CHAPTER 1: INTRODUCTION

Why do we need to study computer organization and/or architecture ?

The computer lies to the heart of computing without it most of the computing disciplines today would be branch of theorititcal mathematics. To be a professional in any field of computing today, one should not regard the computer as just a black box that executes programs by magic. All students of computing should acquire some understanding and appereciation of a computer system's fuctional componensts, their characteristcis ,their performance and their interaction. Students needs to understand computer archtecture in order to structure a program so that it runs more efficiently on real machines.
Example: we need 6 bit system and there is avaible and there is available of 2bits,4 bits or 8 bits, now one should know which will be more beneficial using 2,2,2 bits bits or 4,4 bits system or 8 bit system and design the system according.

Choosing the best alternative for right organization.
Concept used in computer organization & computer architecture can be used o other courses fields.

Computer architecture:

- Computer architecture refers to those attributes of a system visible to a programmer or that have direct impact on the logical execution of a program.
- Computer architecture is the study of the structure, behavior and design of a computer system.

Examples of architectural attributes include:

- Instruction set.
- Number of bits used to represent various data types.
- I/O mechanism and techniques for addressing memory.

Computer Organization:

Computer organization refers to the operational units and their interconnection that realize the architectural specification.

- Organization attributes include those hardware details transparent to the programmer, such as control signals, interfaces, technologies used.

For example, it is an architectural design issue whether a computer will have multiple instruction or not, it is an organizational issue whether that instruction will be implemented by a special multiply unit or by a mechanism of repeated addition. The organization decision may be based on the anticipated frequency of use of multiply instruction, the relative speed of the approaches and the cost of the physical size of special multiply unit. The computer architecture of a computer system may be same with different organization.

History of computer/ Miles stones in computer organization

Mechanical era (1623-1945):

- The idea of using machines to solve mathematical problems can be traced back to 17th century where mathematician designed and implemented calculators that were capable of addition, subtraction, multiplication & division included Wilhelm Schickhard, Blaise Pascal, Gottfried Leibnitz.

- The first multi-purpose programmable computing device was Charles Babbage's difference and analytical engine.

- First commercial use of mechanical computers was in US census Bureau by Herman Hollerith.

Two major drawbacks of mechanical computers are :

1. Speed of operation limited by the inertia of moving parts.
2. Cumbersome unreliable and expensive.

Electronic era

First Generation (1945-1956)

- Vacuum tubes for circuitry, magnetic drum for memory.

- Enormous high power.

- ENIAC(Enhanced Numeric Integrator and Computer)

Second generation (1956-1963)

- Transistors.

- Smaller, faster, cheap, more energy efficient more reliable.

- More complex arithmetic and logic unit, control units.

Third generation (1964-1971)

- Integrated circuits, SSI (Small Scale Integration)-1 million transistors.

- LSI (Large Scale Integration)-10 million transistors.

- Third generation computer through keyboards and monitors and interface with an operating system, which allowed the device to run many different applications at one time with central program that monitored the memory.

Fourth Generation (1971-present)

- Used micro-processor, fourth generation of computers as thousands of integrated circuits were built on a single silicon chip.
- Intel 4004 chip developed in 1971 located all the components of the computer from CPU to memory to I/O controls on a single chip.
- IBM introduced first computer, 1984 apple macintosh

Fifth Generation (Present)

- Based on artificial intelligence, still development phases.
- Voice recognition, self-decision, nano technology, self-organization.

Moore's law: Moore observed that the number of transistors that would be put on a single chip was doubling every year and was corrected many years. This pace slowed to a doubling every 18 months in 1970.

Von neuman machine/IAS computer: imp

It was designed by the Princeton institute for advanced studies (IAS)

It consists of:

- Main memory: which stores both data and instruction.
- Arithmetic and logic unit: capable of operating on binary data.
- A control unit which interprets the instruction in memory and causes them to be executed.
- Input & Output equipment operated by control unit.

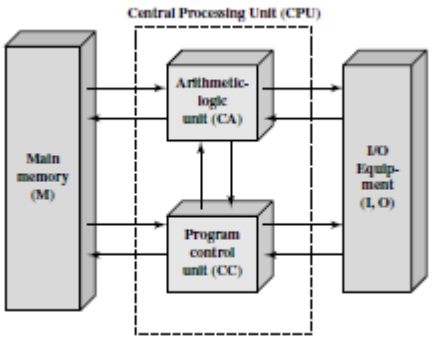


Figure 2.1 Structure of the IAS Computer

Memory location

- 1000 storage locations called words of 40 binary digits (bits).
- Both data and instruction stored.
- All numbers and data represents in binary form.
- Each number represented by 1 sign bit, 39 bits value or two 20 bits instruction consisting of 8 bits opcode, 12 bits addressing address.

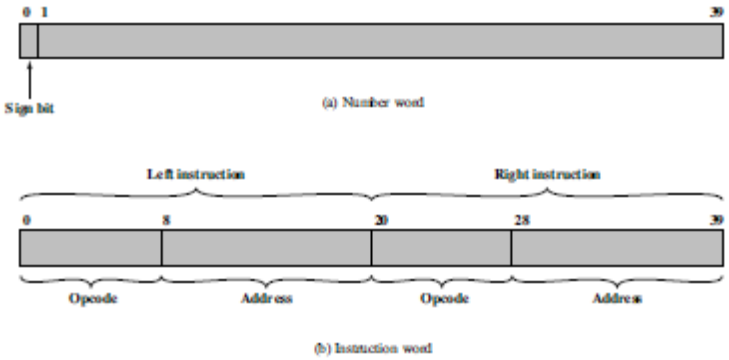


Figure 2.2 IAS Memory Formats

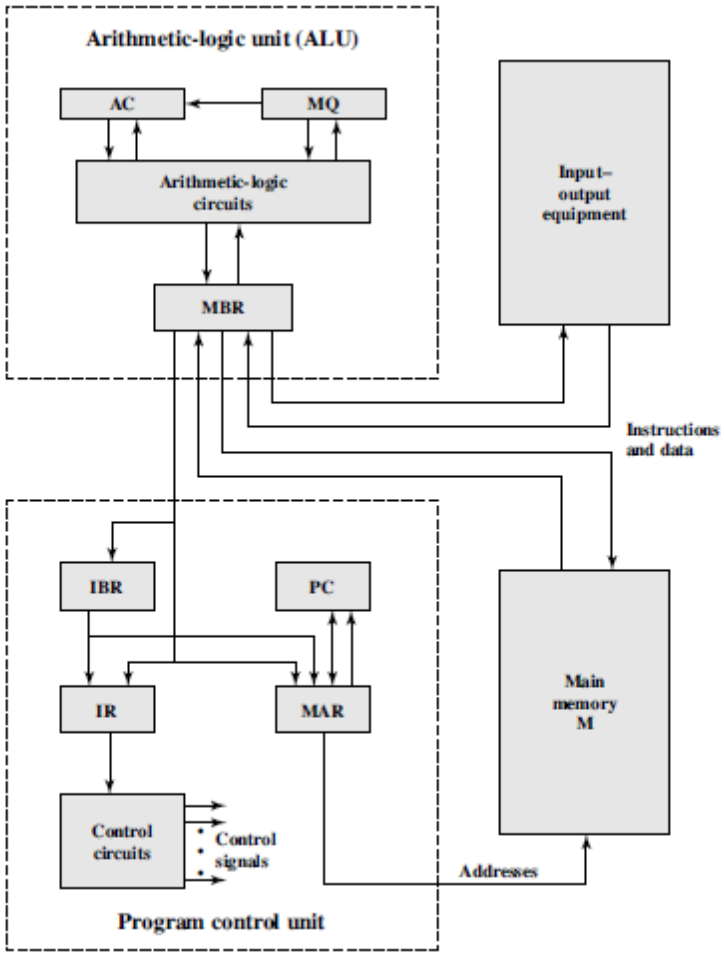


Figure 2.3 Expanded Structure of IAS Computer

Different registers

Memory buffer register (MBR): Contains a word to be stored in memory or sent to the I/O unit, or is used to receive a word from memory or from the I/O unit.

Memory address register (MAR): Specifies the address in memory of the word to be written from or read into the MBR.

Instruction register (IR): Contains the 8-bit opcode instruction being executed.

Instruction buffer register (IBR): Employed to hold temporarily the right hand instruction from a word in memory.

Program counter (PC): Contains the address of the next instruction-pair to be fetched from memory.

Accumulator (AC) and multiplier quotient (MQ): Employed to hold temporarily operands and results of ALU operations.

Operation

IAS operates by repetitively performing an instruction cycle. Instruction cycle contains fetch and execute cycle.

Fetch cycle

- The opcode of the next instruction is loaded into IR and addr. portion into MAR.
- The instruction can be taken from IBR from MBR and then to IBR, IR and MAR.

Execute cycle

- Control circuitry interprets the opcode and executes the instruction by sending out the appropriate control signal to cause data to be moved or perform ALU operations.

The total of 21 instructions that can be grouped onto:

1. **Data transfer:** between memory and/ OR ALU registers.

2. **Unconditional branch:** Normally, the control unit executes instructions in sequence from memory. This sequence can be changed by a branch instruction, which facilitates repetitive operations.

3. **Conditional branch:** The branch can be made dependent on a condition, thus allowing decision points.

4. **Arithmetic:** Operations performed by the ALU.

5. **Address modify:** Permits addresses to be computed in the ALU and then inserted into instructions stored in memory. This allows a program considerable addressing flexibility.

Table 2.1 The IAS Instruction Set

Instruction Type	Opcode	Symbolic Representation	Description
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD −M(X)	Transfer −M(X) to the accumulator
	00000011	LOAD |M(X)|	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD −|M(X)|	Transfer −|M(X)| to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP+ M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP+ M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)
Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD |M(X)|	Add |M(X)| to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB |M(X)|	Subtract |M(X)| from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2; i.e., shift left one bit position
	00010101	RSH	Divide accumulator by 2; i.e., shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

Table: IAS instruction set

** see intel/Pentium generations table from william stalling book Computer Organization and

Architecture Designing for Performance

Operations

-IAS operates by repetitively performing an instruction cycle.
-instruction cycle- fetch and execute cycle

Fetch cycle

- op-code of the next instruction is loaded into IR & address portion into MAR.
- The instruction can be taken from IBR or from MBR and then to IBR, IR and MAR

Execute cycle

- Control circuitry interprets the op-code and executes the instruction by sending out the appropriate control signals to cause data to be moved or perform ALU operations.

The total of 21 instruction can be grouped into:

1. data transfer: between memory and OR ALU register.
2. Conditional branch
3. Un-conditional branch
4. Arithmetic
5. Address modify: permits address to be computed in the ALU and then inserted into instruction in memory

Partial flowchart of IAS operation from William Stalling book.

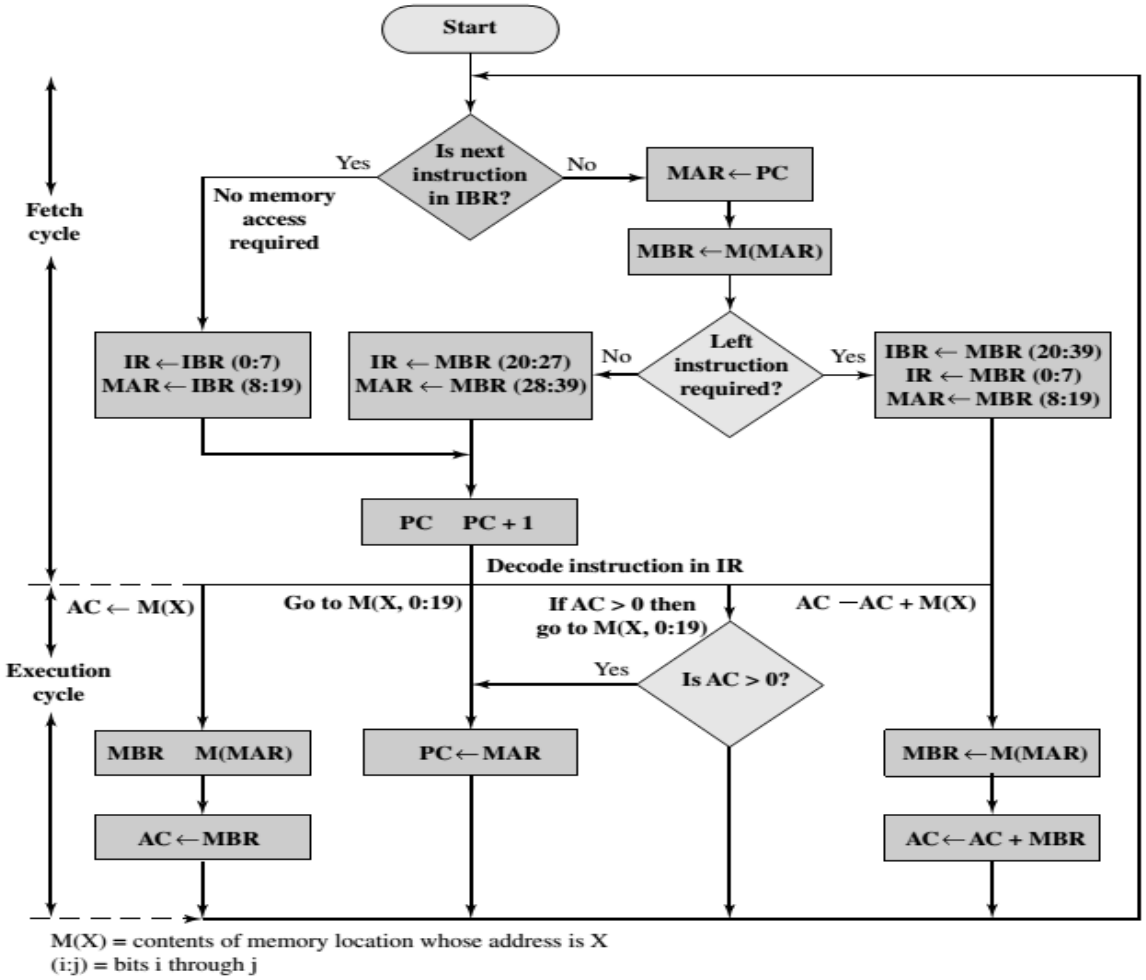


Figure 2.4 Partial Flowchart of IAS Operation

Functional view of computer

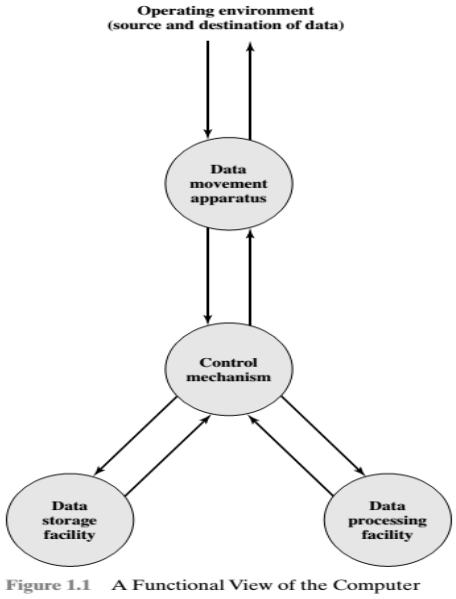


Figure 1.1 A Functional View of the Computer

Functions

- Data processing
- Data storage
- Data movement
- Control

Instruction Fetch and Execute

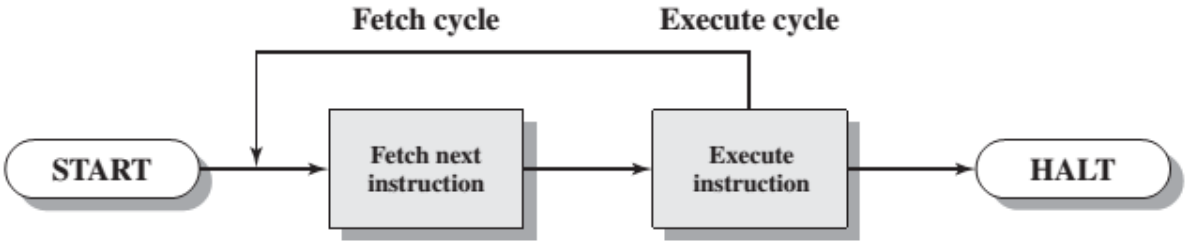


Figure 3.3 Basic Instruction Cycle

At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next. Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence. So, for example, consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence can also be altered as required. The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

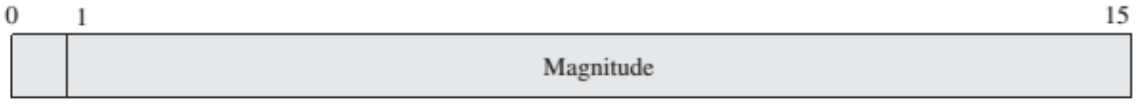
Processor-I/O: Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.

Data processing: The processor may perform some arithmetic or logical operations on data.

Control: An instruction may specify that the sequence of execution be altered.

Example of fetch and execute cycle

(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

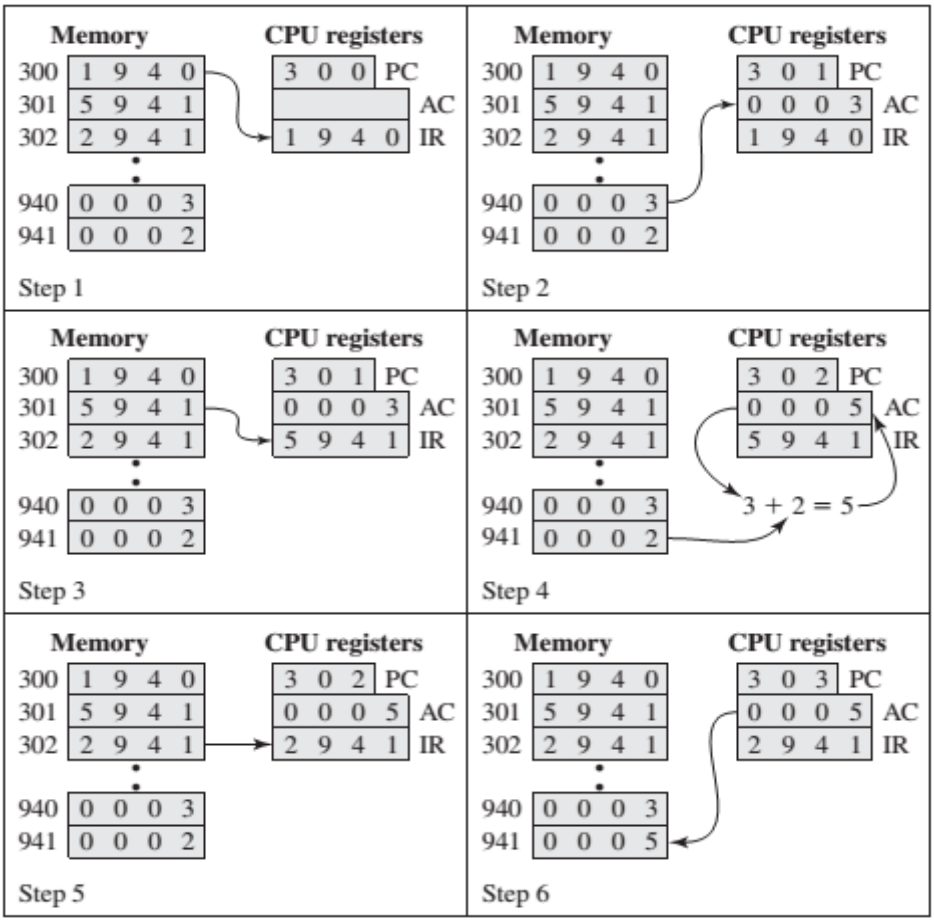


Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

Figure 3.5: Example of Program Execution (contents of memory and registers in hexadecimal) Figure 3.5 illustrates a partial program execution, showing the relevant portions of memory and processor registers. The program fragment shown adds the contents of the
memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute cycles, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR and the PC is incremented. Note that this process involves the use of a memory address register (MAR) and a memory buffer register (MBR). For simplicity, these intermediate registers are ignored.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301 and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.

5. The next instruction (2941) is fetched from location 302 and the PC is incremented.
6. The contents of the AC are stored in location 941.

Instruction cycle state diagram:

imp

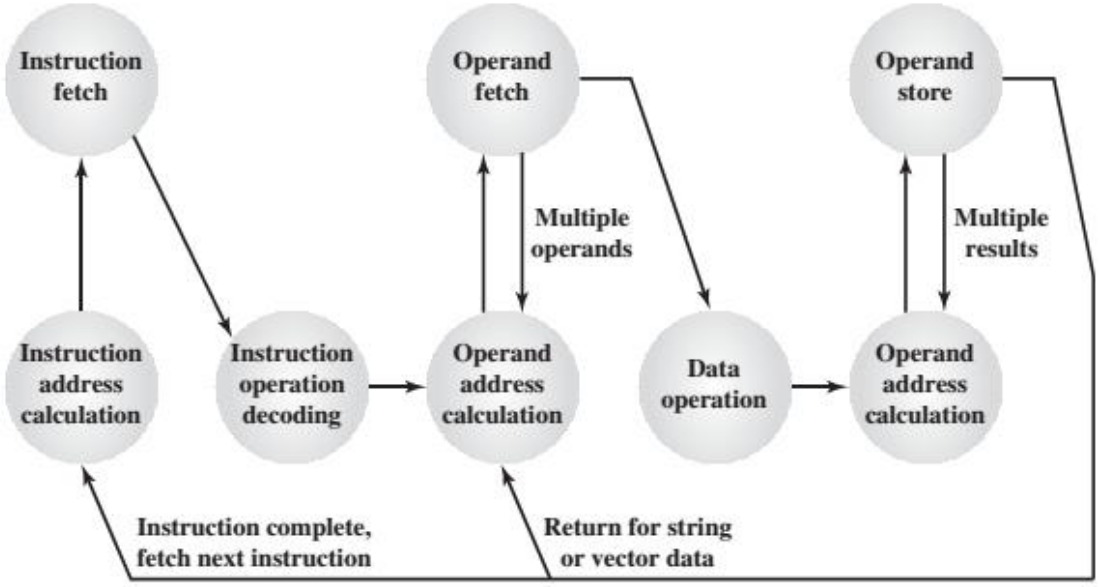


Figure 3.6 Instruction Cycle State Diagram

The above figure shows the sate diagram of instruction cycle, for any given instruction cycle. Some states may be null and other may be visited more than once. The different states can be described as:

- ***Instruction address calculation (iac):*** Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.
- **Instruction fetch (if):** Read instruction from its memory location into the processor.
- ***Instruction operation decoding (iod):*** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand address calculation (oac):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
- ***Operand fetch (of):*** Fetch the operand from memory or read it in from I/O.
- ***Data operation (do):*** Perform the operation indicated in the instruction.
- ***Operand store (os):*** Write the result into memory or out to I/O.

States in the upper part of Figure 3.6 involve an exchange between the processor and either memory or an I/O module. States in the lower part of the diagram involve only internal processor operations. The oac state appears twice, because an instruction may involve a read, a write, or both. However, the action performed during that state is fundamentally the

same in both cases, and so only a single state identifier is needed. Also note that the diagram allows for multiple operands and multiple results, because some instructions on some machines require this.

Interrupts in instruction cycle

Interrupts is an asynchronous service request from hardware or software to CPU. Interrupts make the CPU execution of it's normal operation to pause to service external devices or errors. The processor and the OS are responsible for recognizing in interrupt suspending the user program, servicing the interrupt and then resuming the user program. The instruction cycle with interrupts is as shown below:

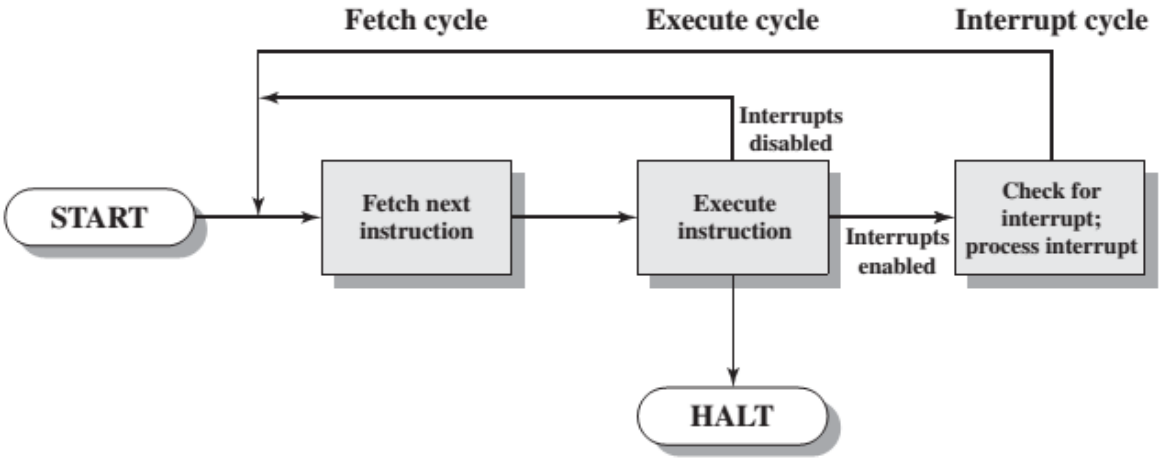


Figure 3.9 Instruction Cycle with Interrupts

imp

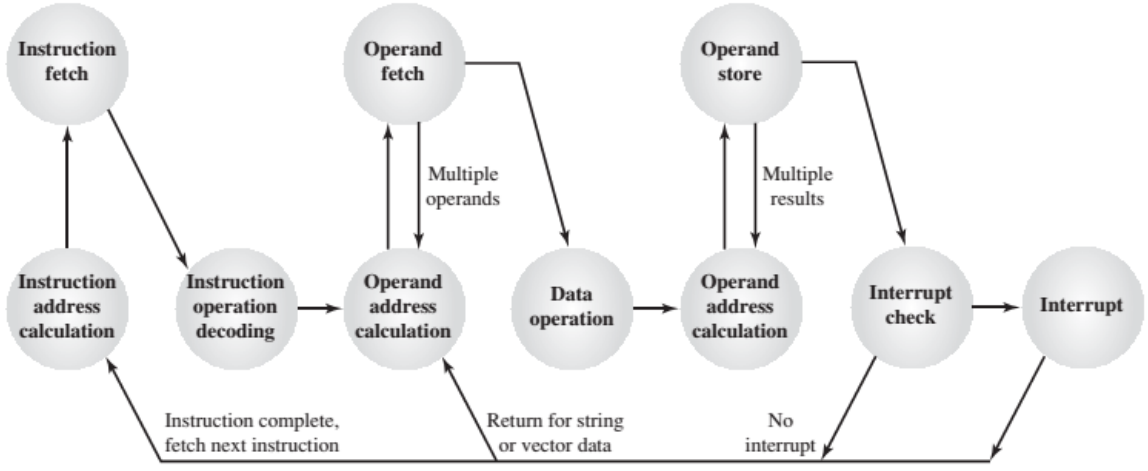


Figure 3.12 Instruction Cycle State Diagram, with Interrupts

Instruction format

An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields. An instruction format must include an opcode and, implicitly or

explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes. The format must, implicitly or explicitly, indicate the addressing mode for each operand. Some of the key design issue are:

Instruction Length

The decision of instruction format length is affected by, memory size, memory organization, bus structure, processor complexity, and processor speed. The most obvious trade-off here is between the desire for a powerful instruction and a need to save space. Programmers want more opcodes, more operands, more addressing modes, and greater address range. More opcodes and more operands make life easier for the programmer, because shorter programs can be written to accomplish given tasks. Similarly, more addressing modes give the programmer greater flexibility in implementing certain functions, such as table manipulations and multiple-way branching. And, of course, with the increase in main memory size and the increasing use of virtual memory, programmers want to be able to address larger memory ranges. All of these things (opcodes, operands, addressing modes, address range) require bits and push in the direction of longer instruction lengths. But longer instruction length may be wasteful. Another trade off, consideration may be either instruction length should be equal to the memory transfer length or multiple, otherwise we will not get an integral number of instruction during fetch cycle.

Allocation of Bits

For a given instruction length, there is clearly a trade-off between the number of op-codes and the power of the addressing capability. More op-codes obviously mean more bits in the op-code field. For an instruction format of a given length, this reduces the number of bits available for addressing. There is one interesting refinement to this trade-off, and that is the use of variable-length op-codes. In this approach, there is a minimum op-code length but, for some op-codes, additional operations may be specified by using additional bits in the instruction. For a fixed-length instruction, this leaves fewer bits for addressing. Thus, this feature is used for those instructions that require fewer operands and/or less powerful addressing. The following interrelated factors go into determining the use of the addressing bits.

- **Number of addressing modes:** Sometimes an addressing mode can be indicated implicitly. For example, certain op-codes might always call for indexing. In other cases, the addressing modes must be explicit, and one or more mode bits will be needed.

Number of operands: Typical instructions on today's machines provide for two operands. Each operand address in the instruction might require its own mode indicator, or the use of a mode indicator could be limited to just one of the address fields.

- **Register versus memory:** With a single user-visible register (usually called the accumulator), one operand address is implicit and consumes no instruction bits. However, single-register programming is awkward and requires many instructions. Even with multiple registers, only a few bits are needed to specify the register. The more that registers can be used for operand references, the fewer bits are needed.
- **Number of register sets:** Machines have one set of general purpose registers, with typically 32 or more registers in the set. These registers can be used to store data and can be used to store addresses for displacement addressing. Some architectures have a collection of two or more specialized sets (such as data and displacement). One advantage

of this latter approach is that, for a fixed number of registers, a functional split requires fewer bits to be used in the instruction. For example, with two sets of eight registers, only 3 bits are required to identify a register; the op-code or mode register will determine which set of registers is being referenced.

• **Address range:** the range of addresses that can be referenced is related to the number of address bits. Because this imposes a severe limitation, direct addressing is rarely used. With displacement addressing, the range is opened up to the length of the address register. Even so, it is still convenient to allow rather large displacements from the register address, which requires a relatively large number of address bits in the instruction.

• **Address granularity:** In a system with 16 or 32 bits word or address can reference a word or a byte at designers choice. Byte addressing is convenient for character manipulation but required for a fixed sixe of memory more address bits.

Variable length instruction: The instruction discussed so far have single fixed length, but the designer may choose instead to provide a variety of instruction formats of different lengths, this tactic makes it easy to provide a large no. of op-codes with different lengths and the address length can also be varied with variable length instruction, these many variations can be provided efficiently and compactly.

Instruction representation

Within the computer each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituents elements of the instruction. It is difficult for both the programmer and reader to deal with binary representation, so it is

common to use symbolic representation. Code are represented by abbreviations called nmemonics that indicate the operations.

ADD: Add, SUB: Subtract, MUL: Multiply, DIV: Divide.

Instruction types

An instruction set should be functionally complete. It should formulate any high level data processing task. Such operations can be grouped as:

1. Data processing: ALU
2. Data movement: I/O instruction
3. Data storage: memory instruction
4. Control operations: test and branch instruction.

Good instruction set should meet following standards:

1. Completeness: to be able to construct a machine level program to evaluate any computable function.
2. Efficient: frequently performed instruction should be done quickly with few instructions.
3. Regular and complete class of instruction: provide logical set of operations.
4. Orthogonal: define instruction, data types and addressing independently.
5. Compatible: with existing hardware and software product of that time.

Number of address/ address in an instruction

In a typical arithmetic or logical instruction 3 address is required, 2 for operands and 1 for result. These address can be explicitly given or implied by instruction. Example: compare instruction Y= (A-B)/[C+ (D*E)] with one, two and three instructions

3-address instruction

Both operands and the destination for the result are explicitly contained in the instruction word.

Instruction	Comment
SUB Y,A,B	Y←A-B
MUL T,D,E	T←D*E
ADD T,T,C	T←T+C
DIV Y,Y,T	Y←Y/T

This format is rarely used due to the length of address themselves and resulting length of the instruction word.

2-address instruction

One of the address is used to specify both an operand and result location. Very common in instruction set.

Instruction	Comment
MOV Y,A	Y←A
SUB Y,B	Y←Y-B
MOV T,D	T←D
MUL T,E	T←T*E
ADD T,C	T←T+C
DIV Y,T	Y←Y/T

1-address instruction

Traditional accumulator based operations, ACC←ACC+X

Instruction	Comment
LOAD D	AC←D
MUL E	AC←AC*E
ADD C	AC←AC+C
STOR Y	Y←AC
LOAD A	AC←A
SUB B	AC←AC-B
DIV Y	A←AC/Y
STOR Y	Y←AC

0 address instruction

Zero address instruction are applicable to a special memory organization called stack. A stack is a last-in-first-out set of locations. Zero address instruction would reference the top of the two stack elements.

PUSH A	TOS←A
PUSH B	TOS←B
SUB	TOS←(A-B)
PUSH D	TOS←D
PUSH E	TOS←E
MUL	TOS←D*E
PUSH C	TOS←C
ADD	TOS←C+(D*E)
DIV	TOS←(A-B)/(C+D*E)
POP Y	Y←TOS

Addressing mode

The manner in which each address field specifies memory location is called addressing modes. That can be of following

1. Immediate mode:

The operand is contained within the instruction itself. Data is a constant at run time. No additional memory reference are required after the fetch of the instruction. Size of the operand is limited, i.e range of value limited.

Operand A

2. Direct addressing mode:

The address field of the instruction contains the effective address of the operand. No calculation required. One additional memory access is required to fetch the operand. Address range is limited by the width of the field that contains the address reference. Address is a constant at run time but data itself can be changed during the program execution.

EA=A

3. Indirect addressing:

The address filed in the instruction specified a memory location which contains the address of the data. In indirect addressing, address field refer to the address of a word in memory which in-turn contains a full length address of operand.

4. Register based addressing modes

Register addressing: similar to direct addressing the only difference is that the address field refers to a register rather than a main memory address.

EA=R

Register indirect: like indirect, but address filed specifies a register that contains the effective address.

Advantage

i. Only a small address filed is needed in the instruction.
ii. No time consuming memory reference are required.

Dis-advantage

i. If registers are heavily used, this will limit the performance of the processor.
ii. If frequently used there will be more immediate steps involved. So only used if the operand in a register remains in use for multiple operations.

5. Displacement or address relative addressing

EA=A+(R)

Displacement address requires that the instruction have two address filed at least one of which is explicit (definite). The value contained in one address field (value=A) is directly added to a register to produce effective address. The most commonly used displacement addressing:

a. Relative addressing:

- A is added to the program counter contents to cause a branch operations in fetching the next instruction.

b. Base register addressing:

- The referenced register contains a main memory address and the address register field contains a displacement (usually unsigned integer) from that address.

c. Indexing:

The address field references a main memory address and the referenced register contains a positive displacement from that address.

6. Stack addressing:

A stack is a linear array of location. It is sometime referred to as a pushdown list or last-in-first-out queue. Items are appended to the top of the stack so that at any given time the block is partially filled. Associated with stack is a pointer whose value is the address of the top of the stack. Alternatively, the top two elements of the stack may be in processor register, in which case the stack pointer references the third element of the stack.

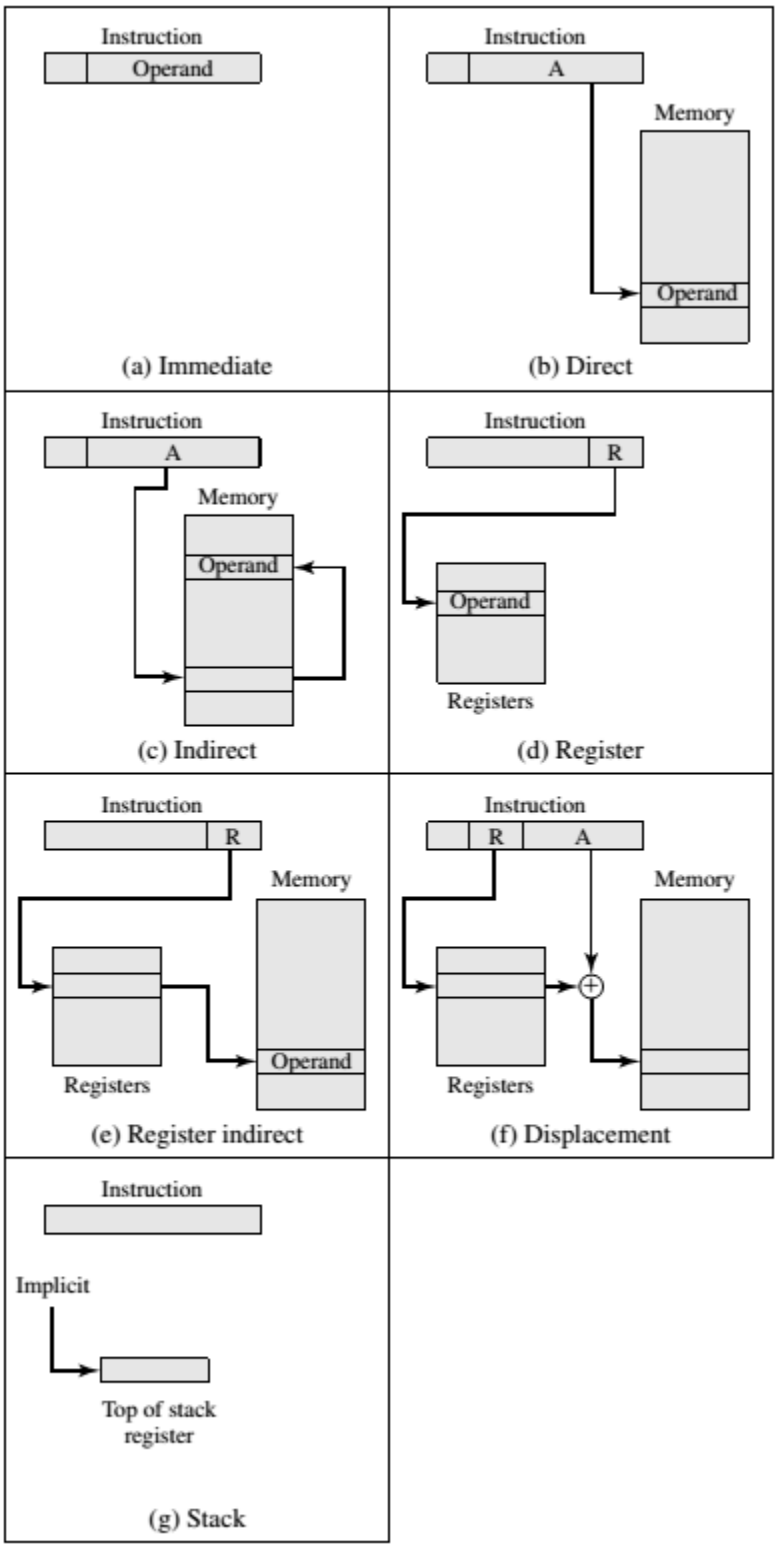


Figure 11.1 Addressing Modes

Stack, reverse polish notation
A+B: infix notation
+AB: prefix or polish notation
AB+: postfix or reverse polish notation
(A*B+C*D) in reverse polish notation is AB*CD*+

Scan the expression from let to right, when an operator is reached perform the operation with operands found on the left side of the operator. Remove the two operands and the operator and replace them by the number obtained from the result of the operation. Continue to scan the expression and repeat the procedure for every operator encountered until there are no more operators.

Example: (A+B)*[C*(D+E) +F]

Convert to reverse polish notation
- First perform all arithmetic inside the inner parameters.
- Then inside outer parentheses.
- Do multiplication and division before addition and subtraction operations.

Final result is: AB+DE+C*F+*

****Note:** examples of computer families, see table 2.3 for IBM 700/7000 series, IBM system/360

The evolution of intel x86 architecture, see table 2.6 from William stallings

Future trends in computer

- ARM, embedded system

Designing computer for performance
Year by year, the cost of computer systems continues to drop dramatically, while the performance and capacity of those systems continue to rise equally dramatically. Most of the today's computer requires some of the following capabilities:
• Image processing
• Speech recognition
• Videoconferencing
• Multimedia authoring
• Voice and video annotation of files
• Simulation modeling

Basically the building blocks for today's computer are virtually same as those of the IAS computer, but the techniques for making performance high has improved a lot and sophisticated. Now we highlight some of the driving factors behind the need to design for performance.

- **Microprocessor Speed**

What gives these processors such mind-boggling power is the relentless pursuit of speed by processor chip manufacturers. The evolution of these machines continues to bear out Moore's law, mentioned previously. So long as this law holds, chipmakers can unleash a new generation of chips every three

Years with four times as many transistors. In memory chips, this has quadrupled the capacity of dynamic random-access memory (DRAM), still the basic technology for computer main memory, every three years. In microprocessors, the addition of new circuits, and the speed boost that comes from reducing the distances between them, has improved performance four or fivefold every three years. But the raw speed of the microprocessor will not achieve its potential unless it is fed a constant stream of work to do in the form of computer instructions. Anything that gets in the way of that smooth flow undermines the power of the processor. Accordingly, while the chipmakers have been busy learning how to fabricate chips of greater and greater density, the processor designers must come up with ever more elaborate techniques for feeding the monster. Among the techniques built into contemporary processors are the following:

- **Branch prediction:** The processor looks ahead in the instruction code fetched from memory and predicts which branches, or groups of instructions, are likely to be processed next. If the processor guesses right most of the time, it can prefetch the correct instructions and buffer them so that the processor is kept busy. The more sophisticated examples of this strategy predict not just the next branch but multiple branches ahead. Thus, branch prediction increases the amount of work available for the processor to execute.
- **Data flow analysis:** The processor analyzes which instructions are dependent on each other's results, or data, to create an optimized schedule of instructions. In fact, instructions are scheduled to be executed when ready, independent of the original program order. This prevents unnecessary delay.
- **Speculative execution:** Using branch prediction and data flow analysis, some processors speculatively execute instructions ahead of their actual appearance in the program execution, holding the results in temporary locations. This enables the processor to keep its execution engines as busy as possible by executing instructions that are likely to be needed.
- **Performance Balance**

  While processor power has raced ahead at breakneck speed, other critical components of the computer have not kept up. The result is a need to look for performance balance: an adjusting of the organization and architecture to compensate for the mismatch among the capabilities of the various components.

Consider an example the speed of processor has increased tremendously or rapidly but not the speed of data transfer from main memory to processor, so the processor has to wait for the data from memory to come causing processor stall or wait for data causing overall slow performance. The no of ways the system architect can solve this problem are:

- Increasing the number of bits that are retrieved from memory.
- Introducing a cache or other buffering schemes.
- Reducing the access to memory incorporating more caches.
- Using high speed buses for faster data transfer.

- Another example I/O peripheral devices. The key idea here is to balance the throughput and processing demands of the processor, main memory, I/O devices, inter connecting structures.

Improvements in Chip Organization and Architecture

As designers wrestle with the challenge of balancing processor performance with that of main memory and other computer components, the need to increase processor speed remains. There are three approaches to achieving increased processor speed:

- **Increase the hardware speed of the processor:** This increase is fundamentally due to shrinking the size of the logic gates on the processor chip, so that more gates can be packed together more tightly and to increasing the clock rate. With gates closer together, the propagation time for signals is significantly reduced, enabling a speeding up of the processor. An increase in clock rate means that individual operations are executed more rapidly.

- Increase the size and speed of caches that are interposed between the processor and main memory. In particular, by dedicating a portion of the processor chip itself to the cache, cache access times drop significantly.

- Make changes to the processor organization and architecture that increase the effective speed of instruction execution. Typically, this involves using parallelism in one form or another.

Here are some obstacles or factors that limit or hold back in increasing the performance using above method.

- **Power:** As the density of logic and the clock speed on a chip increase, so does the power density (Watts/cm2). The difficulty of dissipating the heat generated on high-density, high-speed chips is becoming a serious design issue.

- **RC delay:** The speed at which electrons can flow on a chip between transistors is limited by the resistance and capacitance of the metal wires connecting them; specifically, delay increases as the RC product increases. As components on the chip decrease in size, the wire interconnects become thinner, increasing resistance. Also, the wires are closer together, increasing capacitance.

- **Memory latency:** Memory speeds lag processor speeds. Difference between access time of processor and memory.

Thus, there will be more emphasis on organization and architectural approaches to improving performance. Two main strategies that have been used to increase performance beyond what can be achieved simply by increasing clock speed are:

- Increasing cache capacity, using two or more levels of caches.
- Using parallel execution, like pipelining & superscalar methods.

Evolution of Intel X86 architecture see William stallings book on computer organization and architecture designing for performance.

Register Transfer and Micro operations

imp

Register Transfer and Register Transfer language(RTL)

Register Transfer Language

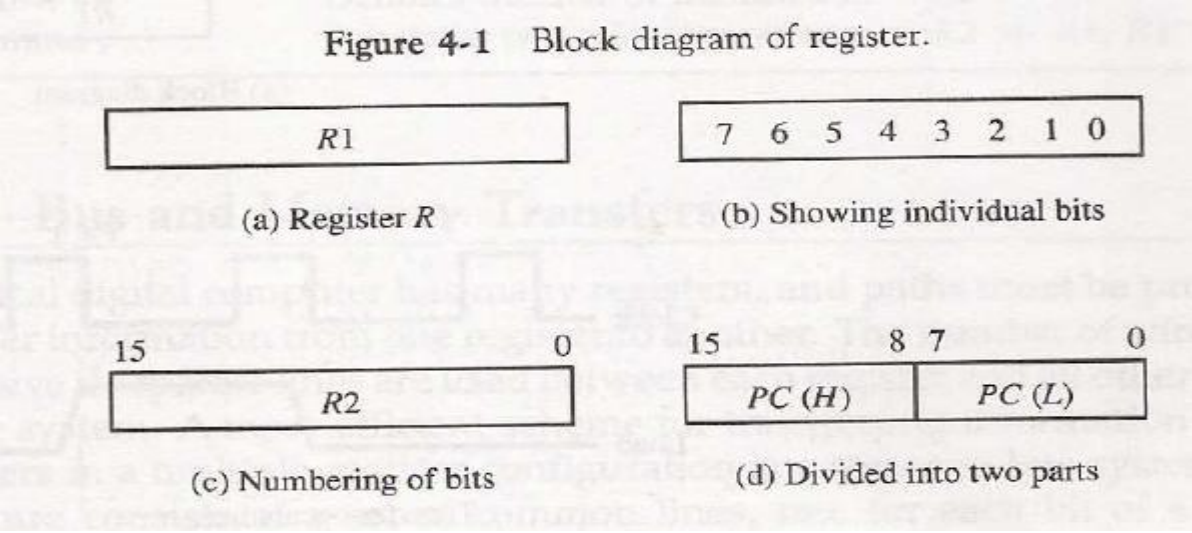
A digital system is an interconnection of digital hardware modules that accomplishes a specific information-processing task. Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored is registers are called micro-operations. A micro-operation is an elementary operation performed on the information stored in one or more registers. The result of the operations may replace the previous binary information of a register or may be transferred to another register. Examples of micro-operations are shift, count, clear and load.

The internal hardware organization of a digital computer is best defined by specifying:

1) The set of registers it contains and their function.
2) The sequence of micro-operations performed on the binary information stored in the registers.
3) The control that initiates the sequence of micro-operations.

The process of specifying the micro-operations in the computer in descriptive explanations could be long and tedious. A more convenient way is to adopt a suitable symbolic notation to describe the micro-operations, transfer among registers is called register transfer language(RTL). A register transfer language is a system for expressing in symbolic form of micro-operations sequences among the register of a digital module.

Register Transfer



Computer registers are those that hold the data, they are designated by capital letters to denote the function of the register, e.g. PC (Program Counter), IR (Instruction Register).

The individual flip-flops in an n-bit register are number in sequence from 0 through n-1, starting from 0 in the right most position and increasing the numbers towards left. Figure shows the representation of register block diagram. The name in the 16-bit register is PC, the

symbol PC (0-7) or PC(L) refers to the low-order byte and PC (8-15) or PC(H) to higher order byte.

Information transfer from one register to another is designated in symbolic form by means of replacement operations. The statement $R_2 \leftarrow R_1$ denotes a transfer of the content or register R_1 into register R_2. It designates a replacement of the content of R_2 by the content of R_1. By definition the content of the source register R_1 does not change after transfer.

We want transfer to occur only under a predefined control condition. This can be shown by means of an if-then statement

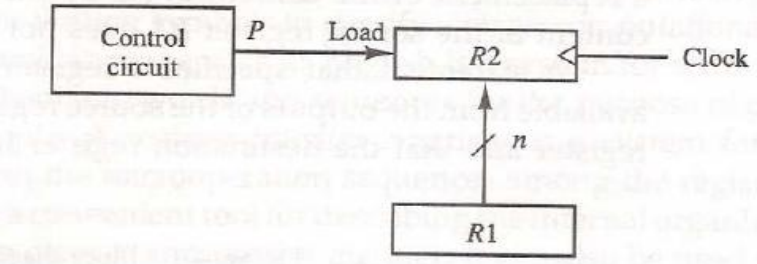
If(P=1), then($R_2 \leftarrow R_1$)

Where P is a control signal generated in control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. A control function is a Boolean variables that is equal to 1 or 0. The control function is included in the statement as follows: P:$R2 \leftarrow R1$

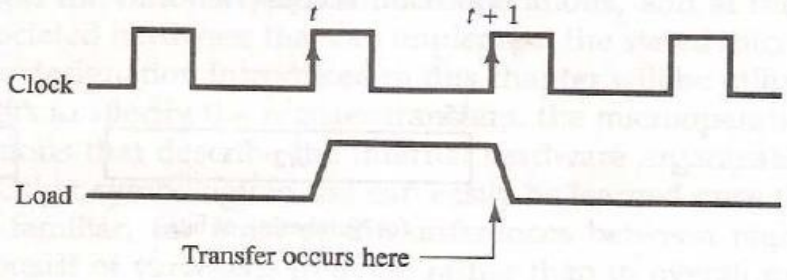
It symbolizes the requirement that the transfer operation be executed by hardware only if P=1.

Figure shows the block diagram that depicts the transfer from R1 to R2.

Figure 4-2 Transfer from $R1$ to $R2$ when $P = 1$.



(a) Block diagram



(b) Timing diagram

The n outputs of register R_1, are connected to the n inputs of register R_2, register R_2 has a load input that is activated by control variable P. It is assumed that the control variable is synchronized with same clock as the one applied to the register. P is active in the control section by the rising edge of a clock pulse at time t. The next positive transition of the clock at time(t+1) find the load input active and data inputs of R_2 are then loaded into the register in parallel. P may go back 0 at time t+1, otherwise the transfer will occur with every clock pulse transition while P remains active.

The statement T:$R_2 \leftarrow R_1$, $R_4 \leftarrow R_3$

Denotes an operation that exchanges the content of two register during one common clock pulse provided that T=1.

Bus and memory transfer

In a digital computer there may be many register, so that there will be many data paths if separate lines are used between each register, so a more efficient way for transferring information between register in multiple register configuration in a common bus system. A bus structure consists of a set of common lines, one for each bit of register through which binary information is transferred one at a time.

The transfer of information from a bus into one of many destination register can be accomplished by connection the bus lines to the inputs of all destination registers and activation the load control of the particular destination register selected. When the bus is included in the statement the register transfer may be symbolized as

BUS←C, R_1←BUS

The content of register C is placed on the bus and the content of the bus is loaded into register R1, by activating load control input. If bus is known to exist in the system, it may be convenient just to show the direct transfer

R_1←C

Memory transfer

The transfer of information from memory word to the outside environment is called read operation. The transfer of new information to be stored into the memory is called write operation. A memory word will be symbolized by letter M. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following letter M.

Example : read DR←M[R]

Consider a memory unit that receives the address from a register called the address register(AR). The data are transferred to another register called data register(DR). This causes a transfer of information into DR from the memory word M selected by the address AR.

Example : $R_3 \leftarrow R_1 + \overline{R_2} + 1$

Assume the input data are in register R_1, $\overline{R_2}$ is the symbol for the 1's complement of R_2. Adding the content of R_1 to 2's complement of R_2 is equivalent to R_1-R_2.

Arithmetic micro-operations

TABLE 4-3 Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

Logical micro-operations

Logic micro-operations specify binary operations for strings of data bits stored in registers. These operations consider each bit of register separately and treat them as binary variables. For example, the exclusive-OR micro operations with the content of two register R_1 and R_2 is symbolized by the statement P:$R_1 \leftarrow R_1 \oplus R_2$

It specifies a logic micro-operations to be executed on the individual bits of the register provided that the control variable P=1. Example consider R_1 contains 1010 & R_2 contains 1100. The exclusive –OR micro operation stated above symbolized the logical operations of logical XOR of individual bits.

1	0	1	0
1	1	0	0
0	1	1	0

The content of R_1 after the execution of the micro-operations is equal the bit-by –bit exclusive OR on pairs of bits in R_2 and previous value of R_1.

Special symbols

Special symbols will be adopted for the logic micro-operations of OR, AND and complement, to distinguish them from the corresponding symbols used to express Boolean functions. Example 'V' will be used to denote an OR , 'Λ' between R_5 & R_6 designated 'OR' micro-operations.

Shift micro-operations

Shift micro-operations are used for serial transfer of data. They are used in conjunction with arithmetic, logical and other data processing operations. The content of a register can be shifted to the left or the right. The information transferred through the serial inputs determines the types of shift. There are three types of shift

1. Logical shift: a logical shift is one that transfers 0 through the serial input. We will adopt the symbol shl and shr for logical shift-left and shift right micro-operations.
   $R_1 \leftarrow$ shl R_1, $R_2 \leftarrow$ shr R_2

TABLE 4-6 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

1	0	1	1
0	1	0	1

Above are two micro-operations that specify 1 bit to the left of the content of register R_1 and 1bit shift to the right of the content of register R_2. The register symbol must be of the same on the both side of the arrow.

2. Circular shift: the circular shift (also known as rotate operations) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial inputs. Use the symbol as cil or cir
3. Arithmetic shift: an arithmetic shift is a micro-operation that shifts a signed binary number to the left or right. An arithmetic shift left multiples a signed binary number by 2 and arithmetic shift right divides the number by 2.

