Unit 3 - Basic Processing Unit

Dr.A.Parivazhagan / Associate Professor / CSE /KARE

==========

OBJECTIVES:

In this lesson, you will learn about Execution of instructions by a processor, the functional units of a processor and how they are interconnected, hardware for generating control signals and microprogrammed control and fixed point and floating-point arithmetic for ALU operation such as adder and subtractor circuits, high-speed adders based on carry-lookahead logic circuits, the Booth algorithm for multiplication of signed numbers, logic circuits for division and arithmetic operations on floating-point numbers conforming to the IEEE standard

CONTENTS:

1. Fundamental Concepts
    - Execution of a complete instruction, Multiple Bus Organization, Hardwired control, Micro-programmed control
2. Computer Arithmetic
    - Addition and Subtraction, Multiplication Algorithm,Division Algorithm
3. Floating Point Arithmetic operations
    - Decimal Arithmetic Unit, Decimal Arithmetic Operations

Introduction

First, we focus on the processing unit, which executes machine-language instructions and coordinates the activities of other units in a computer. We examine its internal structure and show how it performs the tasks of fetching, decoding, and executing such instructions. The processing unit is often called the central processing unit (CPU). Addition and subtraction of two numbers are basic operations at the machine-instruction level in all computers. And arithmetic and logic operations, are implemented in ALU of the processor.

1. FUNDAMENTAL CONCEPTS

A typical computing task consists of a series of operations specified by a sequence of machine-language instructions that constitute a program. The processor fetches one instruction at a time and performs the operation specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered. The processor uses the program counter, PC, to keep track of the address of the next instruction to be fetched and executed. After fetching an instruction, the contents of the PC are updated to point to the next instruction in sequence. A branch instruction may cause a different value to be loaded into the PC.

When an instruction is fetched, it is placed in the instruction register, IR, from where it is interpreted, or decoded, by the processor's control circuitry. The IR holds the instruction until its execution is completed. Consider a 32-bit computer in which each instruction is contained in one word in the memory, as in RISC-style instruction set architecture. To execute an instruction, the processor has to perform the following steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR. In register transfer notation, the required action is

$$IR \leftarrow [[PC]]$$

2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is

$$PC \leftarrow [PC] + 4$$

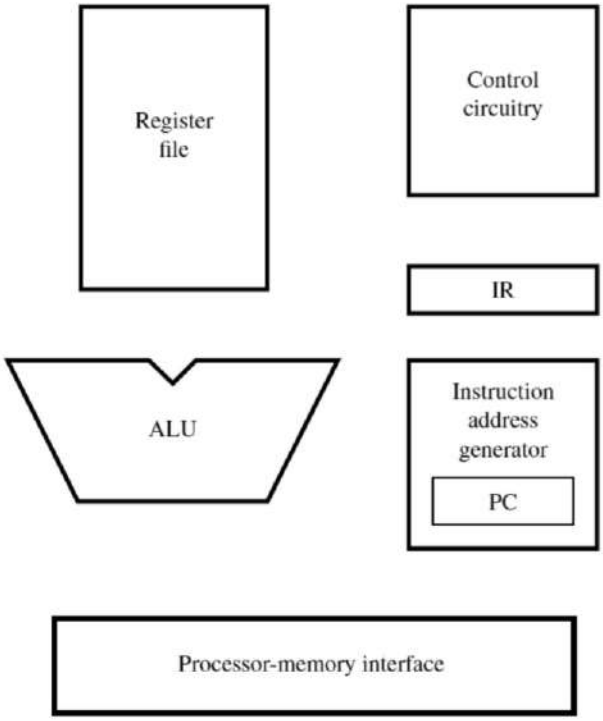
3. Carry out the operation specified by the instruction in the IR.

Fetching an instruction and loading it into the IR is usually referred to as the instruction fetch phase. Performing the operation specified in the instruction constitutes the instruction execution phase. With few exceptions, the operation specified by an instruction can be carried out by performing one or more of the following actions:

- Read the contents of a given memory location and load them into a processor register.
- Read data from one or more processor registers.
- Perform an arithmetic or logic operation and place the result into a processor register.
- Store data from a processor register into a given memory location.

The hardware components needed to perform these actions are shown in Figure. The processor communicates with the memory through the processor-memory interface, which transfers data from and to the memory during Read and Write operations. The instruction

address generator updates the contents of the PC after every instruction is fetched. The register file is a memory unit whose storage locations are organized to form the processor's general-purpose registers. During execution, the contents of the registers named in an instruction that performs an arithmetic or logic operation are sent to the arithmetic and logic unit (ALU), which performs the required computation. The results of the computation are stored in a register in the register file.



EXECUTION OF A COMPLETE INSTRUCTION

Atypical computation operates on data stored in registers. These data are processed by combinational circuits, such as adders, and the results are placed into a register. A clock signal is used to control the timing of data transfers. The registers comprise edge-triggered flip-flops into which new data are loaded at the active edge of the clock. In this chapter, we assume that the rising edge of the clock is the active edge. The clock period, which is the time between two successive rising edges, must be long enough to allow the combinational circuit to produce the correct result. Let us now examine the actions involved in fetching and executing instructions. We illustrate these actions using a few representative RISC-style instructions

SINGLE BUS ORGANIZATION

ALU and all the registers are interconnected via a Single Common Bus. Data & address lines of the external memory-bus is connected to the internal processor-bus via MDR

and MAR respectively. (MDRà Memory Data Register, MAR à Memory Address Register). MDR has 2 inputs and 2 outputs. Data may be loaded into MDR either from memory-bus (external) or from processor-bus (internal).

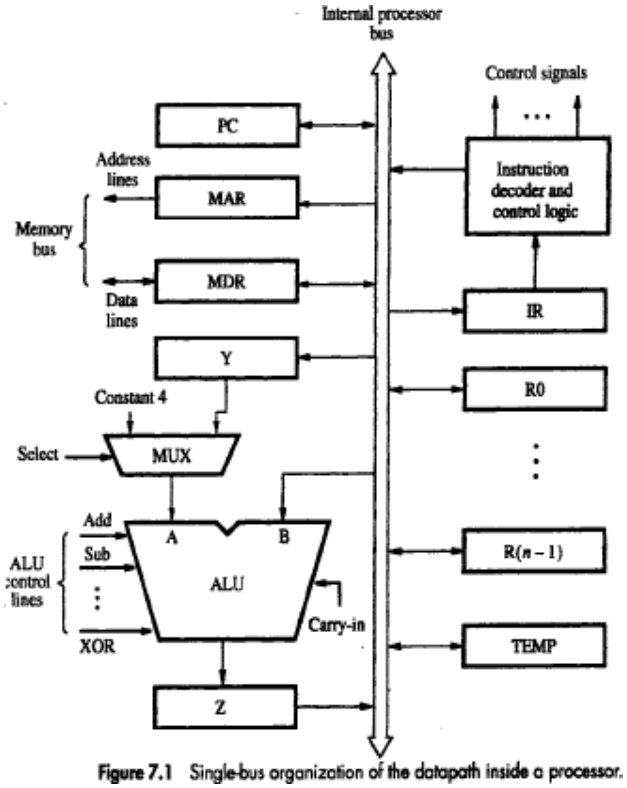


Figure 7.1 Single-bus organization of the datapath inside a processor.

MAR‟s input is connected to internal-bus; MAR‟s output is connected toexternal-bus. Instruction Decoder & Control Unit is responsible for issuing the control-signals to all the units inside the processor. We implement the actions specified by the instruction (loaded in the IR). Register R0 through R(n-1) are the Processor Registers. The programmer can access these registers for general-purpose use. Only processor can access 3 registers Y, Z & Temp for temporary storage during program-execution. The programmer cannot access these 3 registers. In ALU,1) "A‟ input gets the operand from the output of the multiplexer (MUX). 2) "B‟ input gets the operand directly from the processor-bus. There are 2 options provided for "A‟ input of the ALU. MUX is used to select one of the 2 inputs. MUX selects either output of Y or constant-value 4(which is used to increment PC content). An instruction is executed by performing one or more of the following operations:

1) Transfer a word of data from one register to another or to the ALU.
2) Perform arithmetic or a logic operation and store the result in a register.
3) Fetch the contents of a given memory-location and load them into a register.
4) Store a word of data from a register into a given memory-location.

Disadvantage: Only one data-word can be transferred over the bus in a clock cycle.

Solution: Provide multiple internal-paths. Multiple paths allow several data-transfers to take place in parallel.

REGISTER TRANSFERS

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control-signals are used: Riin & Riout. These are called Gating Signals. Riin=1 = data on bus is loaded into Ri. Riout=1 as content of Ri is placed on bus. Riout=0, makes bus can be used for transferring data from other registers.For example, Move R1, R2; This transfers the contents of register R1 to register R2. This can be accomplished as follows:

1) Enable the output of registers R1 by setting R1out to 1 (Figure 7.2). This places the contents of R1 on processor-bus.

2) Enable the input of register R2 by setting R2out to 1. This loads data from processor-bus into register R4.

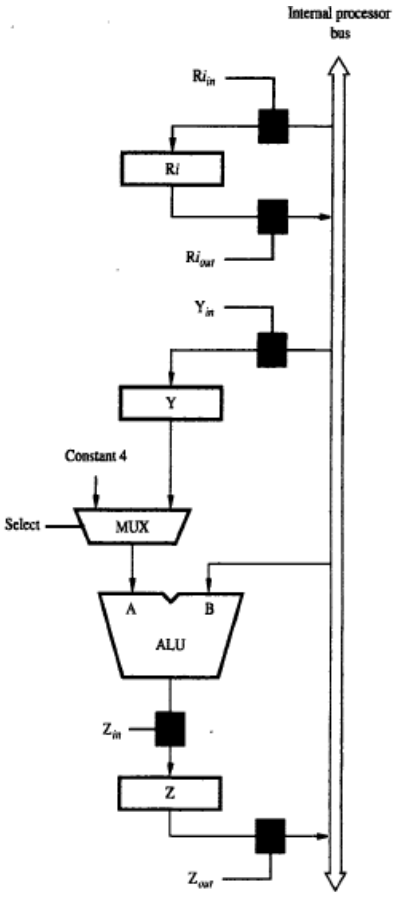


Figure 7.2 Input and output gating for the registers in Figure 7.1.

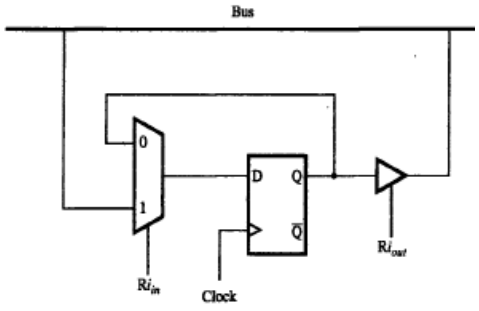


Figure 7.3 Input and output gating for one register bit.

