ASC).

Example 6.12 The TI/ASC arithmetic processor design

There were four pipeline arithmetic units built into the TI-ASC system, as shown in Fig. 6.26. The instruction-processing unit handled the fetching and decoding of instructions. There were a large number of working registers in the processor which also controlled the operations of the memory buffer unit and of the arithmetic units.

There were two sets of operand buffers, $\{X, Y, Z\}$ and $\{X', Y', Z'\}$, in each arithmetic unit. X', X, Y' and Y were used for input operands, and Z' and Z were used to output results. Note that intermediate results could be also routed from Z-registers to either X- or Y-registers. Both processor and memory buffers accessed the main memory for instructions and operands/results, respectively.

Each pipeline arithmetic unit had eight stages as shown in Fig. 6.27a. The PAU was a static multifunction pipeline which could perform only one function at a time. Figure 6.27a shows all the possible interstage connections for performing arithmetic, logical, shifting, and data conversion functions.

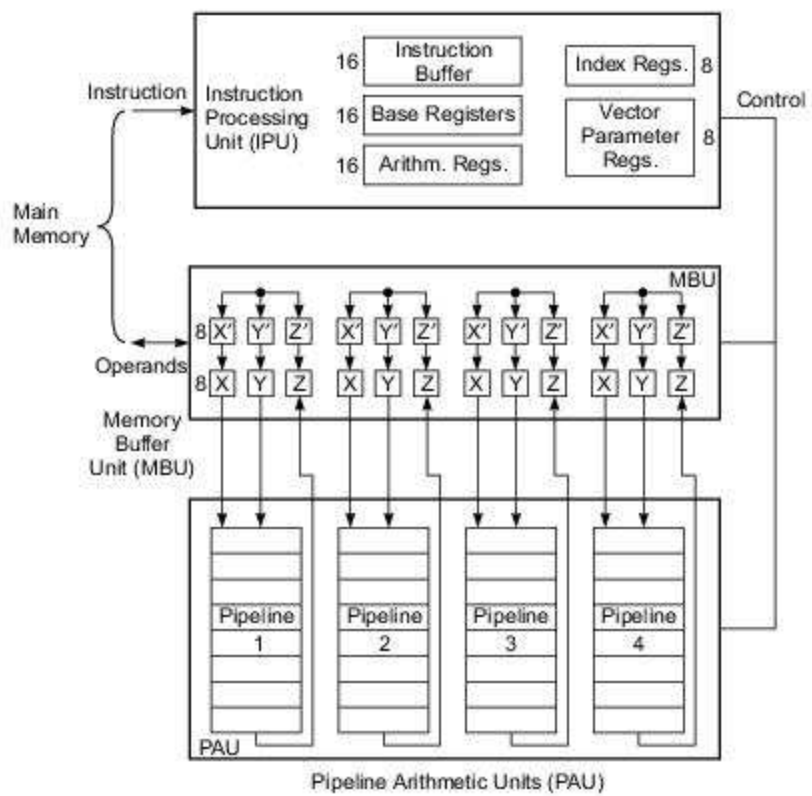
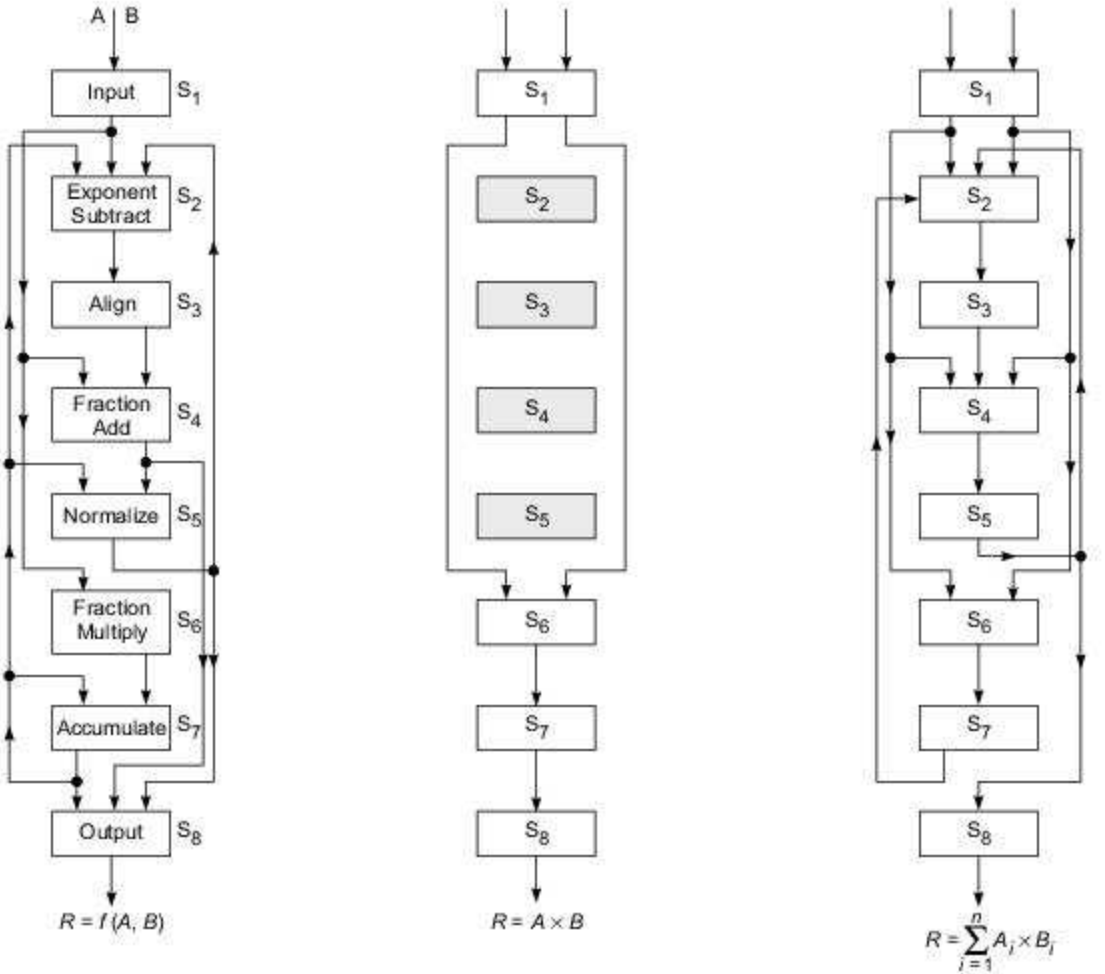


