Author: Dr. Deepti Mehrotra **Vetter: Dr. Sandeep Arya**

Lesson: Parallel computer models **Lesson No. : 01**

1.0 Objective

The main aim of this chapter is to learn about the evolution of computer systems, various attributes on which performance of system is measured, classification of computers on their ability to perform multiprocessing and various trends towards parallel processing.

1.1 Introduction

From an application point of view, the mainstream of usage of computer is experiencing a trend of four ascending levels of sophistication:

- Data processing
- Information processing
- Knowledge processing
- Intelligence processing

With more and more data structures developed, many users are shifting to computer roles from pure data processing to information processing. A high degree of parallelism has been found at these levels. As the accumulated knowledge bases expanded rapidly in recent years, there grew a strong demand to use computers for knowledge processing. Intelligence is very difficult to create; its processing even more so. Todays computers are very fast and obedient and have many reliable memory cells to be qualified for data-information-knowledge processing.

Parallel processing is emerging as one of the key technology in area of modern computers. Parallel appears in various forms such as lookahead, vectorization concurrency, simultaneity, data parallelism, interleaving, overlapping, multiplicity, replication, multiprogramming, multithreading and distributed computing at different processing level.

1.2 The state of computing

Modern computers are equipped with powerful hardware technology at the same time loaded with sophisticated software packages. To access the art of computing we firstly review the history of computers then study the attributes used for analysis of performance of computers.

1.2.1 Evolution of computer system

Presently the technology involved in designing of its hardware components of computers and its overall architecture is changing very rapidly for example: processor clock rate increase about 20% a year, its logic capacity improve at about 30% in a year; memory speed at increase about 10% in a year and memory capacity at about 60% increase a year also the disk capacity increase at a 60% a year and so overall cost per bit improves about 25% a year.

But before we go further with design and organization issues of parallel computer architecture it is necessary to understand how computers had evolved. Initially, man used simple mechanical devices – abacus (about 500 BC) , knotted string, and the slide rule for

computation. Early computing was entirely mechanical like : mechanical adder/subtracter (Pascal, 1642) difference engine design (Babbage, 1827) binary mechanical computer (Zuse, 1941) electromechanical decimal machine (Aiken, 1944). Some of these machines used the idea of a stored program a famous example of it is the Jacquard Loom and Babbage's Analytical Engine which is also often considered as the first real computer. Mechanical and electromechanical machines have limited speed and reliability because of the many moving parts. Modern machines use electronics for most information transmission.

Computing is normally thought of as being divided into generations. Each successive generation is marked by sharp changes in hardware and software technologies. With some exceptions, most of the advances introduced in one generation are carried through to later generations. We are currently in the fifth generation.

Ist generation of computers (1945-54)

The first generation computers where based on vacuum tube technology. The first large electronic computer was **ENIAC** (Electronic Numerical Integrator and Calculator), which used high speed vacuum tube technology and were designed primarily to calculate the trajectories of missiles. They used separate memory block for program and data. Later in 1946 John Von Neumann introduced the concept of stored program, in which data and program where stored in same memory block. Based on this concept **EDVAC** (Electronic Discrete Variable Automatic Computer) was built in 1951. On this concept IAS (Institute of advance studies, Princeton) computer was built whose main characteristic was CPU consist of two units (Program flow control and execution unit).

In general key features of this generation of computers where

1) The switching device used where vacuum tube having switching time between 0.1 to 1 milliseconds.

2) One of major concern for computer manufacturer of this era was that each of the computer designs had a unique design. As each computer has unique design one cannot upgrade or replace one component with other computer. Programs that were written for one machine could not execute on another machine, even though other computer was also designed from the same company. This created a major concern for designers as there were no upward-compatible machines or computer architectures with multiple, differing

implementations. And designers always tried to manufacture a new machine that should be upward compatible with the older machines.

3) Concept of specialized registers where introduced for example index registers were introduced in the Ferranti Mark I, concept of register that save the return-address instruction was introduced in UNIVAC I, also concept of immediate operands in IBM 704 and the detection of invalid operations in IBM 650 were introduced.

4) Punch card or paper tape were the devices used at that time for storing the program. By the end of the 1950s IBM 650 became one of popular computers of that time and it used the drum memory on which programs were loaded from punch card or paper tape. Some high-end machines also introduced the concept of core memory which was able to provide higher speeds. Also hard disks started becoming popular.

5) In the early 1950s as said earlier were design specific hence most of them were designed for some particular numerical processing tasks. Even many of them used decimal numbers as their base number system for designing instruction set. In such machine there were actually ten vacuum tubes per digit in each register.

6) Software used was machine level language and assembly language.

7) Mostly designed for scientific calculation and later some systems were developed for simple business systems.

8) Architecture features

Vacuum tubes and relay memories

CPU driven by a program counter (PC) and accumulator

Machines had only fixed-point arithmetic

9) Software and Applications

Machine and assembly language

Single user at a time

No subroutine linkage mechanisms

Programmed I/O required continuous use of CPU

10) examples: ENIAC, Princeton IAS, IBM 701

IInd generation of computers (1954 – 64)

The transistors were invented by Bardeen, Brattain and Shockely in 1947 at Bell Labs and by the 1950s these transistors made an electronic revolution as the transistor is

smaller, cheaper and dissipate less heat as compared to vacuum tube. Now the transistors were used instead of a vacuum tube to construct computers. Another major invention was invention of magnetic cores for storage. These cores where used to large random access memories. These generation computers has better processing speed, larger memory capacity, smaller size as compared to pervious generation computer.

The key features of this generation computers were

1) The IInd generation computer were designed using Germanium transistor, this technology was much more reliable than vacuum tube technology.

2) Use of transistor technology reduced the switching time 1 to 10 microseconds thus provide overall speed up.

2) Magnetic cores were used main memory with capacity of 100 KB. Tapes and disk peripheral memory were used as secondary memory.

3) Introduction to computer concept of instruction sets so that same program can be executed on different systems.

4) High level languages, FORTRAN, COBOL, Algol, BATCH operating system.

5) Computers were now used for extensive business applications, engineering design, optimation using Linear programming, Scientific research

6) Binary number system very used.

7) Technology and Architecture

Discrete transistors and core memories

I/O processors, multiplexed memory access

Floating-point arithmetic available

Register Transfer Language (RTL) developed

8) Software and Applications

High-level languages (HLL): FORTRAN, COBOL, ALGOL with compilers and subroutine libraries

Batch operating system was used although mostly single user at a time

9) Example : CDC 1604, UNIVAC LARC, IBM 7090

IIIrd Generation computers*(1965 to 1974)*

In 1950 and 1960 the discrete components (transistors, registers capacitors) were manufactured packaged in a separate containers. To design a computer these discrete

unit were soldered or wired together on a circuit boards. Another revolution in computer designing came when in the 1960s, the Apollo guidance computer and Minuteman missile were able to develop an integrated circuit (commonly called ICs). These ICs made the circuit designing more economical and practical. The IC based computers are called third generation computers. As integrated circuits, consists of transistors, resistors, capacitors on single chip eliminating wired interconnection, the space required for the computer was greatly reduced. By the mid-1970s, the use of ICs in computers became very common. Price of transistors reduced very greatly. Now it was possible to put all components required for designing a CPU on a single printed circuit board. This advancement of technology resulted in development of minicomputers, usually with 16-bit words size these system have a memory of range of 4k to 64K.This began a new era of microelectronics where it could be possible design small identical chips (a thin wafer of silicon's). Each chip has many gates plus number of input output pins.

Key features of IIIrd Generation computers:

1) The use of silicon based ICs, led to major improvement of computer system. Switching speed of transistor went by a factor of 10 and size was reduced by a factor of 10, reliability increased by a factor of 10, power dissipation reduced by a factor of 10. This cumulative effect of this was the emergence of extremely powerful CPUS with the capacity of carrying out 1 million instruction per second.

2) The size of main memory reached about 4MB by improving the design of magnetic core memories also in hard disk of 100 MB become feasible.

3) On line system become feasible. In particular dynamic production control systems, airline reservation systems, interactive query systems, and real time closed lop process control systems were implemented.

4) Concept of Integrated database management systems were emerged.

5) 32 bit instruction formats

6) Time shared concept of operating system.

7) Technology and Architecture features

Integrated circuits (SSI/MSI)

Microprogramming

Pipelining, cache memories, lookahead processing

8) Software and Applications

Multiprogramming and time-sharing operating systems

Multi-user applications

9) Examples : IBM 360/370, CDC 6600, TI ASC, DEC PDP-82

IVth Generation computer (*(1975 to 1990)*

The microprocessor was invented as a single VLSI (Very large Scale Integrated circuit) chip CPU. Main Memory chips of 1MB plus memory addresses were introduced as single VLSI chip. The caches were invented and placed within the main memory and microprocessor. These VLSIs and VVSLIs greatly reduced the space required in a computer and increased significantly the computational speed.

1) Technology and Architecture feature

LSI/VLSI circuits,

semiconductor memory

Multiprocessors,

vector supercomputers,

multicomputers

Shared or distributed memory

Vector processors

Software and Applications

Multprocessor operating systems,

languages,

compilers,

parallel software tools

Examples : VAX 9000, Cray X-MP, IBM 3090, BBN TC2000

Fifth Generation computers(1990 onwards)

In the mid-to-late 1980s, in order to further improve the performance of the system the designers start using a technique known as “instruction pipelining”. The idea is to break the program into small instructions and the processor works on these instructions in different stages of completion. For example, the processor while calculating the result of the current instruction also retrieves the operands for the next instruction. Based on this concept later superscalar processor were designed, here to execute multiple instructions

in parallel we have multiple execution unit i.e., separate arithmetic-logic units (ALUs). Now instead executing single instruction at a time, the system divide program into several independent instructions and now CPU will look for several similar instructions that are not dependent on each other, and execute them in parallel. The example of this design are VLIW and EPIC.

1) Technology and Architecture features

ULSI/VHSIC processors, memory, and switches

High-density packaging

Scalable architecture

Vector processors

2) Software and Applications

Massively parallel processing

Grand challenge applications

Heterogenous processing

3) Examples : Fujitsu VPP500, Cray MPP, TMC CM-5, Intel Paragon

Elements of Modern Computers

The hardware, software, and programming elements of modern computer systems can be characterized by looking at a variety of factors in context of parallel computing these factors are:

- Computing problems
- Algorithms and data structures
- Hardware resources
- Operating systems
- System software support
- Compiler support

Computing Problems

- Numerical computing complex mathematical formulations tedious integer or floating -point computation
- Transaction processing accurate transactions large database management information retrieval
- Logical Reasoning logic inferences symbolic manipulations

Algorithms and Data Structures

- Traditional algorithms and data structures are designed for sequential machines.
- New, specialized algorithms and data structures are needed to exploit the capabilities of parallel architectures.
- These often require interdisciplinary interactions among theoreticians, experimentalists, and programmers.

Hardware Resources

- The architecture of a system is shaped only partly by the hardware resources.
- The operating system and applications also significantly influence the overall architecture.
- Not only must the processor and memory architectures be considered, but also the architecture of the device interfaces (which often include their advanced processors).

Operating System

- Operating systems manage the allocation and deallocation of resources during user program execution.
- UNIX, Mach, and OSF/1 provide support for multiprocessors and multicomputers
- multithreaded kernel functions virtual memory management file subsystems network communication services
- An OS plays a significant role in mapping hardware resources to algorithmic and data structures.

System Software Support

- Compilers, assemblers, and loaders are traditional tools for developing programs in high-level languages. With the operating system, these tools determine the bind of resources to applications, and the effectiveness of this determines the efficiency of hardware utilization and the system's programmability.
- Most programmers still employ a sequential mind set, abetted by a lack of popular parallel software support.

System Software Support

- Parallel software can be developed using entirely new languages designed specifically with parallel support as its goal, or by using extensions to existing sequential languages.
- New languages have obvious advantages (like new constructs specifically for parallelism), but require additional programmer education and system software.
- The most common approach is to extend an existing language.

Compiler Support

- Preprocessors use existing sequential compilers and specialized libraries to implement parallel constructs
- Precompilers perform some program flow analysis, dependence checking, and limited parallel optimzations
- Parallelizing Compilers requires full detection of parallelism in source code, and transformation of sequential code into parallel constructs
- Compiler directives are often inserted into source code to aid compiler parallelizing efforts

1.2.3 Flynn's Classical Taxonomy

Among mentioned above the one widely used since 1966, is Flynn's Taxonomy. This taxonomy distinguishes multi-processor computer architectures according two independent dimensions of *Instruction stream* and *Data stream*. An instruction stream is sequence of instructions executed by machine. And a data stream is a sequence of data including input, partial or temporary results used by instruction stream. Each of these dimensions can have only one of two possible states: *Single* or *Multiple*. Flynn's classification depends on the distinction between the performance of control unit and the data processing unit rather than its operational and structural interconnections. Following are the four category of Flynn classification and characteristic feature of each of them.

1. Single instruction stream, single data stream (SISD)

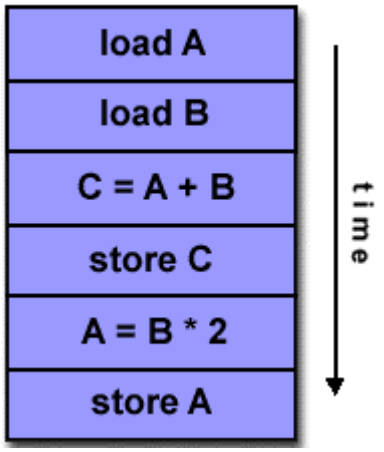


Figure 1.1 Execution of instruction in SISD processors

The figure 1.1 is represents a organization of simple SISD computer having one control unit, one processor unit and single memory unit.

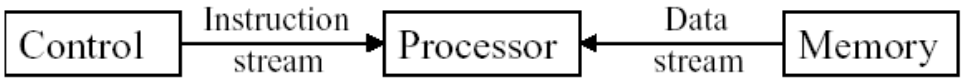


Figure 1.2 SISD processor organization

- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- Instructions are executed sequentially.
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes

b) Single instruction stream, multiple data stream (SIMD) processors

- A type of parallel computer
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle as shown in figure 13.5 where there are multiple processor executing instruction given by one control unit.

- Multiple data: Each processing unit can operate on a different data element as shown if figure below the processor are connected to shared memory or interconnection network providing multiple data to processing unit

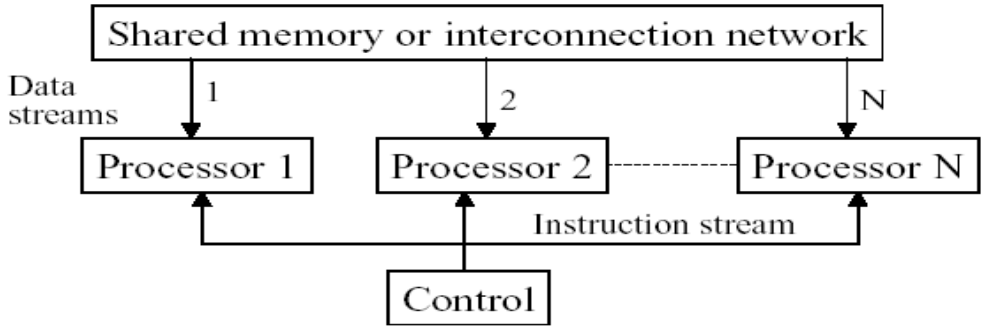


Figure 1.3 SIMD processor organization

- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Thus single instruction is executed by different processing unit on different set of data as shown in figure 1.3.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays e.g., Connection Machine CM-2, Maspar MP-1, MP-2 and Vector Pipelines processor e.g., IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

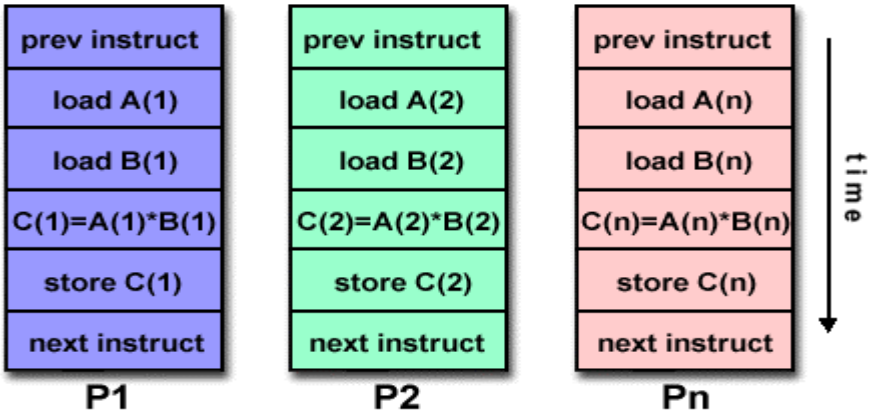


Figure 1.4 Execution of instructions in SIMD processors

c) Multiple instruction stream, single data stream (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams as shown in figure 1.5 a single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.

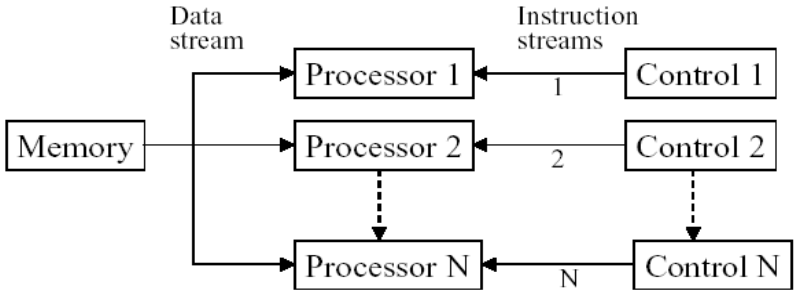


Figure 1.5 MISD processor organization

- Thus in these computers same data flow through a linear array of processors executing different instruction streams as shown in figure 1.6.
- This architecture is also known as systolic arrays for pipelined execution of specific instructions.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:

1. multiple frequency filters operating on a single signal stream
2. multiple cryptography algorithms attempting to crack a single coded message.

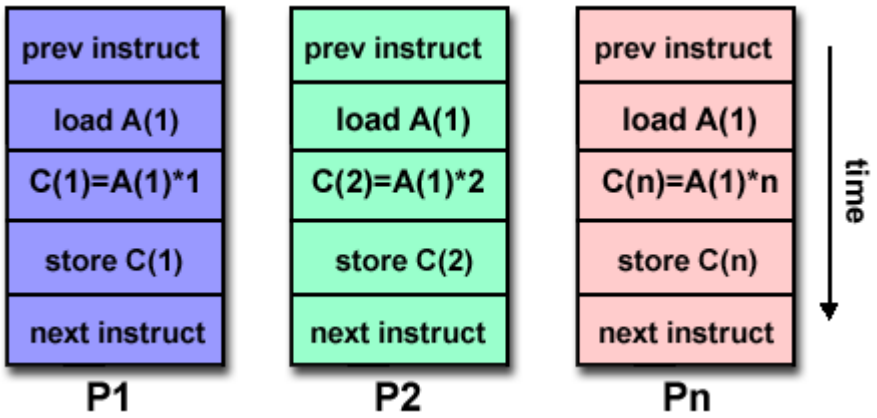


Figure 1.6 Execution of instructions in MISD processors

d) Multiple instruction stream, multiple data stream (MIMD)

- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream as shown in figure 1.7 multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control
- Execution can be synchronous or asynchronous, deterministic or non-deterministic

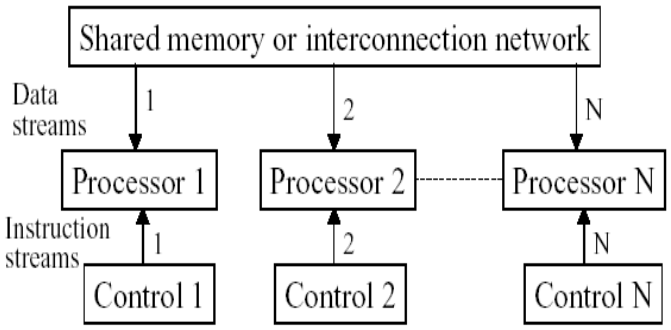


Figure 1.7 MIMD processor organizations

- As shown in figure 1.8 there are different processor each processing different task.
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

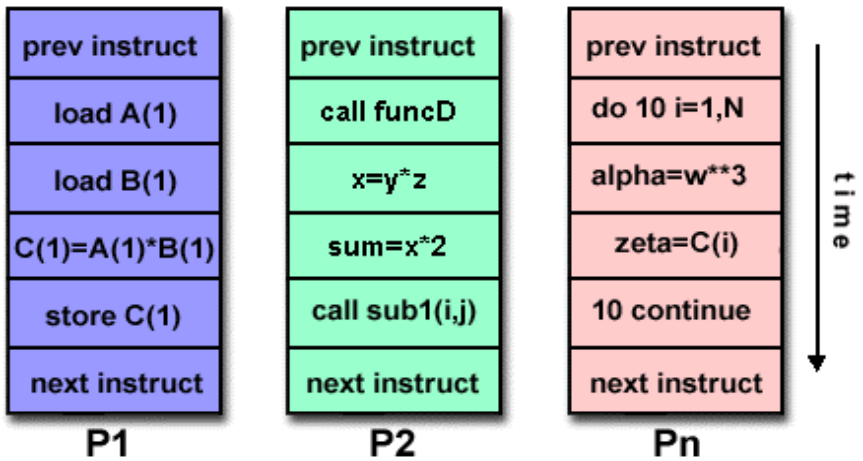


Figure 1.8 execution of instructions MIMD processors

Here the some popular computer architecture and there types

SISD IBM 701, IBM 1620, IBM 7090, PDP VAX11/ 780

SISD (With multiple functional units) IBM360/91 (3); IBM 370/168 UP

SIMD (Word Slice Processing) Illiac – IV ; PEPE

SIMD (Bit Slice processing) STARAN; MPP; DAP

MIMD (Loosely Coupled) IBM 370/168 MP; Univac 1100/80

MIMD(Tightly Coupled) Burroughs- D – 825

1.2.4 PERFORMANCE ATTRIBUTES

Performance of a system depends on

- hardware technology
- architectural features
- efficient resource management
- algorithm design
- data structures
- language efficiency
- programmer skill
- compiler technology

When we talk about performance of computer system we would describe how quickly a given system can execute a program or programs. Thus we are interested in knowing the turnaround time. Turnaround time depends on:

- disk and memory accesses
- input and output
- compilation time
- operating system overhead
- CPU time

An ideal performance of a computer system means a perfect match between the machine capability and program behavior. The machine capability can be improved by using better hardware technology and efficient resource management. But as far as program behavior is concerned it depends on code used, compiler used and other run time conditions. Also a machine performance may vary from program to program. Because there are too many programs and it is impractical to test a CPU's speed on all of them,

benchmarks were developed. Computer architects have come up with a variety of metrics to describe the computer performance.

Clock rate and CPI / IPC : Since I/O and system overhead frequently overlaps processing by other programs, it is fair to consider only the CPU time used by a program, and the user CPU time is the most important factor. CPU is driven by a clock with a constant cycle timeτ (usually measured in nanoseconds, which controls the rate of internal operations in the CPU. The clock mostly has the constant cycle time (t in nanoseconds). The inverse of the cycle time is the clock rate ($f = 1/\tau$, measured in megahertz). A shorter clock cycle time, or equivalently a larger number of cycles per second, implies more operations can be performed per unit time. The size of the program is determined by the instruction count (Ic). The size of a program is determined by its instruction count, I*c*, the number of machine instructions to be executed by the program. Different machine instructions require different numbers of clock cycles to execute. CPI (cycles per instruction) is thus an important parameter.

Average CPI

It is easy to determine the average number of cycles per instruction for a particular processor if we know the frequency of occurrence of each instruction type.

Of course, any estimate is valid only for a specific set of programs (which defines the instruction mix), and then only if there are sufficiently large number of instructions.

In general, the term CPI is used with respect to a particular instruction set and a given program mix. The time required to execute a program containing I*c* instructions is just $T = I_c * CPI * \tau$.

Each instruction must be fetched from memory, decoded, then operands fetched from memory, the instruction executed, and the results stored.

The time required to access memory is called the memory cycle time, which is usually k times the processor cycle time τ. The value of k depends on the memory technology and the processor-memory interconnection scheme. The processor cycles required for each instruction (CPI) can be attributed to cycles needed for instruction decode and execution (*p*), and cycles needed for memory references (*m* k*).

The total time needed to execute a program can then be rewritten as

$T = I_c * (p + m*k)*\tau$.

MIPS: The *millions of instructions per second*, this is calculated by dividing the number of instructions executed in a running program by time required to run the program. The MIPS rate is directly proportional to the clock rate and inversely proportion to the CPI. All four systems attributes (instruction set, compiler, processor, and memory technologies) affect the MIPS rate, which varies also from program to program. MIPS does not proved to be effective as it does not account for the fact that different systems often require different number of instruction to implement the program. It does not inform about how many instructions are required to perform a given task. With the variation in instruction styles, internal organization, and number of processors per system it is almost meaningless for comparing two systems.

MFLOPS (pronounced ``megaflops'') stands for ``millions of floating point operations per second.'' This is often used as a ``bottom-line'' figure. If one know ahead of time how many operations a program needs to perform, one can divide the number of operations by the execution time to come up with a MFLOPS rating. For example, the standard algorithm for multiplying **n*n** matrices requires $\mathbf{2n^3 - n}$ operations (n^2 inner products, with **n** multiplications and **n-1**additions in each product). Suppose you compute the product of two 100 *100 matrices in 0.35 seconds. Then the computer achieves

$(2(100)^3 - 100)/0.35 = 5{,}714{,}000$ ops/sec = 5.714 MFLOPS

The term ``theoretical peak MFLOPS'' refers to how many operations per second would be possible if the machine did nothing but numerical operations. It is obtained by calculating the time it takes to perform one operation and then computing how many of them could be done in one second. For example, if it takes 8 cycles to do one floating point multiplication, the cycle time on the machine is 20 nanoseconds, and arithmetic operations are not overlapped with one another, it takes 160ns for one multiplication, and (1,000,000,000 nanosecond/1sec)*(1 multiplication / 160 nanosecond) = $6.25*10^6$ multiplication /sec so the theoretical peak performance is 6.25 MFLOPS. Of course, programs are not just long sequences of multiply and add instructions, so a machine rarely comes close to this level of performance on any real program. Most machines will achieve less than 10% of their peak rating, but vector processors or other machines with internal pipelines that have an effective CPI near 1.0 can often achieve 70% or more of their theoretical peak on small programs.

Throughput rate : Another important factor on which system's performance is measured is throughput of the system which is basically how many programs a system can execute per unit time Ws. In multiprogramming the system throughput is often lower than the CPU throughput Wp which is defined as

Wp = f/(Ic * CPI)

Unit of Wp is programs/second.

Ws <Wp as in multiprogramming environment there is always additional overheads like timesharing operating system etc. An Ideal behavior is not achieved in parallel computers because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. Efficiency is a measure of the fraction of time for which a PE is usefully employed. In an ideal parallel system efficiency is equal to one. In practice, efficiency is between zero and one

s of overhead associated with parallel execution

Speed or Throughput (W/Tn) - the execution rate on an n processor system, measured in FLOPs/unit-time or instructions/unit-time.

Speedup (Sn = T1/Tn) - how much faster in an actual machine, n processors compared to 1 will perform the workload. The ratio T1/T∞is called the *asymptotic speedup*.

Efficiency (En = Sn/n) - fraction of the theoretical maximum speedup achieved by n processors

Degree of Parallelism (DOP) - for a given piece of the workload, the number of processors that can be kept busy sharing that piece of computation equally. Neglecting overhead, we assume that if k processors work together on any workload, the workload gets done k times as fast as a sequential execution.

Scalability - The attributes of a computer system which allow it to be gracefully and linearly scaled up or down in size, to handle smaller or larger workloads, or to obtain proportional decreases or increase in speed on a given application. The applications run on a scalable machine may not scale well. Good scalability requires the algorithm *and* the machine to have the right properties

Thus in general there are five performance factors (I*c*, p, m, k, t) which are influenced by four system attributes:

- instruction-set architecture (affects *Ic* and *p*)

- compiler technology (affects *Ic* and *p* and *m*)
- CPU implementation and control (affects *p *t*.) cache and memory hierarchy (affects memory access latency, *k ´t*)
- Total CPU time can be used as a basis in estimating the execution rate of a processor.

Programming Environments

Programmability depends on the programming environment provided to the users.

Conventional computers are used in a sequential programming environment with tools developed for a uniprocessor computer. Parallel computers need parallel tools that allow specification or easy detection of parallelism and operating systems that can perform parallel scheduling of concurrent events, shared memory allocation, and shared peripheral and communication links.

Implicit Parallelism

Use a conventional language (like C, Fortran, Lisp, or Pascal) to write the program.

Use a parallelizing compiler to translate the source code into parallel code.

The compiler must detect parallelism and assign target machine resources.

Success relies heavily on the quality of the compiler.

Explicit Parallelism

Programmer writes explicit parallel code using parallel dialects of common languages.

Compiler has reduced need to detect parallelism, but must still preserve existing parallelism and assign target machine resources.

Needed Software Tools

Parallel extensions of conventional high-level languages.

Integrated environments to provide different levels of program abstraction validation, testing and debugging performance prediction and monitoring visualization support to aid program development, performance measurement graphics display and animation of computational results

1.3 MULTIPROCESSOR AND MULTICOMPUTERS

Two categories of parallel computers are discussed below namely shared common memory or unshared distributed memory.

1.3.1 Shared memory multiprocessors

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** , ***NUMA and COMA***.

Uniform Memory Access (UMA):

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

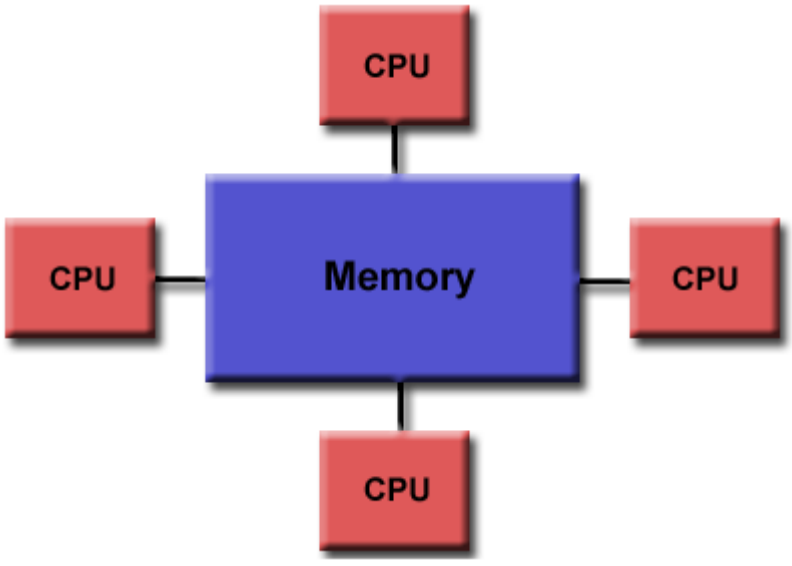


Figure 1.9 Shared Memory (UMA)

Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower

If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

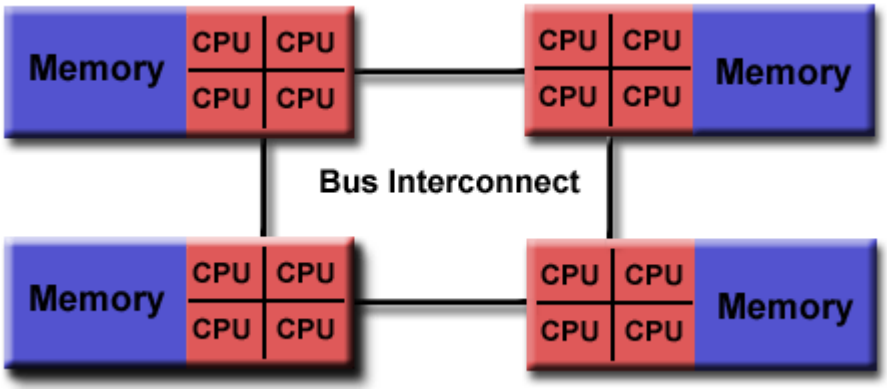


figure 1.10 **Shared Memory (NUMA)**

The COMA model : The COMA model is a special case of NUMA machine in which the distributed main memories are converted to caches. All caches form a global address space and there is no memory hierarchy at each processor node.

Advantages:

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:

- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-

CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.

- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

1.3.2 Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.

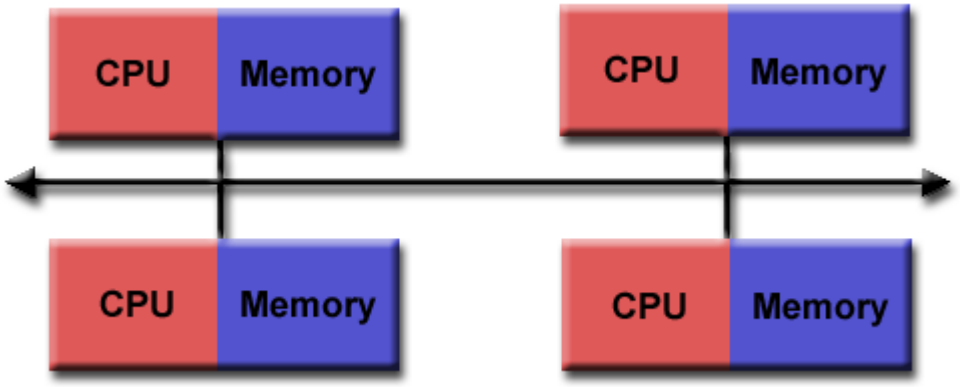


Figure 1.11 distributed memory systems

- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.

- Modern multicomputer use hardware routers to pass message. Based on the interconnection and routers and channel used the multicomputers are divided into generation
    - 1st generation : based on board technology using hypercube architecture and software controlled message switching.
    - 2nd Generation: implemented with mesh connected architecture, hardware message routing and software environment for medium distributed – grained computing.
    - 3rd Generation : fine grained multicomputer like MIT J-Machine.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

Advantages:

- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access (NUMA) times

1.4 MULTIVECTOR AND SIMD COMPUTERS

A vector operand contains an ordered set of n elements, where n is called the length of the vector. Each element in a vector is a scalar quantity, which may be a floating point number, an integer, a logical value or a character.

A vector processor consists of a scalar processor and a vector unit, which could be thought of as an independent functional unit capable of efficient vector operations.

1.4.1Vector Hardware

Vector computers have hardware to perform the vector operations efficiently. Operands can not be used directly from memory but rather are loaded into registers and are put back in registers after the operation. Vector hardware has the special ability to overlap or pipeline operand processing.

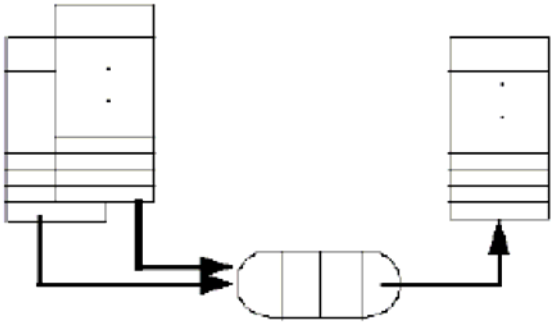


Figure 1.12 Vector Hardware

Vector functional units pipelined, fully segmented each stage of the pipeline performs a step of the function on different operand(s) once pipeline is full, a new result is produced each clock period (cp).

Pipelining

The pipeline is divided up into individual segments, each of which is completely independent and involves no hardware sharing. This means that the machine can be working on separate operands at the same time. This ability enables it to produce one result per clock period as soon as the pipeline is full. The same instruction is obeyed repeatedly using the pipeline technique so the vector processor processes all the elements of a vector in exactly the same way. The pipeline segments arithmetic operation such as floating point multiply into stages passing the output of one stage to the next stage as input. The next pair of operands may enter the pipeline after the first stage has processed the previous pair of operands. The processing of a number of operands may be carried out simultaneously.

The loading of a vector register is itself a pipelined operation, with the ability to load one element each clock period after some initial startup overhead.

1.4.2 SIMD Array Processors

The Synchronous parallel architectures coordinate Concurrent operations in lockstep through global clocks, central control units, or vector unit controllers. A synchronous array of parallel processors is called an array processor. These processors are composed of N identical processing elements (PES) under the supervision of a one control unit (CU) This Control unit is a computer with high speed registers,

local memory and arithmetic logic unit.. An array processor is basically a single instruction and multiple data (SIMD) computers. There are N data streams; one per processor, so different data can be used in each processor. The figure below show a typical SIMD or array processor

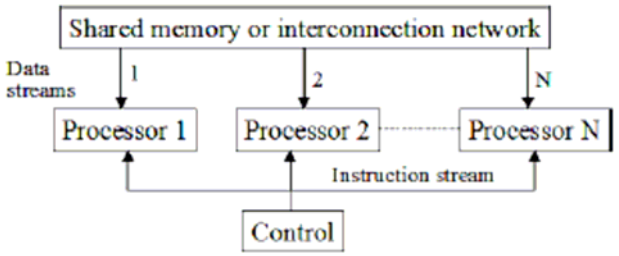


Figure 1.13 Configuration of SIMD Array Processor

These processors consist of a number of memory modules which can be either global or dedicated to each processor. Thus the main memory is the aggregate of the memory modules. These Processing elements and memory unit communicate with each other through an interconnection network. SIMD processors are especially designed for performing vector computations. SIMD has two basic architectural organizations

a. Array processor using random access memory

b. Associative processors using content addressable memory.

All N identical processors operate under the control of a single instruction stream issued by a central control unit. The popular examples of this type of SIMD configuration is ILLIAC IV, CM-2, MP-1. Each PEi is essentially an arithmetic logic unit (ALU) with attached working registers and local memory PEMi for the storage of distributed data. The CU also has its own main memory for the storage of program. The function of CU is to decode the instructions and determine where the decoded instruction should be executed. The PE perform same function (same instruction) *synchronously in a lock step fashion under command of CU. In order to maintain synchronous operations* a global

clock is used. Thus at each step i.e., when global clock pulse changes all processors execute the same instruction, each on a different data (single instruction multiple data). SIMD machines are particularly useful at in solving problems involved with vector calculations where one can easily exploit data parallelism. In such calculations the same set of instruction is applied to all subsets of data. Lets do addition to two vectors each having N element and there are N/2 processing elements in the SIMD. The same addition instruction is issued to all N/2 processors and all processor elements will execute the instructions simultaneously. It takes 2 steps to add two vectors as compared to N steps on a SISD machine. The distributed data can be loaded into PEMs from an external source via the system bus or via system broadcast mode using the control bus.

The array processor can be classified into two category depending how the memory units are organized. It can be

a. Dedicated memory organization
b. Global memory organization

A SIMD computer C is characterized by the following set of parameter

C= <N,F,I,M>

Where N= the number of PE in the system . For example the iliac –IV has N=64 , the BSP has N= 16.

F= a set of data routing function provided by the interconnection network

I= The set of machine instruction for scalar vector, data routing and network manipulation operations

M = The set of the masking scheme where each mask partitions the set of PEs into disjoint subsets of enabled PEs and disabled PEs.

1.5 PRAM AND VLSI MODELS

1.5.1 PRAM model (Parallel Random Access Machine):

PRAM Parallel random access machine; a theoretical model of parallel computation in which an arbitrary but finite number of processors can access any value in an arbitrarily large *shared memory* in a single time step. Processors may execute different instruction streams, but work *synchronously*. This model assumes a shared memory, multiprocessor machine as shown:

1. The machine size n can be arbitrarily large

2. The machine is synchronous at the instruction level. That is, each processor is executing it's own series of instructions, and the entire machine operates at a basic time step (cycle). Within each cycle, each processor executes exactly one operation or does nothing, i.e. it is *idle.* An instruction can be any random access machine instruction, such as: fetch some operands from memory, perform an ALU operation on the data, and store the result back in memory.

3. All processors implicitly synchronize on each cycle and the synchronization overhead is assumed to be zero. Communication is done through reading and writing of shared variables.

4. Memory access can be specified to be UMA, NUMA, EREW, CREW, or CRCW with a defined conflict policy.

The PRAM model can apply to SIMD class machines if all processors execute identical instructions on the same cycle, or to MIMD class machines if the processors are executing different instructions. Load imbalance is the only form of overhead in the PRAM model.

The four most important variations of the PRAM are:

- **EREW** - Exclusive read, exclusive write; any memory location may only be accessed once in any one step. Thus forbids more than one processor from reading or writing the same memory cell simultaneously.
- **CREW** - Concurrent read, exclusive write; any memory location may be read any number of times during a single step, but only written to once, with the write taking place after the reads.
- **ERCW** – This allows exclusive read or concurrent writes to the same memory location.
- **CRCW** - Concurrent read, concurrent write; any memory location may be written to or read from any number of times during a single step. A CRCW PRAM model must define some rule for resolving multiple writes, such as giving priority to the lowest-numbered processor or choosing amongst processors randomly. The PRAM is popular because it is theoretically tractable and because it gives

algorithm designers a common target. However, PRAMs cannot be emulated *optimally* on all *architectures*.

1.5.2 VLSI Model:

Parallel computers rely on the use of VLSI chips to fabricate the major components such as processor arrays memory arrays and large scale switching networks. The rapid advent of very large scale intergrated (VSLI) technology now computer architects are trying to implement parallel algorithms directly in hardware. An AT^2 model is an example for two dimension VLSI chips

1.6 Summary

Architecture has gone through evolutional, rather than revolutional change.