All operations and data transfers within the processor take place within time-periods defined by the processor-clock. The control-signals that govern a particular transfer are asserted at the start of the clock cycle.

Input & Output Gating for one Register Bit

A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop. Riin=1 makes mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock. Riin=0 makes mux feeds back the value currently stored in flip-flop (Figure). Q output of flip-flop is connected to bus via a tri-state gate. Riout=0 makes gate's output is in the high-impedance state. Riout=1 makesthe gate drives the bus to 0 or 1, depending on the value of Q.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION

The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs. One of the operands is output of MUX and, the other operand is obtained directly from processor-bus. The result (produced by the ALU) is stored temporarily in register Z. The sequence of operations for [R3] [R1]+[R2] is as follows:

1) R1out, Yin
2) R2out, SelectY, Add, Zin
3) Zout, R3in

Instruction execution proceeds as follows:

Step 1 --> Contents from register R1 are loaded into register Y.

Step2 --> Contents from Y and from register R2 are applied to the A and B inputs of ALU; Addition is performed & Result is stored in the Z register.

Step 3 --> The contents of Z register is stored in the R3 register.

The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

CONTROL-SIGNALS OF MDR

The MDR register has 4 control-signals (Figure). MDRin & MDRout control the connection to the internal processor data bus & MDRinE & MDRoutE control the connection to the memory Data bus. MAR register has 2 control-signals. MARin controls the connection to the internal processor address bus & MARout controls the connection to the memory address bus.

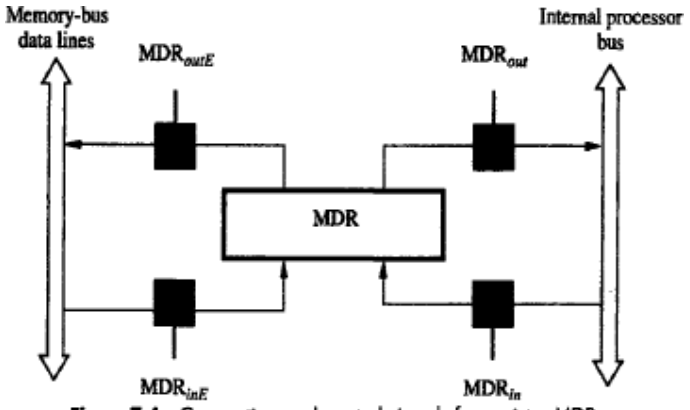
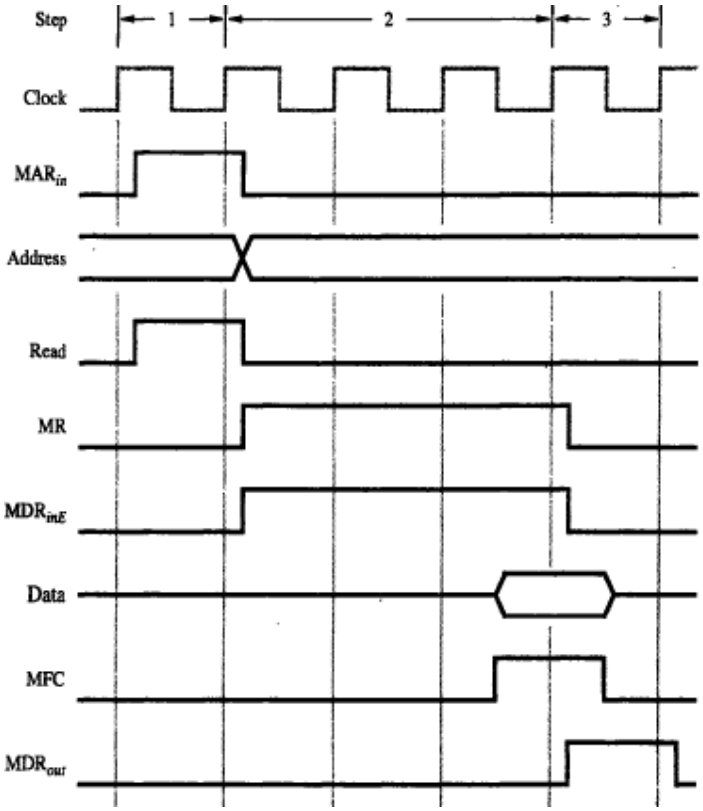


Figure 7.4 Connection and control signals for register MDR.

FETCHING A WORD FROM MEMORY

To fetch instruction/data from memory, processor transfers required address to MAR. At the same time, processor issues Read signal on control-lines of memory-bus. When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers. The response time of each memory access varies (based on cache miss, memory-mapped I/O). To accommodate this, MFC is used. (MFC makes Memory Function Completed). MFC is a signal sent from addressed-device to the processor. MFC informs the processor that the requested operation has been completed by addressed-device.



Consider the instruction Move (R1),R2. The sequence of steps is (Figure): R1out,

MARin, Read ;desired address is loaded into MAR & Read command is issued. MDRinE, WMFC; load MDR from memory-bus & Wait for MFC response from memory. MDRout, R2in; load R2 from MDR where WMFC=control-signal that causes processor's control. circuitry to wait for arrival of MFC signal.

Storing a Word in Memory

Consider the instruction Move R2,(R1). This requires the following sequence: R1out, MARin; desired address is loaded into MAR. R2out, MDRin, Write; data to be written are loaded into MDR & Write command is issued. MDRoutE, WMFC ;load data into memory location pointed by R1 from MDR.

EXECUTION OF A COMPLETE INSTRUCTION

Consider the instruction Add (R3),R1 which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:

1) Fetch the instruction.
2) Fetch the first operand.
3) Perform the addition &
4) Load the result into R1.

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1

Instruction execution proceeds as follows:

Step1: The instruction-fetch operation is initiated by loading contents of PC into MAR & sending a Read request to memory. The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC‟s content), and the result is stored in Z.

Step2: Updated value in Z is moved to PC. This completes the PC increment operation and PC will now point to next instruction.

Step3: Fetched instruction is moved into MDR and then to IR. The step 1 through 3 constitutes the Fetch Phase. At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control-signals for steps 4 through 7. The step 4 through 7 constitutes the Execution Phase.

Step4: Contents of R3 are loaded into MAR & a memory read signal is issued.

Step5: Contents of R1 are transferred to Y to prepare for addition.

Step6: When Read operation is completed, memory-operand is available in MDR, and the addition is performed.

Step7: Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

BRANCHING INSTRUCTIONS

Control sequence for an unconditional branch instruction is as follows: Instruction execution proceeds as follows:

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7 Control sequence for an unconditional Branch instruction.

Step 1-3: The processing starts & the fetch phase ends in step3.

Step 4: The offset-value is extracted from IR by instruction-decoding circuit. Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.

Step 5: The result, which is the branch-address, is loaded into the PC.

The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address. The branch target address is usually obtained by adding the offset in the contents of PC. The offset X is usually the difference between the branch target-address and the address immediately following the branch instruction.

In case of conditional branch, we have to check the status of the condition-codes before loading a new value into the PC. e.g.: Offset-field-of-IRout, Add, Zin, If N=0 then End If N=0, processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to

load a new value into PC.

MULTIPLE BUS ORGANIZATION

The disadvantage of Single-bus organization is only one data-word can be transferred over the bus in a clock cycle. This increases the steps required to complete the execution of the instruction. The solution to reduce the number of steps, most processors provide multiple internal-paths. Multiple paths enable several transfers to take place in parallel.

Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Figure 7.9 Control sequence for the instruction Add R4,R5,R6

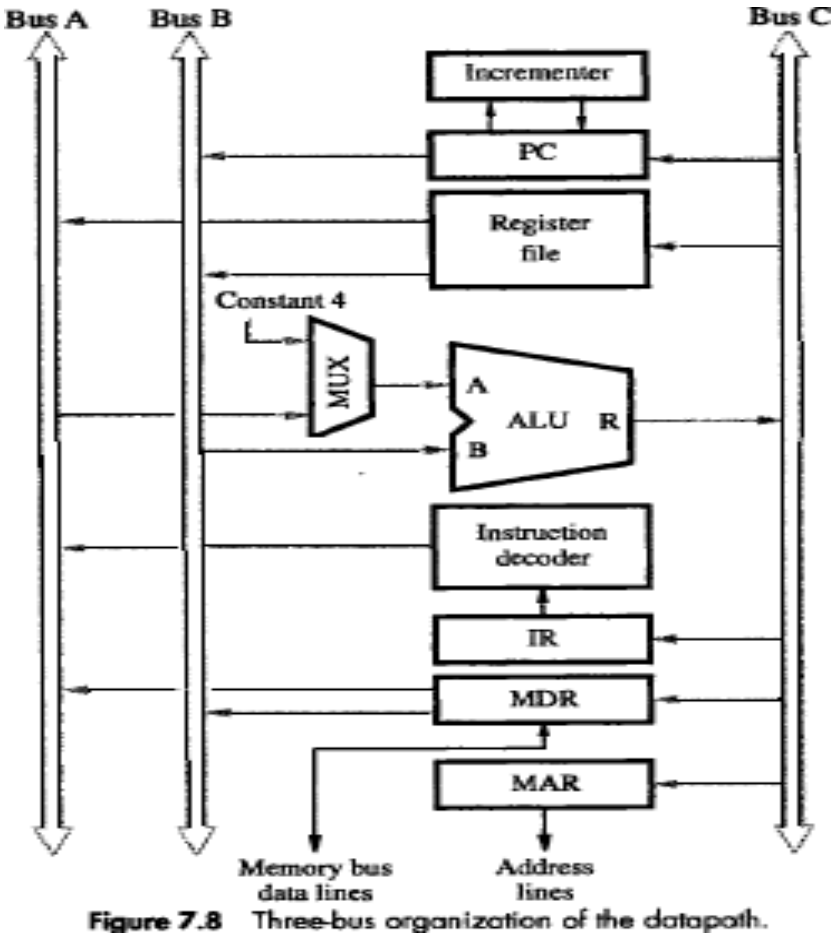


Figure 7.8 Three-bus organization of the datapath.

As shown in figure, three buses can be used to connect registers and the ALU of the processor. All general-purpose registers are grouped into a single block called the Register

File. Register-file has 3 ports:

1) Two output-ports allow the contents of 2 different registers to be simultaneously placed on buses A & B.

2) Third input-port allows data on bus C to be loaded into a third register during the same clock-cycle.