The arithmetic shift must leave the sign bit unchanged because the sign of the number remains the same when it is multiples or divided by 2. The left most bit in register holds the sign bit and the remaining bits hold the numbers. Figure shows a typical register of n bits. Bit R_{n-1} in the left most position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shift the number(including the sign bit) to the right. Thus R_{n-1} remains the same R_{n-2} receives the bit from R_{n-1} and so on for other bits in the register. The bit R_0 is lost.

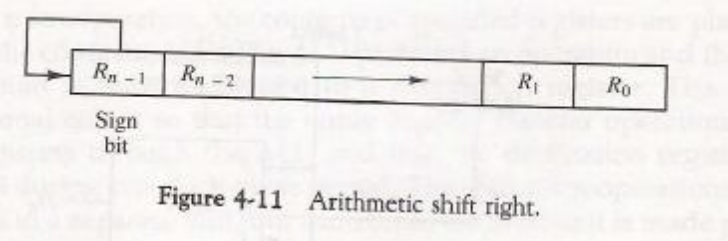


Figure 4-11 Arithmetic shift right.

imp

Introduction to HDL and VHDL

Hardware Description Language(HDL) is a specialized computer language used to program the structure, design and operation of electronic circuits and most commonly digital logic circuits.

A hardware description language enables a precise, formal description of an electronic circuits that allows for the automated analysis, simulation and simulated testing of an electronic circuit. It also allows for the compilation of an HDL program into a lower level specification of physical electronic components.

What are the advantages of using HDLs?

- Can express large, complex design.
- Flexible modelling capabilities.

- Description can include the very abstract to the very structural level
- Productivity
    - Logic synthesis: easily develop, use the available logics.
    - Design changes are fast and easily done.
    - Optimization of design is easier.
    - Exploration of alternative design can be done quickly.
- Reusability:
    - Packages, libraries, design all can be reused.
    - Vendors and technology independence
    - E.g. CMOS, ECL, gates same code.
- Documentation
    - Textual documentation is part of the code, not a separate document.

Example:

```
// This module creates an address/accumulator

Module adder (

Input clk, // input clock

Input reset_n, // async active low

Input first_select, // on first data item use this

Input rd_fifo, //enable for ff

Input [7:0] data, //data in

Output reg[11:0] acc_out //data out of accumulator

);

Always @ (posedge clk, nedge reset_n)

If(!reset_n) acc_out<=12'h000;

Else

If(rd_fifo==1 'b')

If(first_selected==1'b') acc_out<=data

Else                    acc_out<=acc_out+data;

End module
```

What can't an HDL do?

- We cannot make architectural tradeoffs, but it can help.
- Does not relieve in understanding digital design.
- If you can't draw the structure you are looking for stop coding.

VDHL(Very High Speed Integrated Circuit (VHSIC) HDL

- More difficult to learn
- Widely used for FPGAs

Verilog

- Simpler to learn.
- Look like C.

VHDLs history

- VHDL is the product of US government request for a new means of describing digital hardware.
- VHSIC program was an initiative of the defense department to push the state of the art in VLSI technology and HDL was proposed as a versatile hardware description language.

Reason for Using VHDL

- International IEEE standard specification language.
- Enables hardware modelling from the gate to system level.
- Provides a mechanism for digital design and design documentation.
- Formal model to communicate.
- Modelling: documentation.
- Testing and validation using simulation.
- Performance prediction.
- Automatic synthesis.
- Concurrency, more than one thing going at a time.
- VHDL allows the designers to work at various levels of abstraction.
- Behavioural, RTL, Boolean equation, gates.
- Allows for various design methodologies, top→down, bottom →up.
- Flexible in describing hardware.
- Provides technology independence.
- Describes a wide variety of digital hardware.

Chapter 3 Central Processing Unit

3.1 CPU Organization and Structure

The part of the computer that performs the bulk of data processing operations is called the central processing unit (CPU). The CPU is made of three major parts as in figure:

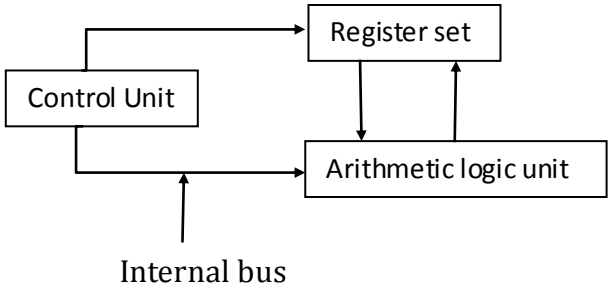


Figure 3.1: Major Components of CPU

The register set stores immediate data used during the execution of the instructions.

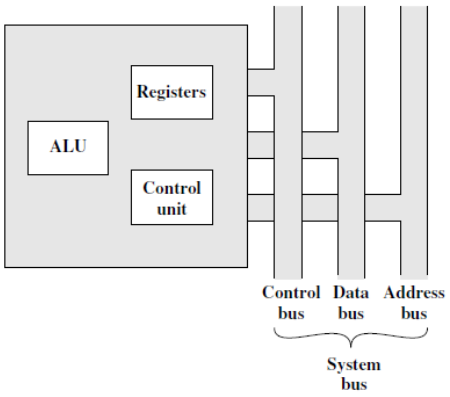
Control unit: It supervises the transfer of information among the registers and instructs the ALU (Arithmetic and Logic Unit) as to which operations to perform.

ALU (Arithmetic and logic Unit): Performs all the arithmetic and logical operations.

A computer consists of set of components or modules of three basic types (processor, memory, I/O) that communicate with each other. The collection of paths connecting the various modules is called the interconnection structure. The design of this structure will depend on the exchange of data and information that must be made among the modules.

Bus interconnection

A bus is a communication pathway connecting two or more devices, which is shared transmission medium. Multiple devices connect to the bus and a signal transmitted by one device is available for reception by all other devices attached to the bus. If two devices transmit during the same time period their signals will overlap and become garbled, thus only one device can transmit successfully at a time. Typically a bus consists of multiple communication pathways or lines, each lines is capable of transmitting signals representing binary 1 or 0.



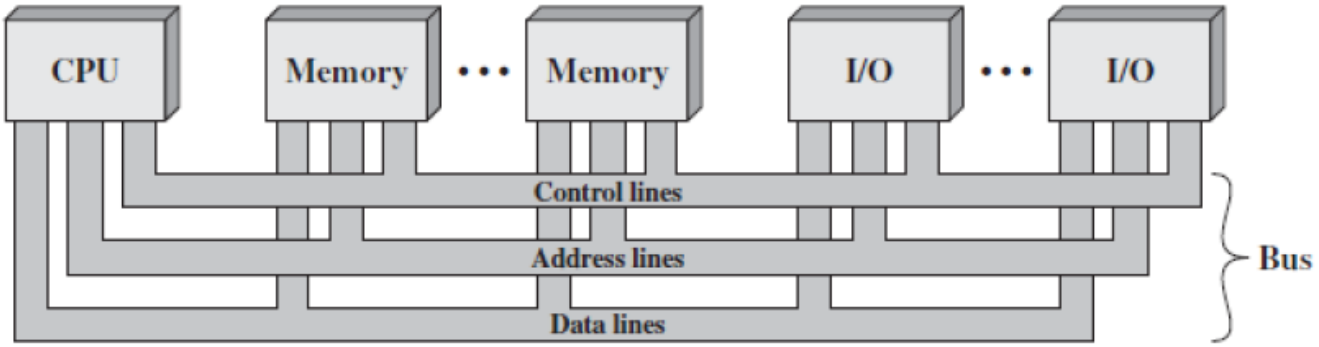


Figure 3.2: Bus interconnection Scheme

- Data line: provide a path for moving data among system modules.
- Address line: used to designate the source or destination of the data on data bus.
- Control lines: used to control the access to and use of data and address lines.

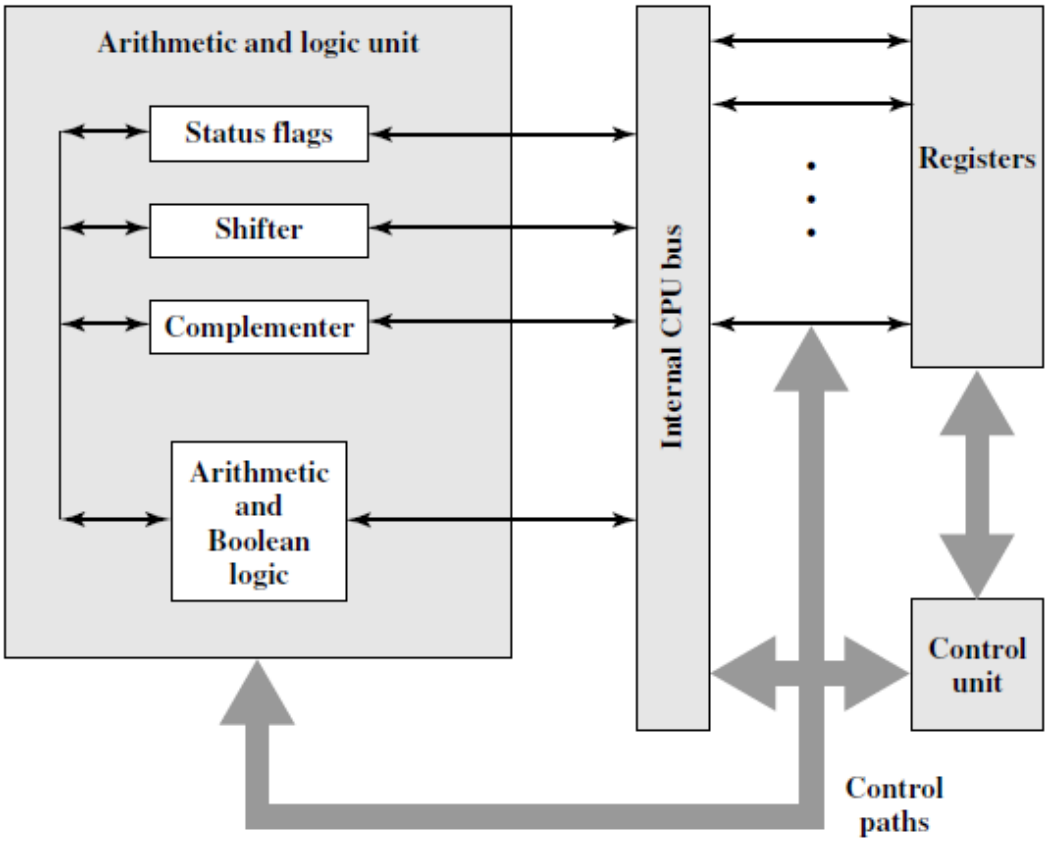


Figure 3.3: Internal structure of the CPU

3.2 Register organization

Register are faster, smaller and more expensive memory that function as a level of memory above main memory and cache. The two categories to register are:

a) **User-visible registers:** a user visible register is one that may be referenced by means of the machine language that the processor executes.
b) **Control and status registers:** Used by the control unit to control the operation of the processor and by privileged, operating system programs to control the execution of programs.

a) **User-visible registers:** a user visible register is one that may be referenced by means of the machine language that the processor executes. We can further categorize as follows:

1. General purpose
2. Data
3. Address
4. Condition codes

1. *General-purpose registers* can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. i.e, any general-purpose register can contain the operand for any op-code. This provides true general-purpose register use. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point and stack operations. In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers.
2. *Data register:* Data registers may be used only to hold data and cannot be employed in the calculation of an operand address.
3. *Address registers* may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:
   i. *Segment pointers:* In a machine with segmented addressing, a segment register holds the address of the base of the segment. There may be multiple registers: for example, one for the operating system and one for the current process.
   ii. *Index registers:* These are used for indexed addressing and may be auto indexed.
   iii. *Stack pointer:* If there is user-visible stack addressing, then typically there is a dedicated register that points to the top of the stack. This allows implicit addressing; i.e, push, pop, and other stack instructions need not contain an explicit stack operand.
   iv. *Conditional codes (also referred to as flags):* Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation. Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but the programmer cannot alter them.

b) Control and status register :

There are a variety of processor registers that are employed to control the operation of the processor. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode. Of course, different machines will have different register organizations and use different terminology. Four register are essential to instruction execution:

1. **Program counter (PC):** Contains the address of an instruction to be fetched.
2. **Instruction register (IR):** Contains the instruction most recently fetched.
3. **Memory Address Register (MAR):** Contains the address of a location in memory.
4. **Memory Buffer Register (MBR):** Contains a word of data to be written to memory or the word most recently read.

Many processor designs include a register or set of registers, often known as the ***program status word* (PSW),** that contain status information. The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

• **Sign:** Contains the sign bit of the result of the last arithmetic operation.

• **Zero:** Set when the result is 0.

• **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.

• **Equal:** Set if a logical compare result is equality.

• **Overflow:** Used to indicate arithmetic overflow.

• **Interrupt Enable/Disable:** Used to enable or disable interrupts.

• **Supervisor:** Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

Note: see example of microprocessor register organization MC68000 & intel 8086 from William Stallings book.

3.3 Instruction cycle

The processing required for a single instruction is called an *instruction cycle*. The instruction cycle can be simply depicted in Figure 3.3. The two steps are referred to as the *fetch cycle* and the *execute cycle*. Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

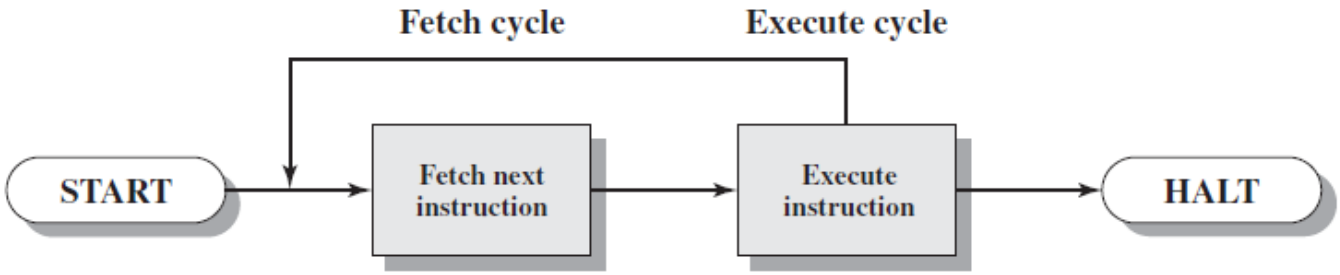


Figure 3.4 Basic instruction cycle

An instruction cycle includes the following stages:

• **Fetch:** Read the next instruction from memory into the processor.

• **Execute:** Interpret the op-code and perform the indicated operation.

• **Interrupt:** If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt. We are now in a position to elaborate somewhat on the instruction cycle. First, we must introduce one additional stage, known as the indirect cycle.

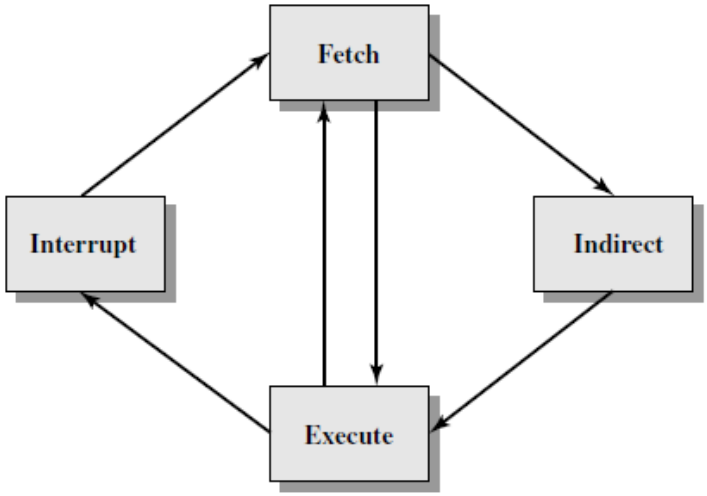


Figure 3.5 instruction cycle

The execution of an instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used, then additional memory accesses are required. We can think of the fetching of indirect addresses as one more instruction stages. The result is shown in Figure 3.5.The main line of activity consists of alternating instruction fetch and instruction execution activities. After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing.

Instruction cycle state diagram

The figure 3.6 shows the state diagram of instruction cycle. For any given instruction cycle some states may be null and others may be visited more than once. The different states may be described as:

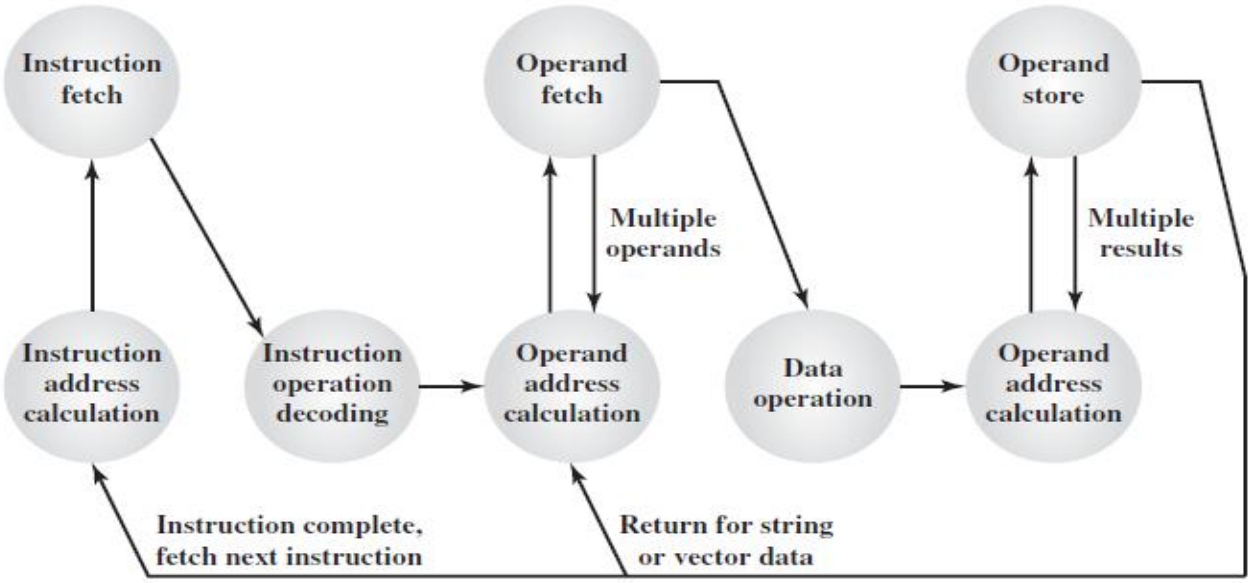


Figure 3.6: Instruction cycle state diagram

Instruction address calculation (iac): Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.

• **Instruction fetch (if):** Read instruction from its memory location into the processor.
• **Instruction operation decoding (iod):** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
• **Operand address calculation (oac):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
• **Operand fetch (of):** Fetch the operand from memory or read it in from I/O.
• **Data operation (do):** Perform the operation indicated in the instruction.
• **Operand store (os):** Write the result into memory or out to I/O.

States in the upper part of Figure 3.5 involve an exchange between the processor and either memory or an I/O module. States in the lower part of the diagram involve only internal processor operations. The oac state appears twice, because an instruction may involve a read, a write, or both. Finally, on some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string (one-dimensional array) of characters. As Figure 3.6 indicates, this would involve repetitive operand fetch and/or store operations.

Interrupts

Interrupts is an asynchronous service request from hardware or software to CPU, interrupts makes CPU execution of its normal operation to pause and then to service external device. The processor and OS are responsible for recognizing an interrupt, suspending the user program servicing the interrupt and then resuming the user program. The instruction cycle with interrupts is as shown in figure.

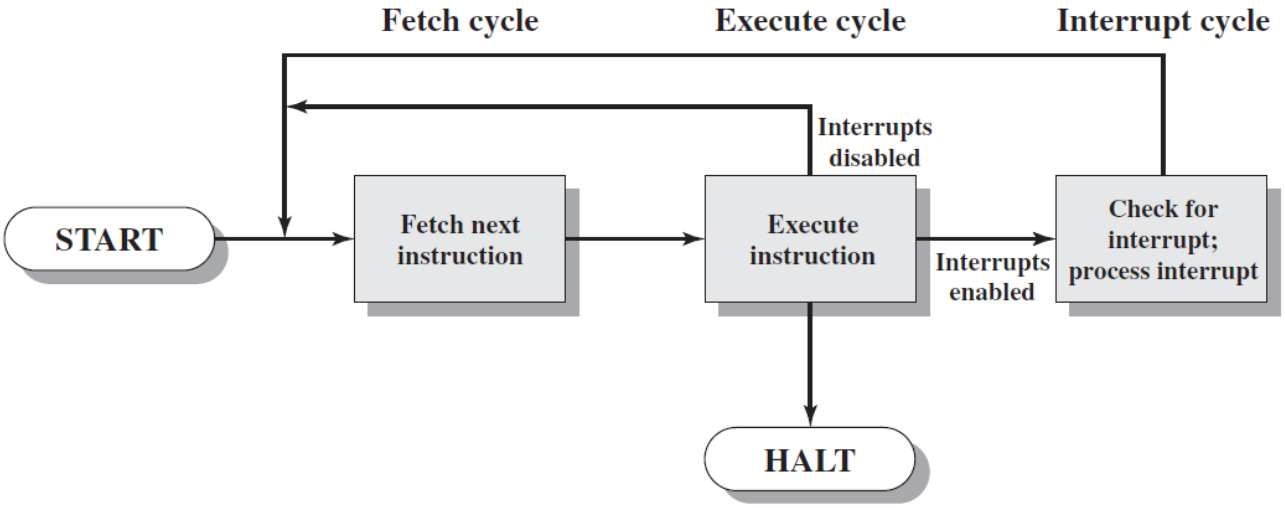


Figure 3.7 Instruction cycle with interrupt.

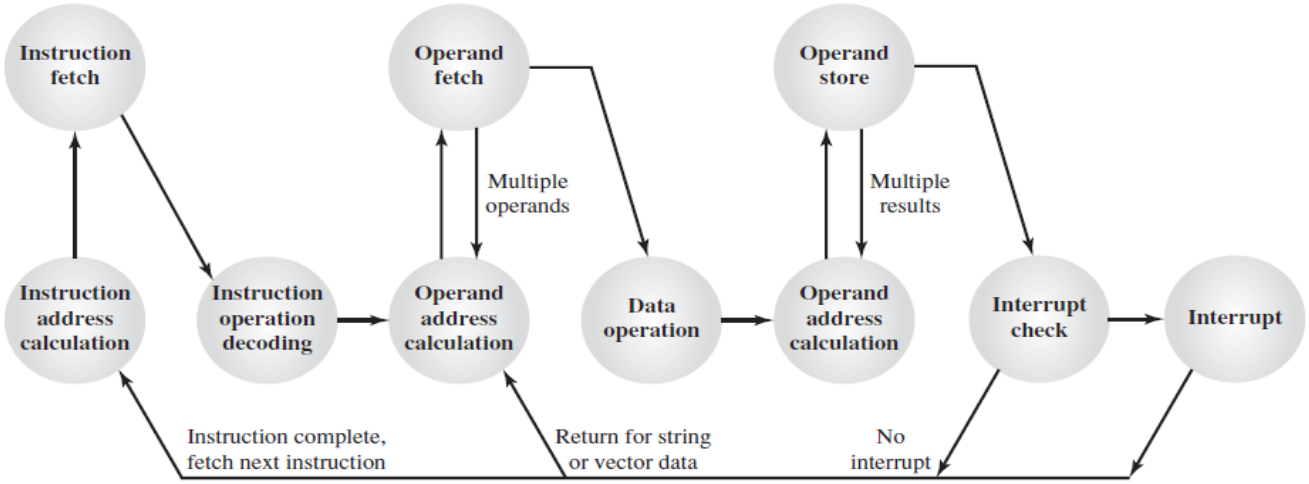


Figure 3.8 Instruction cycle state diagram with interrupt

In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

- It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
- It sets the program counter to the starting address of an *interrupt handler* routine.

The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this program determines the nature of the interrupt and performs whatever actions are needed.

3.4 **The arithmetic and logic unit**

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. ALU is a multi operation combinational logic digital function i.e. can perform set of arithmetic and logic of operations. Figure shows the block diagram of 4 bit ALU.

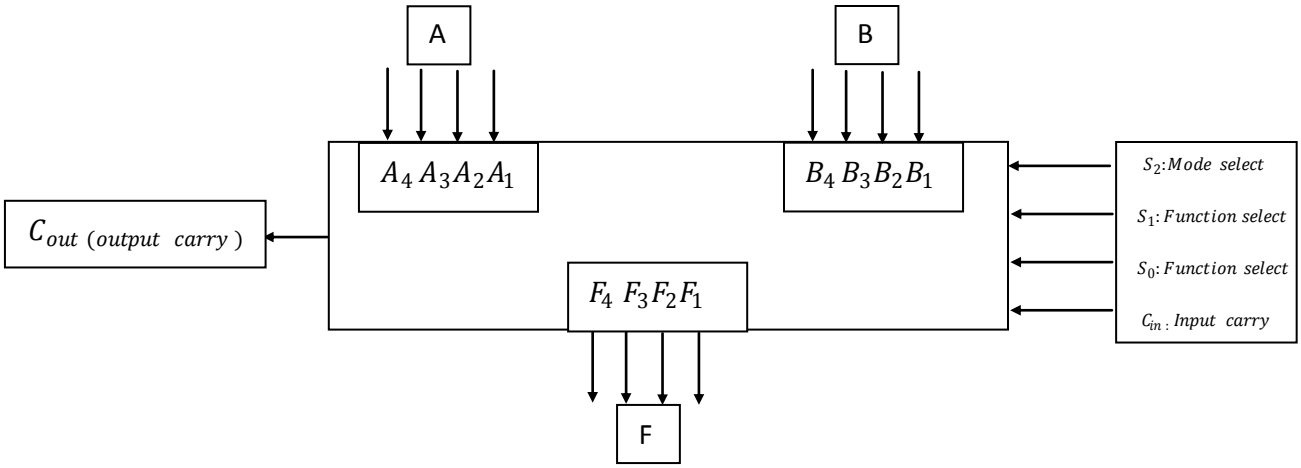
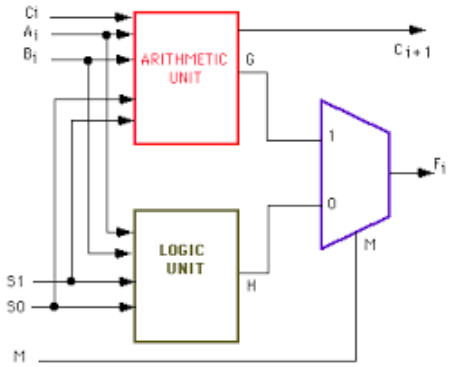


Figure: Block diagram of 4 bit ALU

S1	S0	Arithmetic (s2=0)	Logical(s2=1)
0	0	Addition	OR
0	1	Subtraction	AND
1	0	Multiplication	NOT
1	1	Division	Ex-OR

The selection lines are decodes within the ALU so that the K selection variables can specify up to 2^k distinct operations. Figure shows 4 bit inputs A and B to generate 4 bit result F. The mode select distinguish between arithmetic and logic operations. The two function select inputs specify the particular arithmetic or logical operations.



3.5 Design principles of Modern System

- Efficient memory utilization.
- Use of caches, different levels.
- Improvement in input and output mechanism.
- Pipelining.
- Parallel processing.
- Multi-core systems.

Chapter 4 Computer Arithmetic

4.1 Integer Representation

In the binary number system, arbitrary numbers can be represented with just two digits zero and one, the minus sign, and the period, or **radix point**. E.g -1101.01012 = -13.312510. For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An eight bit word can be represented as 0=0000000, 1=0000001, 7=00000111.

In general, if an n-bit sequence of binary digits is interpreted as an unsigned integer A, its value is

$A=\sum_{i=0}^{n-1} 2^i a_i$

4.1.1 Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative. The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost bits hold the magnitude of the integer.

+ 7 = 0111; -7 = 1111 (sign magnitude)

Drawbacks

- Addition and subtraction requires a consideration of both sign of numbers and their relative magnitudes to carry out the required operation.
- Two representations for 0.

4.1.1.1 Two's complement representation

Like sign magnitude two's complement representation uses the most significant bit as sign bit making it easy to test whether the integer is negative or positive. It differs from the use of sign magnitude representation in the way the other bits are interpreted.

Method:

For negation take the Boolean complement of each bits of corresponding positive number, and then add one to the resulting bit pattern.

Consider n nit integer A in two's complement representation. If A is positive then the sign bit $a_{n-1\ zero}$. The remaining bit represents the magnitude of the number.

$A=\sum_{i=0}^{n-1} 2^i a_i$. The range of positive integer that may be represented is $0- 2^{n-1}-1$.

Now for negative numbers integer A, the sign bit a_{n-1} is 1. The range of negative integer that can be represented is from -1 to -2^{n-1}

	Sign representation	Two's complement
+7	0111	0111
-7	1111	1001

4.1.1.2 Float point representation

MR^E

The float point representation of a number has two parts. The first part represents a signed, fixed point number called mantissa. The second part gives the position of the decimal point called exponent.

	Fraction	Exponent
+6132.789	+0.6132789	+04

0.6132789×10^4

In the same it can be used for binary numbers.

	Fraction	Exponent
+1001.11	0.1001110	000100

$(0.1001110) \times 2^4$

Only the mantissa m and the exponent e are physically represented in the register (including their signs).

Negation

In sign magnitude negation is done by inverting the sign bit.

In two's complement method:

- Take Boolean complement if each bit. Then add 1

4.2 Integer Arithmetic

4.2.1 Addition and subtraction with sign twos complement data

Addition

0011
+ 0100
0111

Subtraction

M-N

1. Add M to r's complement of N.
2. Inspect the result obtained in step 1 for an end carry.
   a. If an end carry occurs discard it.
   b. If an end carry does not occur take r's complement of the number obtained in step 1 and place a negative sign in front.

M-N, M=1010100, N=1000100

1's complement of N= 0111011

2's complement of N= (0111011+1) =0111100

1010100
+ 0111100
10010000
Here end carry is generated , so discard it, hence final answer is 0010000

M=1000100, N=1010100

1's complement of N= 0101011

2's complement of N= (0101011+1) =0101100

1000100
+ 0101100
111000
Here no end carry , take 2's complement of 111000 and place a negative sign =-10000

On any addition the result may be larger than the word size being used then it is called overflow. When overflow occurs the ALU must signal so that the process is stopped.

Overflow rule

If two numbers are added and they are both positive or negative, then overflow occurs if and only if the result has opposite sign.

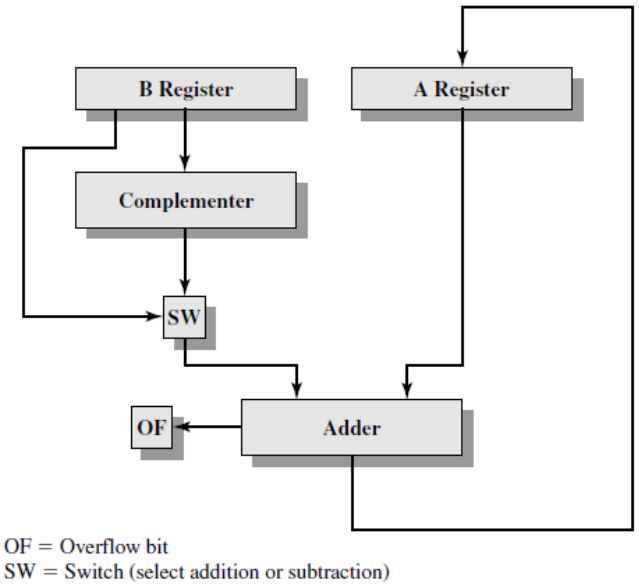


Figure 4.1: block diagram of hardware for addition and subtraction

- Figure shows the data path and hardware elements needed to accomplish addition and subtraction.
- Binary adder is the central element for addition to produce sum and overflow indication.
- For addition of two number in register A and B passed to adder.
- For subtraction the subtrahend (register B) is 2's complemented and added so that the nos. are subtracted.
- Overflow indication stored in 1 bit overflow flag, 0= no overflow, 1= overflow.

4.4 Unsigned Binary Multiplication Algorithm

4.4.1 Multiplication

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software.

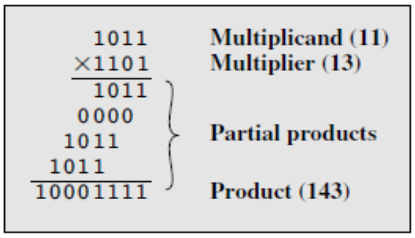


Figure 4.2 : multiplication of unsigned binary integers

This method is not efficient for computer operations so for computer operation, we follow the following algorithms for unsigned number.

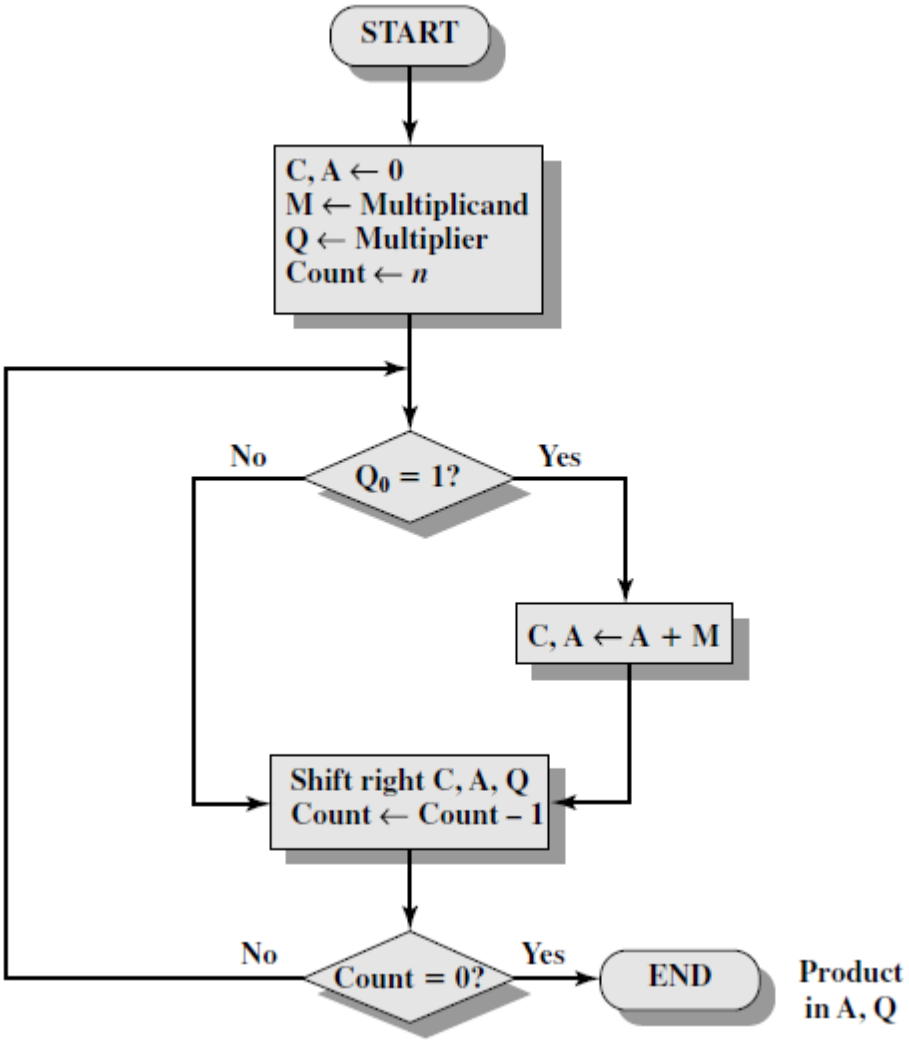


Figure 4.3: flow chart for unsigned binary multiplication

Example: 1101x1011= 1000111, 13x11=143

Count	C	A	Q	M	operations
4	0	0000	1101	1011	Initial values
	0	1011	1101	1011	Add(A+M)
3	0	0101	1110	1011	Shift(right)
2	0	0010	1111	1011	Shift(right)
	0	1101	1111	1011	Add(A+M)
1	0	0110	1111	1011	Shift(right)
	1	0001	1111	1011	Add
0	0	1000	1111	1011	Shift(right)

Final value AQ= $(10001111)_2$

Perform for 7*-3=-21

4.5 Booths algorithm

Multiplication of signed number or negative number is not possible by above method so for that we need booths algorithm

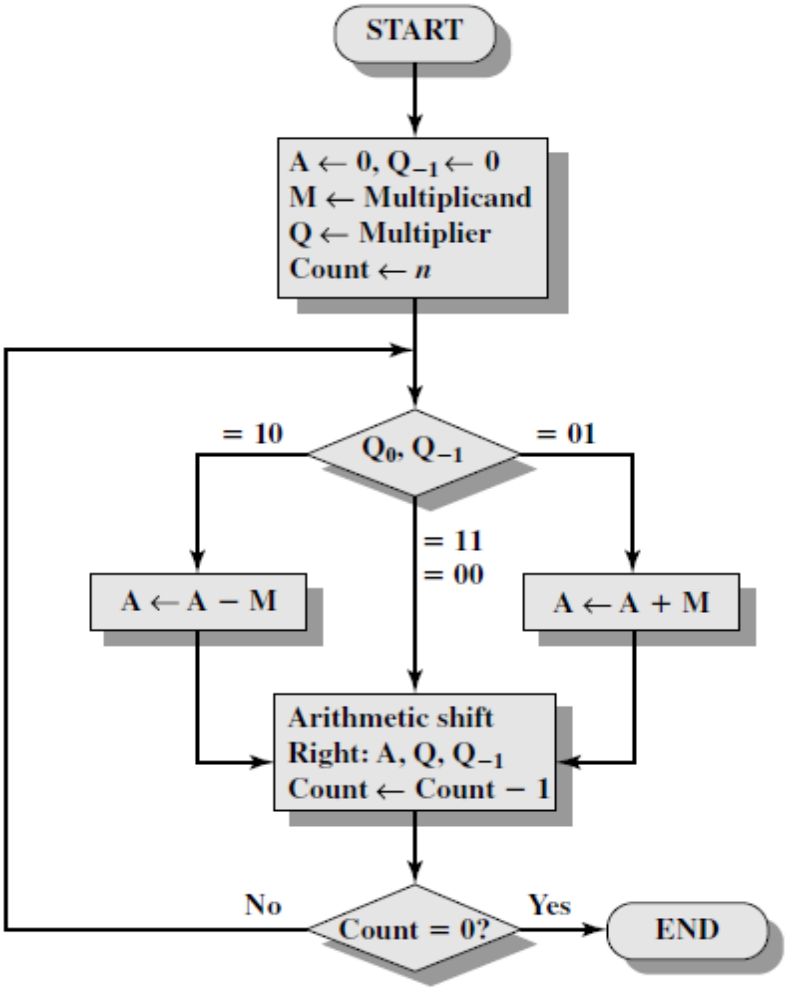


Figure 4.4: flowchart of booth's algorithm for two's complement multiplication.

Perform: 7x-3=-21= $(11101011)_2$

Count	A	Q	Q_{-1}	M	operations
4	0000	0111	0	1101	Initial values
	0011	0111	0	1101	A←A-M
3	0001	1011	1	1101	Shift
2	0000	1101	1	1101	Shift
1	0000	0110	1	1101	Shift
	1101	0110	1	1101	A←A+M
0	1110	1011	0	1101	Shift

AQ= $(11101011)_2$

4.6 Unsigned Binary Division Algorithm

Algorithm

1. Load M with divisor, AQ with dividend (using sign bit).
2. Shift AQ left 1 position (logical shift).
3. If M and A have same sign, A←A-M, else A←A+M
4. Q_0←1, if sign bit of A has not changed.

Else Q_0=0 and restore A.

5. Repeat 2 to 4 till counter is Zero.
6. Remainder in A and quotient in Q.

Example: 5/2 = 2, 1

Count	A	Q	M	-M	operation
4	0000	0101	0010	1110	Initial values
	0000	1010	0010	1110	Shift AQ
Changed	1110	1010	0010	1110	A←A-M
3	0000	1010	0010	1110	
	0001	0100	0010	1110	Shift AQ
changed	1111	0100			A←A-M
2	0001	0100	0010	1110	
	0010	1000	0010	1110	Shift AQ
Not	0000	1000			A←A-M
1	0000	1001	0010	1110	
	0001	0010			Shift AQ
Changed	1111	0010			A←A-M
0	0001	0010	0010	1110	

Remainder=A=0001, quotient =Q=0010

Do for 7/3

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input carry. Suppose we apply two BCD digits to a four-bit binary adder. The adder will form the sum in *binary* and produce a result that ranges from 0 through 19.

Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

Figure 4.5 Derivation table of BCD adder

These binary numbers are listed in Table 4.5 and are labeled by symbols K, $Z8$, $Z4$, $Z2$, and $Z1$. K is the carry, and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The output sum of two decimal digits must be represented in BCD and should appear in the form listed in the columns under "BCD Sum." The problem is to find a rule by which the binary sum is converted to the correct BCD digit representation of the number in the BCD sum. In examining the contents of the table, it becomes apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain an invalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required. The logic circuit that detects the necessary correction can be derived from the entries in the table. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 through 1111 that need a correction have a 1 in position $Z8$. To distinguish them from binary 1000 and 1001, which also have a 1 in position $Z8$, we specify further that either $Z4$ or $Z2$ must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage. A BCD adder that adds two BCD digits and produces a sum digit in BCD is shown in Fig. below.

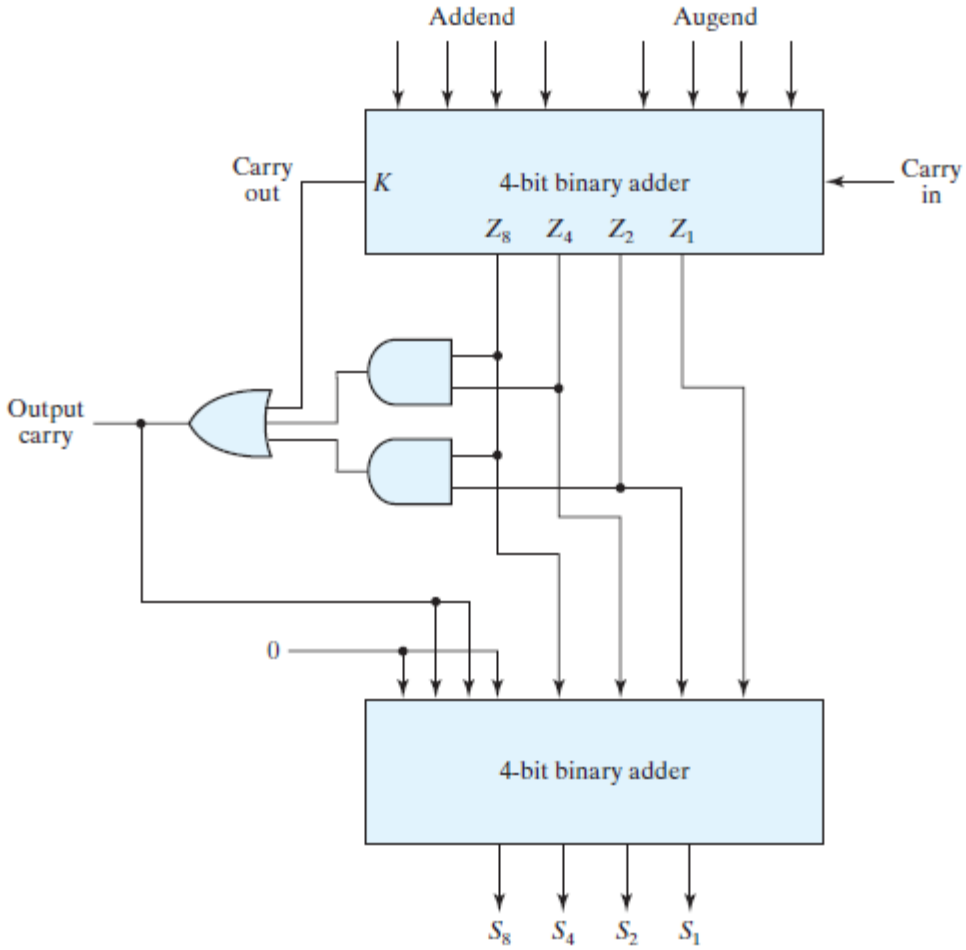


Figure: 4.5 Block diagram of BCD adder

The two decimal digits, together with the input carry, are first added in the top four-bit adder to produce the binary sum. When the output carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom four-bit adder. The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. A decimal parallel adder that adds *n* decimal digits needs *n* BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher order stage.

4.8 Arithmetic pipeline

Pipeline units are usually found in very high speed computers. They are used to implement for floating operations like multiplication. The floating point operations are easily decomposed into sub operations and then calculated. Example: a pipeline unit for floating addition and subtraction. Consider the two normalized floating point numbers

$X = 0.9504\ x\ 10^3$ & Y= $0.8200\ x\ 10^2$

Now

Segment 1: Two exponents are subtracted 3-2=1. The larger exponent is chosen as the exponent of the result.

Segment 2: Shift the mantissa of y to the right to obtain $X = 0.9504\ x\ 10^3$ & Y= $0.08200\ x\ 10^3$.. This aligns the mantissa under the same exponent.

Segment 3: the addition of two mantissa in segment 3 produces the sum $Z = 1.0324\ x\ 10^3$

Segment 4: the sum is adjusted by the normalizing the result so that it has a fraction with non zero first digit.

$Z = 0.10324\ x\ 4$

The comparator, shifter, adder-subtractor, incrementer and decrementer in the floating point pipeline are implemented with the combination circuits, suppose the time delay of the four segments are t1=60ns, t2=70ns, t3=100ns & t4=80ns, and the interface register have a delay of tr=10 ns. The clock cycle is choosen to be tp=t3+tr=110ns. An equivalent non pipeline floating adder-subtractor will have delay time of tn=t1+t2+t3+t4+t5+tr=320ns.

Speed up=320/110=2.9 over a non pipelined adder.

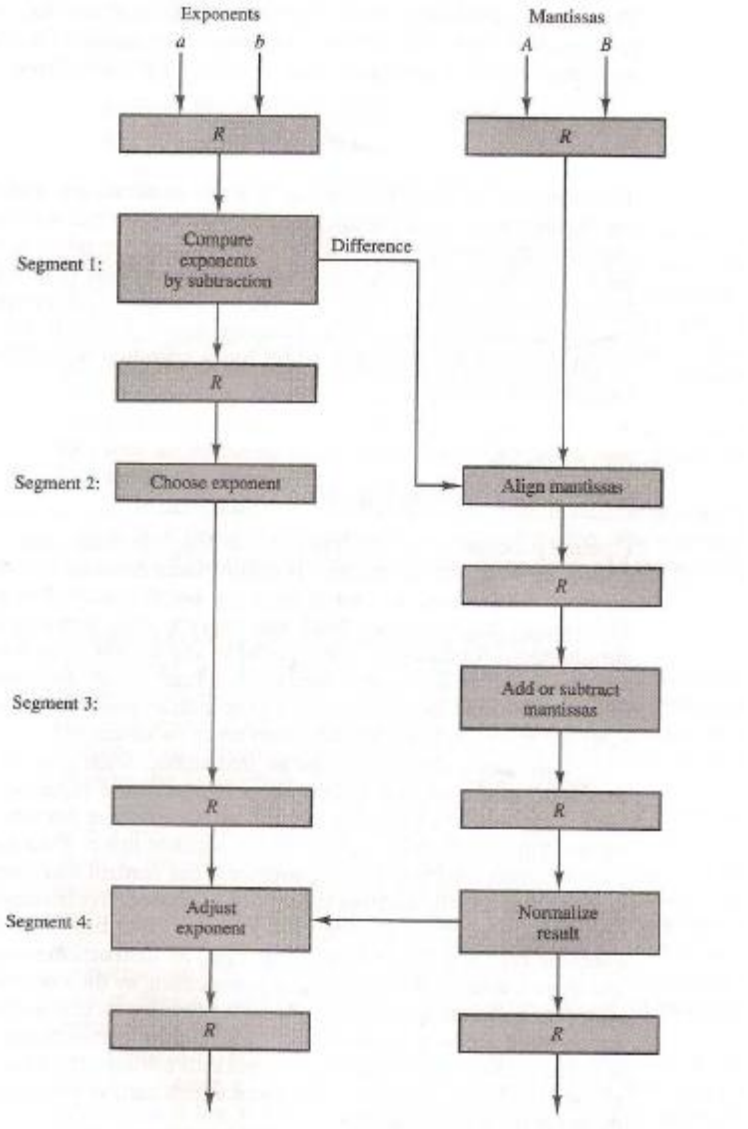


Figure 4.6 : Arithmetic pipeline for floating point addition and subtraction.

Chapter 5: Control Unit

Micro operations

The execution of a program consists of the sequential execution of instruction. Each instruction is executed during in instruction cycle made of up of shorter sub cycles (e.g. fetch, indirect, execute, interrupt). The performance of each sub cycles involves one or more short operations called micro-operations.

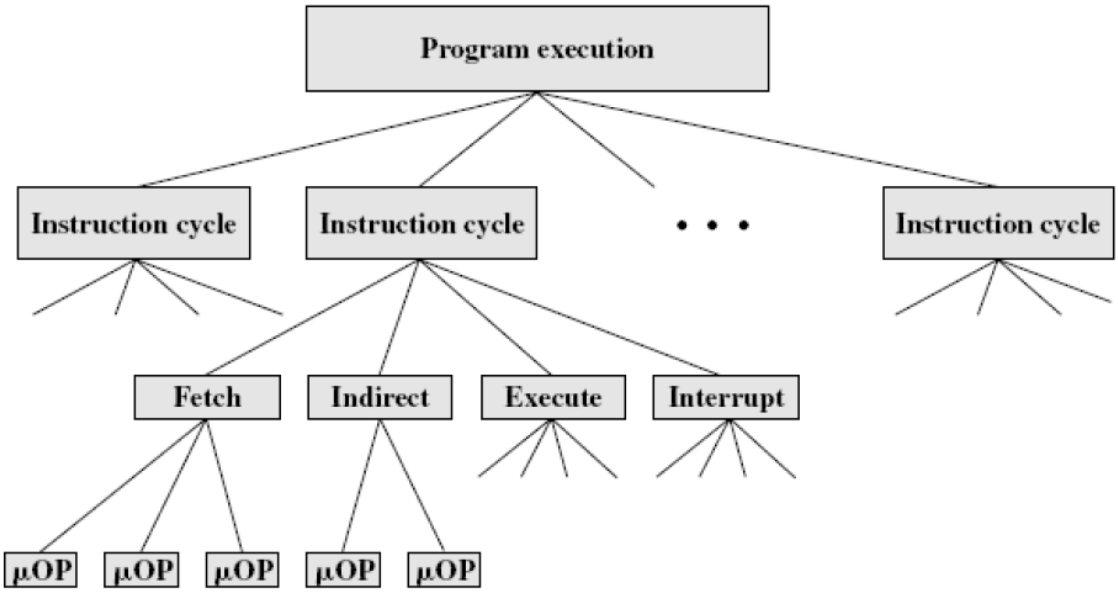


Figure 5.1 Constituents elements of a program execution

imp

1. The Fetch Cycle

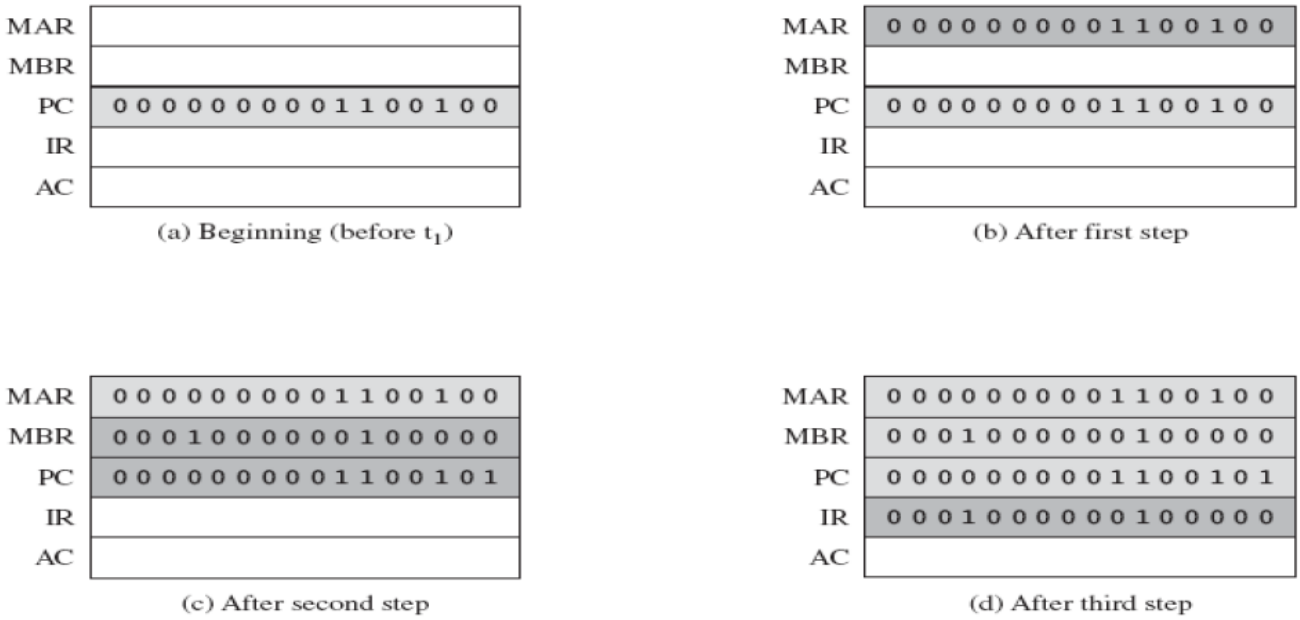
Fetch cycle occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. Four registers are involved:

• **Memory address register (MAR):** It is connected to the address lines of the system bus. It specifies the address in memory for a read or writes operation.

• **Memory buffer register (MBR):** It is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.

• **Program counter (PC):** Holds the address of the next instruction to be fetched.

• **Instruction register (IR):** Holds the last instruction fetched.