Fig. 6.26 The architecture of the TI Advanced Scientific Computer (ASC) (Courtesy of Texas Instruments, Inc.)

Both fixed-point and floating-point arithmetic functions could be performed by this pipeline. The PAU also supported vector in addition to scalar arithmetic operations. It should be noted that different functions required different pipeline stages and different interstage connection patterns.



(a) Pipeline stages and interconnections

(b) Fixed-point multiplication

(c) Floating-point dot product

Fig. 6.27 The multiplication arithmetic pipeline of the TI Advanced Scientific Computer and the interstage connections of two representative functions (Shaded stages are unutilized)

For example, fixed-point multiplication required the use of only segments S_1, S_6, S_7, and S_8 as shown in Fig. 6.27b. On the other hand, the floating-point dot product function, which performs the dot product operation between two vectors, required the use of all segments with the complex connections shown in Fig. 6.27c. This dot product was implemented by essentially the following accumulated summation of a sequence of multiplications through the pipeline:

$$Z \leftarrow A_i \times B_i + Z \tag{6.25}$$

where the successive operands (A_i, B_i) were fed through the X- and Y-buffers, and the accumulated sums through the Z-buffer recursively.

The entire pipeline could perform the *multiply* (×) and the *add* (+) in a single flow through the pipeline. The two levels of buffer registers isolated the loading and fetching of operands to or from the PAU, respectively, as in the concept of using a pair in the prefetch buffers described in Fig. 6.11.

Even though the TI-ASC is no longer in production, the system provided a unique design for multifunction arithmetic pipelines. Today, most supercomputers implement arithmetic pipelines with dedicated functions for much simplified control circuitry and faster operations.

6.5 SUPERSCALAR PIPELINE DESIGN

Pipeline Design Parameters Some parameters used in designing the scalar base processor and superscalar processor are summarized in Table 6.1 for the pipeline processors to be studied below. All pipelines discussed are assumed to have k stages.

The *pipeline cycle* for the scalar base processor is assumed to be 1 time unit, called the *base cycle*. We defined the instruction *issue rate*, *issue latency*, and *simple operation latency* in Section 4.1.1. The *instruction-level parallelism* (ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline.

For the base processor, all of these parameters have a value of 1. All processor types are designed relative to the base processor. The ILP is needed to fully utilize a given pipeline processor.

Table 6.1 *Design Parameters for Pipeline Processors*

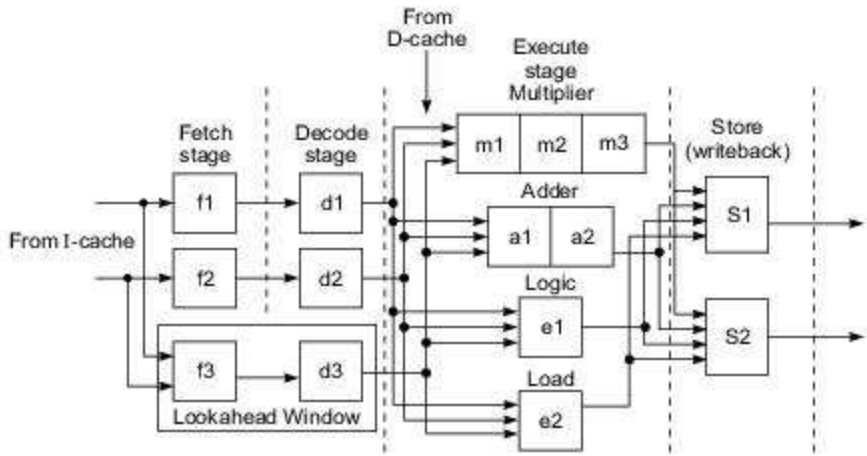
Machine type	*Scalar base machine of k pipeline stages*	*Superscalar machine of degree m*
Machine pipeline cycle	1 (base cycle)	1
Instruction issue rate	1	m
Instruction issue latency	1	1
Simple operation latency	1	1
ILP to fully utilize the pipeline	1	m

Note: All timing is relative to the base cycle for the scalar base machine. ILP: Instruction level parallelism.