Sustaining features are those that are proven to improve performance. Starting with the von Neumann architecture (strictly sequential), architectures have evolved to include processing lookahead, parallelism, and pipelining. Also a variety of parallel architectures are discussed like SIMD, MIMD, Associative Processor, Array Processor, multicomputers, Mutiprocessor. The performance of system is measured as CPI, MIPS. It depends on the clock rate lets say t. If C is the total number of clock cycles needed to execute a given program, then total CPU time can be estimated as

$T = C * t = C / f$.

Other relationships are easily observed:

$CPI = C / Ic$

$T = Ic * CPI * t$

$T = Ic * CPI / f$

Processor speed is often measured in terms of millions of instructions per second, frequently called the MIPS rate of the processor. The multiprocessor architecture can be broadly classified as tightly coupled multiprocessor and loosely coupled multiprocessor. A tightly coupled Multiprocessor is also called a UMA, for uniform memory access, because each CPU can access memory data at the same (uniform) amount of time. This is the true multiprocessor. A loosely coupled Multiprocessor is called a NUMA. Each of its node computers can access their local memory data at one (relatively fast) speed, and

remote memory data at a much slower speed. PRAM and VSLI are the advance technologies that are used for designing the architecture.

1.7 Keywords

multiprocessor A computer in which processors can execute separate instruction streams, but have access to a single *address space*. Most multiprocessors are *shared memory* machines, constructed by connecting several processors to one or more memory banks through a *bus* or *switch*.

multicomputer A computer in which processors can execute separate instruction streams, have their own private memories and cannot directly access one another's memories. Most multicomputers are *disjoint memory* machines, constructed by joining *nodes* (each containing a microprocessor and some memory) via *links*.

M**IMD** Multiple Instruction, Multiple Data; a category of *Flynn's taxonomy* in which many instruction streams are concurrently applied to multiple data sets. A MIMD *architecture* is one in which *heterogeneous* processes may execute at different rates.

MIPS one Million Instructions Per Second. A performance rating usually referring to integer or non-floating point instructions

vector processor A computer designed to apply arithmetic operations to long vectors or arrays. Most vector processors rely heavily on *pipelining* to achieve high performance

pipelining Overlapping the execution of two or more operations

1.8 Self assessment questions

1. Explain the different generations of computers with respect to progress in hardware.
2. Describe the different element of modern computers.
3. Explain the Flynn’s classification of computer architectures.
4. Explain performance factors vs system attributes.
5. A 40-MHZ processor was used to execute a benchmark program with the following instruction mix and clock cycle counts:

Instruction type	Instruction count	Clock cycle count
Integer arithmetic	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2

Determine the effective CPI, MIPS rate, and execution time for this program. (10 marks)

6. Explain the UMA, NUMA and COMA multiprocessor models.
7. Differentiate between shared-memory multiprocessors and distributed-memory multicomputers.

1.9 References/Suggested readings

Advance computer Architecture by Kai Hwang

Author: Dr. Deepti Mehrotra **Vetter: Dr. Sandeep Arya**

Lesson: Program & Network properties **Lesson No. : 02**

2.0 Objective

In this lesson we will study about fundamental properties of programs how parallelism can be introduced in program. We will study about the granularity, partitioning of programs , program flow mechanism and compilation support for parallelism. Interconnection architecture both static and dynamic type will be discussed.

2.1 Introduction

The advantage of multiprocessors lays when parallelism in the program is popularly exploited and implemented using multiple processors. Thus in order to implement the parallelism we should understand the various conditions of parallelism.

What are various bottlenecks in implementing parallelism? Thus for full implementation of parallelism there are three significant areas to be understood namely computation models for parallel computing, interprocessor communication in parallel architecture and system integration for incorporating parallel systems. Thus multiprocessor system poses a number of problems that are not encountered in sequential processing such as designing a parallel algorithm for the application, partitioning of the application into tasks, coordinating communication and synchronization, and scheduling of the tasks onto the machine.

2.2 Condition of parallelism

The ability to execute several program segments in parallel requires each segment to be independent of the other segments. We use a dependence graph to describe the relations. The nodes of a dependence graph correspond to the program statement (instructions), and directed edges with different labels are used to represent the ordered relations among the statements. The analysis of dependence graphs shows where opportunity exists for parallelization and vectorization.

2.2.1 Data and resource Dependence

Data dependence: The ordering relationship between statements is indicated by the data dependence. Five type of data dependence are defined below:

1. Flow dependence: A statement S2 is flow dependent on S1 if an execution path exists from s1 to S2 and if at least one output (variables assigned) of S1feeds in as input (operands to be used) to S2 also called RAW hazard and denoted as $S_1 \rightarrow S_2$

2. Antidependence: Statement S2 is antidependent on the statement S1 if S2 follows S1 in the program order and if the output of S2 overlaps the input to S1 also called RAW hazard and denoted as $S_1 \mapsto S_2$

3. Output dependence : two statements are output dependent if they produce (write) the same output variable. Also called WAW hazard and denoted as $S_1 \multimap S_2$

4. I/O dependence: Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file referenced by both I/O statement.

5. Unknown dependence: The dependence relation between two statements cannot be determined in the following situations:

- The subscript of a variable is itself subscribed(indirect addressing)
- The subscript does not contain the loop index variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is non linear in the loop index variable.

Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

Consider the following fragment of any program:

S1 Load R1, A

S2 Add R2, R1

S3 Move R1, R3

S4 Store B, R1

• here the Forward dependency S1to S2, S3 to S4, S2 to S2

• Anti-dependency from S2to S3

• Output dependency S1 toS3

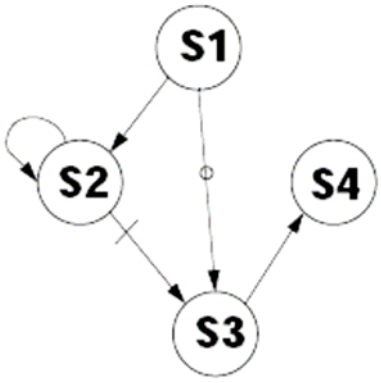


Figure 2.1 Dependence graph

Control Dependence: This refers to the situation where the order of the execution of statements cannot be determined before run time. For example all condition statement, where the flow of statement depends on the output. Different paths taken after a conditional branch may depend on the data hence we need to eliminate this data dependence among the instructions. This dependence also exists between operations

performed in successive iterations of looping procedure. Control dependence often prohibits parallelism from being exploited.

Control-independent example:

```
for (i=0;i<n;i++) {
a[i] = c[i];
if (a[i] < 0) a[i] = 1;
}
```

Control-dependent example:

```
for (i=1;i<n;i++) {
if (a[i-1] < 0) a[i] = 1;
}
```

Control dependence also avoids parallelism to being exploited. Compilers are used to eliminate this control dependence and exploit the parallelism.

Resource dependence:

Data and control dependencies are based on the independence of the work to be done.

Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc. ALU conflicts are called ALU dependence. Memory (storage) conflicts are called storage dependence.

Bernstein's Conditions - 1

Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.

Notation

Ii is the set of all input variables for a process *Pi* . *Ii* is also called the read set or domain of *Pi*. *Oi* is the set of all output variables for a process *Pi* .Oi is also called write set

If P*1* and P*2* can execute in parallel (which is written as P*1* || P*2*), then:

$$\mathbf{I}_1 \cap \mathbf{O}_2 = \varnothing$$

$$\mathbf{I}_2 \cap \mathbf{O}_1 = \varnothing$$

$$\mathbf{O}_1 \cap \mathbf{O}_2 = \varnothing$$

Bernstein's Conditions - 2

In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, antiindependent, and output-independent. The parallelism relation || is commutative (P*i* || P*j* implies P*j* || P*i*), but not transitive (P*i* || P*j* and P*j* || P*k* does not imply P*i* || P*k*) . Therefore, || is not an equivalence relation. Intersection of the input sets is allowed.

2.2.2 Hardware and software parallelism

Hardware parallelism is defined by machine architecture and hardware multiplicity i.e., functional parallelism times the processor parallelism .It can be characterized by the number of instructions that can be issued per machine cycle. If a processor issues *k* instructions per machine cycle, it is called a *k-issue* processor. Conventional processors are *one-issue* machines. This provide the user the information about **peak attainable performance**. Examples. Intel i960CA is a three-issue processor (arithmetic, memory access, branch). IBM RS -6000 is a four-issue processor (arithmetic, floating-point, memory access, branch).A machine with *n k*-issue processors should be able to handle a maximum of *nk* threads simultaneously.

Software Parallelism

Software parallelism is defined by the control and data dependence of programs, and is revealed in the program's flow graph i.e., it is defined by dependencies with in the code and is a function of algorithm, programming style, and compiler optimization.

2.2.3 The Role of Compilers

Compilers used to exploit hardware features to improve performance. Interaction between compiler and architecture design is a necessity in modern computer development. It is not necessarily the case that more software parallelism will improve performance in conventional scalar processors. The hardware and compiler should be designed at the same time.

2.3Program Partitioning & Scheduling

2.3.1 Grain size and latency

The size of the parts or pieces of a program that can be considered for parallel execution can vary. The sizes are roughly classified using the term "granule size," or simply "granularity." The simplest measure, for example, is the number of instructions in a

program part. Grain sizes are usually described as fine, medium or coarse, depending on the level of parallelism involved.

Latency

Latency is the time required for communication between different subsystems in a computer. Memory latency, for example, is the time required by a processor to access memory. Synchronization latency is the time required for two processes to synchronize their execution. Computational granularity and communication latency are closely related. Latency and grain size are interrelated and some general observation are

- As grain size decreases, potential parallelism increases, and overhead also increases.
- Overhead is the cost of parallelizing a task. The principle overhead is communication latency.
- As grain size is reduced, there are fewer operations between communication, and hence the impact of latency increases.
- Surface to volume: inter to intra-node comm.

Levels of Parallelism

Instruction Level Parallelism

This fine-grained, or smallest granularity level typically involves less than 20 instructions per grain. The number of candidates for parallel execution varies from 2 to thousands, with about five instructions or statements (on the average) being the average level of parallelism.

Advantages:

There are usually many candidates for parallel execution

Compilers can usually do a reasonable job of finding this parallelism

Loop-level Parallelism

Typical loop has less than 500 instructions. If a loop operation is independent between iterations, it can be handled by a pipeline, or by a SIMD machine. Most optimized program construct to execute on a parallel or vector machine. Some loops (e.g. recursive) are difficult to handle. Loop-level parallelism is still considered fine grain computation.

Procedure-level Parallelism

Medium-sized grain; usually less than 2000 instructions. Detection of parallelism is more difficult than with smaller grains; interprocedural dependence analysis is difficult and history-sensitive. Communication requirement less than instruction level SPMD (single procedure multiple data) is a special case Multitasking belongs to this level.

Subprogram-level Parallelism

Job step level; grain typically has thousands of instructions; medium- or coarse-grain level. Job steps can overlap across different jobs. Multiprograming conducted at this level No compilers available to exploit medium- or coarse-grain parallelism at present.

Job or Program-Level Parallelism

Corresponds to execution of essentially independent jobs or programs on a parallel computer. This is practical for a machine with a small number of powerful processors, but impractical for a machine with a large number of simple processors (since each processor would take too long to process a single job).

Communication Latency

Balancing granularity and latency can yield better performance. Various latencies attributed to machine architecture, technology, and communication patterns used. Latency imposes a limiting factor on machine scalability. Ex. Memory latency increases as memory capacity increases, limiting the amount of memory that can be used with a given tolerance for communication latency.

Interprocessor Communication Latency

- Needs to be minimized by system designer
- Affected by signal delays and communication patterns Ex. n communicating tasks may require n (n - 1)/2 communication links, and the complexity grows quadratically, effectively limiting the number of processors in the system.

Communication Patterns

- Determined by algorithms used and architectural support provided
- Patterns include permutations broadcast multicast conference
- Tradeoffs often exist between granularity of parallelism and communication demand.

2.3.2 Grain Packing and Scheduling

Two questions:

How can I partition a program into parallel “pieces” to yield the shortest execution time?

What is the optimal size of parallel grains?

There is an obvious tradeoff between the time spent scheduling and synchronizing parallel grains and the speedup obtained by parallel execution.

One approach to the problem is called “grain packing.”

Program Graphs and Packing

A program graph is similar to a dependence graph Nodes = { (n,s) }, where n = node name, s = size (larger s = larger grain size).

Edges = { (v,d) }, where v = variable being “communicated,” and d = communication delay.

Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes. Packing is done to eliminate unnecessary communication delays or reduce overall scheduling overhead.

Scheduling

A schedule is a mapping of nodes to processors and start times such that communication delay requirements are observed, and no two nodes are executing on the same processor at the same time. Some general scheduling goals

- Schedule all fine-grain activities in a node to the same processor to minimize communication delays.
- Select grain sizes for packing to achieve better schedules for a particular parallel machine.

Node Duplication

Grain packing may potentially eliminate interprocessor communication, but it may not always produce a shorter schedule. By duplicating nodes (that is, executing some instructions on multiple processors), we may eliminate some interprocessor communication, and thus produce a shorter schedule.

Program partitioning and scheduling

Scheduling and allocation is a highly important issue since an inappropriate scheduling of tasks can fail to exploit the true potential of the system and can offset the gain from parallelization. In this paper we focus on the scheduling aspect. The objective of scheduling is to minimize the completion time of a parallel application by properly

allocating the tasks to the processors. In a broad sense, the scheduling problem exists in two forms: *static* and *dynamic*. In static scheduling, which is usually done at compile time, the characteristics of a parallel program (such as task processing times, communication, data dependencies, and synchronization requirements) are known before program execution

A parallel program, therefore, can be represented by a node- and edge-weighted directed acyclic graph (DAG), in which the node weights represent task processing times and the edge weights represent data dependencies as well as the communication times between tasks. In dynamic scheduling only, a few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be made on-the-fly. The goal of a dynamic scheduling algorithm as such includes not only the minimization of the program completion time but also the minimization of the scheduling overhead which constitutes a significant portion of the cost paid for running the scheduler. In general dynamic scheduling is an NP hard problem.

2.4 Program flow mechanism

Conventional machines used control flow mechanism in which order of program execution explicitly stated in user programs. Dataflow machines which instructions can be executed by determining operand availability.

Reduction machines trigger an instruction’s execution based on the demand for its results.

Control Flow vs. Data Flow In Control flow computers the next instruction is executed when the last instruction as stored in the program has been executed where as in Data flow computers an instruction executed when the data (operands) required for executing that instruction is available

Control flow machines used shared memory for instructions and data. Since variables are updated by many instructions, there may be side effects on other instructions. These side effects frequently prevent parallel processing. Single processor systems are inherently sequential.

Instructions in dataflow machines are unordered and can be executed as soon as their operands are available; data is held in the instructions themselves. *Data tokens* are passed from an instruction to its dependents to trigger execution.

Data Flow Features

No need for shared memory program counter control sequencer Special mechanisms are required to detect data availability match data tokens with instructions needing them enable chain reaction of asynchronous instruction execution

A Dataflow Architecture – 1 The Arvind machine (MIT) has N PEs and an N -by –N interconnection network. Each PE has a token-matching mechanism that dispatches only instructions with data tokens available. Each datum is tagged with

- address of instruction to which it belongs
- context in which the instruction is being executed

Tagged tokens enter PE through local path (pipelined), and can also be communicated to other PEs through the routing network. Instruction address(es) effectively replace the program counter in a control flow machine. Context identifier effectively replaces the frame base register in a control flow machine. Since the dataflow machine matches the data tags from one instruction with successors, synchronized instruction execution is implicit.

An I-structure in each PE is provided to eliminate excessive copying of data structures. Each word of the I-structure has a two-bit tag indicating whether the value is empty, full, or has pending read requests.

This is a retreat from the pure dataflow approach. Special compiler technology needed for dataflow machines.

Demand-Driven Mechanisms

Data-driven machines select instructions for execution based on the availability of their operands; this is essentially a bottom-up approach.

Demand-driven machines take a top-down approach, attempting to execute the instruction (a *demander*) that yields the final result. This triggers the execution of instructions that yield its operands, and so forth. The demand-driven approach matches naturally with functional programming languages (e.g. LISP and SCHEME).

Pattern driven computers : An instruction is executed when we obtain a particular data patterns as output. There are two types of pattern driven computers

String-reduction model: each demander gets a separate copy of the expression string to evaluate each reduction step has an operator and embedded reference to demand the corresponding operands each operator is suspended while arguments are evaluated

Graph-reduction model: expression graph reduced by evaluation of branches or subgraphs, possibly in parallel, with demanders given pointers to results of reductions. based on sharing of pointers to arguments; traversal and reversal of pointers continues until constant arguments are encountered.

2.5 System interconnect architecture.

Various types of interconnection networks have been suggested for SIMD computers. These are basically classified have been classified on network topologies into two categories namely

Static Networks

Dynamic Networks

Static versus Dynamic Networks

The topological structure of an SIMD array processor is mainly characterized by the data routing network used in interconnecting the processing elements.

The topological structure of an SIMD array processor is mainly characterized by the data routing network used in the interconnecting the processing elements. To execute the communication the routing function f is executed and via the interconnection network the PEi copies the content of its Ri register into the Rf(i) register of PEf(i). The f(i) the processor identified by the mapping function f. The data routing operation occurs in all active PEs simultaneously.

2.5.1 Network properties and routing

The goals of an interconnection network are to provide low-latency high data transfer rate wide communication bandwidth. Analysis includes latency bisection bandwidth data-routing functions scalability of parallel architecture

These Network usually represented by a graph with a finite number of nodes linked by directed or undirected edges.

Number of nodes in graph = network size .

Number of edges (links or channels) incident on a node = node degree d (also note in and out degrees when edges are directed).

Node degree reflects number of I/O ports associated with a node, and should ideally be small and constant.

Network is symmetric if the topology is the same looking from any node; these are easier to implement or to program.

Diameter : The maximum distance between any two processors in the network or in other words we can say **Diameter**, is the maximum number of (routing) processors through which a message must pass on its way from source to reach destination. Thus diameter measures the maximum delay for transmitting a message from one processor to another as it determines communication time hence smaller the diameter better will be the network topology.

Connectivity: How many paths are possible between any two processors i.e., the multiplicity of paths between two processors. Higher connectivity is desirable as it minimizes contention.

Arch connectivity of the network: the minimum number of arcs that must be removed for the network to break it into two disconnected networks. The arch connectivity of various network are as follows

- 1 for linear arrays and binary trees
- 2 for rings and 2-d meshes
- 4 for 2-d torus
- d for d-dimensional hypercubes

Larger the arch connectivity lesser the conjunctions and better will be network topology.

Channel width : The channel width is the number of bits that can communicated simultaneously by a interconnection bus connecting two processors

Bisection Width and Bandwidth: In order divide the network into equal halves we require the remove some communication links. The minimum number of such communication links that have to be removed are called the Bisection Width. **Bisection width basically provide us the information about** the largest number of messages which can be sent simultaneously (without needing to use the same wire or routing processor at the same time and so delaying one another), no matter which processors are sending to which other processors. Thus larger the bisection width is the better the network topology is considered. Bisection Bandwidth is the minimum volume of communication allowed

between two halves of the network with equal numbers of processors This is important for the networks with weighted arcs where the weights correspond to the *link width i.e.*, (how much data it can transfer). The Larger bisection width the better network topology is considered.

Cost the cost of networking can be estimated on variety of criteria where we consider the the number of communication links or wires used to design the network as the basis of cost estimation. Smaller the better the cost

Data Routing Functions: A data routing network is used for inter –PE data exchange. It can be static as in case of hypercube routing network or dynamic such as multistage network. Various type of data routing functions are Shifting, Rotating, Permutation (one to one), Broadcast (one to all), Multicast (many to many), Personalized broadcast (one to many), Shuffle, Exchange Etc.

Permutations

Given n objects, there are n ! ways in which they can be reordered (one of which is no reordering). A permutation can be specified by giving the rule for reordering a group of objects. Permutations can be implemented using crossbar switches, multistage networks, shifting, and broadcast operations. The time required to perform permutations of the connections between nodes often dominates the network performance when n is large.

Perfect Shuffle and Exchange

Stone suggested the special permutation that entries according to the mapping of the k-bit binary number a b … k to b c … k a (that is, shifting 1 bit to the left and wrapping it around to the least significant bit position). The inverse perfect shuffle reverses the effect of the perfect shuffle.

Hypercube Routing Functions

If the vertices of a n-dimensional cube are labeled with n-bit numbers so that only one bit differs between each pair of adjacent vertices, then n routing functions are defined by the bits in the node (vertex) address. For example, with a 3-dimensional cube, we can easily identify routing functions that exchange data between nodes with addresses that differ in the least significant, most significant, or middle bit.

Factors Affecting Performance

Functionality – how the network supports data routing, interrupt handling, synchronization, request/message combining, and coherence

Network latency – worst-case time for a unit message to be transferred

Bandwidth – maximum data rate

Hardware complexity – implementation costs for wire, logic, switches, connectors, etc.

Scalability – how easily does the scheme adapt to an increasing number of processors, memories, etc.?

2.5.2 Static connection Networks

In static network the interconnection network is fixed and permanent interconnection path between two processing elements and data communication has to follow a fixed route to reach the destination processing element. Thus it Consist of a number of point-to-point links. Topologies in the static networks can be classified according to the dimension required for layout i.e., it can be 1-D, 2-D, 3-D or hypercube.

One dimensional topologies include Linear array as shown in figure 2.2 (a) used in some pipeline architecture.

Various 2-D topologies are

- The ring (figure 2.2(b))
- Star (figure 2.2(c))
- Tree (figure 2.2(d))
- Mesh (figure 2.2(e))
- Systolic Array (figure 2.2(f))

3-D topologies include

- Completely connected chordal ring (figure 2.2(g))
- Chordal ring (figure 2.2(h))
- 3 cube (figure 2.2(i))

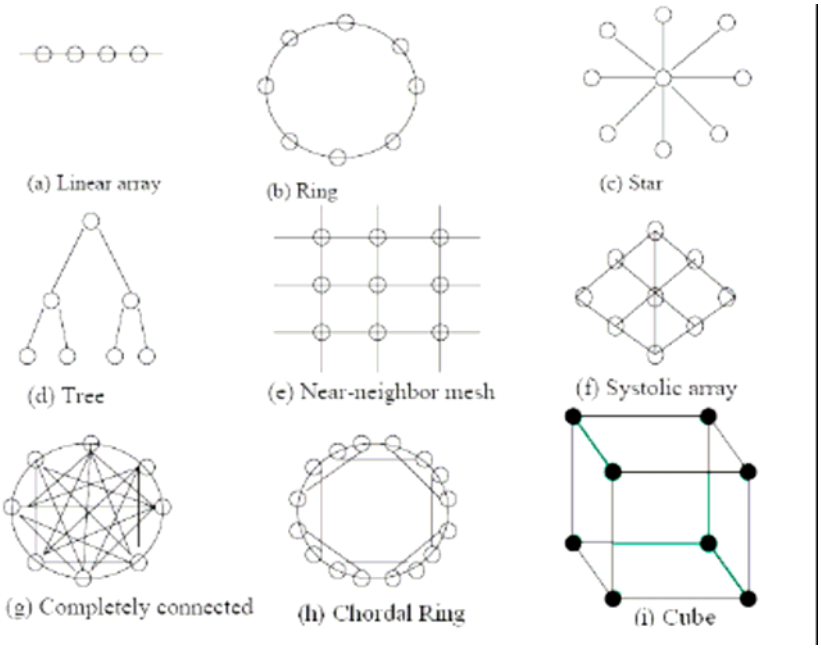


Figure 2.2 Static interconnection network topologies.

Torus architecture is also one of popular network topology it is extension of the mesh by having wraparound connections Figure below is a 2D Torus This architecture of torus is a symmetric topology unlike mesh which is not. The wraparound connections reduce the torus diameter and at the same time restore the symmetry. It can be

- o 1-D torus
- 2-D torus
- 3-D torus

The torus topology is used in Cray T3E

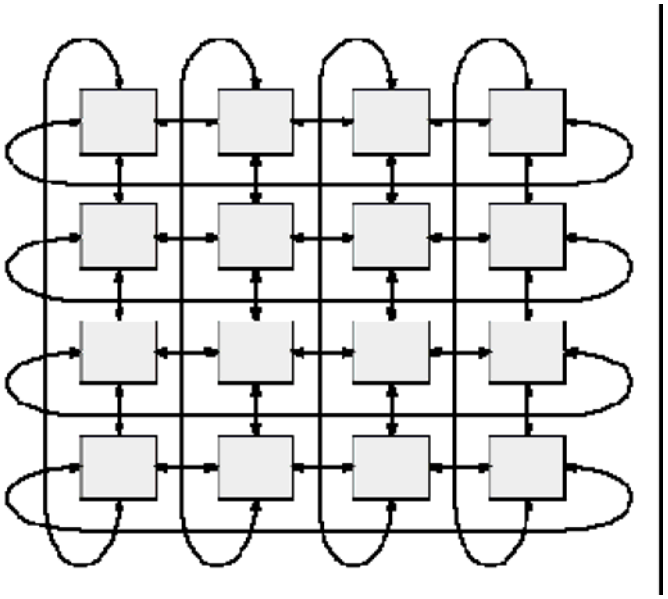


Figure 2.3 Torus technology

We can have further higher dimension circuits for example 3-cube connected cycle. A D-dimension W-wide hypercube contains W nodes in each dimension and there is a connection to a node in each dimension. The mesh and the cube architecture are actually 2-D and 3-D hypercube respectively. The below figure we have hypercube with dimension 4.

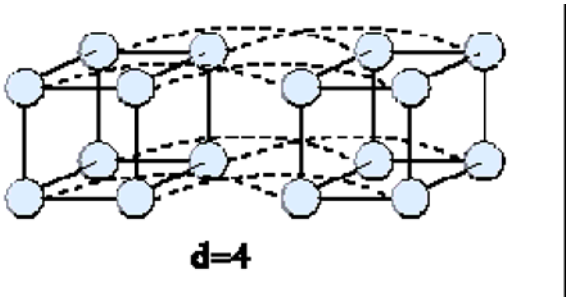


Figure 2.4 4-D hypercube.

2.5.3 Dynamic connection Networks

The dynamic networks are those networks where the route through which data move from one PE to another is established at the time communication has to be performed. Usually all processing elements are equidistant and an interconnection path is established when two processing element want to communicate by use of switches. Such systems are more difficult to expand as compared to static network. Examples: Bus-based, Crossbar, Multistage Networks. Here the Routing is done by comparing the bit-level representation

of source and destination addresses. If there is a match goes to next stage via pass-through else in case of it mismatch goes via cross-over using the switch.

There are two classes of dynamic networks namely

- single stage network
- multi stage

2.5.3.1 Single Stage Networks

A single stage switching network with N input selectors (IS) and N output selectors (OS). Here at each network stage there is a 1- to-D demultiplexer corresponding to each IS such that 1<D<N and each OS is an M-to-1 multiplexer such that 1<M<=N. Cross bar network is a single stage network with D=M=N. In order to establish a desired connecting path different path control signals will be applied to all IS and OS selectors. The single stage network is also called as recirculating network as in this network connection the single data items may have to recirculate several time through the single stage before reaching their final destinations. The number of recirculation depends on the connectivity in the single stage network. In general higher the hardware connectivity the lesser is the number of recirculation. In cross bar network only one circulation is needed to establish the connection path. The cost of completed connected cross bar network is O(N2) which is very high as compared to other most recirculating networks which have cost O(N log N) or lower hence are more cost effective for large value of N.

2.5.3.2 Multistage Networks

Many stages of interconnected switches form a multistage SIMD network. It is basicaaly consist of three characteristic features

- The switch box,
- The network topology
- The control structure

Many stages of interconnected switches form a multistage SIMD networks. Eachbox is essentially an interchange device with two inputs and two outputs. The four possible states of a switch box are which are shown in figure 3.6

- Straight
- Exchange
- Upper Broadcast

- Lower broadcast.

A two function switch can assume only two possible state namely state or exchange states. However a four function switch box can be any of four possible states. A multistage network is capable of connecting any input terminal to any output terminal. Multi-stage networks are basically constructed by so called shuffle-exchange switching element, which is basically a 2 x 2 crossbar. Multiple layers of these elements are connected and form the network.

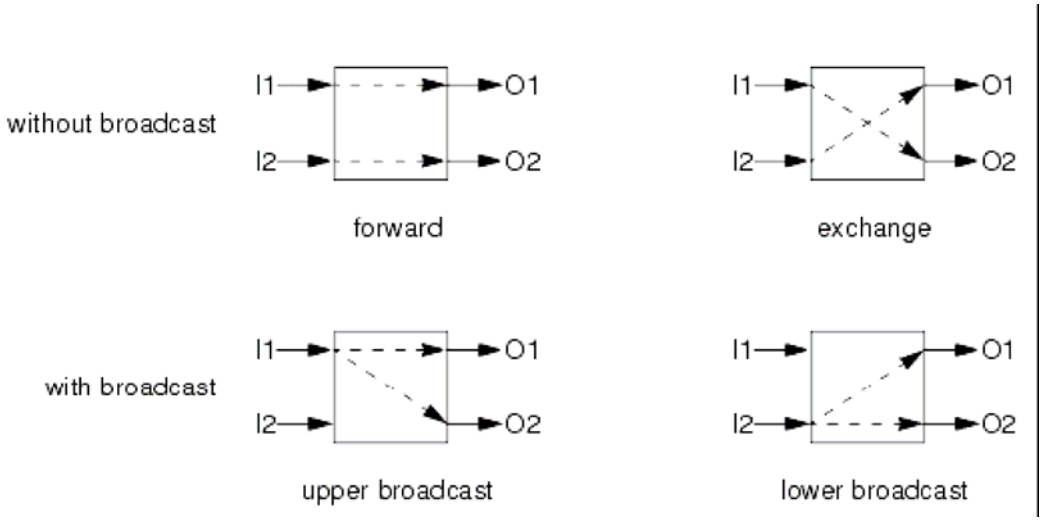


Figure 2.5 A two-by-two switching box and its four interconnection states

A multistage network is capable of connecting an arbitrary input terminal to an arbitrary output terminal. Generally it is consist of n stages where $N = 2^n$ is the number of input and output lines. And each stage use N/2 switch boxes. The interconnection patterns from one stage to another stage is determined by network topology. Each stage is connected to the next stage by at least N paths. The total wait time is proportional to the number stages i.e., n and the total cost depends on the total number of switches used and that is $N\log_2 N$. The control structure can be individual stage control i.e., the same control signal is used to set all switch boxes in the same stages thus we need n control signal. The second control structure is individual box control where a separate control signal is used to set the state of each switch box. This provide flexibility at the same time require n2/2 control signal which increases the complexity of the control circuit. In between path is use of partial stage control.

Examples of Multistage Networks

Banyan

Baseline

Cube

Delta

Flip

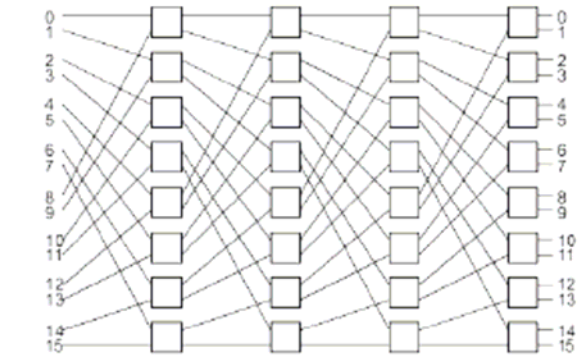
Indirect cube

Omega

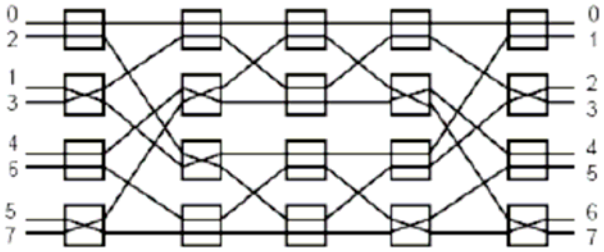
Multistage network can be of two types

- One side networks : also called full switch having input output port on the same side
- Two sided multistage network : which have an input side and an output side. It can be further divided into three class
    - Blocking: In Blocking networks, simultaneous connections of more than one terminal pair may result conflicts in the use of network communication links. Examples of blocking network are the Data Manipulator, Flip, N cube, omega, baseline. All multistage networks that are based on shuffle-exchange elements, are based on the concept of blocking network because not all possible here to make the input-output connections at the same time as one path might block another. The figure 2.6 (a) show an omega network.
    - Rearrangeable : In rearrangeable network, a network can perform all possible connections between inputs and outputs by rearranging its existing connections so that a connection path for a new input-output pair can always be established. An example of this network topology is Benes Network (see figure 2.6 (b) showing a 8** Benes network)which support synchronous data permutation and a synchronous interprocessor communication.
    - Non blocking : A non –blocking network is the network which can handle all possible connections without blocking. There two possible cases first one is the Clos network (see figure 2.6(c)) where a one to one connection

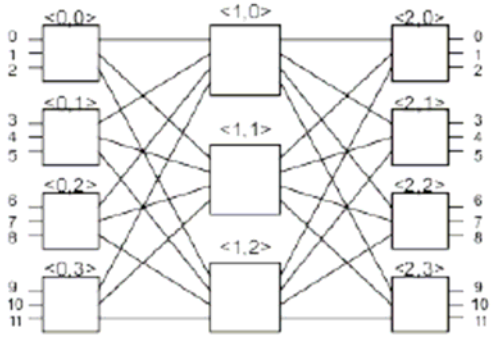
is made between input and output. Another case of one to many connections can be obtained by using crossbars instead of the shuffle-exchange elements. The cross bar switch network can connect every input port to a free output port without blocking.



(a) Omega Network



(b) Benes Network



(c) Clos Network

Figure 2.6 Several Multistage Interconnection Networks

Mesh-Connected Illiac Networks

A single stage recirculating network has been implemented in the ILLiac –IV array with N= 64 PEs. Here in mesh network nodes are arranged as a q-dimensional lattice. The

neighboring nodes are only allowed to communicate the data in one step i.e., each PEi is allowed to send the data to any one of PE(i+1) , PE (i-1), Pe(i+r) and PE(i-r) where r= square root N(in case of Iliac r=8). In a *periodic mesh*, nodes on the edge of the mesh have wrap-around connections to nodes on the other side this is also called a *toroidal mesh*.

Mesh Metrics

For a q-dimensional non-periodic lattice with kq nodes:

• Network connectivity = q

• Network diameter = q(k-1)

• Network narrowness = k/2

• Bisection width = kq-1

• Expansion Increment = kq-1

• Edges per node = 2q

Thus we observe the output of IS k is connected to inputs of OSj where j = k-1,K+1,k-r,k+r as shown in figure below.

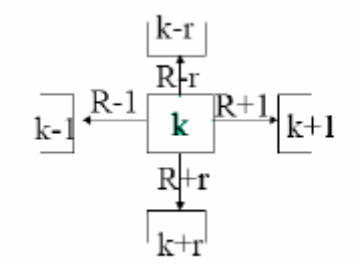


Figure2.7 routing function of mesh Topology

Similarly the OSj gets input from ISk for K= j-1, j+1,j-r,j+r. The topology is formerly described by the four routing functions:

• R+1(i)= (i+1) mod N => (0,1,2…,14,15)

• R-1(i)= (i-1) mod N => (15,14,…,2,1,0)

• R+r(i)= (i+r) mod N => (0,4,8,12)(1,5,9,13)(2,6,10,14)(3,7,11,15)

• R-r(i)= (i-r) mod N => (15,11,7,3)(14,10,6,2)(13,9,5,1)(12,8,4,0)

The figure given below show how each PEi is connected to its four nearest neighbors in the mesh network. It is same as that used for IILiac –IV except that w had reduced it for N=16 and r=4. The index are calculated as module N.

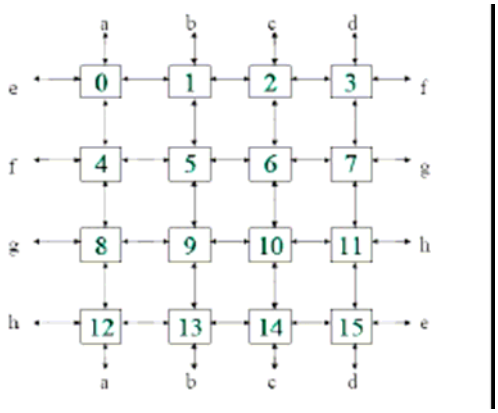


Figure 2.8 Mesh Connections

Thus the permutation cycle according to routing function will be as follows: Horizontally, all PEs of all rows form a linear circular list as governed by the following two permutations, each with a single cycle of order N. The permutation cycles (a b c) (d e) stands for permutation a->b, b->c, c->a and d->e, e->d in a circular fashion with each pair of parentheses.

R+1 = (0 1 2 ….N-1)

R–1 = (N-1 ….. 2 1 0).

Similarly we have vertical permutation also and now by combining the two permutation each with four cycles of order four each the shift distance for example for a network of N = 16 and r = square root(16) = 4, is given as follows:

R +4 = (0 4 8 12)(1 5 9 13)(2 6 10 14)(3 7 11 15)

R –4 = (12 8 4 0)(13 9 5 1)(14 10 6 2)(15 11 7 3)

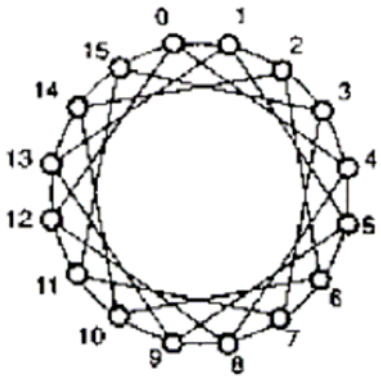


Figure 4.9 Mesh Redrawn

Each PEi is directly connected to its four neighbors in the mesh network. The graph shows that in one step a PE can reach to four PEs, seven PEs in two step and eleven PEs in three steps. In general it takes I steps (recirculations) to route data from PEi to another PEj for a network of size N where I is upper –bound given by

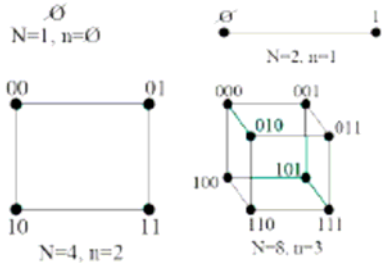
I<= square root(N) -1

Thus in above example for N=16 it will require at most 3 steps to route data from one PE to another PE and for Illiac –IV network with 64 PE need maximum of 7 steps for routing data from one PE to Another.

Cube Interconnection Networks

The cube network can be implemented as either a recirculating network or as a multistage network for SIMD machine. It can be 1-D i.e., a single line with two pE each at end of a line, a square with four PEs at the corner in case of 2-D, a cube for 3-D and hypercube in 4-D. in case of n-dimension hypercube each processor connects to 2n neighbors. This can be also visualized as the unit (hyper) cube embedded in d-dimensional Euclidean space, with one corner at 0 and lying in the positive orthant. The processors can be thought of as lying at the corners of the cube, with their (x1,x2,...,xd) coordinates identical to their processor numbers, and connected to their nearest neighbors on the cube. The popular examples where cube topology is used are : iPSC, nCUBE, SGI O2K.

Vertical lines connect vertices (PEs) whose address differ in the most significant bit position. Vertices at both ends of the diagonal lines differ in the middle bit position. Horizontal lines differ in the least significant bit position. The unit – cube concept can be extended to an n- dimensional unit space called an n cube with n bits per vertex. A cube network for an SIMD machine with N PEs corresponds to an n cube where $n = \log_2 N$. We use binary sequence to represent the vertex (PE) address of the cube. Two processors are neighbors if and only if their binary address differs only in one digit place



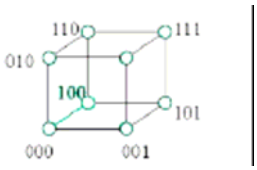
For an n-dimensional cube network of N PEs is specified by the following n routing functions

Ci (An-1 …. A1 A0)= An-1…Ai+1 A'i Ai-1……A0 for i =0,1,2,…,n-1

A n- dimension cube each PE located at the corner is directly connected to n neighbors. The addresses of neighboring PE differ in exactly one bit position. Pease's binary n cube the flip flop network used in staran and programmable switching network proposed for Phoenix are examples of cube networks.

In a recirculating cube network each ISa for 0<=A+< N-1 is connected to n OSs whose addresses are An-1…Ai+1 A'i Ai-1……A0 . When the PE addresses are considered as the corners of an m-dimensional cube this network connects each PE to its m neighbors. The interconnections of the PEs corresponding to the three routing function C0, C1 and C2 are shown separately in below figure.