Sequence of events

1. At the beginning of the fetch cycle the address of the next instruction to be executed is in PC.
2. The first step is to move the address to the MAR, because it is connected to address line.
3. Second step is to bring in the instruction. The desired address is placed on address bus, control unit issues READ command and data placed on MBR through data bus, also incremented PC by 1. Here the MBR load and PC incremented is done simultaneously without interference.
4. Third step is to move the content of MBR to the instruction register (IR).

Thus, the simple fetch cycle actually consists of three steps and four micro-operations. Each micro-operation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving time. Symbolically, we can write this sequence of events as follows:

$$t_1: \quad MAR \leftarrow (PC)$$
$$t_2: \quad MBR \leftarrow Memory$$
$$\quad\quad PC \leftarrow (PC) + I$$
$$t_3: \quad IR \leftarrow (MBR)$$

Where I is the instruction length. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation represents successive time units. In words, we have (t1, t2, t3).

Note that the second and third micro-operations both take place during the second time unit. The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

$$t_1: \text{MAR} \leftarrow (\text{PC})$$
$$t_2: \text{MBR} \leftarrow \text{Memory}$$
$$t_3: \text{PC} \leftarrow (\text{PC}) + I$$
$$\text{IR} \leftarrow (\text{MBR})$$

The groupings of micro-operations must follow two simple rules:

i. The proper sequence of events must be followed. Thus MAR ←(PC) must precede MBR←memory, because the memory read operation makes use of the address in the MAR.
ii. Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable.

2. The Indirect Cycle

Once an instruction is fetched, the next step is to fetch source operands. If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle. It includes the following micro-operations:

$$t_1: \text{MAR} \leftarrow (\text{IR(Address)})$$
$$t_2: \text{MBR} \leftarrow \text{Memory}$$
$$t_3: \text{IR(Address)} \leftarrow (\text{MBR(Address)})$$

The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand. Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address. The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle.

3. The Interrupt Cycle (how interrupt is handled)

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs. The nature of this cycle varies greatly from one machine to another. We present a very simple sequence of events:

$$t_1: \text{MBR} \leftarrow (\text{PC})$$
$$t_2: \text{MAR} \leftarrow \text{Save\_Address}$$
$$\text{PC} \leftarrow \text{Routine\_Address}$$
$$t_3: \text{Memory} \leftarrow (\text{MBR})$$

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing

routine. These two actions may each be a single micro-operation. However, because most processors provide multiple types and/or levels of interrupts, it may take one or more additional micro-operations to obtain the Save_Address and the Routine_Address before they can be transferred to the MAR and PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

4. The Execute Cycle

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each case, the same micro-operations are repeated each time around. This is not true of the execute cycle, because of the variety opcodes, there are a number of different sequences of micro-operations that can occur. Let us consider several hypothetical examples. First, consider an add instruction:

ADD R1, X, which adds the contents of the location X to register R1. The following sequence of micro-operations might occur:

t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: R1 ← (R1) + (MBR)

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory location is read. Finally, the contents of R1 and MBR are added by the ALU. Again, this is a simplified example. Additional micro-operations may be required to extract the register reference from the IR and perhaps to stage the ALU inputs or outputs in some intermediate registers.

5. The Instruction Cycle:

Instruction cycle can be decomposed into a sequence of elementary micro operation. The complete picture with sequence of micro operations together is shown as in figure.

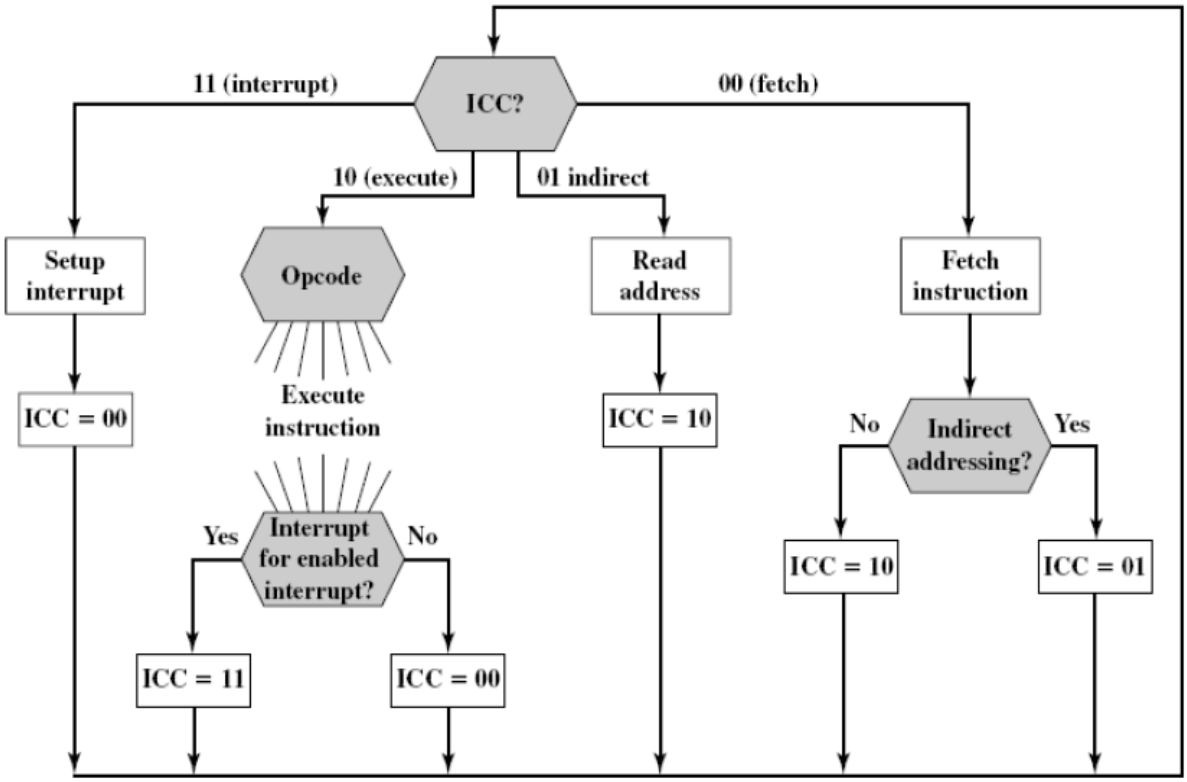


Figure 5.3: flowchart for instruction cycle.

We assume a new 2 bit register called instruction cycle code (ICC). The ICC designates the state of the processor in terms of which portion of the cycle it is:

00: Fetch
01: Indirect
10: Execute
11: Interrupt

At the end of each of the four cycles, the ICC is set appropriately. The indirect cycle is always followed by the execute cycle. The interrupt cycle is always followed by the fetch cycle. For both the fetch and execute cycles, the next cycle depends on the state of the system.

CONTROL OF THE PROCESSOR

The behavior or functioning of the processor into elementary operations, called micro-operations. By reducing the operation of the processor to its most fundamental level, we are able to define exactly what it is that the control unit must cause to happen. Thus, we can define the functional requirements for the control unit: *those functions that the control unit must perform.* A definition of these functional requirements is the basis for the design and implementation of the control unit. With the information at hand, the following three-step process leads to a characterization of the control unit:

1. Define the basic elements of the processor.
2. Describe the micro-operations that the processor performs.

3. Determine the functions that the control unit must perform to cause the micro-operations to be performed

The basic functional elements of the processor are the following:

- ALU
- Registers
- Internal data paths
- External data paths
- Control unit

The execution of a program consists of operations involving these processor elements. As we have seen, these operations consist of a sequence of micro-operations. All micro-operations fall into one of the following categories:

- Transfer data from one register to another.
- Transfer data from a register to an external interface (e.g., system bus).
- Transfer data from an external interface to a register.
- Perform an arithmetic or logic operation, using registers for input and output.

The control unit performs two basic tasks:

- Sequencing: The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.

- Execution: The control unit causes each micro-operation to be performed.

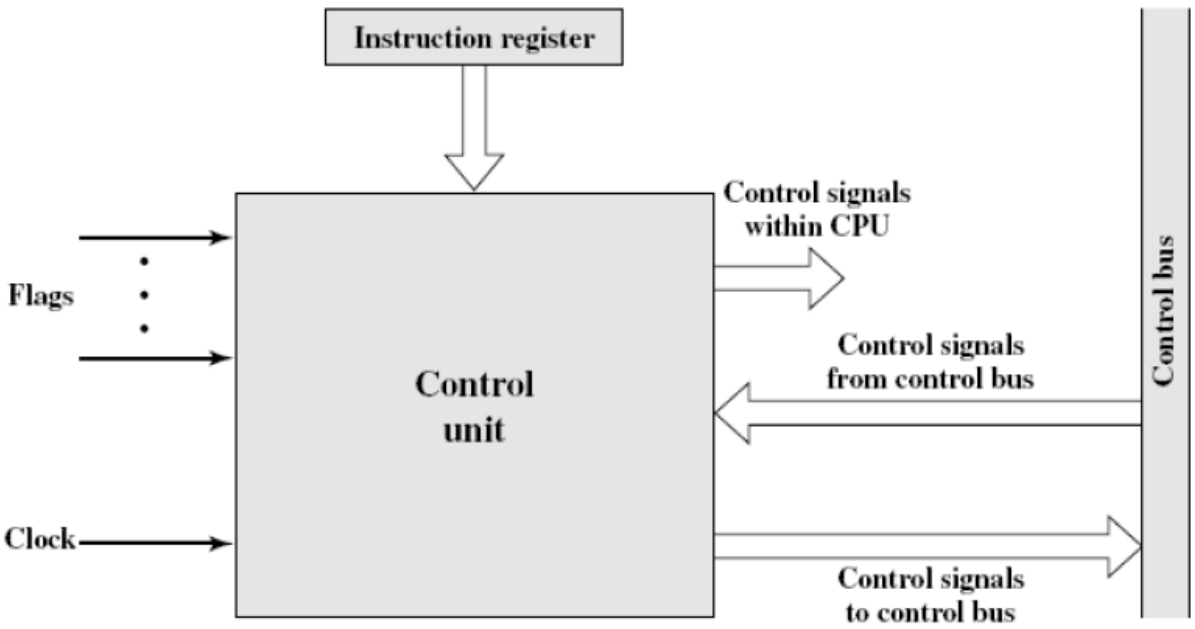


Figure 5.4 Block Diagram of Control Unit

Figure 5.4 is a general model of the control unit, showing all of its inputs and outputs. The inputs are:

Clock: This is how the control unit "keeps time." The control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is sometimes referred to as the processor cycle time, or the clock cycle time.

Instruction register: The opcode and addressing mode of the current instruction are used to determine which micro-operations to perform during the execute cycle.

Flags: These are needed by the control unit to determine the status of the processor and the outcome of previous ALU operations. For example, for the increment-and-skip-if-zero (ISZ) instruction, the control unit will increment the PC if the zero flag is set.

Control signals from control bus: The control bus portion of the system bus provides signals to the control unit. Such as interrupt signal and acknowledgements.

The outputs are as follows:

• *Control signals within the processor:* These are two types: those that cause data to be moved from one register to another, and those that activate specific ALU functions.

• *Control signals to control bus:* These are also of two types: control signals to memory, and control signals to the I/O modules.

A Control Signals Example

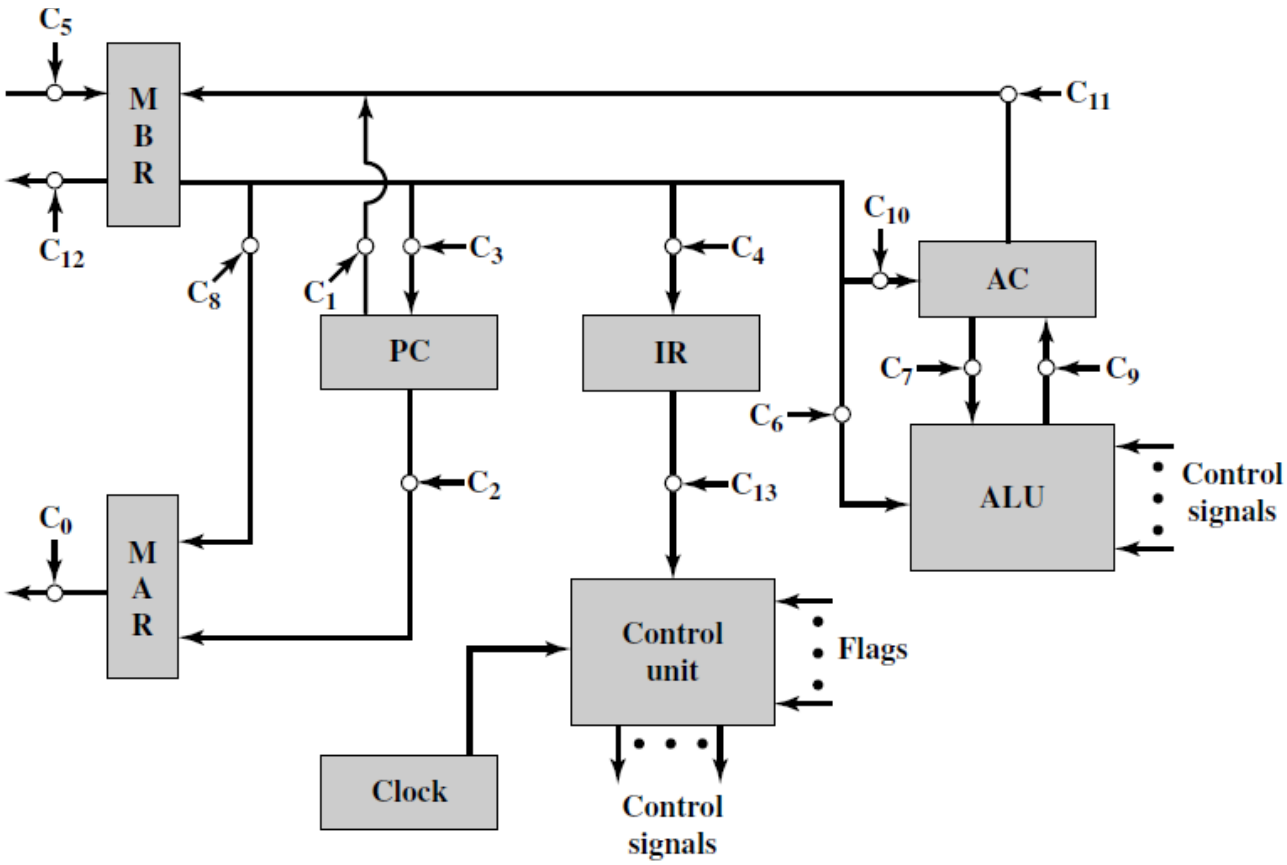


Figure 5.5 Data paths and Control signal

To illustrate the functioning of the control unit, let us examine a simple example. Figure 5.5 illustrates the example. This is a simple processor with a single accumulator (AC). The data paths between elements are indicated. The control paths for signals emanating from the control unit are not shown, but the terminations of control signals are labeled Ci and indicated by a circle. The control unit receives inputs from the clock, the instruction register, and flags. With each clock cycle, the control unit reads all of its inputs and emits a set of control signals. Control signals go to three separate destinations:

Data paths: The control unit controls the internal flow of data. For example, on instruction fetch, the contents of the memory buffer register are transferred to the instruction register. For each path to be controlled, there is a switch (indicated by a circle in the figure).A control signal from the control unit temporarily opens the gate to let data pass.

ALU: The control unit controls the operation of the ALU by a set of control signals. These signals activate various logic circuits and gates within the ALU.

System bus: The control unit sends control signals out onto the control lines of the system bus (e.g., memory READ).

The control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading all of its inputs, the control unit emits a sequence of control signals that causes micro-operations to occur. It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize.
Table 5.1 indicates the control signals that are needed for some of the micro-operation sequences described earlier. For simplicity, the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR are not shown.

	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

C_R = Read control signal to system bus.
C_W = Write control signal to system bus.

Table 5.1: Micro-operations and Control Signals

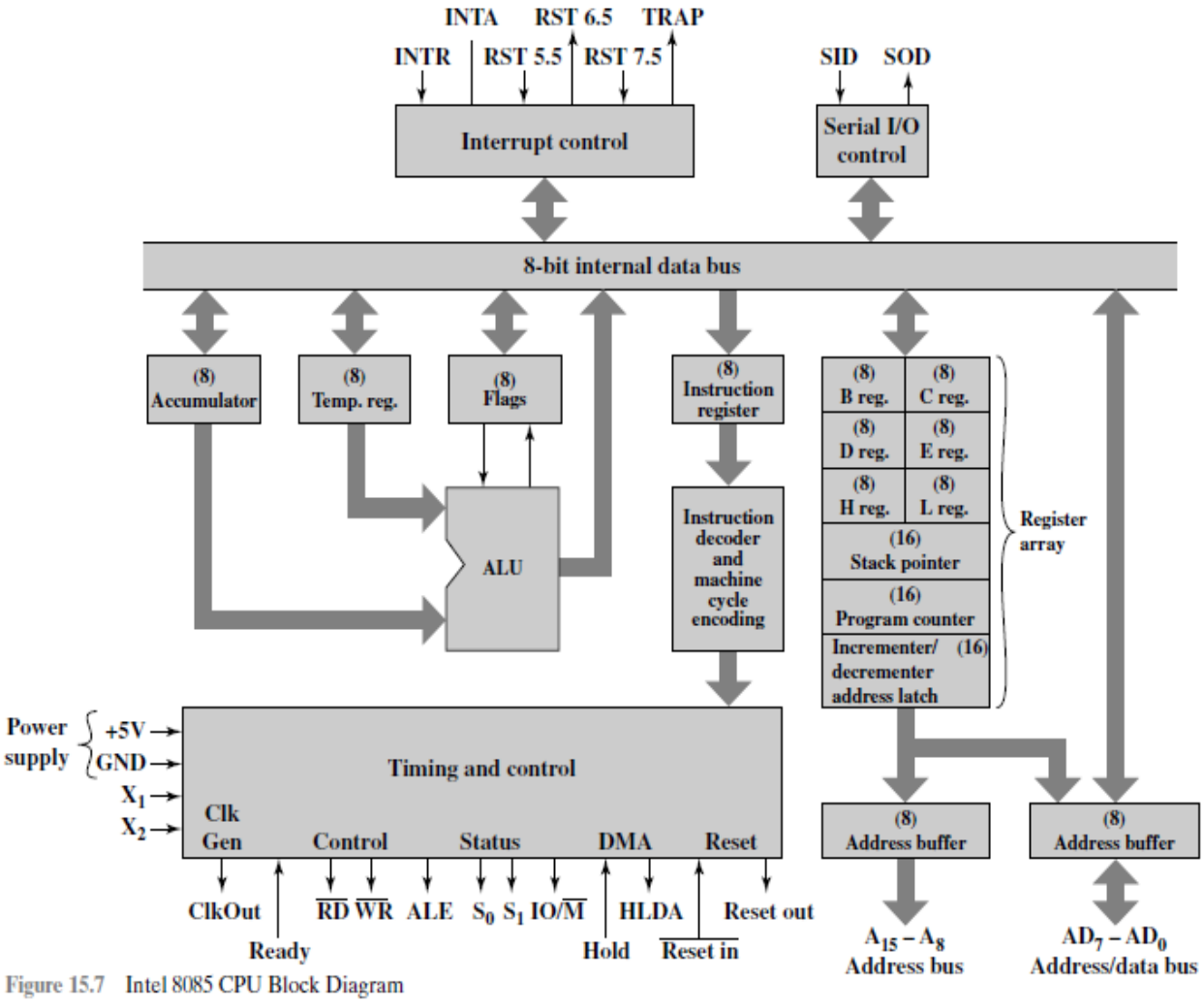


Figure 15.7 Intel 8085 CPU Block Diagram

Figure: 5.6 Intel 8085 CPU Block Diagram

Types of control unit

Till now we have discussed control unit in terms of inputs and function, now we focus our attention on the technique used for implementation.

1. Hardwired implementation
2. Micro programmed implementation

imp

1. Hardwired implementation

In hardwired implementation, the control unit is essentially a combinational circuit. i.e. its input logic signals are transferred into a set of output logic signals which are control signal

Control unit inputs

The key inputs are the instruction register, the clock, flags, and control bus signals. In the case of the flags and control bus signals, each individual bit typically has some meaning (e.g. Overflow).The other two inputs, however, are not directly useful to the control unit. First consider the instruction register. The control unit makes use of the op-code and will perform different

actions (issue a different combination of control signals) for different instructions. To simplify the control unit logic, there should be a unique logic input for each opcode. This function can be performed by a decoder, which takes an encoded input and produces a single output. In general, a decoder will have n binary inputs and 2^n binary outputs. Each of the 2^n different input patterns will activate a single unique output. The de-coder for a control unit will typically have to be more complex than that, to account for variable-length opcodes. The clock portion of the control unit issues a repetitive sequence of pulses. This is useful for measuring the duration of micro-operations. Essentially, the period of the clock pulses must be long enough to allow the propagation of signals along data paths and through processor circuitry. However, as we have seen, the control unit emits different control signals at different time units within a single instruction cycle. Thus, we would like a counter as input to the control unit, with a different control signal being used for T1, T2,….. and so forth. At the end of an instruction cycle, the control unit must feed back to the counter to reinitialize it at. T_1.

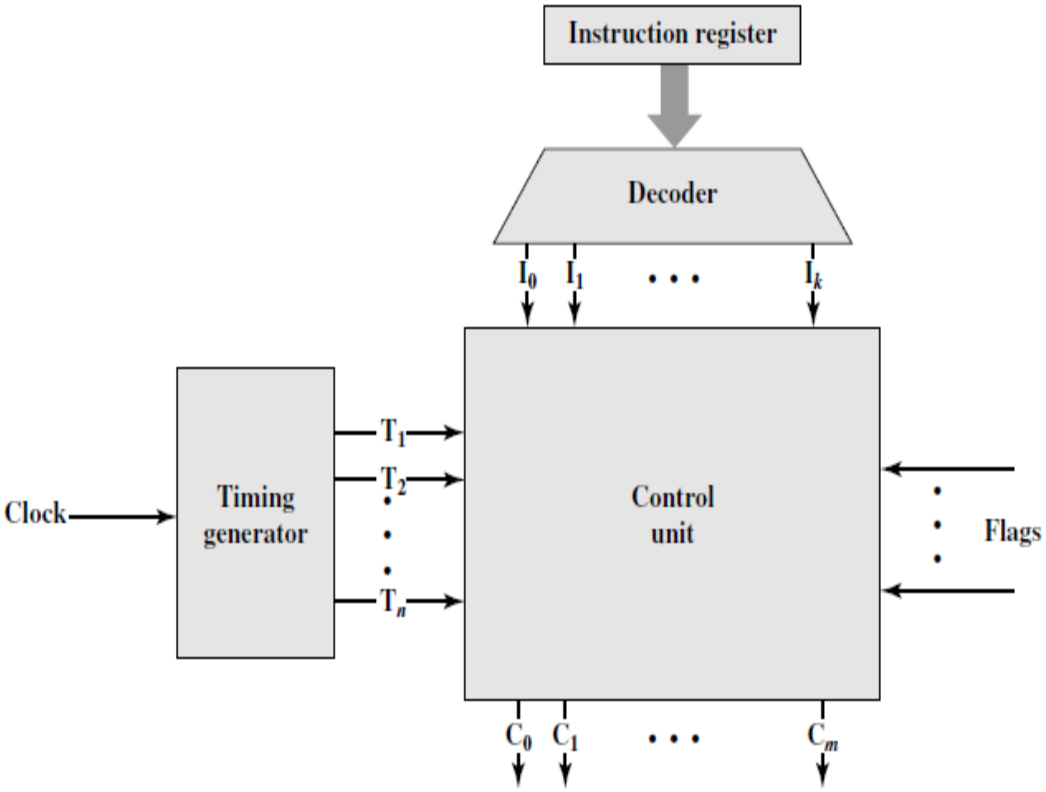


Figure 5.6: Control unit with decoded inputs.

Control Unit Logic

To define the hardwired implementation of a control unit, all that remains is to discuss the internal logic of the control unit that produces output control signals as a function of its input signals. Let us consider a single control signal, C5. This signal causes data to be read from the external data bus into the MBR. We can see that it is used twice in Table 5.1. Let us define two new control signals, P and Q that have the following interpretation:

PQ = 11 Interrupt Cycle
PQ = 10 Execute Cycle
PQ = 01 Indirect Cycle
PQ = 00 Fetch Cycle

Then the following Boolean expression defines C5: as That is, the control signal C5 will be asserted during the second time unit of both the fetch and indirect cycles.

$$C_5 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2$$

This expression is not complete.C5 is also needed during the execute cycle. For our simple example, let us assume that there are only three instructions that read from memory: LDA, ADD, and AND. Now we can define C5 as

$$C_5 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2 + P \cdot \overline{Q} \cdot (LDA + ADD + AND) \cdot T_2$$

This same process could be repeated for every control signal generated by the processor. The result would be a set of Boolean equations that define the behavior of the control unit and hence of the processor.

Problems with hardwired approach

- Complex sequencing and micro operations logic
- Difficult to design and test.
- Inflexible design.
- Difficult to add new instructions.

Micro programmed control unit

The concept of micro-programming was developed by Maurice wilkes in 1951, using diode matrices for memory element. A micro-program consists of sequence of micro-instruction in a micro- programming. Micro-programmed control unit is a relatively logic circuit that is capable of sequencing through micro-instruction and generating control signals to execute each micro-instruction. Consider that for each micro-operation all that the control unit is allowed is to do is generate a set of control signals. i.e. each control lines of control unit are either on or off. This condition can be represented by a binary digit for each control lines. So we would construct a control word in which each bit represents one control line. Then each micro-operation would be represented by a different pattern of 1's and 0's in control word. Now the sequence of control word or micro-operations is not fixed, so we put the control words in a memory location called control memory.

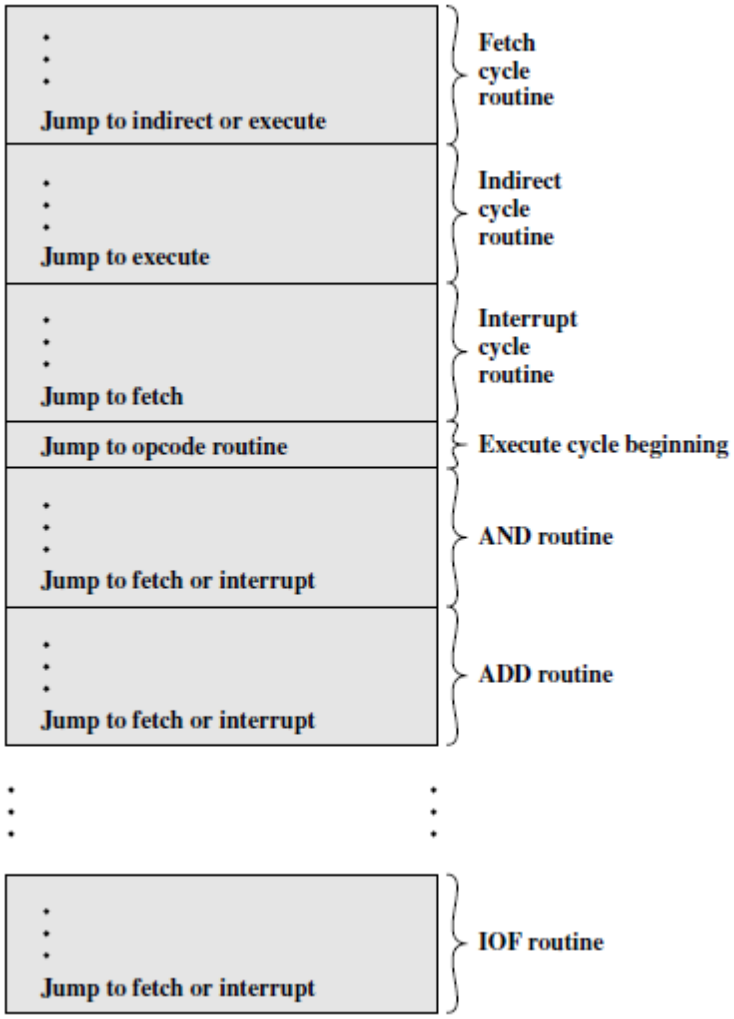


Figure 5.7: Organization of Control Memory

Figure shows the sequence of micro-operations to be performed during each cycle.

Working of micro-programmed implementation

The control memory of Figure5.7 contains a program that describes the behavior of the control unit. It follows that we could implement the control unit by simply executing that program. Figure 5.7 shows the key elements of such an implementation. The set of microinstructions is stored in the *control memory.* The *control address register* contains the address of the next microinstruction to be read. When a microinstruction is read from the control memory, it is transferred to a *control buffer register.*

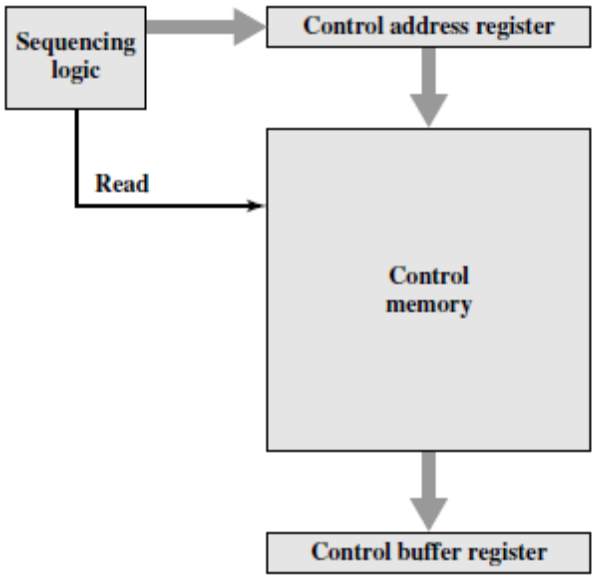


Figure 5.7: Control Unit micro architecture

Thus, reading a microinstruction from the control memory is the same as *executing* that microinstruction. The third element shown in the figure is a sequencing unit that loads the control address register and issues a read command. Let us examine this structure in greater detail, as depicted in Figure 16.4.Comparing this with Figure 16.4, we see that the control unit still has the same inputs (IR,ALU flags, clock) and outputs (control signals).The control unit functions as follows:

1. To execute an instruction, the sequencing logic unit issues a READ command to the control memory.
2. The word whose address is specified in the control address register is read into the control buffer register.
3. The content of the control buffer register generates control signals and next address information for the sequencing logic unit.
4. The sequencing logic unit loads a new address into the control address register based on the next-address information from the control buffer register and the ALU flags.

All this happens during one clock pulse. The last step just listed needs elaboration. At the conclusion of each microinstruction, the sequencing logic unit loads a new address into the Control address register. Depending on the value of the ALU flags and the control buffer register, one of three decisions is made:

• **Get the next instruction:** Add 1 to the control address register.
• **Jump to a new routine based on a jump microinstruction:** Load the address field of the control buffer register into the control address register.
• **Jump to a machine instruction routine:** Load the control address register based on the opcode in the IR.

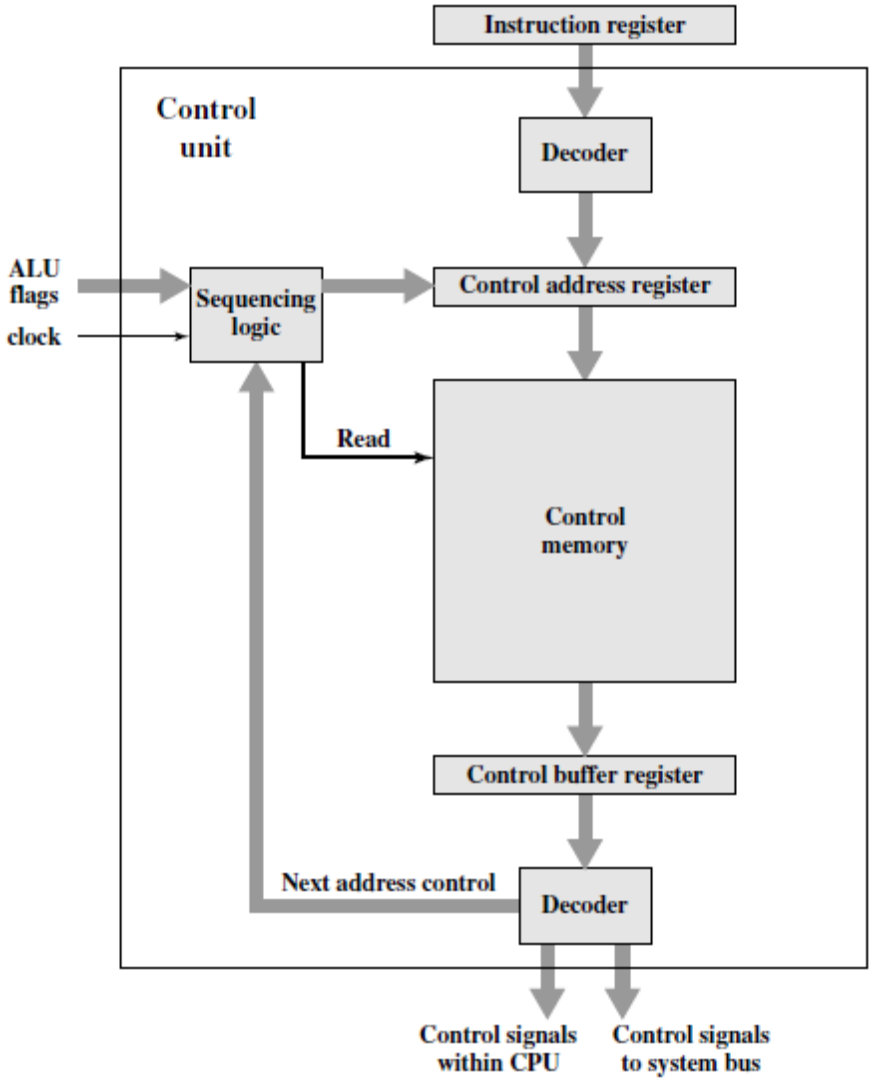


Figure 5.8: Functioning of Micro-programmed Control Unit

Advantage and Disadvantage

The principle advantage of the use of micro-programming is to implement a control unit is that it simplifies the design of control unit, so cheaper and less error prone. A hardwired control unit must contain complex logic for sequencing through the many micro-operations of instruction cycle. The decoder and sequencing logic unit of micro-programmed control unit are very simple piece of logic. Principle disadvantage of micro-programmed unit is that it will be slower than hardwired unit. Micro-programmed is a dominant technique for implementing control units in pure CICS architecture and hardware control unit in RISC architecture.

RISC: Reduced Instruction Set Computer
CISC: Complex Instruction Set Computer

The two basic tasks performed by a micro-programmed control unit are as follows:

1) Micro instruction sequencing: the micro-programmed control unit gets the next microinstruction from control memory.

2) Micro-instruction execution: the micro-programmed control unit generated the control signal needed to execute the micro- instruction. The control unit design must consider both which affect the format of the micro-instruction and the timing of control unit.

Micro- instruction sequencing

Two concerns are involved in the design of a microinstruction sequencing technique, the size of the microinstruction and the address-generation time. The first concern is obvious; minimizing the size of the control memory reduces the cost of that component. The second concern is simply a desire to execute microinstructions as fast as possible. In executing a micro-program, the address of the next microinstruction to be executed is in one of these categories:

- Determined by instruction register
- Next sequential address
- Branch

First occurs only once per instruction cycle, just after instruction is fetched. Second is the most common; however the design cannot be optimized just for sequential access. Third branches both conditional and unconditional are necessary and tend to occur more often.

Sequencing Techniques imp

Based on the current microinstruction, condition flags, and the contents of the instruction register, a control memory address must be generated for the next microinstruction. A wide variety of techniques have been used. We can group them into three general categories, as illustrated in Figures 16.6 to 16.8. These categories are based on the format of the address information in the microinstruction:

- Two address fields
- Single address field
- Variable format

Two address fields

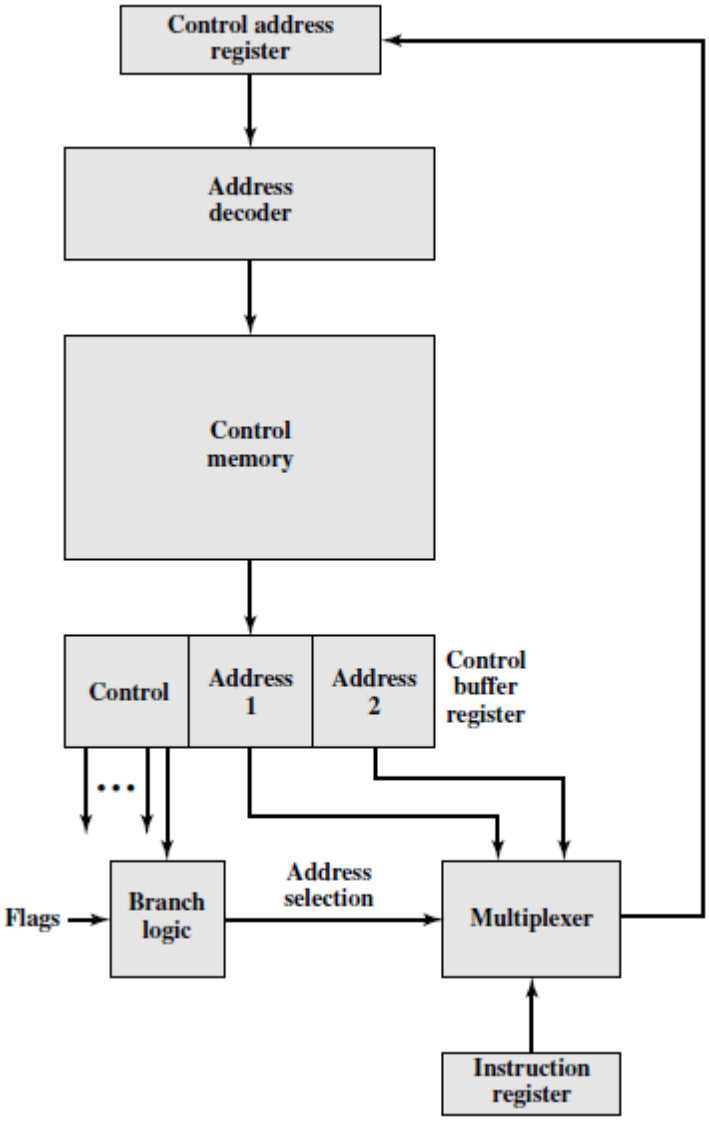


Figure 5.8: Branch Control Logic: Two Address field.

The simplest approach is to provide two address field in each micro-instruction. A multiplexer is provided that serves as a destination for both address fields plus the instruction register. Based on the address selection input multiplexer transmits either the op-code or one of the two address to Control Access Register (CAR). The CAR subsequently decodes to produce next micro-instruction address. The address selection signals are provided by a branch logic module whose input consists of control unit flags plus from control portion of micro-instruction.

Single address field

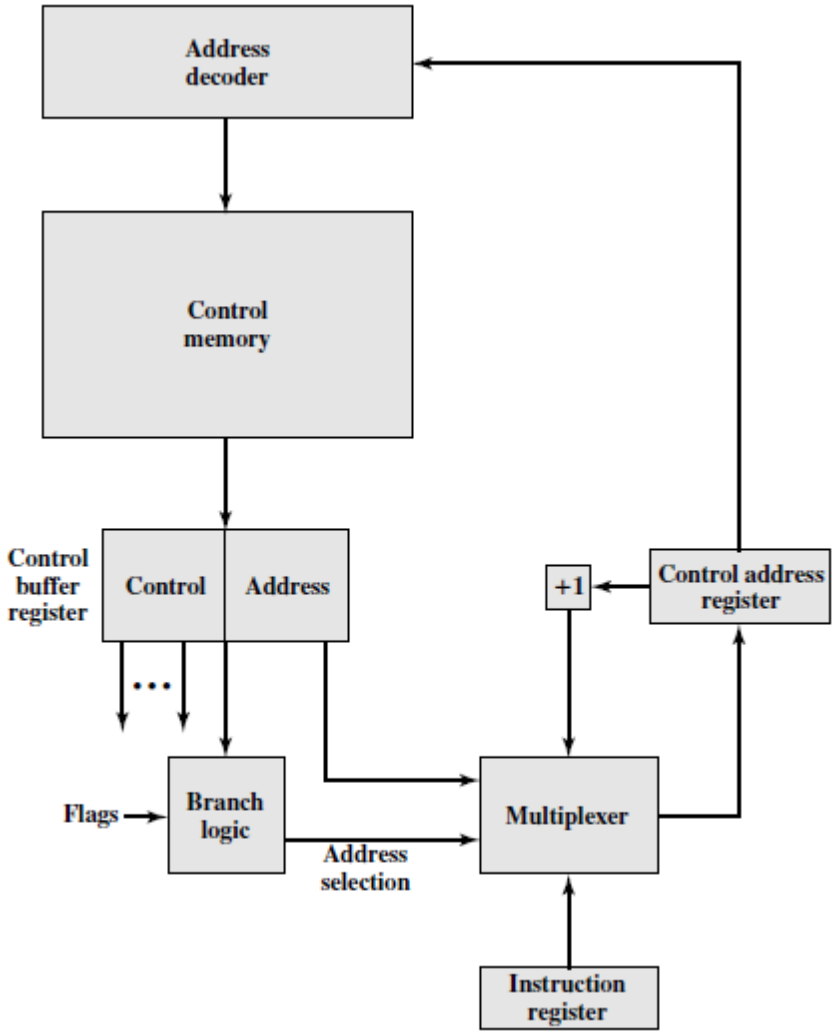


Figure 5.9: Branch Control Logic: Single Address field

A common approach is to use single address field, with this the next address can be specified by
- Address field
- Instruction register code
- Next sequential address

The address) selection signal determines which option is selected. This approach reduces the number of address fields to one.

Variable format

Another is to provide for two entirely different micro-instruction formats. One bit designates which format is being used, in one format the remaining bits are used to activate control signals. In other format some bits drive the branch logic module and the remaining bits provide the address. For the first format the next address is either the next sequential address or an address derived from the instruction register. With the second format either a conditional or unconditional branch is being specified. One dis-advantage of this approach is that one entire cycle is consumed with each branch micro-instruction.

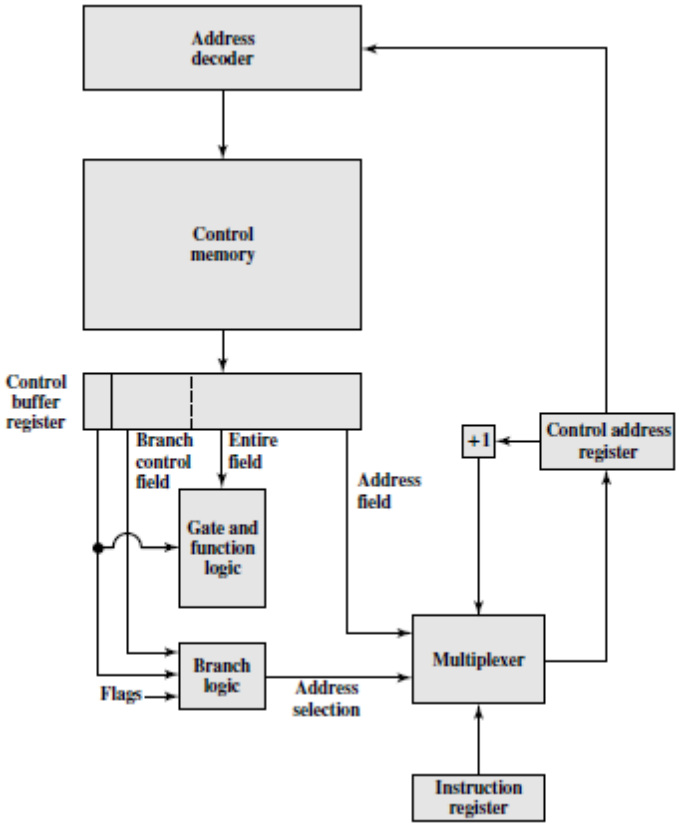


Figure 5.10: Branch Control Logic: Variable Format

Micro-Instruction execution

The micro-instruction is made up of two parts fetch and execute. The fetch is to fetch data and execute is to perform instruction. The fetch is to fetch data and execute is to perform instruction. The effect of the execution of a micro-instruction is to generate control signals for both internal control to processor and the external control to processor. An organization of control unit is:

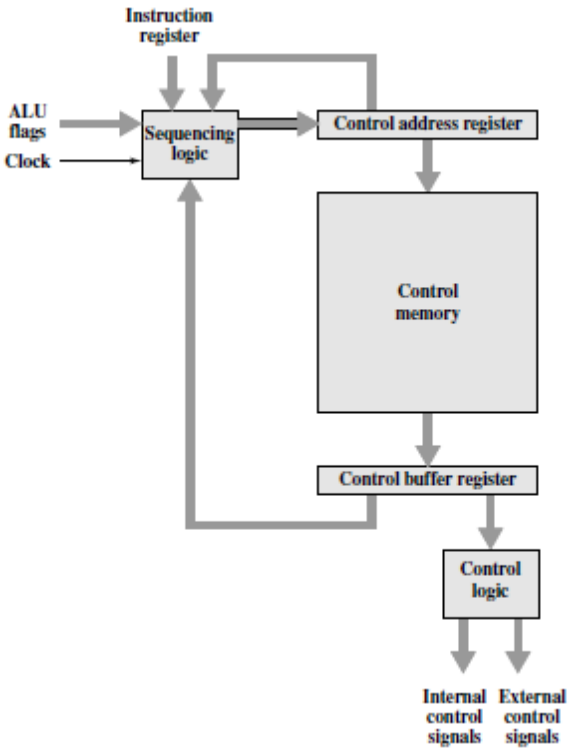


Figure 5.11: Control Unit Organization

Here are the major modules of control unit. The sequencing logic module contains the logic to perform the function. It generates address of the next micro-instruction using input as instruction register, ALU, flags, CAR and control buffer register. The last may provide an actual address control bits or both. The module is driven by clock that determines the timing of the micro-instruction cycle. Control logic module generates control signal as a function of some of bits in the micro-instruction. It should be clear that the format and content of the micro-instruction will determine the complexity of the control logic module.

Micro programming applications

For typically large microprocessor system today, there are many instruction and associated register level hardware. The are many control points to be manipulated.

Emulation:
- The use of a micro-program on one machine to execute programs originally written to run on another machine.

- By changing the microcode of a machine can execute software from another machine.

Classification of micro-instruction

- Vertical micro-programming
- Horizontal micro-programming

** Assignment to submit classification on micro-instruction

imp

Reduced Instruction Set Computer (RISC)

In early 1980s a number of computer designers recommended that computers users use fewer instructions with a simple construct so that they can be executed much faster within the CPU without having to use memory as often. The concept of RISC architecture involves attempt to reduce the execution time by simplifying the instruction set of the computer.
The major characteristics of RISC processor are:

1. Relatively few instruction.
2. Few and simple addressing modes.
3. Memory access limited to load and store instruction.
4. All operations done within the register of CPU.
5. Fixed length, easily decoded instruction format.
6. Single cycle instruction execution.
7. Hardwired rather than micro programmed control.

Complex Instruction Set Computer (CISC)

The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in high level language. The major characteristics of CICS architecture are:

1. Large number of instruction typically from hundreds to 250 instructions.
2. Some instruction that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes typically from 5 to 20 different modes.
4. Variable length instruction formats.
5. Instruction that manipulate operands in memory.
6. Simple compilers will do the job.
7. Multiple cycle instruction (single HLL instruction is broken into smaller instruction).
8. Micro programmed implementation.
9. Simple control unit.

6. Memory Organization

6.1. Computer system overview

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional capacity, but the general purpose computer required more memory to store the programs and data. There is not much of memory to accommodate all programs used in a typical computer, also not all accumulated information is used by processor at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU. The memory unit that communicated directly with the CPU is called the *main memory*. Devices that provide backup storage are called *auxiliary memory*. The most common auxiliary memory devices used in computer system are magnetic disk and tapes, used for storing system programs, large data files and other back up information. Only programs that are currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.

The memory system can be classified into different categories according to the characteristics as follows:

1 **Location:** Internal (e.g. processor registers, main memory, cache). External (e.g. optical disk, magnetic disk, tapes).

2 **Capacity:** Number of words, bytes, for internal memory this is typically expressed in terms of bytes, or word, common capacity lengths are 8, 16, and 32. External memory capacity is typically expressed in terms of bytes.

3 **Unit of transfer:** For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

   i. **Word:** The "natural" unit of organization of memory. The size of the word is typically equal to the number of bits used to represent an integer and to the instruction length.

   ii. **Addressable units:** In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level.

   iii. **Unit of transfer:** For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an addressable unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks.

4. **Access method:** it refers to the access method used for data and information. It includes
    i. **Sequential access:** Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Thus, the time to access an arbitrary record is highly variable.
    ii. **Direct access:** individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable.
    iii. **Random access:** Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed.
    iv. **Associative:** This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address.
5. **Performance: three** performance parameters are
    1. **Access time (latency):** For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.
    2. **Memory cycle time:** This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively.
    3. **Transfer rate:** This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/ (cycle time). For non-random-access memory, the following relationship holds:

$$T_N = T_A + \frac{n}{R}$$

Where, TN: Average time to read or write N bits

TA: Average access time

n: Number of bits

R: Transfer rate, in bits per second (bps)

6. **Physical type:** Semiconductor, magnetic, optical, magneto-optical.
7. **Physical characteristics:** Volatile/ non- volatile, Erasable/ non Erasable
8. **Organization:** By *organization* is meant the physical arrangement of bits to form words.

6.2. The memory hierarchy

The design constraints on a computer's memory can be summed up by three questions, how much? How fast and how expensive?

- Faster access: greater cost per bit.
- Greater capacity: smaller cost per bit.
- Greater capacity: slower access time.

The hierarchy shows the way how the memory comp

hierarchy the following occurs:

a) Decrease cost per bit.
b) Increasing capacity.
c) Increasing access time.
d) Decreasing frequency of access by processor.

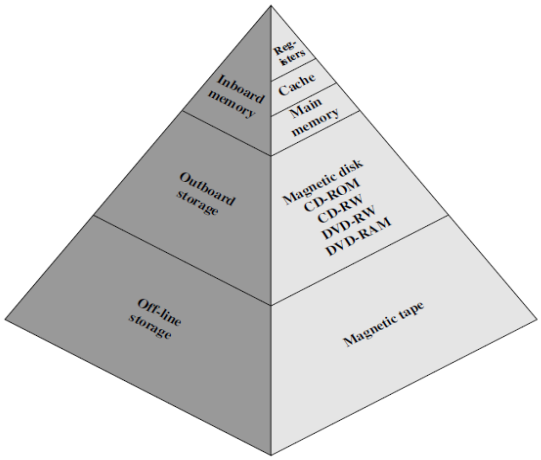


Figure 6.1: Memory hierarchy

6.3. Semiconductor main memory

In earlier computers, the most common form of random-access storage for computer main memory employed an array of doughnut-shaped ferromagnetic loops referred to as cores. It stored data by using induction (hysteresis loop) method. 1 core = 1-bit storage. Today, the use of semiconductor chips for main memory is almost universal. The characteristics of those include it store 1 bit or 1 or 0 and are capable of being read and written. The basic element of semiconductor memory is called cell. Figure below shows the basic cell of a semiconductor memory.

Figure below depicts the operation of a memory cell. Most commonly, the cell has three functional terminals capable of carrying an electrical signal. The select terminal, as the name suggests, selects a memory cell for a read or write operation. The control terminal indicates read or write. For writing, the other terminal provides an electrical signal that sets the state of the cell to 1 or 0. For reading, that terminal is used for output of the cell's state.

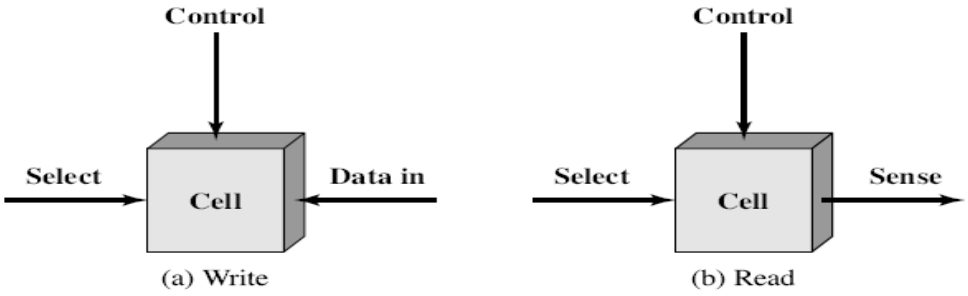


Figure 6.2: Memory cell operation

6.3.1. Random Access Memory (RAM)

The characteristics of RAM is to both read and write data from memory easily and rapidly, both reading and writing are accomplished through the use of electrical signals. RAM is that it is volatile. A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost.

6.3.1.1. Types of RAM

a) Dynamic Random Access Memory(DRAM):

A dynamic RAM (DRAM) is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as a binary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic RAM require periodic charge refreshing to maintain data storage. The term dynamic refers to this tendency of the stored charge to leak away, even with power continuously applied.

Figure shows the typical DRAM structure for an individual cell that stores 1 bit. The address line is activated when the bit value from this cell is to be read written. The transistor acts as a switch, allowing or stopping the current flow.

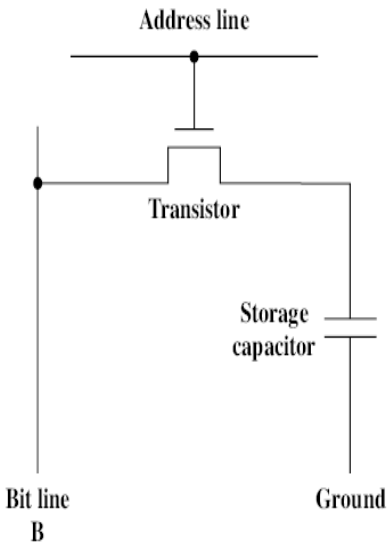


Figure 6.3: Single DRAM cell

For the write operation, a voltage signal is applied to the bit line; a high voltage represents 1, and a low voltage represents 0. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor.