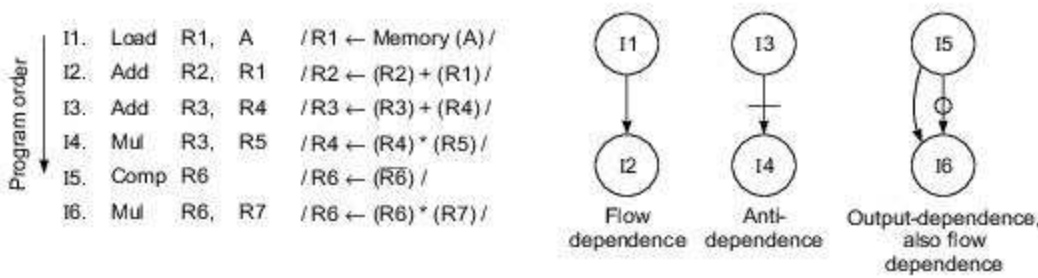
We study below the structure of superscalar pipelines, the data dependence problem, the factors causing pipeline stalling, and multi-instruction issuing mechanisms for achieving parallel pipelining operations. For a superscalar machine of degree m, m instructions are issued per cycle and the ILP should be m in order to fully utilize the pipeline. As a matter of fact, the scalar base processor can be considered a degenerate case of a superscalar processor of degree 1.

Superscalar Pipeline Structure In an m-issue superscalar processor, the instruction decoding and execution resources are increased to form effectively m pipelines operating concurrently. At some pipeline stages, the functional units may be shared by multiple pipelines.

This resource-shared multiple-pipeline structure is illustrated by a design example in Fig. 6.28a. In this design, the processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem. There are essentially two pipelines in the design. Both pipelines have four processing stages labeled fetch, decode, execute, and store, respectively.



(a) A dual-pipleline, superscalar processor with four functional units in the execution stage and a lookahead window producing out-of-order issues



(b) A sample program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the multiplier

Fig. 6.28 A two-issue superscalar processor and a sample program for parallel execution

Each pipeline essentially has its own fetch unit, decode unit, and store unit. The two instruction streams flowing through the two pipelines are retrieved from a single source stream (the I-cache). The fan-out from a single instruction stream is subject to resource constraints and a data dependence relationship among the successive instructions.

For simplicity, we assume that each pipeline stage requires one cycle, except the execute stage which may require a variable number of cycles. Four functional units, multiplier, adder, logic unit, and load unit, are available for use in the execute stage. These functional units are shared by the two pipelines on a dynamic basis. The multiplier itself has three pipeline stages, the adder has two stages, and the others each have only one stage.

The two store units (S1 and S2) can be dynamically used by the two pipelines, depending on availability at a particular cycle. There is a *lookahead window* with its own fetch and decoding logic. This window is used for instruction lookahead in case out-of-order instruction issue is desired to achieve better pipeline throughput.

It requires complex logic to schedule multiple pipelines simultaneously, especially when the instructions are retrieved from the same source. The aim is to avoid pipeline stalling and minimize pipeline idle time.

Data Dependences Consider the example program in Fig. 6.28b. A dependence graph is drawn to indicate the relationship among the instructions. Because the register content in R1 is loaded by I1 and then used by I2, we have flow dependence: I1→ I2.

Because the result in register R4 after executing I4 may affect the operand register R4 used by I3, we have antidependence: I3 +→ I4. Since both I5 and I6 modify the register R6, and R6 supplies an operand for I6, we have both flow and output dependence: I5 → I6 and I5 o→ I6 as shown in the dependence graph.

To schedule instructions through one or more pipelines, these data dependences must not be violated. Otherwise, erroneous results may be produced.

Pipeline Stalling This is a problem which may seriously lower pipeline utilization. Proper scheduling avoids pipeline stalling. The problem exists in both scalar and superscalar processors. However, it is more serious in a superscalar pipeline. Stalling can be caused by data dependences or by resource conflicts among instructions already in the pipeline or about to enter the pipeline. We use an example to illustrate the conditions causing pipeline stalling.

Consider the scheduling of two instruction pipelines in a two-issue superscalar processor. Figure 6.29a shows the case of no data dependence on the left and flow dependence (I1 → I2) on the right. Without data dependence, all pipeline stages are utilized without idling.

With dependence, instruction I2 entering the second pipeline must wait for two cycles (shaded time slots) before entering the execution stages. This delay may also pass to the next instruction I4 entering the pipeline.

In Fig. 6.29b, we show the effect of branching (instruction I2). A delay slot of four cycles results from a branch taken by I2 at cycle 5. Therefore, both pipelines must be flushed before the target instructions I3 and I4 can enter the pipelines from cycle 6. Here, delayed branch or other amending actions are not taken.

In Fig. 6.29c, we show a combined problem involving both resource conflict and data dependence. Instructions I1 and I2 need to use the same functional unit, and I2 → I4 exists.

The net effect is that I2 must be scheduled one cycle behind because the two pipeline stages (e_1 and e_2) of the same functional unit must be used by 11 and I2 in an overlapped fashion. For the same reason, I3 is also delayed by one cycle. Instruction I4 is delayed by two cycles due to the flow dependence on I2. The shaded boxes in all the timing charts correspond to idle stages.

Superscalar Pipeline Scheduling Instruction issue and completion policies are critical to superscalar processor performance. Three scheduling policies are introduced below. When instructions are issued in program order, we call it *in-order issue*. When program order is violated, *out-of-order issue* is being practiced.

Similarly, if the instructions must be completed in program order, it is called *in-order completion*. Otherwise, *out-of-order completion* may result. In-order issue is easier to implement but may not yield the optimal performance. In-order issue may result in either in-order or out-of-order completion.