• Examples



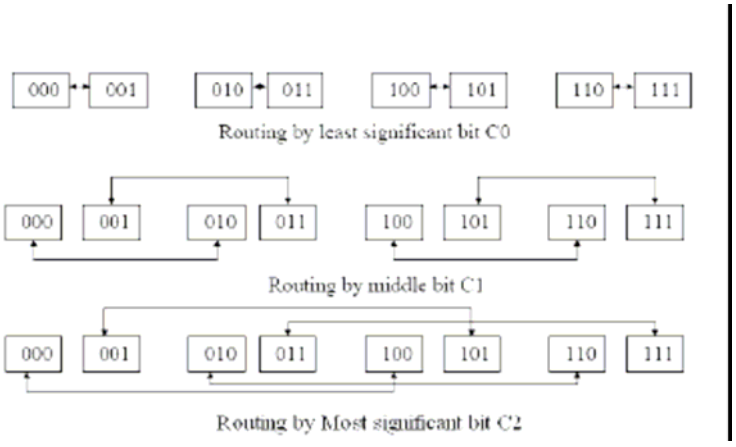


Figure 2.10 The recirculating Network

It takes n<= $\log_2 N$ steps to rotate data from any PE to another.

Example: N=8 => n=3

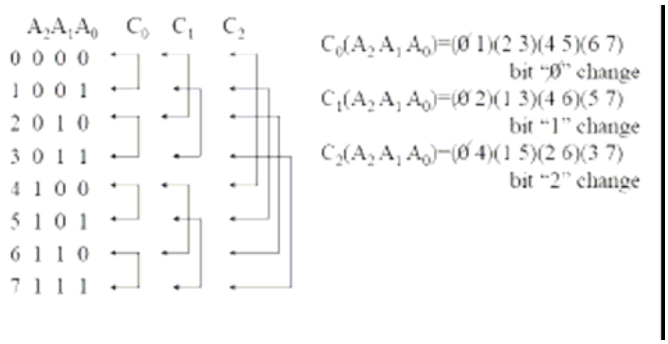


Figure 2.11 Possible routing in multistage Cube network for N = 8

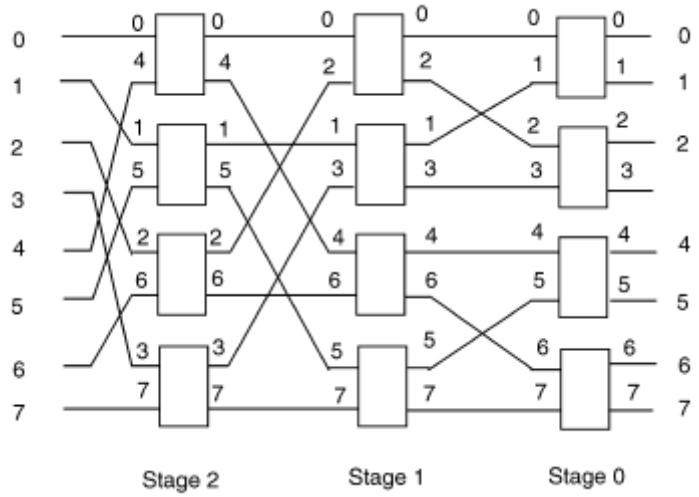


Figure 2.12 **A multistage Cube network for N = 8**

The same set of cube routing functions i.e., C0,C1, C2 can also be implemented by three stage network. Two functions switch box is used which can provide either straight and exchange routing is used for constructing multistage cube networks. The stages are numbered as 0 at input end and increased to n-1 at the output stage i.e., the stage I implements the Ci routing function or we can say at ith stage connect the input line to the output line that differ from it only at the ith bit position.

This connection was used in the early series of Intel Hypercubes, and in the CM-2.

Suppose there are 8 process ring elements so 3 bits are required for there address. and that processor 000 is the root. The children of the root are gotten by toggling the first address bit, and so are 000 and 100 (so 000 doubles as root and left child). The children

of the children are gotten by toggling the next address bit, and so are 000, 010, 100 and 110. Note that each node also plays the role of the left child. Finally, the leaves are gotten by toggling the third bit. Having one child identified with the parent causes no problems as long as algorithms use just one row of the tree at a time. Here is a picture.

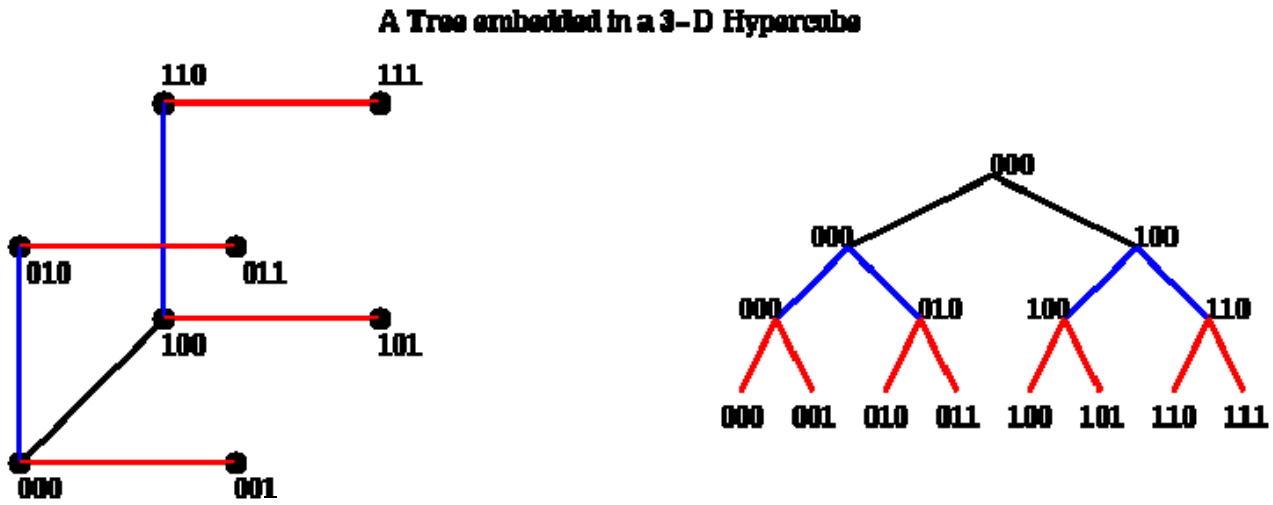


Figure 2.13 A tree embedded in 3-D hypercube

Shuffle-Exchange Omega Networks

A shuffle-exchange network consists of $n=2^k$ nodes and it is based on two routing functions shuffle (S) and exchange (E). Let A= An-1…A1A0be the address of a PE than a shuffle function is given by:

S(A)=S(An-1…A1A0)=A.n-2…A1A0An-1, 0<A<1

The cyclic shifting of the bits in A to the left for one bit osition is performed by the S function. Which is effectively like shuffling the bottom half of a card deck into the top half as shown in figure below.

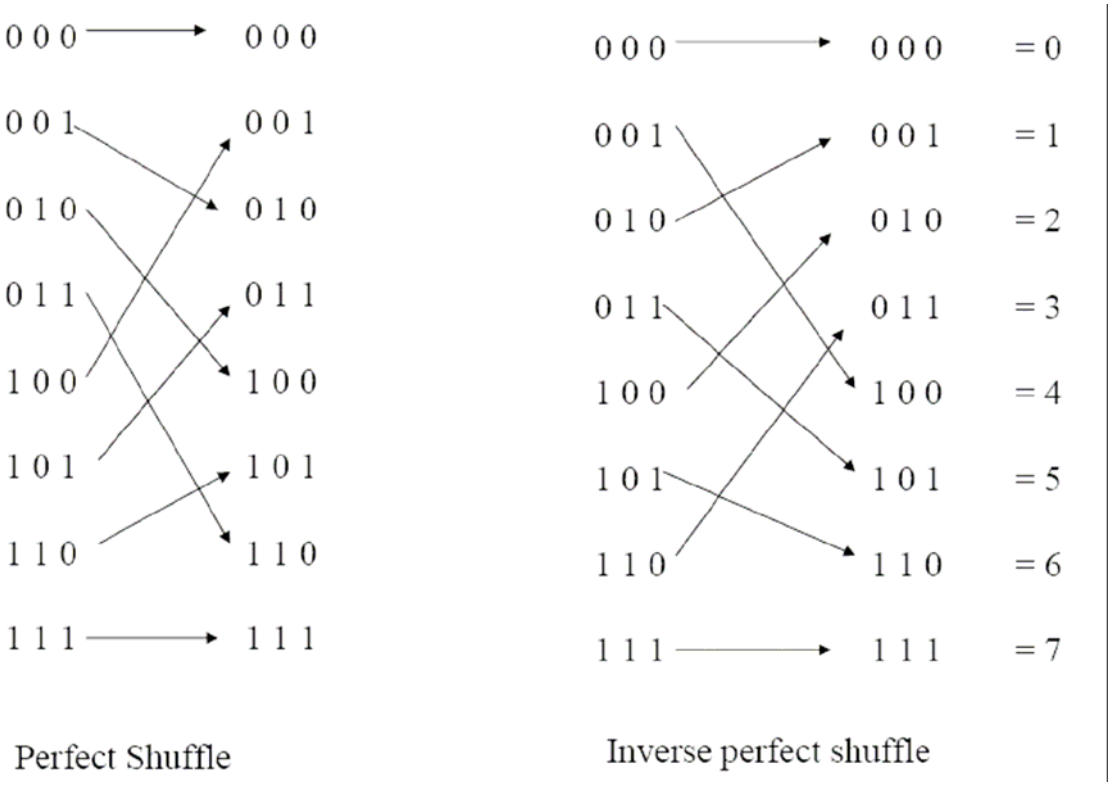


Figure 2.14 Perfect shuffle and inverse perfect shuffle

There are two type of shuffle the perfect shuffle cuts the deck into two halves from the centre and intermix them evenly. *Perfect shuffle provide the routing* connections of node i with node 2i mod(n-1), except for node n-1 which is connected to itself. The inverse perfect shuffle does the opposite to restore the original order it is denoted as exchange routing function E and is defined as :

E(An-1…A1A0)= (An-1…A1A0')

This obtained by complementing the least significant digit means data exchange between two PEs with adjacent addresses. The E(A) is same as the cube routing function as described earlier. *Exchange routing function* connects nodes whose numbers differ in their lowest bit.

The shuffle exchange function can be implemented as either a recirculating network or multistage network. The implementation of shuffle and exchange network through recirculating network is shown below. Use of shuffle and exchange topology for parallel processing was proposed by Stone. It is used for solving many parallel algorithms efficiently. The example where it is used include FFT (fast Fourier transform), sorting, matrix transposition , polynomial evaluations etc.

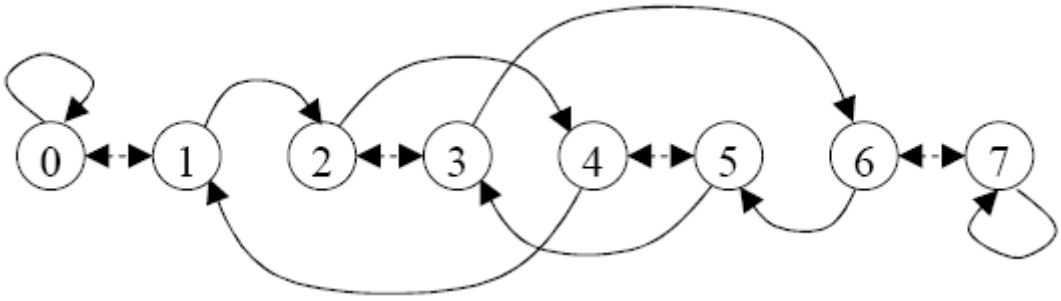
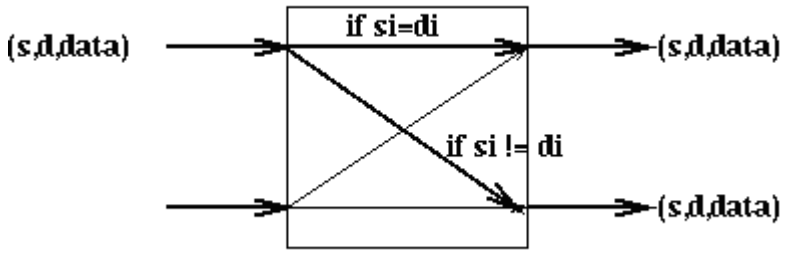


Figure2.15 shuffle and exchange recirculating network for N=8

The shuffle –exchange function have been implemented as multistage Omega network by LAwrie. An N by N omega network, consists of n identical stages. Between two adjacent column there is a perfect shuffle interconnection. Thus after each stage there is a N/2 four-function interchange boxes under independent box control. The four functions are namely straight exchange upper broadcast and lower broadcast. The shuffle connects output P n-l...Pl P0 of stage i to input P n-2...PlP0Pn-l of stage i-1. Each interchange box in an omega network is controlled by the n-bit destination tags associated with the data on its input lines.



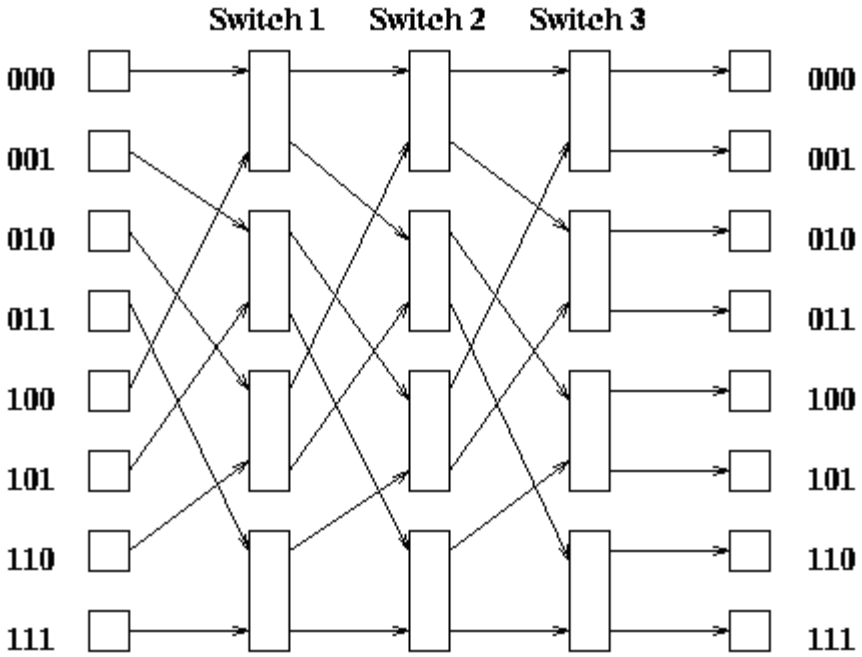


Figure 2.16

The diameter is m=log_2 p, since all message must traverse m stages. The bisection width is p. This network was used in the IBM RP3, BBN Butterfly, and NYU Ultracomputer. If we compare the omega network with cube network we find Omega network can perform one to many connections while n-cube cannot. However as far as bijections connections n-cube and Omega network they perform more or less same.

2.6 Summary

Fine-grain exploited at instruction or loop levels, assisted by the compiler.

Medium-grain (task or job step) requires programmer and compiler support.

Coarse-grain relies heavily on effective OS support.

Shared-variable communication used at fine- and medium grain levels.

Message passing can be used for medium- and coarse grain communication, but fine - grain really need better technique because of heavier communication requirements.

Control flow machines give complete control, but are less efficient than other approaches.

Data flow (eager evaluation) machines have high potential for parallelism and throughput and freedom from side effects, but have high control overhead, lose time waiting for unneeded arguments, and difficulty in manipulating data structures. Reduction (lazy

evaluation) machines have high parallelism potential, easy manipulation of data structures, and only execute required instructions. But they do not share objects with changing local state, and do require time to propagate tokens

Summary of properties of various static network

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

Summary of properties of various dynamic networks

Network Characteristics	Bus System	Multistage Network	Crossbar Switch
Minimum Latency for unit data transfer	Constant	$O(\log_k n)$	Constant
Bandwidth per processor	O(w/n) to O(w)	O(w) to O(nw)	O(w) to O(nw)
Wiring Complexity	O(w)	$O(nw \log_k n)$	$O(n^2w)$
Switching complexity	O(n)	$O(n \log_k n)$	$O(n^2)$
Connectivity and routing capability	Only one to one at a time	Some permutations and broadcast , if network unblocked	All permutations one at a time.

Metrics of dynamic connected nework

Network	Diameter	Bisection Width	Connectivity	Cost (# of links)
Crossbar	1	p	1	p^2
Omega Network	$\log p$	$p/2$	2	$p\log p$
Dynamic Tree	$2\log p$	1	2	$p-1$

2.7 Keywords

Dependence graph : A directed graph whose nodes represent calculations and whose edges represent dependencies among those calculations. If the calculation represented by

node k depends on the calculations represented by nodes i and j, then the dependence graph contains the edges i-k and j-k.

data dependency : a situation existing between two statements if one statement can store into a location that is later accessed by the other statement

granularity The size of operations done by a process between communications events. A fine grained process may perform only a few arithmetic operations between processing one message and the next, whereas a coarse grained process may perform millions

control-flow computers refers to an *architecture* with one or more program counters that determine the order in which instructions are executed.

dataflow A model of parallel computing in which programs are represented as *dependence graphs* and each operation is automatically *blocked* until the values on which it depends are available. The parallel functional and parallel logic programming models are very similar to the dataflow model.

network A physical communication medium. A network may consist of one or more *buses*, a *switch*, or the *links* joining processors in a *multicomputer*.

Static networks: point-to-point direct connections that will not change during program execution

Dynamic networks: switched channels dynamically configured to match user program communication demands include buses, crossbar switches, and multistage networks

routing The act of moving a message from its source to its destination. A routing technique is a way of handling the message as it passes through individual nodes.

Diameter D of a network is the maximum shortest path between any two nodes, measured by the number of links traversed; this should be as small as possible (from a communication point of view).

Channel bisection width b = minimum number of edges cut to split a network into two parts each having the same number of nodes. Since each channel has w bit wires, the wire bisection width B = bw. Bisection width provides good indication of maximum communication bandwidth along the bisection of a network, and all other cross sections should be bounded by the bisection width.

Wire (or channel) length = length (e.g. weight) of edges between nodes.

2.8 **Self assessment questions**

1. Perform a data dependence analysis on each of the following Fortran program fragments. Show the dependence graphs among the statements with justification.

   a.

   S1: A = B + D

   S2: C = A X 3

   S3: A = A + C

   S4: E = A / 2

   b.

   S1: X = SIN(Y)

   S2: Z = X + W

   S3: Y = -2.5 X W

   S4: X = COS(Z)

3 Explain data, control and resource dependence.

4 Describe Bernstein's conditions.

5 Explain the hardware cum software parallelism and the role of compilers

6 Explain the following a) Instruction level b) Loop level c) Procedure level d) Subprogram level and e) Job/program level parallelism.

7 Compare dataflow, control-flow computers and reduction computer architectures.

8 Explain reduction machine models with respect to demand-driven mechanism

9 Explain the following a) Node degree and network diameter b) Bisection width c) Data-routing functions d) Hyper-cube routing functions.

10 Explain the following static connection network topologies a) Linear array b) Ring and chordal ring c)Barrel shifter d) Tree and star e) Fat tree f) Mesh and Torus

11. Describe the important features of buses, multistage networks and crossbar switches in building dynamic networks.

2.9 References/Suggested readings

Advance Computer architecture: Kai Hwang

Author: Dr. Deepti Mehrotra **Vetter: Dr. Sandeep Arya**

Lesson: Processors and memory hierarchy **Lesson No. : 03**

3.0 Objective

The present lesson we will discuss the present modern processor technology and the supporting memory hierarchy.

3.1 Introduction

In today's era there are large variety of multiprocessor exist. In this lesson we will study about advances in the processor technology. Superscalar and Vector Processors architecture will be discussed in detail. We will also discuss memory organization that is needed to support this technology and finally we will discuss about virtual memory concept and issues involved in implementing it specially in multiprocessors.

3.2 Advanced processor technology

3.2.1 Design Space of Processors

Processors can be “mapped” to a space that has clock rate and cycles per instruction (CPI) as coordinates. Each processor type occupies a region of this space. The newer technologies are enabling higher clock rates. So the various processors are gradually moving from low to higher speeds towards the right of the design space. Manufacturers are also trying to lower the number of cycles per instruction. Thus the “future processor space” is moving toward the lower right of the processor design space.

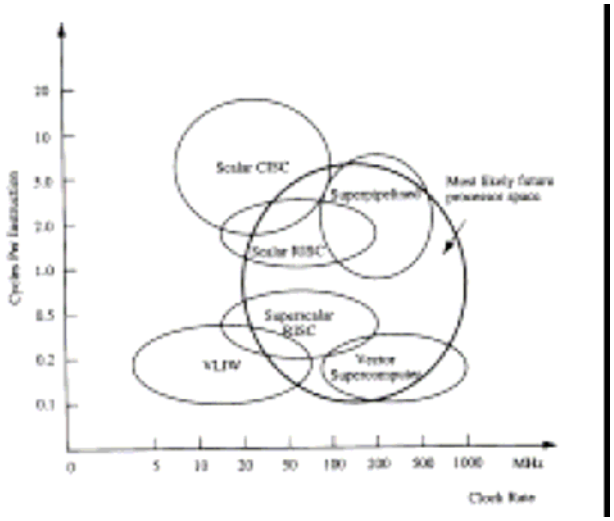


Figure 3.1 Design space of modern processor families

CISC and RISC Processors

The popular examples of Complex Instruction Set Computing (CISC) processors are Intel 80486, the Motorola 68040, the VAX/8600, and the IBM S/390. CISC architecture typically use microprogrammed control units, have lower clock rates, and higher CPI and are located at the upper left of design space.

Reduced Instruction Set Computing (RISC) processors like the Intel i860, SPARC, MIPS R3000, and IBM RS/6000 have hard-wired control units, higher clock rates, and lower CPI approximately one to two cycles and are located below CISC processors in design space. Designed to issue one instruction per cycle RISC and CISC scalar processors should have same performance if clock rate and program lengths are equal. RISC moves less frequent operations into software, thus dedicating hardware resources to the most frequently used operations.

RISC Scalar Processors: A special subclass of RSIC processors are the superscalar processors which allow multiple instruction to be issued simultaneously during the cycle.

The effective CPI of a superscalar processor should be less than that of a generic scalar RISC processor.

Clock rates of scalar RISC and superscalar RISC machines are similar.

Superpipelined Processors: These processors typically use a multiphase clock (actually several clocks that are out of phase with each other, each phase perhaps controlling the issue of another instruction) running at a relatively high rate. The CPI in these machines tends to be relatively high (unless multiple instruction issue is used). Processors in vector supercomputers are mostly superpipelined and use multiple functional units for concurrent scalar and vector operations.

VLIW Machines Very Long Instruction Word machines typically have many more functional units that superscalars (and thus the need for longer – 256 to 1024 bits – instructions to provide control for them). These machines mostly use microprogrammed control units with relatively slow clock rates because of the need to use ROM to hold the microcode.

Instruction pipeline

The execution cycle of a typical instruction includes four phases fetch, decode, execute and write –back. These instructions are executed as instruction pipeline before we discuss pipeline in details lets see some fundamental definitions associated with instruction pipeline.

Instruction pipeline cycle – the time required for each phase to complete its operation (assuming equal delay in all phases)

Instruction issue latency – the time (in cycles) required between the issuing of two adjacent instructions

Instruction issue rate – the number of instructions issued per cycle (the degree of a superscalar)

Simple operation latency – the delay (after the previous instruction) associated with the completion of a simple operation (e.g. integer add) as compared with that of a complex operation (e.g. divide).

Resource conflicts – when two or more instructions demand use of the same functional unit(s) at the same time.

Pipelined Processors A base scalar processor: issues one instruction per cycle has a one-cycle latency for a simple operation has a one-cycle latency between instruction issues can be fully utilized if instructions can enter the pipeline at a rate on one per cycle

For a variety of reasons, instructions might not be able to be pipelines as aggressively as in a base scalar processor. In these cases, we say the pipeline is under pipelined. CPI rating is 1 for an ideal pipeline. Underpipelined systems will have higher CPI ratings, lower clock rates, or both.

Processors and Coprocessors

Central processing unit (CPU) is essentially a scalar processor which may have many functional units some systems may include one or more coprocessors which perform floating point or other specialized operations –

INCLUDING I/O, regardless of what the textbook says. Coprocessors cannot be used without the appropriate CPU.

Other terms for coprocessors include attached processors or slave processors.

Coprocessors can be more “powerful” than the host CPU.

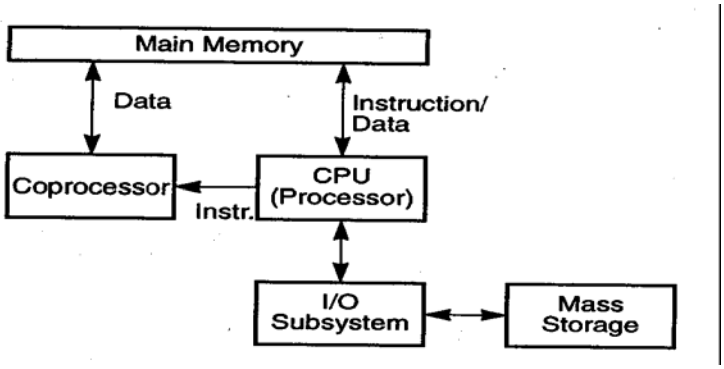


Figure 3.2 CPU with attached coprocessor

Instruction Set Architectures

Computers are classified on the basis on instruction set they have

CISC

- Many different instructions
- Many different operand data types
- Many different operand addressing formats
- Relatively small number of general purpose registers
- Many instructions directly match high-level language constructions

RISC

- Many fewer instructions than CISC (freeing chip space for more functional units!)
- Fixed instruction format (e.g. 32 bits) and simple operand addressing
- Relatively large number of registers
- Small CPI (close to 1) and high clock rates

Architectural Distinctions

CISC

- Unified cache for instructions and data (in most cases)
- Microprogrammed control units and ROM in earlier processors (hard-wired controls units now in some CISC systems)

RISC

- Separate instruction and data caches
- Hard-wired control units

RISC Scalar Processors

- Designed to issue one instruction per cycle RISC and CISC scalar processors should have same performance if clock rate and program lengths are equal.
- RISC moves less frequent operations into software, thus dedicating hardware resources to the most frequently used operations.
- Representative systems: Sun SPARC, Intel i860, Motorola M88100, AMD 29000

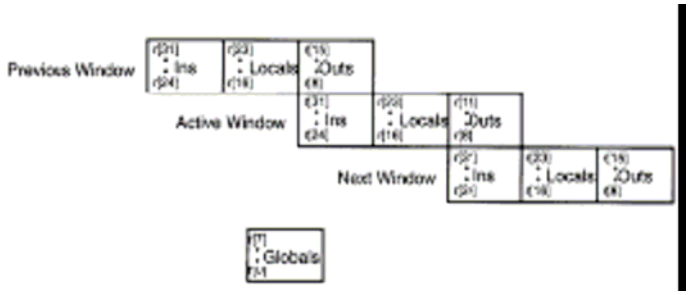
Lets take the case study of RISC scalor processor SPARC

SPARCs and Register Windows

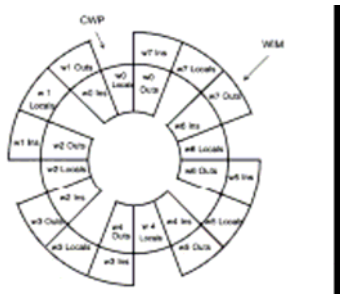
The SPARC architecture makes clever use of the logical procedure concept.

Each procedure usually has some input parameters, some local variables, and some arguments it uses to call still other procedures.

The SPARC registers are arranged so that the registers addressed as “Outs” in one procedure become available as “Ins” in a called procedure, thus obviating the need to copy data between registers. This is similar to the concept of a “stack frame” in a higher level language.



(a) three overlapping register window and the global registers



(b) Eight register windows forming a circular stack

Figure 3.3 The concept of overlapping register windows in the SPARC architecture

3.3Superscalar and vector processors

3.3.1 Superscalar Processors

Scalar processor: executes one instruction per cycle, with only one instruction pipeline.

Superscalar processor: multiple instruction pipelines, with multiple instructions issued per cycle, and multiple results generated per cycle.

This subclass of the RISC processors that allows multiple instructions to be issued simultaneously during each cycle. The effective CPI of a superscalar processor should be less than that of a generic scalar RISC processor. Clock rates of scalar RISC and superscalar RISC machines are similar.

A typical superscalar will have multiple instruction pipelines an instruction cache that can provide multiple instructions per fetch multiple buses among the function units. In theory, all functional units can be simultaneously active.

Superscalar Constraints

It should be obvious that two instructions may not be issued at the same time (e.g. in a superscalar processor) if they are not independent.

This restriction ties the instruction-level parallelism directly to the code being executed.

The instruction-issue degree in a superscalar processor is usually limited to 2 to 5 in practice.

Superscalar Pipelines

One or more of the pipelines in a superscalar processor may stall if insufficient functional units exist to perform an instruction phase (fetch, decode, execute, write back).

Ideally, no more than one stall cycle should occur.

In theory, a superscalar processor should be able to achieve the same effective parallelism as a vector machine with equivalent functional units.

A typical superscalar will have multiple instruction pipelines an instruction cache that can provide multiple instructions per fetch multiple buses among the function units In theory, all functional units can be simultaneously active.

3.3.2 VLIW Architecture

VLIW = Very Long Instruction Word Instructions usually hundreds of bits long. Each instruction word essentially carries multiple “short instructions.” Each of the “short instructions” are effectively issued at the same time. (This is related to the long words frequently used in microcode.) Compilers for VLIW architectures should optimally try to predict branch outcomes to properly group instructions.

Pipelining in VLIW Processors

Decoding of instructions is easier in VLIW than in superscalars, because each “region” of an instruction word is usually limited as to the type of instruction it can contain.

Code density in VLIW is less than in superscalars, because if a “region” of a VLIW word isn’t needed in a particular instruction, it must still exist (to be filled with a “no op”).

Superscalars can be compatible with scalar processors; this is difficult with VLIW parallel and non-parallel architectures.

VLIW Opportunities

“Random” parallelism among scalar operations is exploited in VLIW, instead of regular parallelism in a vector or SIMD machine.

The efficiency of the machine is entirely dictated by the success, or “goodness,” of the compiler in planning the operations to be placed in the same instruction words.

Different implementations of the same VLIW architecture may not be binary-compatible with each other, resulting in different latencies.

3.3.3 Vector Processors

Vector processors issue one instructions that operate on multiple data items (arrays). This is conducive to pipelining with one result produced per cycle.

A vector processor is a coprocessor designed to perform vector computations. A vector is a one-dimensional array of data items (each of the same data type). Vector processors are often used in multipipelined supercomputers.

Architectural types include:

- register-to-register (with shorter instructions and register files)
- memory-to -memory (longer instructions with memory addresses)

Register-to-Register Vector Instructions

Assume Vi is a vector register of length n, si is a scalar register, M(1:n) is a memory array of length n, and "ï" is a vector operation.

The ISA of a scalar processor is augmented with vector instructions of the following types:

Vector-vector instructions:

f1: Vi -> Vj (e.g. MOVE Va, Vb)

f2: Vj x Vk -> Vi (e.g. ADD Va, Vb, Vc)

Vector-scalar instructions:

f3: s x Vi -> Vj (e.g. ADD R1, Va, Vb)

Vector-memory instructions:

f4: M -> V (e.g. Vector Load)

f5: V -> M (e.g. Vector Store)

Vector reduction instructions:

f6: V -> s (e.g. ADD V, s)

f7: Vi x Vj ->s (e.g. DOT Va, Vb, s)

Pipelines in Vector Processors

Vector processors can usually effectively use large pipelines in parallel, the number of such parallel pipelines effectively limited by the number of functional units.

As usual, the effectiveness of a pipelined system depends on the availability and use of an effective compiler to generate code that makes good use of the pipeline facilities.

Symbolic Processors

Symbolic processors are somewhat unique in that their architectures are tailored toward the execution of programs in languages similar to LISP, Scheme, and Prolog.
In effect, the hardware provides a facility for the manipulation of the relevant data objects with “tailored” instructions.
These processors (and programs of these types) may invalidate assumptions made about more traditional scientific and business computations.

Superpipelining

Superpipelining is a new and special term meaning pipelining. The prefix is attached to increase the probability of funding for research proposals. There is no theoretical basis distinguishing superpipelining from pipelining. Etymology of the term is probably similar to the derivation of the now-common terms, methodology and functionality as pompous substitutes for method and function. The novelty of the term superpipelining lies in its reliance on a prefix rather than a suffix for the pompous extension of the root word.

3.4 Hierarchical Memory Technology

As we now variety of memories are available in market. These memories are categorized according to their properties like speed of accessing the data, capacity to storage of data, whether it is volatile or non volatile nature, how data are stored and how it is accessed, rate with data are transferred etc. As an end user the most important points that one consider while designing the memory organization for a computer are: Its size (capacity), speed (access time), cost and how frequently it will be accessed by the processor.

If we want the increase the speed of the system the major concern it to improve the performance of the two important and most used components of the system that are processor and memory. If the relative speed of processors and memories are considered, it is observed technology present today are so that the processors speed increase by a factor of about 10000 if the speed of memory is doubled. Hence even if the speed of processor is increased the overall speed of system will not increase in same ratio because of bottle neck created by memory. The main choice of memory designers is establish a balance between speed and capacity.
The common used devices for storage are registers, RAM, ROM, Hard disk, Magnetic tape, CD ROM etc. Among these fastest memory units are registers having access times

below 10ns but has the lowest capacity of few KB of words while the slow devices are like magnetic disk and magnetic tape can storage large amount of data i.e., have high capacity of few GBytes but same time access times of several seconds. Thus to implement a balance between speed and capacity we should employ a memory hierarchy in a system such that high speed memories, which are expensive and faster and comparatively smaller size should hold preferably the most recently accessed items kept that need to be close to the CPU and successively large and slow memories are kept away from the CPU to hold complete back up of data. This way of designing a memory system is called **a memory hierarchy as shown in figure** 3.4. Memory in system is usually characterized as appearing at various levels (0, 1, …) in a hierarchy, with level 0 being CPU registers and level 1 being the cache closest to the CPU. Each level is characterized by following parameters:

- access time ti (round-trip time from CPU to ith level)
- memory size si (number of bytes or words in the level)
- cost per byte ci
- transfer bandwidth bi (rate of transfer between levels)
- unit of transfer xi (grain size for transfers)

As one goes down the hierarchy the following occur:

- Decrease in cost per bit
- Increase in capacity
- Increase in access time
- Decrease in frequency of access of the memory by the processor

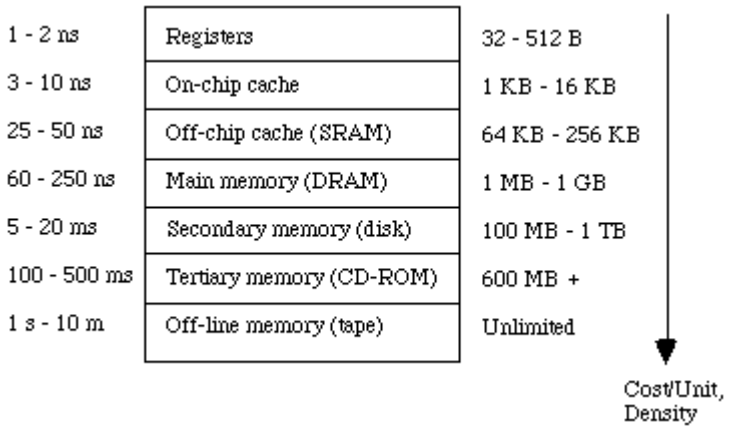


Figure 3. 4The memory hierarchy

The following three principles which let to an effective implementation memory hierarchy for a system are:

1 M**ake the Common Case Fast :** This principle says the data which is more frequently used should be kept in faster device. It is based on a fundamental law, called Amdahl's Law , which *states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used*. Thus if faster mode use relatively less frequent data then most of the time faster mode device will not be used hence the speed up achieved will be less than if faster mode device is more frequently used.

2. **Principle of Locality :** It is very common trend of Programs to reuse data and instructions that are used recently. Based on this observation comes important program property called *locality of references*: the *instructions* and *data* in a program that will be used in the near future is based on its accesses in the recent past. There is a famous **40/10 rule that** comes from empirical observation is:

"A program spends 40% of its time in 10% of its code"

These localities can be categorized of three types:

a. Temporal locality: states that data items and code that are recently accessed are likely to be accessed in the near future. Thus if location M is referenced at time t, then it (location M) will be referenced again at some time t+Dt.

b. Spatial locality: states that items try to reside in proximity in the memory i.e., the items whose addresses are near to each other are likely to be referred together in time. Thus we can say memory accesses are clustered with respect to the address space. Thus if location M is referenced at time t, then another location M±Dm will be referenced at time t+Dt.

c. Sequential locality: Programs are stored sequentially in memory and normally these programs has sequential trend of execution. Thus we say instructions are stored in memory in certain array patterns and are accessed sequentially one memory locations after another. Thus if location M is referenced at time t, then locations M+1, M+2, … will be referenced at time t+Dt, t+Dt', etc. In each of these patterns, both Dm and Dt are "small."

H&P suggest that 90 percent of the execution time in most programs is spent executing only 10 percent of the code. One of the implications of the locality is data and instructions should have separate data and instruction caches. The main advantage of separate caches is that one can fetch instructions and operands simultaneously. This concept is basis of the design known as Harvard architecture, after the Harvard Mark series of electromechanical machines, in which the instructions were supplied by a separate unit.

3. **Smaller is Faster :** *Smaller pieces of hardware will generally be faster than larger pieces.*

This according to above principles suggested that one should try to keep recently accessed items in the fastest memory.

While designing the memory hierarchy following points are always considered

Inclusion property : If a value is found at one level, it should be present at all of the levels below it.

$M_1 \subset M_2 \subset \dots \subset M_n$

The implication of the inclusion property is that all items of information in the "innermost" memory level (cache) also appear in the outer memory levels.The inverse, however, is not necessarily true. That is, the presence of a data item in level Mi+1 does not imply its presence in level Mi. We call a reference to a missing item a "miss."

The Coherence Property

The value of any data should be consistent at all level. The inclusion property is, of course, never completely true, but it does represent a desired state. That is, as information is modified by the processor, copies of that information should be placed in the appropriate locations in outer memory levels. The requirement that copies of data items at successive memory levels be consistent is called the "coherence property."

Coherence Strategies

Write-through

As soon as a data item in M i is modified, immediate update of the corresponding data item(s) in M i+1, Mi+2, … Mn is required. This is the most aggressive (and expensive) strategy.

Write-back

The update of the data item in M i+1 corresponding to a modified item in Mi is not updated unit it (or the block/page/etc. in M i that contains it) is replaced or removed. This is the most efficient approach, but cannot be used (without modification) when multiple processors share Mi+1, …, Mn.

Locality: As any program use mainly some portion of it at a given time i.e., the programs access a restricted portion of their address space in any time. So the portion that program may need should kept at higher level and remaining program at lower level. Locality is entirely program-dependent. Most caches implement locality assuming sequential code. In most programs, memory references are assumed to occur in patterns that are strongly related (statistically) to each as discussed in reference of locality.

Working Sets

The set of addresses (bytes, pages, etc.) referenced by a program during the interval from t to t+w, where w is called the working set parameter, changes slowly.

This set of addresses, called the working set, should be present in the higher levels of M if a program is to execute efficiently (that is, without requiring numerous movements of data items from lower levels of M). This is called the working set principle.

Hit Ratios

When a needed item (instruction or data) is found in the level of the memory hierarchy being examined, it is called a hit. Otherwise (when it is not found), it is called a miss (and the item must be obtained from a lower level in the hierarchy).

The hit ratio, h, for Mi is the probability (between 0 and 1) that a needed data item is found when sought in level memory Mi. The miss ratio is obviously just 1-hi. We assume h0 = 0 and hn = 1.

To evaluate the effectiveness of the memory hierarchy the following formula is used:

Memory_stall_cycles = IC * Mem_Refs * Miss_Rate * Miss_Penalty

Where IC = Instruction count

Mem_Refs = Memory References per Instruction

Miss_Rate = the fraction of accesses that are not in the given memory

Miss_Penalty = the additional time to service the miss

The hit ratio is an important measure of the performance of a memory level and is the probability that a reference is to a value already in a given level of the hierarchy. The miss ratio is 1 - h. Thus if any data is not present in given level of memory it should go to lower hierarchy level. The Miss penalty time is the sum of the access frequencies times their corresponding access times where the access frequency is the product of the hit ratio for the given level with the miss ratios of all higher levels

Memory Generalities

It is almost always the case that memories at lower-numbered levels, when compare to those at higher-numbered levels are faster to access, are smaller in capacity, are more expensive per byte, have a higher bandwidth, and have a smaller unit of transfer.

In general, then, ti-1 < ti, si-1 < si, ci-1 > ci, bi-1 > bi, and xi-1 < xi.

Access Frequencies

The access frequency fi to level Mi is

(1-h1) * (1-h2)* … * hi.

Note that f1 = h1, and $\sum_{i=1}^{n} f_i = 1$

Effective Access Times

There are different penalties associated with misses at different levels in the memory hierarchy. A cache miss is typically 2 to 4 times as expensive as a cache hit (assuming success at the next level).

A page fault (miss) is 3 to 4 magnitudes as costly as a page hit. The effective access time of a memory hierarchy can be expressed as

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i = h_1 t_1 + (1-h_1)h_2 t_2 + \cdots + (1-h_1)(1-h_2)\cdots(1-h_{n-1})h_n t_n$$

The first few terms in this expression dominate, but the effective access time is still dependent on program behavior and memory design choices.