For the read operation, when the address line is selected, the transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to a sense amplifier. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains logic 1

or logic 0. The readout from the cell discharges the capacitor, which must be restored to complete the operation.

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analog device. The capacitor can store any charge value within a range; a threshold value determines whether the charge is interpreted as 1 or 0. Figure shows the circuit of one DRAM cell.

b) Static Random Access Memory (SRAM):

A static RAM (SRAM) is a digital device that uses the same logic elements used in the processor. In a SRAM, binary values are stored using traditional flip-flop logic-gate configurations (see Chapter 20 for a description of flip-flops). A static RAM will hold its data as long as power is supplied to it.

Figure shows a typical SRAM structure for an individual cell. Four transistors (T1, T2, T3, and T4) are cross connected in an arrangement that produces a stable logic state. In logic state 1, point C1 is high and point C2 is low; in this state, T1 and T4 are off and T2 and T3 are on. In logic state 0, point C1 is low and point C2 is high; in this state, T1 and T4 are on and T2 and T3 are off. Both states are stable as long as the direct current (dc) voltage is applied. Unlike the DRAM, no refresh is needed to retain data. As in the DRAM, the SRAM address line is used to open or close a switch. The address line controls two transistors (T5 and T6). When a signal is applied to this line, the two transistors are switched on, allowing a read or write operation. For a write operation, the desired bit value is applied to line B, while its complement is applied to line. This forces the four transistors (T1, T2, T3, and T4) into the proper state. For a read operation, the bit value is read from line B.

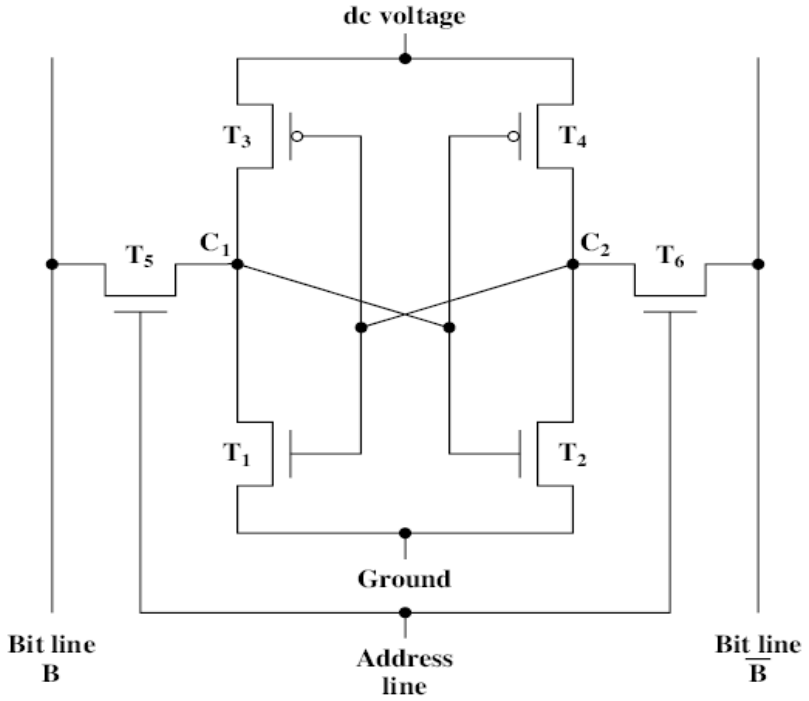


Figure 6.4: SRAM cell

** suppose logic 1 is stored then C1 is high and C2 is low. To read B & $\overline{B}$ are pre charged to Vdd before address line is allowed to be high. Now for logic high combination T2 & T3 will be on and T1 & T4 will be off. The value from line $\overline{B}$ will be drained and B will be retained to dc voltage so B=1 & $\overline{B}$ =0. For opposite case C1=0 & C2=1.

SRAM VERSUS DRAM: Both static and dynamic RAMs are volatile, i.e. power must be continuously supplied to the memory to preserve the bit values. A dynamic memory cell is simpler and smaller than a static memory cell. Thus, a DRAM is more dense (smaller cells more cells per unit area) and less expensive than a corresponding SRAM. On the other hand, a DRAM requires the supporting refresh circuitry.

6.3.1.2. Module organization

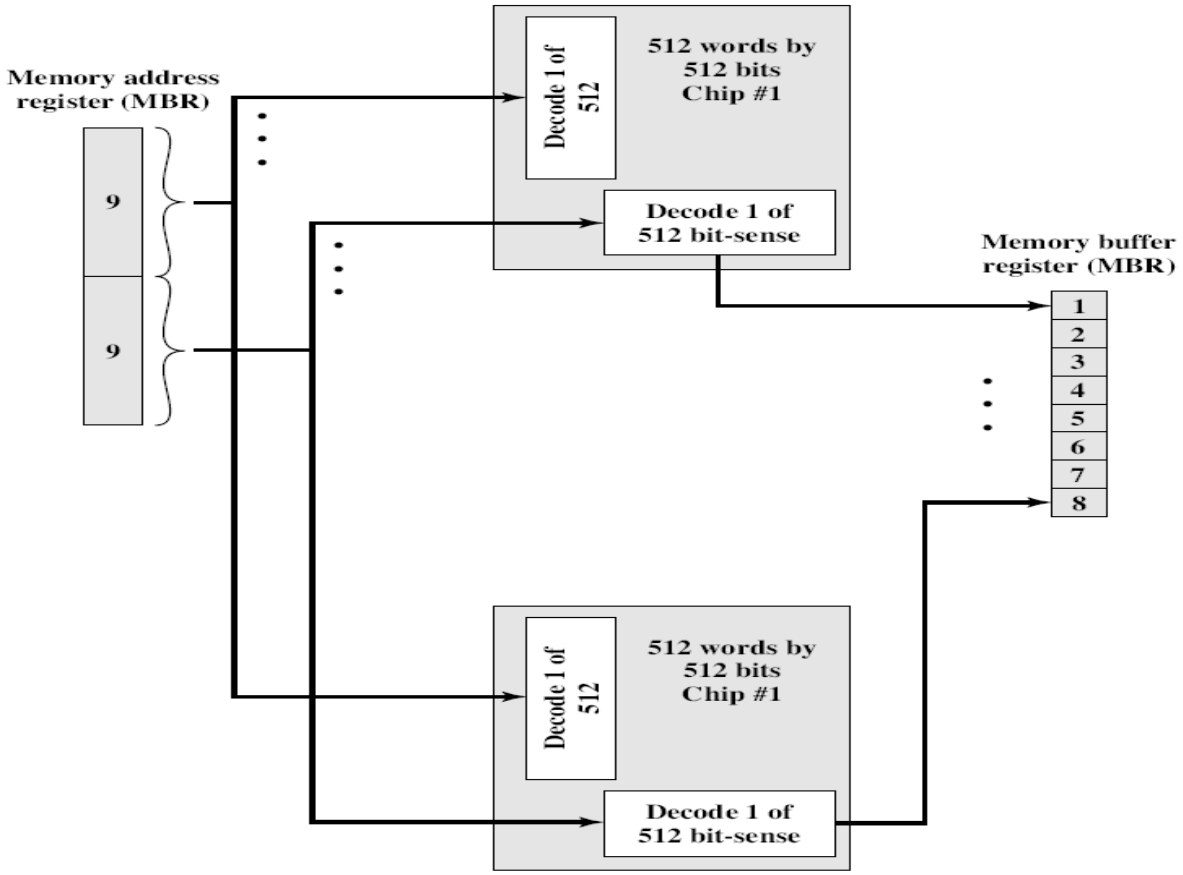


Figure 5.5: 256-KByte Memory Organization

If a RAM chip contains only 1 bit per word, then clearly we will need at least number of chips equal to the number of bits per word. As an example, Figure 5.5 shows how a memory module consisting of 256K 8-bit words could be organized. For 256K words, an 18-bit address is needed and is supplied to the module from some external source (e.g., the address lines of a bus to which the module is attached). The address is presented to 8 256K 1-bit chips, each of which provides the input/output of 1 bit. This organization works as long as the size of memory equals the number of bits per chip. In the case in which larger memory is required, an array of chips is needed.

6.3.1.3. Advanced DRAM organization

a. Synchronous DRAM (SDRAM)

One of the most widely used forms of DRAM is the synchronous DRAM (SDRAM). Unlike the traditional DRAM, which is asynchronous, the SDRAM exchanges data with the processor synchronized to an external clock signal and running at the full speed of the processor/memory bus without imposing wait states. In a typical DRAM, the processor presents addresses and control levels to the memory, indicating that a set of data at a particular location in memory should be either read from or written into the DRAM. After a delay of access time, the DRAM

either writes or reads the data. During the access-time delay, the DRAM performs various internal functions, such as activating the high capacitance of the row and column lines, sensing the data, and routing the data out through the output buffers. The processor must simply wait through this delay, slowing system performance. With synchronous access, the DRAM moves data in and out under control of the system clock. The processor or other master issues the instruction and address information, which is latched by the DRAM. The DRAM then responds after a set number of clock cycles. Meanwhile, the master can safely do other tasks while the SDRAM is processing the request. Figure 5.6 shows the internal logic of IBM's 64-Mb SDRAM [IBM01], which is typical of SDRAM organization.

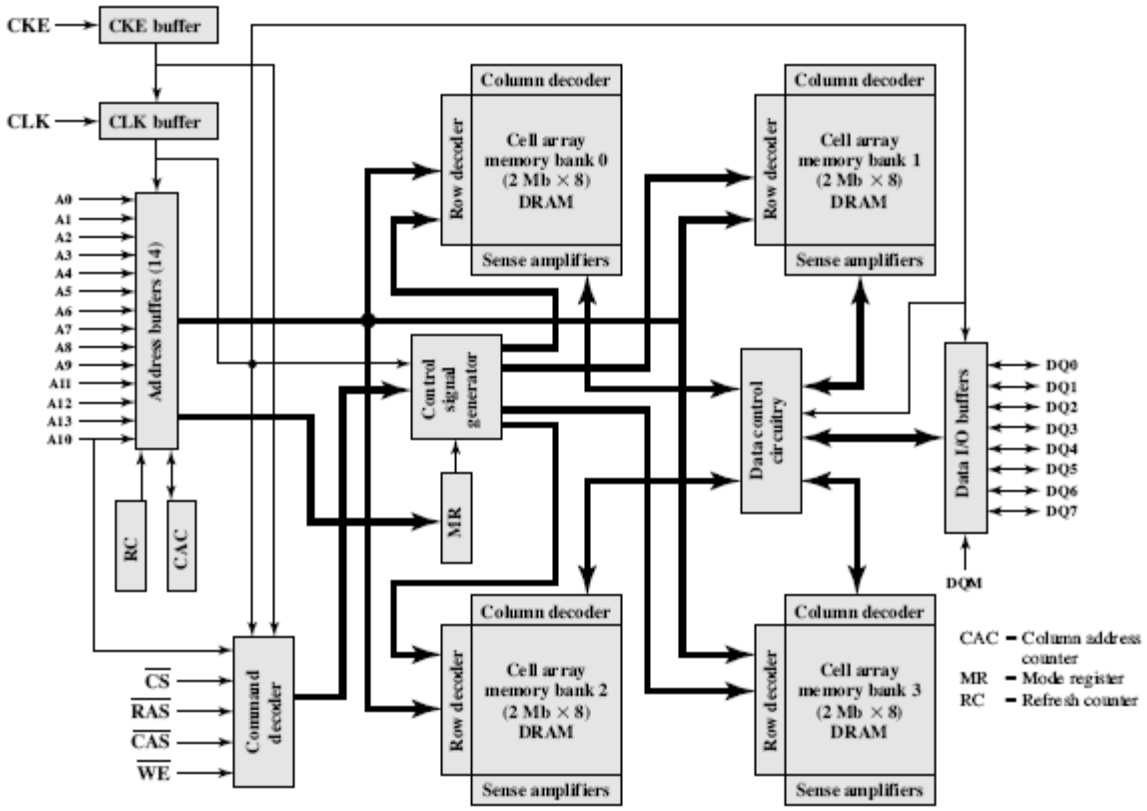


Figure 6.6: Synchronous DRAM

b. Rambus DRAM (RDRAM)

RDRAM, developed by **Rambus,** has been adopted by Intel for its Pentium and Itanium processors. It has become the main competitor to SDRAM. RDRAM chips are vertical packages, with all pins on one side. The chip exchanges data with the processor over 28 wires no more than 12 centimeters long. The bus can address up to 320 RDRAM chips and is rated at 1.6 Gbps. The special RDRAM bus delivers address and control information using an asynchronous block-oriented protocol. After an initial 480 ns access time, this produces the 1.6 Gbps data rate. What makes this speed possible is the bus itself, which defines impedances, clocking, and signals very

precisely. RDRAM gets a memory request over the high-speed bus. This request contains the desired address, the type of operation, and the number of bytes in the operation.

Figure illustrates the RDRAM layout. The configuration consists of a controller and a number of RDRAM modules connected via a common bus. The controller is at one end of the configuration, and the far end of the bus is a parallel termination of the bus lines. The bus includes 18 data lines (16 actual data, two parity) cycling at twice the clock rate; that is, 1 bit is sent at the leading and following edge of each clock signal. There is a separate set of 8 lines (RC) used for address and control signals. There is also a clock signal that starts at the far end from the controller propagates to the controller end and then loops back. A RDRAM module sends data to the controller synchronously to the clock to master, and the controller sends data to an RDRAM synchronously with the clock signal in the opposite direction. The remaining bus lines include a reference voltage, ground, and power source.

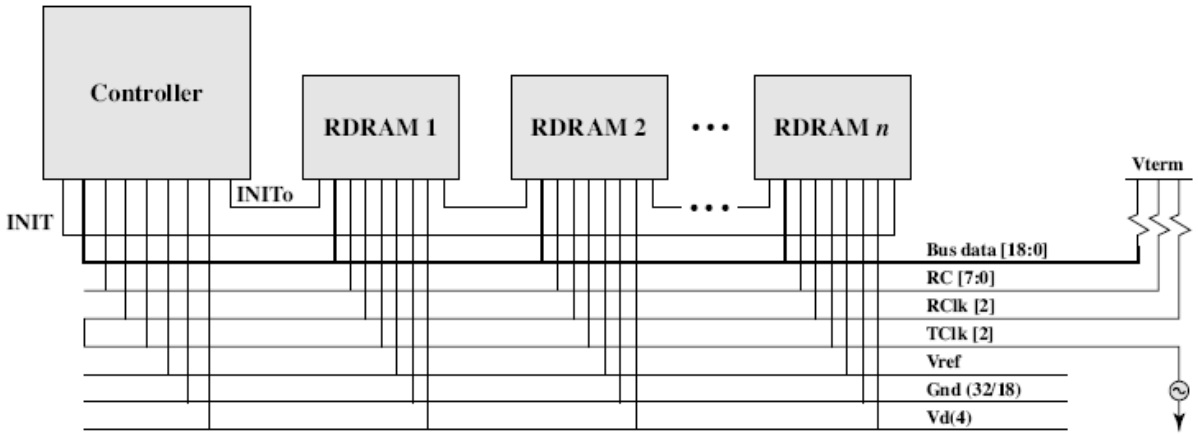


Figure 6.7: RDRAM Structure

c. DDRSRAM

SDRAM is limited by the fact that it can only send data to the processor once per bus clock cycle. Double-data-rate SDRAM can send data twice per clock cycle, once on the rising edge of the clock pulse and once on the falling edge of clock pulse. Theoretically, a DDR module can transfer data at a clock rate in the range of 200 to 600 MHz, a DDR2 module transfers at a clock rate of 400 to 1066 MHz and a DDR3 module transfers at a clock rate of 800 to 1600 MHz. In practice, somewhat smaller rates are achieved.

d. Cache DRAM

Cache DRAM integrates a small SRAM cache (16 Kb) on a generic DRAM chip. It can be sued in two ways:

a) as a true cache consisting a number of 64-bit line, more effective for ordinary random access to memory

b) SRAM on the CDRAM can also be used as a buffer to support the serial access to a block of data.

6.3.2. Read Only Memory (ROM)

Read Only Memory (ROM) is a nonvolatile, permanent data storage that cannot be changed. While it is possible to read a ROM many times, it is not possible to write new data into it. An important application of ROMs is microprogramming.

6.3.2.1. Types of ROM:

a. Read Only Memory(ROM):

A ROM is created like any other integrated circuit chip, with the data actually wired into the chip as part of the fabrication process. This presents two problems:

- The data insertion step includes a relatively large fixed cost, whether one or thousands of copies of a particular ROM are fabricated.
- There is no room for error. If one bit is wrong, the whole batch of ROMs must be thrown out.

b. Programmable ROM (PROM):

- May be written once.
- Writing process is performed electrically may be performed by supplier or customer.
- Special equipment required writing or programming process.
- Useful for small production runs.

c. Erasable PROM(EPROM):

- Optically erasable by exposing to UV light for 20 minutes.
- Read and write done electrically.
- All storage cell must be erased to write again.
- Can be erased, write, read multiple times and can hold data virtually indefinitely.

d. Electrically EPROM(EEPROM):

- Can be written many times while remaining in the system.
- Does not have to be erased first.

- Programming can be done at individual bytes level.
- Write requires longer time than read.
- Used in system for development, personalization and other task requiring unique information to be stored.

e. Flash memory:

- Intermediate between EPROM & EEPROM in both cost and functioning.
- Like EEPROM used electrical erasing technology.
- Fast erasing of entire blocks or just few blocks.
- Higher density than EEPROM because it uses only one transistor per bit.

**Read chip logic from computer Organization and architecture design for performance by William Stallings.

6.4. Auxiliary Memory

6.4.1. Magnetic disk

- The magnetic disk is a metal or plastic platter coated with magnetizable materials.
- Metal or plastic platter is called substrate, traditionally made from aluminum or aluminum alloy materials, recently glass substrate.

Magnetic Read and Write Mechanism

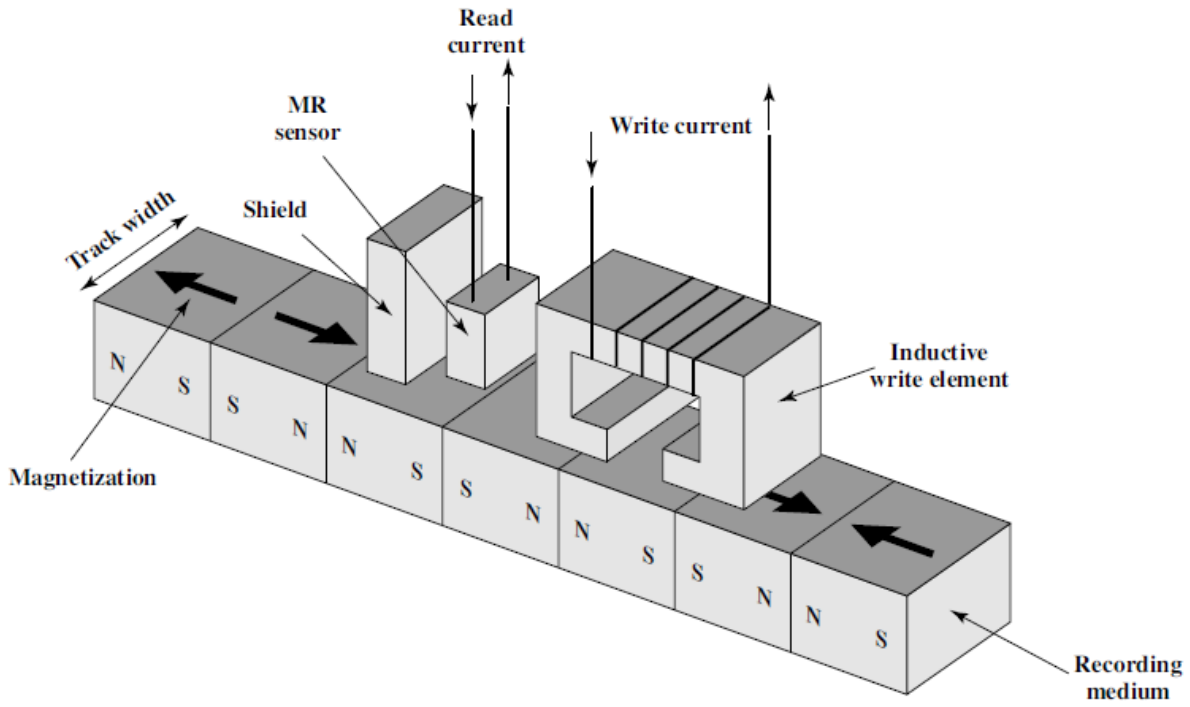


Figure 6.8: inductive wire/ Magnetoresistive Read Head

- Data are recorded and later retrieved from the disk via conducting coil called *head* during the read or write operation the head is stationary while platter rotates beneath it.
- The write mechanism exploits the fact that electricity flowing through a coil produce a magnetic field. For writing data, a pulse of electricity is sent to write head that results in magnetic patterns which are stored on the surface below.
- Write head itself is made from easily magnetizable materials shaped like a rectangular doughnut with a gap on one side and few turns of conducting wire on opposite side.
- An electric current on wire induces a magnetic field that magnetizes small areas of recording medium. Reversing the direction of the currents reverses the direction of magnetization.
- Traditional read mechanism exploits the fact that a magnetic field moving relative to coil produces an electric current in coil. Same way when surface of the disk passes under the head it generates a current of the same polarity as the one already recorded.
- For rigid disk system require a separate read head positioned for convenience close to write head. The read head consists of partially shielded magnetoresistive(MR) sensor. By passing a current through MR sensor, resistance changes are detected as voltage signal. The MR design allows higher frequency operation which equals to greater storage capacity.

Disk characteristics

- Single Vs multiple platters.
- Fixed Vs movable head.
- Removable Vs non removable platters.
- Data access time
  - Seek time: time to position the head over correct track.
  - Rotational latency: wait for the desired sector to come under head.
  - Access time: seek time plus rotational latency.
  - Block transfer time: time to read block of data.
- Disk capacity= M*T*P*S bytes

  S=bytes stored/sector

  P=sector per track

  T=track per surface

  M= no of surface

6.4.2. Magnetic tapes

- Magnetic tape system uses same technique for reading and recording as disk system.

- The medium is flexible polyester tape coated with magnetizable material.
- Tape is similar to home tape recorder system.

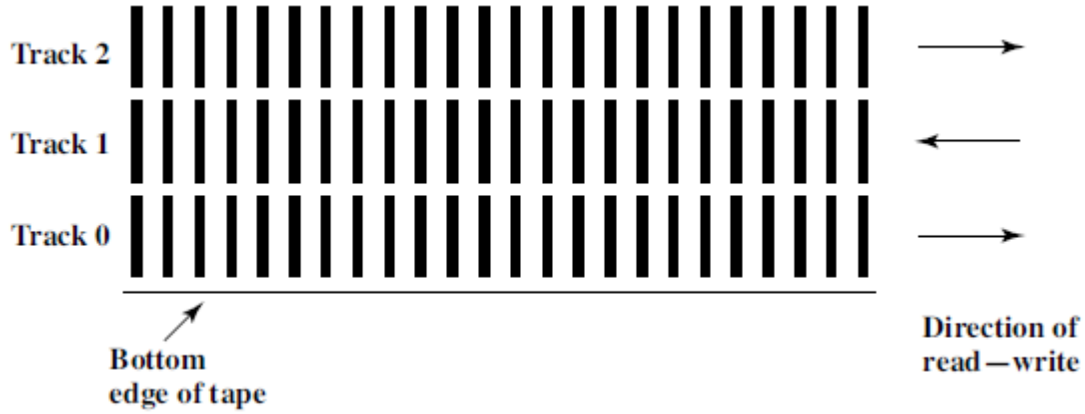


figure 6.9: Read write of tracks in magnetic tapes

- Data on tapes are structured as a number of parallel tracks running lengthwise.
- Earlier system used 9 tracks, later 18 or 36 tracks.
- Modern system use serial recording in which data are laid out as a sequence of bits along each track.
- The typical recording technique used in serial tapes is called serpentine recording.
- In this technique the first set of bits is recorded along the whole length of the tapes.
- When the end of the tape is reached the head is repositioned to record new track and again recorded on its whole length in opposite direction. This process continues back and forth for the whole length of the tapes.
- To increase the speed some read write head is capable of writing on a number of adjacent tracks simultaneously.

6.4.3. Optical memory

6.4.3.1. Compact Disk

The CD (Compact Disk) is a non-erasable disk. Different optical disk is CD-audio, CD-ROM, CD-R, DVD etc.

Compact disk

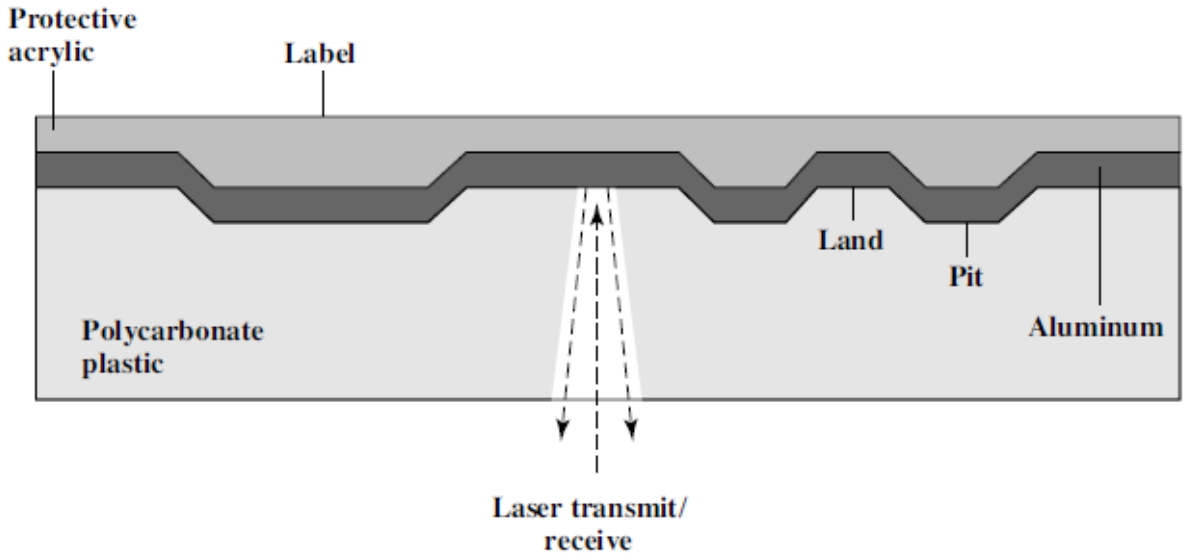


Figure 6.10: CD operation

- The disk is formed from resign, such as polycarbonate.
- Digitally recorded information is imprinted as a series of microscopic pits on the surface of the polycarbonate.
- This is done first of all with a finely focused high intensity laser to create a master disk. The master is used in turn to make a die to stamp out copies of only poly carbonate.
- The pitted surface is then coated with a highly reflective usually aluminum or gold. The shiny surface is protected against dust and scratches by a top coat of acrylic.
- Information is retrieved from CD or CD-ROM by a low powered laser housed in an optical disk player or drive unit.
- The intensity of the light reflected by the surface changes as it encounters pits or lands.
- If the laser beam falls on a pit which has a somewhat rough surface the light scatters and a low intensity is reflected back from surface.
- A land is smooth surface which reflects back at higher intensity.
- The change between pits and land is detected by a photo sensor and converted into a digital signal.
- The beginning or end of pit represents a 1, when no change in elevation occurs between intervals 0 is recorded.

6.4.3.2. CD-Rewritable

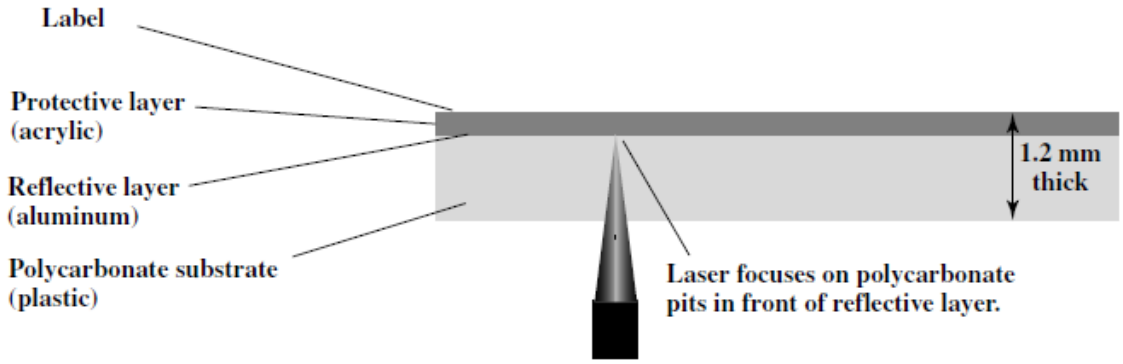
- CD-RW optical disk can be repeatedly written and overwritten. Although a number of approaches have been tried the only pure optical approach that has proved attractive is called phase change.

- The phase change disk uses two materials with different reflectivities.
- The amorphous state-in which molecules exhibit random orientation that reflects light poorly and crystalline state which has a smooth surface that reflects light well.
- A beam of laser can be used to change the material from one phase to other. i.e. erasing.
- The primary disadvantage of phase change optical disk is that the material eventually and permanently loses its desirable properties.

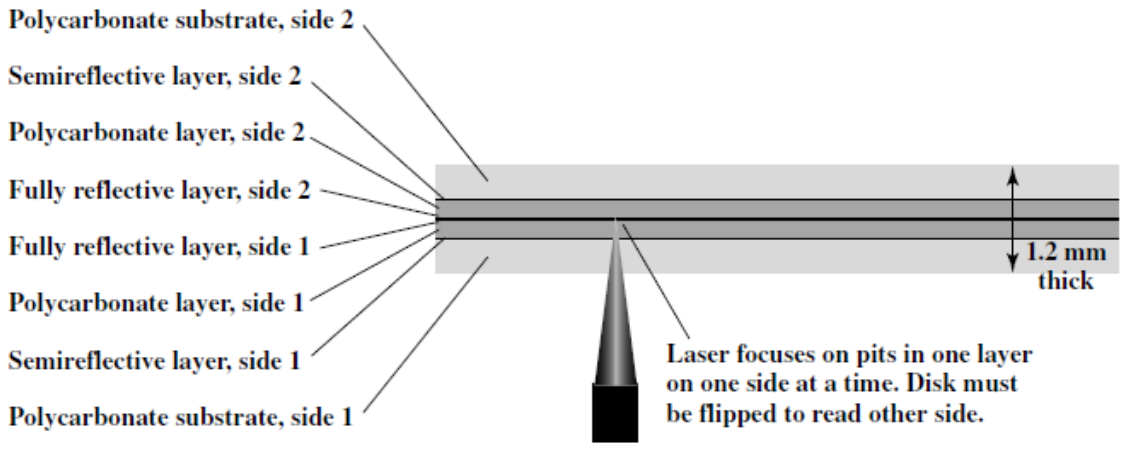
6.4.3.3. Digital Versatile Disk(DVD)

- The DVD's greater capacity is due to three differences than from CD's.

1. Bits are packed more closely on DVD. The spacing between loops of a spiral on CD is 1.64 μm & minimum distance between pits along spiral is 0.834 μm but for DVD loop spacing of 0.74 μm & minimum distance between pits 0.4 μm. capacity is 4.7 GB.
2. The DVD employs a second layer of pits and lands on top of the first layer. It has a semi-reflective layer on top and by adjusting focus the laser drive can read each layer separately. Almost doubles the capacity to 8.5GB.
3. The DVD-ROM can be two sided that could bring total capacity up to 17 GB.



(a) CD-ROM–Capacity 682 MB



(b) DVD-ROM, double-sided, dual-layer–Capacity 17 GB

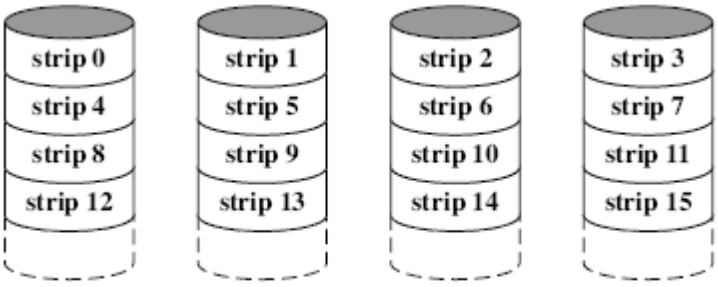
Figure 6.11: CD-ROM and DVD-ROM

6.4.4. RAID (Redundant Array of Independent Disks)

- Disk drive performance has not kept pace with improvements in other parts of the system.
- Limited in many cases by electro-mechanical transport means.
- This lead to the development of array of disks that operate independently and in parallel.
- With multiple disk separate I/O request can be handled in parallel and block of data can be accessed in parallel if distributed across multiple disks.
- With many parallel disk operating as if they are a single unit, redundancy techniques can be used to guard data loss in unit.
- There are 0-6 levels of RAID schemes.

RAID 0

- No redundancy techniques are used.
- Data is distributed over all disk in array.
- Data is divided into strips for actual storage, the divided data strips are stored in separate disks.
- RAID 0 for high data transfer capacity.
  - The strips of data can be accessed in parallel.
- RAID 0 for high I/O request rate.
  - If in case of multiple number of I/O request, if data is divided along the disks then more I/O request can be handled at a time.
- Can support low response time by having the block transfer size equal to strip, i.e. support multiple strip transfer in parallel.



(a) RAID 0 (Nonredundant)

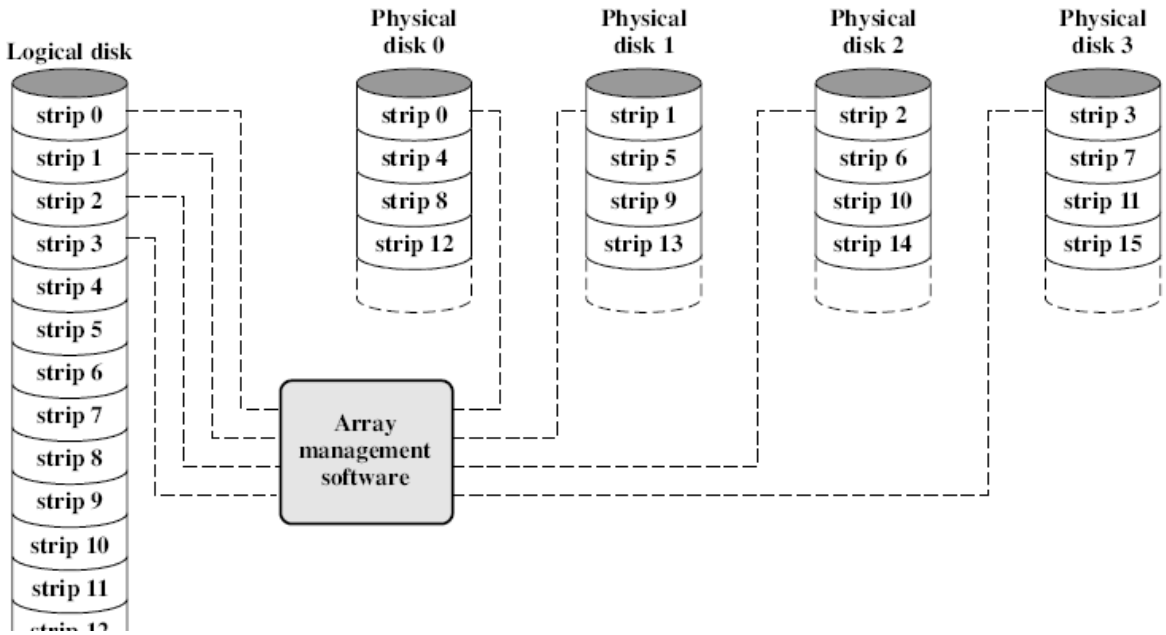
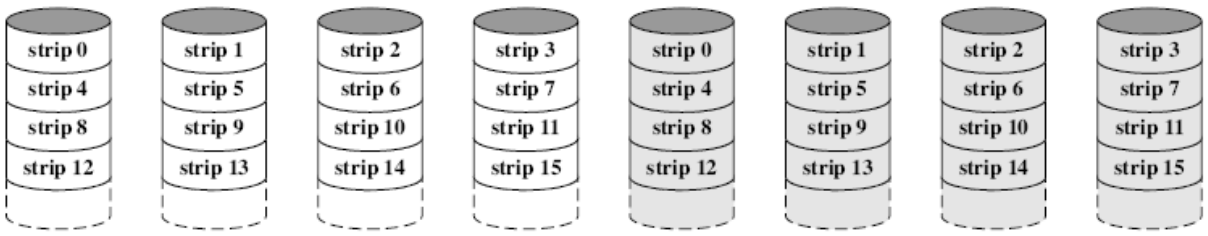


figure 6.12: data mapping for a RAID 0 array

RAID 1

- All disks are mirrored-duplicated, data is stored on a disk and its mirror disks.
- Read from either the disk or its mirror.
- Write must be done to both the data disk and mirror.
- Faulty recovery is easy, and can use data from mirror disk.
- Expensive.
- RAID 1 can achieve high I/O request rate if the bulk of the request are reads (because reading involves read from single disk where as any write operation require write to both disk)

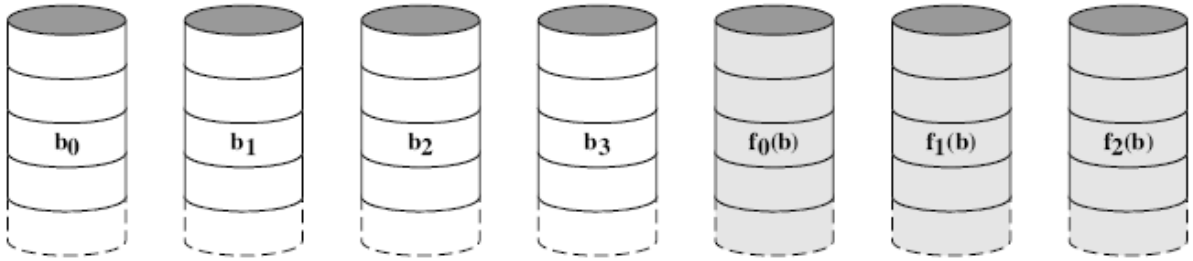


(b) RAID 1 (Mirrored)

RAID 2

- All disks are used for every access. The spindles(disks) are synchronized so that each disk head is in the same position on each disk at any given time.
- Data strips are very small often as small as a single byte or word.
- An error correcting code are stored in the corresponding bit position on multiple parity disk (or additional disk).

- Typically hamming codes is used which is able to correct single bit error and detect double bit errors.
- RAID 2 require fewer disk than RAID 1, but still costlier.
- The number of additional disk is proportional to log of number of data disk.
- on single read, all disks are simultaneously accessed, the requested data and associated error correcting code are delivered to array controller.
- If any single bit error occurs, then controller can recognize and correct the error instantly.
- On single write, all data disk and additional disk must be accessed for write operation.

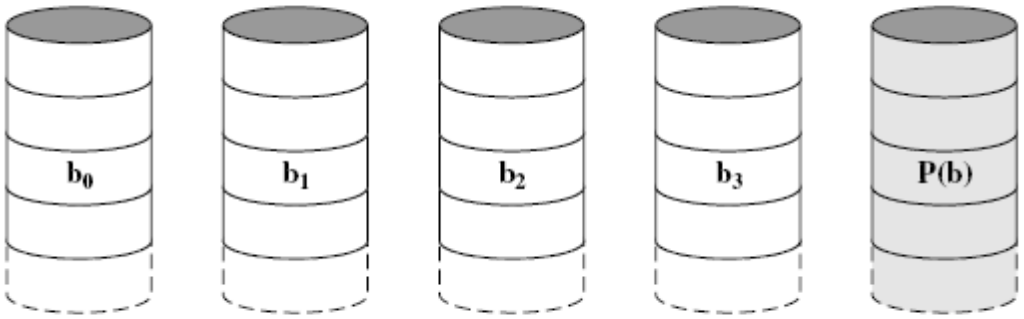


(c) RAID 2 (Redundancy through Hamming code)

RAID 3

any two

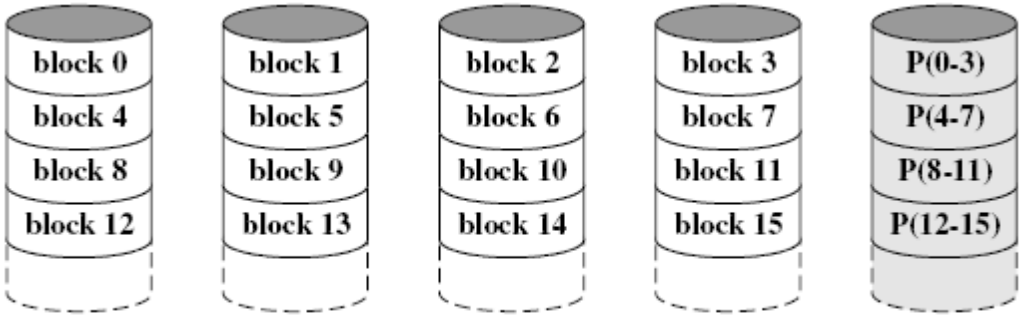
RAID 3 and RAID 4



(d) RAID 3 (Bit-interleaved parity)

- RAID 3 is organized in a similar fashion to RAID 2.
- RAID 3 requires only a single redundant disk.
- RAID 3 employs parallel access with data distributes in small strips.
- A simple parity bit is computed for the set of individual bits in the same position on all data disk.
- Is a drive fails, the parity drive is accessed and data is reconstructed from remaining device, once the failed drive is replaced the missing data can be restored on the new drive and operation resumed.

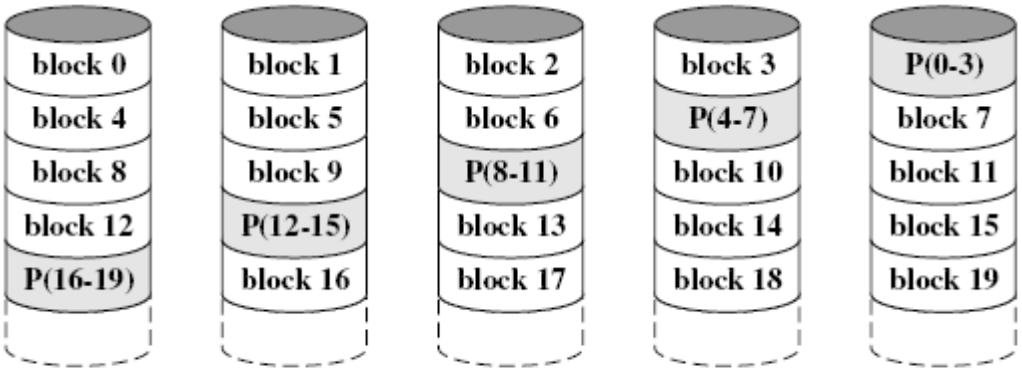
RAID 4



(e) RAID 4 (Block-level parity)

- RAID 4 make use of an independent access techniques, so that separate I/O request can be satisfied in parallel.
- Data striping is used and the strips are relatively large.
- A bit by bit parity strip is calculated across corresponding strips on each data and the parity bits are stored in the corresponding strip on parity disk.
- It involves a write penalty when an I/O Write request of small size is performed, for every write, the parity strip must also be recalculated and written. Thus one write on the disk needs two physical disk access. Thus the parity drive always written to and can cause a bottle neck.

RAID 5

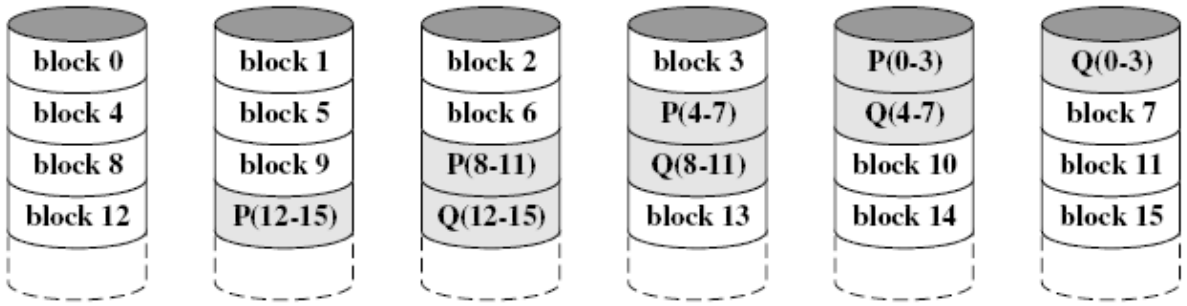


(f) RAID 5 (Block-level distributed parity)

- Organized in similar fashion to RAID 4, the difference is that RAID 5 distributes the parity strips across all disk.
- Strips arranged in round robin scheme.
- The distribution of parity strips across all drives avoids the potential I/O bottle neck found in RAID 4.

**a round robin is an arrangement of choosing all elements in a group equally I some order (circular order), simple way of thinking of round robin is that taking turns. The name of the algorithm comes from round-robin principle.

RAID 6



(g) RAID 6 (Dual redundancy)

- In RAID 6 two different parity calculation are carried out and stored in separate blocks on different disk.
- RAID 6 requires N+2 disk where N is the data disk.
- P & Q are two different check algorithm, one of the two is exclusive or calculation and other is an independent data check algorithm.
- This makes possible to generate data even if two disk fails.
- Advantage extremely high data availability.
- Disadvantage substantial write penalty because each write has written in two parity blocks.

6.4.5. Flash memory

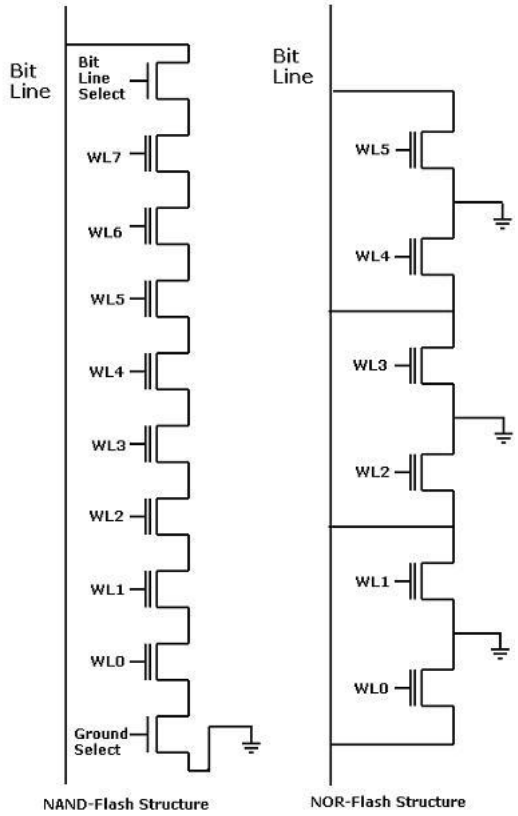
Flash memory technology is a mix of EPROM and EEPROM technologies. The term flash was chosen because if large chunk of memory could be erased at one time. The name therefore distinguishes flash device from EEPROMS where each byte is erased individually.

The more common elementary flash cell consists of one transistor with floating gate similar to an EPROM cell. The electrical functionality of the flash memory cell is similar to that of an EPROM or EEPROM, electrons are trapped onto the floating gate. These electrons modify the threshold voltage od the storage transistor.

Architecture

As other semiconductors the flash memory chip size is the major contributor to the cost of the device. Some of the array structure used are NOR, NAND.

NOR cell: The NOR architecture is currently the most popular flash architecture. It is commonly used in EPROM and EEPROM design aside from active transistor the largest contributor to area in the cell array is the metal to diffusion contact. NOR architecture requires one contact per two cells, which consumes the most of all flash area. The electrons trapping is floating gate is done by hot electrons injection. Electrons are removed by Fowler-Nordheim tunneling.

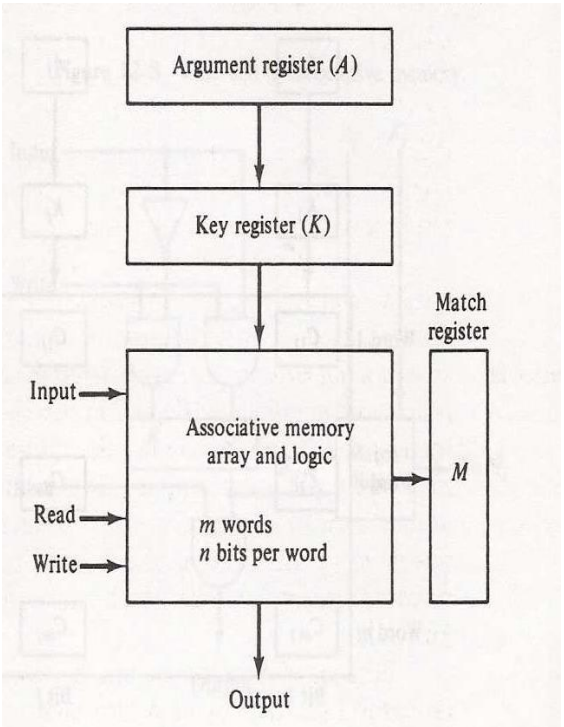


NAND cell

To reduce the cell area, the NAND configuration was developed. The NAND configuration is considerably is that when a cell is read, the sense amplifier sees a weak signal than the on NOR configuration since several transistors in series are used. The weak signal slows down the speed of the read circuitry which can be overcome by operating in series access method. This memory will not be competitive for random access applications.

6.5. Associative memory

Many data processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. The established way to search a table is to store all items where they can be addressed in sequence. The search procedure is strategic for choosing a sequence of address reading the content of memory at each address and comparing the information read with the item being searched until a match occurs. The number of access to memory



depends upon on the location of the item and efficiently of search algorithm.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of data itself rather than by an address. A memory unit accessed by content is called an associative memory or Content Addressable Memory(CAM). This type of memory accessed simultaneously and in parallel on the basis of the data content rather than by specific address or location. When word is written in an associative memory no address is given and word are stored in empty unused location. When a word is to be read from an associative memory, the content of word or part of the word is specified. The memory allocates all words which match the specified content and marks them for reading. Because of its organization the associative memory is uniquely suited to do parallel searches by data association. Searches can be done on entire word or specified word. An associative memory is more expensive than random access memory because each cell must have storage capacity as well as logic circuit for matching its content with external argument. For this reason, associative memory is used in application where the search time is very critical and must be very short.

6.5.1. Hardware organization

The block diagram of an associative memory is shown in figure. It consists of a memory array and logic for m words with n bits per word. The argument registers A and key register k each have n bits one for each bit word. The matching register M has m bits, one for each memory word in memory is compared in parallel with the content of the argument register and **set** the corresponding match register if there is a match. Reading is accomplished by sequential access to memory for those words whose corresponding bits in the match register have been set. The key register provides a mask for choosing a particular field or key in the argument word. The comparison is only done to those bits in the argument that have 1's in their corresponding position of the key register.

A numerical example:

A	101	111100	
K	111	000000	
Word1	100	111110	no match
Word2	101	000001	match

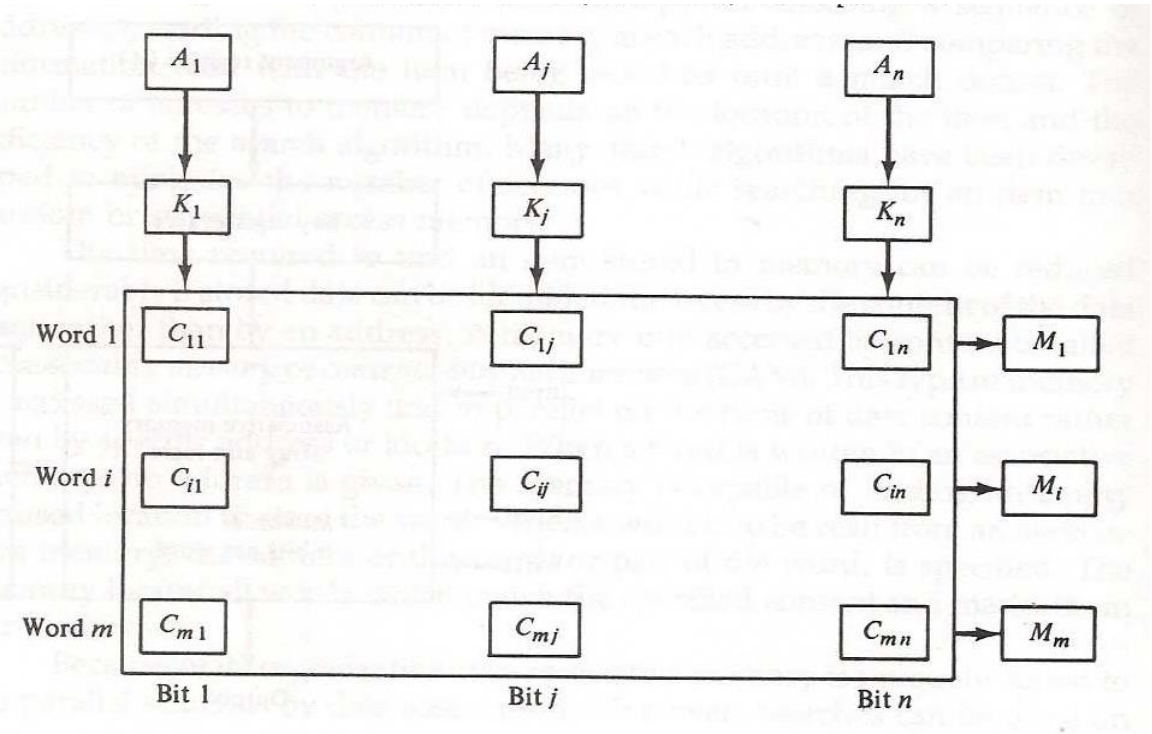


Figure: associative memory of m words, n cells per word

The relation between the memory array and external register in an associative memory is shown above. The cells in the array are marked by letter C with two subscripts the first subscript gives the word number and the second specified the bit position in the word. A_j in the augment register is compared with all bits in column j of the array provided that k_j =1. This is done for all columns j=1,2…n. if a match occurs between all the unmasked bits of the argument and bits in word i, the corresponding bits M_i in the match register is set 1. If one or more unmasked bits if the argument and word do not match M_j is cleared to 0. The internal organization of a typical cell c_{ij} is as shown in figure below. It consists of flip-flops storage elements F_{ij} and circuits for writing reading and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during read operation. The match logic compares the content of the storage cells with the corresponding unmasked bit if the argument and provides an output for the decision logic that sets the bits in M.