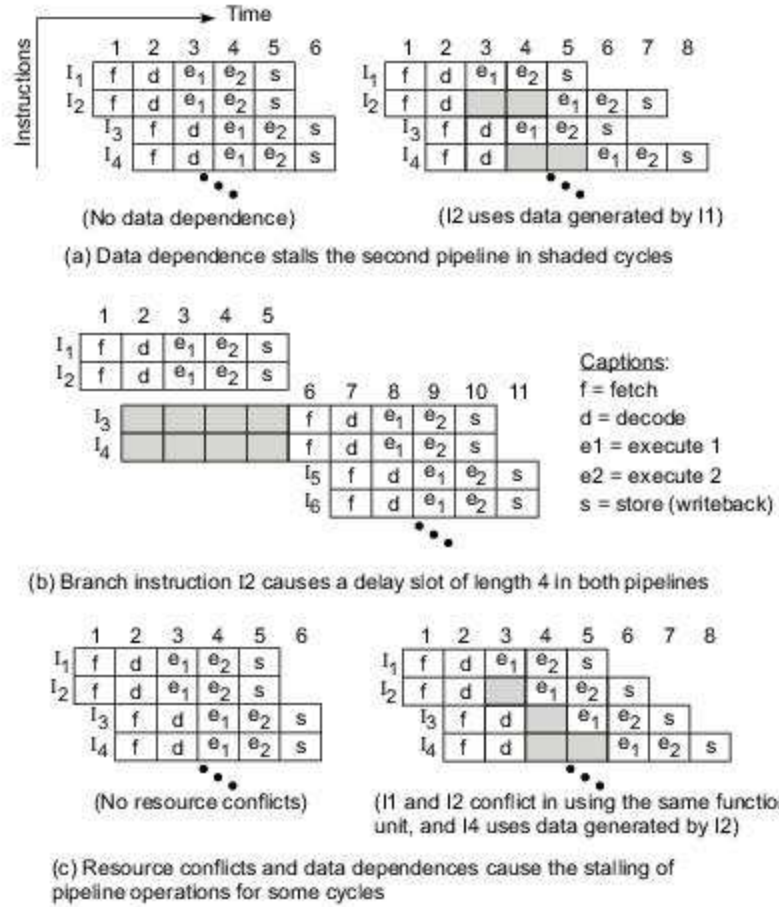


Fig. 6.29 Dependences and resource conflicts may stall one or two pipelines in a two-issue superscalar processor

Out-of-order issue usually ends up with out-of-order completion. The purpose of out-of-order issue and completion is to improve performance. These three scheduling policies are illustrated in Fig. 6.30 by execution of the example program in Fig. 6.28b on the dual-pipeline hardware in Fig. 6.28a.

It is demonstrated that performance can be improved from an in-order to an out-of-order schedule. The performance is often indicated by the total execution time and the utilization rate of pipeline stages. Not all programs can be scheduled out of order. Data dependence and resource conflicts do impose constraints.

In-Order Issue Figure 6.30a shows a schedule for the six instructions being issued in program order I1, I2, …, I6. Pipeline 1 receives I1, I3, and I5, and pipeline 2 receives I2, I4, and I6 in three consecutive cycles. Due to I1 → I2, I2 has to wait one cycle to use the data loaded in by I1.

I3 is delayed one cycle for the same adder used by I2. I6 has to wait for the result of I5 before it can enter the multiplier stages. In order to maintain in-order completion, I5 is forced to wait for two cycles to come out of pipeline 1. In total, nine cycles are needed and five idle cycles (shaded boxes) are observed.

In Fig. 6.30b, out-of-order completion is allowed even if in-order issue is practiced. The only difference between this out-of-order schedule and the in-order schedule is that I5 is allowed to complete ahead of I3 and I4, which are totally independent of I5. The total execution time does not improve. However, the pipeline utilization rate does.

Only three idle cycles are observed. Note that in Figs. 6.29a and 6.29b, we did not use the lookahead window. In order to shorten the total execution time, the window can be used to reorder the instruction issues.