Hierarchy Optimization

Given most, but not all, of the various parameters for the levels in a memory hierarchy, and some desired goal (cost, performance, etc.), it should be obvious how to proceed in determining the remaining parameters.

3.5 Virtual Memory

The virtual memory is a method of using the hard disk as if it is part of main memory, so that the program with size can be larger than the actual physical memory available can even execute. This technique is especially useful for the multiprogramming system where more than one program reside in main memory, such a system is managed efficiently with help of operating system. The objective of *virtual memory* is to have as much as possible of the program in main memory and remaining on the hard disk and the operating system with some hardware support, swaps data between memory and disk such that it interfere the running of the program minimum. If swapping in and out of main storage becomes the dominant activity, then the situation is referred as *thrashing,* and it reduces the efficiencies greatly. Virtual memory can be thought as a way to provide an illusion to the user that disk is an extension of main memory. A virtual memory system provides a mechanism for translating a program generated address into main memory location. Program uses virtual memory addresses. This virtual address is converted to a physical address which indicates physical location of data in main memory as shown in figure 3.5. Virtual memory can be defined as

A mapping from a virtual address space to a physical address space.

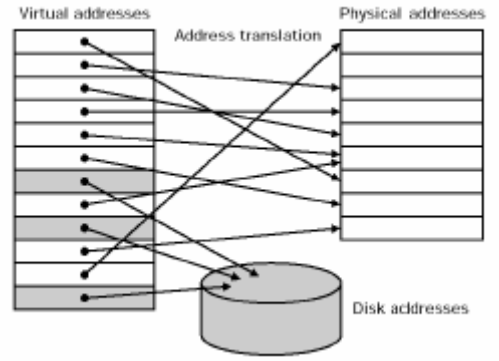


Figure 3.5 Mapping from virtual address to physical address

Virtual memory is used by almost all *uniprocessors* and *multiprocessors*, but is not available on some *array processors* and *multicomputers*, which still employ real memory storage only on each *node*. Any program under execution should reside in the main memory as CPU cannot directly access hard disk. The main memory usually starts at physical address 0. Certain locations may be reserved for special purposes program; like the operating system that usually reside at the low addresses. The rest of physical store may be partitioned into pieces for the processes that are in the ready list. Each

process may have its own *virtual memory* i.e., the memory perceived by the process. Each process virtual memory is limited only by the machine address size.

The key term here is mapping. The processor generates an address that is sent through a mapping function to produce the physical address of the memory. Different processes are mapped to different physical memory. The principles behind *virtual memory* are as follows:

- Load as much as possible of the process into the main memory;
- Keep a copy of the complete process (its memory image) in a disk file; this is called the *swap* file.
- The virtual memory manager (in the kernel) organises the process's (virtual) memory into chunks called *pages*; pages are normally 512, 1024, 2048 or 4096 bytes or it can unequal size segments.
- Some pages are *in* main memory, others are not -- but *all* are in the swap file.
- If only part of the process be loaded is sufficient to complete the process. The memory manager with the hardware that supports do address translation;
- However, during execution if the process needs some data that is not in main memory a *page fault*. When a page fault occurs, the memory manager must read in from the swap file the page that is needed. Although it reduces speed as reading a disk file takes a minimum of 10-millisec. while reading memory takes maybe only 10-nanosec. Which page to replaced is decided by the *page replacement policy* .

The use of the mapping function between the processor and the memory adds flexibility, because it allows to store data in different places in the memory although it appears differently for different logical arrangement from the perspective of the program. The various advantages are :

1. Due this flexibility programs can be compiled for a standard address space, and can then be loaded into the available memory and run without modification. For example, in a multitasking environment more than one program can be to fit into memory with an arbitrary combination of other programs.

2. Virtual memory provides a way to make main memory appear larger than it is. It saves the programmer from having to explicitly move portions of the program or data in and out from disk. It is thus most useful for running large programs on small machines.

Mapping Efficiency

The efficiency with which the virtual to physical mapping can be accomplished significantly affects the performance of the system.

Efficient implementations are more difficult in multiprocessor systems where additional problems such as coherence, protection, and consistency must be addressed.

Virtual Memory Models (1)

Private Virtual Memory

In this scheme, each processor has a separate virtual address space, but all processors share the same physical address space.

Advantages:

• Small processor address space

• Protection on a per -page or per-process basis

• Private memory maps, which require no locking

Disadvantages

• The synonym problem – different virtual addresses in different/same virtual spaces point to the same physical page

• The same virtual address in different virtual spaces may point to different pages in physical memory

Virtual Memory Models (2)

Shared Virtual Memory

All processors share a single shared virtual address space, with each processor being given a portion of it. Some of the virtual addresses can be shared by multiple processors.

Advantages:

• All addresses are unique

• Synonyms are not allowed

Disadvantages

• Processors must be capable of generating large virtual addresses (usually > 32 bits)

• Since the page table is shared, mutual exclusion must be used to guarantee atomic updates

• Segmentation must be used to confine each process to its own address space

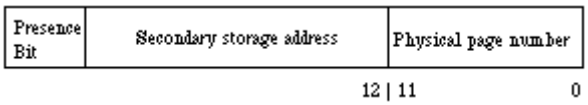
• The address translation process is slower than with private (per processor) virtual memory

Implementing Virtual Memory

There are basically three approaches to implementing virtual memory: Paging, segmentation, and a combination of the two called paged segmentation.

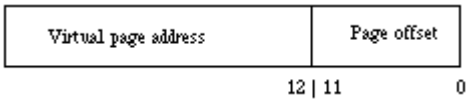
3.5.1 Paging memory management:

Memory is divided into fixed-size blocks called pages. Main memory contains some number of pages which is smaller than the number of pages in the virtual memory. For example, if the page size is 2K and the physical memory is 16M (8K pages) and the virtual memory is 4G (2 M pages) then there is a factor of 254 to 1 mapping. A page map table is used for implementing a mapping, with one entry per virtual page. The entry of a page table is as following:



The Presence Bit indicates that page is in the main memory substitutes the physical page from the table for the virtual page portion of the address. When the presence bit indicates that the page is not in main memory, it triggers a page fault exception, and the operating System (OS) initiate the disk transfer and brings the page into memory. While a disk transfer is in progress, the OS may cause another task to execute or it may simply stall the current task and sit idle. In either case, once the transfer is complete, the OS stores the new physical page number into the page table and jumps back to the instruction causing the page fault so that the access can be reissued.

The secondary storage address is used to locate the data on disk. Physical page address is substituted for the virtual page address as a result of the lookup. The virtual address from the CPU is split into the offset and page number. The page number is the index of the table from where data to be fetched. A virtual address takes the following form for a page of 4K words:



Thus, any virtual page may be stored at any physical page location and the addressing hardware performs the translation automatically.

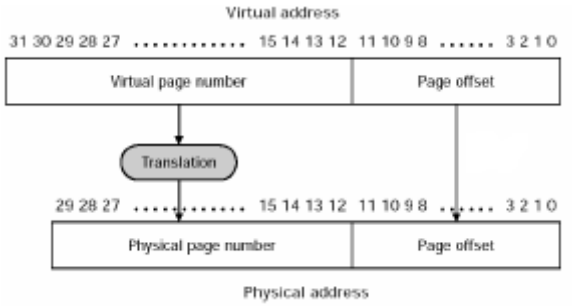


Figure 3.6 Translational of virtual address into physical address

The pages are not necessarily stored in contiguous memory locations, and therefore every time a memory reference occurs to a page which is not the page previously referred to, the physical address of the new page in main memory must be determined. In fact, most paged memory management systems (and segmented memory management systems as well) maintain a ``page translation table" using associative memory to allow a fast determination of the physical address in main memory corresponding to a particular virtual address. The page offset (low order 12 bits) is the location of the desired word within a page. Thus, the virtual address translation hardware appears as follows:

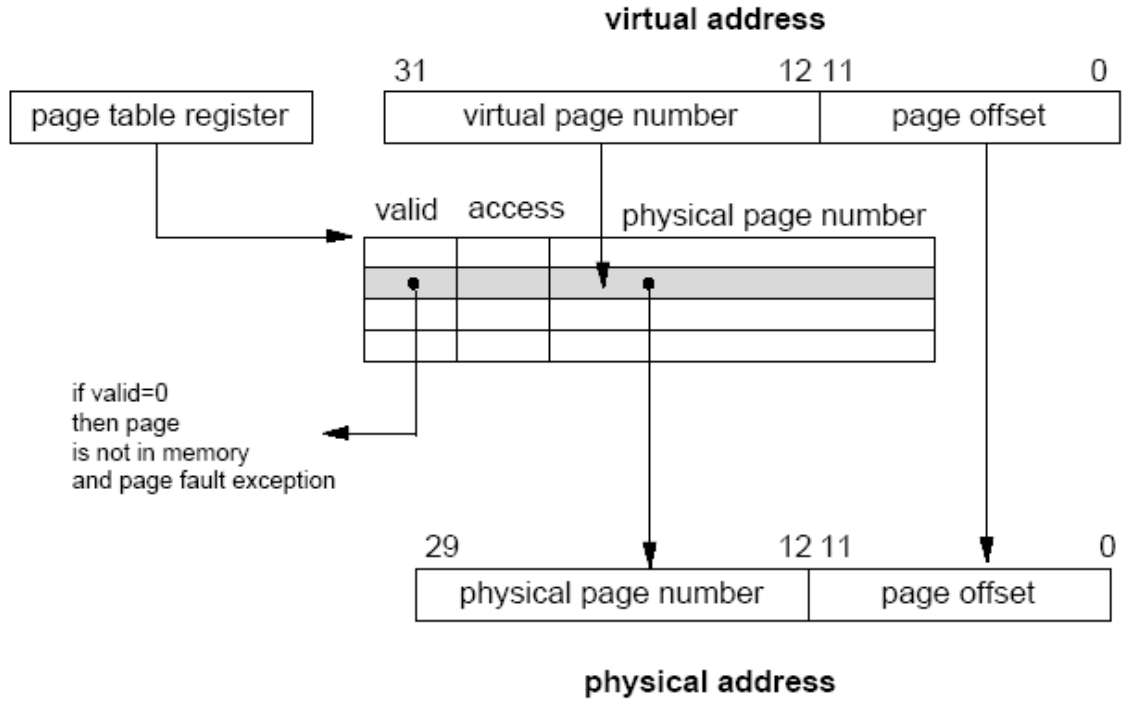


Figure 3.6 Virtual mapping technique

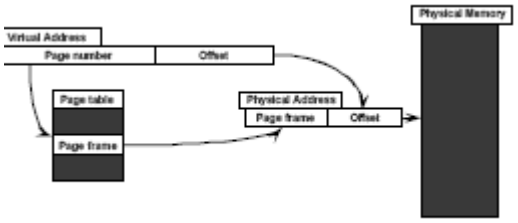


Figure 3.7 page mapping technique

The following is an example of a paged memory management configuration using a fully associative page translation table: Consider a computer system which has 16 M bytes (2^{24} bytes) of main memory, and a virtual memory space of 2^{32} bytes. Figure 3.6 shows a sketch of the page translation table required to manage all of main memory if the page size is 4K 2^{12}bytes. Note that the associative memory is 20 bits wide (32 bits - 12 bits, the virtual address size -- the page size). Also to manage 16 M bytes of memory with a page size of 4 K bytes, a total of $(16M = 2^{24})/(4K = 2^{12}) = 2^{12} = 4096$ associative memory locations are required.

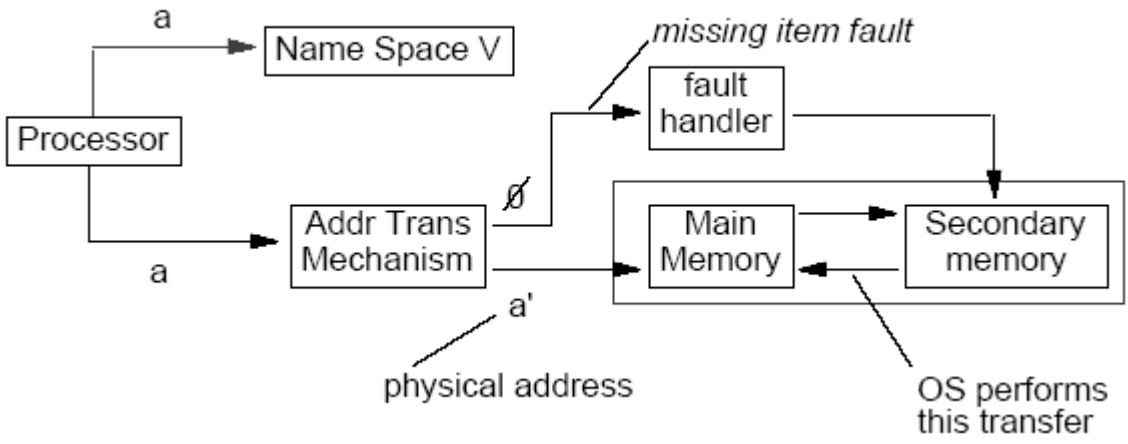


Figure 3.8 Paged memory management address translation

Some other attributes are usually included in a page translation table as well, by adding extra fields to the table. For example, pages or segments may be characterized as read only, read-write, *etc.* Another common information included is the access privileges, that ensures that no program inadvertently corrupt data for another program. It is also usual to have a bit (the ``dirty" bit) which indicates whether or not a page has been written to, so that the page will be written back onto the disk if a memory write has occurred into that page.

There is a kind of trade-off between the page size for a system and the size of the page translation table (PTT). If a processor has a small page size, then the PTT must be quite large to map all of the virtual memory space. For example, if a processor has a 32 bit virtual memory address, and a page size of 512 bytes (2^9bytes), then there are 2^{23} possible page table entries. If the page size is increased to 4 Kbytes (2^{12}bytes), then the PTT requires ``only" 2^{20}, or 1 M page table entries. These large page tables will normally not be very full, since the number of entries is limited to the amount of physical memory available. One way these large, sparse PTT's are managed is by mapping the PTT itself into virtual memory.

One problem found in virtual memory systems, particularly paged memory systems, is that when there are a large number of processes executing ``simultaneously" as in a multiuser system, the main memory may contain only a few pages for each

process, and all processes may have only enough code and data in main memory to execute for a very short time before a page fault occurs. This situation, often called ``thrashing," severely degrades the throughput of the processor because it actually must spend time waiting for information to be read from or written to the disk.

Pages are usually loaded in this manner, which is called "on demand". However, schemes have also been devised in which the historical behavior of a task is recorded, and when the task is suspended, its working set of pages is reloaded before it restarts. Once main memory pages are all allocated to virtual pages, and then any accesses to additional virtual pages force pages to be replaced (or "swapped") using an appropriate replacement policy.

3.5.2 Segmented memory management

In a segmented memory management system the blocks to be replaced in main memory are potentially of unequal length and here the segments correspond to logical blocks of code or data for example, a subroutine or procedure. Segments, then, are ``atomic," in the sense that either the whole segment should be in main memory, or none of the segment should be there. The segments may be placed anywhere in main memory, but the instructions or data in one segment should be contiguous, as shown in Figure 3.9.

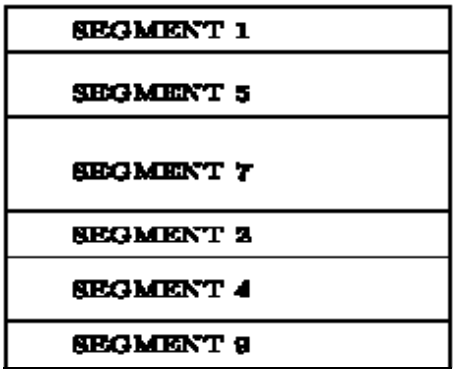


Figure 3.9: A segmented memory organization

Segmentation is implemented in a manner much like paging, through a lookup table. The difference is that each segment descriptor in the table contains the base address of the segment and a length. Each process can be assigned a different segment, and so it is completely unaware that other processes are sharing the memory with it. Relocation is effectively done dynamically at run time by the virtual memory mapping mechanism.

The maximum number of segments is typically small compared to the virtual address range (for example 256), in order to keep the size of the segment tables small

A virtual address in the segmentation scheme consists of a segment number and a segment offset. A segment descriptor also typically carries some protection information, such as the read/write permission of the segment and the process ID. It will also have some housekeeping information such as a presence bit and dirty bit. In a pure segmentation scheme the segment offset field grows and shrinks (logically) depending on the length of the segment.

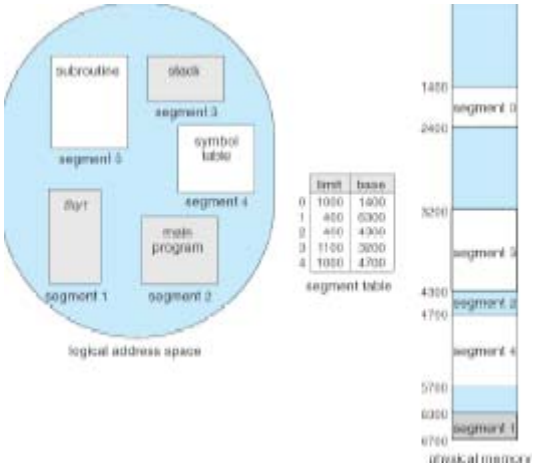


Figure 3.10 Memory allocation using segmentation technique

When segments are replaced, it can only be replaced by a segment of the same size, or by a smaller segment. Each swap may results in a ``memory fragmentation", with many small segments residing in memory, having small gaps between them this is called external fragmentation. The allocation and deallocation of space for processes with segments of different sizes leaves a collection of holes that are too small for a new process to be fit into, yet there may be more than enough empty memory if all of the holes were collected into one space This situation is also known as checkerboarding. External fragmentation is addressed through a process known as compaction in which all active processes are temporarily suspended and then relocated in order to gather together all of the memory holes. Compaction is a costly process, especially in machines that have large amounts of physical memory. Because the main memory is being completely rearranged, many memory access operations are required, and few of them can take any advantage of the cache. Segmentation also suffers from the large unbroken nature of a

segment. It is very costly to swap processes because an entire segment must be written out.

This organization appears to be efficient if two processes to share the same code in a segmented memory system for example a same procedure is used by two processes concurrently, there need only be a single copy of the code segment in memory. Although segmented memory management is not as popular as paged memory management, but most processors which presently claim to support segmented memory management actually support a hybrid of paged and segmented memory management, where the segments consist of multiples of fixed size blocks.

3.5.4 Paged Segmentation

Both paged and segmented memory management provide the users of a computer system with all the advantages of a large virtual address space. The principal advantage of the paged memory management system over the segmented memory management system is that it is simpler to implement. Also, the paged memory management does not suffer from external fragmentation in the same way as segmented memory management. Although internal fragmentation does occur i.e., the fragmentation is within a page. As whole page is swapped in or out of memory, even if it is not full of data or instructions. Paged memory management is really a special case of segmented memory management. In the case of paged memory management, all of the segments are exactly the same size (typically 256 bytes to 16 K bytes) virtual ``pages" in auxiliary storage (disk) are mapped into fixed page-sized blocks of main memory with predetermined page boundaries. The pages do not necessarily correspond to complete functional blocks or data elements, as is the case with segmented memory management. As with segmentation, the logical size of the page number portion of the address grows and shrinks according to the size of the segment.

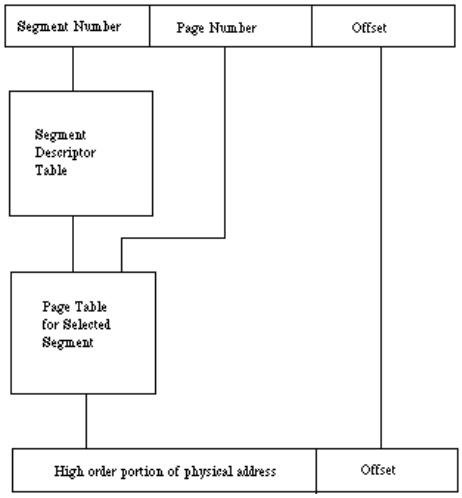


Figure 3.11 page segmented translation

To facilitate the use of memory hierarchies, the memory addresses normally generated by modern processors executing application programs are not physical addresses, but are rather virtual addresses of data items and instructions.

Physical addresses, of course, are used to reference the available locations in the real physical memory of a system.

Virtual addresses must be mapped to physical addresses before they can be used.

3.5.3 page replacement algorithms

If the size of the process is of ten pages actually only five of them will be currently in use. According to principle of demand paging only five pages are loaded as memory requirement for each page is less so load more number of program in memory or we can say degree of multiprogramming increase. As only some part of the program that is needed for execution is in memory. Thus logical address space which is address generated by CPU and correspond to complete program can therefore be much larger than physical address space. If a page is required and not present in main memory a page fault occur which result a hard trap. The operating system determines where the desired page resides in the hard disk and load into main memory. What will happens if there is no free frame to load the page. As the page miss result a interrupt which reduce the system efficiency and hence it is advisable to design a algorithm to reduce the frequency of page fault. Although inclusive of this algorithm result in increase in complexity but improve

overall efficiency increase. The page algorithm finds a page that is not use or not will be used in near future and swaps it with page required by memory. The aim of any page replacement is to result minimum of page fault. There are two classes of replacement schemes;

a. fixed replacement schemes : in these algorithm the number of pages allocated to a in process is fixed

b. variable replacement schemes, in these algorithm the number of pages available for a process varies during whole life cycle of the process.

Now we should modify the page fault handling technique to incorporate page replacement algorithm as follows

1. Find the desired page on the hard disk

2. Find a free frame to load the page

a. if frame available use it

b. If no frame is free select a page using page replacement algorithm to be swapped to the hard disk

c. Write the victim page on hard disk and load the required page to main memory and make change in frame table accordingly.

3. restart the processor.

In this section we will study various page replacement algorithms. The aim of designing the replacement algorithm is minimize the page fault. We study the system of pure demand paging where each page is brought to main memory only if it is required. We evaluate an algorithm by calculating the page fault for particular memory reference sequences also called reference string. Let us assume the reference string be

0,1,2,3,1,0,1,4,0,1,2,3,4,6,1

Let there are 3 frames in the system i.e., 3 pages can be in the memory at a time per process.

The first and the simplest algorithm is First In First out (FIFO) algorithm. The algorithm suggests that the oldest page should be removed. The concept is implemented through a fixed size queue, thus a page is inserted at the tail in queue and all elements should move one step such that the head of the queue is taken out. Lets see how many page fault are reported for page replacement algorithm

Reference string

0	1	2	3	1	0	1	4	0	1	2	3	4	6	1
0	0	0	3		3	3	4			4	4		6	6
	1	1	1		0	0	0			2	2		2	1
		2	2		2	1	1			1	3		3	3
1	2	3	4		5	6	7			8	9		10	11

The last row show the page fault

Let now consider a case there are 4 frames available for pages. The obvious observation should be that if we increase the number of pages the page fault should reduce.

Reference string

0	1	2	3	1	0	1	4	0	1	2	3	4	6	1
0	0	0	0				4	4	4	4	3	3	3	3
	1	1	1				1	0	0	0	0	4	4	4
		2	2				2	2	1	1	1	1	6	6
			3				3	3	3	2	2	2	2	1
1	2	3	4				5	6	7	8	9	10	11	12

But from above table we see that number of page fault has been increased from 11 to 12 on increasing the available frames. This is called BELADY ANOMALY : For some page replacement scheme the page fault rate increase with increase in frame number. However we should not choose the algorithm which has belady anomaly.

Optimal Page replacement algorithm

This algorithm is based on the principle that replaces the page that will not be used for longest time period.

Reference string

0	1	2	3	1	0	1	4	0	1	2	3	4	6	1
0	0	0	0				0			2	3		6	
	1	1	1				1			1	1		1	
		2	3				4			4	4		4	
1	2	3	4				5			6	7		8	

The optimal algorithm gives the minimum number of page fault but at the same time it is difficult to implement as it is not possible to have complete knowledge of reference string in advance i.e., it is difficult to project which page will referred next. Thus optimal algorithms give best result but are theoretical concept.

Random page replacement

Random replacement algorithm will select the page to be replaced randomly from the pages present in main memory. It is very simple technique and does not require any additional hardware overhead but its performance is unpredictable as there is no strategy adopted for choosing a page to be replaced from the memory.

LRU Page replacement

This algorithm is based on the principle to reference of locality which states that if a page is used in recent past there are good chances that it will be used in near future. Thus for choosing a page for replacement the choice will be the page that has not been used for the longest period of time.

Reference string

0	1	2	3	1	0	1	4	0	1	2	3	4	6	1
0	0	0	3		3		4			2	2	2	6	6
	1	1	1		1		1			1	1	4	4	4
		2	2		0		0			0	3	3	3	1
1	2	3	4		5		6			7	8	9	10	11

Thus number of page fault is 11. LRU does not suffer from Belady anomaly and considered to a good page replacement algorithm. Now the question arise how to implement it through the hardware.

3.6 Summary

Various types of processor architecture has been discussed like superscalar processor and VLIW processor RISC, CISC. Difference between RISC and CISC architecture is

CISC	RISC
Variable length Instruction	Fixed length Instruction
Variable format	Fixed field decoding
Memory operands	Load/store architecture
Complex operations	Simple operations

Superscalar processor can initiate multiple instructions in the same clock cycle; typical superscalar processor fetches and decodes several instructions at the same time. Nearly all modern microprocessors, including the Pentium, PowerPC, Alpha, and SPARC are superscalar

VLIW reduces the effort required to detect parallelism using hardware or software techniques. The main advantage of VLIW architecture is its simplicity in hardware structure and instruction set. Unfortunately, VLIW does require careful analysis of code in order to “compact” the most appropriate ”short” instructions into a VLIW word.

The mismatch, speed of CPU and main memory continues to worsen, resulting CPUs are continually held back by slow memory. The only solution to above problem is to organize the memory in hierarchy in a computer. The key idea is to improve the performance by using the principle Locality of reference, i.e., to keep the things that are used frequently up in the pyramid, and things that rarely needed to access at the lower levels. This principle can be implemented using the concept of

(a) ***cache memory*** a small block of high-speed memory placed between main memory and CPU. (b) ***Virtual memory*** where main memory acts as a read/write buffer for disk storage. The key term here is mapping. The processor generates an address that is sent through a mapping function to produce the physical address of the memory.

3.7 keywords

RISC Reduced instruction set computer;

CISC Complex instruction set computers

RAM Random Access Memory; computer memory which can be written to and read from in any order

virtual memory A system that stores portions of an *address space* that are not being actively used in some medium other than main high-speed memory, such as a disk or slower auxiliary memory medium. When a reference is made to a value not presently in main memory, the virtual memory manager must *swap* some values in main memory for the values required.

locality of reference the observation that references to memory tend to cluster.

Temporal locality refers to the observation that a particular datum or instruction, once referenced, is often referenced again in the near future.

Spatial locality refers to the observation that once a particular location is referenced, a nearby location is often referenced in the near future.

3.8 Self assessment questions

1. Define the following basic terms related to modern processor technology.

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

2. Describe the design space of modern processor families.
3. With diagrams, explain the pipelined execution of successive instructions in a base scalar processor and in two underpipelined cases.
4. Explain the architectural models of a basic scalar computer system using block diagrams.
5. Differentiate the characteristics of CISC and RISC architectures.
6. Describe the structure of a superscalar pipeline with the help of a diagram.
7. Draw and explain the block diagram of a typical superscalar RISC processor architecture consisting of an integer unit and a floating-point unit.
8. Explain the architecture of VLIW processor and its pipeline operations with the help of diagrams.

3.9 References/Suggested readings

Advance Computer architecture: Kai Hwang

Computer system architecture Morris Mano

Author: Dr. Deepti Mehrotra **Vetter: Dr. Sandeep Arya**

Lesson: Bus, cache and shared memory **Lesson No. : 04**

4.0 Objective

In this lesson we had discussed about bus that is used for interconnection between different processor. We will discussed about use of cache memory in multiprocessor environment and various addressing scheme used for cache memory. Also we will discuss how shared memory concept is used in multiprocessor. Various issues regarding event ordering specially in case of memory events that deal with shared memory creates synchronization problem we will also discuss various models designed to overcome these issues.

4.1 Introduction

We will deal with physical address caches, virtual address cache, cache implementation using direct fully associative and sector mapping. We had already studied the about the basic of memory is last lesson. In this lesson we will study about how the memory is shared between in multiprocessor and various consistency issues like atomicity event ordering and strong and weak consistency.

4.2 Backplane Buses — A backplane bus interconnects processors, data storage and peripheral devices in a tightly coupled hardware. The system bus must be designed to allow communication between devices on the devices on the bus without disturbing the internal activities of all the devices attached to the bus. These are typically `intermediate' buses, used to connect a variety of other buses to the CPU-Memory bus. They are called Backplane Buses because they are restricted to the *backplane* of the system.

4.2.1 Backplane bus specification

They are generally connected to the CPU-Memory bus by a *bus adaptor*, which handles translation between the buses. Commonly, this is integrated into the CPU-Memory bus controller logic. While these buses can be used to directly control devices, they are used as 'bridges` to other buses. For example, AGP bus devices – *i.e.* video cards – act as bridges between the CPU-Memory bus and the actual display device: the monitor(s).) For this reason, these buses are sometimes called *mezzanine buses*.

- Allow processors, memory and I/O devices to coexist on single bus
- Balance demands of processor-memory communication with demands of I/O device-memory communication
- Interconnects the circuit boards containing processor, memory and I/O interfaces an interconnection structure within the chassis
- Cost advantage: one single bus for all components
- The backplane bus is divided into four groups
- Data address and control lines form the data transfer bus (DTB) in VME bus.
- DTB Arbitration bus that provide control of DTB to requester using the arbitration logic.
- Interrupt and Synchronization bus used for handling interrupt

- Utility bus include signals that provide periodic timing and coordinate the power up and power down sequence of the system.

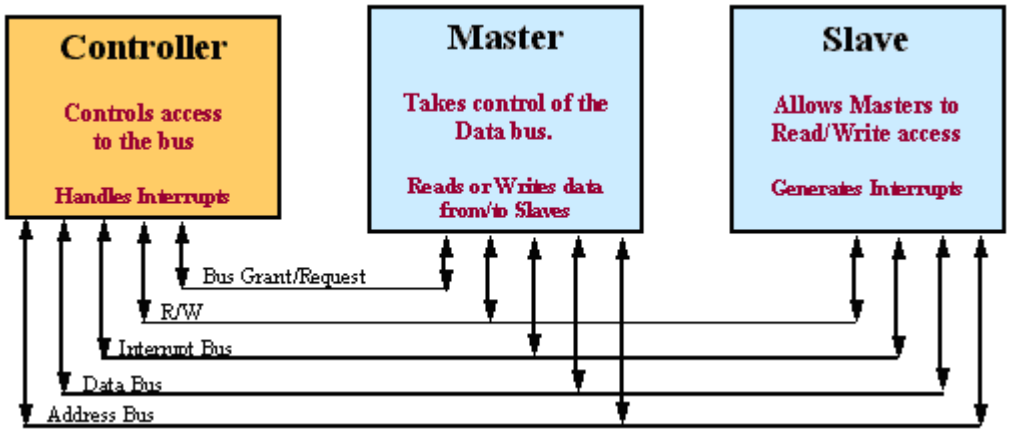


figure 4.1 **VMEbus System**

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

Functional module : A functional module is collection of electronic circuitry that reside on one functional board and works to achieve special bus control function. These functions are:

A arbitrator is a functional module that accepts bus request from the requester module and grant control of the DTB to one request at a time.

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

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

A location monitor is a functional module that monitors data transfer over the DTB.

A power monitor watches the status of the power source and signals when power 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 impedence, the propagation time and termination values between the backplane and the plug in board.

4.2.2 ASYNCHRONOUS DATA TRANSFER

All the operations in a digital system are synchronized by a clock that is generated by a pulse generator. The CPU and I/O interface can be designed independently or they can share common bus. If CPU and I/O interface share a common bus, the transfer of data between two units is said to synchronous. There are some disadvantages of synchronous data transfer, such as:

- It is not flexible, as all bus devices run on the same clock rate.
- Execution times are the multiples of clock cycles (if any operation needs 3.1 clock cycles, it will take 4 cycles).
- Bus frequency has to be adapted to slower devices. Thus, one cannot take full advantage of the faster ones.
- It is particularly not suitable for an I/O system in which the devices are comparatively much slower than processor.

In order to overcome all these problems, an asynchronous data transfer is used for input/ output system.

The word 'asynchronous' means 'not in step with the elapse of time'. In case of asynchronous data transfer, the CPU and I/O interface are independent of each other. Each uses its own internal clock to control its registers. There are two popular techniques used for such data transfer: strobe control and handshaking.

Strobe Control

In strobe control**,** a control signal, called strobe pulse, which is supplied from one unit to other, indicates that data transfer has to take place. Thus, for each data transfer, a strobe is activated either by source or destination unit. A strobe is a single control line that informs the destination unit that a valid data is available on the bus. The data bus carries the binary information from source unit to destination unit.

Data transfer from source to destination

The steps involved in data transfer from source to destination are as follows:

(i) The source unit places data on the data bus.

(ii) A source activates the strobe after a brief delay in order to ensure that data values are steadily placed on the data bus.

(iii) The information on data bus and strobe signal remain active for some time that is sufficient for the destination to receive it.

(iv) After this time the sources remove the data and disable the strobe pulse, indicating that data bus does not contain the valid data.

(v) Once new data is available, strobe is enabled again.

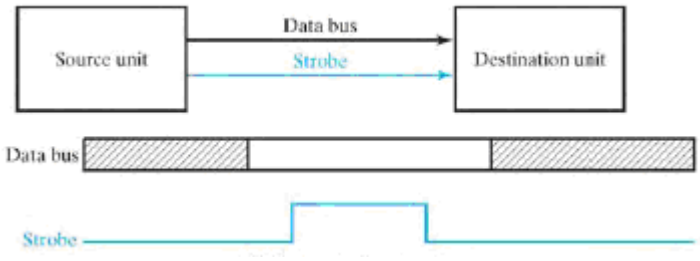


Figure 4.2 *Source- Initiated Strobe for Data Transfer*

Data transfer from destination to source

The steps involved in data transfer from destination to source are as follows:

1. The destination unit activates the strobe pulse informing the source to provide the data.
2. The source provides the data by placing the data on the data bus.

(iii) Data remains valid for some time so that the destination can receive it.

(iv) The falling edge of strobe triggers the destination register.

(v) The destination register removes the data from the data bus and disables the strobe.

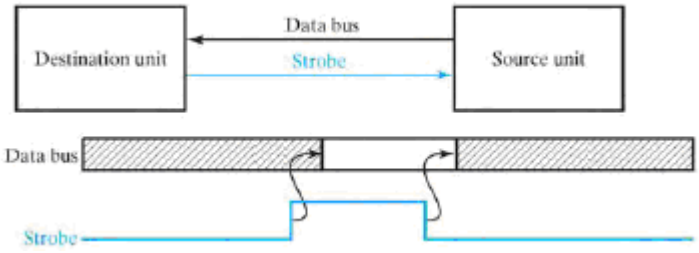


Figure 4.3 *Destination- Initiated Strobe for Data Transfer*

The disadvantage of this scheme is that there is no surety that destination has received the data before source removes the data. Also, destination unit initiates the transfer without knowing whether source has placed data on the data bus.

Thus, another technique, known as handshaking, is designed to overcome these drawbacks.

Handshaking

The handshaking technique has one more control signal for acknowledgement that is used for intimation. As in strobe control, in this technique also, one control line is in the same direction as data flow, telling about the validity of data. Other control line is in reverse direction telling whether destination has accepted the data.

Data transfer from source to destination

In this case, there are two control lines as shown in Figure 3.13: request and reply. The sequence of actions taken is as follows

(i) Source initiates the data transfer by placing the data on data bus and enable request signal.

(ii) Destination accepts the data from the bus and enables the reply signal.

(iii) As soon as source receives the reply, it disables the request signal. This also invalidates the data on the bus.

(iv) Source cannot send new data until destination disables the reply signal.

(v) Once destination disables the reply signal, it is ready to accept new signal.

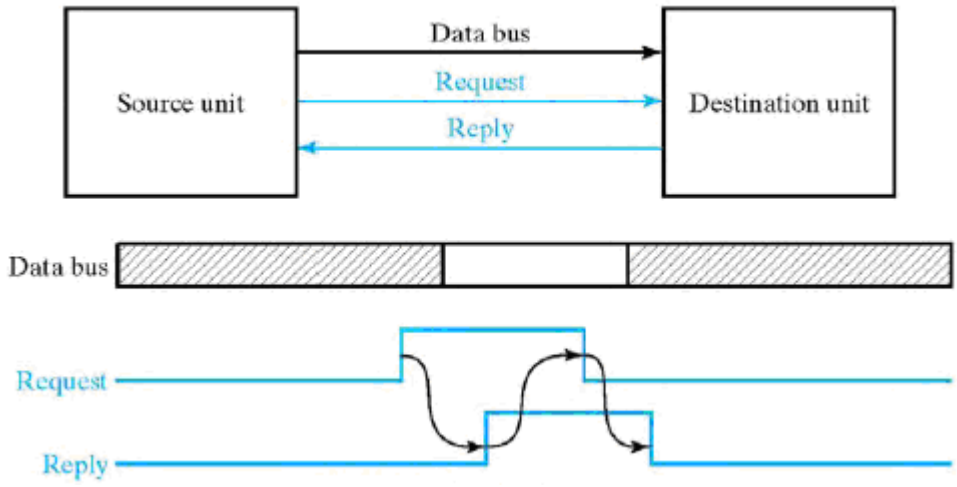


Figure 4.5 *Source-Initiated Data Transfer Using Handshaking Technique*

Data transfer from destination to source

The steps taken for data transfer from destination to source are as follows:

(i) Destination initiates the data transfer sending a request to source to send data telling the latter that it is ready to accept data.

(ii) Source on receiving request places data on data bus.

(iii) Also, source sends a reply to destination telling that it has placed the requisite data on the data bus and has disabled the request signal so that destination does not have new request until it has accepted the data.

(iv) After accepting the data, destination disables the reply signal so that it can issue a fresh request for data.

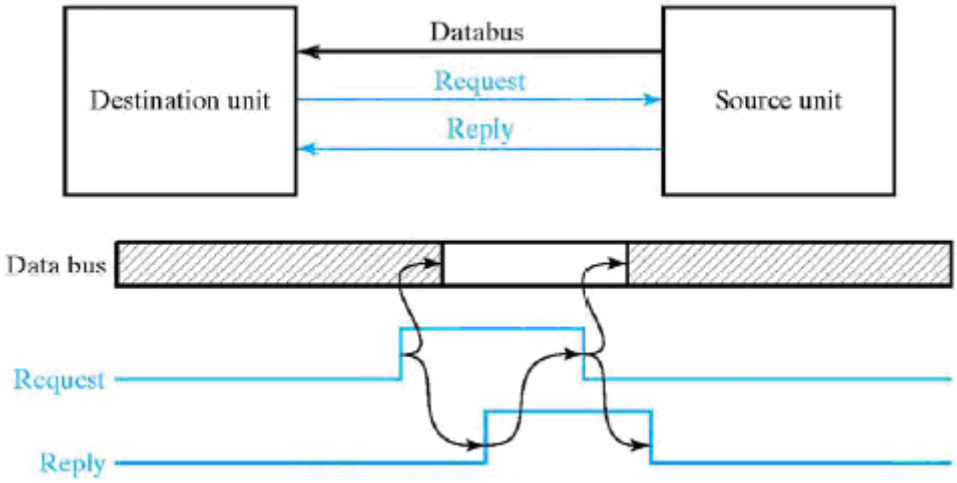


Figure 4.6 *Destination-Initiated Data Transfer Using Handshaking Technique*

Advantage of asynchronous bus transaction

• It is not clocked.

• It can accommodate a wide range of devices.

4.2.3 Bus Arbitration

Since at a unit time only one device can transmit over the bus, hence one important issue is to decide who should access the bus. Bus arbitration is the process of determining the bus master who has the bus control at a given time when there is a request for bus from one or more devices.

Devices connected to a bus can be of two kinds:

1. Master: is active and can initiate a bus transfer.
2. Slave: is passive and waits for requests.

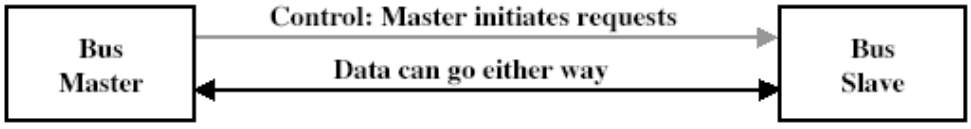


Figure 4.7 Bus arbitration

In most computers, the processor and DMA controller are bus masters whereas memory and I/O controllers are slaves. In some systems, certain intelligent I/O controllers are also bus masters. Some devices can act both as master and as slave, depending on the circumstances:

• CPU is typically a master. A coprocessor, however, can initiate a transfer of a parameter from the CPU here CPU acts like a slave.

• An I/O device usually acts like a slave in interaction with the CPU. Several devices can perform direct access to the memory, in which case they access the bus like a master.

• The memory acts only like a slave.