Buses A and B are used to transfer source-operands to A & B inputs of ALU. The result is transferred to destination over bus C. Incrementer Unit is used to increment PC by 4. Instruction execution proceeds as follows:

Step 1: Contents of PC are passed through ALU using R=B control-signal & loaded into MAR to start memory Read operation. At the same time, PC is incremented by 4.

Step2: Processor waits for MFC signal from memory.

Step3: Processor loads requested-data into MDR, and then transfers them to IR.

Step4: The instruction is decoded and add operation takes place in a single step.

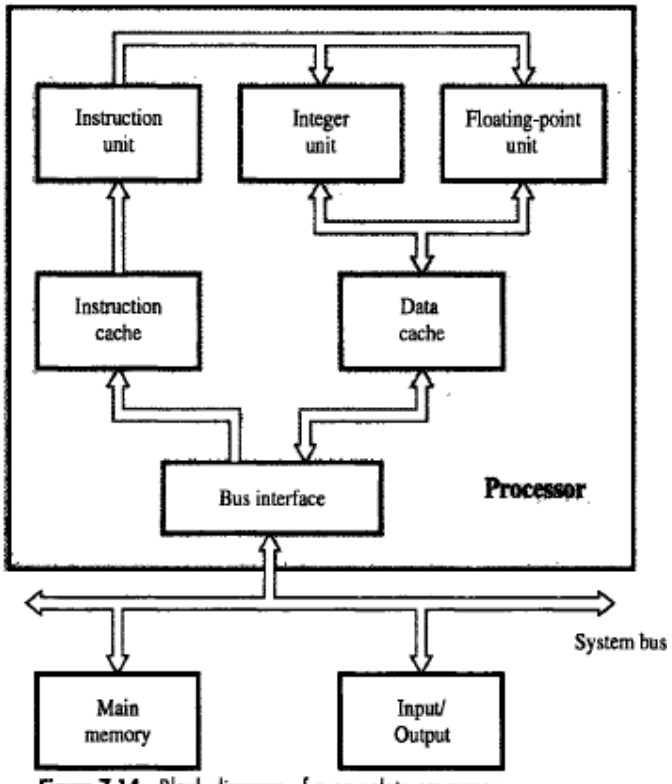


Figure 7.14 Block diagram of a complete processor.

COMPLETE PROCESSOR

This has separate processing-units to deal with integer data and floating-point data. Integer unit has to process integer data. (Figure). Floating unit has to process floating point

data. Data-Cache is inserted between these processing-units & main-memory. The integer and floating unit gets data from data cache. Instruction-Unit fetches instructions from an instruction-cache or from main-memory when desired instructions are not already in cache.

Processor is connected to system-bus & hence to the rest of the computer by means of a Bus Interface. Using separate caches for instructions & data is common practice in many processors today. A processor may include several units of each type to increase the potential for concurrent operations. The 80486 processor has 8-kbytes single cache for both instruction and data. Whereas the Pentium processor has two separate 8 kbytes caches for instruction and data.

Note: To execute instructions, the processor must have some means of generating the control-signals. There are two approaches for this purpose:

1) Hardwired control and

2) Microprogrammed control.

HARDWIRED CONTROL

Hardwired control is a method of control unit design (Figure). The control-signals are generated by using logic circuits such as gates, flip-flops, decoders etc. Decoder / Encoder Block is a combinational-circuit that generates required control-outputs depending on state of all its inputs. Instruction decoder decodes the instruction loaded in the IR. If IR is an 8 bit register, then instruction decoder generates 28(256 lines); one for each instruction. It consists of a separate output-lines INS1 through INSm for each machine instruction. According to code in the IR, one of the output-lines INS1 through INSm is set to 1, and all other lines are set to 0. Step-Decoder provides a separate signal line for each step in the control sequence. Encoder gets the input from instruction decoder, step decoder, external inputs and condition codes. It uses all these inputs to generate individual control-signals: Yin, PCout, Add, End and so on. For example (Figure 7.12), Zin=T1+T6.ADD+T4.BR; This signal is asserted during time-slot T1 for all instructions during T6 for an Add instruction. During T4 for unconditional branch instruction, when RUN=1, counter is incremented by 1 at the end of every clock cycle. When RUN=0, counter stops counting. After execution of each instruction, end signal is generated. End signal resets step counter. Sequence of operations carried out by this machine is determined by wiring of logic circuits, hence
the name “hardwired”.

Advantage: Can operate at high speed.

Disadvantages: 1) Since no. of instructions/control-lines is often in hundreds, the complexity of control unit is very high.

2) It is costly and difficult to design.

3) The control unit is inflexible because it is difficult to change the design.

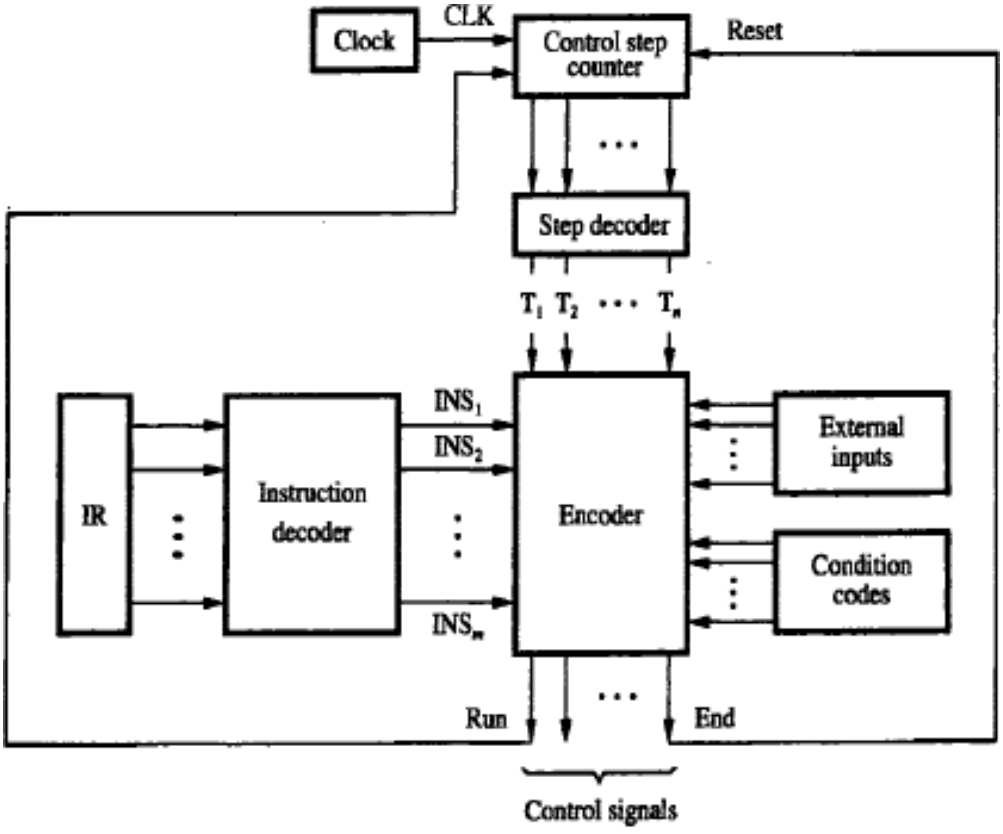


Figure 7.11 Separation of the decoding and encoding functions.

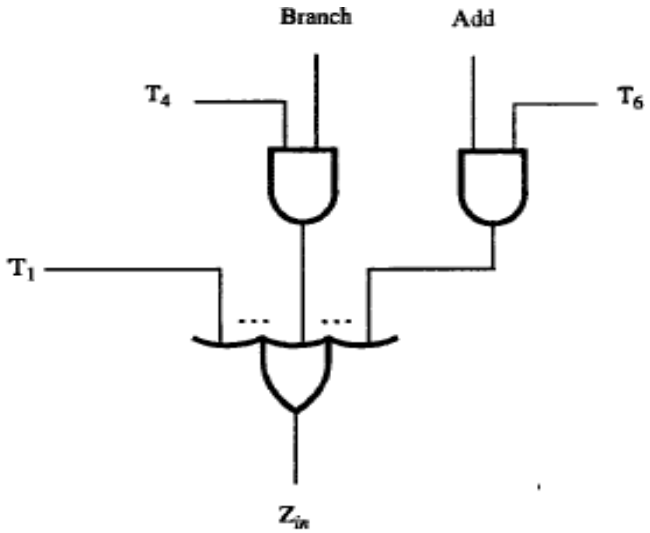
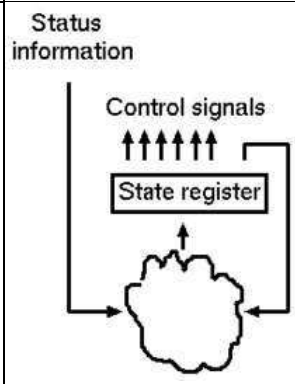
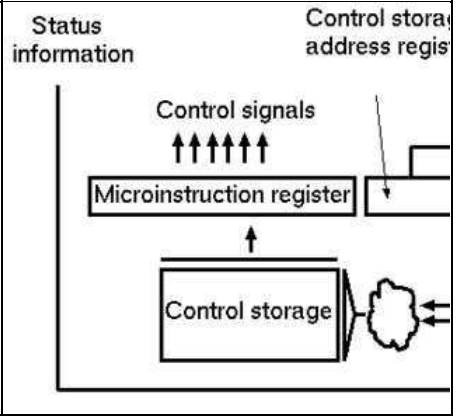


Figure 7.12 Generation of the Z_{in} control signal

HARDWIRED CONTROL VS MICROPROGRAMMED CONTROL

Attribute	**Hardwired Control**	**Microprogrammed Control**
Definition	Hardwired control is a control mechanism to generate control-signals by using gates, flip- flops, decoders, and other digital circuits.	Micro programmed control is a control mechanism to generate control-signals by using a memory called control store (CS), which contains the control-signals.
Speed	Fast	Slow
Control functions	Implemented in hardware.	Implemented in software.
Flexibility	Not flexible to accommodate new system specifications or new instructions.	More flexible, to accommodate new system specification or new instructions redesign is required.
Ability to handle large or complex instruction sets	Difficult.	Easier.
Ability to support Operating systems& diagnostic features	Very difficult.	Easy.
Design process	Complicated.	Orderly and systematic.
Applications	Mostly RISC microprocessors.	Mainframes, some microprocessors.
Instruction set size	Usually under 100 instructions.	Usually over 100 instructions.
ROM size	-	2K to 10K by 20-400 bit microinstructions.
Chip area efficiency	Uses least area.	Uses more area.
Diagram		

MICROPROGRAMMED CONTROL

Microprogramming is a method of control unit design (Figure). Control-signals are generated by a program similar to machine language programs. Control Word(CW) is a word whose individual bits represent various control-signals (like Add, PCin). Each of the control-steps in control sequence of an instruction defines a unique combination of 1s & 0s in CW. Individual control-words in microroutine are referred to as microinstructions (Figure).