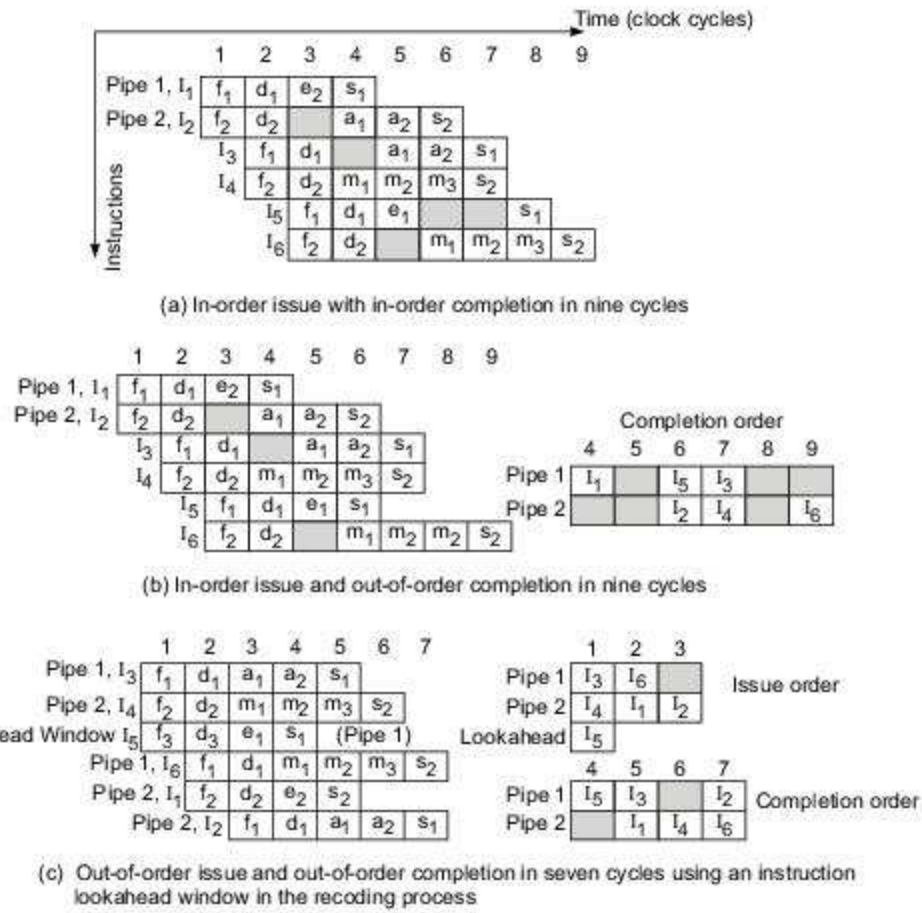


Fig. 6.30 Instruction issue and completion policies for a superscalar processor with and without instruction lookahead support (Timing charts correspond to parallel execution of the program in Fig. 6.28)

Out-of-Order Issue By using the lookahead window, instruction I5 can be decoded in advance because it is independent of all the other instructions. The six instructions are issued in three cycles as shown: I5 is fetched and decoded by the window, while I3 and I4 are decoded concurrently.

It is followed by issuing I6 and I1 at cycle 2, and I2 at cycle 3. Because the issue is out of order, the completion is also out of order as shown in Fig. 6.30c. Now, the total execution time has been reduced to seven cycles with no idle stages during the execution of these six instructions.

The in-order issue and completion is the simplest one to implement. It is rarely used today even in a conventional scalar processor due to some unnecessary delays in maintaining program order. However, in a multiprocessor environment, this policy is still attractive. Allowing out-of-order completion can be found in both scalar and superscalar processors.

Some long-latency operations, such as loads and floating-point operations, can be hidden in out-of-order completion to achieve a better performance. Output dependence and antidependence are the two relations preventing out-of-order completion. Out-of-order issue gives the processor more freedom to exploit parallelism, and thus pipeline efficiency is enhanced.

The above example clearly demonstrates the advantages of instruction lookahead and of out-of-order issue and completion as far as pipeline optimization is concerned. It should be noted that multiple-pipeline scheduling is an NP-complete problem. Optimal scheduling is very expensive to obtain.

Simple data dependence checking, a small lookahead window, and scoreboarding mechanisms are needed, along with an optimizing compiler, to exploit instruction parallelism in a superscalar processor.

Motorola 88110 Architecture The Motorola 88110 was an early superscalar RISC processor. It combined the three-chip set, one CPU (88100) chip and two cache (88200) chips, in a single-chip implementation, with additional improvements. The 88110 employed advanced techniques for exploiting instruction-level parallelism, including instruction issue, out-of-order instruction completion, speculative execution, dynamic instruction rescheduling, and two on-chip caches. The unit also supported demanding graphics and digital signal processing applications.

The 88110 employed a symmetrical superscalar instruction dispatch unit which dispatched two instructions each clock cycle into an array of 10 concurrent units. It allowed out-of-order instruction completion and some out-of-order instruction issue, and branch prediction with speculative execution past branches.

The instruction set of the 88110 extended that of the 88100 in integer and floating-point operations. It added a new set of capabilities to support 3-D color graphics image rendering. The 88110 had separate, independent instruction and data paths, along with split caches for instructions and data. The instruction cache was 8K-byte, 2-way set-associative with 128 sets, two blocks for each set, and 32 bytes (8 instructions) per block. The data cache resembled that of the instruction set.

The 88110 employed the MESI cache coherence protocol. A write-invalidate procedure guaranteed that one processor on the bus had a modified copy of any cache block at any time. The 88110 was implemented with 1.3 million transistors in a 299-pin package and driven by a 50-MHz clock. Interested readers may refer to Diefendorff and Allen (1992) for details.

Superscalar Performance To compare the relative performance of a superscalar processor with that of a scalar base machine, we estimate the ideal execution time of N independent instructions through the pipeline.

The time required by the scalar base machine is

$$T(1, 1) = k + N - 1 \text{ (base cycles)} \tag{6.26}$$

The ideal execution time required by an m-issue superscalar machine is

$$T(m, 1) = k + \frac{N - m}{m} \text{ (base cycles)} \tag{6.27}$$

where k is the time required to execute the first m instructions through the m pipelines simultaneously, and the second term corresponds to the time required to execute the remaining $N - m$ instructions, m per cycle, through m pipelines.

The ideal speedup of the superscalar machine over the base machine is

$$S(m, 1) = \frac{T(1,1)}{T(m,1)} = \frac{N+k-1}{N/m+k-1} = \frac{m(N+k-1)}{N+m(k-1)} \tag{6.28}$$

As $N \to \infty$, the speedup limit $S(m, 1) \to m$, as expected.

Example 6.13 DEC Alpha 21064 superscalar architecture

As illustrated in Fig. 6.31, this was a 64-bit superscalar processor. The design emphasized speed, multiple-instruction issue, multiprocessor applications, software migration from the VAX/VMS and MIPS/OS, and a long list of usable features. The clock rate was 150 MHz with the first chip implementation.