In some systems especially one where multiple processors share a bus. When more than one bus master simultaneously needs the bus, only one of them gains control of the bus and become active bus master. The others should wait for their turn. The 'bus arbiter' decides who would become current bus master.Bus arbitration schemes usually try to balance two factors:

• Bus priority: the highest priority device should be serviced first

• Fairness: Even the lowest priority device should never be completely locked out from the bus

Lets understand the sequence of events take place, where the bus arbitration consists are following:

1. Asserting a bus mastership request
2. Receiving a grant indicating that the bus is available at the end of the current cycle. A bus master cannot use the bus until its request is granted
3. Acknowledging that mastership has been assumed
4. A bus master must signal to the arbiter after finish using the bus

The 68000 has three bus arbitration control pins:

BR - The bus request signal assigned by the device to the processor intending to use the buses.

BG - The bus grant signal is assigned by the processor in response to a BR, indicating that the bus will be released at the end of the current bus cycle. When BG is asserted BR can be de-asserted. BG can be routed through a bus arbitrator e.g. using daisy chain or through a specific priority-encoded circuit.

BGACK - At the end of the current bus cycle the potential bus master takes control of the system buses and asserts a bus grant acknowledge signal to inform the old bus master that it is now controlling the buses. This signal should not be asserted until the following conditions are met:

1. A bus grant has been received
2. Address strobe is inactive, which indicates that the microprocessor is not using the bus
3. Data transfer acknowledge is inactive, which indicates that neither memory nor peripherals are using the bus
4. Bus grant acknowledge is inactive, which indicates that no other device is still claiming bus mastership.

On a typical I/O bus, however, there may be multiple potential masters and there is a need to arbitrate between simultaneous requests to use the bus. The arbitration can be either central or distributed.

Centralized bus arbitration in which a dedicated arbiter has the role of bus arbitration.

In the central scheme, it is assumed that there is a single device (usually the CPU) that has the arbitration hardware. The central arbiter can determine priorities and can force termination of a transaction if necessary. Central arbitration is simpler and lower in cost for a uniprocessor system. It does not work as well for a symmetric multiprocessor design unless the arbiter is independent of the CPUs.

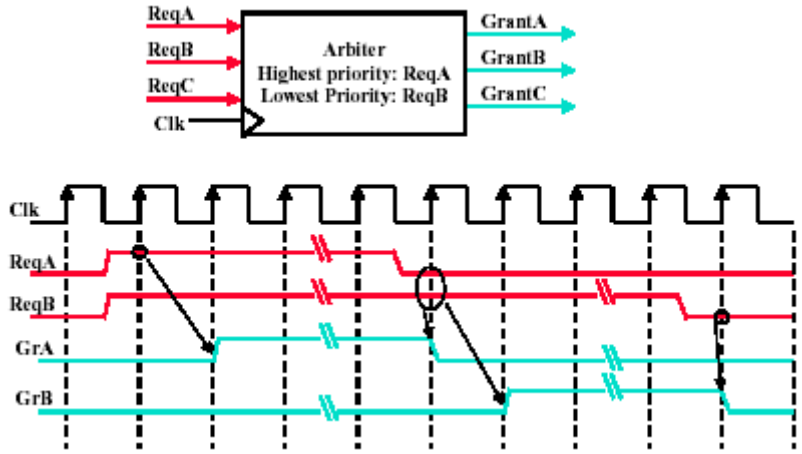
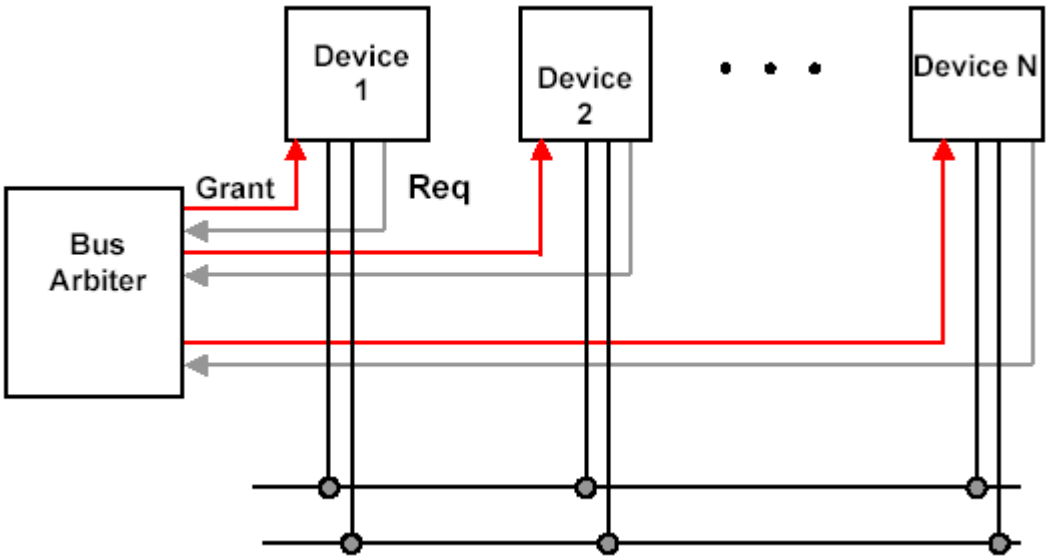


Figure 4.8 Centralized bus arbitrator



Fiure 4.9 Indepentent request with central arbitrator

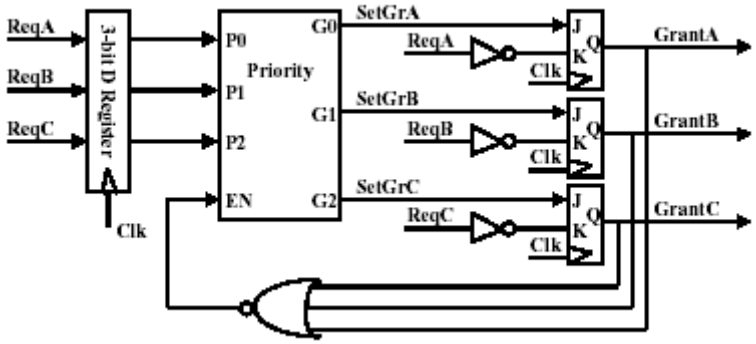


Figure 4.10 Central arbitrator

Distributed bus arbitration in which all bus masters cooperate and jointly perform the arbitration. In this case, every bus master has an arbitration section. For example: there are as many request lines on the bus as devices; each device monitors each request line after each bus cycle each device knows if he is the highest priority device which requested the busy if yes, it takes it. In the distributed scheme, every potential master carries some hardware for arbitration and all potential masters compete equally for the bus. The arbitration scheme is often based on some preassigned priorities for the devices, but these can be changed. Distributed arbitration can be done either by self-selection – where code indicates identity on the bus for example NuBus 16 devices, or by collision detection as an example Ethernet. NuBus has four arbitration lines. A candidate to bus

master asserts its arbitration level on the 4-bit open collector arbitration bus. If a competing master sees a higher level on the bus than its own level, it ceases to compete for the bus. Each potential bus master simultaneously drives and samples the bus. When one bus master has gained bus mastership and then relinquished it, it would not attempt to re-establish bus mastership until all pending bus requests have been dealt with (fairness).

Daisy-chaining technique is a hybrid of central and distributed arbitration. In this techniques all devices that can request are attached serially. The central arbiter issues grant signal to the closest device requesting it. Devices request the bus by passing a signal to their neighbors who are closer to the central arbiter. If a closer device also requests the bus, then the request from the more distant device is blocked i.e., the priority scheme is fixed by the device's physical position on the bus, and cannot be changed in software. Sometimes, multiple request and grant lines are used with daisy-chaining to enable requests from devices to bypass a closer device, and thereby implement a restricted software- controllable priority scheme. Daisy-chaining is low cost technique and also susceptible to faults. It is may lead to starvation for distant devices if a high priority devices (one nearest to arbitrator) frequently request for Bus.

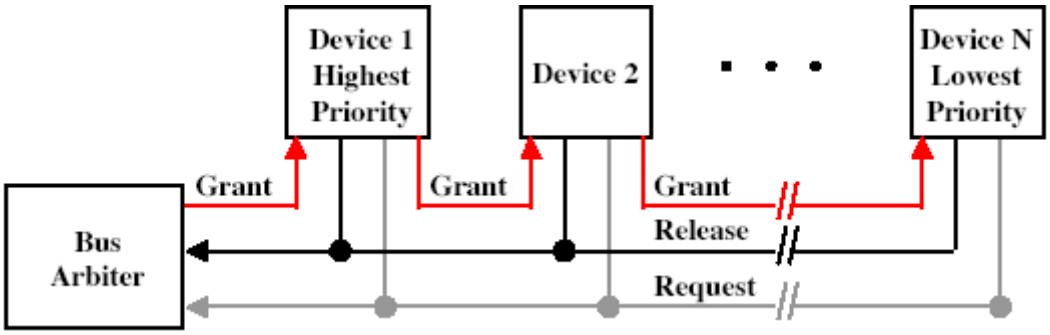


Figure 4.11 Daisy chaining arbitration scheme.

Polling

Polling is technique that identifies the highest priority resource by means of software. The program that takes care of interrupt begins at the branch address and poll the interrupt source in sequence. The priority is determined in the order in which each interrupt is entered. Thus highest priority resource is first tested if interrupt signal is on the control branch to the service routine other wise the source having next- lower-priority will be tested and so on.

Disadvantage : The disadvantage of polling is that if there are many interrupts, the tome required to poll exceed the time available to service the I/O device. To overcome this problem hardware interrupt unit can be used to speed up the operation. The hardware unit accepts the interrupt request and issue the interrupt grant to the device having highest priority. As no polling is required all decision are done by hardware unit, each interrupt source has its own interrupt vector to access its own service routine. This hardware unit can establish priority either by a serial or a parallel connection of interrupt lines.

4.3 Cache memory organization

Cache memory are the fast memory that lies between registers and RAM in memory hierarchy. It holds recently used data and/or instructions and has a size from few kB to several MB.

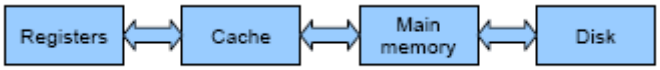


Figure 4.12 Memory structure for a processor

4.3.1Cache addressing models

The figure 4.13 shows a cache and main memory structure. A cache consists of C slots and each *slot* in the cache can hold K memory words. Here the main memory with 2^n-1 words i.e., M words with each having a unique n-bit address and cache memory having C*K words where K is the Block size and C are the number of lines. Each word that resides in the cache is a subset of main memory. Since there are more blocks in main memory than number of lines in cache, an individual line cannot be uniquely and permanently dedicated to a particular block. Therefore, each line includes a tag that identifies which particular block of main memory is currently occupying that line of cache. The tag is usually a portion of the main memory address. The cache memory is accessed but by pattern matching on a *tag* stored in the cache.

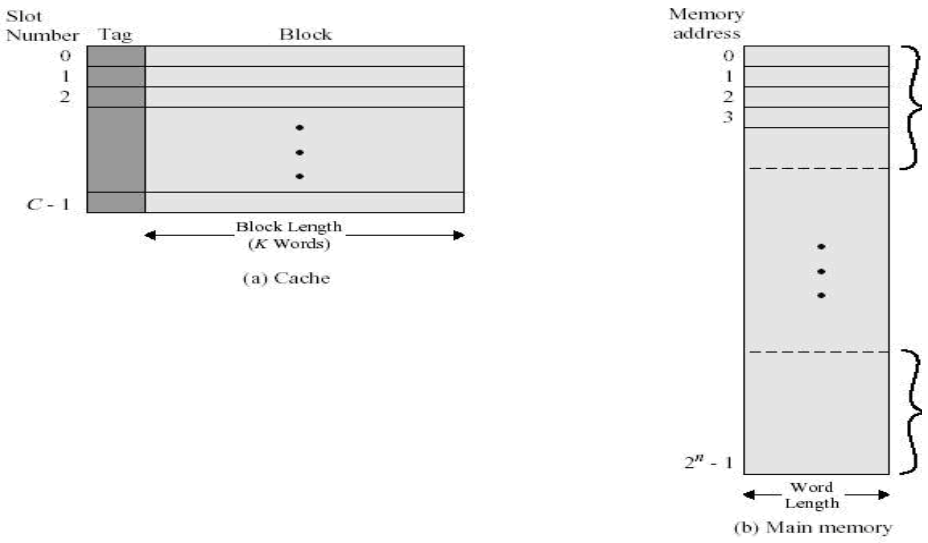


Figure 4.13 Cache / Main memory structure

For the doing comparison of address generated by CPU the memory controller use some algorithm which determines whether the value currently being addressed in memory is available in the cache. The transformation data from main memory to cache memory is referred as a mapping process. Let us derive an address translation scheme using cache as a linear array of entries, each entry having the following structure as shown in figure below:

Data - The block of data from memory that is stored in a specific line in the cache

Tag - A small field of length K bits, used for comparison, to check the correct address of data

Valid Bit - A one-bit field that indicates status of data written into the cache.

The N-bit address is produced by the processor to access cache data is divided into three fields:

Tag - A K-bit field that corresponds to the K-bit tag field in each cache entry,

Index - An M-bit field in the middle of the address that points to one cache entry

Byte Offset – L Bits that finds particular data in a line if valid cache is found.

It follows that the length of the virtual address is given by N = K + M + L bits.

Cache Address Translation. As shown in Figure 4.14, we assume that the cache address has length 42 bits. Here, bits 12-41 are occupied by the Tag field, bits 2-11 contain the Index field, and bits 0,1 contain the Offset information. The index points to the line in cache that supposedly contains the data requested by the processor. After the cache line is

retrieved, the Tag field in the cache line is compared with the Tag field in the cache address. If the tags do not match, then a cache miss is detected and the comparator outputs a zero value. Otherwise, the comparator outputs a one, which is *and*-ed with the valid bit in the cache row pointed to by the Index field of the cache address. If the valid bit is a one, then the Hit signal output from the *and* gate is a one, and the data in the cached block is sent to the processor. Otherwise a cache miss is registered.

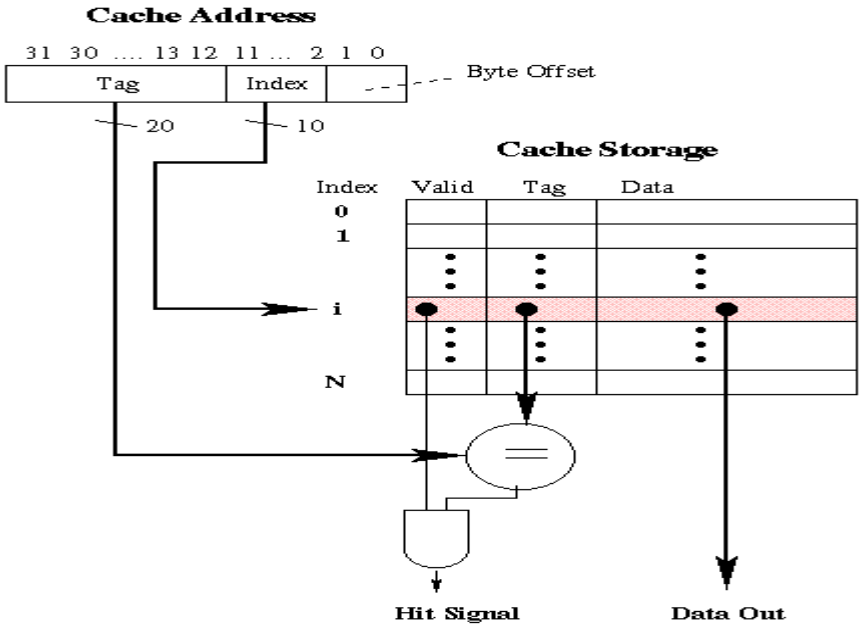


Figure 4.14. Schematic diagram of cache

Most multiprocessor system use private cache associated with different processor.

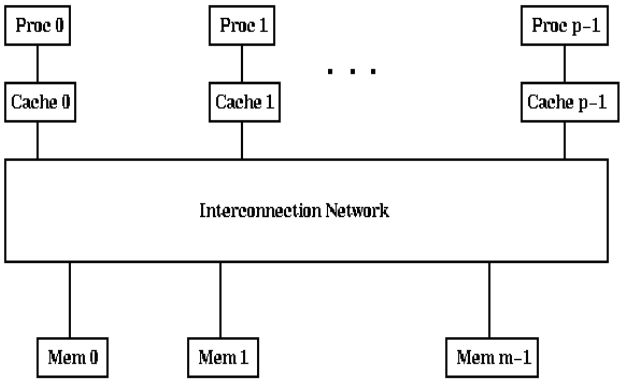


Figure 4.15 A memory hierarchy for a shared memory multiprocessor.

Cache can be addressed either by physical address or virtual address.

Physical address cache: when cache is addressed by physical address it is called physical address cache. The cache is indexed and tagged with physical address. Cache lookup must occur after address translation in TLB or MMU. No aliasing is allowed so that the address is always uniquely translated without confusion. This provide an advantage that we need no cache flushing, no aliasing problem and fewer cache bugs in OS kernel. The short coming is the slowdown in accessing the cache until the MMU/TLB finishes translating the address.

Advantage of physically addressed caches

- no cache flushing on a context switch
- no synonym problem (several different virtual addresses can span the same physical addresses : a much better hit ratio between processes)

Disadvantage of physically addressed caches

- do virtual-to-physical address translation on every access
- increase in hit time because must translate the virtual address before access the cache

Virtual Address caches: when a cache is indexed or tagged with virtual address it is called 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.

Advantage of virtually-addressed caches

- do address translation only on a cache miss
- faster for hits because no address translation

Disadvantage of virtually-addressed caches

cache flushing on a context switch (example : local data segments will get an erroneous hit for virtual addresses already cached after changing virtual address space, if no cache flushing).

synonym problem (several different virtual addresses cannot span the same physical addresses without being duplicated in cache).

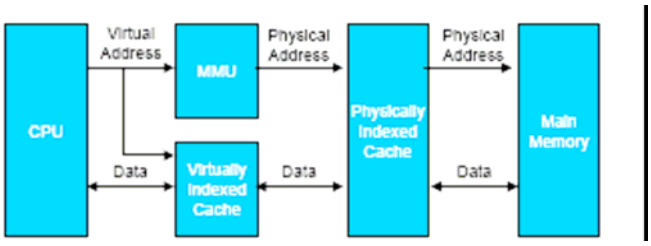


Figure 4.16 Virtual address for cache

Aliasing: The major problem with cache organization in multiprocessor is that multiple virtual addresses can map to a single physical address i.e., different virtual address cache logically addressed data have the same index/tag in the cache. Most processors guarantee that all updates to that single physical address will happen in program order. To deliver on that guarantee, the processor must ensure that only one copy of a physical address resides in the cache at any given time.

4.3.2 Direct-Mapped Caches

The easiest way of organizing a cache memory employs direct mapping that is based on a simple algorithm to map data block i from the main memory into data block j in the cache. There is a one-to-one correspondence between each block of data in the cache and each memory block thus to find a memory block i, then there is one and only one place in the cache where i is stored

If we have 2^n words in main memory and 2^k words in cache memory. In cache memory each word consists of data word and its associated tag. The n-bit memory address is divided into three fields : low order k bits are referred as the index field and used to address a word in the cache. The remaining n-k high-order bits are called the *tag.* The index field is further divided into the *slot* field, which will be used find a particular slot in the cache; and the offset field is used to identify a particular memory word in the slot. When a block is stored in the cache, its *tag* field is stored in the *tag field* of the cache slot.

When CPU generates an address the index field is used to access the cache. The tag field of CPU address is compared with the tag in word read from the cache. If the two tags match, there is a hit and else there is a miss and the required word is read from main memory. Whenever a ``cache miss'' occurs, the cache line will be replaced by a new line

of information from main memory at an address with the same index but with a different tag.

Lets us understand how direct mapping is implemented with following simple example Figure 4.17. The memory is composed of 32 words and accessed by a 5-bit address. Let the address has a 2-bit tag (set) field, a 2-bit slot (line) field and a 1-bit word field. The cache memory holds $2^2 = 4$ lines each having two words. When the processor generates an address, the appropriate line (slot) in the cache is accessed. For example, if the processor generates the 5-bit address 11110_2, line 4 in set 4 is accessed. The memory space is divided into sets and the sets into lines. The Figure 4.14 reveals that there are four possible lines that can occupy cache line 4 lines 4 in set 0, in set 1, in set 2 and set 4. In this example the processor accessed line 4 in set 4. Now “How does the system resolve this issue?"

Figure 4.14 shows how a direct mapped cache resolves the contention between lines. Each line in the cache memory has a tag or label that identifies which set this particular line belongs to. When the processor accesses line 4, the tag belonging to line 4 in the cache is sent to a comparator. At the same time the set field from the processor is also sent to the comparator. If they are the same, the line in the cache is the desired line and a hit occurs. If they are not the same, a miss occurs and the cache must be updated. Figure 4.17 provides a skeleton structure of a direct mapped cache memory system.

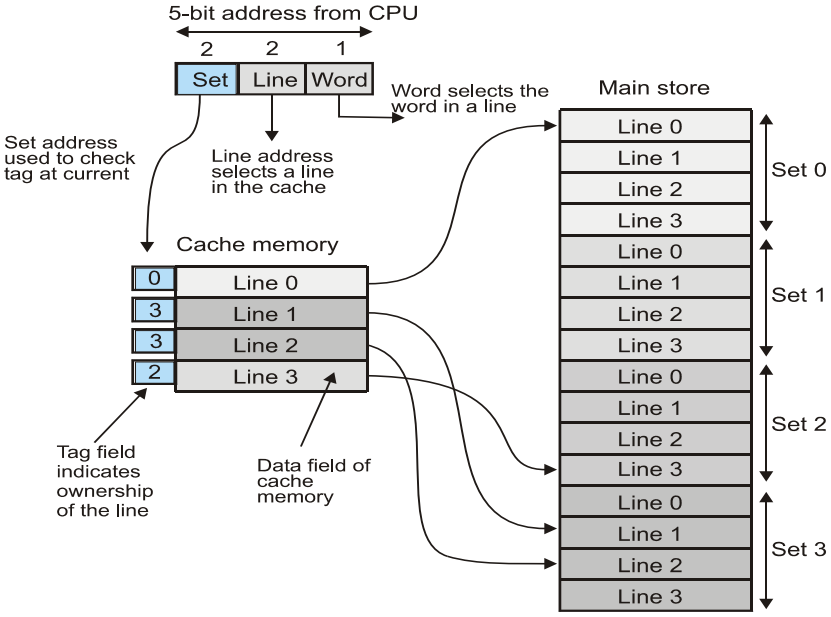


Figure 4.17 Resolving contention between lines in a direct-mapped cache

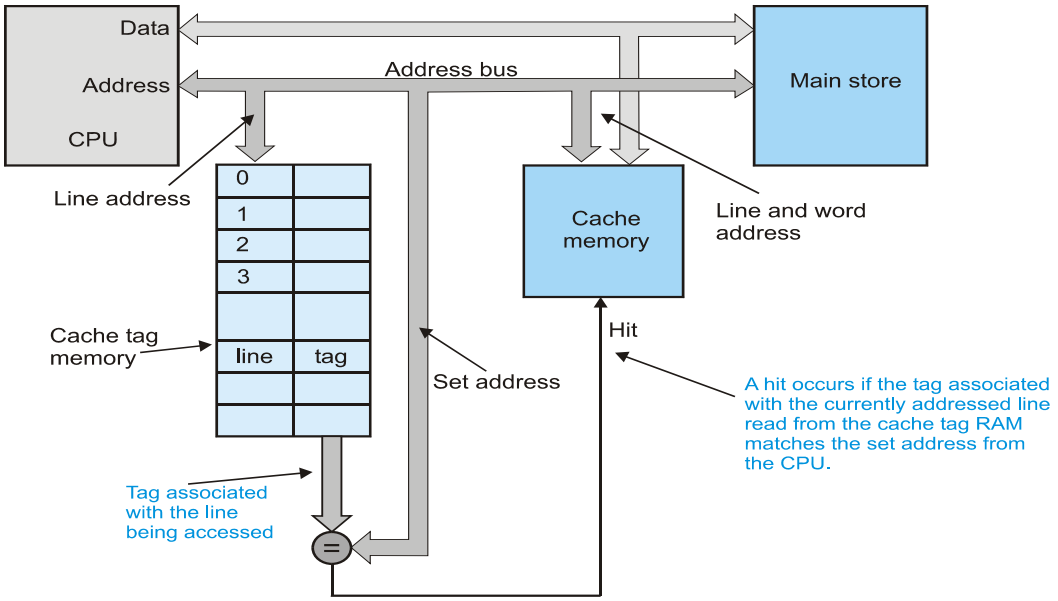


Figure 4.18 Implementation of direct-mapped cache

The advantage of direct mapping are as follows

It's simplicity.

Both the cache memory and the cache tag RAM are widely available devices.

The direct mapped cache requires no complex line replacement algorithm. If line *x* in set *y* is accessed and a miss takes place, line *x* from set *y* in the main store is loaded into the frame for line *x* in the cache memory and the tag set to *y i.e.*,, there is no decision to be taken regarding which line has to be rejected when a new line is to be loaded.

It inherents parallelism. Since the cache memory holding the data and the cache tag RAM are entirely independent, they can both be accessed simultaneously. Once the tag has been matched and a hit has occurred, the data from the cache will also be valid.

The disadvantage of direct mapping are as follows

it is inflexible

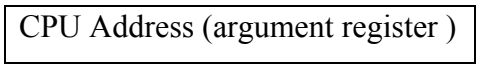
A cache has one restriction *a particular memory address can be mapped into only one cache location also,* all addresses with the same index field are mapped to the same cache location. Consider the following fragment of code:

```
REPEAT
        Get_data
        Compare
UNTIL match OR end_of_data
```

Let the Get data routine and compare routine use two blocks, both these blocks have same index but have different tags are repeated accessed. Consequently, the performance of a direct-mapped cache can be very poor under above circumstances. However, statistical measurements on real programs indicate that the very poor worst-case behavior of direct-mapped caches has no significant impact on their average behavior.

4.3.3 Associative Mapping:

One way of organizing a cache memory which overcomes the limitations of direct mapped cache such that there is no restriction on what data it can contain can be done with associative cache memory. An associative memory is the fastest and most flexible way of cache organization. It stores both the address and the value (data) from main memory in the cache. An associative memory has an *n*-bit input. An address from the processor is divided into three fields: the tag, the line, and the word.The mapping is done with storing tag information in n-bit argument register and comparing it with address tag in each location simultaneously. If the input tag matches a stored tag, the data associated with that location is output. Otherwise the associative memory produces a miss output. Unfortunately, large associative memories are not yet cost-effective. Once the associative cache is full, a new line can be brought in only by overwriting an existing line that requires a suitable line replacement policy. Associative cache memories are efficient because they place no restriction on the data they hold, as permits any location of cache to store any word from main memory.



Address	Data
01101001	10010100
10010001	10101010

Figure 4.19 Associative cache

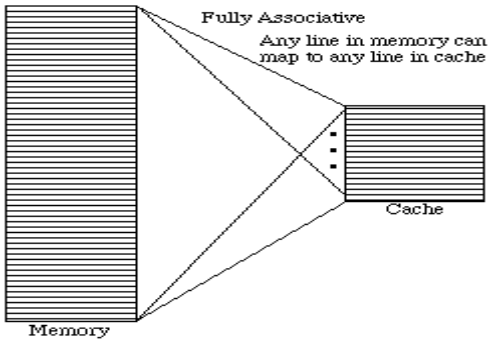


Figure 4.20Associative mapping

All of the comparisons are done simultaneously, so the search is performed very quickly. This type of memory is very expensive, because each memory location must have both a comparator and a storage element. Like the direct mapped cache, the smallest unit of data transferred into and out of the cache is the line. Unlike the direct-mapped cache, there's no relationship between the location of lines in the cache and lines in the main memory.

When the processor generates an address, the word bits select a word location in both the main memory and the cache. The tag resolves which of the lines is actually present. In an associative cache any of the 64K lines in the main store can be located in any of the lines in the cache. Consequently, the associative cache requires a 16-bit tag to identify one of the 2^{16} lines from the main memory. Because the cache's lines are not ordered, the tags are not ordered, it may be anywhere in the cache or it may not be in the cache.

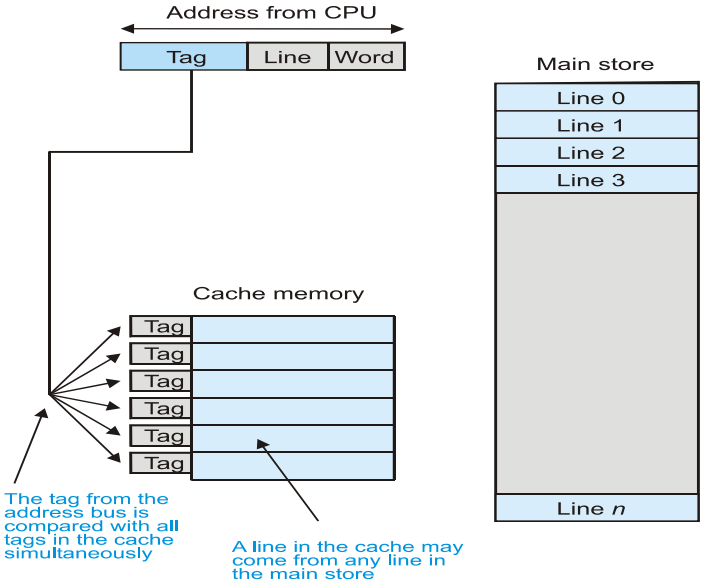


Figure 4.21 Associative-mapped cache

4.3.4 Set associative Mapping:

Most computers use set associative mapping technique as it is a compromise between the direct-mapped cache and the fully associative cache. In a set associative cache memory several direct-mapped caches connected in parallel. Let to find memory block b in the cache, there are n entries in the cache that can contain b we say that this type of cache is called n-*way set associative*. For example, if n = 2, then we have a two-way set associative cache. This is the simplest arrangement and consists of two direct-mapped cache memories. Thus for n parallel sets, a n-way comparison is performed in parallel against all members of the set. Usually $n = 2^k$, for k = 1, 2, 4 are chosen for a set associative cache (k = 0 corresponds to direct mapping). As n is small (typically 2 to 14), the logic required to perform the comparison is not complex. This is a widely used technique in practice (e.g. 80486 uses 4-way, P4 uses 2-way for the instruction cache, 4-way for the data cache).

Figure 4.22 describes the common 4-way set associative cache. When the processor accesses memory, the appropriate line in each of four direct-mapped caches is accessed simultaneously. Since there are four lines, a simple associative match can be used to determine which (if any) of the lines in cache are to supply the data. In figure 4.22 the hit output from each direct-mapped cache is fed to an OR gate which generates a hit if any of the caches generate a hit.

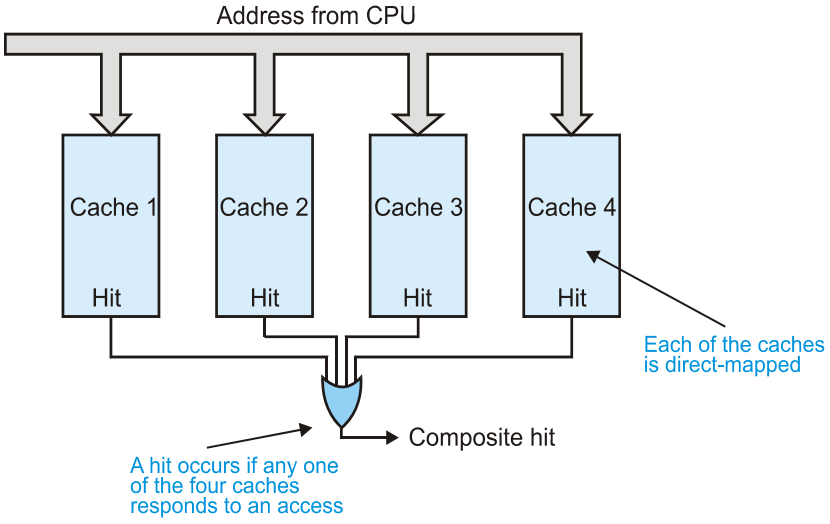


Figure 4.22 Set associative-mapped cache

4.3.4 CACHE performance Issues

As far as the performance of cache is considered the trade off exist among the cache size, set number, block size and memory speed. Important aspect in cache designing with regard to performance are :

a. **the cycle count** : This refers to the number of basic machine cycles needed for cache access, update and coherence control. This count is affected by underlying static or dynamic RAM technology, the cache organization and the cache hit ratios. The write through or write back policy also affect the cycle count. The cycle count is directly related to the hit ratio, which decreases almost linearly with increasing values of above cache parameters.
b. **Hit ratio:** The processor generates the address of a word to be read and send it to cache controller, if the word is in the cache it generates a Hit signal and also deliver it to the processor. If the data is not found in the cache, then it generates a MISS signal and that data is delivered to the processor from main memory, and simultaneously loaded into the cache. The hit ratio is number of hits divided by total number of CPU references to memory (hits plus misses). When cache size approaches
c. **Effect of Block Size:** With a fixed cache size, cache performance is sensitive to the block size. This block size is determined mainly by the temporal locality in typical program.
d. Effect of set number in set associative number.

4.4 Shared memory organization

4.4.1 Interleaved memory organization

In multiprocessor with a goal of broaden the effective memory bandwidth a technique used is ``interleaved memories”. Here several sets of data and address lines connected to independent ``banks" of memory, arranged so that adjacent memory words reside in different memory banks. Such memory system allows simultaneous access to adjacent memory locations. Memory may be n-way interleaved, where n is usually a power of two. 2, 4 and 8-way interleaving is common in large mainframes. In such systems, the

cache size typically would be sufficient to contain a data word from each bank. Figure 4.23 shows an example of interleaved memory.

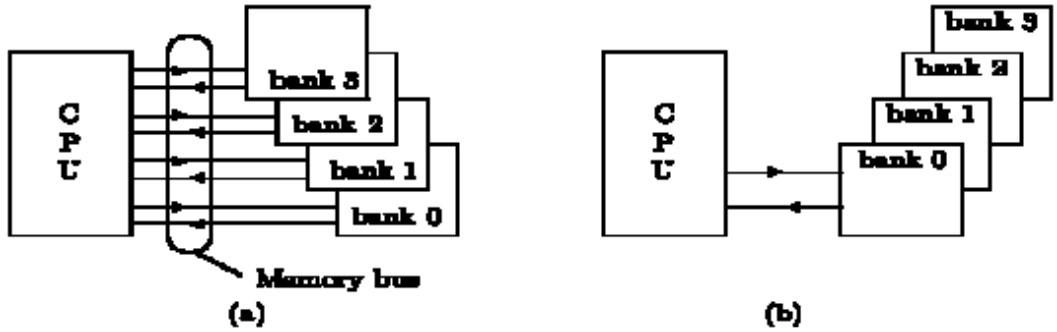


Figure:4.23 4-way interleaved Memory

High order interleaving uses the high order a bits as the module address and the low order b bits are used as the word address in each module.

4.4.2 Bandwidth and Fault tolerance

A single memory module is assumed to deliver one word per mmeory and thus has a bandwidth of 1. The memory bamdwidth B of m-way interleaved memory is upper bounded by m and lower bounded by 1. the Hellerman estimate of B is

$B=m^{0.56}$

Where m is number of interleaved memory modules.

High and low order interleaving can be combined to yield many different interleaved memory organizations. When one module is failed the remaining modules can be still be used by opening window in a address space.

4.4.3 Memory allocation scheme

The idea of using 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 OS kernel which handles the allocation and deallocation of main memory to executing process is called the memory manager. The memory manager monitors

the amount of memory available and decides which processes should reside in main memory and which should be put back to disk if the main memory reaches the limit.

Allocation scheme: Memory swapping is the process of moving blocks of information between the level of memory hierarchy. This swapping policy can be made either preemptive or non preemptive. A non preemptive allocation the incoming process 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. A swap device is a configurable section of a disk which is set aside for temporary storage of information being swapped out of memory. The portion of the disk memory space set aside for a swap device is called the swap space.

4.5 Sequential and weak Consistency Model

When we are dealing with shared-**memory** multiprocessor systems many problems may rise like

- **memory** access to shared writable data (e.g. critical sections) may occur out of program order
- may cause deadlock or incorrect program behavior
- some limitations needed
- In systems with caches, problem is more severe since multiple copies of a data may exist

In order to overcome this problem we need to design **Memory** **consistency** **model** **that may lead to a** set of allowable **memory** access orderings The major tradeoff these models are stricter **memory** **consistency** models are easier to program with, but performance is limited in such models

Memory System Coherence: A **memory** scheme is coherent if the value returned on a Load instruction is always the value given by the latest Store with the same address

Before we discuss them in details lets study few commonly used terms

Memory Requests Initiating A **memory** access is *initiated* when a processor has sent the request and the completion of the request is out of its control.

Memory Requests issued An initiated request is *issued* when it has left the processor (including buffers) and is in transit in the **memory** system.

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 location. 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. There are three primitive memory operations for the purpose of specifying memory consistency model

a) A load by a processor Pi is considered performed with respect to processor Pk at a point of time when issuing of store to same location by Pk cannot affect the value returned by the load.
b) A store by Pi is performed with respect to Pk at one time when we issued load to the same address by Pk returns the value by this store.

For simplicity, we denote Li(X) and Si(X) as Load and Store accesses by processor i on variable X, and {Si(X)}+ as the sequence of Stores following the Store Si(X), including Si(X).

c) Load Globally: A Load is *globally performed* if it is performed and if the Store that is the source of the returned value has been performed with respect to all processor.

For a multiprocessor system this event ordering is very important of obtaining a correct and predictable execution. A correctly written shared-memory parallel program will use mutual exclusion to guard access to shared variables.

P1 **P2**

x = x + 1;

S; S;

x = x + 2;

In general, in a correct parallel program we obtain exclusive access to a set of shared variables, manipulate them any way we want, and then relinquish access, distributing the new values to the rest of the system. The other processors don't need to see any of the intermediate values; they only need to see the final values.

Here's a figure that shows a classification of shared memory accesses:

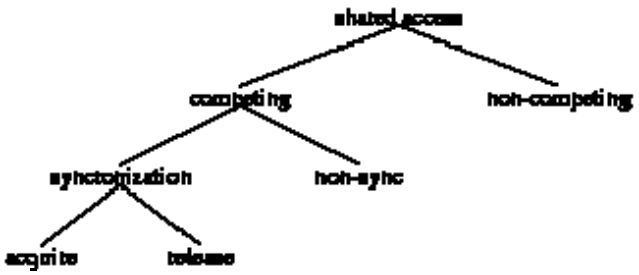


Figure 4.24 classification of shared memory

The various types of memory accesses are defined as follows:

Shared Access

Actually, we can have shared access to variables *vs.* private access. But the questions we're considering are only relevant for shared accesses, so that's all we're showing.

Competing *vs.* Non-Competing

If we have two accesses from different processors, and at least one is a write, they are competing accesses. They are considered as competing accesses because the result depends on which access occurs first (if there are two accesses, but they're both reads, it doesn't matter which is first).

Synchronizing *vs.* Non-Synchroning

Ordinary competing accesses, such as variable accesses, are non-synchronizing accesses. Accesses used in synchronizing the processes are (of course) synchronizing accesses.

Acquire *vs.* Release

Finally, we can divide synchronization accesses into accesses to acquire locks, and accesses to release locks.

Remember that synchronization accesses should be much less common than other competing accesses. So we can further weaken the memory models we use by treating synchronize accesses differently from other accesses.

Atomicity: a shared **memory** access is atomic if the **memory** updates are known to all processors at the same time. **Memory** systems in which accesses are atomically performed for all and any copies of the data are referred to as systems with atomic accessibility . As far as atomicity is considered the multiprocessor memory behavior can be described in three categories:

1. program order are preserved and uniform observation sequence by all processor.
2. Out –of- program-order allowed and uniform observation sequence by all processor.
3. out –of-program-order allowed and nonuniform sequences observed by different processors.

Event ordering

If two accesses are to the same **memory** location, they are conflicting and if these conflicting accesses a1 and a2 on different processors are not ordered and executes simultaneously, causing a race condition, then they form a competing pair, and a1 and a2 are competing accesses in order overcome this problem consistency model is designed

strict model

In the strict model, **any read to a memory location X returns the value stored by the most recent write operation to X**. If we have a bunch of processors, with no caches,

talking to memory through a bus then we will have strict consistency. The point here is the precise serialization of all memory accesses. In a multiprocessor system, storage accesses are strongly ordered if (1) access to global data by any processor are initiated, issued and performed in program order, and if (2) at the time when a Store on global data by processor *i* is observed by processor *k*, all access to global data performed with respect to *i* before issuing of the Store must be performed with respect to *k*.