A sequence of CWs corresponding to control-sequence of a machine instruction constitutes the microroutine. The microroutines for all instructions in the instruction-set of a computer are stored in a special memory called the Control Store (CS). Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS. µPC is used to read CWs sequentially from CS. (µPC Microprogram Counter). Every time new instruction is loaded into IR, o/p of Starting Address Generator is loaded into µPC. Then, µPC is automatically incremented by clock; causing successive microinstructions to be read from CS. Hence, control-signals are delivered to various parts of processor in correct sequence.

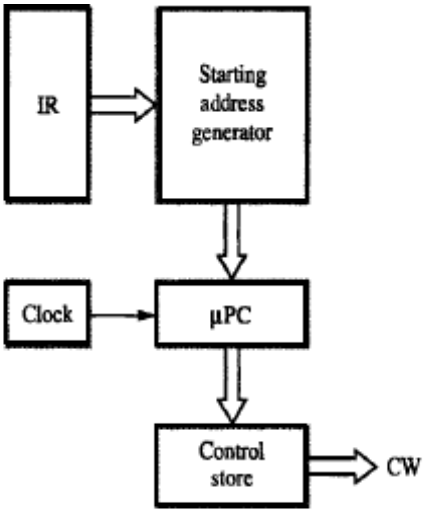


Figure 7.16 Basic organization of a microprogrammed control unit.

Micro-instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	:
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	

Figure 7.15 An example of microinstructions for Figure 7.6.

Advantages

• It simplifies the design of control unit. Thus it is both, cheaper and less error prone implement.

• Control functions are implemented in software rather than hardware.

• The design process is orderly and systematic.

• More flexible, can be changed to accommodate new system specifications or to correct the design errors quickly and cheaply.

• Complex function such as floating point arithmetic can be realized efficiently.

Disadvantages

• A microprogrammed control unit is somewhat slower than the hardwired control unit, because time is required to access the microinstructions from CM.

• The flexibility is achieved at some extra hardware cost due to the control memory and its access circuitry.

Organization Of Microprogrammed Control Unit To Support Conditional Branching

Drawback of previous Microprogram control

□ It cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.

Solution:

□ Use conditional branch microinstruction.

In case of conditional branching, microinstructions specify which of the external inputs, condition- codes should be checked as a condition for branching to take place. Starting and Branch Address Generator Block loads a new address into µPC when a microinstruction instructs it to do so (Figure). To allow implementation of a conditional branch, inputs to this block consist of external inputs and condition-codes & contents of IR.

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17 Microroutine for the instruction Branch < 0.

µPC is incremented every time a new microinstruction is fetched from microprogram memory except in following situations:

1) When a new instruction is loaded into IR, µPC is loaded with starting-address of microroutine for that instruction.

2) When a Branch microinstruction is encountered and branch condition is satisfied, µPC is loaded with branch-address.

3) When an End microinstruction is encountered, µPC is loaded with address of first CW in microroutine for instruction fetch cycle.

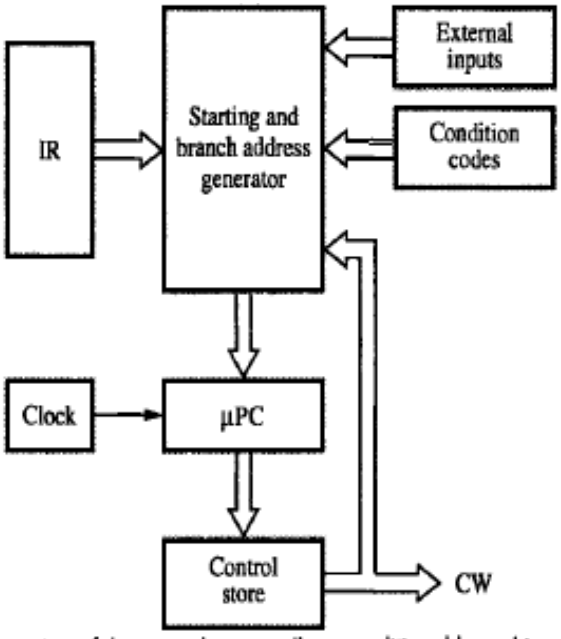


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

Microinstructions

A simple way to structure microinstructions is to assign one bit position to each control-signal required in the CPU. There are 42 signals and hence each microinstruction will have 42 bits.

Drawbacks of microprogrammed control:

1) Assigning individual bits to each control-signal results in long microinstructions because the number of required signals is usually large.

2) Available bit-space is poorly used because only a few bits are set to 1 in any given microinstruction.

Solution: Signals can be grouped because

1) Most signals are not needed simultaneously.

2) Many signals are mutually exclusive. E.g. only 1 function of ALU can be activated at a time. For ex: Gating signals: IN and OUT signals (Figure); Control-signals: Read, Write; ALU signals: Add, Sub, Mul, Div, Mod.

Grouping control-signals into fields requires a little more hardware because decoding-circuits must be used to decode bit patterns of each field into individual control-signals.

Advantage: This method results in a smaller control-store (only 20 bits are needed to store the patterns for the 42 signals)

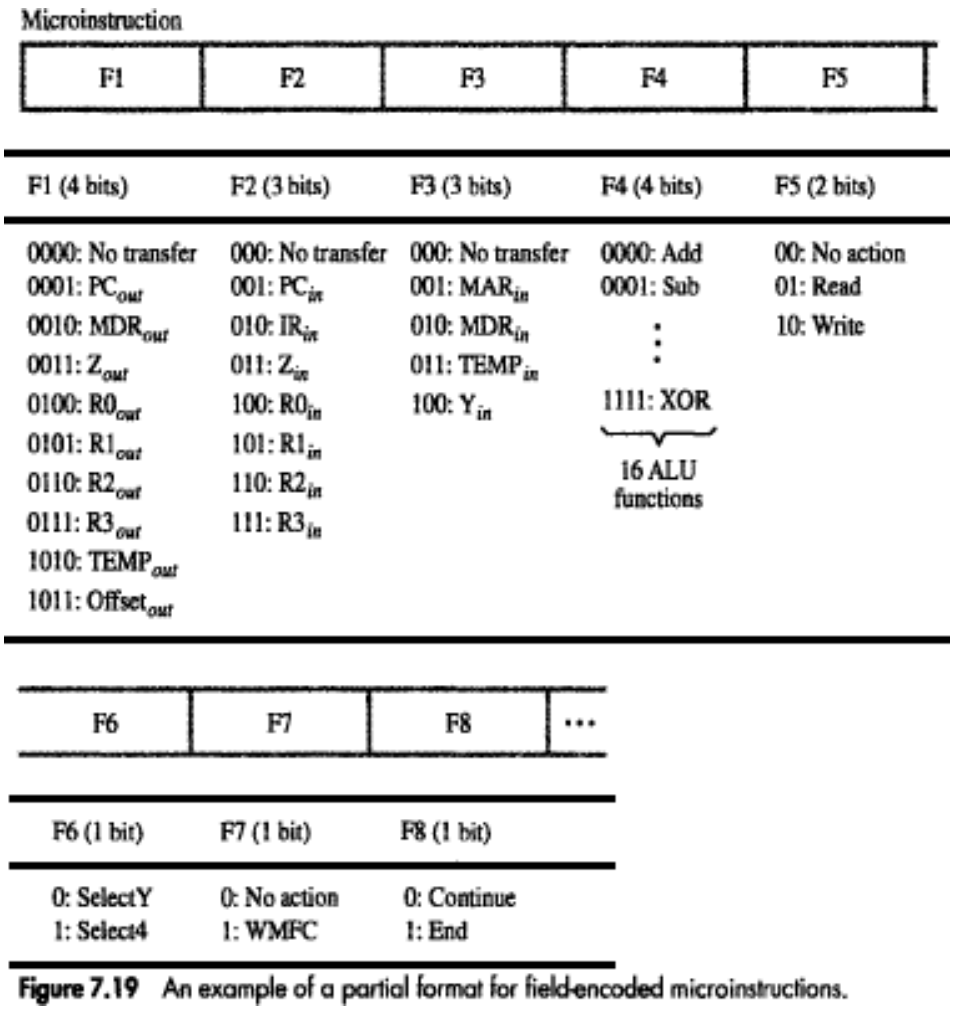


Figure 7.19 An example of a partial format for field-encoded microinstructions.

Techniques of Grouping of Control-Signals

The grouping of control-signal can be done either by using

1) Vertical organization &

2) Horizontal organisation.

Vertical Organization	Horizontal Organization
Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a vertical organization.	The minimally encoded scheme in which many resources can be controlled with a single microinstuction is called a horizontal organization.
Slower operating-speeds.	Useful when higher operating-speed is desired.
Short formats.	Long formats.
Limited ability to express parallel microoperations.	Ability to express a high degree of parallelism.
Considerable encoding of the control information.	Little encoding of the control information.

Microprogram Sequencing

The task of microprogram sequencing is done by microprogram sequencer. Two important factors must be considered while designing the microprogram sequencer,

1) The size of the microinstruction &

2) The address generation time.

The size of the microinstruction should be minimum so that the size of control memory required to store microinstructions is also less. This reduces the cost of control memory. With less address generation time, microinstruction can be executed in less time resulting better throughout. During execution of a microprogram the address of the next microinstruction to be executed has 3 sources:

1) Determined by instruction register.

2) Next sequential address &

3) Branch.

Microinstructions can be shared using microinstruction branching.

Disadvantage of microprogrammed branching:

1) Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control-store.

2) Execution time is longer because it takes more time to carry out the required branches.

Consider the instruction Add src,Rdst ;which adds the source-operand to the contents of Rdst and places the sum in Rdst. Let source-operand can be specified in following addressing modes

a) Indexed b) Autoincrement c) Autodecrement d) Register indirect & e) Register direct

Each box in the chart corresponds to a microinstruction that controls the transfers and operations indicated within the box. The microinstruction is located at the address indicated by the octal number (001,002).