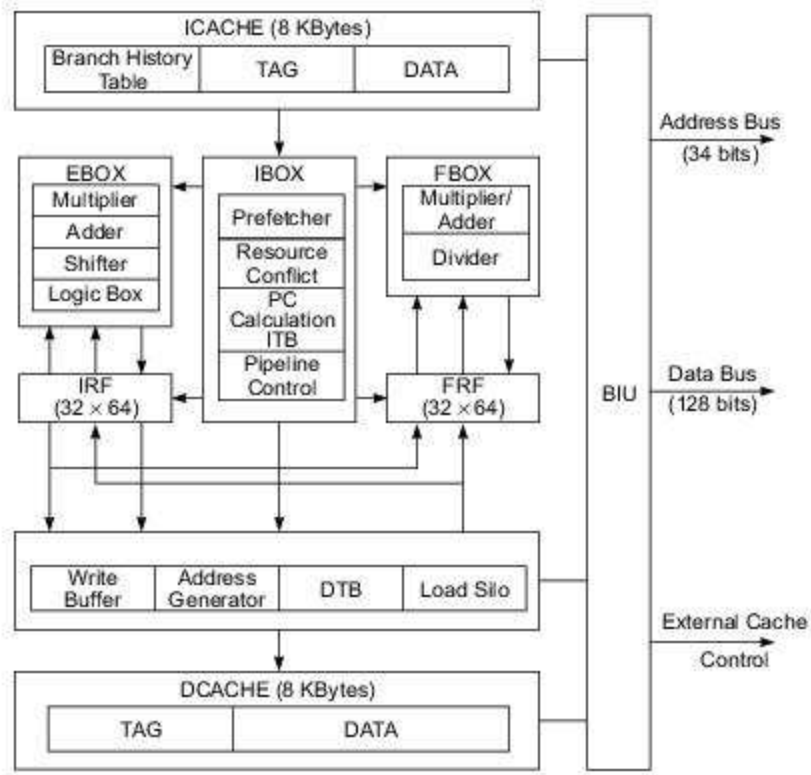


Fig. 6.31 Architecture of the DEC Alpha 21064 processor (Courtesy of Digital Equipment Corporation)

Unlike others, the Alpha architecture had thirty-two 64-bit integer registers and thirty-two 64-bit floating-point registers. The integer pipeline had 7 stages, and the floating-point pipeline had 10 stages. All Alpha instructions were 32 bits.

The first Alpha implementation issued two instructions per cycle, with larger number of issues in later implementations. Pipeline timing hazards, load delay slots, and branch delay slots were all minimized by hardware support. The Alpha was designed to support fast multiprocessor interlocking and interrupts.

A privileged library of software was developed to run full VMS and to run OSF/1 using different versions of the software library that mirrored many of the VAX/VMS and MIPS/OS features, respectively. This library made Alpha an attractive architecture for multiple operating systems. The processor was designed to have a 300-MIPS peak and a 150-Mflops peak at 150 MHz.

Note 6.2 Innovation versus commercial success

The relationship between innovative design ideas and the commercial success of a product is not always simple, as an idealist may believe.

Most of the processors used as examples in this chapter are no longer in commercial production. Rapid advances in technology and immense pressures from the market-place are usually the two main reasons behind the introduction and the demise of newer processor models. However, the innovative design ideas introduced in a new processor often have a life longer than the processor itself, since these same ideas are often carried forward in subsequent designs of the same or other processor families.

For example, IBM 360/91, Motorola 68040, Motorola 88110 and DEC Alpha 21064 were all recognized for their innovative designs when they were introduced, but they achieved different degrees of commercial success. Our aim in this book is to study the innovative ideas embodied in processor and system designs; but we must also appreciate that the commercial success of a product often depends on many other crucial factors.

Summary

Instruction pipelines in processors usually have a linear structure—the execution of each instruction progresses linearly, one stage at a time, from the first to the last pipeline stage. In theory, such a linear pipeline can be designed with synchronous or asynchronous timing mode; in practice, processor pipelines today operate in synchronous mode, i.e. with a common clock signal. We studied the timing and clocking requirements of linear pipelines, and discussed the related speedup, efficiency and throughput issues. A simple model was presented which can be used in determining the optimal number of pipeline stages, based on a trade-off between cost and throughput.

Dynamic or nonlinear pipelines are designed to perform a number of different functions, by appropriate scheduling of operations on the pipeline stages. Reservation tables are used for different functions; collision free schedules and latency analysis are needed for efficient operation of nonlinear pipelines. We studied how concepts of collision vectors, state transition diagrams and greedy cycles are used to determine bounds on minimum average latency (MAL), and thereby optimum schedules in terms of MAL.

For a given machine instruction set, instruction pipeline design begins with analysis of the execution phases of instructions through the processor; we used the MIPS R4000 instruction pipeline as a specific example. Processor performance can be enhanced by techniques such as prefetch buffers, multiple functional units, and data forwarding; in addition, hazard avoidance is a constant goal in pipeline design and scheduling. Dynamic instruction scheduling was discussed, with a look at both Tomasulo's algorithm and the technique of scoreboarding developed at CDC.

Branches in the flow of execution of instructions have a major impact on pipeline performance, since they may result in the instruction pipeline being flushed. We used a simple model to estimate the effect of branches on processor throughput, and discussed several useful branch handling techniques such as dynamic branch prediction, branch target buffer, and delayed branch.