Sequential Consistency (SC) : Sequential consistency is a slightly weaker model than strict consistency. It was defined by Lamport as **the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program.**

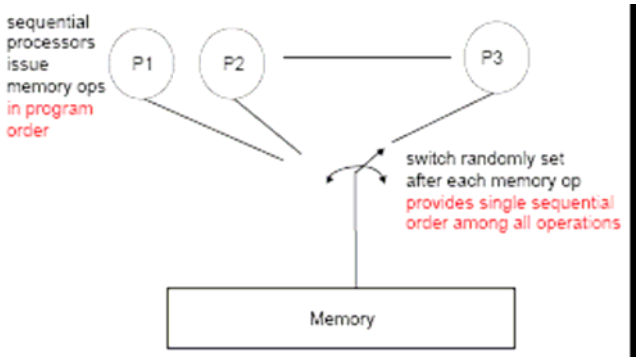


Figure 4.25 Sequential consistency memory model

A 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. In system with caches, one must be careful to maintain SC. If the **memory** has no atomic accessibility, the condition "the order in which all Stores are performed is the same with respect to all processors" must be added, otherwise it may not guarantee **sequential consistency**

Sufficient Conditions for **Sequential Consistency**

Every processor issues memory ops in program order

• Processor must wait for store to complete before issuing next memory operation
• After load, issuing processor must waits for load to complete, and store that produced the value to complete before issuing next operation
– Ensures write atomicity (2 conditions)
• Writes to same location are serialized
• Can't read result of store until all processors will see new value
• Easily implemented with shared bus

In brief we can say

- 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
- 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

To main SC, potential dependencies on every data access to shared **memory** have to be assumed. However, most of these data are not synchronizing variables (shared variables used to control the concurrency between several processes). In these system there is **weak consistency**, two types of shared variables are distinguished:

- the shared operands appearing in algorithms whose value do not control the concurrent execution;
- synchronizing variables which protect the access to shared writable operands or implement synchronization among different processes

It is assumed that at run time, the system can distinguish between accesses to synchronizing variables and to other variables. Synchronizing variables can be distinguished by the type of instruction

Weak consistency results if we only consider competing accesses as being divided into synchronizing and non-synchronizing accesses, and require the following properties:

1. Accesses to synchronization variables are sequentially consistent.
2. No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
3. No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.

Conditions of **Weak Consistency**: In a multiprocessor system, **memory** accesses are weakly ordered if

- accesses to global synchronizing variables are strongly ordered
- no access to a synchronizing variable is issued by a processor before all its previous global data access have been globally performed
- no access to global data is issued by a processor before its previous access to a synchronizing variable has been globally performed
- Dependency conditions on shared variables are not checked continuously but only at explicit synchronizing points. Between two consecutive operations on hardware-recognized synchronization variables, no assumption can be made by the programmer of a process about the order in which Stores are observed by other process. However, the order of successive Stores by a processor to the same address must be respected

Here's a valid scenario under weak consistency, which shows its real strength:

P1: W(x)1 W(x)2 S

P2: R(x)0 R(x)2 S R(x)2

P3: R(x)1 S R(x)2

In other words, there is no requirement that a processor broadcast the changed values of variables *at all* until the synchronization accesses take place. In a distributed system

based on a network instead of a bus, this can dramatically reduce the amount of communication needed (notice that nobody would deliberately write a program that behaved like this in practice; you'd never want to read variables that somebody else is updating. The only reads would be after the S. I've mentioned in lecture that there are a few parallel algorithms, such as relaxation algorithms, that don't require normal notions of memory consistency. These algorithms wouldn't work in a weakly consistent system that really deferred all data communications until synchronization points).

The DSB model : the DSB model is specified by the following three conditions:

1. All pervious synchronization accesses must be performed before load or store access is allowed to perform with respect to any other processor.
2. all pervious load and store accesses must be performed before synchronization access is allowed to perform with respect to any other processor.
3. synchronization accesses are sequentially consistent with respect to one another.

4.6 Keywords

bus A single physical communications medium shared by two or more devices. The *network* shared by processors in many *distributed computers* is a bus, as is the shared data path in many *multiprocessors*.

cache A high-speed memory, local to a single processor , whose data transfers are carried out automatically in hardware. Items are brought into a cache when they are referenced, while any changes to values in a cache are automatically written when they are no longer needed, when the cache becomes full, or when some other process attempts to access them. Also To bring something into a cache.

shared memory: Memory that appears to the user to be contained in a single *address space* and that can be accessed by any process. In a *uniprocessors* or *multiprocessor* there is typically a single memory unit, or several memory units interleaved to give the appearance of a single memory unit..

atomic operation Not interruptible. An atomic operation is one that always appears to have been executed as a unit.

Event Ordering: Used to declare whether a memory event is legal when several processes access a common set of memory locations.

4.7 Summary

In this lesson we had learned how the various processor are connected to each other through the bus. how the data transfer take place from one processor to another in **asynchronous transfer mode.** In this method of transmission does not require a common clock, but separates fields of data by stop and start bits. How through arbitration one system get control over the network and message transmission take place. How cache memory in multiprocessor is organized and how its address are generated both for physical and virtual address. Lastly we had discussed about the shared memory organization and how consistency is maintained in it. There are various issues of synchronization and event handling on which various consistency models are designed.

4.8 Self assessment questions

1. With the help of a diagram, explain the backplane buses, system interfaces, and slot connections to various functional boards in a multiprocessor system.
2. Discuss the typical time sequence for information transfer between a master and a slave over a system bus.
3. Differentiate between synchronous and asynchronous bus timing protocols.
4. With the bus transaction timing diagram, explain the daisy-chained bus arbitration.
5. Explain the 2 interleaved memory organizations with m= 2^a modules and w= 2^b words per module.
6. With diagrams, explain the eight-way low-order interleaving and pipelined access of eight consecutive words in a C- access memory.

4.9 References/Suggested readings

Advance Computer architecture: Kai Hwang

Author: Dr. Deepti Mehrotra **Vetter: Dr. Sandeep Arya**

Lesson: Pipelining and Superscalar Techniques **Lesson No. : 05**

5.0 Objective

The main objective of this lesson is to known the basic properties of pipelining, classification of pipeline processors and the required memory support. The main aim this lesson is to learn the how pipelining is implemented in vector processors, and various limitations of pipelining and how they are overcame by using superscalar pipeline architecture.

5.1 Introduction

Pipeline is similar to the assembly line in industrial plant. To achieve pipelining one must divide the input process into a sequence of sub tasks and each of which can be executed concurrently with other stages. The various classification or pipeline line processor are arithmetic pipelining, instruction pipelining, processor pipelining have also been briefly discussed. Limitations of pipelining are discussed and shift to Pipeline architecture to

Superscalar architecture is also discussed. Superscalar pipeline organization and design are discussed.

5.2 Linear pipelining

Pipelining is a technique of that decompose any sequential process into small subprocesses, which are independent of each other so that each subprocess can be executed in a special dedicated segment and all these segments operates concurrently. Thus whole task is partitioned to independent tasks and these subtask are executed by a segment. The result obtained as an output of a segment (after performing all computation in it) is transferred to next segment in pipeline and the final result is obtained after the data have been through all segments. Thus it could understand if take each segment consists of an input register followed by a combinational circuit. This combinational circuit performs the required sub operation and register holds the intermediate result. The output of one combinational circuit is given as input to the next segment.

The concept of pipelining in computer organization is analogous to an industrial assembly line. As in industry there different division like manufacturing, packing and delivery division, a product is manufactured by manufacturing division, while it is packed by packing division a new product is manufactured by manufacturing unit. While this product is delivered by delivery unit a third product is manufactured by manufacturing unit and second product has been packed. Thus pipeline results in speeding the overall process. Pipelining can be effectively implemented for systems having following characteristics:

- A system is repeatedly executes a *basic function*.
- A basic function must be divisible into independent *stages* such that each stage have minimal overlap.
- The complexity of the stages should be roughly similar.

The pipelining in computer organization is basically flow of information. To understand how it works for the computer system lets consider an process which involves four steps / segment and the process is to be repeated six times. If single steps take t nsec time then time required to complete one process is 4 t nsec and to repeat it 6 times we require 24t nsec.

Now let's see how problem works behaves with pipelining concept. This can be illustrated with a space time diagram given below figure 5.1, which shows the segment utilization as function of time. Lets us take there are 6 processes to be handled (represented in figure as P1, P2, P3, P4, P5 and P6) and each process is divided into 4 segments (S1, S2, S3, S4). For sake of simplicity we take each segment takes equal time to complete the assigned job i.e., equal to one clock cycle. The horizontal axis displays the time in clock cycles and vertical axis gives the segment number. Initially, process1 is handled by the segment 1. After the first clock segment 2 handles process 1 and segment 1 handles new process P2. Thus first process will take 4 clock cycles and remaining processes will be completed one process each clock cycle. Thus for above example total time required to complete whole job will be 9 clock cycles (with pipeline organization) instead of 24 clock cycles required for non pipeline configuration.

	1	2	3	4	5	6	7	8	9
P1	S1	S2	S3	S4					
P2		S1	S2	S3	S4				
P3			S1	S2	S3	S4			
P4				S1	S2	S3	S4		
P5					S1	S2	S3	S4	
P6						S1	S2	S3	S4

Figure 5.1 Space –time diagram for pipeline

Speedup ratio : The speed up ratio is ratio between maximum time taken by non pipeline process over process using pipelining. Thus in general if there are n processes and each process is divided into k segments (subprocesses). The first process will take k segments to complete the processes, but once the pipeline is full that is first process is complete, it will take only one clock period to obtain an output for each process. Thus first process will take k clock cycles and remaining n-1 processes will emerge from the pipe at the one process per clock cycle thus total time taken by remaining process will be (n-1) clock cycle time.

Let t_p be the one clock cycle time.

The time taken for n processes having k segments in pipeline configuration will be

$= k*t_p + (n-1)*t_p = (k+n-1)*t_p$

the time taken for one process is t_n thus the time taken to complete n process in non pipeline configuration will be

$= n*t_n$

Thus speed up ratio for one process in non pipeline and pipeline configuration is

$= n*t_n / (n+k-1)*t_p$

if n is very large compared to k than

$= t_n / t_p$

if a process takes same time in both case with pipeline and non pipeline configuration than $t_n = k*t_p$

Thus speed up ratio will $S_k = k*t_p/t_p = k$

Theoretically maximum speedup ratio will be k where k are the total number of segments in which process is divided. The following are various limitations due to which any pipeline system cannot operate at its maximum theoretical rate i.e., k (speed up ratio).

a. Different segments take different time to complete there suboperations, and in pipelining clock cycle must be chosen equal to time delay of the segment with maximum propagation time. Thus all other segments have to waste time waiting for next clock cycle. The possible solution for improvement here can if possible subdivide the segment into different stages i.e., increase the number of stages and if segment is not subdivisible than use multiple of resource for segment causing maximum delay so that more than one instruction can be executed in to different resources and overall performance will improve.
b. Additional time delay may be introduced because of extra circuitry or additional software requirement is needed to overcome various hazards, and store the result in the intermediate registers. Such delays are not found in non pipeline circuit.
c. Further pipelining can be of maximum benefit if whole process can be divided into suboperations which are independent to each other. But if there is some resource conflict or data dependency i.e., a instruction depends on the result of pervious instruction which is not yet available than instruction has to wait till result become available or conditional or non conditional branching i.e., the bubbles or time delay is introduced.

Efficiency : The efficiency of linear pipeline is measured by the percentage of time when processor are busy over total time taken i.e., sum of busy time plus idle time. Thus if n is number of task , k is stage of pipeline and t is clock period then efficiency is given by

$\eta = n/ [k + n -1]$

Thus larger number of task in pipeline more will be pipeline busy hence better will be efficiency. It can be easily seen from expression as $n \rightarrow \infty$, $\eta \rightarrow 1$.

$\eta = S_k/k$

Thus efficiency η of the pipeline is the speedup divided by the number of stages, or one can say actual speed ratio over ideal speed up ratio. In steady stage where $n >> k$, η approaches 1.

Throughput: The number of task completed by a pipeline per unit time is called throughput, this represents computing power of pipeline. We define throughput as

$W= n/[k*t + (n-1) *t] = \eta/t$

In ideal case as η -> 1 the throughout is equal to 1/t that is equal to frequency. Thus maximum throughput is obtained is there is one output per clock pulse.

Que 5.1. A non-pipeline system takes 60 ns to process a task. The same task can be processed in six segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speed up that can be achieved?

Soln. Total time taken by for non pipeline to complete 100 task is = 100 * 60 = 6000 ns

Total time taken by pipeline configuration to complete 100 task is

= (100 + 6 –1) *10 = 1050 ns

Thus speed up ratio will be = 6000 / 1050 = 4.76

The maximum speedup that can be achieved for this process is = 60 / 10 = 6

Thus, if total speed of non pipeline process is same as that of total time taken to complete a process with pipeline than maximum speed up ratio is equal to number of segments.

Que 5.2. A non-pipeline system takes 50 ns to process a task. The same task can be processed in a six segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speed up that can be achieved?

Soln. Total time taken by for non pipeline to complete 100 task is = 100 * 50 = 5000 ns

Total time taken by pipeline configuration to complete 100 task is

= (100 + 6 –1) *10 = 1050 ns

Thus speed up ratio will be = 5000 / 1050 = 4.76

The maximum speedup that can be achieved for this process is = 50 / 10 = 5

The two areas where pipeline organization is most commonly used are arithmetic pipeline and instruction pipeline. An arithmetic pipeline where different stages of an arithmetic operation are handled along the stages of a pipeline i.e., divides the arithmetic operation into suboperations for execution of pipeline segments. An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle as different stages of pipeline. RISC architecture supports pipelining more than a CISC architecture does. There are three prime disadvantages of pipeline architecture.

1. The first is complexity i.e., to divide the process into dependent subtask
2. Many intermediate registers are required to hold the intermediate information as output of one stage which will be input of next stage. These are not required for single unit circuit thus it is usually constructed entirely as combinational circuit
3. The third disadvantage is its inability to continuously run the pipeline at full speed, i.e. the pipeline stalls for some cycle. There are phenomena called pipeline hazards which disrupt the smooth execution of the pipeline if these hazards are not handled properly they may gave wrong result. Often it is required insert delays in the pipeline flow in order to manage these hazards such delays are called bubbles. Often it is managed by using special hardware techniques while sometime using software techniques such as compiler or code reordering, etc. Various types of pipeline hazards include:
   - structural hazards that happens due to hardware conflicts
   - data hazards that happen due to data dependencies
   - control hazards that happens when there is change in flow of statement like due to branch, jump, or any other control flow changes conditions
   - Exception hazard that happens due to some exception or interrupt occurred while execution in a pipeline system.

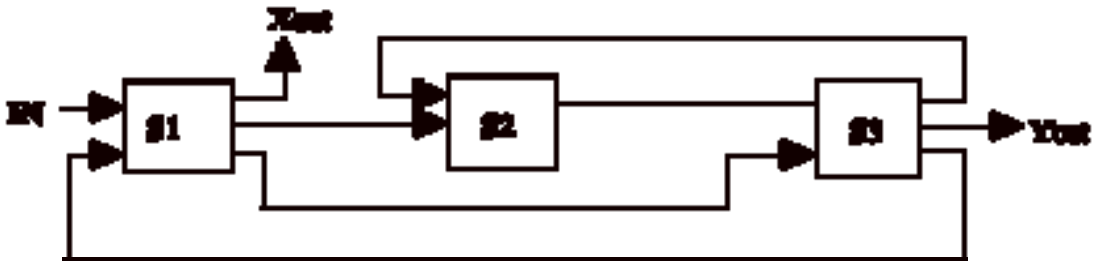
5.3 Non linear pipeline

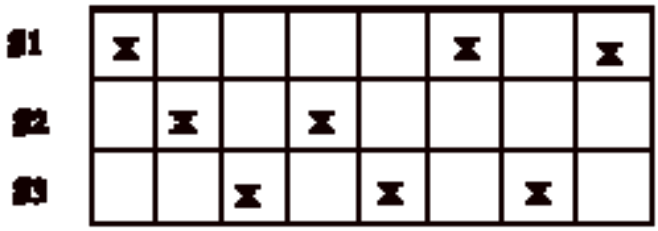
A dynamic pipeline can be reconfigured to perform variable function at different times. The traditional linear pipelines are static pipeline because they used to perform

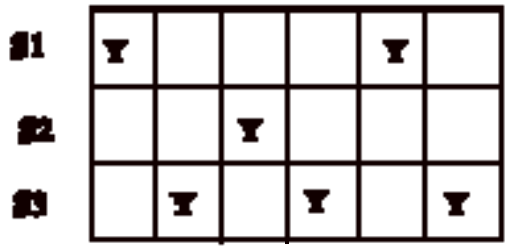
fixed function. A dynamic pipeline allows feed forward and feedback connections in addition to streamline connection. A dynamic pipelining may initiate tasks from different reservation tables simultaneously to allow multiple numbers of initiations of different functions in the same pipeline.

5.3.1 Reservation Tables and latency analysis

Reservation tables are used how successive pipeline stages are utilized for a specific evaluation function. These reservation tables show the sequence in which each function utilizes each stage. The rows correspond to pipeline stages and the columns to clock time units. The total number of clock units in the table is called the evaluation time. A reservation table represents the flow of data through the pipeline for one complete evaluation of a given function. (For example, think of X as being a floating square root, and Y as being a floating cosine. A simple floating multiply might occupy just S1 and S2 in sequence.) We could also denote multiple stages being used in parallel, or a stage being drawn out for more than one cycle with these diagrams.

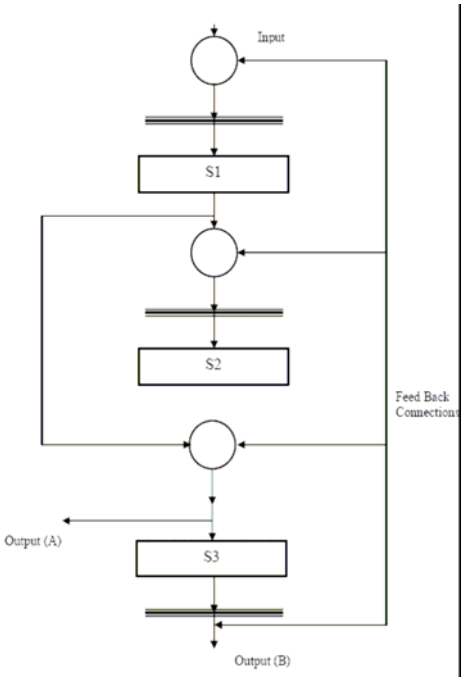






We determine the next start time for one or the other of the functions by lining up the diagrams and sliding one with respect to another to see where one can fit into the open slots. Once an X function has been scheduled, another X function can start after 1, 3 or 6 cycles. A Y function can start after 2 or 4 cycles. Once a Y function has been scheduled, another Y function can start after 1, 3 or 5 cycles. An X function can start after 2 or 4 cycles. After two functions have been scheduled, no more can be started until both are complete.

Consider another example of non linear pipeline where both feed forward and feedback connections are given. The pipeline is dual functional denoted as function A and function B. The pipeline stages are numbered as S1, S2 and S3. The feed forward connection connects a stage Si to a stage Sj such that $j \geq i + 2$ and feedback connection connects to Si to a stage Sj such that $j <= i$



A market entry in the (i,j)th square of the table indicates the stage Si will be used j time units after initiation of the function evaluation. Here is the reservation table for both

Time	t0	t1	t2	t3	t4	t5	t6	t7
S1	A			A			A	
S2		A						A
S3			A		A	A		

Table Reservation Table for function A

Time	t0	t1	t2	t3	t4	t5	t6
S1	B				B		
S2			B			B	
S3		B		B			B

Table Reservation Table for function B

Job Sequencing and Collision Prevention

Initiation the start a single function evaluation collision may occur as two or more initiations attempt to use the same stage at the same time. Thus it is required to properly schedule queued tasks awaiting initiation in order to avoid collisions and to achieve high throughput. We can define collision as:

1. A collision occurs when two tasks are initiated with latency (initiation interval) equal to the column distance between two “X” on some row of the reservation table.

2. The set of column distances F ={l1,l2,…,lr} between all possible pairs of "X" on each row of the reservation table is called the forbidden set of latencies.

3. The collision vector is a binary vector C = (Cn…C2 C1), Where Ci=1 if i belongs to F (set of forbidden latencies) and Ci=0 otherwise.

Some fundamental concepts used in it are**:**

Latency - number of time units between two initiations (any positive integer 1, 2,…)

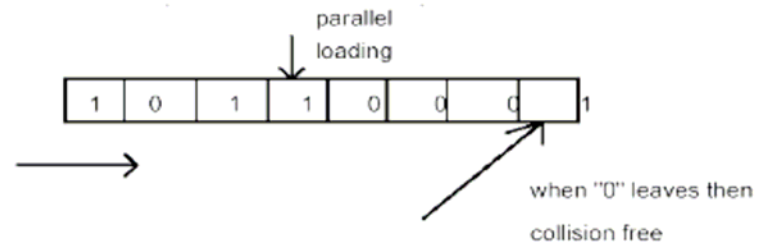
Latency sequence – sequence of latencies between successive initiations

Latency cycle – a latency sequence that repeats itself

Control strategy – the procedure to choose a latency sequence

Greedy strategy – a control strategy that always minimizes the latency between the current initiation and the very last initiation

Example: Let us consider a Reservation Table with the following set of forbidden latencies F and permitted latencies P (complementation of F).



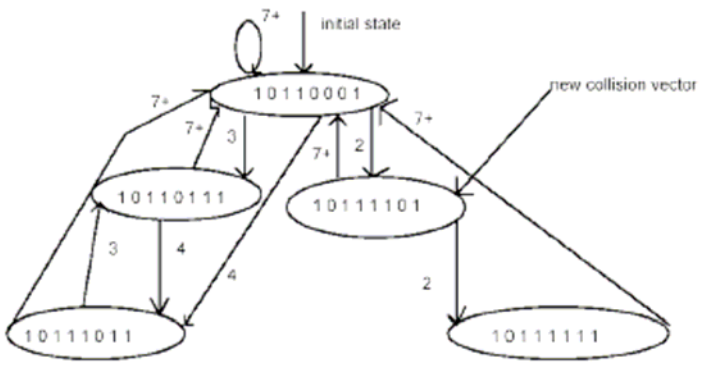
Forbidden list = F = {1,5,6,8}

Collision vector: C={1 0 1 1 0 0 0 1)

8 7 6 5 4 3 2 1

	0	1	2	3	4	5	6	7	8
1	X								X
2		X	X					X	
3				X					
4					X	X			
5							X	X	

1 0 1 1 0 0 0 1
1 0 1 1 0 1 1 1
OR 1 0 1 1
1 0 1 1 1 0 1 1




```
1 0 1 1 0 0 0 1   initial
0 0 1 0 1 1 0 0   →2 (initial)
1 0 1 1 1 1 0 1

1 0 1 1 0 0 0 1   initial
0 0 0 1 0 1 1 0   →2 (initial)
1 0 1 1 0 1 1 1

1 0 1 1 0 0 0 1   initial
0 0 0 0 1 0 1 1   →4 (initial)

1 0 1 1 0 0 0 1   initial
0 0 0 0 0 0 0 1   →7 (initial)
1 0 1 1 0 0 0 1
```

It has been observed that

1. The collision vector shows both permitted and forbidden latencies from the same reservation table.

2. One can use n-bit shift register to hold the collision vector for implementing a control strategy for successive task initiations in the pipeline. Upon initiation of the first task, the collision vector is parallel-loaded into the shift register as the initial state. The shift register is then shifted right one bit at a time, entering 0's from the left end. A collision free initiation is allowed at time instant t+k a bit 0 is being shifted at of the register after k shifts from time t.

A **state diagram** is used to characterize the successive initiations of tasks in the pipeline in order to find the shortest latency sequence to optimize the control strategy. A **state** on the diagram is represented by the contents of the shift register after the proper number of shifts is made, which is equal to the latency between the current and next task initiations.

3. The successive collision vectors are used to prevent future task collisions with previously initiated tasks, while the collision vector C is used to prevent possible collisions with the current task. If a collision vector has a "1" in the ith bit (from the right), at time t, then the task sequence should avoid the initiation of a task at time t+i.

4. Closed logs or cycles in the state diagram indicate the steady – state sustainable latency sequence of task initiations without collisions. The **average latency** of a cycle is the sum of its latencies (period) divided by the number of states in the cycle.

5. The throughput of a pipeline is inversely proportional to the reciprocal of the average latency. A latency sequence is called **permissible** if no collisions exist in the successive initiations governed by the given latency sequence.

6. The maximum throughput is achieved by an optimal scheduling strategy that achieves the (MAL) minimum average latency without collisions.

Simple cycles are those latency cycles in which each state appears only once per each iteration of the cycle. A single cycle is a **greedy cycle** if each latency contained in the cycle is the minimal latency (outgoing arc) from a state in the cycle. A good task-initiation sequence should include the greedy cycle.

Procedure to determine the greedy cycles

1. From each of the state diagram, one chooses the arc with the smallest latency label unit; a closed simple cycle can formed.

2. The average latency of any greedy cycle is no greater than the number of latencies in the forbidden set, which equals the number of 1's in the initial collision vector.

3. The average latency of any greedy cycle is always lower-bounded by the

MAL in the collision vector

Two methods for improving dynamic pipeline throughput have been proposed by Davidson and Patel these are

- The reservation of a pipeline can be modified with insertion of non complete delays
- Use of internal buffer at each stage.

Thus high throughput can be achieved by using the modified reservation table yielding a more desirable latency pattern such the each stage is maximum utilized. Any computation can be delayed by inserting a non compute stage.

Reconfigurable pipelines with different function types are more desirable. This requires an extensive resource sharing among different functions. To achieve this one need a more complicated structure of pipeline segments and their interconnection controls like bypass techniques to avoid unwanted stage.

A dynamic pipeline would allow several configurations to be simultaneously present like arithmetic unit performing both addition as well as multiplication at same time. But to achieve this tremendous control overhead and increased interconnection complexity would be expected.

5.4 Instruction pipeline

As we know that in general case, the each instruction to execute in computer undergo following steps:

- Fetch the instruction from the memory.
- Decode the instruction.
- Calculate the effective address.
- Fetch the operands from the memory.
- Execute the instruction (EX).
- Store the result back into memory (WB).

For sake of simplicity we take calculation of the effective address and fetch operand from memory as single segment as operand fetch unit. Thus below figure shows how the instruction cycle in CPU can be processed with five segment instruction pipeline.

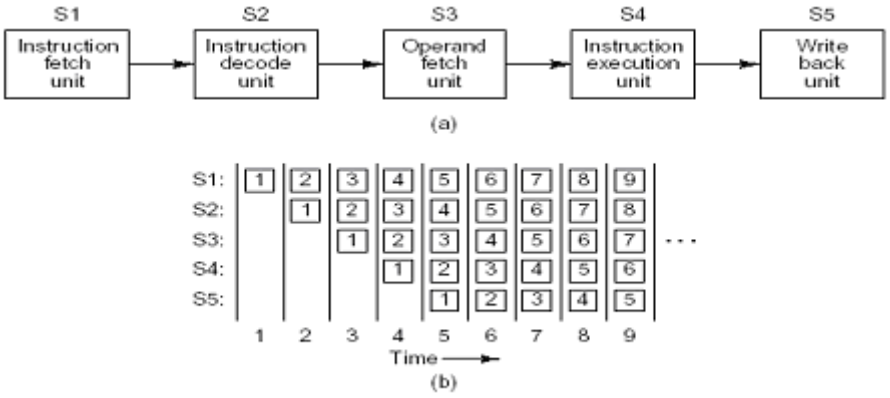


Figure 5.14 (a) A five stage instruction pipeline (b) Space time diagram of pipeline

While the instruction is decoded (ID) in segment 2 the new instruction is fetched (IF) from segment 1. Similarly in third time cycle when first instruction effective operand is fetch (OF), the 2nd instruction is decoded and the 3rd instruction is fetched. In same manner in fourth clock cycle, and subsequent cycles all subsequent instructions can be fetched and placed in instruction FIFO. Thus up to five different instructions can be processed at the same time. The figure show how the instruction pipeline works, where

time is in the horizontal axis and divided into steps of equal duration. Although the major difficulty with instruction pipeline is that different segment may take different time to operate the forth coming information. For example if operand is in register mode require much less time as compared if operand has to be fetched from memory that to with indirect addressing modes. The design of an instruction pipeline will be most effective if the instruction cycle is divided into segments of equal duration. As there can be resource conflict, data dependency, branching, interrupts and other reasons due to pipelining can branch out of normal sequence these will be discussed in unit 5.7.

Que 5.3 Consider a program of 15,000 instructions executed by a linear pipeline processor with a clock rate of 25MHz. The instruction pipeline has five stages and one instruction is issued per clock cycle. Calculate speed up ratio, efficiency and throughput of this pipelined processor?

Soln: Time taken to execute without pipeline is = 15000 * 5* (1/25) microsecs

Time taken with pipeline = (15000 + 5 -1)*(1/ 25) microsecs

Speed up ratio = (15000*5*25) / (15000+ 5 -1)*25 = 4.99

Efficiency = Speed up ratio/ number of segment in pipeline = 4.99/5= 0.99

Throughput = number of task completed in unit time = 0.99 * 25 = 24.9 MIPS

Principles of designing pipeline processor

Buffers are used to speed close up the speed gap between memory access for either instructions or operands. Buffering can avoid unnecessary idling of the processing stages caused by memory access conflicts or by unexpected branching or interrupts. The concepts of busing eliminates the time delay to store and to retrieve intermediate results or to from the registers.

The computer performance can be greatly enhanced if one can eliminate unnecessary memory accesses and combine transitive or multiple fetch-store operations with faster register operations. This is carried by register tagging and forwarding.

Another method to smooth the traffic flow in a pipeline is to use buffers to close up the speed gap between the memory accesses for either instructions or operands and arithmetic and logic executions in the functional pipes. The instruction or operand buffers provide a continuous supply of instructions or operands to the appropriate pipeline units. Buffering can avoid unnecessary idling of the processing stages caused by memory

access conflicts or by unexpected branching or interrupts. Sometimes the entire loop instructions can be stored in the buffer to avoid repeated fetch of the same instructions loop, if the buffer size is sufficiently large. It is very large in the usage of pipeline computers.

Three buffer types are used in various instructions and data types. Instructions are fetched to the instruction fetch buffer before sending them to the instruction unit. After decoding, fixed point and floating point instructions and data are sent to their dedicated buffers. The store address and data buffers are used for continuously storing results back to memory. The storage conflict buffer is used only used when memory

Busing Buffers

The sub function being executed by one stage should be independent of the other sub functions being executed by the remaining stages; otherwise some process in the pipeline must be halted until the dependency is removed. When one instruction waiting to be executed is first to be modified by a future instruction, the execution of this instruction must be suspended until the dependency is released.

Another example is the conflicting use of some registers or memory locations by different segments of a pipeline. These problems cause additional time delays. An efficient internal busing structure is desired to route the resulting stations with minimum time delays.

In the AP 120B or FPS 164 attached processor the busing structure are even more sophisticated. Seven data buses provide multiple data paths. The output of the floating point adder in the AP 120B can be directly routed back to the input of the floating point adder, to the input of the floating point multiplier, to the data pad, or to the data memory. Similar busing is provided for the output of the floating point multiplier. This eliminates the time delay to store and to retrieve intermediate results or to from the registers.

Internal Forwarding and Register Tagging

To enhance the performance of computers with multiple execution pipelines

1. **Internal Forwarding** refers to a short circuit technique for replacing unnecessary memory accesses by register -to-register transfers in a sequence of fetch-arithmetic-store operations

2. **Register Tagging** refers to the use of tagged registers, buffers and reservations stations for exploiting concurrent activities among multiple arithmetic units.

The computer performance can be greatly enhanced if one can eliminate unnecessary memory accesses and combine transitive or multiple fetch-store operations with faster register operations. This concept of internal data forwarding can be explored in three directions. The symbols Mi and Rj to represent the ith word in the memory and jth fetch, store and register-to register transfer. The contents of Mi and Rj are represented by (Mi) and Rj

Store-Fetch Forwarding

The store the n fetch can be replaced by 2 parallel operations, one store and one register transfer.

2 memory accesses

Mi -> (R1) (store)

R2 -> (Mi) (Fetch)

Is being replaced by

Only one memory access

Mi -> (R1) (store)

R2 -> (R1) (register Transfer)

Fetch-Fetch Forwarding

The following fetch operations can be replaced by one fetch and one register transfer. One memory access has been eliminated.

2 memory accesses

R1 -> (Mi) (fetch)

R2 -> (Mi) (Fetch)

Is being replaced by

Only one memory access

R1 -> (Mi) (Fetch)

R2 -> (R1) (register Transfer)

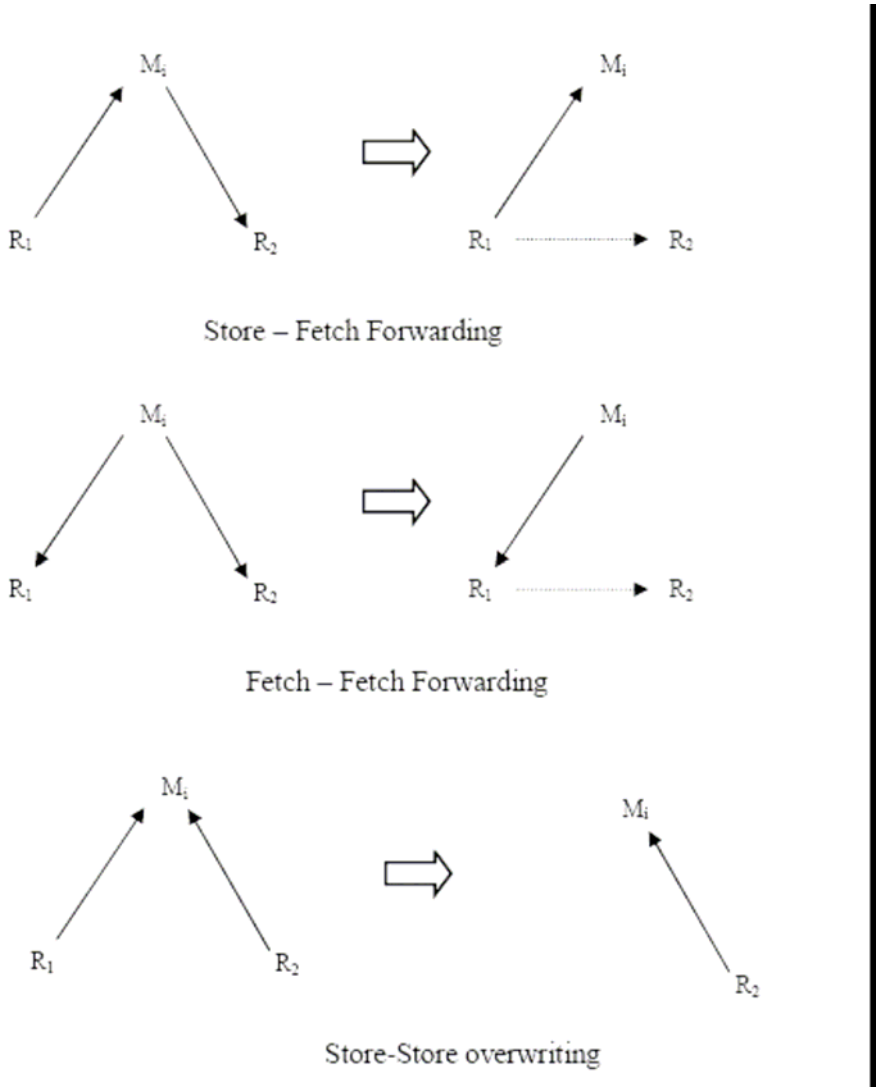


Figure .2 Internal Forwarding Examples thick arrows for memory accesses and dotted arrows for register transfers

Store-Store Overwriting

The following two memory updates of the same word can be combined into one; since the second store overwrites the first. 2 memory accesses

Mi -> (R1) (store)

Mi -> (R2) (store)

Is being replaced by

Only one memory access

Mi -> (R2) (store)

The above steps shows how to apply internal forwarding to simplify a sequence of arithmetic and memory access operations

Forwarding and Data Hazards

Sometimes it is possible to avoid data hazards by noting that a value that results from one instruction is not needed until a late stage in a following instruction, and sending the data directly from the output of the first functional unit back to the input of the second one (which is sometimes the same unit). In the general case, this would require the output of every functional unit to be connected through switching logic to the input of every functional unit.

Data hazards can take three forms:

Read after write (RAW): Attempting to read a value that hasn't been written yet. This is the most common type, and can be overcome by forwarding.

Write after write (WAW): Writing a value before a preceding write has completed. This can only happen in complex pipes that allow instructions to proceed out of order, or that have multiple write-back stages (mostly CISC), or when we have multiple pipes that can write (superscalar).

Write after read (WAR): Writing a value before a preceding read has completed. These also require a complex pipeline that can sometimes write in an early stage, and read in a later stage. It is also possible when multiple pipelines (superscalar) or out-of-order issue are employed.

The fourth situation, read after read (RAR) does not produce a hazard.

Forwarding does not solve every RAW hazard situation. For example, if a functional unit is merely slow and fails to produce a result that can be forwarded in time, then the pipeline must stall. A simple example is the case of a load, which has a high latency. This is the sort of situation where compiler scheduling of instructions can help, by rearranging independent instructions to fill the delay slots. The processor can also rearrange the instructions at run time, if it has access to a window of prefetched instructions (called a

prefetch buffer). It must perform much the same analysis as the compiler to determine which instructions are dependent on each other, but because the window is usually small, the analysis is more limited in scope. The small size of the window is due to the cost of providing a wide enough datapath to predecode multiple instructions at once, and the complexity of the dependence testing logic.

Out of order execution introduces another level of complexity in the control of the pipeline, because it is desirable to preserve the abstraction of in-order issue, even in the presence of exceptions that could flush the pipe at any stage. But we'll defer this to later.

Branch Penalty Hiding

The control hazards due to branches can cause a large part of the pipeline to be flushed, greatly reducing its performance. One way of hiding the branch penalty is to fill the pipe behind the branch with instructions that would be executed whether or not the branch is taken. If we can find the right number of instructions that precede the branch and are independent of the test, then the compiler can move them immediately following the branch and tag them as branch delay filling instructions. The processor can then execute the branch, and when it determines the appropriate target, the instruction is fetched into the pipeline with no penalty.

Of course, this scheme depends on how the pipeline is designed. It effectively binds part of the pipeline design for a specific implementation to the instruction set architecture. As we've seen before, it is generally a bad idea to bind implementation details to the ISA because we may need to change them later on. For example, if we decide to lengthen the pipeline so that the number of delay slots increases, we have to recompile our code to have it execute efficiently -- we no longer have strict binary compatibility among models of the same "ISA".

The filling of branch delays can be done dynamically in hardware by reordering instructions out of the prefetch buffer. But this leads to other problems. Another way to hide branch penalties is to avoid certain kinds of branches. For example, if we have

IF A < 0

THEN A = -A

we would normally implement this with a nearby branch. However, we could instead use an instruction that performs the arithmetic conditionally (skips the write back if the condition fails). The advantage of this scheme is that, although one pipeline cycle is wasted, we do not have to flush the rest of the pipe (also, for a dynamic branch prediction scheme, we need not put an extra branch into the prediction unit). These are called predicated instructions, and the concept can be extended to other sorts of operations, such as conditional loading of a value from memory.

Branch Prediction

Branches are the bane of any pipeline, causing a potentially large decrease in performance as we saw earlier. There are several ways to reduce this loss by predicting the action of the branch ahead of time.

Simple static prediction assumes that all branches will be taken or not. The designer decides which way is predicted from instruction trace statistics. Once the choice is made, the compiler can help by properly ordering local jumps. A slightly more complex static branch prediction heuristic is that backward branches are usually taken and forward branches are not (backwards taken, forwards not or BTFN). This assumes that most backward branches are loop returns and that most forward branches are the less likely cases of a conditional branch.

Compiler static prediction involves the use of special branches that indicate the most likely choice (taken or not, or more typically taken or other, since the most predictable branches are those at the ends of loops that are mostly taken). If the prediction fails in this case, then the usual cancellation of the instructions in the delay slots occurs and a branch penalty results.

Dynamic instruction scheduling

As discussed above the static instruction scheduling can be optimized by compiler the dynamic scheduling is achived either by using scoreboard or with Tomasulo’s register tagging algorithm and discussed in superscalar processors

5.5 Arithmetic pipeline

Pipeline arithmetic is used in very high speed computers specially involved in scientific computations a basic principle behind vector processor and array processor. They are used to implement floating – point operations, multiplication of fixed – point numbers and similar computations encountered in computation problems. These computation problems can easily decomposed in suboperations. Arithmetic pipelining is well implemented in the systems involved with repeated calculations such as calculations involved with matrices and vectors. Let us consider a simple vector calculation like

A[i] + b[i] * c[i] for I = 1,2,3,……,8

The above operation can be subdivided into three segment pipeline such each segment has some registers and combinational circuits. Segment 1 load contents of b[i] and c[i] in register R1 and R2 , segment 2 load a[i] content to R3 and multiply content of R1, R2 and store them R4 finally segment 3 add content of R3 and R4 and store in R5 as shown in figure 5.12 below.

Clock pulse number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	B1	C1	-	-	-
2	B2	C2	B1*C1	A1	
3	B3	C3	B2*C2	A2	A1+ B1*C1
4	B4	C4	B3*C3	A3	A2+ B2*C2
5	B5	C5	B4*C4	A4	A3+ B3*C3
6	B6	C6	B5*C5	A5	A4+ B4*C4
7	B7	C7	B6*C6	A6	A5+ B5*C5
8	B8	C8	B7*C7	A7	A6+ B6*C6
9			B8*C8	A8	A7+ B7*C7
10					A8+ B8*C8

Figure 5.12 Content of registers in pipeline

To illustrate the operation principles of a pipeline computation, the design of a pipeline floating point adder is given. It is constructed in four stages. The inputs are

A = a x 2p

B = b x 2q

Where a and b are 2 fractions and p and q are their exponents and here base 2 is assumed.

To compute the sum

C = A+ B = c x 2r = d x 2s

Operations performed in the four pipeline stages are specified.