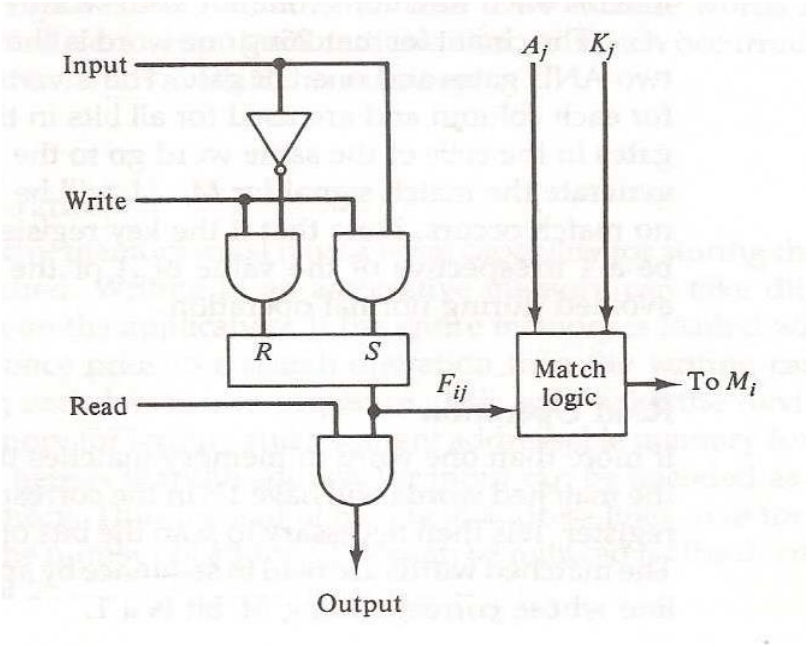


Figure: One cell of associative memory

6.5.2. Match logic

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First we neglect the key bits and compare the argument in A with the bits stored in the cells of words. Word i is equal to the argument in A if $A_j=F_{ij}$ for j=1,2 …n. two bits are equal if they are both 1 0r 0. The equality of two bits can be expressed logically by the Boolean function.

$$x_i = A_j F_{ij} + A'_j F'_{ij}$$

Where $x_i=1$, if the pair of bits in position j are equal.

The condition for setting the corresponding match bits M_i to 1. The Boolean function for this condition is $M_i = x_1 x_2 x_3 \dots x_n$

And constitute the AND operations of all pairs of matched bits in a word.

Now for key bits, the requirement is that if $k_j = 0$, the corresponding bits of A_j and F_{ij} needs no comparison, only when $k_j = 1$, must they be compared. This requirement is achieved by Oring each item with k'_j

Thus,

$$x_j + k'_j = \begin{cases} x_j & if\ k_j = 1 \\ 1 & if\ k_j = 0 \end{cases}$$

A term $(x_j + k'_j)$ will be in the 1 state if it's pairs of bits is not compared. This is necessary because each termed is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when $k_j = 1$.

The match logic for word i in an associative memory can now be expressed by the following Boolean expressing.

$$M_i = (x_1 + K'_1)(x_2 + K'_2) \dots (x_n + K'_n)$$

Each term in the expression will be equal to 1, if the corresponding k_j =0, if k_j =1, the term will be either 0 or 1 depending on the value of x_j. A match will occur and M_j will be 1, if all terms are equal to 1.

$M_j = \prod_{j=1}^{n} = (A_j F_{ij} + A'_j F'_{ij} + k'_j)$

6.5.3. Read Write logic

Read operations

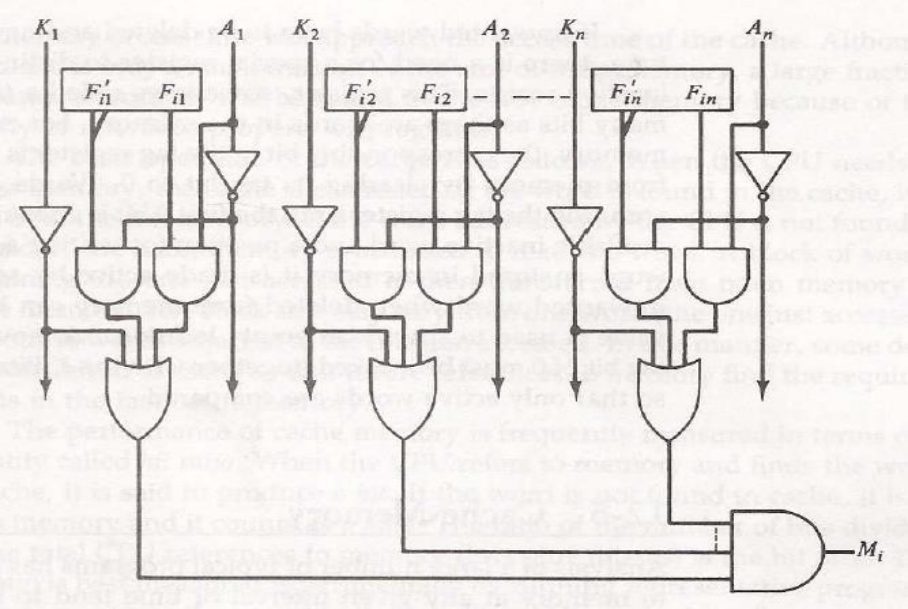


Figure: Match logic for one word of associative memory

If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the match register. It is then necessary to scan the bits of the match register one at a time. The matched word is read in sequence by applying a read signal to each word line whose M_j bit is 1.

In most application the associative memory stores a table with no two identical items under a given key. In this case only one word may match the unmasked argument field. By connecting output M_j directly to the read line in the same word position, the content of the match word will be presented automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude words having zero content, an all zero output will indicate the no match occurred and that the searched item is not available in memory.

Note that if the key register contains all 0's and outputs M_j will be 1, irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

Write operation

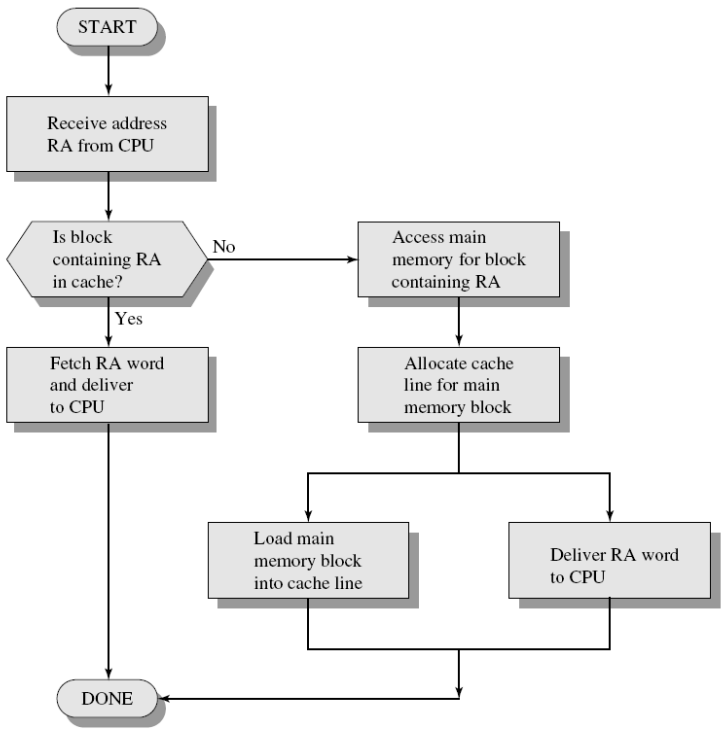
Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded with new information at once prior to a search operation, then the writing can be done by addressing each location in sequence. This will make the device random access memory for reading. The advantage here is the address for input can be decoded as in random access memory. Thus instead of having m address line one for each word in memory, the number of address line can be reduced by the decoder to d lines where $m = 2^d$

If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register sometimes called tag register. For active word stored corresponding tag register is 1, a word is deleted by clearing the tag bit to 0. After the new word is stored in memory in memory it is made active by setting its tag bit to 1.

6.6. Cache memory

A cache memory is a small very fast memory that retains copies of recently used information from main memory. It operates transparent to the programmer automatically decoding which values to be kept and which to be overwrite.

The processor operates at high clock rates but the memory is slow so it requires cache. The overall system performance depends on the successful access from cache. An access to an item which is in cache is called a *hit* and an access to an item which is not in the cache is called a *miss*.



imp

Figure: shows the flow chart of read operation in cache

Typical cache organization

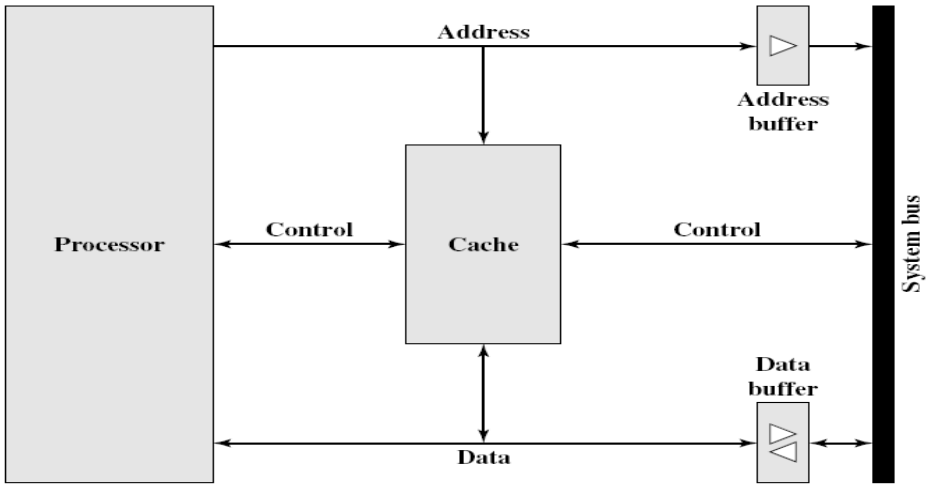


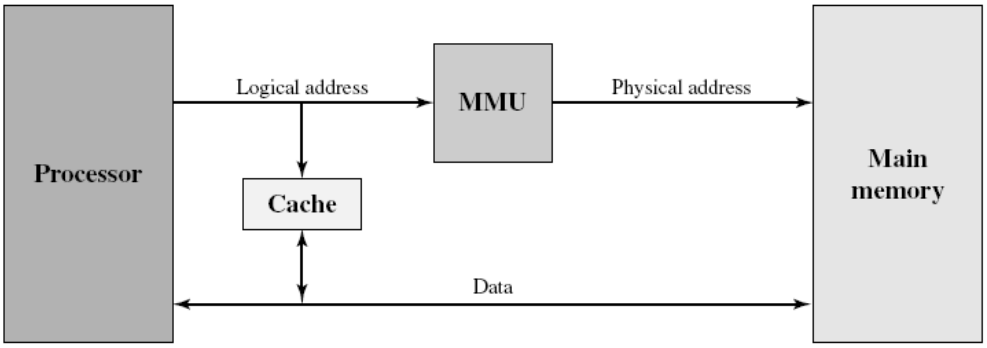
Figure: Typical cache organization

Figure shows the cache organization. In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from which main memory is reached. When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor.

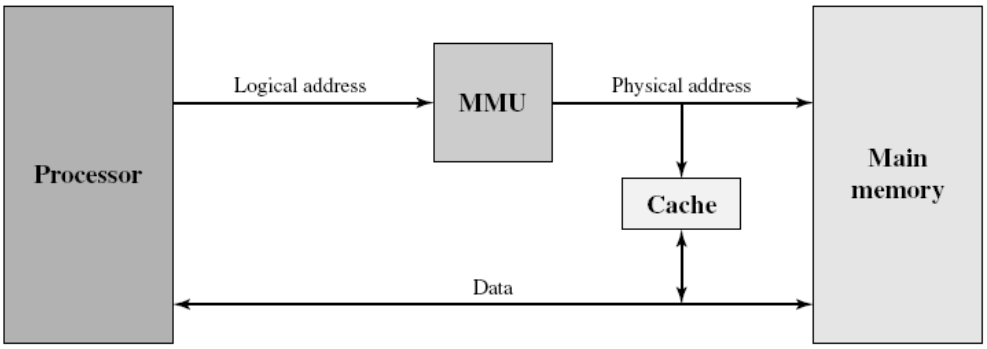
6.6.1. Elements of cache design:

a. Cache address:

A virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads to and writes from main memory it needs a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory. A **logical cache**, also known as a **virtual cache**, stores data using **virtual addresses**. The processor accesses the cache directly, without going through the MMU. A physical cache stores data using main memory **physical addresses** as shown in figure for logical address the cache is inserted in between the processor and MMU and foe physical address between MMU and main memory.



(a) Logical cache



(b) Physical cache

One obvious advantage of the logical cache is that cache access speed is faster than for a physical cache, because the cache can respond before the MMU performs an address translation. The disadvantage has to do with the fact that a single virtual address can be used for many spaces (physical address) for different application. So either it needs to completely flush the cache memory or add extra bits to identify the exact location.

b. Cache size:

For faster operation of the system we want higher cache size but as the size increases the cost per bit also increases, so there is a tradeoff between the size and speed. Hence is the designers wish how much cache to be used. Following table shows the different cache sizes used for different systems.

Processor	Type	Year of Introduction	L1 Cache[a]	L2 Cache	L3 Cache
IBM 360/85	Mainframe	1968	16 to 32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128 to 256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256 to 512 KB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 KB to 1 MB	2 MB
IBM S/390 G4	Mainframe	1997	32 kB	256 KB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 KB	4 MB
SGI Origin 2001	High-end server	2001	32 kB/32 kB	4 MB	—
Itanium 2	PC/server	2002	32 kB	256 KB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1 MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24–48 MB

imp

c. Mapping function:

For fast memory access there should be no time lost in searching for words in the cache for this mapping is used. The transformation of data from main memory to cache memory is referred to as mapping process. There are three types of mapping:

1. Associate mapping
2. Direct mapping
3. Set-Associative mapping

To explain the concept of mapping first consider a specific example of a memory organization the main memory can store 32k words of 12 bits each. The cache is capable of storing 512 of these words at any given time. for every word there is a duplicate copy on main memory. The CPU first sends 15-bit address to cache if there is a hit, the CPU accepts 12-bit data from cache if there is a miss the CPU read the word from main memory and the word is then transferred to cache.

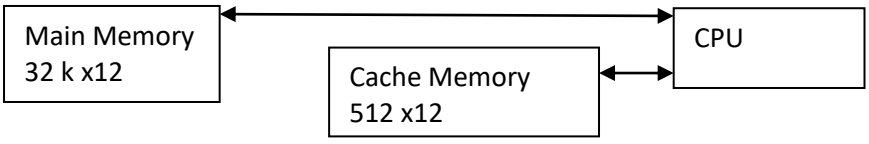


Figure: example of cache memory

1. Associative mapping:

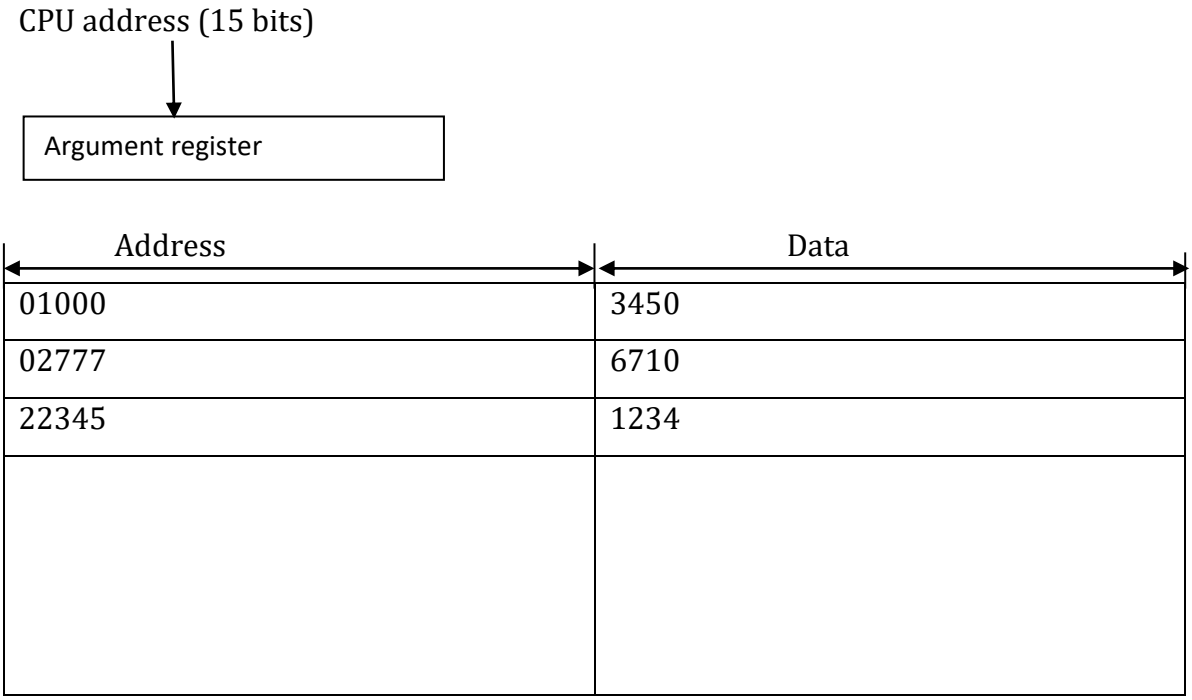


Figure: associative mapping

- The fastest and most flexible cache organization.
- Stores both address and data of memory word.
- The address value of 15 bits is shown as a five-digit octal number and its 12-bit word is shown as a four-digit octal number.
- A CPU address of 15 bits is placed in the argumented register and the associative memory is searched for matching address, if address is found the corresponding 12 bit is read and sent to CPU.
- If no match occurs the main memory is accessed for word also the address data pair must be placed in cache.
- If cache is full then what value to be displaced is determined by replacement algorithms

2. Direct mapping:

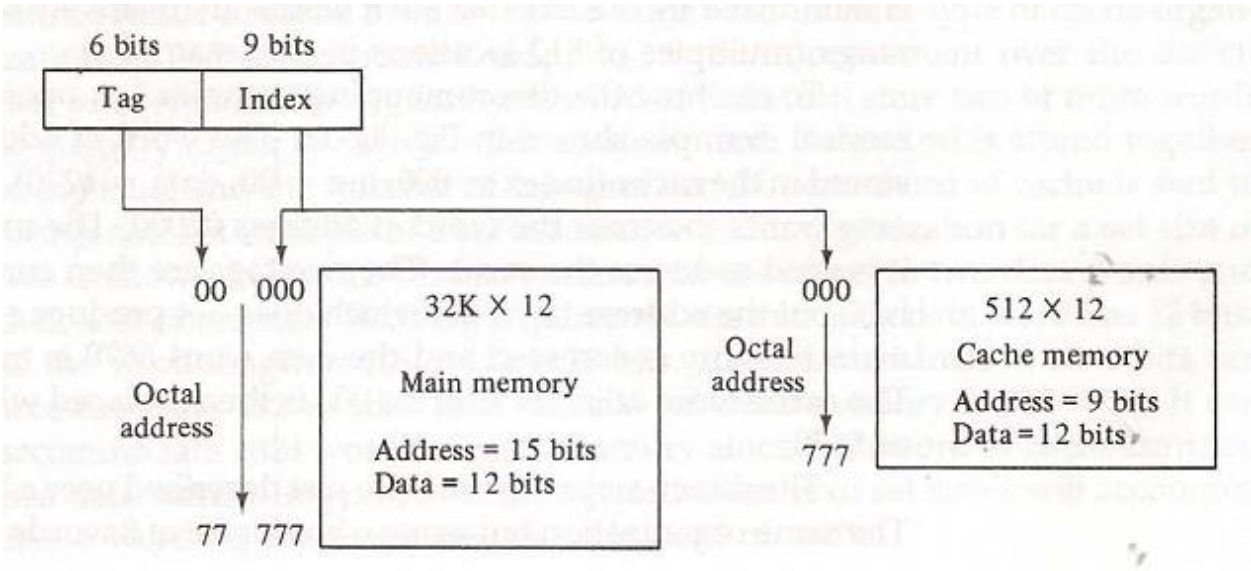


Figure: address relationship between main memory and cache

- The CPU address is divided into two parts:
  a. Index field
  b. Tag field

  Here in example 9 bits as index and 6 bits as tag.
- The number of bits in the index field is equal to the number of address bits required to access cache memory.
- When CPU generates a memory request the index field is used as the address to access the cache.
- Then tag field in of CPU is compared with the tag in the word read from the cache, if tag match then there is a hit and desired data word is in cache. If no match, then miss and required word is read from main memory.
- It also stores the value and tag in cache or replaces the old ones.

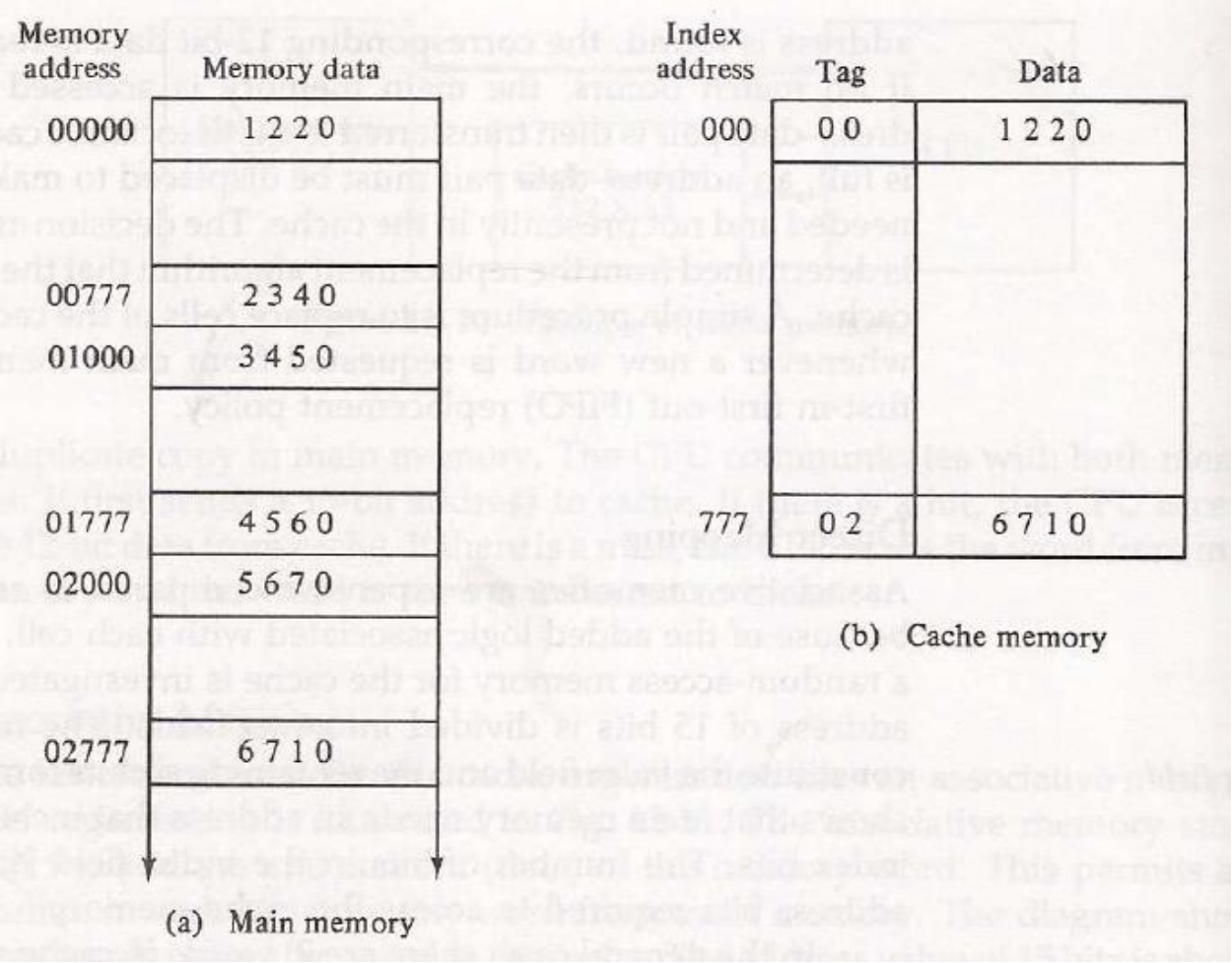


Figure: direct mapping cache organization

Disadvantage:

If hit rate drops considerably if two or more words whose address have the same index but different tags are accessed repeatedly.

** see direct mapping example of how direct mapping done using numerical value from Morris mano book.

3. Set- Associate Mapping
    - The disadvantage of direct mapping is it cannot store two words of same index with different tags.
    - In set associate mapping two or more words of memory under same index address with different tags can be stored.
    - Each data word is stored together with its tag and the number of tag-data item in one word of cache is said to form a set.
    - For searching, CPU sends address first it searches for matched index in cache, after matching index it then matches the tag. For matched tag data is sent to CPU, if no match is found it is searched from main memory

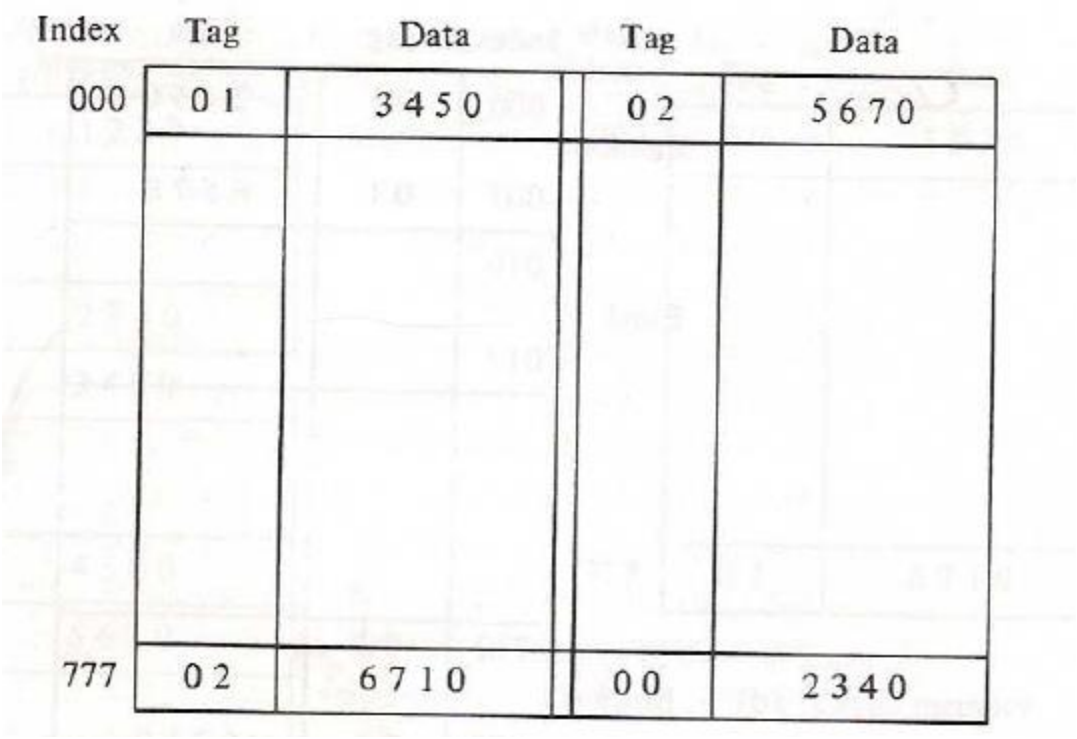


Figure: two way set associative mapping cache.

- The hit ratio will improve as the set size increases because more words with same index but different tags can reside in cache.

6.6.2. Replacement algorithms

The most common replacement algorithms are:

- Random replacement: replacements are done at random.
- First-In-First –Out(FIFO): replaces the item that has been in the set the longest.
- Least Recently Used(LRU): replaces the item that have been least recently used.

Writing into cache:

During read operation the main memory is not involved in transfer. For write operation it can be done in two ways:

a. Write through: in this to update main memory with every memory write operation, the cache memory also is updated in parallel. Advantage of this is that main memory always contains the same data(valid) as cache.
b. Write back: in this only the cache location is updated during write operation. The location is then marked by a flag so that when the word is removed from cache it is copied to main memory. Advantage is that during the time the word resides in the cache it may be updated number of times and will be used from cache only so it needs to be updated in main memory only if it is being removed from cache.

Cache initialization

The cache is initialized when the power is applied to the computer or when the main memory is loaded with a complete set of programs. After initialization the cache is considered to be empty, but in fact it contains some non-valid data. it is customary to include each word with valid bit to indicate whether the word is valid or not.

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache is set to 1 the first time this word is loaded from main memory and stays this way unless it is re initialized. The use of valid bit is that the replacement of data in cache is done for only those data whose valid bit is 0.

*** the cache used can be of single level or multiple levels depending upon the design and performance required.

6.7. Memory Management Hardware

In multiprogramming environment where many programs reside in memory it becomes necessary to move programs and data around the memory to vary the amount of memory in use by a given program and to prevent program from changing other programs. A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory.

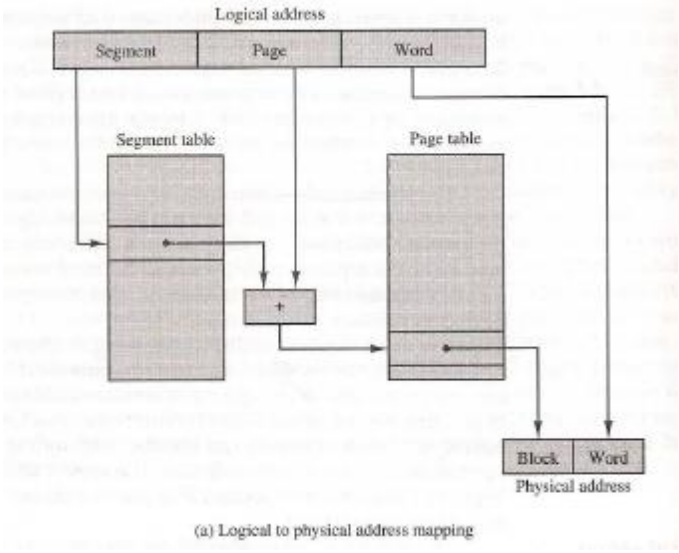
The basic components of a memory management unit are:

1. A facility for dynamic storage location that maps logical memory references into physical memory address.
2. A provision for sharing common programs stored in memory by different users.
3. Protection of information against unauthorized access between users and preventing users from changing operating system function.

    1. The dynamic storage relocation hardware is mapping process similar to paging system. It is more convenient to divide programs and data into logical parts called segments. A segments is a set of logically related instruction or data elements associated with a given name. segments may be generated by programmer or by operating system.

       The address generated by a segmented program is called logical address. Logical address space is associated with variable length segments rather than fixed length pages. The logical address may be equal or differ than actual physical address. Shared programs are placed in a unique segment in each user's logical address space so that a single physical copy can be shared.

Segmented page mapping



(a) Logical to physical address mapping

Consider the logical address as shown in figure. The logical address is partitioned into three fields. The segment field specifies a segment number, the page field specifies the page within the segment and word field gives the specific word within the page.

The mapping of the logical address into a physical address is done by means of two tables a & b. the segment number of the logical address specifies the address for the segment table. The entry in the segment table is pointer for page table base. The page table base is added to the page number given in logical address. The sum produces a pointer address to an entry in the page table. The value found in the page table provides the block number in physical memory.

In this process memory reference from CPU will require three access to memory. One from segment table, one from page table and third from main memory. This will slow the system significantly. Alternative approach will be to use Translation Lookaside Buffer(LTB). The first time a given block is referenced its value together with the corresponding segment and page number are entered into associative memory. Thus mapping process is first attempted by associative search with given segment and page number.

** for numerical example see Morris mano book.

2. The sharing of common programs is an integral part of a multiprogramming system. For example, several users may wish to use compiler that is shared between them. Other systems programs residing in the memory are also shared by all users in a multiprogramming system without having to produce multiple copies, this type of management is done by memory management hardware.

3. Memory protection

Memory protection can be assigned to the physical address or logical address. The protection of memory through the physical address can be done by assigning to each block in memory a number of protection bits. Every time a page is moved from one block to another it would be necessary update the block protection bits. A much better place to apply protection is in logical address space rather than the physical address space. This can be done by including protection information within the segment table or segment register of memory management hardware.

The content of each entry in the segment table or a segment register is called descriptor. A typical descriptor would contain in addition to base address field one or two additional field for protection purpose.

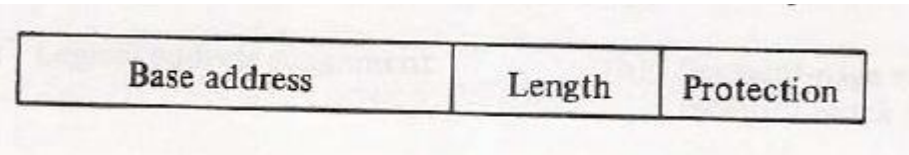


Figure: format of typical segment descriptor

The base address field give the base of the page table address in a segmented page organization or the block base address in segmented register organization. This address is used in mapping from logical to physical address. The length field give the segment size by specifying the maximum number of pages assigned to the segment. A size violation occurs if the page number falls outside the segment length boundary. Thus a given program and its data cannot access memory not assigned to it by the operating system.

The protection field in a segment descriptor specifies the access right available to the particular segment. Some of the access rights that are used for protecting the programs residing in memory are:

1. Full read & write privileges: given to a program when it is executing its own instructions.
2. Write protection: useful for sharing system programs such as utility programs & other library routines.
3. Execution only: protects programs from being copied.
4. System only: occasional user from accessing operation system segments.

Chapter 7: Input/output organization

An important key element of any computer system is I/O (Input Output) module. Different modules interface with system bus or central switch, or different peripheral devices connect to processor through I/O module. So an I/O module is not just a simple mechanical connector that connects a device to the computer but rather contains logic for performing a communication functions between the peripheral and bus.

Some of the reason why we cannot directly connect peripheral devices to the system bus:

1. There are a wide variety of peripherals with different methods of operations, so it would be practically impossible to define methods for every peripheral device within a single processor.
2. The data transfer rate some of the peripheral are slower and some faster than that of memory or processor, so we cannot directly connect to system bus due to this speed mismatch.
3. Peripherals often use different data formats and word length than the computer they are attached to.

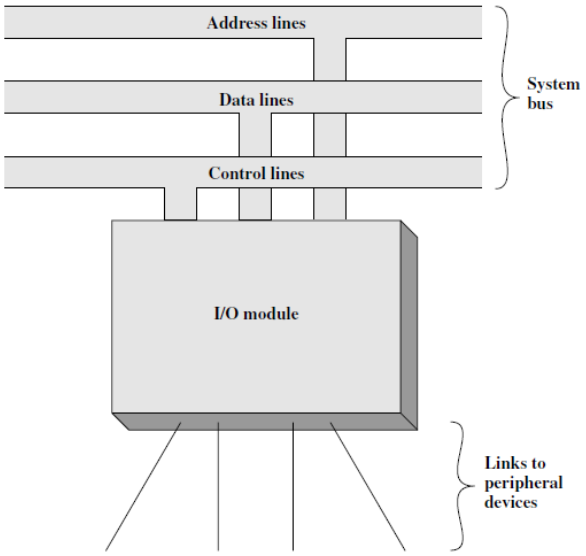


Figure: generic model of I/O devices

4.1 External devices

External devices are those devices that provide a way of communicating medium between external environment and the computer. An external device is attached to the computer through a link to an I/O module. The link is used to exchange control, status, and data between I/O module and external devices. We can broadly classify external devices into three categories as:

Human readable: those used for communicating with computer user, example: video display terminal and printers.

Machine readable: those used for communicating with equipment. Example magnetic tapes, magnetic disks.

Communications: those used for communicating with remote devices, i.e. devices that allows to exchange of data with remote devices.

In general terms the nature of external devices is indicated in the figure. The interface module to the I/O module is in the form of control, data and status signal.

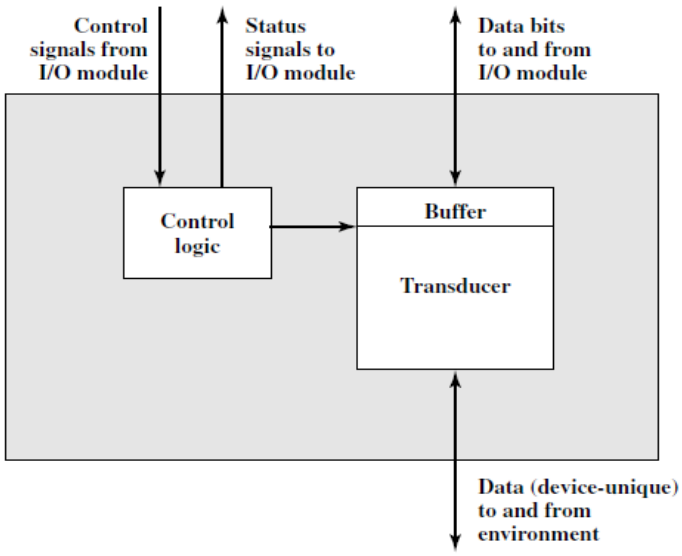


Figure: Block diagram of external devices

Control signal determines the function that the device will perform such as send data to the I/O module, accept data from I/O module, report status or perform some control functions particular to the device.

Data are transferred in the form of bits to be sent or received from I/O module.

Status signals indicate the state of the devices like device ready or not.

Control logic associated with the device controls the device's operation in response to the direction from I/O module. The transducer converts data from electrical to other forms of energy during output and from other form to electrical during input. A buffer is associated with the transducer to temporarily hold data being transferred.

Example of external devices

The most common means of computer/user interaction is a keyboard/monitor arrangement. The user provides input through the keyboard. This input is then transmitted to the computer and may also be displayed on the monitor. The basic unit of exchange is the character. Associated with each character is a code, typically 7 or 8 bits in length. The most commonly used text code is the International Reference Alphabet (IRA) or American Standard Code for Information Interchange (ASCII). Each character in this code is represented by a unique 7-bit binary code; thus, 128 different characters can be represented. Characters are of two types: *printable* and *control*. Printable characters are the alphabetic, numeric, and special characters

that can be printed on paper or displayed on a screen. Some of the control characters have to do with controlling the printing or displaying of characters. For keyboard input, when the user depresses a key, this generates an electronic signal that is interpreted by the transducer in the keyboard and translated into the bit pattern of the corresponding IRA code. This bit pattern is then transmitted to the I/O module in the computer. At the computer, the text can be stored in the same IRA code. On output, IRA code characters are transmitted to an external device from the I/O module. The transducer at the device interprets this code and sends the required electronic signals to the output device either to display the indicated character or perform the requested control function.

Disk Drive

A disk drive contains electronics for exchanging data, control, and status signals with an I/O module plus the electronics for controlling the disk read/write mechanism. In a fixed-head disk, the transducer is capable of converting between the magnetic patterns on the moving disk surface and bits in the device's buffer.

4.2 I/O Modules

The major functions or requirements for an I/O module fall into the following categories:

- Control and timing
- Processor communication
- Device communication
- Data buffering
- Error detection

The internal resources, such as main memory and the system bus, must be shared among a number of activities, including data I/O. Thus, the I/O function includes a *control and timing* requirement, to coordinate the flow of traffic between internal resources and external devices.

Processor communication involves the communication between the processor and I/O module. It involves the following:

Command decoding: The I/O module accepts the command from processor sent as signals on control bus and decodes it.

Data: Data are exchanged between the processor and the I/O module over the data bus.

Status reporting: It is important to know the status of the I/O module because of the speed mismatch between the processor and I/O module. E.g. BUSY and READY status signals.

Address recognition: I/O module must recognize the unique address of the I/O peripherals that it controls.

The I/O module must be able to perform *device communication*. This communication involves commands, status information, and data. The transfer rate into and out of main memory or the processor is quite high so we need a *data buffering* to match the different data rates of the

peripheral devices. An I/O module is often responsible for error detection and subsequently reporting errors to the processor, example of errors includes mechanical and electrical malfunctions reported by the device like, paper jam, bad disk track.

4.2.1 I/O Module structure

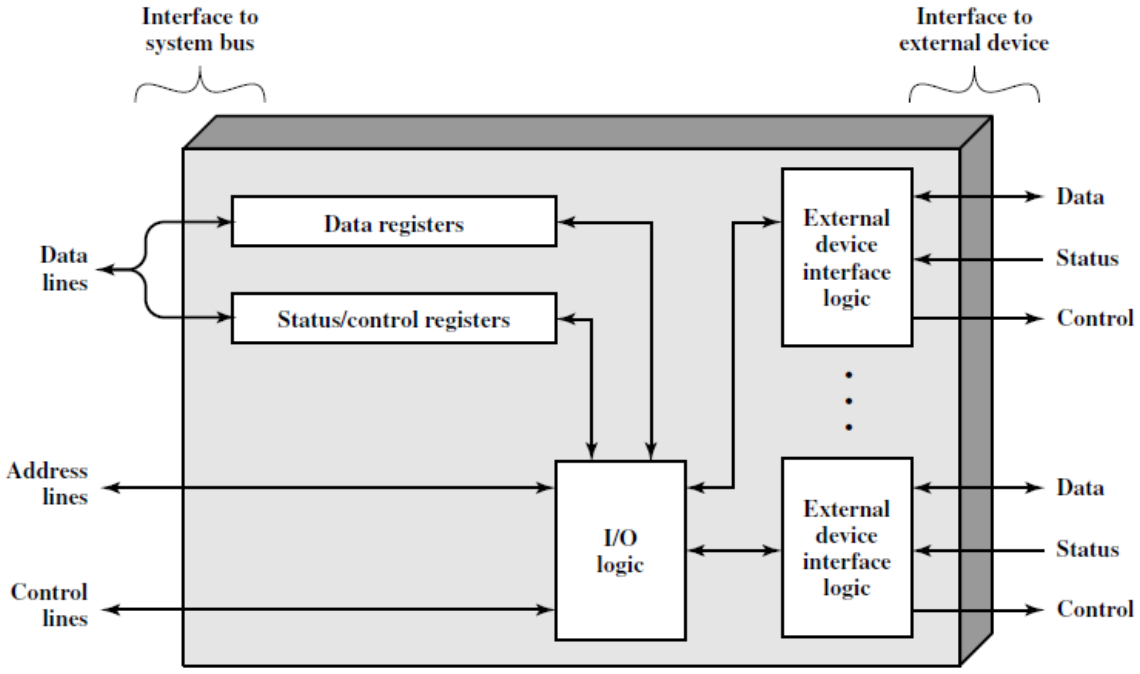


Figure: Block diagram of an I/O module

I/O modules vary considerably in complexity and the number of external devices they control. The module is connected to computer through a *system bus*. Data transfer to and from the module are buffered in one or more *data registers*. There may be one or more *status registers* that provide current status information. A status register may also function as a *control register* to accept detailed control information from processor. The logic within the module interacts with the processor via a set of control lines to issue commands to I/O module.

Each I/O module has a unique address or if it controls more than one external device a unique set of address. I/O module contains specific logic to interface with each device that it controls. I/O module may hide details of timing, formats and mechanics of external device so that processor can function in terms of simple read and write commands.

4.3 I/O techniques

Three techniques are possible for I/O operations:

1. Programmed I/O
2. Interrupt Driven I/O
3. Direct Memory access(DMA)

1. Programmed I/O

Programmed I/O is result of I/O instructions written in computer program. Each data item transfer is initiated by an instruction in the program. Usually the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer data to and from CPU to memory. Transferring the data under program control requires constant monitoring of the peripheral by CPU, but in programmed I/O technique once the data transfer is initiated, the CPU is required to monitor the interface to see when again a transfer can be made.

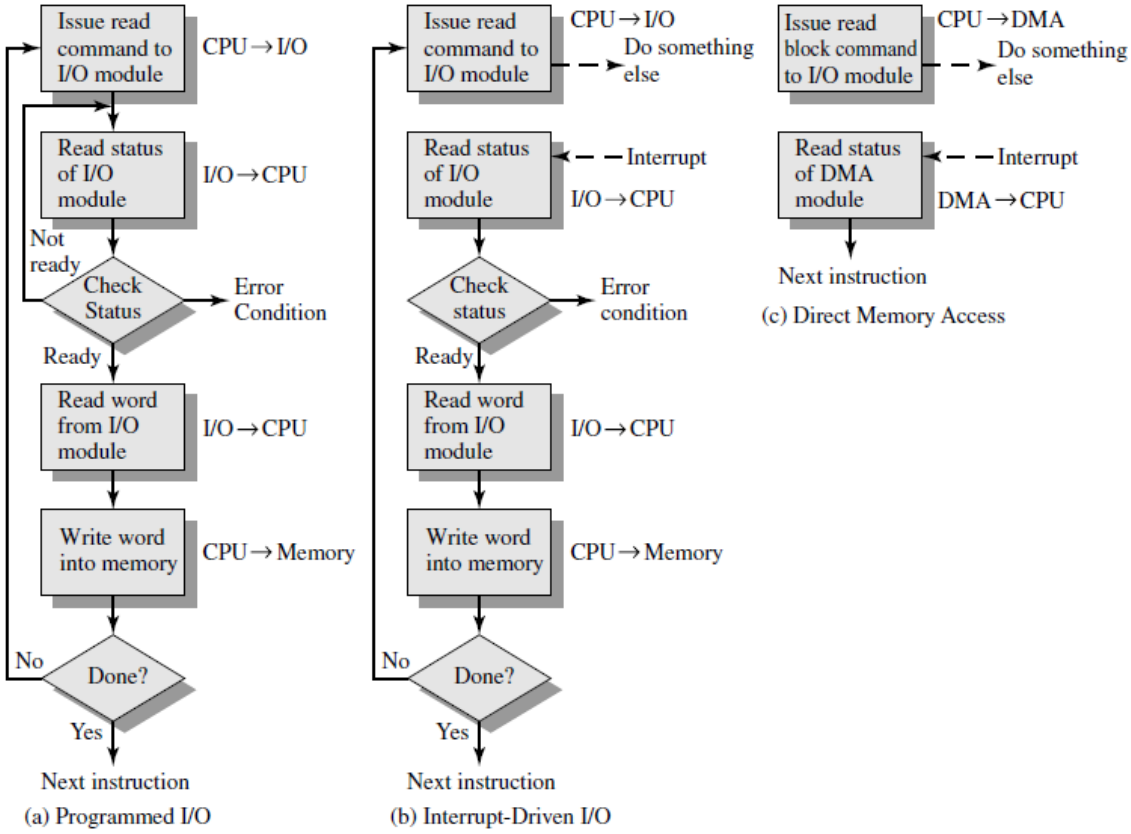


Figure: Flow chart of three techniques

Steps followed in the programmed I/O techniques

- CPU issues the read command to I/O module.
- Read status of the I/O module.
- Check the status of I/O module. If the module is not ready then wait some time and again check for the status, if error condition occurs then run error handling subroutine, if the module is ready then read word from I/O module.
- Then if needed write word into memory.
- Check if all tasks are completed if yes then go to next instruction else start from step 2.

Here we have to analyze the programmed I/O techniques from point of view of I/O commands issued by the processor to the module and then from the point of view of the instruction executed by the processor.

To execute an I/O related instruction the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module may receive when it is addressed by a processor:

Control: Used to activate a peripheral and tell it what to do.

Test: Used to test various status conditions associated with an I/O module and its peripherals.

Read: Causes the I/O module to obtain an item of data from the peripheral and place it in an internal buffer

Write: Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit that data item to the peripheral.

There will be many I/O devices connected through I/O module to the system, each device is given a unique identifier or address. Thus each I/O module must interpret the address line. When the processor, main memory and I/O share a common bus two modes of addressing are possible:

Memory mapped I/O:

Processor treats the status and data register of I/O module as memory locations and use same machine instruction to access both memory and I/O devices. There is single address space for memory location and I/O devices.

Isolated I/O:

Address space of I/O module is isolated from the memory address space. (The bus may be equipped with memory read and memory write plus input and output command lines. Now the command line specifies whether the address refers to a memory location or I/O devices). Separate instructions in the instruction set are used to perform I/O.

2. Interrupt Driven I/O

The problem with programmed I/O technique is that the processor has to wait a long time for I/O module to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded. And alternative way could be:

- Issue an I/O command to the module by the CPU.
- CPU continues with its other task while the module performs its task.
- Module signals the CPU when I/O operations is finished (i.e. generate interrupt) when it is ready to exchange data.
- CPU responds to the interrupt by executing an interrupt service routine and then continues on with it's with its primary task after interrupt service routine completion.

Advantage of this technique is that CPU involvement is less than programmed I/O technique. CPU recognizes and responds to interrupt at the end of an instruction cycle.

3. Direct memory access (DMA)

Both the programmed and interrupt driven I/O require the continued involvement of the CPU in ongoing I/O operations. Direct Memory Access takes the CPU out of the task except for initialization of the operations. Using this techniques large amount of data can be transferred between memory and peripheral without severely impacting CPU performance.

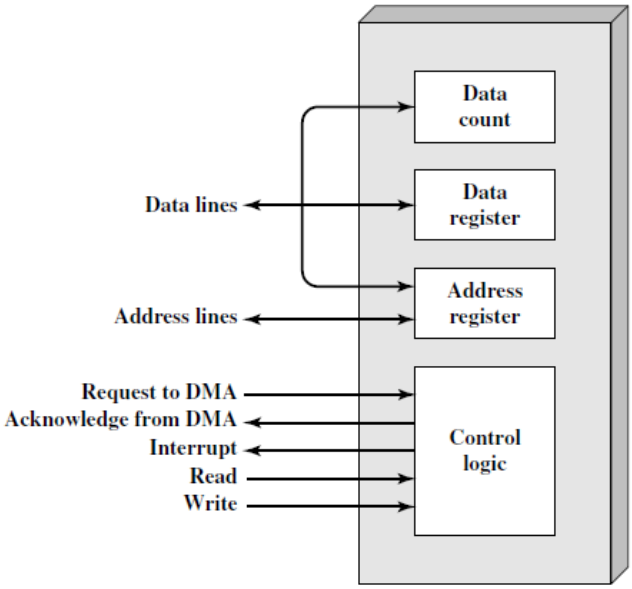


Figure: DMA block diagram

When the processor whishes to read or write a block of data, it issues a command to the DMA module by sending DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and DMA module.
- The address of the I/O device involved communicated through data lines.
- The starting location in memory to read from or write to communicated on the data lines and stored by the DMA module in its register.
- The number of words to be read or written again communicated via the data lines and stored in the data count register.

The processor then continues with other work. The DMA module transfer the entire block of data one word at a time directly to or from memory, without going through the processor. When the transfer is complete the DMA module sends an interrupt signal to the processor thus the processor is involved only at the beginning and end of the transfer.

DMA module must use the bus only when the processor does not need it or it must force the processor to suspend operation temporarily. The later technique is more commonly referred to as *cycle stealing* because the DMA module in effect steals a bus cycle.

4.4 I/O channels and processor

With the evolution of computer we have seen different techniques used in the computer to enhance the performance. Considering the I/O part we can see the following evolutionary steps:

1. The CPU directly controls peripheral devices, earlier stages of computer system.
2. A controller or I/O module is added so that CPU uses programmed I/O without interrupts.
3. The same configuration as in step 2 is used, but now interrupts are employed. So that CPU need not spend time waiting for an I/O operation to be performed, thus increasing efficiency.
4. The I/O module is given direct access to memory via DMA. It can now move a block of data to or from memory without involving the CPU, except at the beginning and end of the transfer.
5. The I/O module is enhanced to become a processor in its own right, with a specialized instruction set tailored for I/O. The CPU directs the I/O processor to execute an I/O program in memory. The I/O processor fetches and executes these instructions without CPU intervention. This allows the CPU to specify a sequence of I/O activities and to be interrupted only when the entire sequence has been performed. For this method I/O module is referred to as an I/O channel.
6. The I/O module has a local memory of its own and is, in fact, a computer in its own right. With this architecture, a large set of I/O devices can be controlled, with minimal CPU involvement. A common use for such architecture has been to control communication with interactive terminals. The I/O processor takes care of most of the tasks involved in controlling the terminals.

As we proceeds along this evolutionary path, more and more of the I/O function is performed without CPU involvement. The CPU is increasingly relieved of I/O-related tasks, improving performance. With the last two steps (5–6), a major change occurs with the introduction of the concept of an I/O module capable of executing a program. For step 5, the I/O module is often referred to as an I/O channel. For step 6, the term I/O processor is often used.

Characteristics of I/O Channels

An I/O channel has the ability to execute I/O instructions, which gives it complete control over I/O operations. In a computer system with such devices, the CPU does not execute I/O instructions. Such instructions are stored in main memory to be executed by a special-purpose processor in the I/O channel itself. Thus, the CPU initiates an I/O transfer by instructing the I/O channel to execute a program in memory. The program will specify the device or devices, the area or areas of memory for storage, priority, and actions to be taken for certain error conditions. The I/O channel follows these instructions and controls the data transfer.

Two types of I/O channels are common, as illustrated in Figure below. A selector channel controls multiple high-speed devices and, at any one time, is dedicated to the transfer of data with one of those devices. Thus, the I/O channel selects one device and effects the data transfer. Each device, or a small set of devices, is handled by a controller, or I/O module, that is much like the I/O modules we have been discussing. Thus, the I/O channel serves in place of the CPU in controlling these I/O controllers. A multiplexor channel can handle I/O with multiple devices at

the same time. For low-speed devices, a byte multiplexor accepts or transmits characters as fast as possible to multiple devices.