We reviewed the standard IEEE floating point representation and the basic principles of floating point arithmetic. Principles of static and multifunctional arithmetic pipelines were studied, with specific examples of arithmetic pipeline design from Motorola 68040, IBM 360/91, and TI Advanced Scientific Computer.

A superscalar pipeline is one in which multiple instructions can be issued in parallel in each clock cycle, so as to better exploit instruction level parallelism in the running program. In this process, data dependences, anti-dependences and output dependences between instructions must also be respected. We reviewed in-order versus out-of-order instruction issue, and carried out basic performance analysis of superscalar pipelines. Motorola 88110 and DEC Alpha 21064 processors were used as specific examples.

Exercises

Problem 6.1 Consider the execution of a program of 15,00,000 instructions by a linear pipeline processor with a clock rate of 1000 MHz. Assume that the instruction pipeline has five stages and that one instruction is issued per clock cycle. The penalties due to branch instructions and out-of-sequence executions are ignored.

(a) Calculate the speedup factor in using this pipeline to execute the program as compared with the use of an equivalent nonpipelined processor with an equal amount of flow-through delay.

(b) What are the efficiency and throughput of this pipelined processor?

Problem 6.2 Study the DEC Alpha architecture in Example 6.13, find more information on DEC Alpha on the web and then answer the following questions with reasoning:

(a) Analyze the scalability of the Alpha processor implementation in terms of superscalar degree.

(b) Analyze the scalability of an Alpha-based multiprocessor system in terms of address space and multiprocessor support.

Problem 6.3 Find the optimal number of pipeline stages k_0 given in Eq. 6.7 using the performance/cost ratio (PCR) given in Eq. 6.6.

Problem 6.4 Prove the lower bound and upper bound on the minimal average latency (MAL) specified in Section 6.2.3.

Problem 6.5 Consider the following reservation

table for a four-stage pipeline with a clock cycle $\tau = 2$ ns.

	1	2	3	4	5	6
S1	X					X
S2		X		X		
S3			X			
S4				X	X	

(a) What are the forbidden latencies and the initial collision vector?

(b) Draw the state transition diagram for scheduling the pipeline.

(c) Determine the MAL associated with the shortest greedy cycle.

(d) Determine the pipeline throughput corresponding to the MAL and given τ.

(e) Determine the lower bound on the MAL for this pipeline. Have you obtained the optimal latency from the above state diagram?

Problem 6.6 You are allowed to insert one noncompute delay stage into the pipeline in Problem 6.5 to make a latency of 1 permissible in the shortest greedy cycle. The purpose is to yield a new reservation table leading to an optimal latency equal to the lower bound.

(a) Show the modified reservation table with five rows and seven columns.

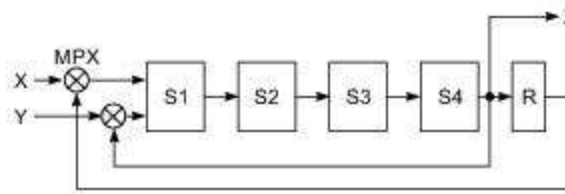
(b) Draw the new state transition diagram for obtaining the optimal cycle.

(c) List all the simple cycles and greedy cycles from the state diagram.

(d) Prove that the new MAL equals the lower bound.

(e) What is the optimal throughput of this pipeline? Indicate the percentage of throughput improvement compared with that obtained in part (d) of Problem 6.5.

Problem 6.7 Consider an adder pipeline with four stages as shown below. The pipeline consists of input lines X and Y and output line Z. The pipeline has a register R at its output where the temporary result can be stored and fed back to S1 at a later point in time. The inputs X and Y are multiplexed with the outputs R and Z.



(a) Assume the elements of the vector A are fed into the pipeline through input X, one element per cycle. What is the minimum number of clock cycles required to compute the sum of an N-element vector A: $s = \sum_{i=1}^{N} A(i)$? In the absence of an operand, a value of 0 is input into the pipeline by default. Neglect the setup time for the pipeline.

(b) Let τ be the clock period of the pipelined adder. Consider an equivalent nonpipelined adder with a flow-through delay of 4τ. Find the actual speedup $S_4(64)$ and the efficiency $\eta_4(64)$ of using the above pipeline adder for $N = 64$.

(c) Find the maximum speedup $S_4(\infty)$ and the efficiency $\eta_4(\infty)$ when N tends to infinity.

(d) Find $N_{1/2}$, the minimum vector length required to achieve half of the maximum speedup.

Problem 6.8 Consider the following pipeline reservation table.

	1	2	3	4
S1	X			X
S2		X		
S3			X	

(a) What are the forbidden latencies?

(b) Draw the state transition diagram.

(c) List all the simple cycles and greedy cycles.

(d) Determine the optimal constant latency cycle and the minimal average latency.

(e) Let the pipeline clock period be $\tau = 2$ ns. Determine the throughput of this pipeline.

Problem 6.9 Consider the five-stage pipelined processor specified by the following reservation table:

	1	2	3	4	5	6
S1	X					X
S2		X			X	
S3			X			
S4				X		
S5		X				X

(a) List the set of forbidden latencies and the collision vector.

(b) Draw a state transition diagram showing all possible initial sequences (cycles) without causing a collision in the pipeline.

(c) List all the simple cycles from the state diagram.

(d) Identify the greedy cycles among the simple cycles.

(e) What is the minimum average latency (MAL) of this pipeline?

(f) What is the minimum allowed constant cycle in using this pipeline?