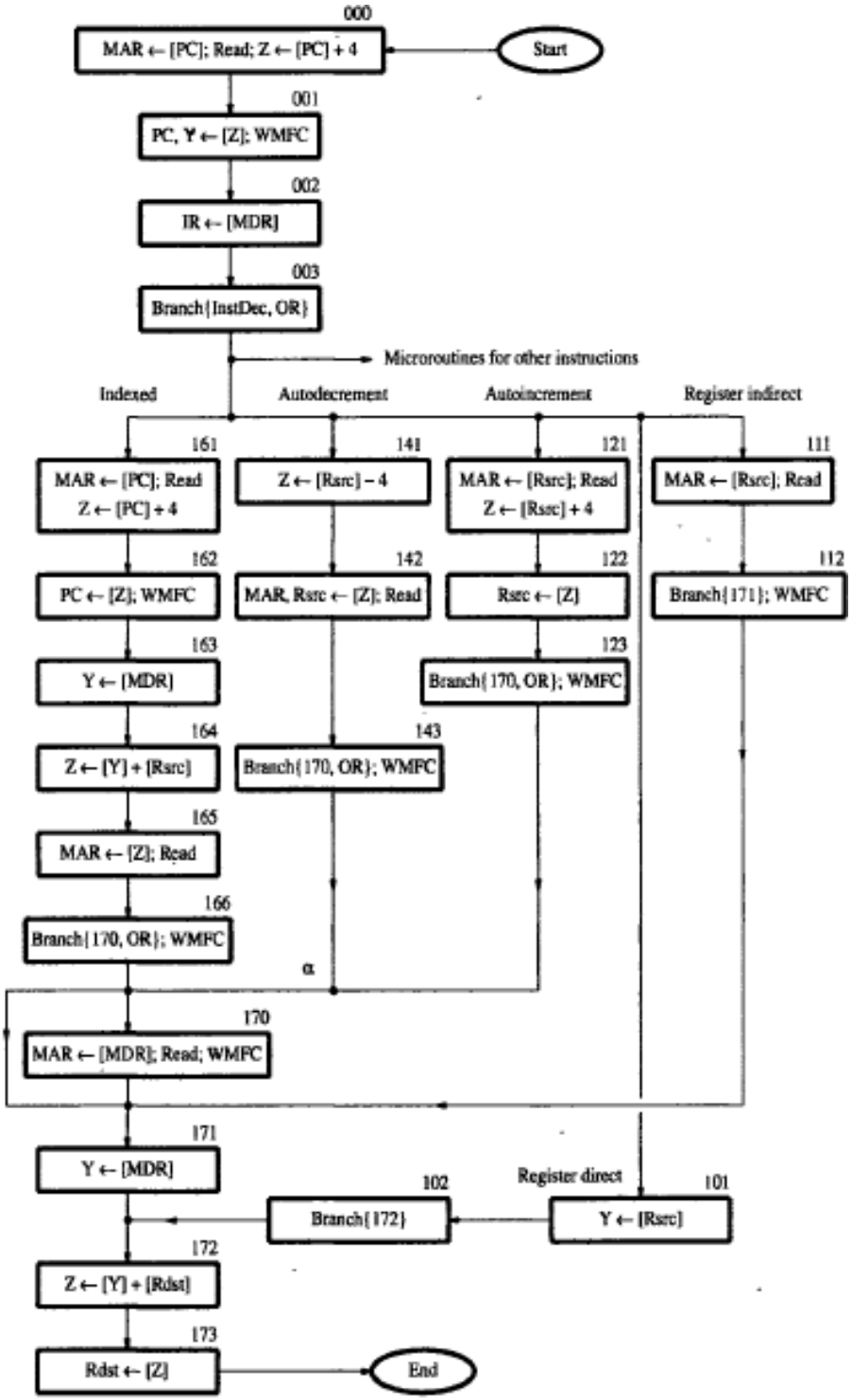


Figure 7.20 Flowchart of a microprogram for the Add src,Rdst instruction.

Branch Address Modification Using Bit-Oring

The branch address is determined by ORing particular bit or bits with the current address of microinstruction. Eg: If the current address is 170 and branch address is 171 then the branch address can be generated by ORing 01(bit 1), with the current address. Consider the point labelled in the figure. At this point, it is necessary to choose between direct and indirect addressing modes. If indirect-mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory. If direct-mode is specified, this fetch must be bypassed by branching immediately to location 171. The most efficient way to bypass microinstruction 170 is to have bit-ORing of current address 170 & branch address 171.

Wide Branch Addressing

The instruction-decoder (InstDec) generates the starting-address of the microroutine that implements the instruction that has just been loaded into the IR. Here, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101. (However, this address cannot be loaded as is into the μPC). The source-operand can be specified in any of several addressing-modes. The bit-ORing technique can be used to modify the starting-address generated by the instruction-decoder to reach the appropriate path.

Use of WMFC

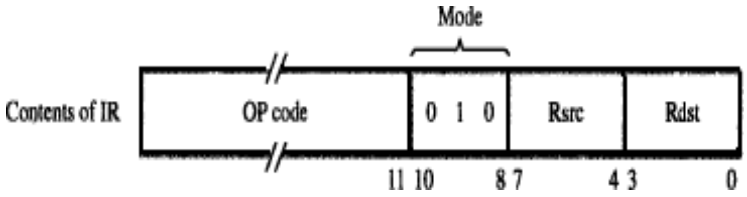
WMFC signal is issued at location 112 which causes a branch to the microinstruction in location 171. WMFC signal means that the microinstruction may take several clock cycles to complete. If the branch is allowed to happen in the first clock cycle, the microinstruction at location 171 would be fetched and executed prematurely. To avoid this problem, WMFC signal must inhibit any change in the contents of the μPC during the waiting-period.

Detailed Examination of Add (Rsrc)+,Rdst

Consider Add (Rsrc)+,Rdst; which adds Rsrc content to Rdst content, then stores the sum in Rdst and finally increments Rsrc by 4 (i.e. auto-increment mode). In bit 10 and 9, bit-patterns 11, 10, 01 and 00 denote indexed, auto-decrement, auto-increment and register modes respectively. For each of these modes, bit 8 is used to specify the indirect version. The processor has 16 registers that can be used for addressing purposes; each specified using a 4-bit-code (Figure 7.21). There are 2 stages of decoding:

1) The microinstruction field must be decoded to determine that an Rsrc or Rdst register is involved.

2) The decoded output is then used to gate the contents of the Rsrc or Rdst fields in the IR into a second decoder, which produces the gating-signals for the actual registers R0 to R15.



Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {μPC ← 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {μPC ← 170; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21 Microinstruction for Add (Rsrc)+,Rdst.

Microinstructions With Next-Address Fields

Drawback of previous organization:

□ The microprogram requires several branch microinstructions which perform no useful operation. Thus, they detract from the operating-speed of the computer.

Solution:

□ Include an address-field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched. (Thus, every microinstruction becomes a branch microinstruction).

The flexibility of this approach comes at the expense of additional bits for the address-field (Figure). Advantage: Separate branch microinstructions are virtually eliminated. (Figure). *Disadvantage:* Additional bits for the address field (around 1/6).

• There is no need for a counter to keep track of sequential address. Hence, μPC is replaced with μAR.

• The next-address bits are fed through the OR gate to the μAR, so that the address can be modified on the basis of the data in the IR, external inputs and condition-codes.

• The decoding circuits generate the starting-address of a given microroutine on the basis of the opcode in the IR. (μAR Microinstruction Address Register).

Prefetching Microinstructions

Disadvantage of Microprogrammed Control: Slower operating-speed because of the time it takes to fetch microinstructions from the control-store.

Solution: Faster operation is achieved if the next microinstruction is pre-fetched while the current one is being executed.

Emulation

• The main function of microprogrammed control is to provide a means for simple, flexible and relatively inexpensive execution of machine instruction.

• Its flexibility in using a machine's resources allows diverse classes of instructions to be implemented.

• Suppose we add to the instruction-repository of a given computer M1, an entirely new set of instructions that is in fact the instruction-set of a different computer M2.

• Programs written in the machine language of M2 can be then be run on computer M1 i.e. M1 emulates M2.

• Emulation allows us to replace obsolete equipment with more up-to-date machines.

• If the replacement computer fully emulates the original one, then no software changes have to be made to run existing programs.

• Emulation is easiest when the machines involved have similar architectures.

Pipelining

==========

OBJECTIVES:

In this lesson, you will be described about the pipelining as a means for executing machine instructions concurrently, various hazards that cause performance degradation in pipelined processors and means for mitigating their effect, hardware and software implications of pipelining, influence of pipelining on instruction set design, superscalar processors and design consideration of data-path unit and control unit for pipelining performance.

CONTENTS:

1. Basic concepts
    - Data hazards
2. Instruction hazards
    - Influence on instruction sets
3. Data path and control considerations
4. Performance considerations
    - Exception handling

Introduction

The basic building blocks of a computer are introduced in preceding chapters. In this chapter, we discuss in detail the concept of pipelining, which is used in modern computers to achieve high performance. We begin by explaining the basics of pipelining and how it can lead to improved performance. Then we examine machine instruction features that facilitate pipelined execution, and we show that the choice of instructions and instruction sequencing can have a significant effect on performance. Pipelined organization requires sophisticated compilation techniques, and optimizing compilers have been developed for this purpose.

Among other things, such compilers rearrange the sequence of operations to maximize the benefits of pipelined execution.

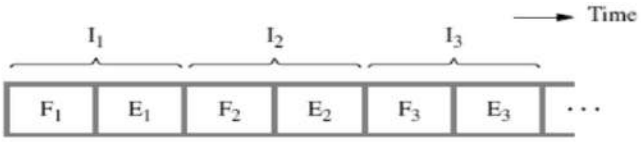
1. BASIC CONCEPTS

The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed. We have encountered concurrent activities several times before. The concept of multiprogramming and explained how it is possible for I/O transfers and computational activities to proceed simultaneously. DMA devices make this possible because they can perform I/O transfers independently once these transfers are initiated by the processor.

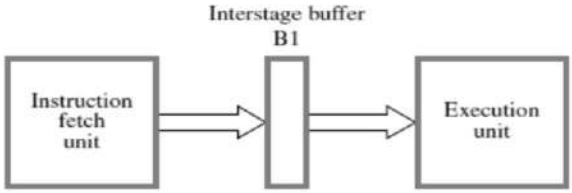
Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let Fi and Ei refer to the fetch and execute steps for instruction Ii . Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure a.

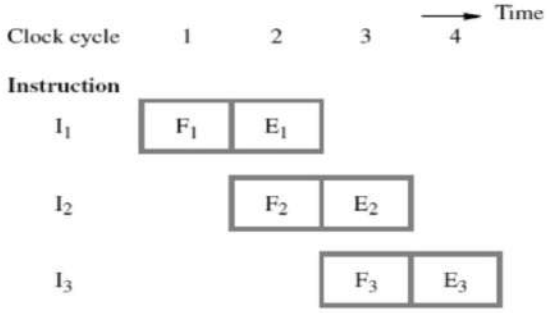
Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure b. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labelled “Execution unit.”