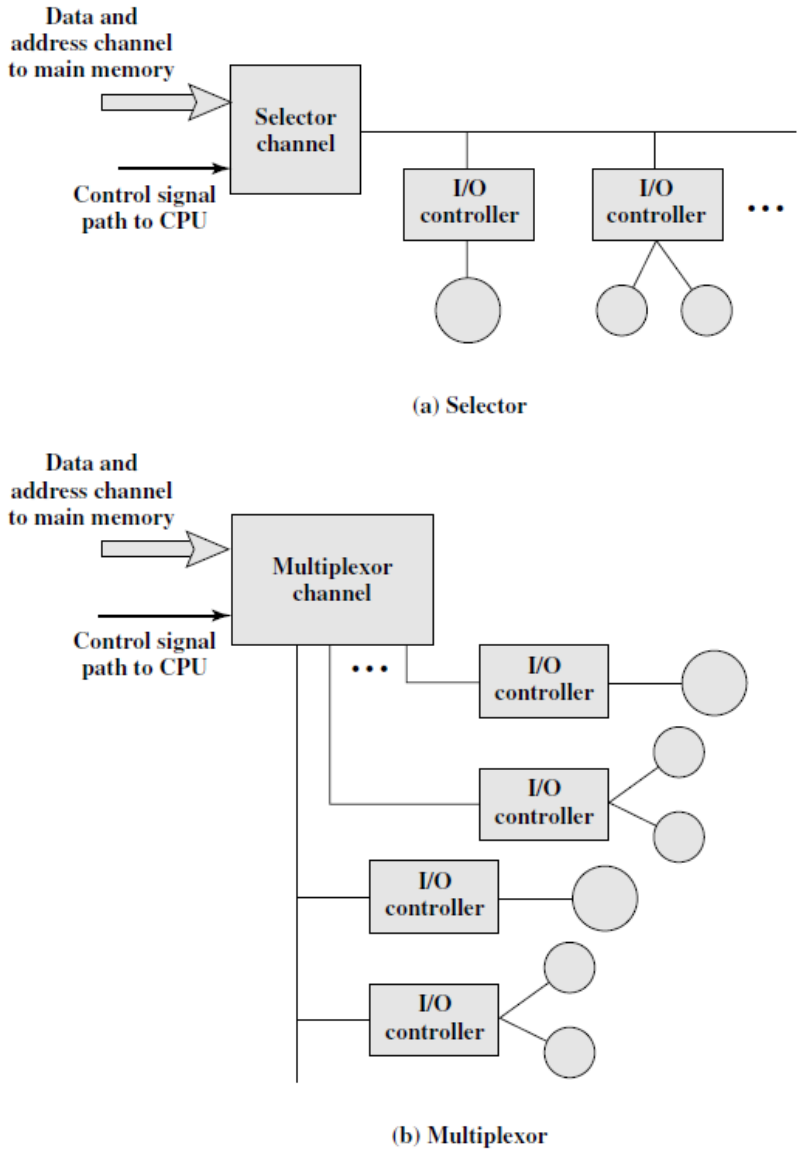


Figure: I/O channels architecture

I/O processor

It is an specialized processor which contains a local memory of its own. It not only loads and stores into memory but also can execute instructions which are among a set of I/O operations. The IOP (Input Output Processor) interfaces to the system and devices. In an IOP based system, I/O devices can directly access the memory without intervention by the processor. The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure the data words from many different sources to the format of the memory for transfer. Data are gathered in the IOP at device rate and bit capacity while CPU is executing its program. After data are assembled into the memory word they are transferred from IOP directly into the

memory by *stealing* one memory cycle from CPU. Similarly an output word transfer from memory to the IOP is directed from the IOP to the output devices at device rate and capacity.

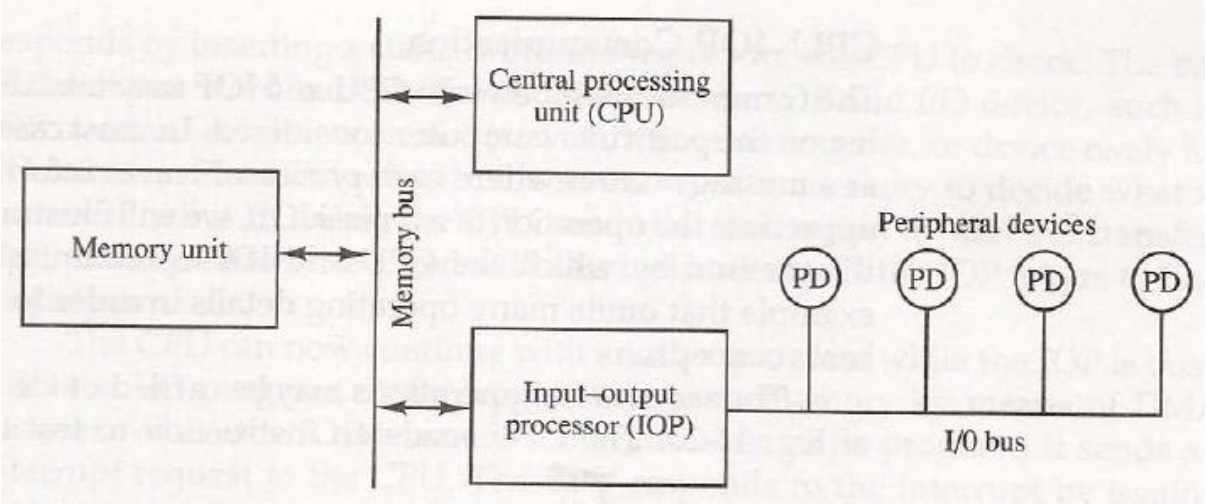


Figure: Block diagram of computer with I/O processor

CPU – IOP Communication

The communication between CPU and IOP may take different forms depending upon a particular computer considered. Here is the flow chart of CPU-IOP communication.

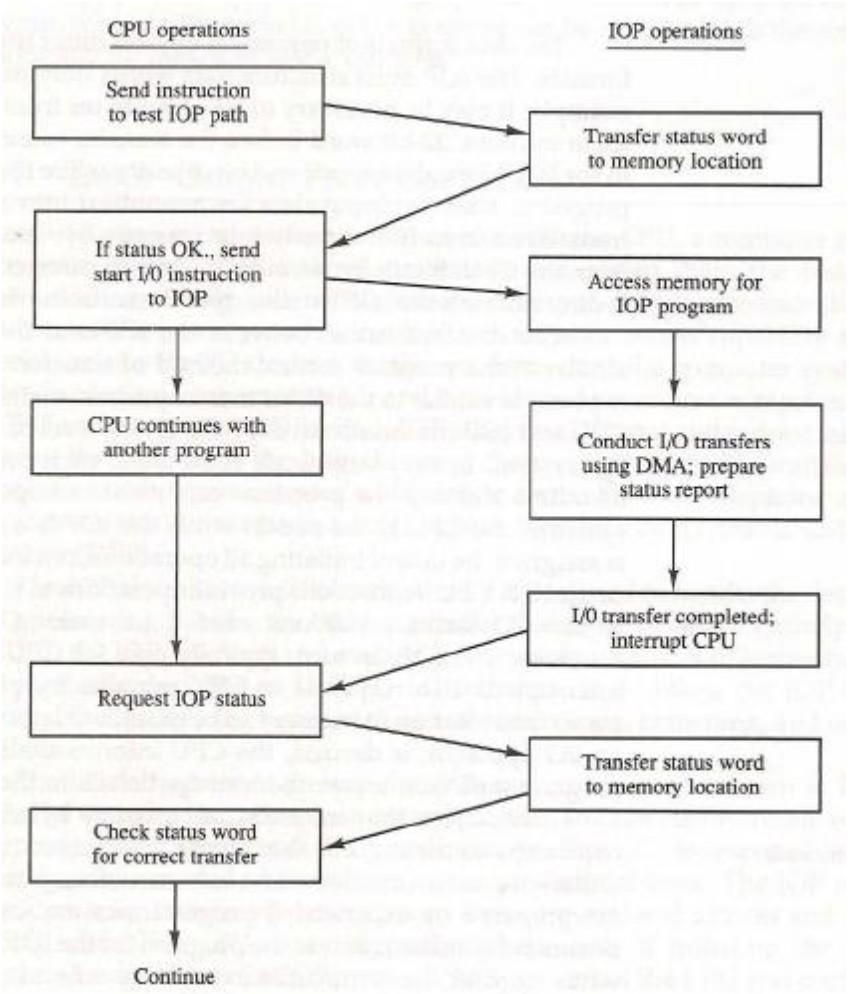


Figure: Flow chart of CPU-IOP Communication

4.5 External Interfaces

The interface to a peripheral from an I/O module must be tailored to the nature and operation of the peripheral. One major characteristic of the interface is whether it is serial or parallel. In a **parallel interface**, there are multiple lines connecting the I/O module and the peripheral, and multiple bits are transferred simultaneously. In a **serial interface**, there is only one line used to transmit data, and bits must be transmitted one at a time. A parallel interface has traditionally been used for higher-speed peripherals, such as tape and disk, while the serial interface has traditionally been used for printers and terminals. With a new generation of high-speed serial interfaces, parallel interfaces are becoming much less common. In either case, the I/O module must engage in a dialogue with the peripheral. In general terms, the dialogue for a write operation is as follows:

1. The I/O module sends a control signal requesting permission to send data.

2. The peripheral acknowledges the request.

3. The I/O module transfers data (one word or a block depending on the peripheral).

4. The peripheral acknowledges receipt of the data.

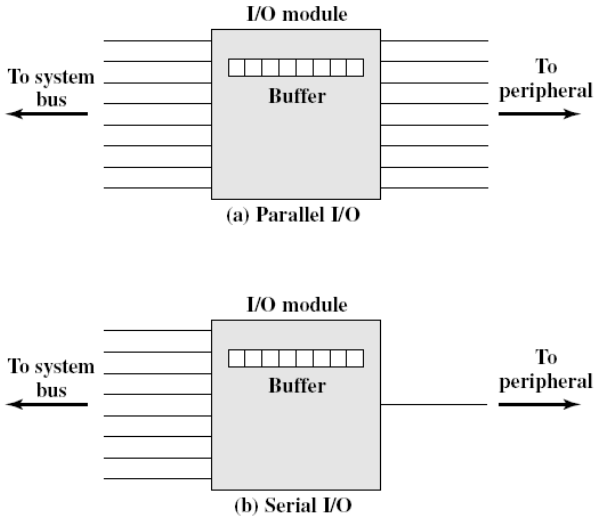


Figure: Parallel and Serial I/O

A read operation proceeds similarly. Key to the operation of an I/O module is an internal buffer that can store data being passed between the peripheral and the rest of the system. This buffer allows the I/O module to compensate for the differences in speed between the system bus and its external lines.

FireWire Serial Bus

With processor speeds reaching gigahertz range and storage devices holding multiple gigabits, the I/O demands for personal computers, workstations, and servers are formidable. Yet the high-speed I/O channel technologies that have been developed for mainframe and supercomputer systems are too expensive and bulky for use on these smaller systems. Accordingly, there has been great interest in developing a high-speed alternative to Small Computer System Interface (SCSI) and other small system I/O interfaces. The result is the IEEE standard 1394, for a High Performance Serial Bus, commonly known as FireWire.

InfiniBand

InfiniBand is a recent I/O specification aimed at the high-end server market. The first version of the specification was released in early 2001 and has attracted numerous vendors. The standard describes an architecture and specifications for data flow among processors and intelligent I/O devices. InfiniBand has become a popular interface for storage area networking and other large storage configurations. In essence, InfiniBand enables servers, remote storage, and other network devices to be attached in a central fabric of switches and links. The switch-based architecture can connect up to 64,000 servers, storage systems, and networking devices.

4.6 Data communication processor

A data communication processor is a processor in an I/O processor that distributes and collects data from many remote terminals connected through telephone or other communication lines. It is a specialized processor designed to communicate directly with data communication networks. With the use of data communication processor, the computer can service fragments of each networks demand in an interspersed manner and thus have apparent behavior of

servicing many users at once. I/O processors communicate with the peripherals through a common I/O bus and use it to transfer the information to and from the I/O processor. A data communication processor communicates with each terminal through a single pair of wire. Both data and control information are transmitted in serial fashion that results in transfer rate much slower. The work of data communication processor is to transmit and collect digital information to and from each terminal, determine if information is data or control and respond to the request accordingly. The processor also must communicate with the CPU and memory in the same manner as any I/O process.

4.7 Buses

A bus is a communication pathway connecting two or more devices. Key characteristics are that it is a shared transmission medium.

A bus line may be classified as:

1. Address bus
2. Data bus
3. Control bus

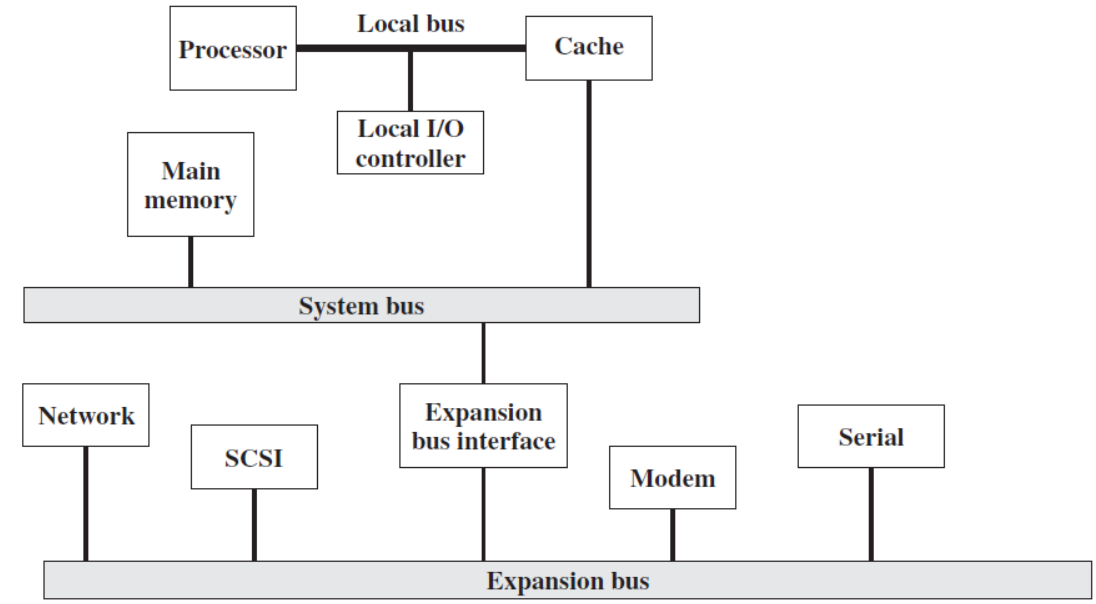
Buses may be serial or parallel, and the bus performance can be limited by:

1. Data propagation delay through longer buses.
2. Demand for access to the bus from all devices at the same time.

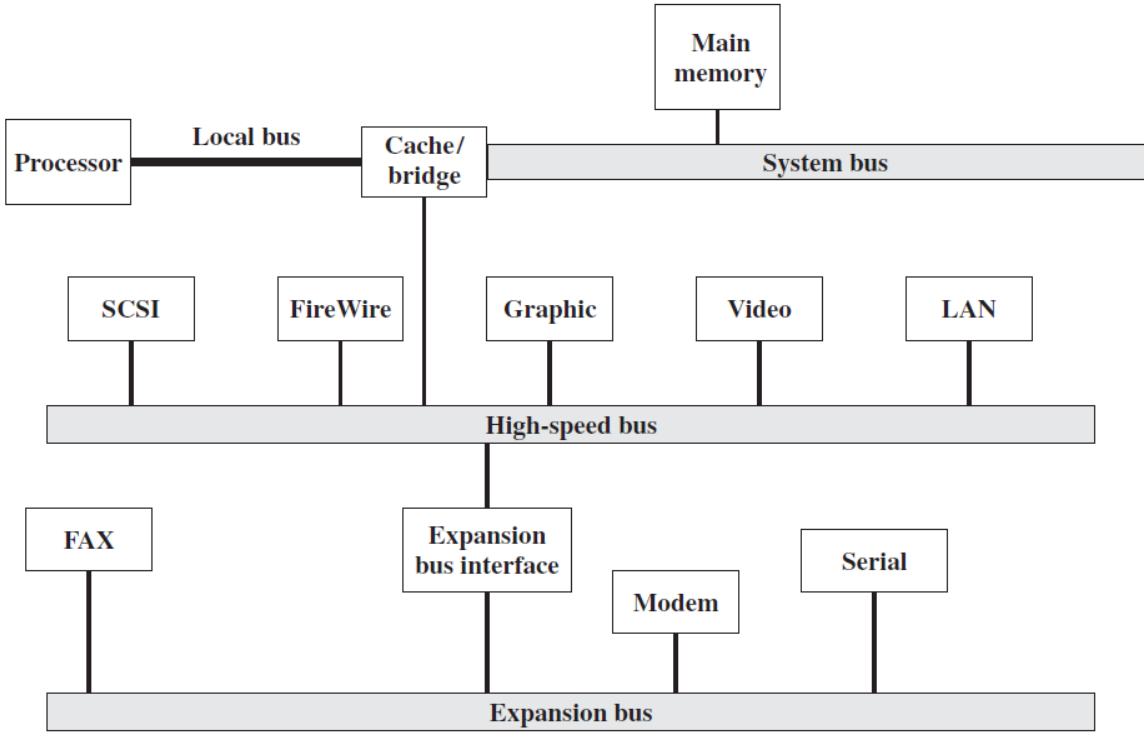
To avoid the bottle neck, multiple buses hierarchies are used. Their property include

- High speed limited access buses close to the processor.
- Slower speed general access buses further away.

Different bus hierarchy



(a) Traditional bus architecture



(b) High-performance architecture

a. Traditional Bus hierarchy:

- Local bus connects processor to cache, local I/O controller.
- May support one or more local devices.
- Processor connected to system via cache.
- An expansion bus connects other devices to system.

- An expansion bus interfaces buffers data transfer between the system buses and I/O controller on expansion bus.
- Network controller includes LANs, WANs, SCSI (Small Computer System Interface) that supports local disk drives and other peripherals.
- This traditional bus performance worsens when higher performance I/O devices are connected.

b. High performance architecture

- Local bus that connects the processor to a cache controller which in turn connected to a system bus that supports main memory.
- Cache controller is integrated into a bridge or buffering device that connects to high speed bus.
- High speed bus arrangements specifically designed to support high capacity I/O devices.
- Lower speed devices are still supported and connected to expansion buses connected to high speed bus through expansion bus interface.
- The advantage of this arrangement is that high speed bus brings high demand devices closer to the processor and at the same time is independent of the processor.

Elements of Bus Design

1. Types:
    a. Dedicated: Bus line that is permanently assigned to one function or to a physical subset of computer components.
    b. Multiplexed: Using same bus for multiple purposes. e.g. first sending address through the bus and making address valid line active, signal address is being transmitted. Then after using same bus to transmit data for read or write data transfer. The advantage will be, the use of fewer lines which saves space and usually cost. The disadvantage would be more complex circuitry needed and at certain events sharing or bus cannot be possible in parallel.

2. Method of arbitration
    - It is the process of insuring only one device places information onto the bus at a time.
    - In master slave mechanism
        - Purpose is to designate one device, either the processor or an I/O module as master.
        - Master: given control of the bus and can place information onto it.
        - Slave: receives information from master
    - Bus arbitration can be obtained in two methods:
    - Centralized: where central bus controller mediates all devices request for the bus.
    - De-centralized: where no centralized controller is used. All devices contain logic to control access to the bus instead.
3. Bus width
   The number of bits which are carried out simultaneously more, more width greater number of bits transfer.

4. Timing
    a. Synchronous timing: time controlled by a common clock signal. All events start at the beginning of a clock cycle, synchronous transfer are timed using a common clock signal
    b. Asynchronous timing: timing based on handshaking protocol. It is more flexible than synchronous bus but more complicated. It can accommodate wider range of devices speeds. Asynchronous transfers are timed with handshake signals.
5. Data transfer type
    a. Read: Reading data from memory, I/O.
    b. Write: Writing data to memory, I/O.
    c. Read-Modify-Write.
    d. Read-After-Write.
    e. Block: some buses support block data transfer.

imp

Chapter 8: Reduced Instruction Set Computer

Reduced Instruction Set Computer (RISC)

In early 1980s a number of computer designers recommended that computers users use fewer instructions with a simple construct so that they can be executed much faster within the CPU without having to use memory as often. The concept of RISC architecture involves attempt to reduce the execution time by simplifying the instruction set of the computer. The major characteristics of RISC processor are:

1. Relatively few instructions.
2. Few and simple addressing modes.
3. Memory access limited to load and store instruction.
4. All operations done within the register of CPU.
5. Fixed length, easily decoded instruction format.
6. Single cycle instruction execution.
7. Hardwired rather than micro programmed control.

Complex Instruction Set Computer (CISC)

The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in high level language. A computer with large number of instruction is classified as Complex Instruction Set Computer (CISC). We have noted the trend to richer instruction sets which include a larger number of instructions and more complex instruction. Two principle reasons have motivated this trend, a desire to simplify compliers and desire to improve performance also the shift to HLL (High Level Language) so that now machines have to designed that provide better supports for HLLs. The translation from high level to machine level is done by compiler program. The task of a compiler is to generate a sequence of machine instruction for each high level language statement directly. The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement is written in high level language.

The major characteristics of CICS architecture are:

1. Large number of instruction typically from hundreds to 250 instructions.
2. Some instruction that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes typically from 5 to 20 different modes.
4. Variable length instruction formats.
5. Instruction that manipulate operands in memory.
6. Simple compilers will do the job.
7. Multiple cycle instruction (single HLL instruction is broken into smaller instruction).
8. Micro programmed implementation.
9. Simple control unit.
10. Indirect addressing required.

Is CISC or RISC better?

- Programs run faster on RISC processor than on CISC processor, however it may not be due to the RISC feature but because of technology, better compilers.
- Similar instruction set of RISC processor results in larger memory requirement compared to similar programs.
- Compilers for CISC processor, however cost of storage unit is being inexpensive.
- Most of the current processors are not typically RISC or CISC; they combine advantage of both approaches.

Comparison

Architecture characteristics	CISC	RISC
- Instruction size	Varies	One size usually 32 bits
- Instruction format	Field placement varies	Regular, consistent placement of fields.
- Instruction semantics Semantics(gap between HLL to LLL)	Varies from simple to complex, possible many dependent operations per instruction	Almost always one simple operation.
- Registers	Few, sometimes special	Many general purpose
- Memory reference	Bundled with operations in many different types of instruction	Not bundled with operations i.e. load or store architecture.
- Hardware design	Exploit micro-coded implementations	Exploit implementation with one pipeline and no micro-code.

Instruction Pipelining/ Instruction level pipelining imp

Pipelining is an implementation technique where multiple instructions are overlapped. Instruction is divided into no. of states so that at a clock cycle one stage of instruction is being processed, while another stage of another instruction will be processed in parallel, thus speeding the processing time. Pipelining does not decrease the time for individual instruction but at the same time it does a lot more stages/ jobs thus increasing the output.

$$speedup = \frac{cycle\ required\ during\ unpipeling\ execution}{cycle\ required\ during\ pipeling\ execution}$$

Suppose we have 5 instructions, if we process those 5 instructions sequentially say each instruction take 4 clock cycles then total clock cycles is 20. But if pipelining is done with 4 stages just 8 clock cycle will be sufficient. Speed up=20/8=2.5

Clc cycle	1	2	3	4	5	6	7	8
Instr. 1	F	D	E	W				
Instr. 2		F	D	E	W			
Instr. 3			F	D	E	W		
Instr. 4				F	D	E	W	
Instr. 5					F	D	E	W

Basic pipeline steps

1. Instruction fetch (IF): Instruction pointed to by pc is fetched from memory.
2. Instruction Decode (ID): Instruction decode.
3. Execution/ Effective address calculation (EX): the ALU operates on operand from ALU, input register & eventually puts the result in ALU register.
4. Memory access/Branch completion (MEM): only for load, store & branch instruction.
5. Write back (WB): the result of the instruction execution is stored into register file.

IF	ID	EX	MEM	WB

Types of pipelining

1. Simple pipeline: This is simple pipelining with single stage completed in single clock cycle.

F	D	E	W		
	F	D	E	W	
		F	W	E	W

2. Super pipeline: The next stage of instruction starts before the completion of first stage of instruction.

F		D		E		W	
	F		D		E		W
		F		D		E	W

3. Super scalar pipeline: Number of same stages of different instruction is carried out at once.

F	D	E	W		
F	D	E	W		
F	D	E	W		
	F	D	E	W	
	F	D	E	W	
	F	D	E	W	

Arithmetic pipeline

Pipeline units are usually found in very high speed computers. They are used to implement for floating operations like multiplication. The floating point operations are easily decomposed into sub operations and then calculated. Example: a pipeline unit for floating addition and subtraction. Consider the two normalized floating point numbers

$X = 0.9504x10^3$ & Y$0.8200x10^2$

Now

Segment 1: Two exponents are subtracted 3-2=1. The larger exponent is chosen as the exponent of the result.

Segment 2: shift the mantissa of y to the right to obtain $X = 0.9504x10^3$ & Y$0.0820x10^3$. This aligns the mantissa under the same exponent.

Segment 3: the addition of two mantissa in segment 3 produces the sum Z$1.0324x10^3$.

Segment 4: the sum is adjusted by the normalizing the result so that it has a fraction with non zero first digit. Z$0.10324x10^4$.

The comparator, shifter, adder-subtractor, incrementer and decrementer in the floating point pipeline are implemented with the combination circuits, suppose the time delay of the four segments are t1=60ns, t2=70ns, t3=100ns & t4=80ns, and the interface register have a delay of tr=10ns,the clock cycle is chosen to be tp= t3+tr=110ns, and equivalent non pipeline floating adder-subtractor will have delay time of tn=t1+t2+t3+t4+t5+tr=320 ns.

Speed up=320/110=2.9 over a non pipelined adder.

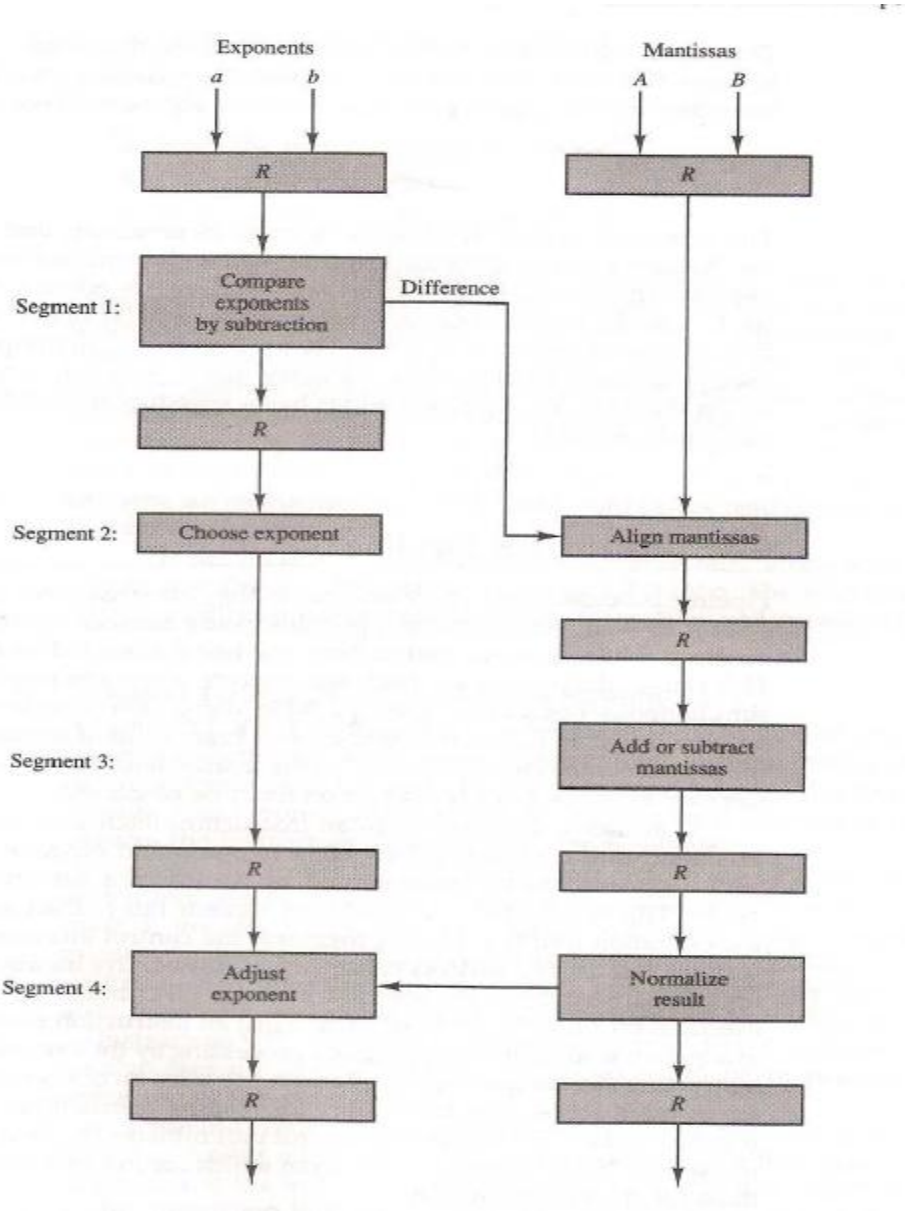


Figure: pipeline for floating point addition and subtraction

Instruction pipeline

An instruction pipeline reads consecutive instruction from memory while previous instruction is being executed in other segments. These causes the instructions fetch and execute phases to overlap and perform simultaneous operations. Computers with complex instruction require other phases in addition to the fetch and execute to process an instruction completely. In the most general case the computers needs to process following sequence of steps.

1. Fetch the instruction into memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operand from memory.
5. Execute the instruction.
6. Store the results in proper place.

Certain difficulties that will prevent the instruction pipeline from operating at its maximum rate are different segments taking different time to operate on the incoming information. Two or more segments may require memory access at the same time causing one segment to wait until another is finished with memory.

Example

1. FI: is the segment that fetches an instruction.
2. DA: is the segment that decodes the instruction and calculates the effective address.
3. FO: is the segment that fetches the operand.
4. EX: is the segment that executes the instruction.

Steps	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction 1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
Branch 3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Figure: timing of instruction pipeline

imp

Pipeline hazards

A pipeline hazards occurs when an instruction cannot complete a step in its execution, due to some events in previous clock cycle. When an instruction must be held for one or more clock pulse in-order to complete a step in its execution, this is called "pipeline stall" or "bubble".

Type of hazards

1. Structural hazards (resource conflicts):

These hazards arises from resources conflicts when hardware cannot support all possible combinations of instruction in simultaneous overlapped execution e.g for unified cache memory

Instruction	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Load	IF	ID	EX	MEM	WB		
Instr 1		IF	ID	EX	MEM	WB	
Instr 2			IF	ID	EX	MEM	WB
Instr 3				IF	ID	EX	MEM
Instr 4					IF	ID	EX

Both MEM and IF require to refer memory but we have unified cache where the instruction and data are stored in a single cache and in each cycle only one request can be processed.

Remedy

- Introduce stall.
- Replicating resources (increase resources).

2. Data hazards: These occur when one step in pipeline must wait the completion of a previous instruction, so there by creating conflict. A good compiler can reduce these hazards but not eliminate them completely.

Types

a. RAW (True dependency): A read after write hazard occurs when an instruction needs an operand whose value will be evaluated in the previous instruction so 2nd instruction must wait till the completion of 1st instruction.

   Add r1, r2, r3

   Mul r4, r1, r5

	1	2	3	4	5	6
Add	IF	ID	EX	MEM	WB	
Mul		IF	ID	EX	MEM	WB

   Here the value of r1 is needed at cycle 4 but it is written at cycle 5

b. WAR (false / anti dependency): a write after read hazard is the reverse of RAW. Here first the write operation will occur before read, but there should be read operation first.

   Example:

   Mul r1, r2, r3

   Add r2, r4, r5

   If the add instruction is completed first then the mul instruction will get new value of r2, but it needs the old value of r2.

   Remedy: reordering

c. WAW: a write after hazard is a situation when j instruction tries to write an operand before it is written by i. the written up being performed in the wrong order , leaving the value written i, rather than value written by j in destination.

   Remedy: data forwarding

3. Control hazards

Most harmful hazard arises from the need to make decision based on the result on one instruction while others are executing. A typical e.g would be the execution of conditional branch instruction if the branch is taken the instruction currently in the pipeline might be invalid.

Remedy

1. Prefetch branch target:
   - When the branch instruction is decoded, begin to fetch the branch target instruction and place in prefetch buffers.
   - If branch is not taken, the sequential instruction is ready in the pipe.
   - If branch is taken the next instruction has been prefetched and results in minimal branch delay.
2. Branch target buffer (BTB): The BTB is associative memory included in the fetch segment of pipeline. Each entry in BTB consists of the address previously executed branch instruction and target instruction for that branch. It also store next few instruction after branch target instruction.

   When pipeline decodes a branch instruction it searches it's associative memory in BTB for the address of instruction, if it is in BTB the instruction is available directly and prefetch continuous from the new path. If the instruction is not in BTB, pipeline shifts to new instruction stream and store the target instruction in BTB.
3. Loop buffer: a variation of BTb when a program loop is detected in the program it is stored in loop buffer in its entirely including all branches.
4. Branch prediction: A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. A correct prediction eliminated the wasted time caused by branch penalties.
5. Delayed branch: A procedure employed in most RISC processor is delayed branch. In this procedure the compiler detects the branch instruction and re arranges the machine language code sequence by inserting useful instruction that keep the pipeline operating without interruptions. An example of delayed branch is insertion of no operation instruction after branch instruction. This causes the computer to fetch the target instruction during the execution of no operation instruction allowing a continuous flow of pipeline

RISC pipeline imp

Instruction pipelining is often used to enhance performance. Let us reconsider this in the context of a RISC architecture. Most instructions are register to register, and an instruction cycle has the following two stages:

• I: Instruction fetch.

• E: Execute. Performs an ALU operation with register input and output.

For load and store operations, three stages are required:

• I: Instruction fetch.

• E: Execute. Calculates memory address

• D: Memory. Register-to-memory or memory-to-register operation.

Figure 13.6a depicts the timing of a sequence of instructions using no pipelining. Clearly, this is a wasteful process. Even very simple pipelining can substantially improve performance. Figure 13.6b shows a two-stage pipelining scheme, in which I and E stages of two different instructions are performed simultaneously. The two stages of the pipeline are an instruction fetch stage, and an execute/memory stage that executes the instruction, including register-to-memory and memory to- register operations. Thus we see that the instruction fetch stage of the second instruction can be performed in parallel with the first part of the execute/memory stage. However, the execute/memory stage of the second instruction must be delayed until the first instruction clears the second stage of the pipeline. This scheme can yield up to twice the execution rate of a serial scheme. Two problems prevent the maximum speed-up from being achieved. First, we assume that a single-port memory is used and that only one memory access is possible per stage. This requires the insertion of a wait state in some instructions. Second, a branch instruction interrupts the sequential flow of execution. To accommodate this with minimum circuitry, a NOOP instruction can be inserted into the instruction stream by the compiler or assembler. Pipelining can be improved further by permitting two memory accesses per stage. This yields the sequence shown in Figure 13.6c. Now, up to three instructions can be overlapped, and the improvement is as much as a factor of 3. Again, branch instructions cause the speedup to fall short of the maximum possible. Also, note that data dependencies have an effect. If an instruction needs an operand that is altered by the preceding instruction, a delay is required. Again, this can be accomplished by a NOOP. The pipelining discussed so far works best if the three stages are of approximately equal duration. Because the E stage usually involves an ALU operation, it may be longer. In this case, we can divide into two sub-stages:

E1: Register file read

E2: ALU operation and register write

Because of the simplicity and regularity of a RISC instruction set, the design of the phasing into three or four stages is easily accomplished. Figure 13.6d shows the result with a four-stage pipeline. Up to four instructions at a time can be under way, and the maximum potential speedup is a factor of 4. Note again the use of NOOPs to account for data and branch delays.

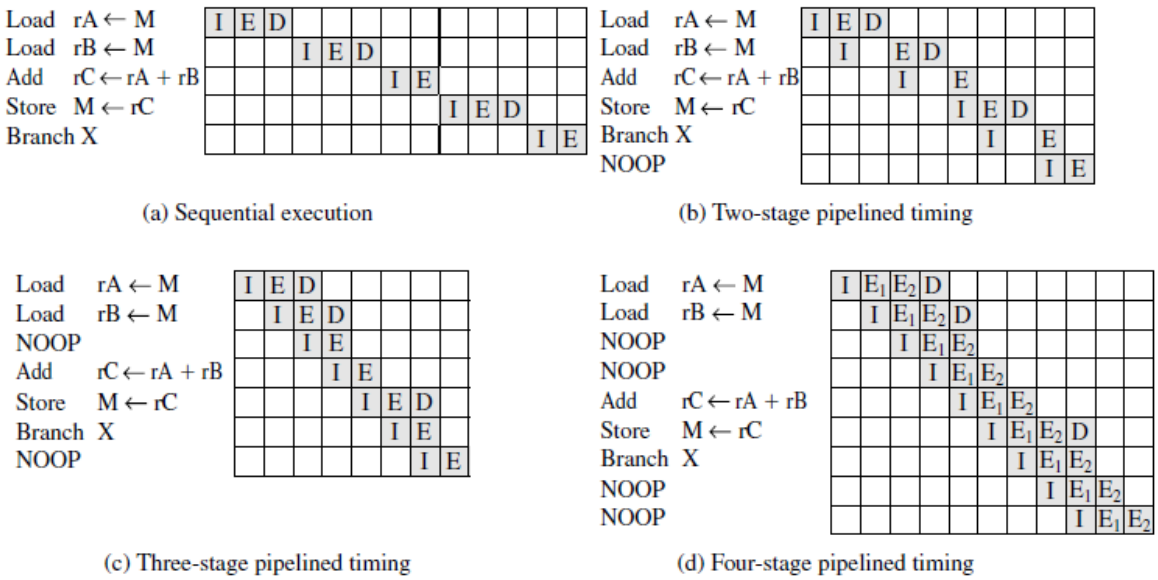


Figure 13.6 The Effects of Pipelining

Optimization of Pipelining

Because of the simple and regular nature of RISC instructions, pipelining schemes can be efficiently employed. There are few variations in instruction execution duration, and the pipeline can be tailored to reflect this. However, we have seen that data and branch dependencies reduce the overall execution rate.

DELAYED BRANCH To compensate for these dependencies, code reorganization techniques have been developed. First, let us consider branching instructions. *Delayed branch,* a way of increasing the efficiency of the pipeline, makes use of a branch that does not take effect until after execution of the following instruction (hence the term *delayed*). The instruction location immediately following the branch is referred to as the *delay slot*. This strange procedure is illustrated in Table 13.8. In the column labeled "normal branch," we see a normal symbolic instruction machine-language program. After 102 is executed, the next instruction to be executed is 105. To regularize the pipeline, a NOOP is inserted after this branch. However, increased performance is achieved if the instructions at 101 and 102 are interchanged. Figure 13.7 shows the result.

Table 13.8 Normal and Delayed Branch

Address	Normal Branch		Delayed Branch		Optimized Delayed Branch	
100	LOAD	X, rA	LOAD	X, rA	LOAD	X, rA
101	ADD	1, rA	ADD	1, rA	JUMP	105
102	JUMP	105	JUMP	106	ADD	1, rA
103	ADD	rA, rB	NOOP		ADD	rA, rB
104	SUB	rC, rB	ADD	rA, rB	SUB	rC, rB
105	STORE	rA, Z	SUB	rC, rB	STORE	rA, Z
106			STORE	rA, Z		

2. ***DELAYED LOAD*** A similar sort of tactic, called the delayed load, can be used on LOAD instructions. On LOAD instructions, the register that is to be the target of the load is locked by the processor. The processor then continues execution of the instruction stream until it reaches an instruction requiring that register, at which point it idles until the load is complete. If the compiler can rearrange instructions so that useful work can be done while the load is in the pipeline, efficiency is increased.

3. **LOOP UNROLLING** Another compiler technique to improve instruction parallelism is loop unrolling. Unrolling replicates the body of a loop some number of times called the unrolling factor (u) and iterates by step u instead of step 1. Unrolling can improve the performance by

   • reducing loop overhead

   • increasing instruction parallelism by improving pipeline performance

   • improving register, data cache, or TLB locality

   Figure illustrates all three of these improvements in an example. Loop overhead is cut in half because two iterations are performed before the test and branch at the end of the loop. Instruction parallelism is increased because the second assignment can be performed while the results of the first are being stored and the loop variables are being updated. If array elements are assigned to registers, register locality will improve because a[i] and a [i+1] are used twice in the loop body, reducing the number of loads per iteration from three to two.

```
do i=2, n-1
        a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a) Original loop

```
do i=2, n-2, 2
        a[i] = a[i] + a[i-1] * a[i+1]
        a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2, 2) = i)  then
   a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b) Loop unrolled twice

Figure 13.8 Loop unrolling

Register file

We know that there are a large proportion of assignment statements in HLL programs, and many of these are of the simple form A←B, Also, there is a significant number of operand accesses per HLL statement. If we couple these results with the fact that most accesses are to local scalars, heavy reliance on register storage is suggested. The reason that register storage is indicated is that it is the fastest available storage device, faster than both main memory and cache. The complete set of register is known as ***register file*** and particular set of register is called ***window***. The register file is physically small, on the same chip as the ALU and control unit, and employs much shorter addresses than addresses for cache and memory. Thus, a strategy is needed that will allow the most frequently accessed operands to be kept in registers and to minimize register-memory operations.

Two basic approaches are possible, one based on ***software and the other on hardware***. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to allocate registers to those variables that will be used the most in a given time period. This approach requires the use of sophisticated program-analysis algorithms. The hardware approach is simply to use more registers so that more variables can be held in registers for longer periods of time.

Register Windows

The use of a large set of registers should decrease the need to access memory. The design task is to organize the registers in such a fashion that this goal is realized. Because most operand references are to local scalars, the obvious approach is to store these in registers, with perhaps a few registers reserved for global variables. On every call, local variables must be saved from the registers into memory, so that the registers can be reused by the called program. Furthermore, parameters must be passed. On return, the variables of the parent program must be restored (loaded back into registers) and results must be passed back to the parent program. The solution is based on two forms. First, a typical procedure employs only a few passed parameters and local variables. Second, the depth of procedure activation fluctuates within a relatively narrow range. To exploit these properties, multiple small sets of registers are used, each assigned to a different procedure. A procedure call automatically switches the processor to use a different fixed-size window of registers, rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing. The concept is illustrated in Figure below. At any time, only one window of registers is visible and is addressable as if it were the only set of registers. The window is divided into three fixed-size areas. *Parameter registers* hold parameters passed down from the procedure that called the current procedure and hold results to be passed back up. *Local registers* are used for local variables, as assigned by the compiler. *Temporary registers* are used to exchange parameters and results with the next lower level (procedure called by current procedure).The temporary registers at one level are physically the same as the parameter registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data. Keep in mind that, except for the overlap, the registers at two different levels are physically distinct. That is, the parameter and local registers at level J are disjoint from the local and temporary registers at level J 1. To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. Instead, the register windows can be used to hold the few most recent procedure activations. Older activations must be saved in memory and later restored when the nesting depth decreases. Thus, the actual organization of the register file is as a circular buffer of overlapping windows.

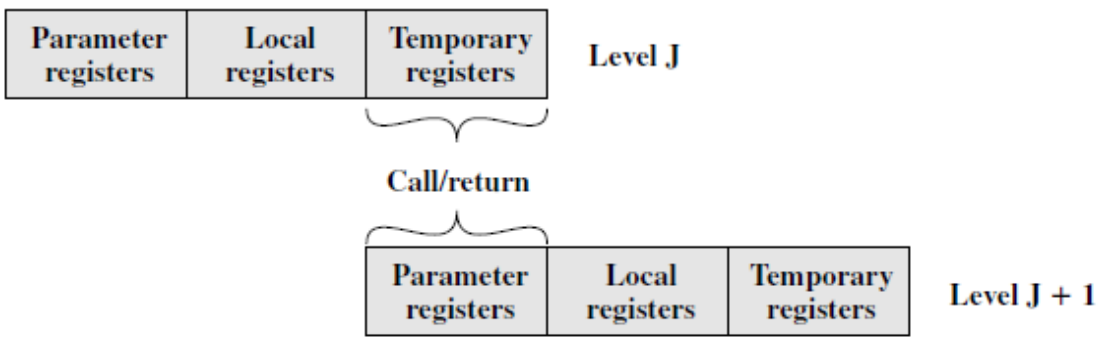


Figure: overlapping register window

Register renaming

We have seen that during pipelining there is possibility of WAW dependencies and WAR dependencies. These dependencies differ from RAW data dependencies and resources conflicts, which reflect the flow of data through a program and the sequence of execution. WAW dependencies and WAR dependencies on the other hand arise because the value in register may no longer reflect the sequence of values dictated by the program flow.

When instruction is issued in sequence and complete in sequence it is possible to specify the contents of each register at each point in the execution. When out-of-order techniques are used the values in register cannot be fully known at each point in time just from consideration of the sequence of instruction dictated by the program. In effect values are in conflict for the use of register and the processor must restore those conflicts by occasionally stalling a pipeline stage.

One method for coping with these types of storage conflicts is based on a traditional resource conflict solution. i.e. duplication of resource. In this context the technique id referred to as ***register renaming.*** In essence register are allocated dynamically by the processor hardware and they are associated with the values needed by instruction at various points in time. When a new register value is created a new register is allocated for that value. Subsequent instruction that access that value as source operand in that register must go through a renaming process, the register reference in those instruction must be revised to refer the register containing the needed value. Thus the same original register reference in several different instructions may refer to different actual register, if different values are intended.

For example

The instructions	can be written as
R7=R1*R2	R7=R1*R2
R1=R0-R2	S1=R0-R2
R3=R3*R1	R3=R3*S1
R1=R4+R4	S2=R4+R4

HereS1, S2 are secreting register not visible to programmer. The register R1 is renamed to S1 in first part and to S2 in second part so that addition can be started before R1 is free.

Chapter 9: Introduction to Parallel Processing

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. Processors execute programs by executing machine instructions in a sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results). This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel. This approach is taken further with superscalar organization, which exploits instruction-level parallelism. With a superscalar machine, there are multiple execution units within a single processor, and these may execute multiple instructions from the same program in parallel. Some of the prominent approaches to parallel organization are:

Symmetric multiprocessors (SMPs): SMP organization consists of multiple processors share a common memory, interconnected by a bus or some sort of switching arrangement. This organization raises the issue of cache coherence.

Clusters: it is a group of interconnected independent computers working together as a whole working as a unified computing resource, that create the illusion of being one machine. The term whole computer means a system that can run on its own part from cluster.

Non-uniform memory access (NUMA): It is a shared memory multiprocessor in which the access time from a given processor to a word in memory varies with the location on the memory word. The NUMA approach is relatively new.

Parallel processing

Parallel processing is a term used to denote a large class of techniques that provide simultaneous data processing tasks for the sole purpose of the increasing the computational speed of computer system. A parallel processing system is able to perform concurrent data processing to achieve faster execution time. For example when an instruction is being is executed in ALU the next instruction can be read from memory. The system can also have two or more CPUs so that multiple processing can be done during a given time interval. This technique used more hardware and thus cost of the system increases. Parallel processing can be done in many ways for example using parallel set of registers, shifters, these are relatively less complex and those having multiple operations.

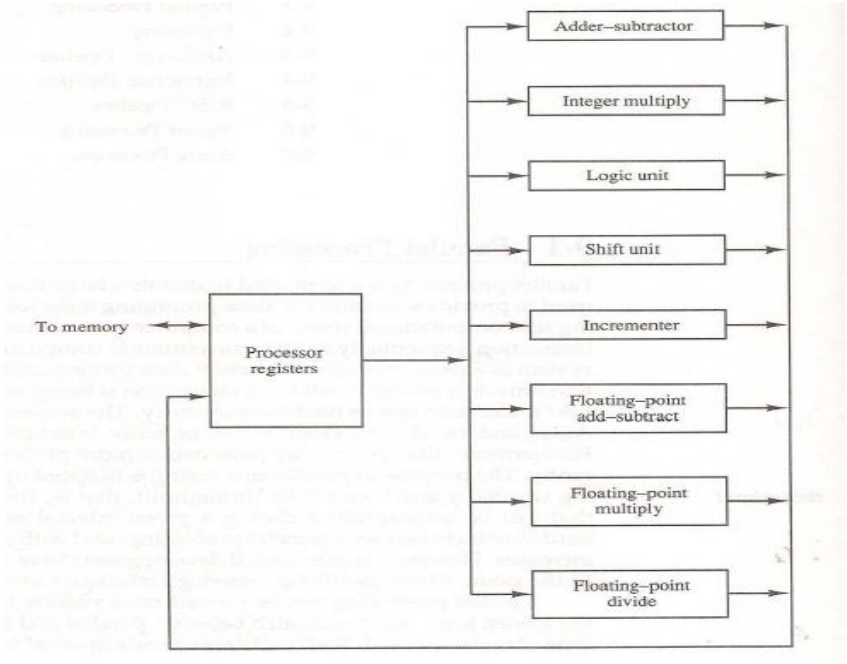


Figure: processor with multiple functional units

- Figure shows one possible way of separating the execution unit into eight functional units.
- All units are independent of each other so that one number can be shifted while other number can be incremented or multiplied.

A multi functional organization is usually associated with a complex control unit to co-ordinate all activities among the various components.

MULTIPLE PROCESSOR ORGANIZATIONS

Types of Parallel Processor Systems

A taxonomy first introduced by Flynn is still the most common way of categorizing systems with parallel processing capability. Flynn proposed the following categories of computer systems:

• **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. Uni-processors fall into this category.

• **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category.

• **Multiple instruction, single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.

• **Multiple instruction, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters, and NUMA systems fit into this category. With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation. MIMDs can be further subdivided by the means in which the processors communicate. If the processors share a common memory, then each processor accesses programs and data stored in the shared memory, and processors communicate with each other via that memory. The most common form of such system is known as a **symmetric multiprocessor (SMP)**. In an SMP, multiple processors share a single memory or pool of memory by means of a shared bus or other interconnection mechanism; a distinguishing feature is that the memory access time to any region of memory is approximately the same for each processor. A more recent development is the **non-uniform memory access (NUMA)** organization.

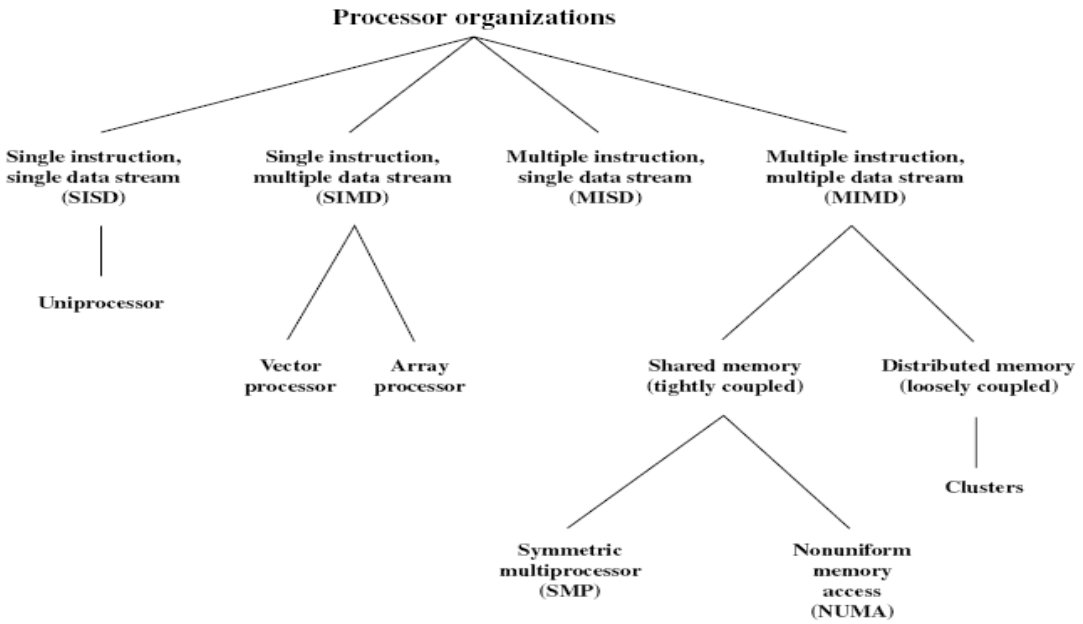
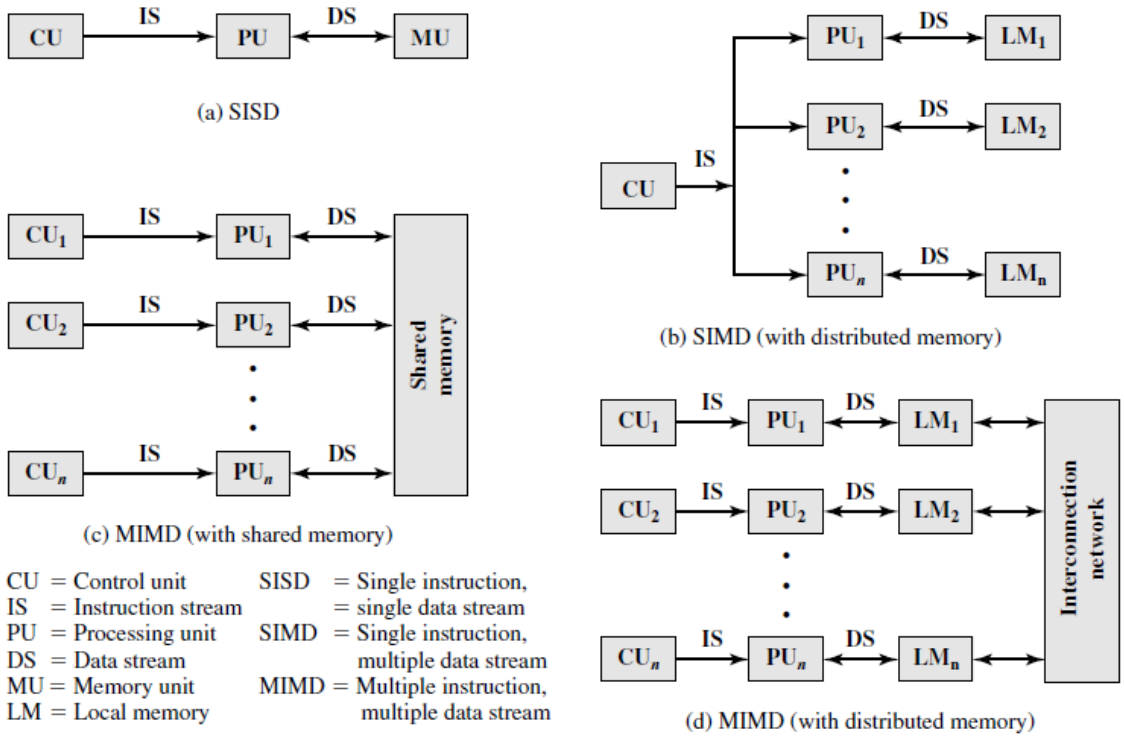


Figure 17.1 A Taxonomy of Parallel Processor Architectures



SYMMETRIC MULTIPROCESSORS

All single-user personal computers and most workstations contained a single general-purpose microprocessor. As demands for performance increase and as the cost of microprocessors continues to drop, vendors have introduced systems with an SMP organization. The term *SMP* refers to a computer hardware architecture and also to the operating system behavior that reflects that architecture. An SMP can be defined as a standalone computer system with the following characteristics:

1. There are two or more similar processors of comparable capability.

2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.