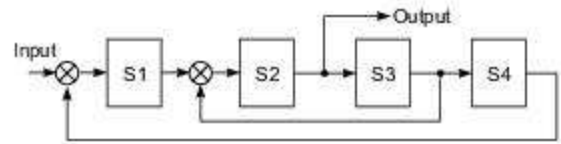
(g) What will be the maximum throughput of this pipeline?

(h) What will be the throughput if the minimum constant cycle is used?

Problem 6.10 The following assembly code is to be executed in a three-stage pipelined processor with hazard detection and resolution in each stage. The stages are instruction fetch, operand fetch (one or more as required), and execution (including a write-back operation). Explain all possible hazards in the execution of the code.

Inc R0	/R0 ← (R0) + 1/
Mul ACC, R0	/ACC ← (ACC) × (R0)/
Store R1, ACC	/R1 ← (ACC)/
Add ACC, R0	/ACC ← (ACC) + (R0)/
Store M,ACC	/M ← (ACC)/

Problem 6.11 Consider the following pipelined processor with four stages. This pipeline has a total evaluation time of six clock cycles. All successor stages must be used after each clock cycle.



(a) Specify the reservation table for this pipeline with six columns and four rows.

(b) List the set of forbidden latencies between task initiations.

(c) Draw the state diagram which shows all possible latency cycles.

(d) List all greedy cycles from the state diagram.

(e) What is the value of the minimal average latency?

(f) What is the maximal throughput of this pipeline?

Problem 6.12 Three functional pipelines f_1, f_2, and f_3 are characterized by the following reservation tables. Using these three pipelines, a composite pipeline network is formed below:

f_1:

	1	2	3	4
SI	X			
S2		X		
S3			X	X

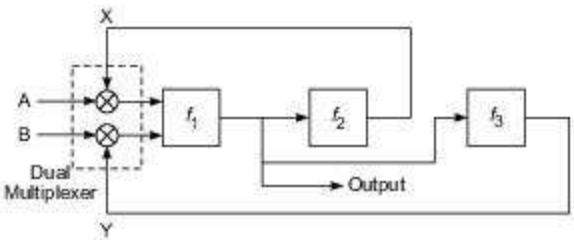
f_2:

	1	2	3	4
T1	X			X
T2		X		
T3			X	

f_3:

	1	2	3	4
U1	X		X	
U2				X
U3		X		

Each task going through this composite pipeline uses the pipeline in the following order: f_1 first, f_2 and f_3 next, f_1 again, and then the output is obtained. The dual multiplexer selects a pair of inputs, (A, B) or (X,Y), and feeds them into the input of f_1. The use of the composite pipeline is described by the combined reservation table.



(a) Complete the following reservation table for this composite pipeline.

	1	2	3	4	5	6	7	8	9	10	11	12
S1	X											
S2										X		
S3				X								
T1												
T2												
T3							X					
U1					X							
U2												
U3												

(b) Write the forbidden list and the initial collision vector.

(c) Draw a state diagram clearly showing all latency cycles.

(d) List all simple cycles and greedy cycles.

(e) Calculate the MAL and the maximal throughput of this composite pipeline.

Problem 6.13 A nonpipelined processor X has a clock rate of 250 MHz and an average CPI (cycles per instruction) of 4. Processor Y, an improved successor of X, is designed with a five-stage linear instruction pipeline. However, due to latch delay and clock skew effects, the clock rate of Y is only 200 MHz.

(a) If a program containing 1000 instructions is executed on both processors, what is the speedup of processor Y compared with that of processor X?

(b) Calculate the MIPS rate of each processor during the execution of this particular program.

Problem 6.14 Design a binary integer multiply pipeline with five stages. The first stage is for partial product generation. The last stage is a 36-bit carry-lookahead adder. The middle three stages are made of 16 carry-save adders (CSAs) of appropriate lengths.

(a) Prepare a schematic design of the five-stage multiply pipeline. All line widths and interstage connections must be shown.

(b) Determine the maximal clock rate of the pipeline if the stage delays are $\tau_1 = \tau_2 = \tau_3 = \tau_4 = 9$ ns, $\tau_5 = 4$ ns, and the latch delay is 1 ns.

(c) What is the maximal throughput of this pipeline in terms of the number of 36-bit results generated per second?

Problem 6.15 Consider a four-stage floating-point adder with a 2-ns delay per stage which equals the pipeline clock period.

(a) Name the appropriate functions to be performed by the four stages.

(b) Find the minimum number of periods required to add 100 floating-point numbers $A_1 + A_2 + \cdots + A_{100}$ using this pipeline adder, assuming that the output Z of stage S_4 can be routed back to either of the two inputs X or Y of the pipeline with delays equal to a multiple of the clock period.

Problem 6.16 Consider two four-stage pipeline adders and a number of noncompute delay elements. Each delay element has a one-unit time delay.

(a) Use the available adders and delays to construct a composite pipeline unit for evaluating the following expression: $b(i) = a(i) + a(i-1) + a(i-2) + a(i-3)$ for all $i = 4, 5,$

..., n. The composite pipeline receives $a(i)$ for $i = 1, 2, \ldots, n$, as the successive inputs.

(b) Consider a third four-stage pipeline adder. Augment the design in part (a) with this third adder to compute the following recursive expression: $x(i) = a(i) + x(i-1)$, for all $i = 4, 5, \ldots, n$. Note that $x(i) = a(i) + x(i-1) = a(i) + [a(i-1) + x(i-2)] = \cdots = b(i) + x(i-4)$, where $b(i)$ is generated by the composite pipeline in part (a).