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1c. In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2). Meanwhile, the execution unit performs the operation specified by instruction I1, which is available to it in buffer B1 (step E1). By the end of the second clock cycle, the execution of instruction I1 is completed and instruction I2 is available. Instruction I2 is stored in B1, replacing I1, which is no longer needed. Step E2 is performed by the execution unit during the third clock cycle, while instruction I3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1c can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure a.

In summary, the fetch and execute units in Figure b constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An inter-stage storage buffer,

B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

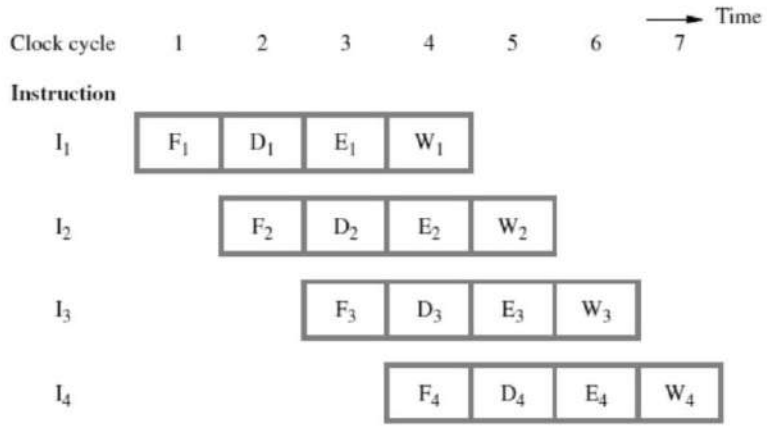
The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory.

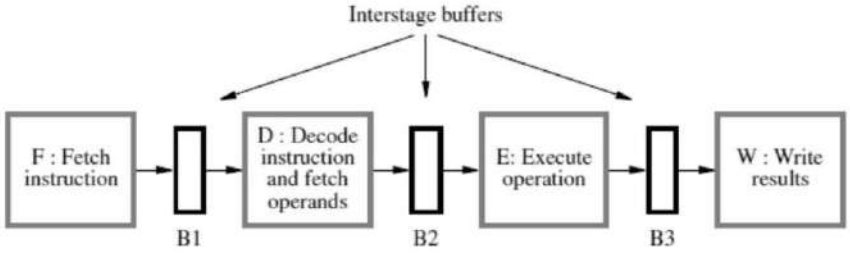
D Decode: decode the instruction and fetch the source operand(s).

E Execute: perform the operation specified by the instruction.

W Write: store the result in the destination location.



(a) Instruction execution divided into four steps



(b) Hardware organization

The sequence of events for this case is shown in Figure a. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure b. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

• Buffer B1 holds instruction I3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.

• Buffer B2 holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I2 (stepW2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.

• Buffer B3 holds the results produced by the execution unit and the destination information for instruction I1.

Role of Cache Memory

Each stage in a pipeline is expected to complete its operation in one clock cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage. If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving performance if the tasks being performed in different stages require about the same amount of time. This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure a. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each instruction fetch required access to the main memory, pipelining would be of little value.

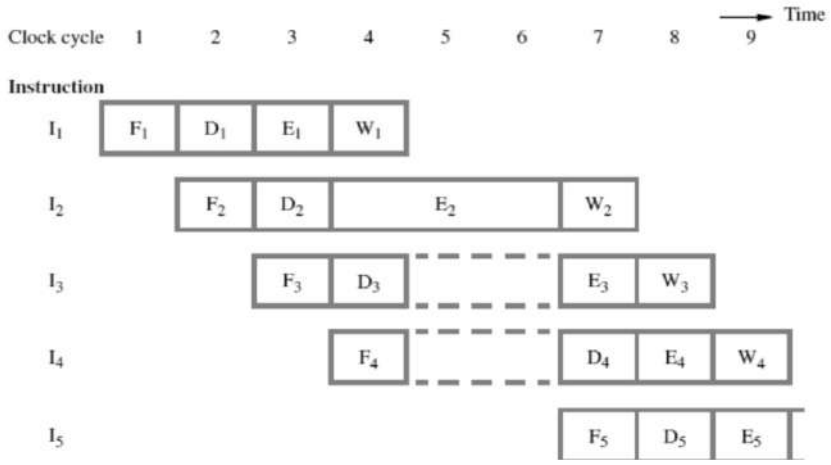
The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

Pipeline Performance

The pipelined processor in Figure completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to

the number of pipeline stages. However, this increase would be achieved only if pipelined operation as depicted in Figure a could be sustained without interruption throughout program execution. Unfortunately, this is not the case.

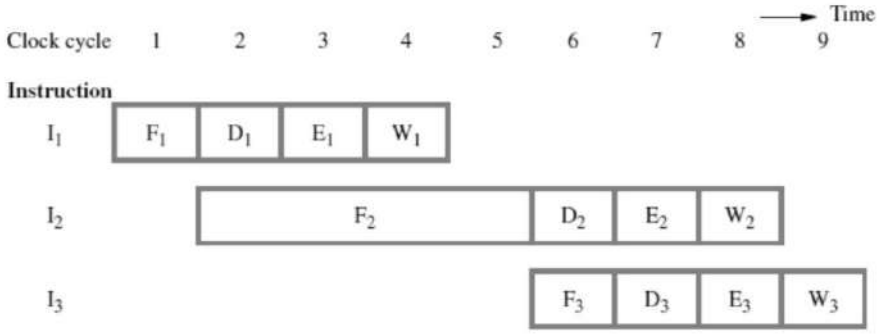
For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four-stage pipeline of Figure b is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure shows an example in which the operation specified in instruction I2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D4 and F5 must be postponed as shown.



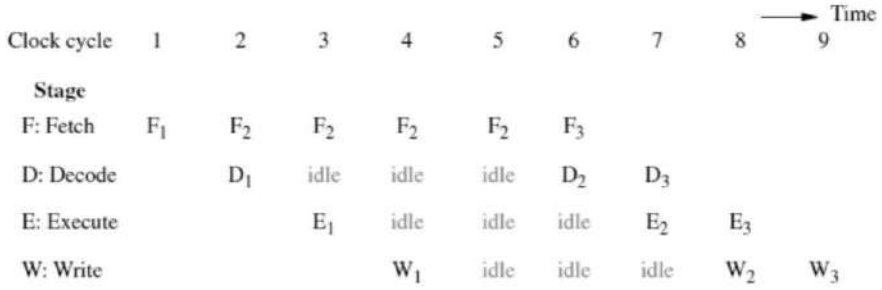
Effect of an execution operation taking more than one clock cycle

Pipelined operation in Figure is said to have been stalled for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a hazard. We have just seen an example of a data hazard. A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls. The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called control hazards or instruction hazards. The effect of a cache miss on pipelined operation is illustrated in Figure. Instruction I1 is

fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I2, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I2 to arrive. We assume that instruction I2 is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.



(a) Instruction execution steps in successive clock cycles



(b) Function performed by each processor stage in successive clock cycles

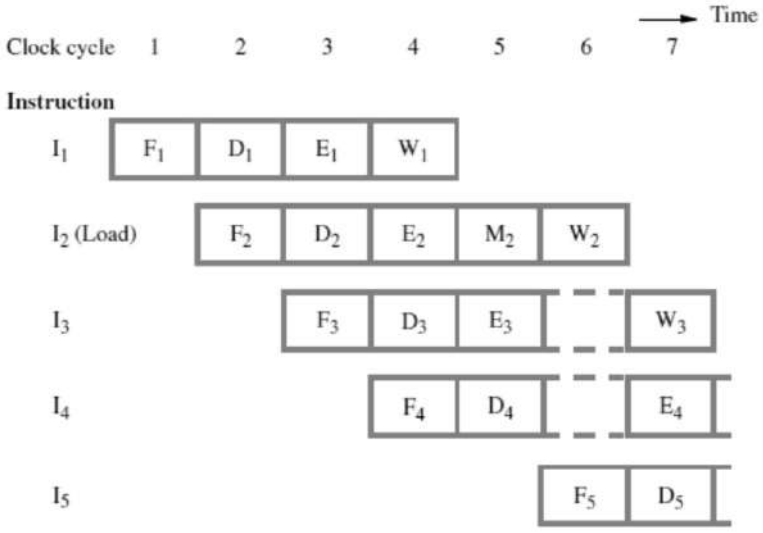
An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure b. This figure gives the function performed by each pipeline stage in each clock cycle. Note that the Decode unit is idle in cycles 3 through 5, the Execute unit is idle in cycles 4 through 6, and the Write unit is idle in cycles 5 through 7. Such idle periods are called stalls. They are also often referred to as bubbles in the pipeline. Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit.

A third type of hazard that may be encountered in pipelined operation is known as a structural hazard. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use

separate instruction and data caches to avoid this delay. An example of a structural hazard is shown in Figure. This figure shows how the load instruction

Load X(R1),R2

can be accommodated in our example 4-stage pipeline. The memory address, X+[R1], is computed in stepE2 in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions I2 and I3 require access to the register file in cycle 6. Even though the instructions and their data are all available, the pipeline is stalled because one hardware resource, the register file, cannot handle two operations at once. If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled. In general, structural hazards are avoided by providing sufficient hardware resources on the processor chip.



Effect of a Load instruction on pipeline timing

It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases, where throughput is measured by the rate at which instruction execution is completed. Any time one of the stages in the pipeline cannot complete its operation in one clock cycle, the pipeline stalls, and some degradation in performance occurs. Thus, the performance level of one instruction completion in each clock cycle is actually the upper limit for the throughput achievable in a pipelined processor organized as in Figure b. An important goal in designing processors is to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact. In the following sections we discuss various hazards, starting with data hazards, followed by control

hazards. In each case we present some of the techniques used to mitigate their negative effect on performance. We discuss the issue of performance assessment in the following section in detail.