3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.

4. All processors can perform the same functions (hence the term *symmetric*).

5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

The operating system of an SMP schedules processes or threads across all of the processors. An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:

• **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.

Availability: In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.

• **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.

• **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

Organization

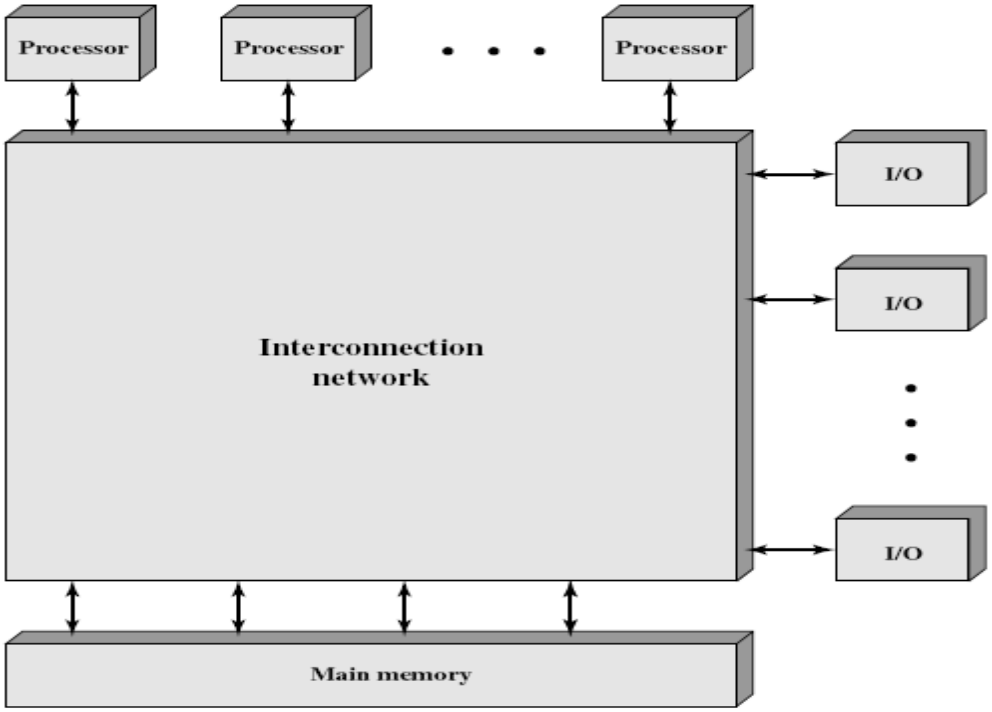


Figure 17.4 Generic Block Diagram of a Tightly Coupled Multiprocessor

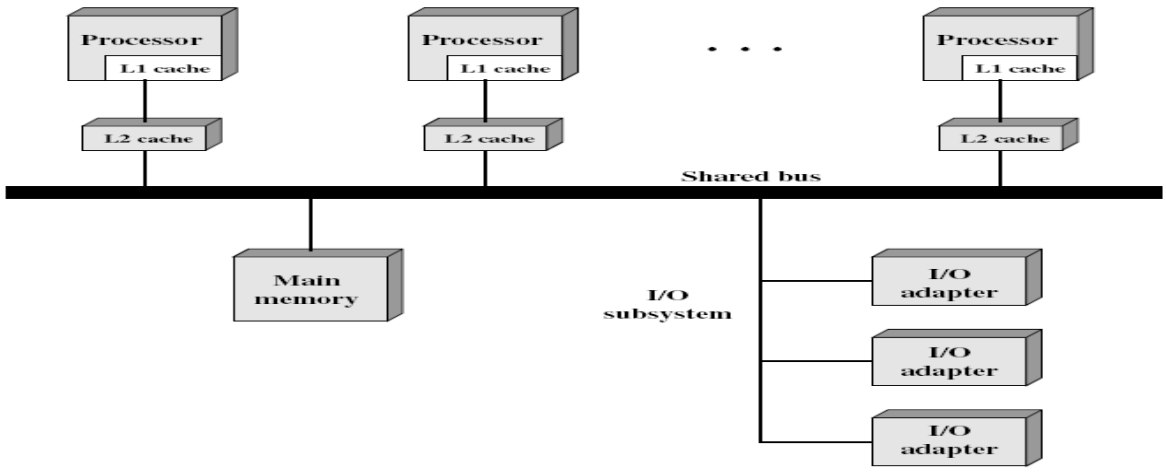


Figure 17.5 Symmetric Multiprocessor Organization

Figure 17.4 depicts in general terms the organization of a multiprocessor system. There are two or more processors. Each processor is self-contained, including a control unit, ALU, registers, and, typically, one or more levels of cache. Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism. The processors can communicate with each other through memory (messages and status information left in common data areas). It may also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible. In some configurations, each processor may also have its own private main memory and I/O channels in addition to the shared resources. The most common organization for personal computers, workstations, and servers is the time-shared bus. The time-shared bus is the simplest mechanism for constructing a multiprocessor system (Figure 17.5).The structure and interfaces are basically the same as for a single-processor system that uses a bus interconnection. The bus consists of control, address, and data lines. To facilitate DMA transfers from I/O processors, the following features are provided:

Addressing: It must be possible to distinguish modules on the bus to determine the source and destination of data.

• **Arbitration:** Any I/O module can temporarily function as "master." A mechanism is provided to arbitrate competing requests for bus control, using some sort of priority scheme.

• **Time-sharing:** When one module is controlling the bus, other modules are locked out and must, if necessary, suspend operation until bus access is achieved. These uniprocessor features are directly usable in an SMP organization. In this latter case, there are now multiple processors as well as multiple I/O processors all attempting to gain access to one or more memory modules via the bus.

The bus organization has several attractive features:

• **Simplicity:** This is the simplest approach to multiprocessor organization. The physical interface and the addressing, arbitration, and time-sharing logic of each processor remain the same as in a single-processor system.

• **Flexibility:** It is generally easy to expand the system by attaching more processors to the bus.

• **Reliability:** The bus is essentially a passive medium, and the failure of any attached device should not cause failure of the whole system.

The main drawback to the bus organization is performance. All memory references pass through the common bus. Thus, the bus cycle time limits the speed of the system. To improve performance, it is desirable to equip each processor with a cache memory. This should reduce the number of bus accesses dramatically.

Cache coherence

In multi processor systems it is customary to have one or two levels of cache associated with each processor. This organization is essential to achieve reasonable performance .it does however create a problem called cache coherence problem i.e. if same data is being is used by more than two processor the data will be in their cache. Now if changing the data in one cache is done but not in other or main memory, then same variable will hold different data in different process or multiple copies of same data can exist simultaneously and if processor are allowed to update their own copies freely and inconsistent view of memory can result . We have two different write polices in cache:

1. Write back: write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.
2. Write through: all the operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

It is clear that a write back policy can result in inconsistency. If two caches contain the same line and the line is updated in one cache, the other cache will unknowingly have an invalid data. Subsequent read to that invalid line produce invalid result. Even with write through policy, inconsistency can occur unless other caches monitor the memory traffic or receive some direct notification of the update. To solve cache coherence problem we discuss two approaches software and hardware.

Software solutions:

Software cache coherence schemes attempt to avoid the need for additional hardware circuitry and logic by relying on the compiler and operating system to deal with the problem. Software approaches are attractive because the overhead of detecting potential problems is transferred from run time to compile time, and the design complexity is transferred from hardware to software. On the other hand, compile-time software approaches generally must make conservative decisions, leading to inefficient cache utilization. Compiler-based coherence mechanisms perform an analysis on the code to determine which data items may become unsafe for caching, and they mark those items accordingly. The operating system or hardware then prevents non-cacheable items from being cached. The simplest approach is to prevent any shared data variables from being cached. This is too conservative, because a shared data structure may be exclusively used during some periods and may be effectively read-only during other periods. It is only during periods when at least one process may update the variable and at least one other process may access the variable that cache coherence is an issue. More efficient approaches analyze the code to determine safe periods for shared variables. The compiler then inserts instructions into the generated code to enforce cache coherence during the critical periods.

Hardware Solutions

Hardware-based solutions are generally referred to as cache coherence protocols. These solutions provide dynamic recognition at run time of potential inconsistency conditions. Because the problem is only dealt with when it actually arises, there is more effective use of caches, leading to improved performance over a software approach. In addition, these

approaches are transparent to the programmer and the compiler, reducing the software development burden. Hardware schemes differ in a number of particulars, including where the state information about data lines is held, how that information is organized, where coherence is enforced, and the enforcement mechanisms. In general, hardware schemes can be divided into two categories: directory protocols and snoopy protocols.

DIRECTORY PROTOCOLS Directory protocols collect and maintain information about where copies of lines reside. Typically, there is a centralized controller that is part of the main memory controller, and a directory that is stored in main memory. The directory contains global state information about the contents of the various local caches. When an individual cache controller makes a request, the centralized controller checks and issues necessary commands for data transfer between memory and caches or between caches. It is also responsible for keeping the state information up to date; therefore, every local action that can affect the global state of a line must be reported to the central controller. Typically, the controller maintains information about which processors have a copy of which lines. Before a processor can write to a local copy of a line, it must request exclusive access to the line from the controller. Before granting this exclusive access, the controller sends a message to all processors with a cached copy of this line, forcing each processor to invalidate its copy. After receiving acknowledgments back from each such processor, the controller grants exclusive access to the requesting processor. When another processor tries to read a line that is exclusively granted to another processor, it will send a miss notification to the controller. The controller then issues a command to the processor holding that line that requires the processor to do a write back to main memory. The line may now be shared for reading by the original processor and the requesting processor. Directory schemes suffer from the drawbacks of a central bottleneck and the overhead of communication between the various cache controllers and the central controller. However, they are effective in large-scale systems that involve multiple buses or some other complex interconnection scheme.

SNOOPY PROTOCOLS

Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor. A cache must recognize when a line that it holds is shared with other caches. When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism. Each cache controller is able to "snoop" on the network to observe these broadcasted notifications, and react accordingly. Snoopy protocols are ideally suited to a bus-based multiprocessor, because the shared bus provides a simple means for broadcasting and snooping. However, because one of the objectives of the use of local caches is to avoid bus accesses, care must be taken that the increased bus traffic required for broadcasting and snooping does not cancel out the gains from the use of local caches. Two basic approaches to the snoopy protocol have been explored: write invalidate and write update (or write broadcast).With a write-invalidate protocol, there can be multiple readers but only one writer at a time. Initially, a line may be shared among several caches for reading purposes. When one of the caches wants to perform a write to the line, it first issues a notice that invalidates that line in the other caches, making the line exclusive to the writing cache. Once the line is exclusive, the owning

processor can make cheap local writes until some other processor requires the same line. With a write-update protocol, there can be multiple writers as well as multiple readers. When a processor wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it. Neither of these two approaches is superior to the other under all circumstances. Performance depends on the number of local caches and the pattern of memory reads and writes. Some systems implement adaptive protocols that employ both write-invalidate and write-update mechanisms.

The MESI Protocol

To provide cache consistency on an SMP, the data cache often supports a protocol known as MESI. For MESI, the data cache includes two status bits per tag, so that each line can be in one of four states:

• **Modified:** The line in the cache has been modified (different from main memory) and is available only in this cache.

• **Exclusive:** The line in the cache is the same as that in main memory and is not present in any other cache.

• **Shared:** The line in the cache is the same as that in main memory and may be present in another cache.

• **Invalid:** The line in the cache does not contain valid data.

Table 17.1 MESI Cache Line States

	M Modified	E Exclusive	S Shared	I Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is …	out of date	valid	valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line …	does not go to bus	does not go to bus	goes to bus and updates cache	goes directly to bus

Vector processing

There is a class of computational problems that are beyond the capabilities of a conventional computer, like the programs that require vector or matrix calculation, where large amount of calculation has to be done. Computers with vector processing capabilities are in demand in specialized applications. Some of the areas of application are long range weather forecasting, petroleum exploration, seismic data analysis, medical diagnosis, aerodynamics and space flight simulator, artificial intelligence and expert systems, image processing.

Vector operations

Many engineering and scientific problems require arithmetic operations on large array of numbers; these are usually formulated as vector and matrix of floating point numbers. A vector is an ordered set of one dimensional array of data items i.e. as V= [V1, V2, V3…….Vn].A conventional sequence computer is capable of processing operands one at time. Consequently operations on vector must be broken into single computations with subscribed variables. For calculation using convectional processor, it is performed as:

For (i=1; i<=100; i))

{C (i) =A (i) +B (i);

}

This constitutes a program loop that read a pair of operands from array A and B and perform a floating point addition. The loop control variable is then updated and steps repeated 100 times.

A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instruction in the program loops. It allows operation to be specified with a single vector instruction of the form C (1:100) =A (1:100) +B (1:100)

The vector instruction includes the initial address of the operands, the length of the vectors and the operations to be performed all in one composite instruction. The addition is done with a pipelined floating point adder similar to the arithmetic pipeline. A possible instruction format for a vector instruction is

Op-code	Base address source 1	Base address source 2	Base address destination	Vector length

This is essentially a three address instruction with three fields specifying the base address of the operand and an additional field that gives the length of the data items in the vector.

See matrix multiplication from book example.

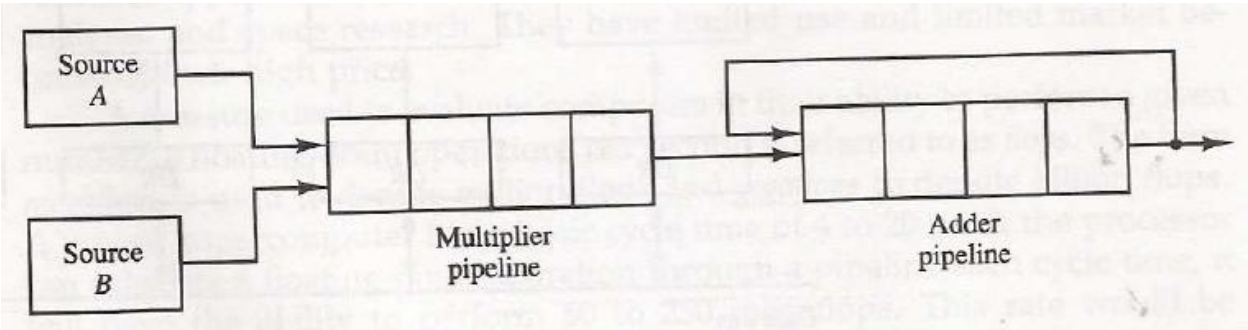


Figure: pipeline for calculating an inner product

Memory interleaving

Pipeline and vector processing often require simultaneous access to memory from two or more sources or fetching of instruction and an operand at the same time, so instead of using two memory buses the memory can be partitioned into a number of modules in a memory array together with its own address and data registers. Figure shows the four modules, the address registers receive information from a common address bus and data register communicated with bi-directional data bus. The two least significant bits of the address can be used to distinguish between four modules. The modular system permits one module to initiate a memory access while other modules are in process of reading or writing a word and each module can accept memory request independent of the state of other modules. This technique is called memory interleaving.

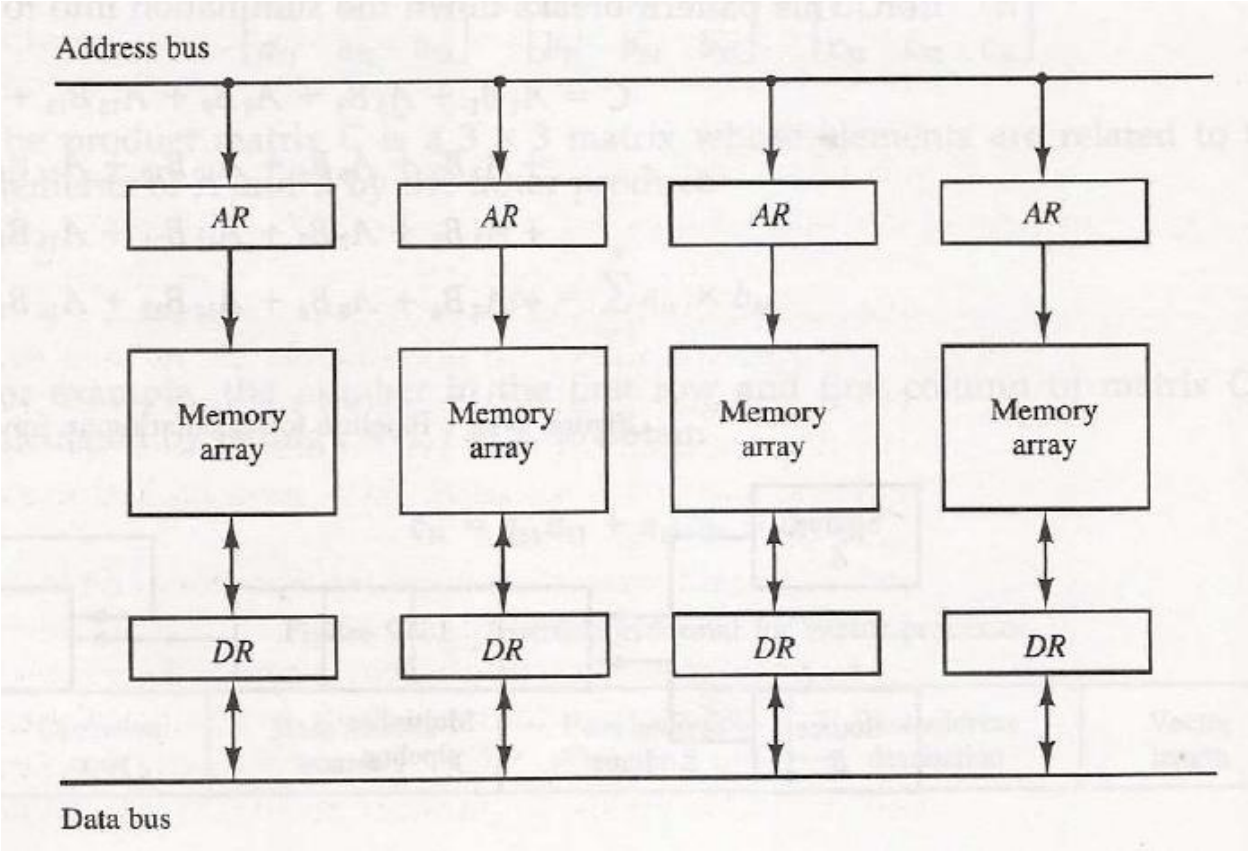


Figure: multiple module memory organization

Array processor

An array processor is a processor that performs computations on large array of data. An attached array processor is an auxiliary processor attached to a general purpose computer, intended to improve the performance of the host computer in specific numerical computation task.

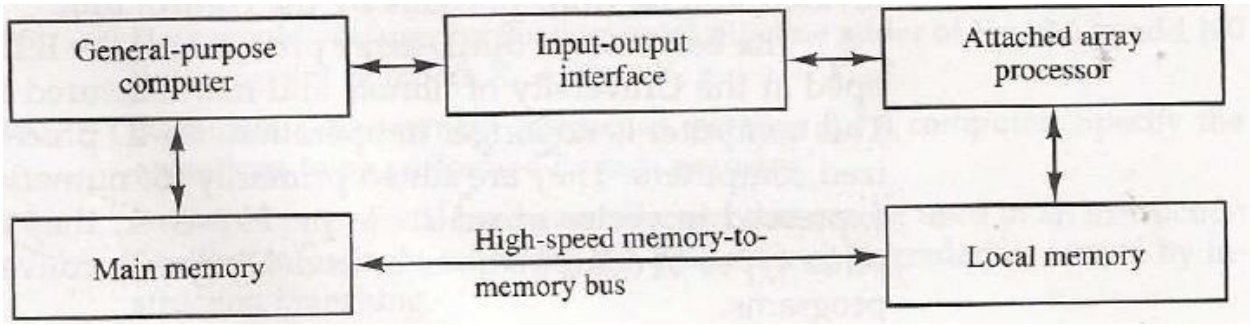


Figure: attached array processor with host computer

Attached array processor

Its purpose is to enhance the performance of the computer by providing vector processing for complex scientific application to a conventional host computer. It achieves high performance by means of parallel processing with multiple functional units, which include arithmetic unit containing one or more pipelined floating point adders and multipliers. The attached array processor can be connected to the back end of the general purpose commercial computers that treats like an external interface.

SIMD array processor

An SIMD array processor is a computer with multiple processing units operating in parallel. The processing units are synchronized to perform the same operation under the control of common control unit, thus providing a single instruction stream with multiple data (SIMD) organization. A block diagram is as shown in figure. It contains a set of identical processing elements (PEs) having its own memory M, each processor elements including ALU, floating point arithmetic unit and working registers. Master control unit controls the operations in processor elements and main memory for storage of programs. Vector instructions are broadcast to all PEs simultaneously each PEs uses operands stored in its local memory. Vector operands are distributed to the local memories prior to the parallel execution of the instruction.

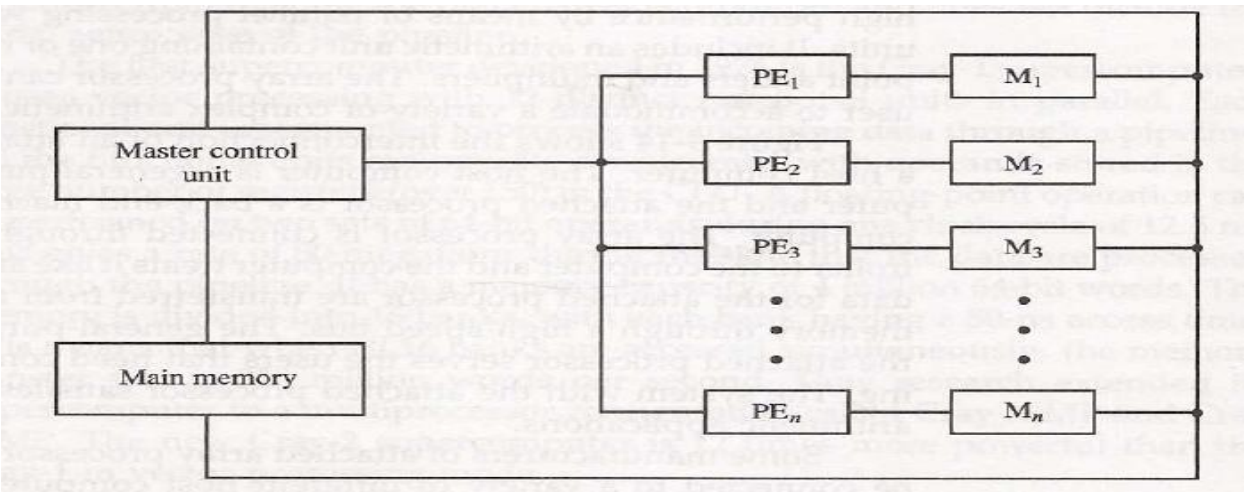


Figure: SIMD Array processor organization

Multithreading

The most important measure of performance for a processor is the rate at which it executes instructions. This can be expressed as

$$MIPS\ rate = f * IPC$$

Where f is the processor clock frequency, in MHz, and *IPC* (instructions per cycle) is the average number of instructions executed per cycle. Accordingly, designers have pursued the goal of increased performance on two fronts: increasing clock frequency and increasing the number of instructions executed or, more properly, the number of instructions that complete during a processor cycle. As designers have increased IPC by using an instruction pipeline and then by using multiple parallel instruction pipelines in a superscalar architecture. With pipelined and multiple-pipeline designs, the principal problem is to maximize the utilization of each pipeline stage. To improve throughput, designers have created ever more complex mechanisms, such as executing some instructions in a different order from the way they occur in the instruction stream and beginning execution of instructions that may never be needed. But this approach may be reaching a limit due to complexity and power consumption concerns.

An alternative approach, which allows for a high degree of instruction-level parallelism without increasing circuit complexity or power consumption, is called ***multithreading.*** In essence, the instruction stream is divided into several smaller streams, known as threads, such that the threads can be executed in parallel. The variety of specific multithreading designs, realized in both commercial systems and experimental systems.

Some of the terms used in multithreading are:

- **Process:** An instance of a program running on a computer. A process embodies two key characteristics:

—**Resource ownership:** A process includes a virtual address space to hold the process image; the process image is the collection of program, data, stack, and attributes that define the process. From time to time, a process may be allocated control or ownership of resources, such as main memory, I/O channels, I/O devices, and files.

—**Scheduling/execution:** The execution of a process follows an execution path (trace) through one or more programs. This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the operating system.

- **Process switch:** An operation that switches the processor from one process to another, by saving all the process control data, registers, and other information for the first and replacing them with the process information for the second

- **Thread:** A dispatchable unit of work within a process. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptible so that the processor can turn to another thread.

- **Thread switch:** The act of switching processor control from one thread to another within the same process. Typically, this type of switch is much less costly than a process switch.

Thus, a thread is concerned with scheduling and execution, whereas a process is concerned with both scheduling/execution and resource ownership. The multiple threads within a process share the same resources. This is why a thread switch is much less time consuming than a process switch. All of the commercial processors and most of the experimental processors so far have used explicit multithreading. These systems concurrently execute instructions from different explicit threads, either by interleaving instructions from different threads on shared pipelines or by parallel execution on parallel pipelines. Implicit multithreading refers to the concurrent execution of multiple threads extracted from a single sequential program. These implicit threads may be defined either statically by the compiler or dynamically by the hardware.

Approaches to Explicit Multithreading

At minimum, a multithreaded processor must provide a separate program counter for each thread of execution to be executed concurrently. The designs differ in the amount and type of additional hardware used to support concurrent thread execution. In general, instruction fetching takes place on a thread basis. The processor treats each thread separately and may use a number of techniques for optimizing single-thread execution, including branch prediction, register renaming, and superscalar techniques.

Broadly speaking, there are four principal approaches to multithreading:

• **Interleaved multithreading:** This is also known as **fine-grained multithreading**. The processor deals with two or more thread contexts at a time, switching from one thread to another at each clock cycle. If a thread is blocked because of data dependencies or memory latencies, that thread is skipped and a ready thread is executed.

• **Blocked multithreading:** This is also known as **coarse-grained multithreading**. The instructions of a thread are executed successively until an event occurs that may cause delay, such as a cache miss. This event induces a switch to another thread. This approach is effective on an in-order processor that would stall the pipeline for a delay event such as a cache miss.

• **Simultaneous multithreading (SMT):** Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor. This combines the wide superscalar instruction issue capability with the use of multiple thread contexts.

• **Chip multiprocessing:** In this case, the entire processor is replicated on a single chip and each processor handles separate threads. The advantage of this approach is that the available logic area on a chip is used effectively without depending on ever-increasing complexity in pipeline design. This is referred to as multi-core.

For the first two approaches, instructions from different threads are not executed simultaneously. Instead, the processor is able to rapidly switch from one thread to another, using a different set of registers and other context information. This results in a better

utilization of the processor's execution resources and avoids a large penalty due to cache misses and other latency events. The SMT approach involves true simultaneous execution of instructions from different threads, using replicated execution resources. Chip multiprocessing also enables simultaneous execution of instructions from different threads. Figure illustrates some of the possible pipeline architectures that involve multithreading and contrasts these with approaches that do not use multithreading. Each horizontal row represents the potential issue slot or slots for a single execution cycle; that is, the width of each row corresponds to the maximum number of instructions that can be issued in a single clock cycle.3 The vertical dimension represents the time sequence of clock cycles. An empty (shaded) slot represents an unused execution slot in one pipeline. A no-op is indicated by N.

The first three illustrations in Figure 17.8 show different approaches with a scalar (i.e., single-issue) processor:

- **Single-threaded scalar:** This is the simple pipeline found in traditional RISC and CISC machines, with no multithreading.
- **Interleaved multithreaded scalar:** This is the easiest multithreading approach to implement. By switching from one thread to another at each clock cycle, the pipeline stages can be kept fully occupied, or close to fully occupied. The hardware must be capable of switching from one thread context to another between cycles.
- **Blocked multithreaded scalar:** In this case, a single thread is executed until a latency event occurs that would stop the pipeline, at which time the processor switches to another thread. Figure 17.8c shows a situation in which the time to perform a thread switch is one cycle, whereas Figure 17.8b shows that thread switching occurs in zero cycles.
- **Superscalar:** This is the basic superscalar approach with no multithreading. Until relatively recently, this was the most powerful approach to providing parallelism within a processor. Note that during some cycles, not all of the available issue slots are used. During these cycles, less than the maximum number of instructions is issued; this is referred to as *horizontal loss*. During other instruction cycles, no issue slots are used; these are cycles when no instructions can be issued; this is referred to as *vertical loss*.
- **Interleaved multithreading superscalar:** During each cycle, as many instructions as possible are issued from a single thread. With this technique, potential delays due to thread switches are eliminated, as previously discussed. However, the number of instructions issued in any given cycle is still limited by dependencies that exist within any given thread.
- **Blocked multithreaded superscalar:** Again, instructions from only one thread may be issued during any cycle, and blocked multithreading is used.
- **Very long instruction word (VLIW):** A VLIW architecture, such as IA-64, places multiple instructions in a single word. Typically, a VLIW is constructed by the compiler, which places operations that may be executed in parallel in the same word. In a simple

VLIW machine (Figure 17.8g), if it is not possible to completely fill the word with instructions to be issued in parallel, no-ops are used.

- **Interleaved multithreading VLIW:** This approach should provide similar efficiencies to those provided by interleaved multithreading on a superscalar architecture.
- **Blocked multithreaded VLIW:** This approach should provide similar efficiencies to those provided by blocked multithreading on a superscalar architecture. The final two approaches illustrated in Figure 17.8 enable the parallel, simultaneous execution of multiple threads:
- **Simultaneous multithreading:** Figure 17.8i shows a system capable of issuing 8 instructions at a time. If one thread has a high degree of instruction-level other cycles, instructions from two or more threads may be issued. If sufficient threads are active, it should usually be possible to issue the maximum number of instructions on each cycle, providing a high level of efficiency.
- **Chip multiprocessor (multicore):** Figure 17.8k shows a chip containing four processors, each of which has a two-issue superscalar processor. Each processor is assigned a thread, from which it can issue up to two instructions per cycle.

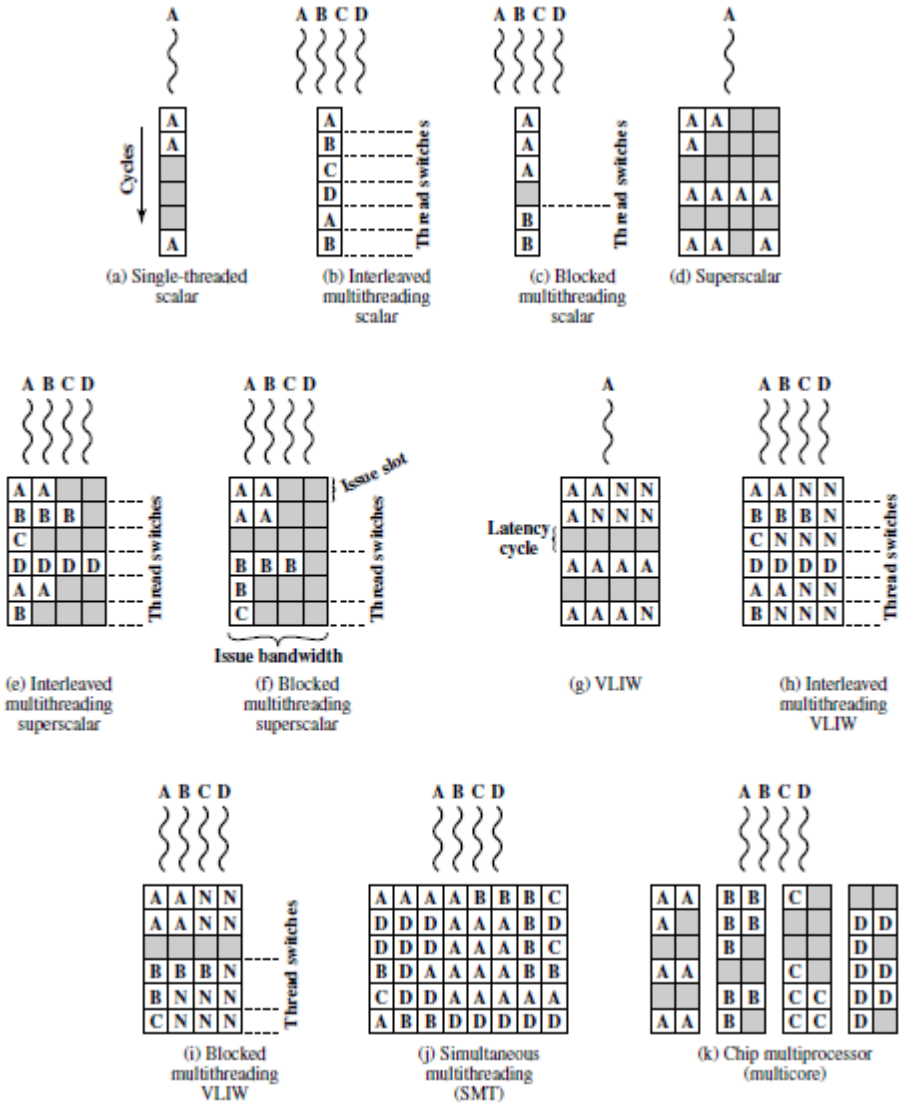


Figure 17.8 Approaches to Executing Multiple Threads

imp

Interconnection structure in multiprocessor system

The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices and a memory unit that may be portioned into a number of separate modules. The interconnection between the components can have different physical configuration depending on the number of transfer paths that are available between the processor and memory in a shared memory system or among the processing elements in a loosely coupled system. Some of these schemes are represented as below:

1. Time shared common bus.
2. Multi-port memory.
3. Crossbar switch.
4. Multi-stage switching network.
5. Hypercube system.

1. Time shared common bus

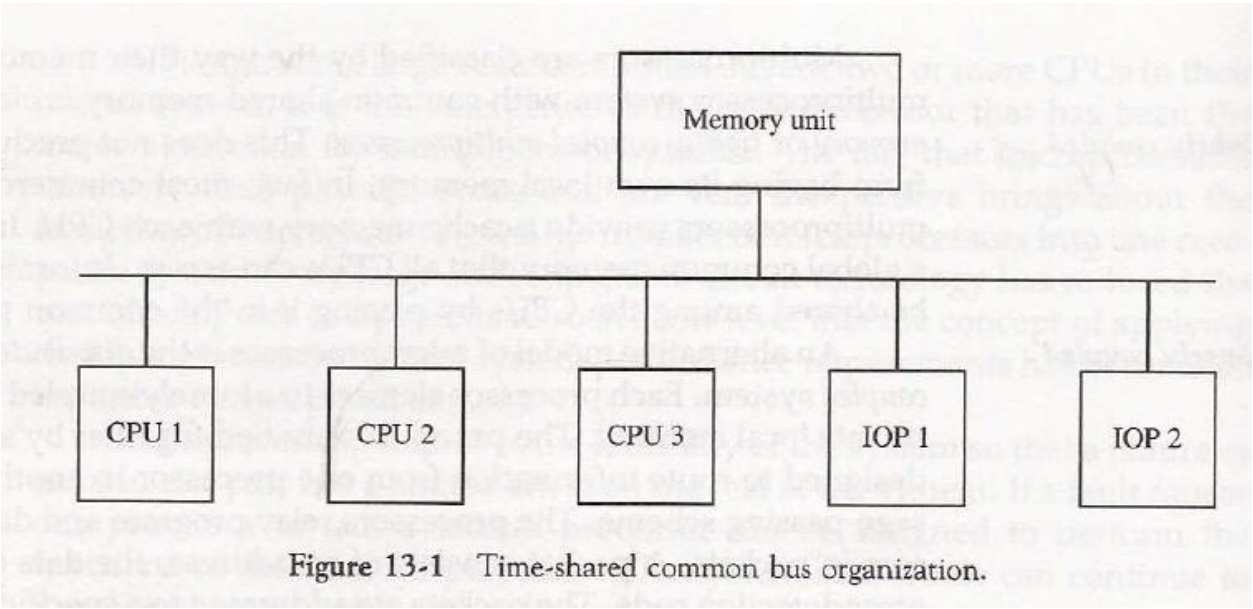


Figure 13-1 Time-shared common bus organization.

- Contains number of multiprocessor connected through a common path to a memory unit. A time shared common bus for five processor is shown as in figure.
- Only one processor can communicate with the memory or another processor at any given time. Transfer operations are conducted by the processor that is in control of the bus at the time.
- Any other processor wishing to initiate a transfer must first determine the availability status of the bus and only after the bus becomes available can process address of the destination unit to initiate the transfer.
- A command is issued to inform the destination unit what operation is to be performed, then receiving unit recognizes its address in the bus and responds to the control signal from sender, after which the transfer is initiated. The conflicts if any must be resolved by incorporating a bus controller that established priorities among the requesting units.
- A single common bus system is restricted to one transfer at a time. This means that when one processor is communicating with memory all other processors are either busy with internal operations or must remain idle for waiting for bus. As consequences the total overflow of the transfer will be limited by the speed of single path.

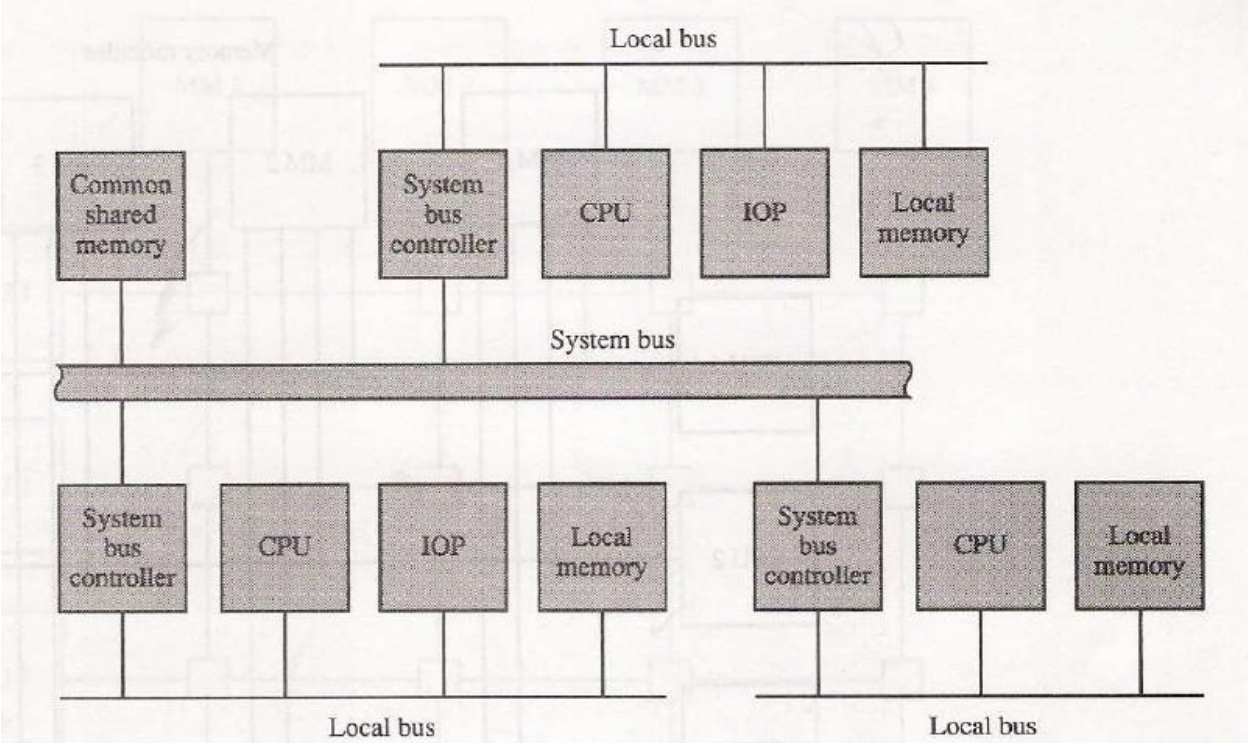


Figure 13-2 System bus structure for multiprocessors.

2. Multiport memory

- A multiport memory system employs separate buses between each memory module and each CPU. As in figure a processor bus consists of address, data and control lines required to communicate with memory. The memory is said to have four ports and each port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time.
- The priority for memory access associated with each processor may be established by the physical port position that is bus occupies in each module. i.e. CPU1 will have priority over CPU2, the CPU3….
- The advantage of multiport memory organization is high transfer rate that can be achieved because of the multiple paths between processor and memory.
- The disadvantage is that it required expensive memory control logic and large number of cables and connectors.

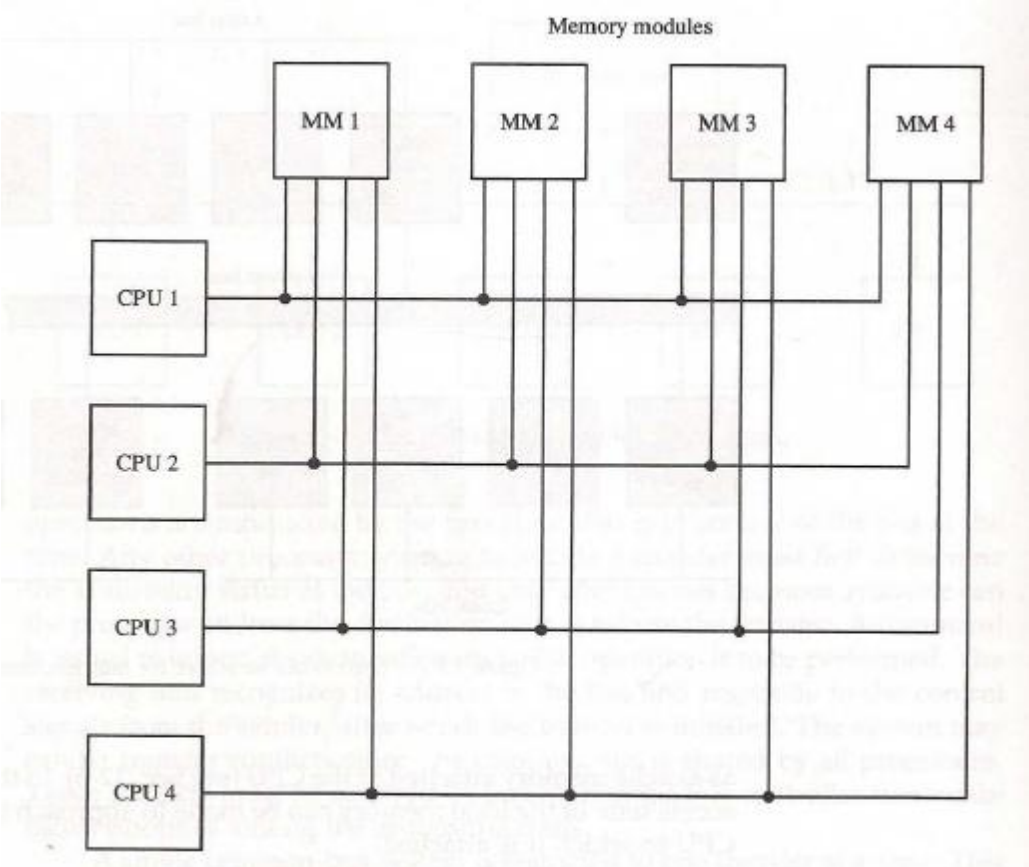


Figure 13-3 Multiport memory organization.

3. Crossbar switch

- The crossbar switch organization consists of a number of cross points that are placed at intersection between processor buses and memory module paths.
- The small square is each cross point is a switch that determines the path from processor to memory module. Each switch point has control logic to set up the transfer path between a processor and memory.
- It examines the address that is placed in the bus to determine whether its particular module is being addressed, it also receives multiple requests for access to same memory module on pre-defined priority basis.

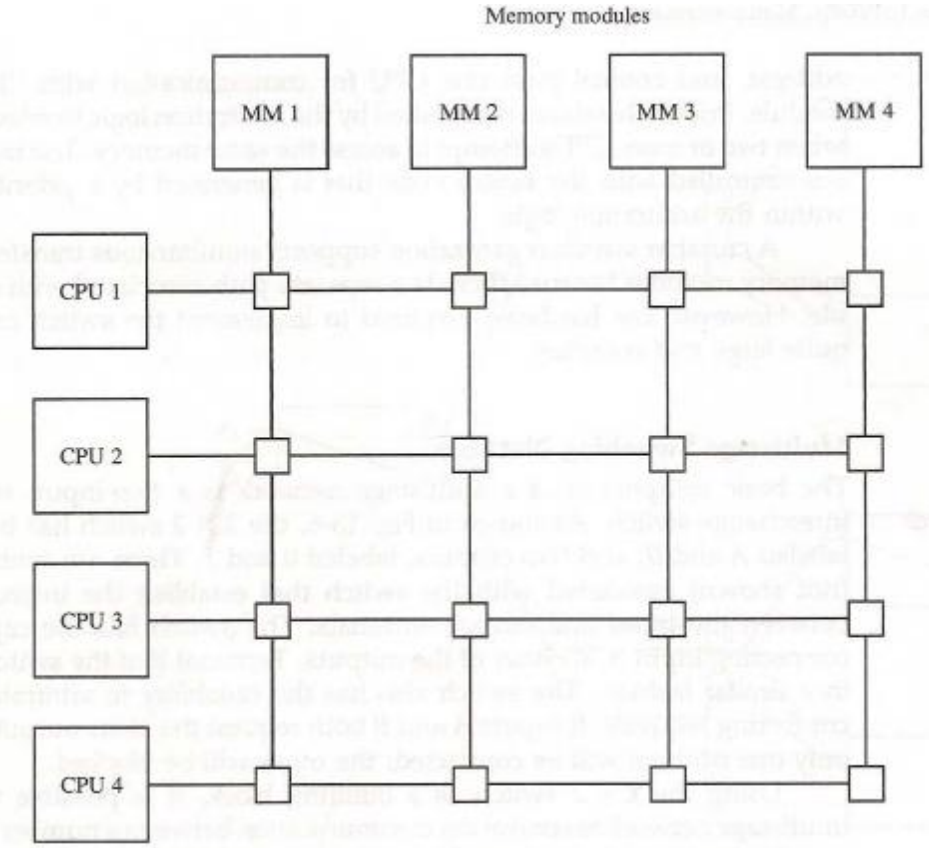
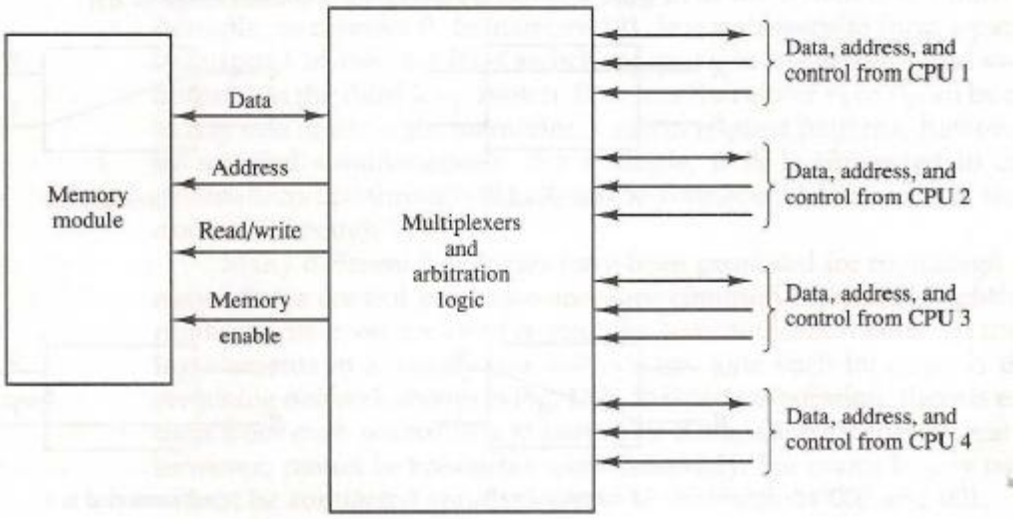


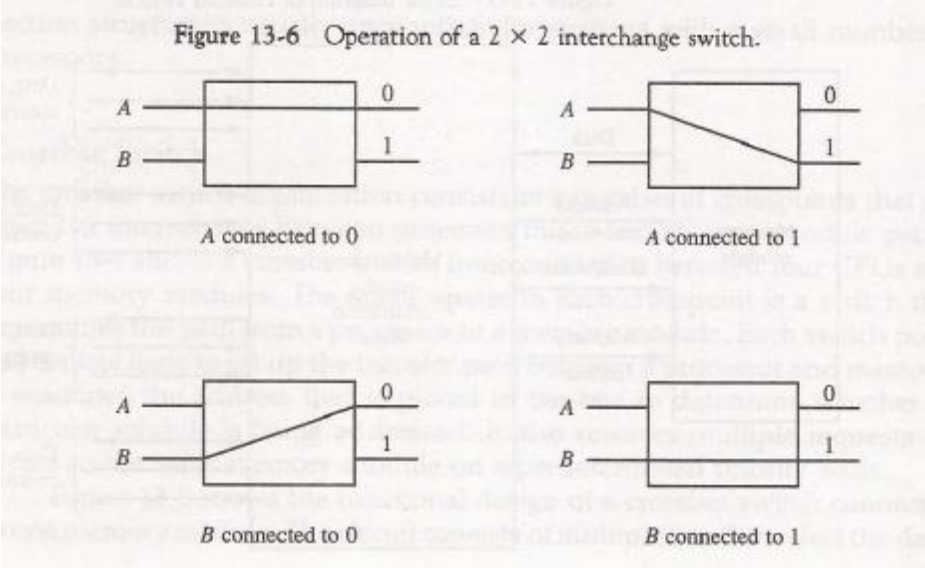
Figure 13-4 Crossbar switch.

Figure 13-5 Block diagram of crossbar switch.



- A crossbar switch organization supports simultaneous transfer from memory module because there is a separate path associated with each module. However the hardware required to implement the switch can become quite large and complex.

4. Multistage switching network



- The basic component of a multistage network is a two-input, two output interchange switch. As in figure switch has two input labeled A and B two outputs 0 & 1. The control signal (not shown) associated with the switch that establishes the interconnection between input and output terminals. The switch has the capability of connecting input A or B to either of the outputs.
- The switch has the capability to arbitrate between conflicting requests i.e. only one will be connected and other will be blocked.
- Figure shows how 2x2 switch can be used as building block. As in figure two processor p1 and p2 are connected through switch to eight memory modules. Marked in binary form 000 through 111. The path from source to destination is determined fro binary bits of the destination number.
- The first bit of the destination number determines the switch output in first level. The second bit specifies the output of the switch in the second level and third bit specifies the output of the switch in third level.
- Example: if P1 is connected to one of the destinations 000 through 011, p2 can be connected to only one of the destinations 100 through 111.
- Many different topologies have been proposed for multistage switching networks to control process memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in loosely couples system. One such topology is the omega switching network in figure.

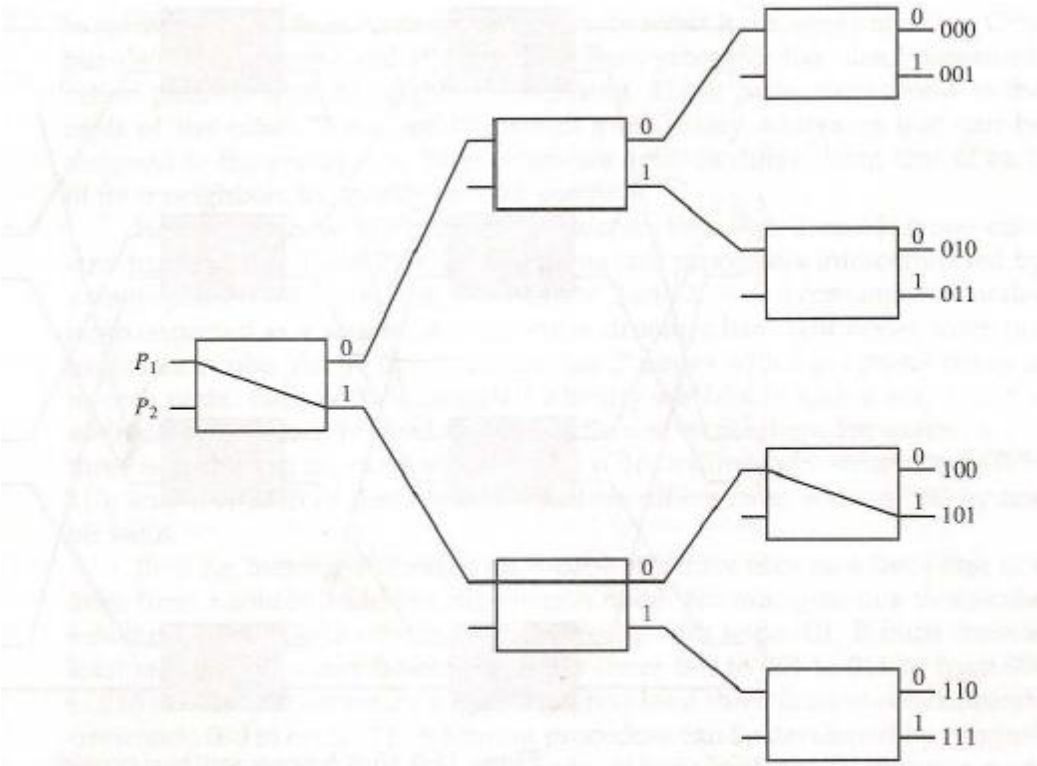


Figure 13-7 Binary tree with 2 × 2 switches.