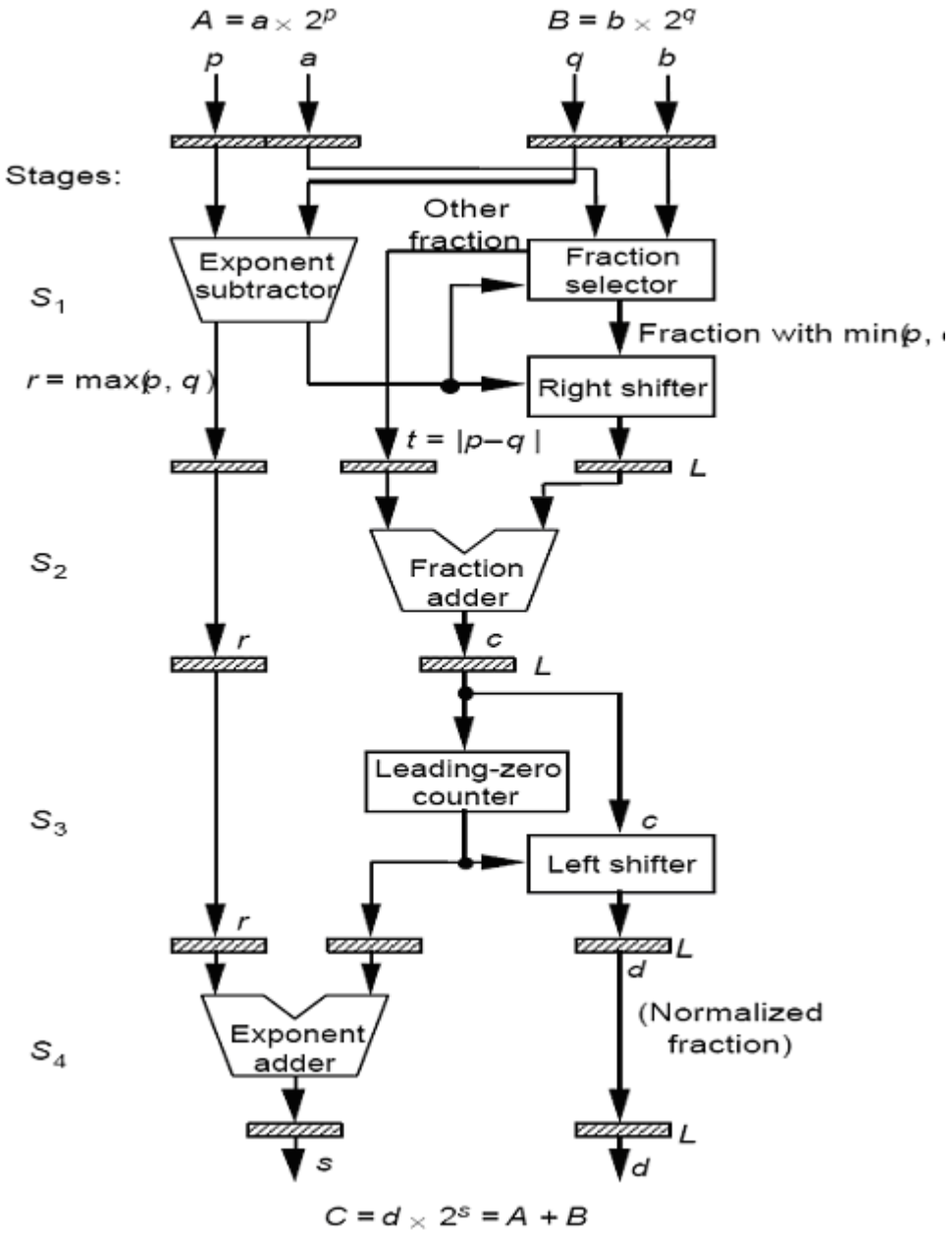
1. Compare the 2 exponents p and q to reveal the larger exponent r =max(p,q) and to determine their difference t =p-q

2. Shift right the fraction associated with the smaller exponent by t bits to equalize the two components before fraction addition.

3. Add the preshifted fraction with the other fraction to produce the intermediate sum fraction c where 0 <= c <1.

4. Count the number of leading zeroes, say u, in fraction c and shift left c by u bits to produce the normalized fraction sum d = c x 2u, with a leading bit 1. Update the large exponent s by subtracting s= r – u to produce the output exponent.

The given below is figure 5.5 show how pipeline can be implemented in floating point addition and subtraction. Segment 1 compare the two exponents this is done using subtraction. Segment2 we chose the larger exponents the one larger exponent as exponent of result also it align the other mantissa by viewing the difference between two and smaller number mantissa should be shifted to right by difference amount. Segment 3 performs addition or subtraction of mantissa while segment 4 normalize the result for that it adjust exponent care must be taken in case of overflow, where we had to shift the mantissa right and increment exponent by one and for underflow the leading zeros of mantissa determines the left shift in mantissa and same number should be subtracted for exponent. Various registers R are used to hold intermediate results.



In order to implement pipelined adder we need extra circuitry but its cost is compensated if we have implement it for large number of floating point numbers. Operations at each stage can be done on different pairs of inputs, e.g. one stage can be comparing the exponents in one pair of operands at the same time another stage is adding the mantissas of a different pair of operands.

Que 5.3 How faster will be addition of 1000 floating point vector element using above pipeline as compared to non pipeline adder?

Soln: The above pipeline has four segment and lets assume each segment require one clock cycle time to execute. To add 1000 pairs of numbers, without a pipelined adder would require 4000 cycles.

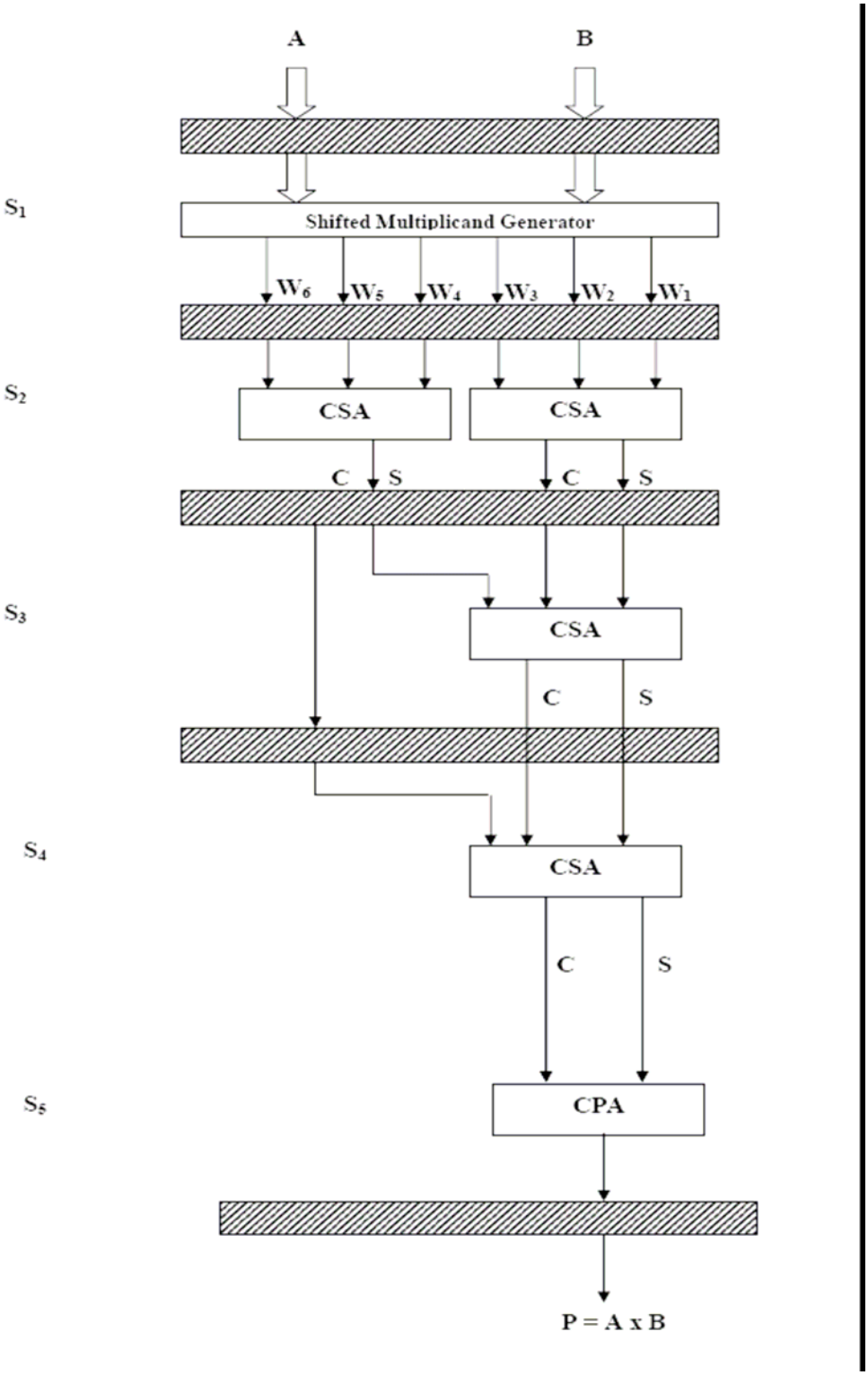
With 4-stage pipelined adder, the last sum will appear after 1000 +4 cycles, so the pipeline is 4000/ 1004 = 3.98 times faster.

Lets take another example of multiplication

The multiplication of 2 fixed point numbers is done by repeated add-shift operations, using ALU which has built in add and shift functions. Multiple number additions can be realized with a multilevel tree adder. The conventional carry propagation adder (CPA) adds 2 input numbers say A and B, to produce one output number called the sum A+B carry save adder (CSA) receives three input numbers, say A,B and D and two output numbers, the sum S and the Carry vector C.

	A	=		1	1	1	1	0	1
	B	=		0	1	0	1	1	0
	D	=		1	1	0	1	1	1
	C	=	1	1	0	1	1	1	
	S	=		0	1	1	1	0	0
	A+B+D	=	1	1	1	0	0	1	0

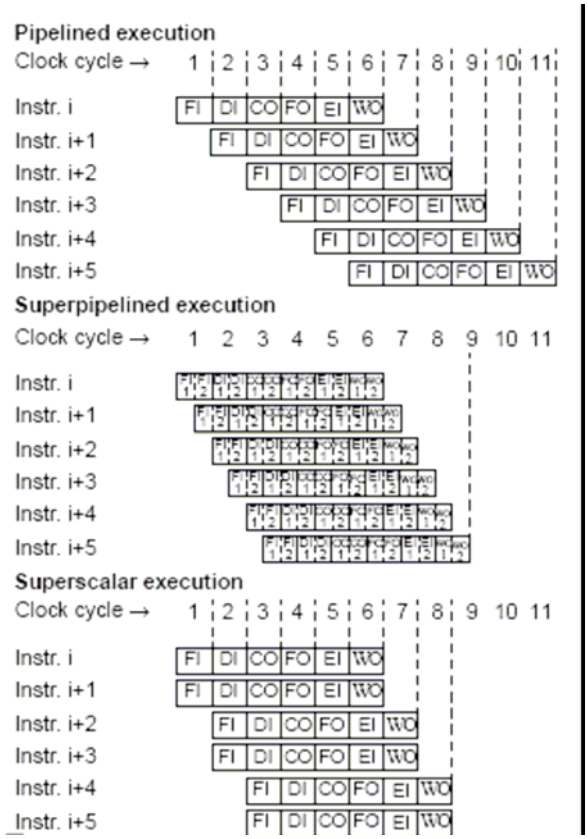
A CSA can be implemented with a cascade of full adders with the carry-out of a lower stage connected to the carry-in of a higher stage. A carry-save adder can be implemented with a set of full adders with all the carry-in terminals serving as the input lines for the third input number D, and all the carry-out terminals serving as the output lines for the carry vector C. This pipeline is designed to multiply two 6 bit numbers. There are five pipeline stages. The first stage is for the generation of all 6 x 6 = 36 immediate product terms, which forms the six rows of shifted multiplicands. The six numbers are then fed into two CSAs in the second stage. In total four CSAs are interconnected to form a three level merges six numbers into two numbers: the sum vector S and the carry vector C. The final stage us a CPA which adds the two numbers C and S to produce the final output of the product A x B.



5.6 **superpipeline and Superscalar technique**

Instruction level parallelism is obtained primarily in two ways in uniprocessors: through pipelining and through keeping multiple functional units busy executing multiple

instructions at the same time. When a pipeline is extended in length beyond the normal five or six stages (e.g., I-Fetch, Decode/Dispatch, Execute, D-fetch, Writeback), then it may be called Superpipelined. If a processor executes more than one instruction at a time, it may be called Superscalar. A superscalar architecture is one in which several instructions can be initiated simultaneously and executed independently These two techniques can be combined into a Superscalar pipeline architecture.



5.6.1 Superpipeline

Superpipelining is based on dividing the stages of a pipeline into substages and thus increasing the number of instructions which are supported by the pipeline at a given moment. For example if we divide each stage into two, the clock cycle period t will be reduced to the half, t/2; hence, at the maximum capacity, the pipeline produces a result every t/2 s. For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; increasing the number of stages over this limit reduces the overall performance. A solution to further improve speed is the superscalar architecture.

Given a pipeline stage time T, it may be possible to execute at a higher rate by starting operations at intervals of T/n. This can be accomplished in two ways:

- Further divide each of the pipeline stages into n substages.
- Provide n pipelines that are overlapped.

The first approach requires faster logic and the ability to subdivide the stages into segments with uniform latency. It may also require more complex inter-stage interlocking and stall-restart logic.

The second approach could be viewed in a sense as staggered superscalar operation, and has associated with it all of the same requirements except that instructions and data can be fetched with a slight offset in time. In addition, inter-pipeline interlocking is more difficult to manage because of the sub-clock period differences in timing between the pipelines.

Even so, staggered clock pipelines may be necessary with superscalar designs in the future, in order to reduce peak power and corresponding power-supply induced noise. Alternatively, designs may be forced to shift to a balanced mode of circuit operation in which logic transitions are balanced by reverse transitions -- a technique used in the Cray supercomputers that resulted in the computer presenting a pure DC load to the power supply, and greatly reduced noise in the system.

Inevitably, superpipelining is limited by the speed of logic, and the frequency of unpredictable branches. Stage time cannot productively grow shorter than the interstage latch time, and so this is a limit for the number of stages.

The MIPS R4000 is sometimes called a superpipelined machine, although its 8 stages really only split the I-fetch and D-fetch stages of the pipe and add a Tag Check stage. Nonetheless, the extra stages enable it to operate with higher throughput. The UltraSPARC's 9-stage pipe definitely qualifies it as a superpipelined machine, and in fact it is a Super-Super design because of its superscalar issue. The Pentium 4 splits the pipeline into 20 stages to enable increased clock rate. The benefit of such extensive

pipelining is really only gained for very regular applications such as graphics. On more irregular applications, there is little performance advantage.

5.6.2 Superscalar

Superscalar processing has its origins in the Cray-designed CDC supercomputers, in which multiple functional units are kept busy by multiple instructions. The CDC machines could pack as many as 4 instructions in a word at once, and these were fetched together and dispatched via a pipeline. Given the technology of the time, this configuration was fast enough to keep the functional units busy without outpacing the instruction memory.

Current technology has enabled, and at the same time created the need to issue instructions in parallel. As execution pipelines have approached the limits of speed, parallel execution has been required to improve performance. As this requires greater fetch rates from memory, which hasn't accelerated comparably, it has become necessary to fetch instructions in parallel -- fetching serially and pipelining their dispatch can no longer keep multiple functional units busy. At the same time, the movement of the L1 instruction cache onto the chip has permitted designers to fetch a cache line in parallel with little cost.

In some cases superscalar machines still employ a single fetch-decode-dispatch pipe that drives all of the units. For example, the UltraSPARC splits execution after the third stage of a unified pipeline. However, it is becoming more common to have multiple fetch-decode-dispatch pipes feeding the functional units.

The choice of approach depends on tradeoffs of the average execute time vs. the speed with which instructions can be issued. For example, if execution averages several cycles, and the number of functional units is small, then a single pipe may be able to keep the units utilized. When the number of functional units grows large and/or their execution time approaches the issue time, then multiple issue pipes may be necessary.

Having multiple issue pipes requires

- being able to fetch instructions for that many pipes at once
- inter-pipeline interlocking
- reordering of instructions for multiple interlocked pipelines
- multiple write-back stages
- multiport D-cache and/or register file, and/or functionally split register file

Reordering may be either static (compiler) or dynamic (using hardware lookahead). It can be difficult to combine the two approaches because the compiler may not be able to predict the actions of the hardware reordering mechanism.

Superscalar operation is limited by the number of independent operations that can be extracted from an instruction stream. It has been shown in early studies on simpler processor models, that this is limited, mostly by branches, to a small number (<10, typically about 4). More recent work has shown that, with speculative execution and aggressive branch prediction, higher levels may be achievable. On certain highly regular codes, the level of parallelism may be quite high (around 50). Of course, such highly regular codes are just as amenable to other forms of parallel processing that can be employed more directly, and are also the exception rather than the rule. Current thinking is that about 6-way instruction level parallelism for a typical program mix may be the natural limit, with 4-way being likely for integer codes. Potential ILP may be three times this, but it will be very difficult to exploit even a majority of this parallelism. Nonetheless, obtaining a factor of 4 to 6 boost in performance is quite significant, especially as processor speeds approach their limits.

Going beyond a single instruction stream and allowing multiple tasks (or threads) to operate at the same time can enable greater system throughput. Because these are naturally independent at the fine-grained level, we can select instructions from different streams to fill pipeline slots that would otherwise go vacant in the case of issuing from a single thread. In turn, this makes it useful to add more functional units. We shall further explore these multithreaded architectures later in the course.

Hardware Support for Superscalar Operation

There are two basic hardware techniques that are used to manage the simultaneous execution of multiple instructions on multiple functional units: Scoreboarding and reservation stations. Scoreboarding originated in the Cray-designed CDC-6600 in 1964, and reservation stations first appeared in the IBM 360/91 in 1967, as designed by Tomasulo.

Scoreboard

A scoreboard is a centralized table that keeps track of the instructions to be performed and the available resources and issues the instructions to the functional units when everything is ready for them to proceed. As the instructions execute, dependences are checked and execution is stalled as necessary to ensure that in-order semantics are preserved. Out of order execution is possible, but is limited by the size of the scoreboard and the execution rules. The scoreboard can be thought of as preceding dispatch, but it also controls execution after the issue. In a scoreboarded system, the results can be forwarded directly to their destination register (as long as there are no write after read hazards, in which case their execution is stalled), rather than having to proceed to a final write-back stage.

In the CDC scoreboard, each register has a matching Result Register Designator that indicates which functional unit will write a result into it. The fact that only one functional unit can be designated for writing to a register at a time ensures that WAW dependences cannot occur. Each functional unit also has a corresponding set of Entry-Operand Register Designators that indicate what register will hold each operand, whether the value is valid (or pending) and if it is pending, what functional unit will produce it (to facilitate forwarding). None of the operands is released to a functional unit until they are all valid, precluding RAW dependences. In addition , the scoreboard stalls any functional unit whose result would write a register that is still listed as an Entry-Operand to a functional unit that is waiting for an operand or is busy, thus avoiding WAR violations. An instruction is only allowed to issue if its specified functional unit is free and its result

register is not reserved by another functional unit that has not yet completed. Four Stages of Scoreboard Control

1. Issue—decode instructions & check for structural hazards (ID1) If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

2. Read operands—wait until no data hazards, then read operands (ID2) A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

3. Execution—operate on operands (EX) The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.

4. Write result—finish execution (WB) Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction. Example:

DIVD F0,F2,F4

ADDD F10,F0,**F8**

SUBD **F8**,F8,F14

CDC 6600 scoreboard would stall SUBD until ADDD reads operands

Three Parts of the Scoreboard

1. Instruction status—which of 4 steps the instruction is in

2. Functional unit status—Indicates the state of the functional unit (FU). 9 fields for each functional unit

Busy—Indicates whether the unit is busy or not

Op—Operation to perform in the unit (e.g., + or –)

Fi—Destination register

Fj, Fk—Source-register numbers

Qj, Qk—Functional units producing source registers Fj, Fk

Rj, Rk—Flags indicating when Fj, Fk are ready and not yet read. Set to

No after operands are read.

3. Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register

Scoreboard Implications

• provide solution for WAR, WAW hazards

• Solution for WAR – Stall Write in WB to allow Reads to take place; Read registers only during Read Operands stage.

• For WAW, must detect hazard: stall in the Issue stage until other completes

• Need to have multiple instructions in execution phase

• Scoreboard keeps track of dependencies, state or operations

– Monitors every change in the hardware.

– Determines when to read ops, when can execute, when can wb.

– Hazard detection and resolution is centralized.

Reservation Stations The reservation station approach releases instructions directly to a pool of buffers associated with their intended functional units (if more than one unit of a particular type is present, then the units may share a single station). The reservation stations are a distributed resource, rather than being centralized, and can be thought of as following dispatch. A reservation is a record consisting of an instruction and its requirements to execute -- its operands as specified by their sources and destination and bits indicating when valid values are available for the sources. The instruction is released to the functional unit when its requirements are satisfied, but it is important to note that satisfaction doesn't require an operand to actually be in a register -- it can be forwarded to the reservation station for immediate release or to be buffered (see below) for later release. Thus, the reservation station's influence on execution can be thought of as more implicit and data dependent than the explicit control exercised by the scoreboard.

Tomasulo Algorithm

The hardware dependence resolution technique used **For IBM 360/91 about 3 years after CDC 6600.** Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

If reservation station free, then issue instruction & send operands (renames registers).

2. Execution—operate on operands (EX)

When both operands ready then execute; if not ready, watch CDB for result

3. Write result—finish execution (WB)

Write on Common Data Bus to all awaiting units; mark reservation station available.

Here the storage of operands resulting from instructions that completed out of order is done through renaming of the registers. There are two mechanisms commonly used for renaming. One is to assign physical registers from a free pool to the logical registers as they are identified in an instruction stream. A lookup table is then used to map the logical register references to their physical assignments. Usually the pool is larger than the logical register set to allow for temporary buffering of results that are computed but not yet ready to write back. Thus, the processor must keep track of a larger set of register names than the instruction set architecture specifies. When the pool is empty, instruction issue stalls.

The other mechanism is to keep the traditional association of logical and physical registers, but then provide additional buffers either associated with the reservation stations or kept in a central location. In either case, each of these "reorder buffers" is associated with a given instruction, and its contents (once computed) can be used in forwarding operations as long as the instruction has not completed.

When an instruction reaches the point that it may complete in a manner that preserves sequential semantics, then its reservation station is freed and its result appears in the logical register that was originally specified. This is done either by renaming the temporary register to be one of the logical registers, or by transferring the contents of the reorder buffer to the appropriate physical register.

Out of Order Issue

To enable out-of-order dispatch of instructions to the pipelines, we must provide at least two reservation stations per pipe that are available for issue at once. An alternative would

be to rearrange instructions in the prefetch buffer, but without knowing the status of the pipes, it would be difficult to make such a reordering effective. By providing multiple reservation stations, however, we can continue issuing instructions to pipes, even though an instruction may be stalled while awaiting resources. Then, whatever instruction is ready first can enter the pipe and execute. At the far end of the pipeline, the out-of-order instruction must wait to be retired in the proper order. This necessitates a mechanism for keeping track of the proper order of instructions (note that dependences alone cannot guarantee that instructions will be properly reordered when they complete).

5.6.3 Superscalar-Superpipeline

Of course we may also combine superscalar operation with superpipelining. The result is potentially the product of the speedup factors.

However, it is even more difficult to interlock between parallel pipes that are divided into many stages. Also, the memory subsystem must be able to sustain a level of instruction throughput corresponding to the total throughput of the multiple pipelines -- stretching the processor/memory performance gap even more. Of course, with so many pipes and so many stages, branch penalties become huge, and branch prediction becomes a serious bottleneck (offsetting this somewhat is the potential for speculative branch execution that arises with a large number of pipes).

But the real problem may be in finding the parallelism required to keep all of the pipes and stages busy between branches. Consider that a machine with 12 pipelines of 20 stages must always have access to a window of 240 instructions that are scheduled so as to avoid all hazards, and that the average of 40 branches that would be present in a block of that size are all correctly predicted sufficiently in advance to avoid stalling in the prefetch unit. This is a tall order, even for highly regular code with a good balance of floating point to integer operations, let alone irregular code that is typically unbalanced in its operation mix. As usual, scientific code and image processing with their regular array operations often provide the best performance for a super-super processor. However, a vector unit could more directly address the array processing problems, and what is left to the scalar unit may not benefit nearly so greatly from all of this complexity. Scalar

processing can certainly benefit from ILP in many cases, but probably in patterns that differ from vector processing.

5.7 Summary

1. The job-sequencing problem is equivalent to finding a permissible latency cycle with the MAL in the state diagram.
2. The minimum number of X's in array single row of the reservation table is a lower bound of the MAL.

Pipelining allows several instructions to be executed at the same time, but they have to be in different pipeline stages at a given moment. Superscalar architectures include all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage. They have the ability to initiate multiple instructions during the same clock cycle. There are two typical approaches today, in order to improve performance:

1. Superpipelining
2. Superscalar

5.8 Keywords

pipelining Overlapping the execution of two or more operations. Pipelining is used within processors by *prefetching* instructions on the assumption that no branches are going to preempt their execution; in *vector processors*, in which application of a single operation to the elements of a vector or vectors may be pipelined to decrease the time needed to complete the aggregate operation; and in *multiprocessors* and *multicomputers*, in which a process may send a request for values before it reaches the computation that requires them..

scoreboard A hardware device that maintains the state of machine resources to enable instructions to execute without conflict at the earliest opportunity.

instruction pipelining strategy of allowing more than one instruction to be in some stage of execution at the same time.

5.9 Self assessment questions

1. Explain an asynchronous pipeline model, a synchronous pipeline model and reservation table of a four-stage linear pipeline with appropriate diagrams.
2. Define the following terms with regard to clocking and timing control.

a) Clock cycle and throughput b) Clock skewing c) Speedup factor

3. Describe the speedup factors and the optimal number of pipeline stages for a linear pipeline unit.
4. Explain the features of non-linear pipeline processors with feedforward and feedbackward connections.
5. With state diagrams, explain the transition diagram for a pipeline unit.
6. Explain the pipelined execution of the following instructions with the following instructions:

a) X = Y + Z b) A = B X C

7. What are the possible hazards that can occur between read and write operations in an instruction pipeline?
8. Explain the Tomasulo's algorithm for dynamic instruction scheduling.
9. With diagrams, explain branch history buffer and a state transition diagram used in dynamic branch prediction.
10. Differentiate between a carry-propagate adder (CPA) and a carry-save adder

5.10 References/Suggested readings

Advance Computer architecture: Kai Hwang

Author: Dr. Deepti Mehrotra **Vetter: Dr. Sandeep Arya**

Lesson: Multiprocessor and Multicomputers **Lesson No. : 06**

6.0 Objective

In this lesson we will study about system architectures of multiprocessor and mulicomputer. Various cache coherence protocols, synchronization methods, and other important concepts involved in building a multiprocessor. Finally we will discuss the mulitcomputers with distributed memories which are not globally shared.

6.1 Introduction

There are various architecture supporting parallel processing exists these are boardly classified as Multiprocessors and Multicomputers. The common classification are **Shared-Memory Multiprocessors Models which include all** UMA**: uniform memory access (all SMP servers),** NUMA**: nonuniform-memory-access (Stanford DASH, SGI Origin 2000, Cray T3E) and** COMA**: cache-only memory architecture (KSR)** Which have **very low remote memory access latency.**

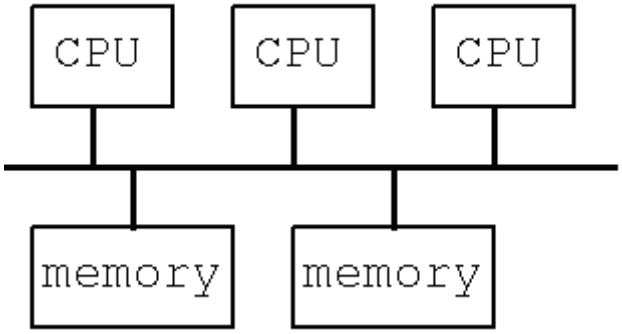


Figure 6.1 Shared memory multiprocessor

The Distributed-Memory Multicomputers Model must have a message-passing network, highly scalable like NORMA **model (no-remote-memory-access), IBM SP2, Intel Paragon, TMC CM-5, INTEL ASCI Red, PC cluster**

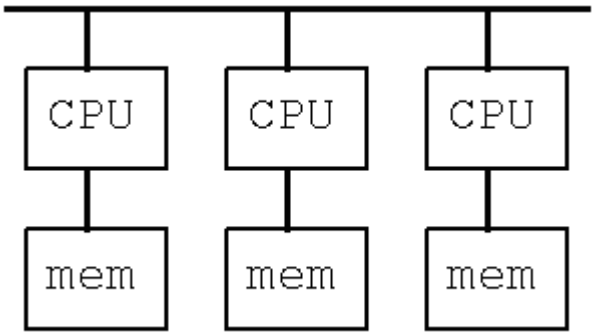


Figure 6.2 Distributed memory multiprocessor

6.2 Multiprocessor system interconnect

In the multiprocessor architecture each processor Pi is attached to its own local memory and private cache. These multiple processors connected to share memory through interprocessor memory network (IPMN). Processors share access to I/O and peripherals through processor-I/O network (PION). Both IPMN and PION are necessary in a shared-resource multiprocessor. An optional interprocessor communication network (IPCN) can permit processor communication without using shared memory.

Interconnection Network Choices

The networks are designed with many choices like timing, switching and control strategy like in case of dynamic network the multiprocessors interconnections are under program control.

- Timing

- Synchronous – controlled by a global clock which synchronizes all network activity.
- Asynchronous – use handshaking or interlock mechanisms for communication and especially suitable for coordinating devices with different speed.

- Switching Method
  - Circuit switching – a pair of communicating devices control the path for the entire duration of data transfer
  - Packet switching – large data transfers broken into smaller pieces, each of which can compete for use of the path
- Network Control
  - Centralized – global controller receives and acts on requests
  - Distributed – requests handled by local devices independently

6.2.1Hierarchical bus system

Digital buses are the fundamental interconnects adopted in most commercial multiprocessor systems with less than 100 processors. The principal limitation to the bus approach is packaging technology.

Complete bus specifications include logical, electrical and mechanical properties, application profiles, and interface requirements.

Bus Systems

A bus system is a hierarchy of buses connection various system and subsystem components. Each bus has a complement of control, signal, and power lines. In a bus-based network, processors share a single communication resource. A bus is a highly non-scalable architecture, because only one processor can communicate on the bus at a time. Used in shared-memory parallel computers to communicate read and write requests to a shared global memory

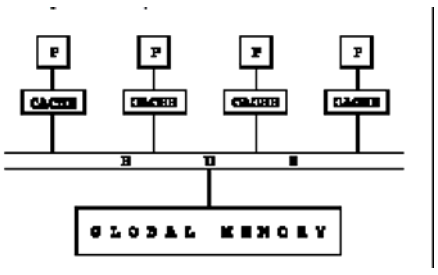


Figure 6.3Bus system

There is usually a variety of buses in a system:

Local bus – (usually integral to a system board) connects various major system components (chips)

Memory bus – used within a memory board to connect the interface, the controller, and the memory cells

Data bus – might be used on an I/O board or VLSI chip to connect various components

Backplane – like a local bus, but with connectors to which other boards can be attached

Hierarchical Bus Systems

There are numerous ways in which buses, processors, memories, and I/O devices can be organized. One organization has processors (and their caches) as leaf nodes in a tree, with the buses (and caches) to which these processors connect forming the interior nodes.

A bus-based interconnection network, used here to implement a shared-memory parallel computer. Each processor (P) is connected to the bus, which in turn is connected to the global memory. A cache associated with each processor stores recently accessed memory values in an effort to reduce the bus traffic.

This generic organization, with appropriate protocols to ensure cache coherency, can model most hierarchical bus organizations.

Bridges

The term bridge is used to denote a device that is used to connect two (or possibly more) buses. The interconnected buses may use the same standards, or they may be different (e.g. PCI and ISA buses in a modern PC). Bridge functions include

- Communication protocol conversion
- Interrupt handling
- Serving as cache and memory agents

6.2.2 Crossbar Switch and Multiport Memory

Single stage networks are sometimes called recirculating networks because data items may have to pass through the single stage many times. The crossbar switch and the multiported memory organization (seen later) are both single-stage networks.

This is because even if two processors attempted to access the same memory module (or I/O device at the same time, only one of the requests is serviced at a time.

Multistage Networks

Multistage networks consist of multiple sages of switch boxes, and should be able to connect any input to any output. A multistage network is called blocking if the simultaneous connections of some multiple inputoutput pairs may result in conflicts in the use of switches or communication links.

A nonblocking multistage network can perform all possible connections between inputs and outputs by rearranging its connections.

Crossbar Networks

Crossbar networks connect every input to every output through a crosspoint switch.

A crossbar network is a single stage, non-blocking permutation network.

In an n-processor, m-memory system, n *. m crosspoint switches will be required. Each crosspoint is a unary switch which can be open or closed, providing a point-to-point connection path between the processor and a memory module.

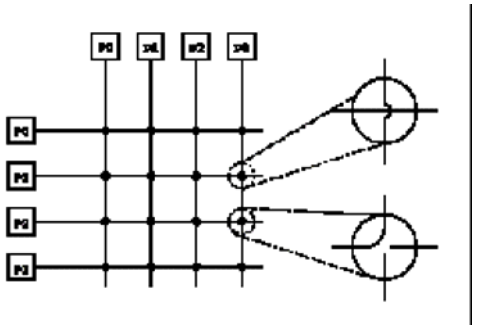


Figure 6.4 Cross bar network

Lets for example consider a 4*4 nonblocking crossbar, used here to connect 4 processors to four memory. Like in below figure processor P1 is connected to memory M3 and P3 is communicating with M2. Pairs of processors can communicate without preventing other processor pairs from communicating.

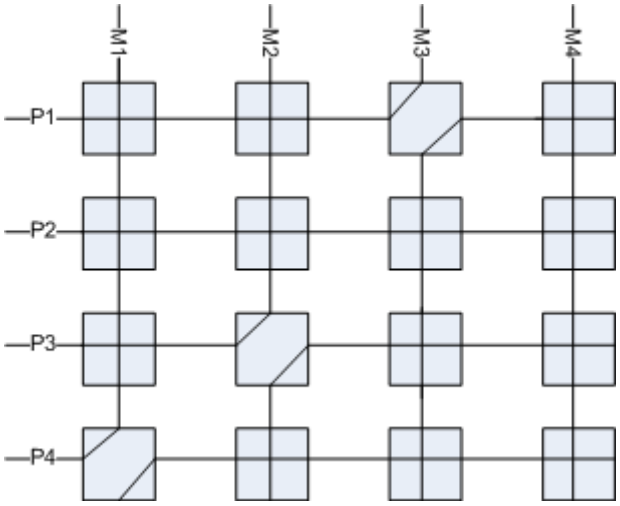


Figure 6.5 Example of 4*4 crossbar network

Crosspoint Switch Design

Out of n crosspoint switches in each column of an *n* .*m* crossbar mesh, only one can be connected at a time. Crosspoint switches must be designed to handle the potential contention for each memory module. A crossbar switch avoids competition for bandwidth by using $O(N^2)$ switches to connect N inputs to N outputs.

Although highly non-scalable, crossbar switches are a popular mechanism for connecting a small number of workstations, typically 20 or fewer.

Each processor provides a request line, a read/write line, a set of address lines, and a set of data lines to a crosspoint switch for a single column. The crosspoint switch eventually responds with an acknowledgement when the access has been completed.

Multiport Memory

Since crossbar switches are expensive, and not suitable for systems with many processors or memory modules, multiport memory modules may be used instead. A multiport memory module has multiple connections points for processors (or I/O devices), and the memory controller in the module handles the arbitration and switching that might otherwise have been accomplished by a crosspoint switch.

A two function switch can assume only two possible state namely state or exchange states. However a four function switch box can be any of four possible states. A multistage network is capable of connecting any input terminal to any output terminal. Multi-stage networks are basically constructed by so called shuffle-exchange switching

element, which is basically a 2 x 2 crossbar. Multiple layers of these elements are connected and form the network.

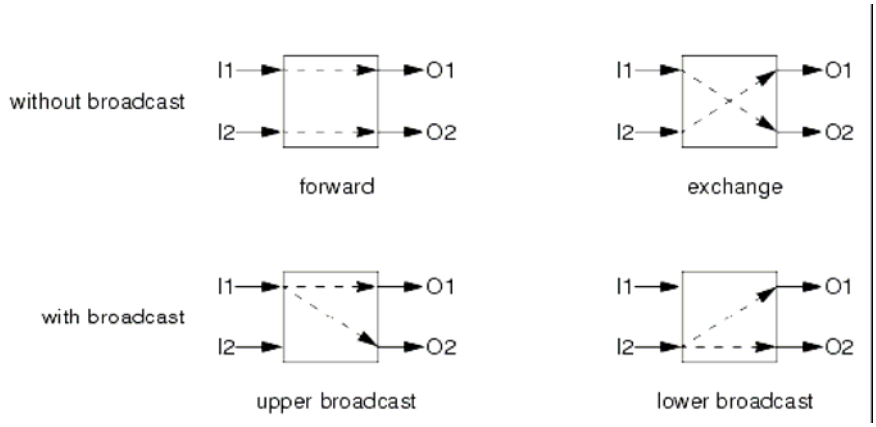


Figure 6.6 A switching element of multistage mnetwork

Multiport Memory Examples Omega Networks

The basic building block of an omega network is a switch with two inputs and two outputs. When a message arrives at this switch, the first bit is stripped off and the switch is set to: straight through if the bit is '0' on the top input or '1' on the bottom input else cross connected. Note that only one message can pass, the other being blocked, if two messages arrive and the exclusive or of the first bits is not '1'.

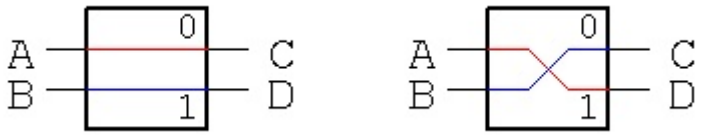


Figure 6.7 switching element for omega network

Then omega networks for connecting two devices, four devices or eight devices are built from this switch are shown below. The messages are sent with the most significant bit of the destination first.

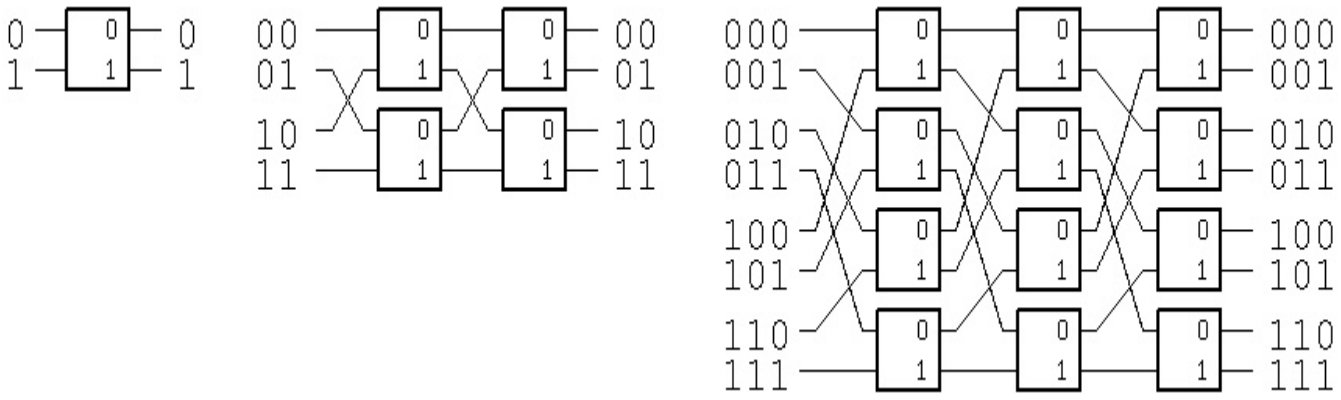


Figure 6.8 Omega network connecting two device, four device and eight device

For 16 devices connected to the same or different 16 devices, the omega network is built from the primitive switch as:

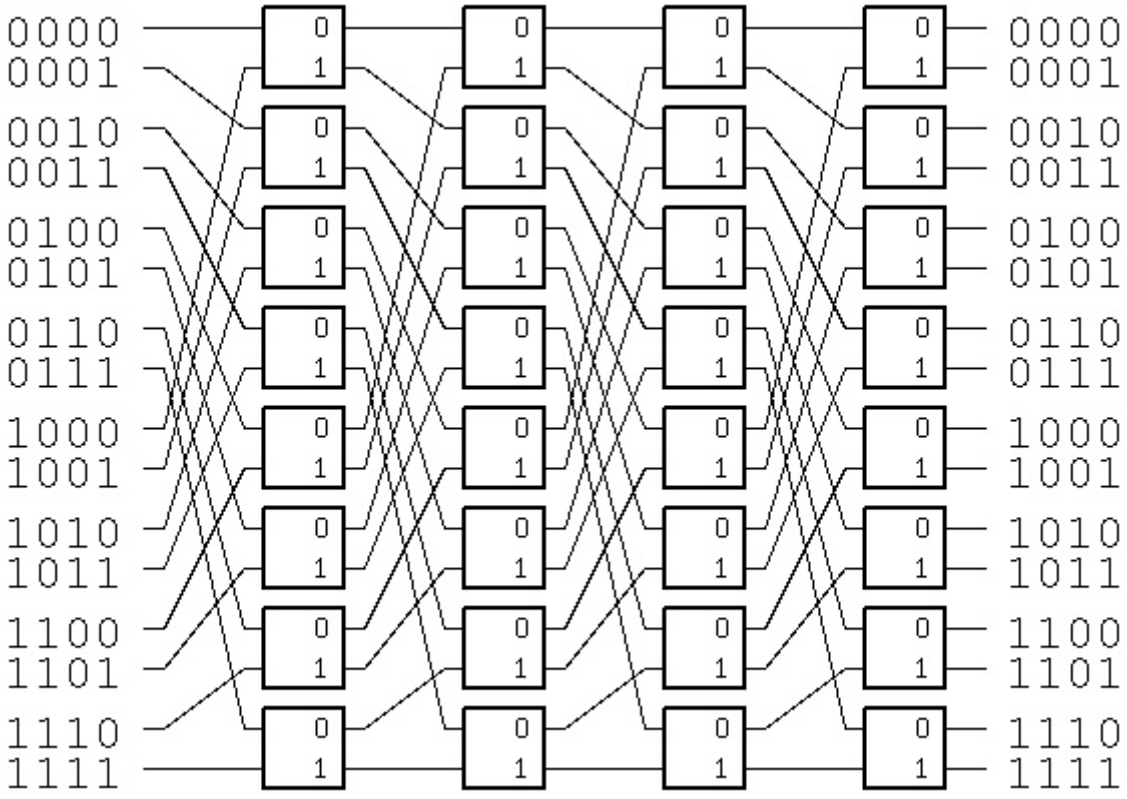


Figure 6.9 Omega network connecting 16 devices

Note that connecting N devices requires N $\log_2(N)$ switches.