DATA HAZARDS

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure. We will now examine the issue of availability of data in some detail. Consider a program that contains two instructions, I1 followed by I2. When this program is executed in a pipeline, the execution of I2 can begin before the execution of I1 is completed. This means that the results generated by I1 may not be available for use by I2. We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that A=5, and consider the following two operations:

$$A \leftarrow 3 + A$$
$$B \leftarrow 4 \times A$$

When these operations are performed in the order given, the result is B = 32. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

$$A \leftarrow 5 \times C$$
$$B \leftarrow 20 + C$$

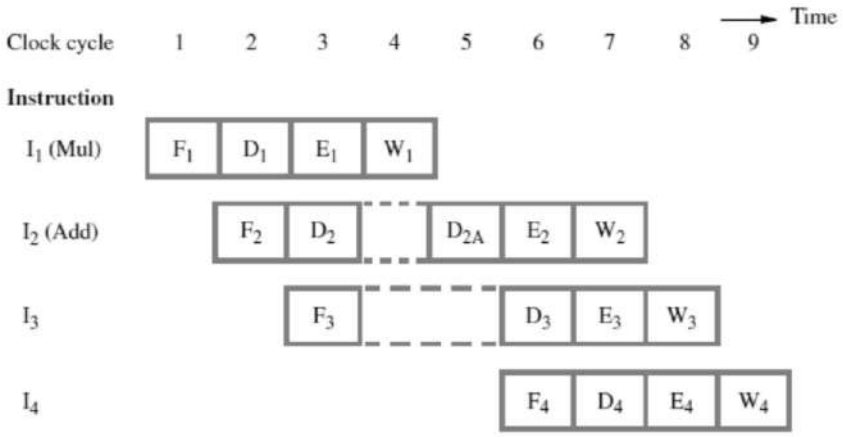
can be performed concurrently, because these operations are independent.

This example illustrates a basic constraint that must be enforced to guarantee correct results. When two operations depend on each other, they must be performed sequentially in the correct order. This rather obvious condition has far-reaching consequences. Understanding its implications is the key to understanding the variety of design alternatives and trade-offs encountered in pipelined computers. Consider the pipeline in Figure 2. The data dependency just described arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

Mul R2,R3,R4

Add R5,R4,R6

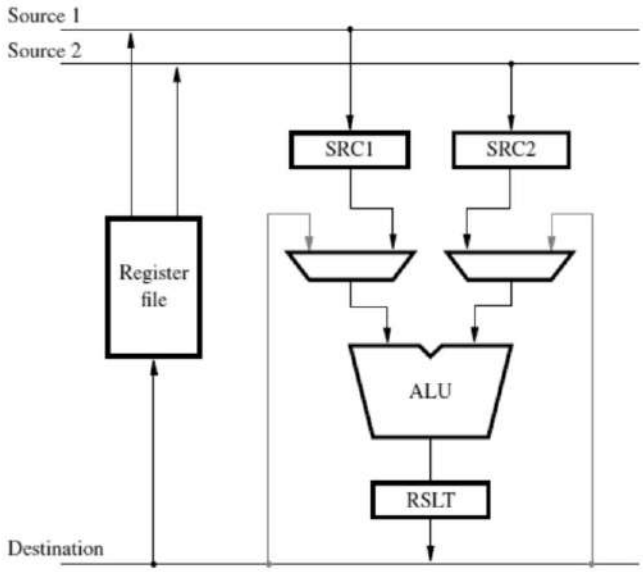
give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D2 must be delayed to clock cycle 5, and is shown as step D2A in the figure. Instruction I3 is fetched in cycle 3, but its decoding must be delayed because step D3 cannot precede D2. Hence, pipelined execution is stalled for two cycles.



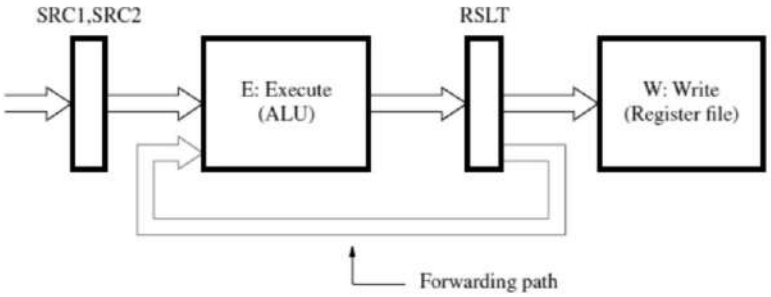
Operand Forwarding

The data hazard just described arises because one instruction, instruction I2 in Figure, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E1. Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I1 to be forwarded directly for use in step E2. Figure a shows a part of the processor datapath involving the ALU and the register file. This arrangement is similar to the three-bus structure in Figure, except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the interstage buffers needed for pipelined operation, as illustrated in Figure b. With reference to Figure b, registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3. The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

When the instructions in Figure are executed in the datapath of Figure, the operations performed in each clock cycle are as follows. After decoding instruction I2 and detecting the data dependency, a decision is made to use data forwarding. The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I1 is available in register RSLT, and because of the forwarding connection, it can be used in step E2. Hence, execution of I2 proceeds without interruption.



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Side Effects

The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I1 and as a source in I2. Sometimes an instruction changes the contents of a register other than the one named as the destination. An instruction that uses an autoincrement or autodecrement addressing mode is an example. In addition to storing new data in its destination location, the

instruction changes the contents of a source register used to access one of its operands. All the precautions needed to handle data dependencies involving the destination location must also be applied to the registers affected by an autoincrement or autodecrement operation. When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect. For example, stack instructions, such as push and pop, produce similar side effects because they implicitly use the autoincrement and autodecrement addressing modes.

Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry. Suppose that registers R1 and R2 hold a double-precision integer number that we wish to add to another double precision number in registers R3 and R4. This may be accomplished as follows:

```
Add R1,R3
AddWithCarry R2,R4
```

An implicit dependency exists between these two instructions through the carry flag. This flag is set by the first instruction and used in the second instruction, which performs the operation

$$R4 \leftarrow [R2] + [R4] + carry$$

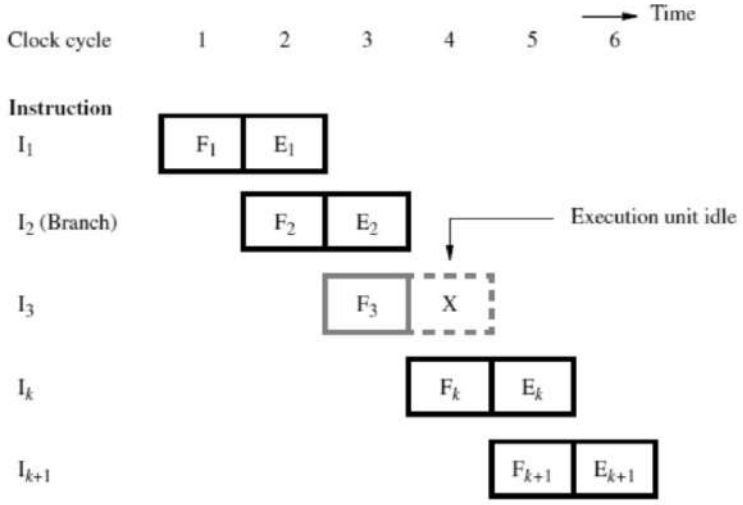
Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have few side effects. Ideally, only the contents of the destination location, either a register or a memory location, should be affected by any given instruction. Side effects, such as setting the condition code flags or updating the contents of an address pointer, should be kept to a minimum. However, it showed that the autoincrement and autodecrement addressing modes are potentially useful. Condition code flags are also needed for recording such information as the generation of a carry or the occurrence of overflow in an arithmetic operation. We show how such functions can be provided by other means that are consistent with a pipelined organization and with the requirements of optimizing compilers.

2. INSTRUCTION HAZARDS

The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. Whenever this stream is interrupted, the pipeline stalls, as Figure illustrates for the case of a cache miss. A branch instruction may also cause the pipeline to stall. We will now examine the effect of branch instructions and the techniques that can be used for mitigating their impact.

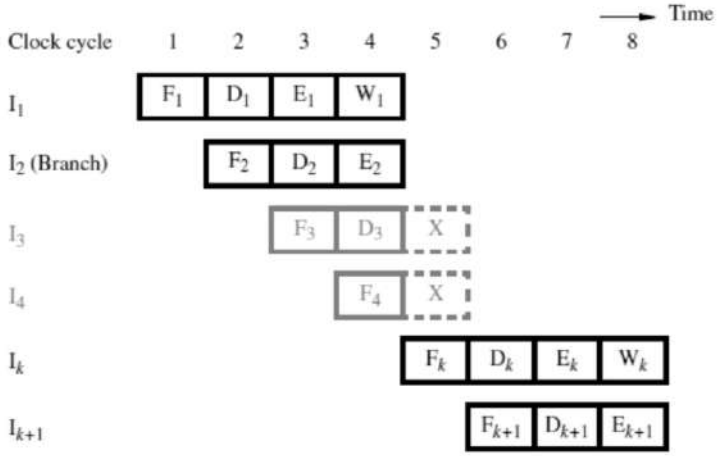
Unconditional Branches

Figure shows a sequence of instructions being executed in a two-stage pipeline. Instructions I1 to I3 are stored at successive memory addresses, and I2 is a branch instruction. Let the branch target be instruction Ik . In clock cycle 3, the fetch operation for instruction I3 is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard I3, which has been incorrectly fetched, and fetch instruction Ik . In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

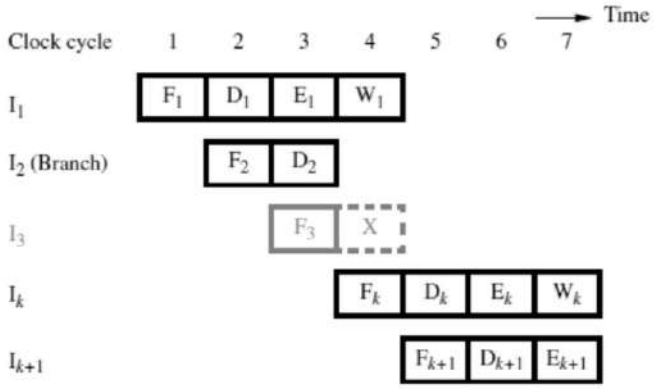


The time lost as a result of a branch instruction is often referred to as the branch penalty. In Figure, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure a shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E2. Instructions I3 and I4 must be discarded, and the target instruction, Ik , is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles. Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible

after an instruction is fetched. With this additional hardware, both of these tasks can be performed in step D2, leading to the sequence of events shown in Figure b. In this case, the branch penalty is only one clock cycle.