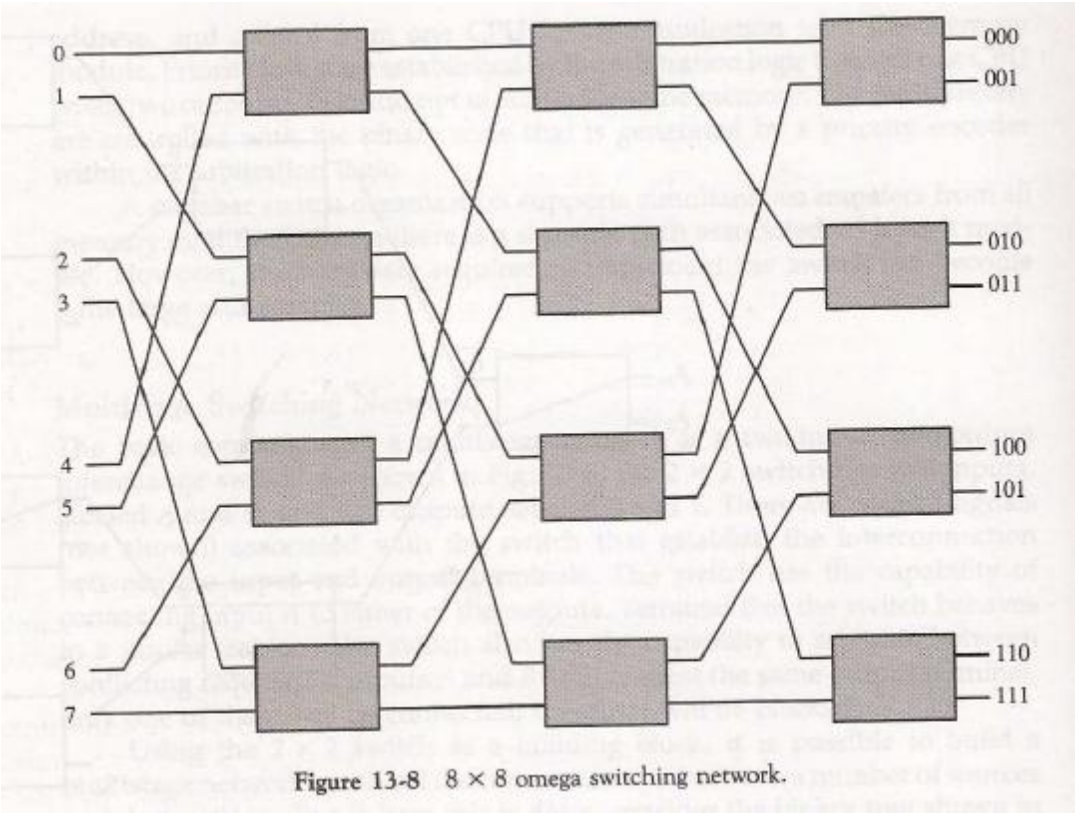


Figure 13-8 8 × 8 omega switching network.

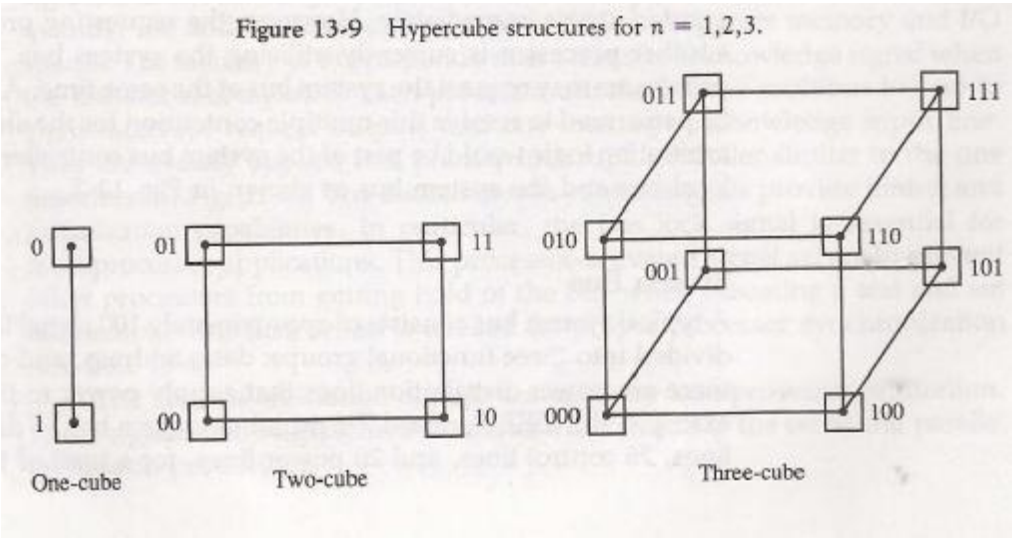
5. Hypercube interconnection

The hyper cube or binary n-cube multi-processor structure is a loosely coupled composed of $N=2^n$ processor interconnected in an n-dimensional binary cube. Each processor from a

node of the cube, although it is customary to refer to ach node as having a processor in effect it contains not only a cpu but also local memory and I/O interface. Each processor has direct communication paths to n other neighbor processor. Figure shows the hypercube structure for n=1 and 2^n=2, for n=2, 2^2=4 nodes connected in a square.

For example the three neighbor of the ode with address 100 in a three cube structure are 000,110 and 101. Each address differs from address 100 by one bit value. Routing message through an n-cube structure may take form one to n links from source node to destination node. For example in a three cube structure node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011(from 000→001→011 or from 000→010→011). It is necessary to go through at least three links to communicate from node 000 to 111. A routing procedure can be developed by computing XOR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along the axes. E.g. in a three cube structure a message at 010 going to 001 produce an XOR of the two address equal to 011. The message can be sent along the second axis to 000 and the through the third axis to 001.

Figure 13-9 Hypercube structures for $n = 1,2,3$.



Chapter 10: Multi-core computers

A multicore computer, also known as a chip multiprocessor, combines two or more processors (called cores) on a single piece of silicon (called a die). Typically, each core consists of all of the components of an independent processor, such as registers, ALU, pipeline hardware, and control unit, plus L1 instruction and data caches. In addition to the multiple cores, contemporary multicore chips also include L2 cache and, in some cases, L3 cache.

HARDWARE PERFORMANCE ISSUES

imp

Microprocessor systems have experienced a steady, exponential increase in execution performance for decades. This increase is due partly to refinements in the organization of the processor on the chip, and partly to the increase in the clock frequency.

Increase in Parallelism

The organizational changes in processor design have primarily been focused on increasing instruction-level parallelism, so that more work could be done in each clock cycle.These changes include, in chronological order:

• Pipelining: Individual instructions are executed through a pipeline of stages so that while one instruction is executing in one stage of the pipeline, another instruction is executing in another stage of the pipeline.

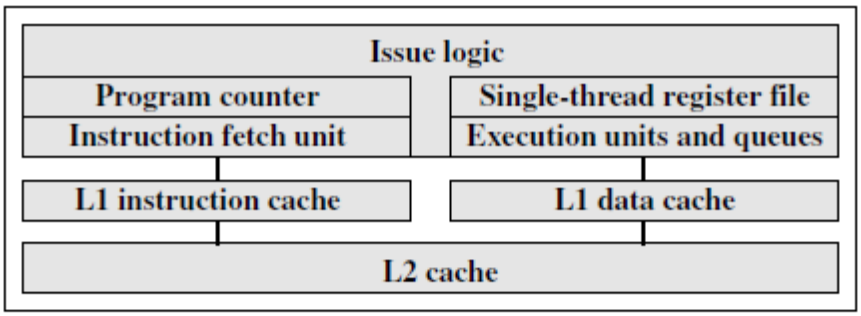
• Superscalar: Multiple pipelines are constructed by replicating execution resources.This enables parallel execution of instructions in parallel pipelines, so long as hazards are avoided.

• Simultaneous multithreading (SMT): Register banks are replicated so that multiple threads can share the use of pipeline resources.

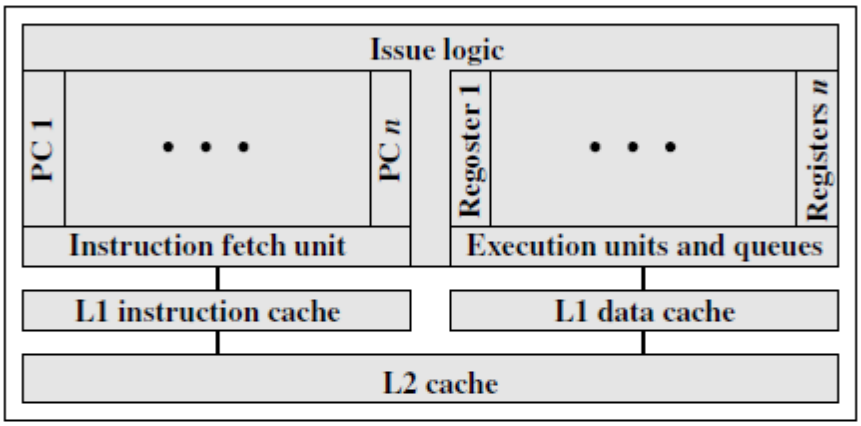
For each of these innovations, designers have over the years attempted to increase the performance of the system by adding complexity. In the case of pipelining, simple three-stage pipelines were replaced by pipelines with five stages, and then many more stages, with some implementations having over a dozen stages. There is a practical limit to how far this trend can be taken, because with more stages, there is the need for more logic, more interconnections, and more control signals.With superscalar organization, performance increases can be achieved by increasing the number of parallel pipelines. Again, there are diminishing returns as the number of pipelines increases. More logic is required to manage hazards and to stage instruction resources. Eventually, a single thread of execution reaches the point where hazards and resource dependencies prevent the full use of the multiple pipelines available. This same point of diminishing returns is reached with SMT, as the complexity of managing multiple threads over a set of pipelines limits the number of threads and number of pipelines that can be effectively utilized.

There is a related set of problems dealing with the design and fabrication of the computer chip. The increase in complexity to deal with all of the logical issues related to very long pipelines, multiple superscalar pipelines, and multiple SMT register banks means that increasing amounts of the chip area is occupied with coordinating and signal transfer logic.

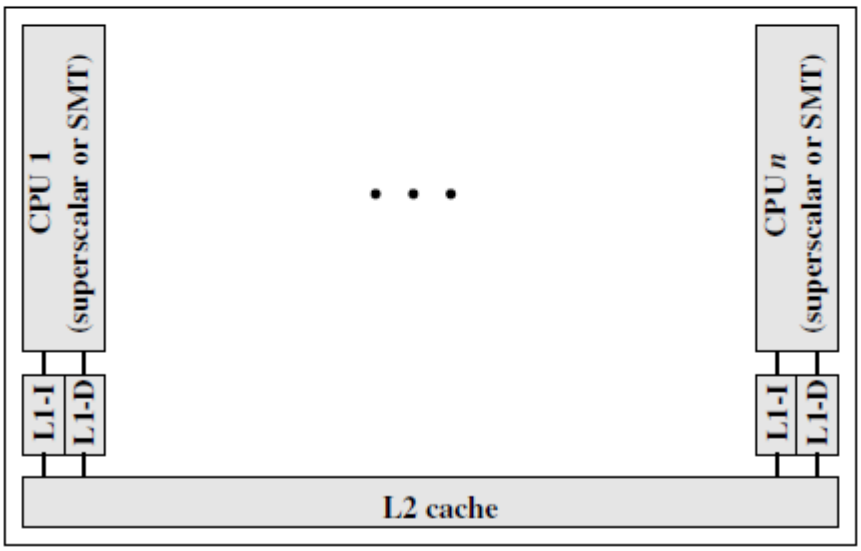
This increases the difficulty of designing, fabricating, and debugging the chips. The increasingly difficult engineering challenge related to processor logic is one of the reasons that an increasing fraction of the processor chip is devoted to the simpler memory logic.



(a) Superscalar



(b) Simultaneous multithreading

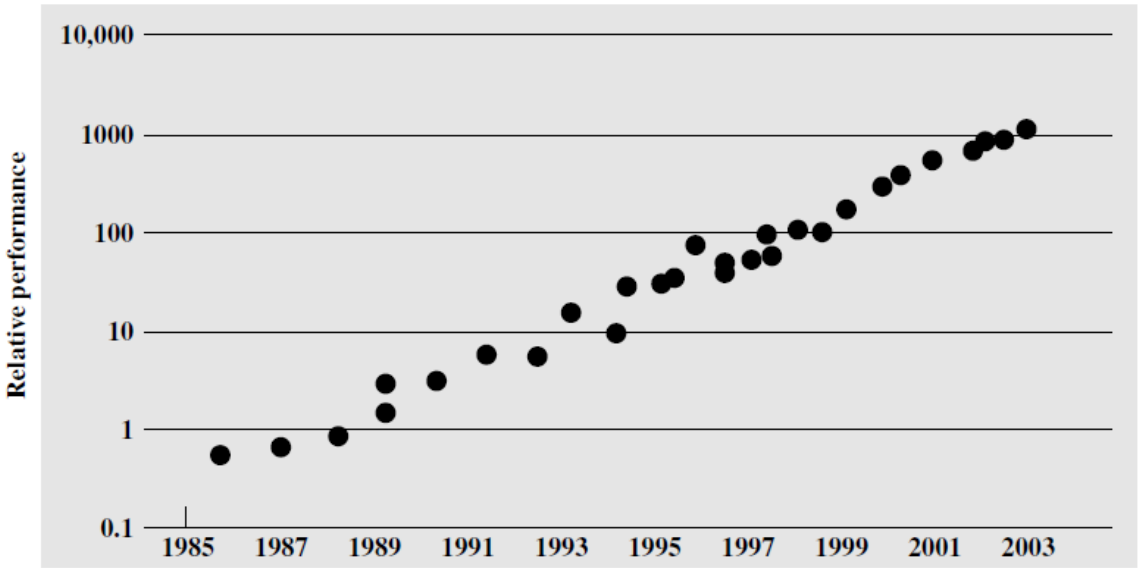


(c) Multicore

Figure 18.1 Alternative Chip Organizations

Power Consumption

To maintain the trend of higher performance as the number of transistors per chip rise, designers have resorted to more elaborate processor designs (pipelining, superscalar, SMT) and to high clock frequencies. Unfortunately, power requirements have grown exponentially as chip density and clock frequency have risen. This is shown in the lowest graph in Figure 18.2. One way to control power density is to use more of the chip area for cache memory. Memory transistors are smaller and have a power density an order of magnitude lower than that of logic (see Figure 18.3a). As Figure 18.3b, from [BORK03], shows, and percentage of the chip area devoted to memory has grown to exceed 50% as the chip transistor density has increased. Figure 18.4, from [BORK07], shows where the power consumption trend is leading. By 2015, we can expect to see microprocessor chips with about 100 billion transistors on a 300 mm2 die. Assuming about 50–60% of the chip area is devoted to memory, the chip will support cache memory of about 100 MB and leave over 1 billion transistors available for logic. How to use all those logic transistors is a key design issue. As discussed earlier in this section, there are limits to the effective use of such techniques as superscalar and SMT. In general terms, the experience of recent decades has been encapsulated in a rule of thumb known as **Pollack's rule** [POLL99], which states that performance increase is roughly proportional to square root of increase in complexity. In other words, if you double the logic in a processor core, then it delivers only 40% more performance. In principle, the use of multiple cores has the potential to provide near-linear performance improvement with the increase in the number of cores. Power considerations provide another motive for moving toward a multicore organization. Because the chip has such a huge amount of cache memory, it becomes unlikely that any one thread of execution can effectively use all that memory. Even with SMT, you are multithreading in a relatively limited fashion and cannot therefore fully exploit a gigantic cache, whereas a number of relatively independent threads or processes have a greater opportunity to take full advantage of the cache memory.



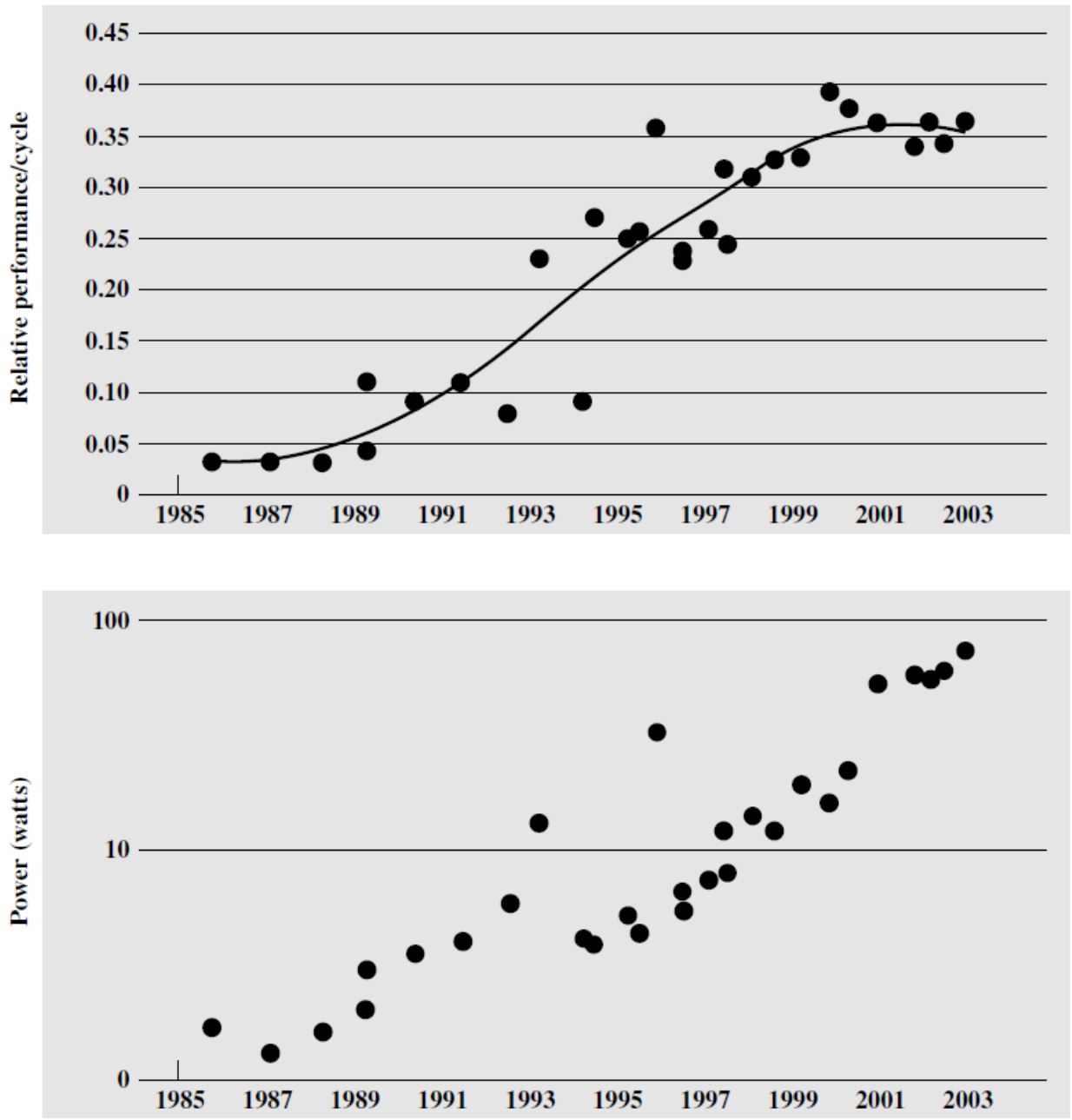


Figure 18.2 Some Intel Hardware Trends

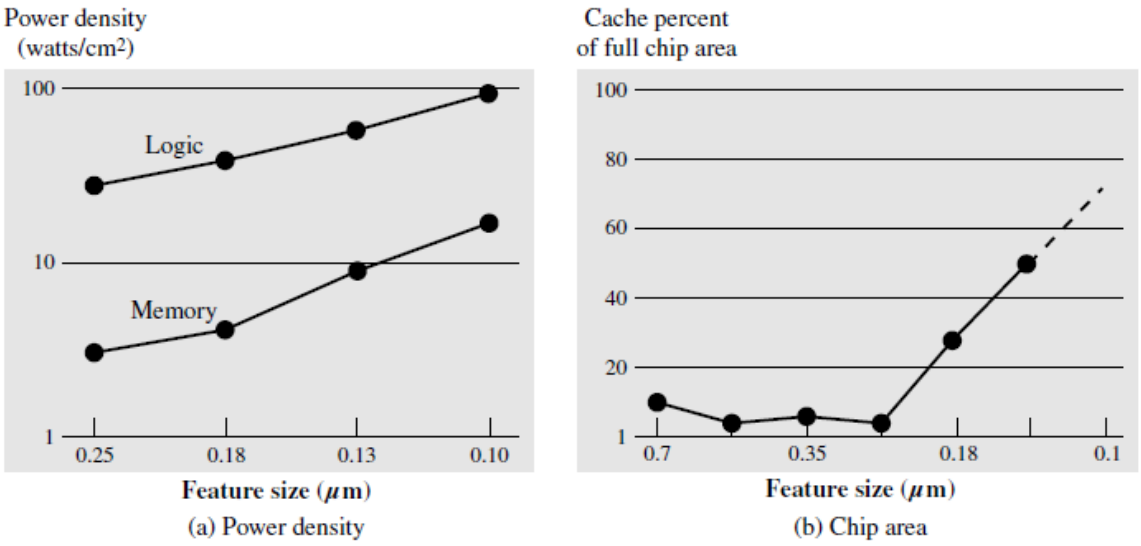


Figure 18.3 Power and Memory Considerations

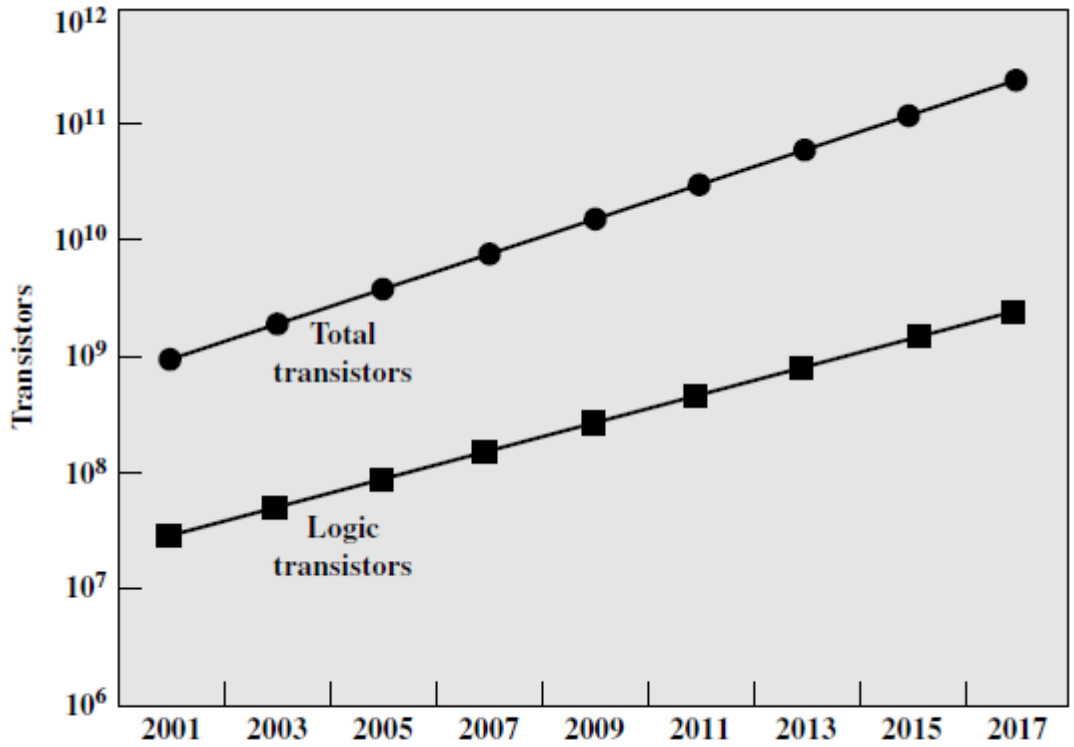


Figure 18.4 Chip Utilization of Transistors

Software performance issues

imp

Software on Multicore

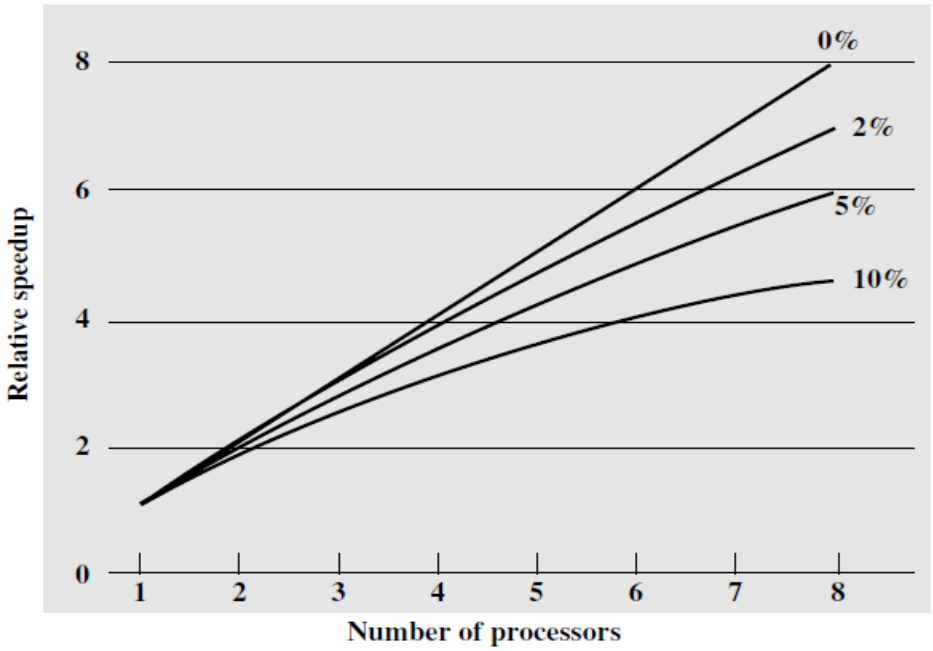
The potential performance benefits of a multicore organization depend on the ability to effectively exploit the parallel resources available to the application. Let us focus first on a single application running on a multicore system. Amdahl's law states that or speed up ratio:

$$speed\ up = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on N parallel processors}}$$

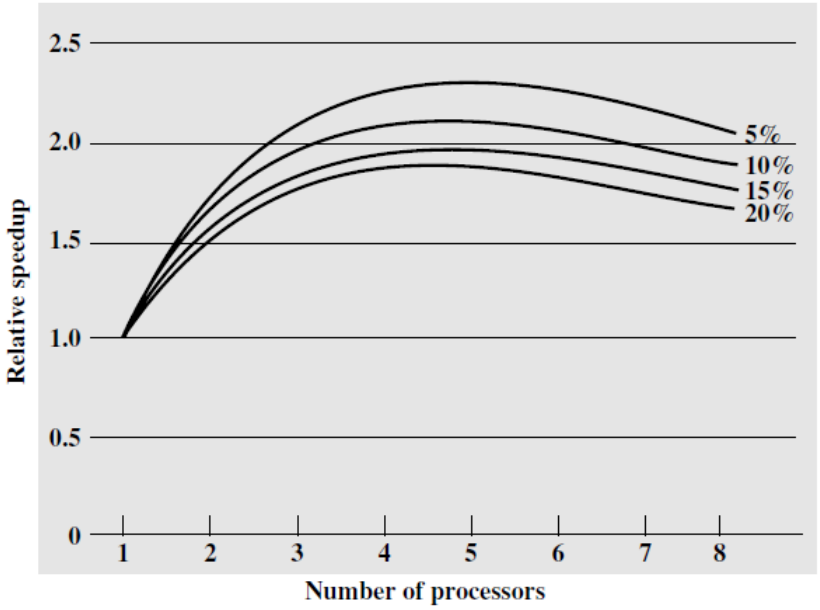
$$speed\ up = \frac{1}{(1 - \mathrm{f}) + \mathrm{f/n}}$$

The law assumes a program in which a fraction (1-f) of the execution time involves code that is inherently serial and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead.

This law appears to make the prospect of a multicore organization attractive. But as Figure 18.5a shows, even a small amount of serial code has a noticeable impact. If only 10% of the code is inherently serial the running the program on a multicore system with 8 processors yields a performance gains of only a factor of 4.7. In addition, software typically incurs overhead as a result of communication and distribution of work to multiple processors and cache coherence overhead. This results in a curve where performance peaks than then begins to degrade because of the increased burden of the overhead of using multiple processors. Figure 18.5b, from [MCDO05], is a representative example.



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions



(b) Speedup with overheads

Figure 18.5 Performance Effect of Multiple Cores

However, software engineers have been addressing this problem and there are numerous applications in which it is possible to effectively exploit a multicore system. [MCDO05] reports on a set of database applications, in which great attention was paid to reducing the serial fraction within hardware architectures, operating systems, middleware, and the database application software. Figure 18.6 shows the result. As this example shows, database management systems and database applications are one area in which multicore systems can be used effectively. Many kinds of servers can also effectively use the parallel multicore organization, because servers typically handle numerous relatively independent transactions in parallel.

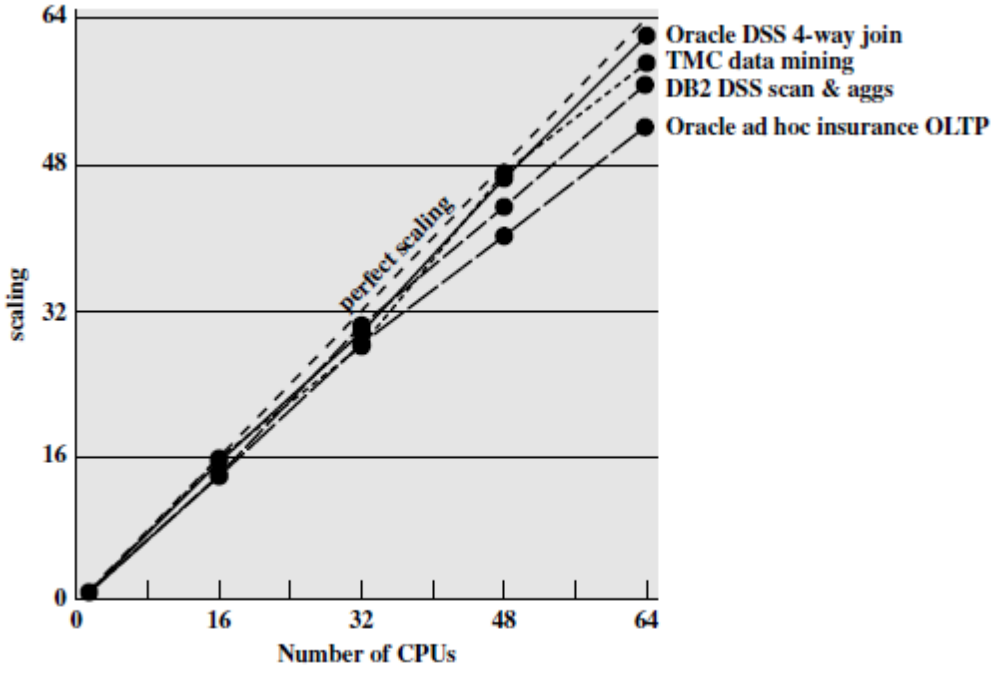


Figure 18.6 Scaling of Database Workloads on Multiple-Processor Hardware

In addition to general-purpose server software, a number of classes of applications benefit directly from the ability to scale throughput with the number of cores. Lists the following examples:

• **Multithreaded native applications:** Multithreaded applications are characterized by having a small number of highly threaded processes. Examples of threaded applications include Lotus Domino or Siebel CRM (Customer Relationship Manager).

• **Multiprocess applications:** Multiprocess applications are characterized by the presence of many single-threaded processes. Examples of multi-process applications include the Oracle database, SAP, and PeopleSoft.

• **Java applications:** Java applications embrace threading in a fundamental way. Not only does the Java language greatly facilitate multithreaded applications, but the Java Virtual Machine is a multithreaded process that provides scheduling and memory management for Java applications.

• **Multiinstance applications:** Even if an individual application does not scale to take advantage of a large number of threads, it is still possible to gain from multicore architecture by running multiple instances of the application in parallel. If multiple application instances require some degree of isolation, virtualization technology (for the hardware of the operating system) can be used to provide each of them with its own separate and secure environment.
Application Example: Valve Game Software

imp

Multi core organization

At a top level of description, the main variables in a multicore organization are as follows:

• The number of core processors on the chip

• The number of levels of cache memory

• The amount of cache memory that is shared

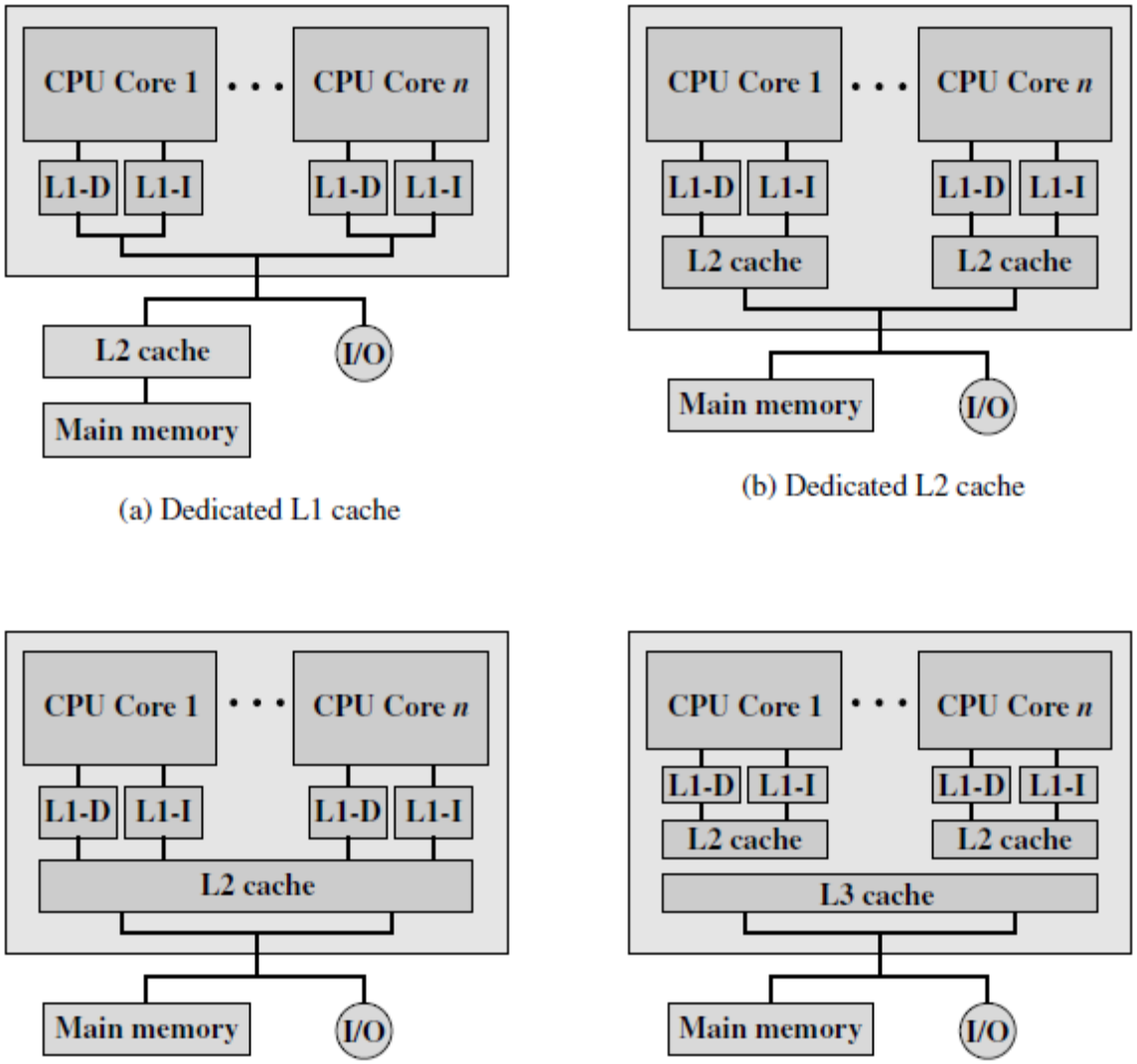


Figure 18.8 Multicore Organization Alternatives

Figure 18.8 shows four general organizations for multicore systems. Figure 18.8a is an organization found in some of the earlier multicore computer chips and is still seen in embedded chips. In this organization, the only on-chip cache is L1 cache, with each core having its own dedicated L1 cache. Almost invariably, the L1 cache is divided into instruction and data caches. An example of this organization is the ARM11 MPCore. The organization of Figure 18.8b is also one in which there is no on-chip cache sharing. In this, there is enough area available on the chip to allow for L2 cache. An example of this organization is the AMD Opteron. Figure 18.8c shows a similar allocation of chip space to memory, but with the use of a shared L2 cache. The Intel Core Duo has this organization. Finally, as the amount of cache memory available on the chip continues to grow, performance considerations dictate splitting off a separate, shared L3 cache, with dedicated L1 and L2 caches for each core processor. The Intel Core i7 is an example of this organization. The use of a shared L2 cache on the chip has several advantages over exclusive reliance on dedicated caches:

1. Constructive interference can reduce overall miss rates. That is, if a thread on one core accesses a main memory location, this brings the frame containing the referenced location

into the shared cache. If a thread on another core soon thereafter accesses the same memory block, the memory locations will already be available in the shared on-chip cache.

2. A related advantage is that data shared by multiple cores is not replicated at the shared cache level.

3. With proper frame replacement algorithms, the amount of shared cache allocated to each core is dynamic, so that threads that have a less locality can employ more cache.

4. Interprocessor communication is easy to implement, via shared memory locations.

5. The use of a shared L2 cache confines the cache coherency problem to the L1 cache level, which may provide some additional performance advantage.

A potential advantage to having only dedicated L2 caches on the chip is that each core enjoys more rapid access to its private L2 cache. This is advantageous for threads that exhibit strong locality. As both the amount of memory available and the number of cores grow, the use of a shared L3 cache combined with either a shared L2 cache or dedicated per core L2 caches seems likely to provide better performance than simply a massive shared L2 cache. Another organizational design decision in a multicore system is whether the individual cores will be superscalar or will implement simultaneous multithreading (SMT). For example, the Intel Core Duo uses superscalar cores, whereas the Intel Core i7 uses SMT cores. SMT has the effect of scaling up the number of hardware level threads that the multicore system supports. Thus, a multicore system with four cores and SMT that supports four simultaneous threads in each core appears the same to the application level as a multicore system with 16 cores. As software is developed to more fully exploit parallel resources, an SMT approach appears to be more attractive than a superscalar approach.

Intel Core Duo

The Intel Core Duo, introduced in 2006, implements two x86 superscalar processors with a shared L2 cache (Figure 18.8c). The general structure of the Intel Core Duo is shown in Figure 18.9. Let us consider the key elements starting from the top of the figure. As is common in multicore systems, each core has its own dedicated **L1 cache**. In this case, each core has a 32-KB instruction cache and a 32-KB data cache. Each core has an independent **thermal control unit**. With the high transistor density of today's chips, thermal management is a fundamental capability, especially for laptop and mobile systems. The Core Duo thermal control unit is designed to manage chip heat dissipation to maximize processor performance within thermal constraints. Thermal management also improves ergonomics with a cooler system and lower fan acoustic noise. In essence, the thermal management unit monitors digital sensors for high-accuracy die temperature measurements. Each core can be defined as in independent thermal zone. The maximum temperature for each thermal zone is reported separately via dedicated registers that can be polled by software. If the temperature in a core exceeds a threshold, the thermal control unit reduces the clock rate for that core to reduce heat generation. The next key element of the Core Duo organization is the **Advanced Programmable Interrupt Controller** (APIC). The APIC performs a number of functions, including the following:

1. TheAPIC can provide interprocessor interrupts, which allow any process to interrupt any other processor or set of processors. In the case of the Core Duo, a thread in one core can generate an interrupt, which is accepted by the local APIC, routed to the APIC of the other core, and communicated as an interrupt to the other core.

2. The APIC accepts I/O interrupts and routes these to the appropriate core.

3. Each APIC includes a timer, which can be set by the OS to generate an interrupt to the local core.

The **power management logic** is responsible for reducing power consumption when possible, thus increasing battery life for mobile platforms, such as laptops. In essence, the power management logic monitors thermal conditions and CPU activity and adjusts voltage levels and power consumption appropriately. It includes an advanced power-gating capability that allows for an ultra fine-grained logic control that turns on individual processor logic subsystems only if and when they are needed. Additionally, many buses and arrays are split so that data required in some modes of operation can be put in a low power state when not needed. The Core Duo chip includes a shared 2-MB **L2 cache**. The cache logic allows for a dynamic allocation of cache space based on current core needs, so that one core can be assigned up to 100% of the L2 cache. The L2 cache includes logic to support the MESI protocol for the attached L1 caches. The key point to consider is when a cache write is done at the L1 level. A cache line gets the M state when a processor writes to it; if the line is not in E or M-state prior to writing it, the cache sends a Read-For-Ownership (RFO) request that ensures that the line exists in the L1 cache and is in the I state in the other L1 cache. The Intel Core Duo extends this protocol to take into account the case when there is multiple Core Duo chips organized as a symmetric multiprocessor (SMP) system. The L2 cache controller allow the system to distinguish between a situation in which data are shared by the two local cores, but not with the rest of the world, and a situation in which the data are shared by one or more caches on the die as well as by an agent on the external bus (can be another processor).When a core issues an RFO, if the line is shared only by the other cache within the local die, we can resolve the RFO internally very fast, without going to the external bus at all. Only if the line is shared with another agent on the external bus do we need to issue the RFO externally.

The **bus interface** connects to the external bus, known as the Front Side Bus, which connects to main memory, I/O controllers, and other processor chips.

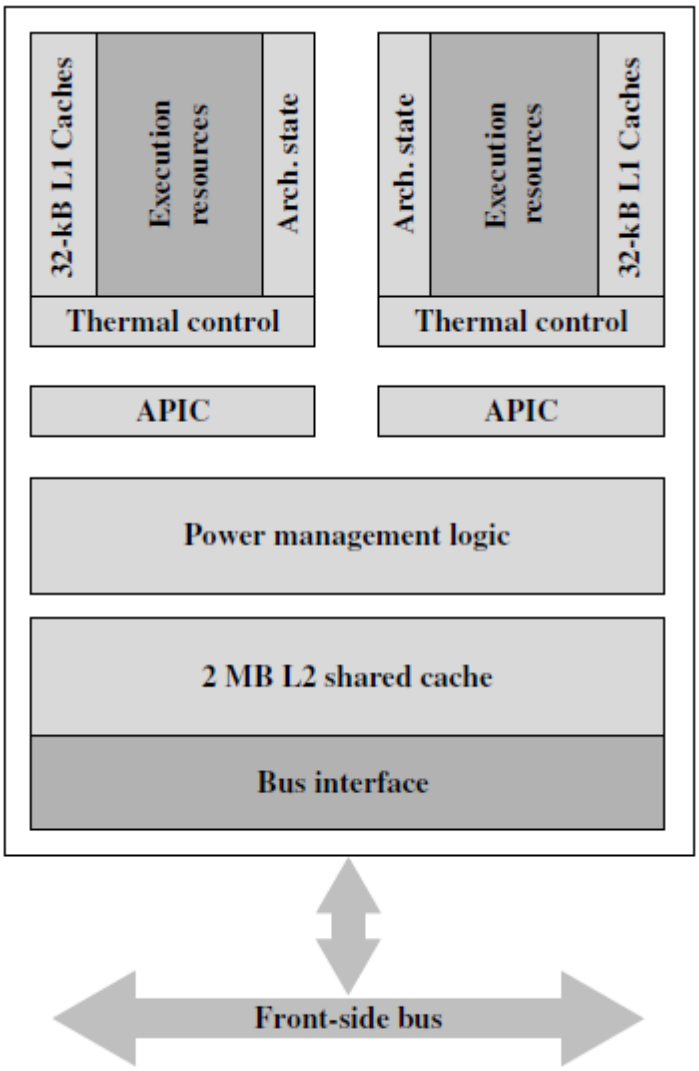


Figure 18.9 Intel Core Duo Block Diagram

Intel Core i7

The Intel Core i7, introduced in November of 2008, implements four x86 SMT processors, each with a dedicated L2 cache, and with a shared L3 cache (Figure 18.8d). The general structure of the Intel Core i7 is shown in Figure 18.10. Each core has its own **dedicated L2 cache** and the four cores share an 8-MB **L3 cache**. One mechanism Intel uses to make its caches more effective is prefetching, in which the hardware examines memory access patterns and attempts to fill the caches speculatively with data that's likely to be requested soon. It is interesting to compare the performance of this three-level on chip cache organization with a comparable two level organization from Intel. Table 18.1 shows the cache access latency, in terms of clock cycles for two Intel multicore systems running at the same clock frequency. The Core 2 Quad has a shared L2 cache, similar to the Core Duo. The Core i7 improves on L2 cache performance with the use of the dedicated L2 caches, and provides a relatively high-speed access to the L3 cache. The Core i7 chip supports two forms of external

communications to other chips. The **DDR3 memory controller** brings the memory controller for the DDR main memory2 onto the chip. The interface supports three channels that are 8 bytes wide for a total bus width of 192 bits, for an aggregate data rate of up to 32 GB/s. With the memory controller on the chip, the Front Side Bus is eliminated. The **Quick Path Interconnect** (QPI) is a cache-coherent, point-to-point link based electrical interconnect specification for Intel processors and chipsets. It enables high-speed communications among connected processor chips. The QPI link operates at 6.4 GT/s (transfers per second). At 16 bits per transfer, that adds up to 12.8 GB/s, and since QPI links involve dedicated bidirectional pairs, the total bandwidth is 25.6 GB/s.

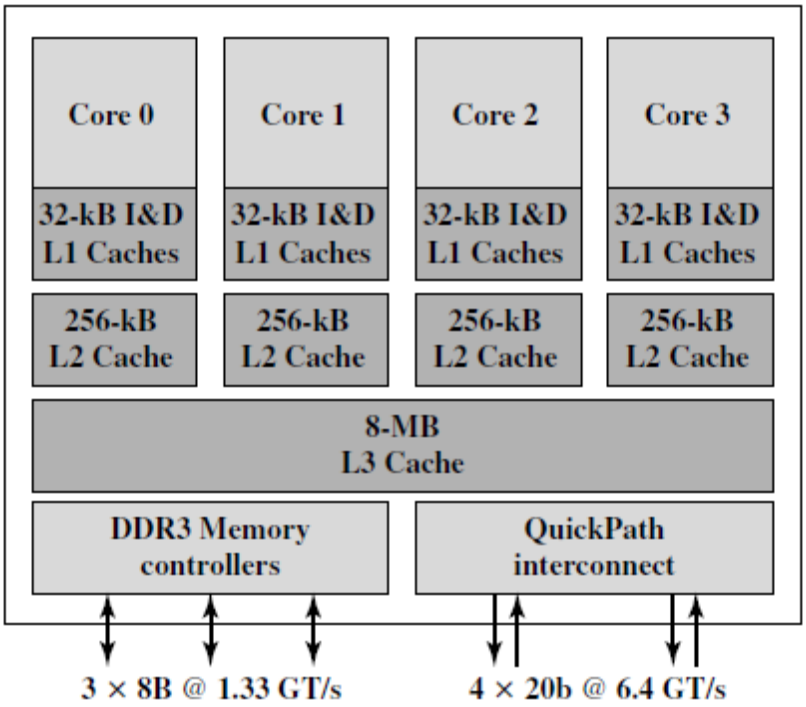


Figure 18.10 Intel Core i7 Block Diagram

The ARM11 MPCore is a multicore product based on the ARM11 processor family. The ARM11 MPCore can be configured with up to four processors, each with its own L1 instruction and data caches, per chip. Table 18.1 lists the configurable options for the system, including the default values. Figure 18.11 presents a block diagram of the ARM11 MPCore. The key elements of the system are as follows:

• **Distributed interrupt controller (DIC):** Handles interrupt detection and interrupt prioritization. The DIC distributes interrupts to individual processors.

• **Timer:** Each CPU has its own private timer that can generate interrupts.

• **Watchdog:** Issues warning alerts in the event of software failures. If the watchdog is enabled, it is set to a predetermined value and counts down to 0. It is periodically reset. If the watchdog value reaches zero, an alert is issued.

• **CPU interface:** Handles interrupt acknowledgement, interrupt masking, and interrupt completion acknowledgement

• **CPU:** A single ARM11 processor. Individual CPUs are referred to as **MP11 CPUs**.

• **Vector floating-point (VFP) unit:** A coprocessor that implements floating point operations in hardware.

• **L1 cache:** Each CPU has its own dedicated L1 data cache and L1 instruction cache.

• **Snoop control unit (SCU):** Responsible for maintaining coherency among L1

data caches.

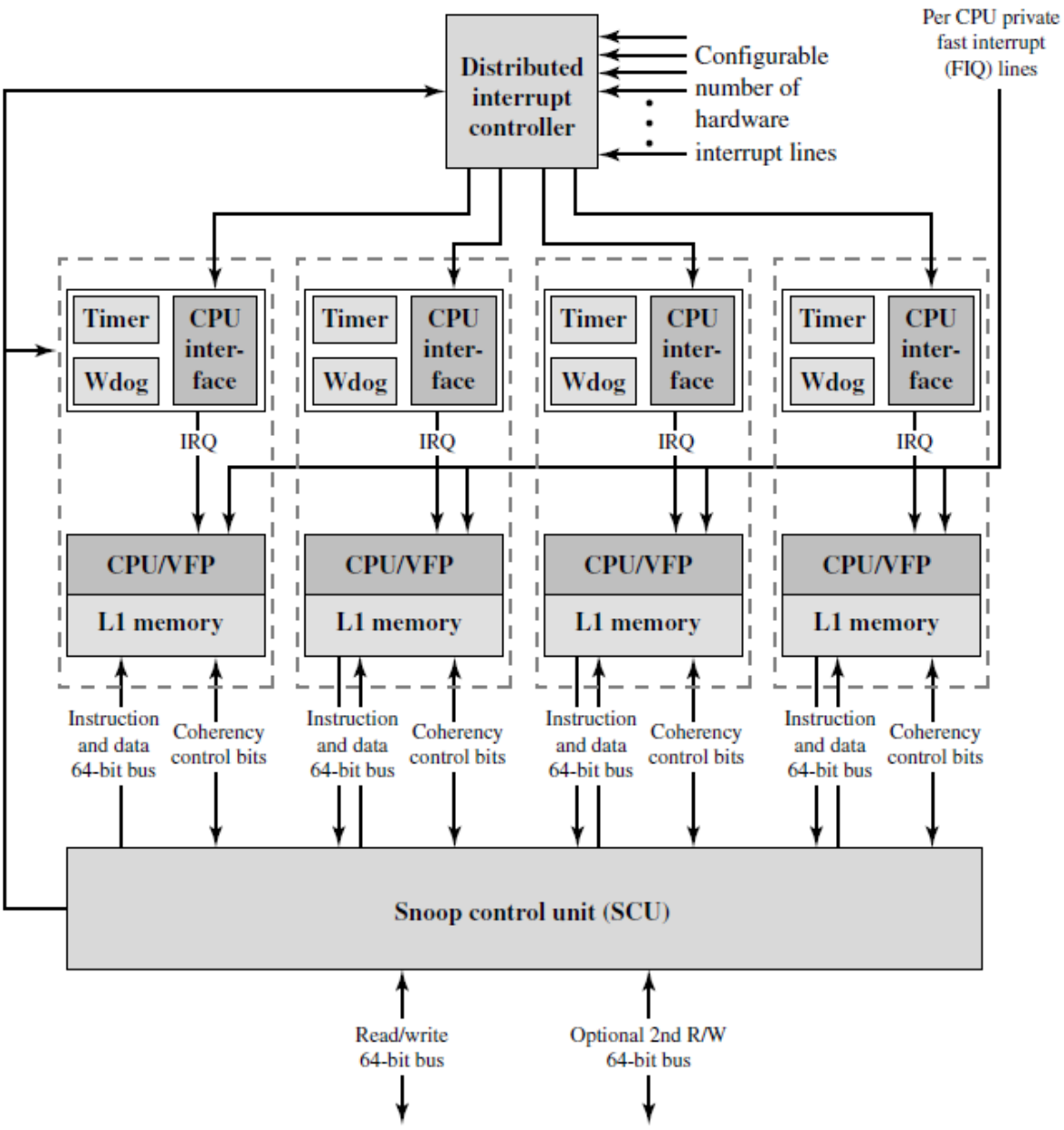


Figure 18.11 ARM11 MPCore Processor Block Diagram