Given a set of random connections of N devices to N devices with an omega network, this is mathematically a permutation, then statistically 1/2 N connections may be made

simultaneously. N-input Omega networks, in general, have log2n stages, with the input stage labeled 0.

The interstage connection (ISC) pattern is a perfect shuffle. Routing is controlled by inspecting the destination address. When the i-th highest order bit is 0, the 2*2 switch in stage i connects the input to the upper output. Otherwise it connects the input to the lower output. $2^k = N$ inputs and a like number of outputs. Between these are $\log_2 N$ stages each having N/2 exchange elements at each stage. An Omega network is typically a (semi-) **blocking** network. When an exchange element is in use, the next message or data packet that needs this element must wait.

Omega Network without Blocking Effects

Blocking exists in an Omega network when the requested permutation would require that a single switch be set in two positions simultaneously. Obviously this is impossible, and requires that one of the permutation requests be blocked and tried in a later pass.

In general, with 2*2 switches, an Omega network can implement $n\ n/2$ permutations in a single pass. For $n = 8$, this is about 10% of all possible permutations.

In general, a maximum of log2*n* passes are needed for an *n*-input Omega network.

Omega Broadcast

An Omega network can be used to broadcast data to multiple destinations by using upper broadcast or lower broadcast switch settings. The switch to which the input is connected is set to the broadcast position (input connected to both outputs). Each additional switch (in later stages) to which an output is directed is also set to the broadcast position.

Omega Broadcast Larger Switches

Larger switches (more inputs and outputs, and more switching patterns) can be used to build an Omega network, resulting in fewer stages. For example, with 4*4 switches, only $\log_4 16$ stages are required for a 16-input switch. A k-way perfect shuffle is used as the ISC for an Omega network using k *k switches.

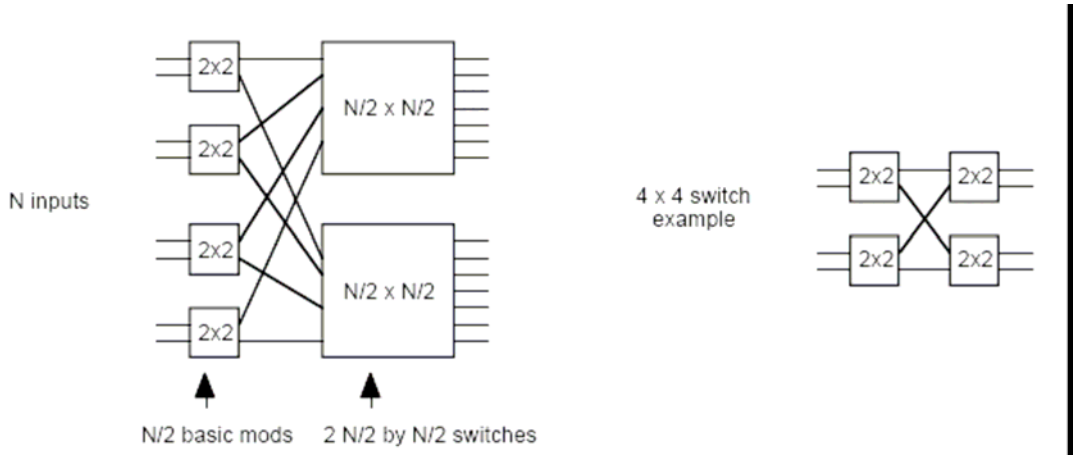


Figure 6.10 Omega network with large switch

*Omega Network with 4*4 Switches Butterfly Networks*

Butterfly networks are built using crossbar switches instead of those found in Omega networks. There are no broadcast connections in a butterfly network, making them a restricted subclass of the Omega networks.

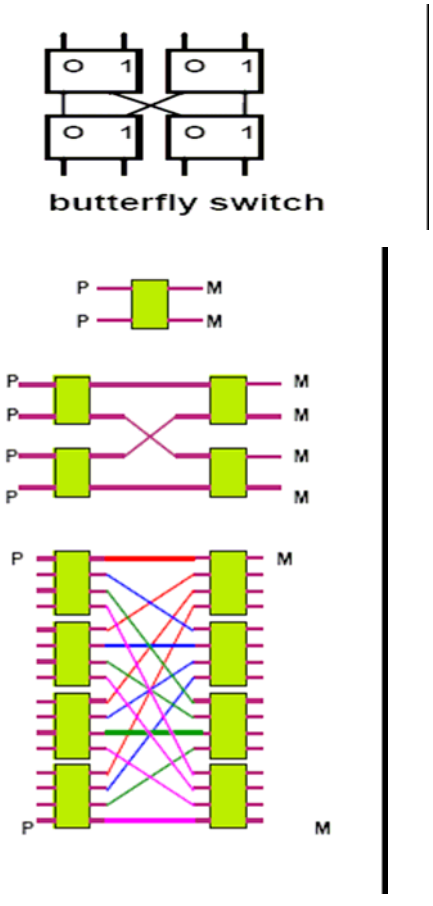


Figure 6.11 a butterfly switch

Hot Spots

When a particular memory module is being heavily accessed by multiple processors at the same time, we say a hot spot exists. For example, if multiple processors are accessing the same memory location with a spin lock implemented with a test and set instruction, then a hot spot may exist. Obviously, hot spots may significantly degrade the network performance.

Dealing With Hot Spots

To avoid the hot spot problems, we may develop special operations that are actually implemented partially by the network.

Consider the instruction Fetch&Add(x,e), which has the following definition (x is a memory location, and the returned value is stored in a processor register):

temp <- x

x <- x + e

return temp

Implementing Fetch&Add

When n processors attempt to execute Fetch&Add on the same location simultaneously, the network performs a serialization on the requests, performing the following steps atomically. *x* is returned to one processor, x+e1 to the next, x+e1+e2, to the next, and so forth. The value x+e1+e2+…+en is stored in x. Note that multiple simultaneous test and set instructions could be handled in a similar manner.

The Cost of Fetch&Add

Clearly a feature like Fetch&Add is not available at no cost.

Each switch in the network must be built to detect the Fetch&Add requests (distinct from other requests), queuing them until the operation can be atomically completed.

Additional switch cycles may be required, increasing network latency significantly.

A multiprocessor may have distributed memory, shared memory or a combination of both.

6.3 Cache Coherence and Synchronization

6.3.1 Cache coherence problem

An important problem that must be addressed in many parallel systems - any system that allows multiple processors to access (potentially) multiple copies of data - is *cache*

coherence. The existence of multiple cached copies of data creates the possibility of inconsistency between a cached copy and the shared memory or between cached copies themselves.

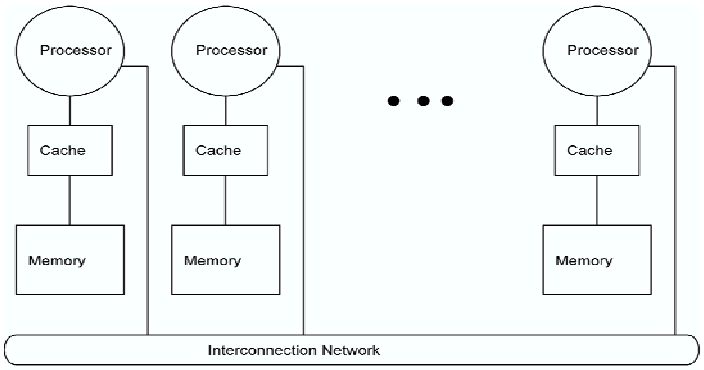


Figure 6.12 cache coherence problem in multiprocessor

There are three common sources of cache inconsistency:

- Inconsistency in data sharing : In a memory hierarchy for a multiprocessor system data inconsistency may occur between adjacent levels or within the same level. The cache inconsistency problem occurs only when multiple private cache are used. Thus it is, the possible that a wrong data being accessed by one processor because another processor has changed it, and not all changes have yet been propagated. Suppose we have two processors, A and B, each of which is dealing with memory word X, and each of which has a cache. If processor A changes X, then the value seen by processor B in *its own cache* will be wrong, *even if processor A also changes the value of X in main memory* (which it - ultimately - should).

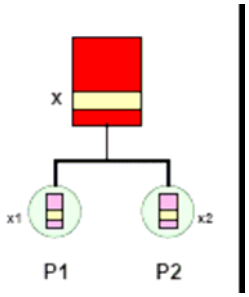


Figure 6.13 Cache coherence problem

In above example initially, x1 = x2 = X = 5.

P1 writes X:=10 using *write-through*.

P2 now reads X and uses its local copy x2, but finds that X is still 5.

Thus P2 does not know that P1 modified X.

Thus the cache inconsistency problem occurs when multiple private cache are used and especially the problem arose by writing the shared variables.

- Process migration(even if jobs are independent): This problem occurs when a process containing shared variable X migrates from process 1 to process2 using the write back cache on the right. Thus another important aspect of coherence is *serialization* of writes - that is, if two processors try to write 'simultaneously', then (i) the writes happen sequentially (and it doesn't really matter who gets to write first - provided we have sensible arbitration); and (ii) *all processors see the writes as occurring in the same order*. That is, if processors A and B both write to X, with A writing first, then any other processors (C, D, E) *all* see the *same* thing.

- DMA I/O – this inconsistency problem occur during the I/O operation that bypass the cache. This problem is present even in a uniprocessor and can be removed by OS cache flushes)

In practice, these issues are managed by a memory bus, which by its very nature ensures write serialization, and also allows us to broadcast invalidation signals (we essentially just put the memory address to be invalidated on the bus). We can add an extra *valid* bit to cache tags to mark then invalid. Typically, we would use a write-back cache, because it has much lower memory bandwidth requirements. Each processor must keep track of which cache blocks are *dirty* - that is, that it has written to - again by adding a bit to the cache tag. If it sees a memory access for a word in a cache block it has marked as dirty, it intervenes and provides the (updated) value. There are numerous other issues to address when considering cache coherence.

One approach to maintaining coherence is to recognize that not every location needs to be shared (and in fact most don't), and simply reserve some space for non-cacheable data such as semaphores, called a coherency domain.

Using a fixed area of memory, however, is very restrictive. Restrictions can be reduced by allowing the MMU to tag segments or pages as non-cacheable. However, that requires the OS, compiler, and programmer to be involved in specifying data that is to be coherently shared. For example, it would be necessary to distinguish between the sharing of semaphores and simple data so that the data can be cached once a processor owns its semaphore, but the semaphore itself should never be cached.

In order to remove this data inconsistency there are a number of approaches based on hardware and software techniques few are given below:

- No caches is used which is not a feasible solution
- Make shared-data non-cacheable this is the simplest software solution but produce low performance if a lot of data is shared
- software flush at strategic times: e.g., after critical sections, this is relatively simple technique but has low performance if synchronization is not frequent
- hardware *cache coherence this can be achieved by making* memory and caches coherent (consistent) with each other, in other words if the memory and other processors see writes then without intervention of the to software
- absolute coherence all copies of each block have same data at all times
- It is not necessary what is required is *appearance of absolute coherence that is done by making* temporary incoherence is OK (e.g., write-back cache)

In general a cache coherence protocols consist of the set of possible states in local caches, the state in shared memory and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. There are basically two kinds of protocols depends on how writes is handled

6.3.2 Snooping Cache Protocol (for bus-based machines);

With a bus interconnection, cache coherence is usually maintained by adopting a "snoopy protocol", where each cache controller "snoops" on the transactions of the other caches

and guarantees the validity of the cached data. In a (single-) multi-stage network, however, the unavailability of a system "bus" where transactions are broadcast makes snoopy protocols not useful. Directory based schemes are used in this case.

In case of snooping protocol processors perform some form of *snooping* - that is, keeping track of other processor's memory writes. ALL caches/memories see and react to ALL bus events. The protocol relies on global visibility of requests (ordered broadcast). This allows the processor to make state transitions for its cache-blocks.

Write Invalidate protocol

The states of a cache block copy changes with respect to read, write and replacement operations in the cache. The most common variant of snooping is a *write invalidate protocol*. In the example above, when processor A writes to X, it broadcasts the fact and all other processors with a copy of X in their cache mark it invalid. When another processor (B, say) tries to access X again then there will be a cache miss and either

(i) in the case of a write-through cache the value of X will have been updated (actually, it might not because not enough time may have elapsed for the memory write to complete - but that's another issue); or

(ii) in the case of a write-back cache processor A must spot the read request, and substitute the *correct* value for X.

Figure 6.14 Write back with cache

Figure 6. 15 Write through with cache

An alternative (but less-common) approach is *write broadcast*. This is intuitively a little more obvious - when a cached value is changed, the processor that changed it broadcasts the new value to all other processors. They then update their own cached values. The trouble with this scheme is that it uses up more memory bandwidth. A way to cut this is to observe that many memory words are *not* shared - that is, they will only appear in one cache. If we keep track of which words are shared and which are not, we can reduce the amount of broadcasting necessary. There are two main reasons why more memory bandwidth is used: in an invalidation scheme, only the *first* change to a word requires an invalidation signal to be broadcast, whereas in a write broadcast scheme all changes must be signaled; and in an invalidation scheme only the *first* change to *any* word in a cache block must be signaled, whereas in a write broadcast scheme every word that is written must be signaled. On the other hand, in a write broadcast scheme we do not end up with a cache miss when trying to access a changed word, because the cached copy will have been updated to the correct value.

Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of Memory Location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Write Broadcast for X	1	1	1
CPU B reads X		1	1	1

Figure 6.16 write back with broadcast

If different processors operate on different data items, these can be cached.

1. Once these items are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic.
2. If a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local.

In both cases, the coherence protocol does not add any overhead.

Write-through vs. Write-back

In a write-back cache, the snooping logic must also watch for reads that access main memory locations corresponding to dirty locations in the cache (locations that have been changed by the processor but not yet written back).

At first it would seem that the simplest way to maintain coherence is to use a write-through policy so that every cache can snoop every write. However, the number of extra writes can easily saturate a bus. The solution to this problem is to use a write-back policy, but that leads to additional problems because there can be multiple writes that do not go to the bus, leading to incoherent data.

One approach is called write-once. In this scheme, the first write is a write-through to signal invalidation to other caches. After that, further writes can occur in write-back mode as long as there is no invalidation. Essentially, the first write takes ownership of the data, and another write from another processor must first deal with the invalidation and may then take ownership. Thus, a cache line has four states:

Invalid

Valid unwritten (valid)

Valid written once (reserved)

Valid written multiple (dirty)

The last two states indicate ownership. The trouble with this scheme is that if a non-owner frequently accesses an owned shared value, it can slow down to main memory speed or slower, and generate excessive bus traffic because all accesses must be to the owning cache, and the owning cache would have to perform a broadcast on its next write to signal that the line is again invalid.

One solution is to grant ownership to the first processor to write to the location and not allow reading directly from the cache. This eliminates the extra read cycles, but then the cache must write-through all cycles in order to update the copies.

We can change the scheme so that when a write is broadcast, if any other processor has a snoop hit, it signals this back to the owner. Then the owner knows it must write through again. However, if no other processor has a copy (signals snooping), it can proceed to write privately. The processor's cache must then snoop for read accesses from other processors and respond to these with the current data, and by marking the line as snooped. The line can return to private status once a write-through results in a no-snoop response.

One interesting side effect of ownership protocols is that they can sometimes result in a speedup greater than the number of processors because the data resides in faster memory. Thus, other processors gain some speed advantage on misses because instead of fetching from the slower main memory, they get data from another processor's fast cache. However, it takes a fairly unusual pattern of access for this to actually be observed in real system performance.

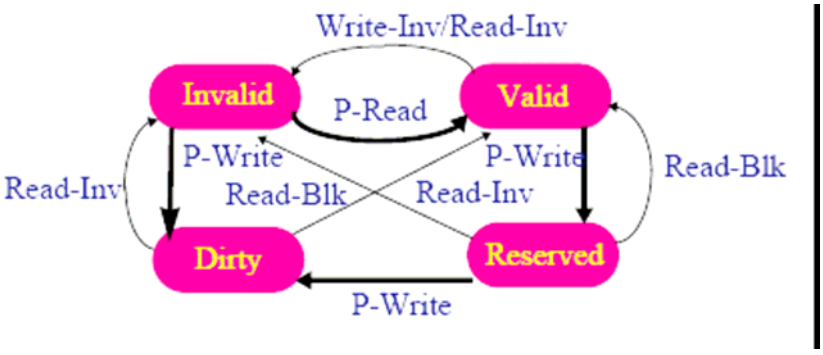


Figure 6.17 write once protocol

Disadvantages:

If multiple processors read and update the same data item, they generate coherence functions across processors.

Since a shared bus has a finite bandwidth, only a constant

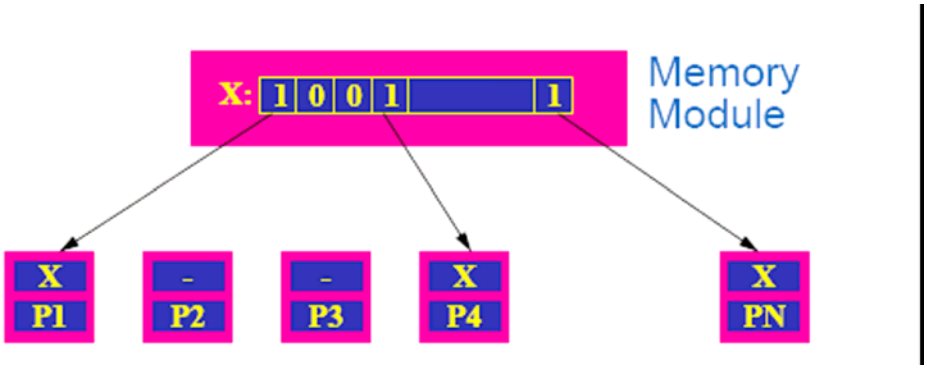
Rather than flush the cache completely, hardware can be provided to "snoop" on the bus, watching for writes to main memory locations that are cached.

Another approach is to have the DMA go through the cache, as if the processor is writing it to memory. This results in all valid cache locations. However, any processor cache accesses are stalled during that time, and it clearly does not work well in a multiprocessor, as it would require copies being written to all caches and a protocol for write-back to memory that avoids inconsistency.

Directory-based Protocols

When a multistage network is used to build a large multiprocessor system, the snoopy cache protocols must be modified. Since broadcasting is very expensive in a multistage network, consistency commands are sent only to caches that keep a copy of the block. This leads to *Directory Based protocols*. A directory is maintained that keeps track of the sharing set of each memory block. Thus each bank of main memory can keep a directory of all caches that have copied a particular line (block). When a processor writes to a location in the block, individual messages are sent to any other caches that have copies. Thus the Directory-based protocols selectively send invalidation/update requests to only those caches having copies—the sharing set leading the network traffic limited only to essential updates. Proposed schemes differ in the latency with which memory operations are performed and the implementation cost of maintaining the directory

The memory must keep a bit-vector for each line that has one bit per processor, plus a bit to indicate ownership (in which case there is only one bit set in the processor vector).



.figure 6.18 Directory based protocol

These bitmap entries are sometimes referred to as the presence bits. Only processors that hold a particular block (or are reading it) participate in the state transitions due to coherence operations. Note that there may be other state transitions triggered by processor read, write, or flush (retiring a line from cache) but these transitions can be handled locally with the operation reflected in the presence bits and state in the directory. If different processors operate on distinct data blocks, these blocks become dirty in the respective caches and all operations after the first one can be performed locally.

If multiple processors read (but do not update) a single data block, the data block gets replicated in the caches in the shared state and subsequent reads can happen without triggering any coherence overheads.

Various directory-based protocols differ mainly in how the directory maintains information and what information is stored. Generally speaking the directory may be central or distributed. Contention and long search times are two drawbacks in using a central directory scheme. In a distributed-directory scheme, the information about memory blocks is distributed. Each processor in the system can easily "find out" where to go for "directory information" for a particular memory block. Directory-based protocols fall under one of three categories:

Full-map directories, limited directories, and chained directories.

This full-map protocol is extremely expensive in terms of memory as it store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data.. It thus defeats the purpose of leaving a bus-based architecture.

A limited-map protocol stores a small number of processor ID tags with each line in main memory. The assumption here is that only a few processors share data at one time. If there is a need for more processors to share the data than there are slots provided in the directory, then broadcast is used instead.

Chained directories have the main memory store a pointer to a linked list that is itself stored in the caches. Thus, an access that invalidates other copies goes to memory and then traces a chain of pointers from cache to cache, invalidating along the chain. The actual write operation stalls until the chain has been traversed. Obviously this is a slow process .

Duplicate directories can be expensive to implement, and there is a problem with keeping them consistent when processor and bus accesses are asynchronous. For a write-through cache, consistency is not a problem because the cache has to go out to the bus anyway, precluding any other master from colliding with its access.

But in a write-back cache, care must be taken to stall processor cache writes that change the directory while other masters have access to the main memory.

On the other hand, if the system includes a secondary cache that is inclusive of the primary cache, a copy of the directory already exists. Thus, the snooping logic can use the secondary cache directory to compare with the main memory access, without stalling the processor in the main cache. If a match is found, then the comparison must be passed up to the primary cache, but the number of such stalls is greatly reduced due to the filtering action of the secondary cache comparison.

A variation on this approach that is used with write-back caches is called dirty inclusion, and simply requires that when a primary cache line first becomes dirty, the secondary line is similarly marked. This saves writing through the data, and writing status bits on every write cycle, but still enables the secondary cache to be used by the snooping logic to monitor the main memory accesses. This is especially important for a read-miss, which must be passed to the primary cache to be satisfied.

The previous schemes have all relied heavily on broadcast operations, which are easy to implement on a bus. However, buses are limited in their capacity and thus other structures are required to support sharing for more than a few processors. These structures may support broadcast, but even so, broadcast-based protocols are limited.

The problem is that broadcast is an inherently limited means of communication. It implies a resource that all processors have access to, which means that either they contend to transmit, or they saturate on reception, or they have a factor of N hardware for dealing with the N potential broadcasts.
Snoopy cache protocols are not appropriate for large-scale systems because of the bandwidth consumed by the broadcast operations

In a multistage network, cache coherence is supported by using cache directories to store information on where copies of cache reside.

A cache coherence protocol that does not use broadcast must store the locations of all cached copies of each block of shared data. This list of cached locations whether centralized or distributed is called a cache directory. A directory entry for each block of data contains a number of pointers to specify the locations of copies of the block.
Distributed directory schemes
In scalable architectures, memory is physically distributed across processors. The corresponding presence bits of the blocks are also distributed. Each processor is responsible for maintaining the coherence of its own memory blocks. Since each memory block has an owner its directory location is implicitly known to all processors. When a processor attempts to read a block for the first time, it requests the owner for the block. The owner suitably directs this request based on presence and state information locally available. When a processor writes into a memory block, it propagates an invalidate to the owner, which in turn forwards the invalidate to all processors that have a cached copy of the block. Note that the communication overhead associated with state update messages is not reduced. Distributed directories permit O(p) simultaneous coherence operations, provided the underlying network can sustain the associated state update messages. From this point of view, distributed directories are inherently more scalable

than snoopy systems or centralized directory systems. The latency and bandwidth of the network become fundamental performance bottlenecks for such systems.

7.4 Multicomputers

In spite of all of the work that had been done on networks, there were very few conscious design decisions involved with the first generation of multicomputers. For the most part, these machines (except for the Cosmic Cube and n-Cube) were built from whatever was at hand, with very little analysis of the design's performance.

In the case of the iPSC/1, Intel used 80286 processors together with off-the-shelf Ethernet transceivers wired in a 7-cube. The design was thrown together in less than a year in response to the realization that the group that spun out to form nCube might be onto something and should be beaten to delivery of their first systems.

The original iPSC had horrible performance, with up to 100ms latency for message transmission, and I/O only through the host node via another Ethernet link which did not support broadcast.

The iPSC/2 and Ametek machines were really the second generation, implementing wormhole routing on a mesh. The iPSC/2 retained the fiction of a hypercube for upward compatibility, but was implemented with a mesh. The n-Cube was actually more sophisticated than the first generation, given that it supported direct DMA transfers through the hypercube, but used store-and-forward for multi-hop messages. It was thus somewhere between the generations.

Fourth Generation Multicomputers

Now that communication is approaching the point that it is no longer the bottleneck in these machines, designers are at last beginning to realize that uniprocessors make poor multicomputer nodes. Thus, research in the area is likely to focus on the nodes, operating systems, and compilation and languages.

- Microthreading
- Lightweight processes
- Network access bypassing the OS, or more OS support in hardware
- Languages and compilers that handle partitioning and placement of data.

6.5 Message passing mechanisms

A message passing system (also referred to as distributed memory) typically combines the local memory and processor at each node of the interconnection network. There is no global memory, so it is necessary to move data from one local memory to another by means of message passing. This is typically done by a Send/Receive pair of commands, which must be written into the application software by a programmer. Thus, programmers must learn the message-passing paradigm, which involves data copying and dealing with consistency issues. Thus message passing in multicomputer network demands special hardware and software support.

A message is logical unit for internodes communication and it is assembled by an arbitrary number of fixed length packets. A packet may be a complete message, containing one or more data values, or a part of a longer message. In the latter case, the packet will have a sequence number indicating that it is the *i*th packet (out of *j* packets). Because we often deal with data transfers involving a single data element, we use *packet* and *message* interchangeably. A packet or message typically has a *header* that holds the destination address, plus possibly some routing information supplied by its sender, and a *message body* (sometimes called *payload*) that carries the actual data. Depending on the *routing algorithm* used, the routing information in the header may be modified by intermediate nodes. A message may also have various other control and error detection or correction information.

Message Passing Costs

The major overheads in the execution of parallel programs arise from communication of information between processing elements. The cost of communication is dependent on a variety of features including:

- programming model semantics,
- network topology,
- data handling and routing, and
- Associated software protocols.
- Time taken to communicate a message between two nodes in a network

= time to prepare a message for transmission + time taken by the message to traverse the network to its destination.

Parameters that determine the communication latency

Startup time (*ts*):

The startup time is the time required to handle a message at the sending and receiving nodes. Includes

1. the time to prepare the message (adding header, trailer & error correction information),
2. the time to execute the routing algorithm, and
3. the time to establish an interface between the local node and the router.

This delay is incurred only once for a single message transfer.

Per-hop time (*th*):

After a message leaves a node, it takes a finite amount of time to reach the next node in its path. The time taken by the header of a message to travel between two directly connected nodes in the network. It is also known as node latency. Is directly related to the latency within the routing switch for determining which output buffer or channel the message should be forwarded to.

Per-word transfer time (*tw*):

If the channel bandwidth is r words per second, then each word takes time $tw = 1/r$ to traverse the link. This time includes network as well as buffering overheads.

Message Passing Organization

Message passing systems are a class of multiprocessors in which each processor has access to its own local memory. Unlike shared memory systems, communications in message passing systems are performed via send and receive operations. A node in such a system consists of a processor and its local memory. Nodes are typically able to store messages in buffers (temporary memory locations where messages wait until they can be sent or received), and perform send/receive operations at the same time as processing. Simultaneous message processing and problem calculating are handled by the underlying operating system. Processors do not share a global memory and each processor has access to its own address space. Two important design factors must be considered in designing interconnection networks for message passing systems.

The **link bandwidth** is defined as the number of bits that can be transmitted per unit time (bits/s).The network **latency** is defined as the time to complete a message transfer.

Depending on the multiplicity of data sources and destinations, data routing or communication can be divided into two classes: one-to-one (one source, one destination) and collective (multiple sources and/or multiple destinations).

A processor sending a message to another processor, independent of all other processors, constitutes a *one-to-one data routing* operation. Such a data routing operation may be physically accomplished by composing and sending a point-to-point message. Typically, multiple independent point-to-point messages coexist in a parallel machine and compete for the use of communication resources. Thus, we are often interested in the amount of time required for completing the routing operation for up to p such messages, each being sent by a different processor. We refer to a batch of up to p independent point-to-point messages, residing one per processor, as a *data routing problem instance.* If exactly p messages are sent by the p processors and all of the destinations are distinct, we have a *permutation routing* problem.

6.5.1 Message routing scheme

Store-and-Forward Routing packets are the basic unit of information flow in store and forward network. Each node is required to use a packet buffer and it is transmitted from the source to designation through a sequence of intermediate node. The intermediate node store the entire message in buffer before passing it on

When a message is traversing a path with multiple links, each intermediate node on the path forwards the message to the next node after it has received and stored the entire message. Suppose that a message of size m is being transmitted through such a network. Assume that it traverses l links. At each link, the message incurs a cost th for the header and twm for the rest of the message to traverse the link. Since there are l such links, the total time is $(th + twm)l$. Therefore, for store-and-forward routing, the total communication cost for a message of size m words to traverse l communication links is.

Latency = [(message length / bandwidth) + fixed switch overhead] * #hops

In current parallel computers, the per-hop time th is quite small. For most parallel algorithms, it is less than twm even for small values of m and thus can be ignored.

Wormhole routing in message passing was introduced in 1987 as an alternative to the traditional store-and-forward routing in order to reduce the size of the required buffers and to decrease the message latency. In wormhole routing, a packet is divided into

smaller units that are called its (flow control bits) such that bits move in a pipeline fashion with the header bit of the packet leading the way to the destination node. When the header bit is blocked due to network congestion, the remaining bits are blocked as well. Switch passes message on before completely arrives and no buffering needed at switch. Latency (relative) independent of number of intermediate hops gives as below

Latency = (message length / bandwidth) + (fixed switch overhead * #hops)

Asynchronous pipelining: The pipelining of successive flits in a packet is done asynchronously using a handshaking protocol.

Virtual channels : A virtual channel is logical link between two nodes. It is formed by a flit buffer in the source node, a physical channel between them and a flit buffer in the receiver node. Four flit buffers are used at the source node and receiver node respectively. One source buffer is paired with one receiver buffer to form a virtual channel when the physical channel is allocated for the pair. Thus the physical channel is time shared by all the virtual channels. By adding the virtual channel the channel dependence graph can be modified and one can break the deadlock cycle. Here the cycle can be converted to spiral thus avoiding a deadlock. Virtual channel can be implemented with either unidirectional channel or bidirectional channels. However a special arbitration line is needed between adjacent nodes interconnected by bidirectional channel. This line determine the direction of information flow. The virtual channel may reduce the effective channel bandwidth available to each request. There exists a tradeoff between network throughput and communication latency in determining the degree of using virtual channels.

Flow control strategies:

A *routing mechanism:* Determine the path a message takes through the network to get from source to destination. It takes as input a message's source and destination nodes. It may also use information about the state of the network. It returns one or more paths through the network from the source to the destination

Classification based on route selection:

A *minimal* routing mechanism : Always selects one of the shortest paths between the source and the destination. Each link brings a message closer to its destination but it can lead to congestion in parts of the network.

A *nonminimal* routing scheme: This technique may route the message along a longer path to avoid network congestion.

Classification on the basis on information regarding the state of the network:

For messaging passing scheme the for smooth control on network traffic flow that involve no congestion or deadlock situation we require to develop some strategies such that if two or more packet collide at a node competing for the buffer the policies must be set to resolve the conflict. These polices can be deterministic or adaptive routing algorithm. In deterministic routing the communication path is completely determined by a unique path for a message, based on its source and destination address. Thus here the path is uniquely predetermined in advance, independent of network condition. It does not use any information regarding the state of the network and hence may result in uneven use of the communication resources in a network.

An *adaptive routing* scheme : it uses information regarding the current state of the network to determine the path of the message. It detects congestion in the network and routes messages around it. Adaptive routing may depend on the network condition and alternate paths are possible. In both types of routing deadlock free algorithm is desired.

The two deterministic algorithms are

a. Dimension ordering routing requires the selection of successive channels to follow a specific order based on the dimensions of multidimensional network. In case of two dimensional mesh networks the scheme is X-Y routing as the routing path along X-dimension is decided first before choosing a path along Y dimension. For hypercube (n-cube network) the scheme called E-cube routing is used.

XY-routing:

Consider a two-dimensional mesh without wraparound connections. A message is sent first along the X dimension until it reaches the column of the destination node and then along the Y dimension until it reaches its destination. Let PSy,Sx represent the position of the source node and PDy,Dx represent that of the destination node. Any minimal routing scheme should return a path of length |Sx - Dx| + |Sy - Dy|. Assume that Dx >= Sx and Dy >= Sx. The message is passed through intermediate nodes PSy,Sx+1, PSy,Sx+2, ..., PSy,Dx along the X dimension Then through nodes PSy+1,Dx, PSy+2,Dx, ..., PDy,Dx along the Y dimension to reach the destination.

E-Cube routing: Consider a *d*-dimensional hypercube of *p* nodes.

Let Ps and Pd be the labels of the source and destination nodes. We know that the binary representations of these labels are *d* bits long. The minimum distance between these nodes is given by the number of ones in Ps $\oplus$ Pd , where o represents the bitwise exclusive-OR operation. Node Ps computes Ps $\oplus$ Pd and sends the message along dimension *k*, where *k* is the position of the least significant nonzero bit in Ps $\oplus$ Pd. At each intermediate step, node Pi , which receives the message, computes Pi $\oplus$ Pd and forwards the message along the dimension corresponding to the least significant nonzero bit. This process continues until the message reaches its destination.

Let Ps = 010 and Pd = 111 represent the source and destination nodes for a message. Node Ps computes 010 $\oplus$ 111 = 101. In the first step, Ps forwards the message along the dimension corresponding to the least significant bit to node 011. Node 011 sends the message along the dimension corresponding to the most significant bit (011 $\oplus$ 111 = 100). The message reaches node 111, which is the destination of the message.

b. Adaptive network using virtual channels: the concept of virtual channels makes adaptive routing more economical and feasible to implement. The logic used is to extend the virtual channels in all connections along the same dimension of a mesh connected network.

Multicast routing algorithm collective data routing, as defined in the Message Passing Interface (MPI) Standard , may be of three types:

1. *One to many.* When a processor sends the same message to many destinations, we call the operation *multicasting.* Multicasting to all processors (one to all) is called *broadcasting.* When different messages are sent from one source to many destinations, a *scatter* operation is performed. The multiple destination nodes may be dynamically determined by problem- and data-dependent conditions or be a topologically defined subset of the processors (e.g., a row of a 2D mesh).

2. *Many to one.* When multiple messages can be merged as they meet at intermediate nodes on their way to their final common destination, we have a *combine* or *fan-in* operation (e.g., finding the sum of, or the maximum among, a set of values).

Combining values from all processors (all to one) is called *global combine.* If the messages reach the destination intact and in their original forms, we have a *gather*

operation. Combining saves communication bandwidth but is lossy in the sense that, in general, the original messages cannot be recovered from the combined version.

3. *Many to many.* When the same message is sent by each of several processors to many destinations, we call the operation *many-to-many multicasting.* If all processors are involved as senders and receivers, we have *all-to-all broadcasting.* When different messages are sent from each source to many (all) nodes, the operation performed is (*all-to-all*) *scatter-gather* (sometimes referred to as *gossiping*) .

The efficiency of any routing algorithm is determined by the two parameters mainly channel traffic and communication latency. The channel traffic at any time is indicated by the number of channel used to deliver the messages involved. The latency is indicated by the longest packet transmission time involved. Optimally routed network should achieve minimum for the both. In order to achieve better result virtual networks are used. Here the virtual channels can be used to generate virtual networks. These adding channels will increase the adaptively in making routing decisions at same time increase the cost. The concept of virtual network leads to portioning of a given physical network into logical subnetworks for multicast communication.

6.6 Summary

*Multiprocessors are based on concept of Shared Memory. There is o*ne shared address space. Different processors use conventional load/stores to access shared data. Communication can be complex / dynamic. Simpler programming model that is compatible with uniprocessors. Hardware controlled caching is useful to reduce latency and contention. The drawbacks of multiprocessors are

• Synchronization

• More complex hardware needed

Multiprocessors are an attractive way to increase performance.

Multicomputers uses the Message Passing technique for communication. Each processor has its own address space and processors send and receive messages to and from each other. Communication patterns explicit and precise Message passing systems: PVM, MPI, OpenMP. The interconnection networks such as time shared, Crossbar Switch and multiport memory has been discussed. Crossbar Switch is the most complex

interconnection system. There is a potential for the highest total transfer rate and the multiport memory is also an expensive memory units since the switching circuitry us included in the memory unit and the final table which has clearly differentiated among all three interconnection networks. Bus based systems are not scalable and not efficient for the processor to snoop and handle the traffic. Directories based system is used in cache coherence for large MPs *Cache coherency protocols maintain exclusive writes in a multiprocessor. Memory consistency policies determine how different processors observe the ordering of reads and writes to memory.* Snoopy caches are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. All processors snoop on (monitor) the bus for transactions. Directory based systems the global memory is augmented with a directory that maintains a bitmap representing cache-blocks and the processors at which they are cached

Multicomputers vs. multiprocessors

- They use similar networks; low latency and high bandwidth are crucial for both architectures
- Traditionally, multiprocessors have been better for fine-grain computations
- Multicomputers have been easier to scale to a large number of processors
- Multiprocessors are easier to program thanks to their shared address space

6.7 Keywords

Multicomputers: A computer in which processors can execute separate instruction streams, have their own private memories and cannot directly access one another's memories. Most multicomputers are *disjoint memory* machines, constructed by joining *nodes* (each containing a microprocessor and some memory) via *links*.

message passing A style of interprocess communication in which processes send discrete messages to one another. Some computer *architectures* are called message passing architectures because they support this model in hardware, although message passing has often been used to construct *operating systems* and *network* software for *uniprocessors* and *distributed computers*.

multiprocessor A computer in which processors can execute separate instruction streams, but have access to a single *address space*. Most multiprocessors are *shared*

memory machines, constructed by connecting several processors to one or more memory banks through a *bus* or *switch*.

cache consistency The problem of ensuring that the values associated with a particular variable in the *caches* of several processors are never visibly different.

6.8 Self assessment questions

1. With diagram, explain the interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory and shared peripherals.

2. Explain the bus systems at board level, backplane level and I/O level.

3. Explain the following terms associated with multicomputer networks and message-passing mechanisms: a) Message, packets and flits b) Store-and-forward routing at packet level c) wormhole routing at flit level d) Buffer deadlock versus channel deadlock d) Virtual networks and subnetworks e)Hot-spot problem.

4.Explain the hierarchical cache/bus architecture for designing a scalable multi-processor.

5. Describe the schematic design of a crosspoint switch in a crossbar network.

6. Discuss the multiport memory organizations for multiprocessor systems.

7. Explain the following : a)Two switch settings of an 8 X 8 Omega network built with 2 X 2 switches b) Broadcast capability of an Omega network built with 4 X 4 switches.

8. Describe the modular construction of butterfly switch networks with 8 X 8 crossbar switches.

9. Describe the Cache coherence problems in data sharing and in process migration.

10. Draw and explain 2 state-transition graphs for a cache block using write-invalidate snoopy protocols.

11. Explain the Goodman's write-once cache coherence protocol using the write-invalidate policy on write-back caches.

12. Discuss the basic concept of a directory-based cache coherence scheme.

13. Mention and explain the three types of cache directory protocols.

14. What is the communication latency and time comparison for a store-and-forward and wormhole-routed network?

15. Explain the following a) Dimension-order routing b) E-cube routing on hypercube c) X-Y routing on a 2D mesh and d) Adaptive routing.

6.9 References/Suggested readings

Advance Computer architecture: Kai Hwang