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

Instruction Queue and Prefetching

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. Typically, the instruction queue can store several instructions. A separate unit, which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit. This leads to the organization shown in Figure. The dispatch unit also performs the decoding function.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at

all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

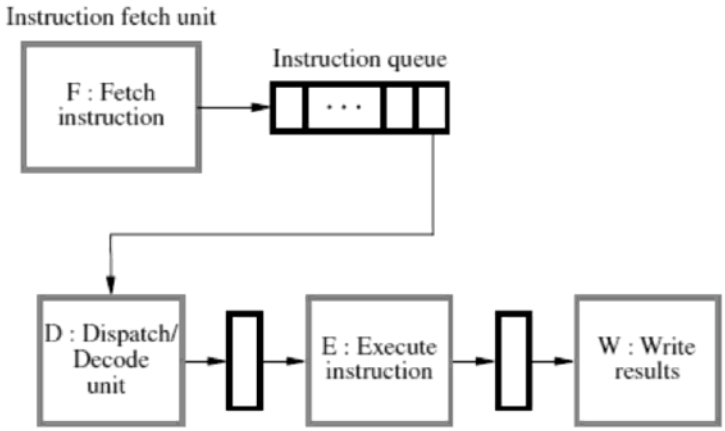
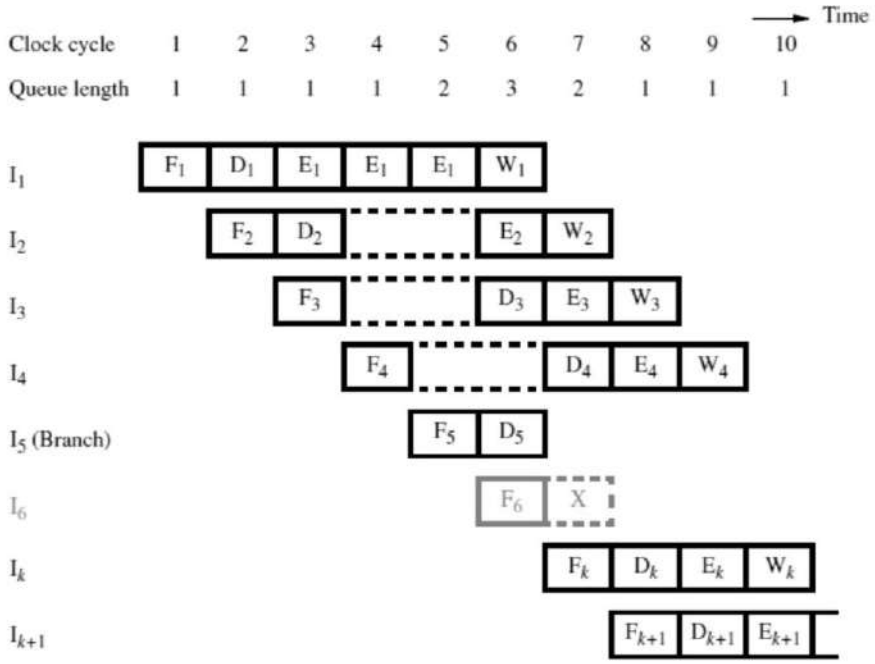


Figure illustrates how the queue length changes and how it affects the relationship between different pipeline stages. We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles. (There is both an F and a D step in each of these cycles.) Suppose that instruction I1 introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6. Instruction I5 is a branch instruction. Its target instruction, Ik , is fetched in cycle 7, and instruction I6 is discarded. The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I6. Instead, instruction I4 is dispatched from the queue to the decoding stage. After discarding I6, the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered. Now observe the sequence of instruction completions in Figure. Instructions I1, I2, I3, I4, and Ik complete execution in successive clock cycles. Hence, the branch instruction does not increase the overall execution time. This is because the instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as branch folding. Note that branch folding occurs only if at the time a branch instruction is encountered, at least one instruction is available in the queue other than the branch instruction. If only the branch instruction is in the queue, execution would proceed as in Figure b.

Therefore, it is desirable to arrange for the queue to be full most of the time, to ensure an adequate supply of instructions for processing. This can be achieved by increasing the rate at which the fetch unit reads instructions from the cache. In many processors, the width of the connection between the fetch unit and the instruction cache allows reading more than one instruction in each clock cycle. If the fetch unit replenishes the instruction queue quickly after a branch has occurred, the probability that branch folding will occur increases.

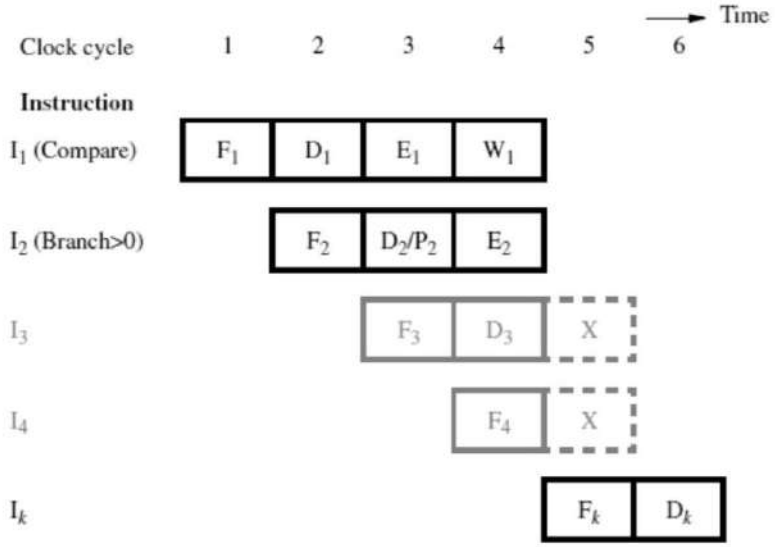


Having an instruction queue is also beneficial in dealing with cache misses. When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as the instruction queue is not empty. Meanwhile, the desired cache block is read from the main memory or from a secondary cache. When fetch operations are resumed, the instruction queue is refilled. If the queue does not become empty, a cache miss will have no effect on the rate of instruction execution. In summary, the instruction queue mitigates the impact of branch instructions on performance through the process of branch folding. It has a similar effect on stalls caused by cache misses. The effectiveness of this technique is enhanced when the instruction fetch unit is able to read more than one instruction at a time from the instruction cache.

Branch Prediction

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch

instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis. Speculative execution means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.



An incorrectly predicted branch is illustrated in Figure for a four-stage pipeline. The figure shows a Compare instruction followed by a Branch>0 instruction. Branch prediction takes place in cycle 3, while instruction I3 is being fetched. The fetch unit predicts that the branch will not be taken, and it continues to fetch instruction I4 as I3 enters the Decode stage. The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4. At this point, the instruction fetch unit realizes that the prediction was incorrect, and the two instructions in the execution pipe are purged. A new instruction, Ik , is fetched from the branch target address in clock cycle 5.

If branch outcomes were random, then half the branches would be taken. Then the simple approach of assuming that branches will not be taken would save the time lost to conditional branches 50 percent of the time. However, better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken, depending on the expected program behavior. For example, a branch instruction at the end of a loop causes a branch to the start of the loop for every pass through the loop except the last

one. Hence, it is advantageous to assume that this branch will be taken and to have the instruction fetch unit start to fetch instructions at the branch target address. On the other hand, for a branch instruction at the beginning of a program loop, it is advantageous to assume that the branch will not be taken.

A decision on which way to predict the result of the branch may be made in hardware by observing whether the target address of the branch is lower than or higher than the address of the branch instruction. A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch instructions of some processors, such as SPARC, include a branch prediction bit, which is set to 0 or 1 by the compiler to indicate the desired behavior. The instruction fetch unit checks this bit to predict whether the branch will be taken or not taken.

With either of these schemes, the branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called static branch prediction. Another approach in which the prediction decision may change depending on execution history is called dynamic branch prediction. The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded. In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

INFLUENCE ON INSTRUCTION SETS

We have seen that some instructions are much better suited to pipelined execution than others. For example, instruction side effects can lead to undesirable data dependencies. In this section, we examine the relationship between pipelined execution and machine instruction features. We discuss two key aspects of machine instructions—addressing modes and condition code flags.

Addressing Modes

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, autoincrement, and autodecrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered. In choosing the addressing modes to be implemented in a

pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline.

Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers. To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction Load X(R1),R2 takes five cycles to complete execution, as indicated in Figure. However, the instruction

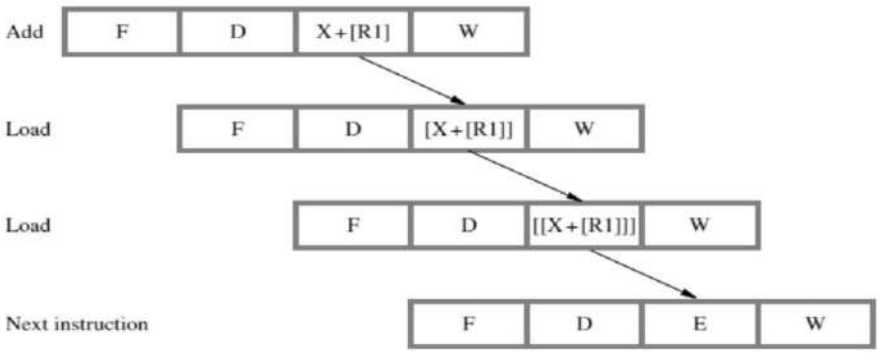
Load (R1),R2

can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

Load (X(R1)),R2

may be executed as shown in Figure 8.16a, assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location X+[R1] in clock cycle 4 and then to read location [X+[R1]] in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.

(a) Complex addressing mode



(b) Simple addressing mode

To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

Add #X,R1,R2
Load (R2),R2
Load (R2),R2

The Add instruction performs the operation R2←X+[R1]. The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure b. This example indicates that, in a pipelined processor, complex addressing modes that involve several accesses to the memory do not necessarily lead to faster execution. The main advantage of such modes is that they reduce the number of instructions needed to perform a given task and thereby reduce the program space needed in the main memory. Their main disadvantage is that their long execution times cause the pipeline to stall, thus reducing its effectiveness. They require more complex hardware to decode and execute them. Also, they are not convenient for compilers to work with.

The instruction sets of modern processors are designed to take maximum advantage of pipelined hardware. Because complex addressing modes are not suitable for pipelined execution, they should be avoided. The addressing modes used in modern processors often have the following features:

• Access to an operand does not require more than one access to the memory.
• Only load and store instructions access memory operands.
• The addressing modes used do not have side effects.

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register. Memory is accessed in the following cycle. None of these modes has any side effects, with one possible exception. Some architectures, such as ARM, allow the address computed in the index mode to be written back into the index register. This is a side effect that would not be allowed under the guidelines above. Note also that relative addressing can be used; this is a special case of indexed addressing in which the program counter is used as the index register. The three features just listed were first emphasized as part of the concept of RISC processors. The SPARC processor architecture, which adheres to these guidelines.

Condition Codes

The condition code flags either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

```
Add          R1,R2
Compare      R3,R4
Branch=0     . . .
```

(a) A program fragment

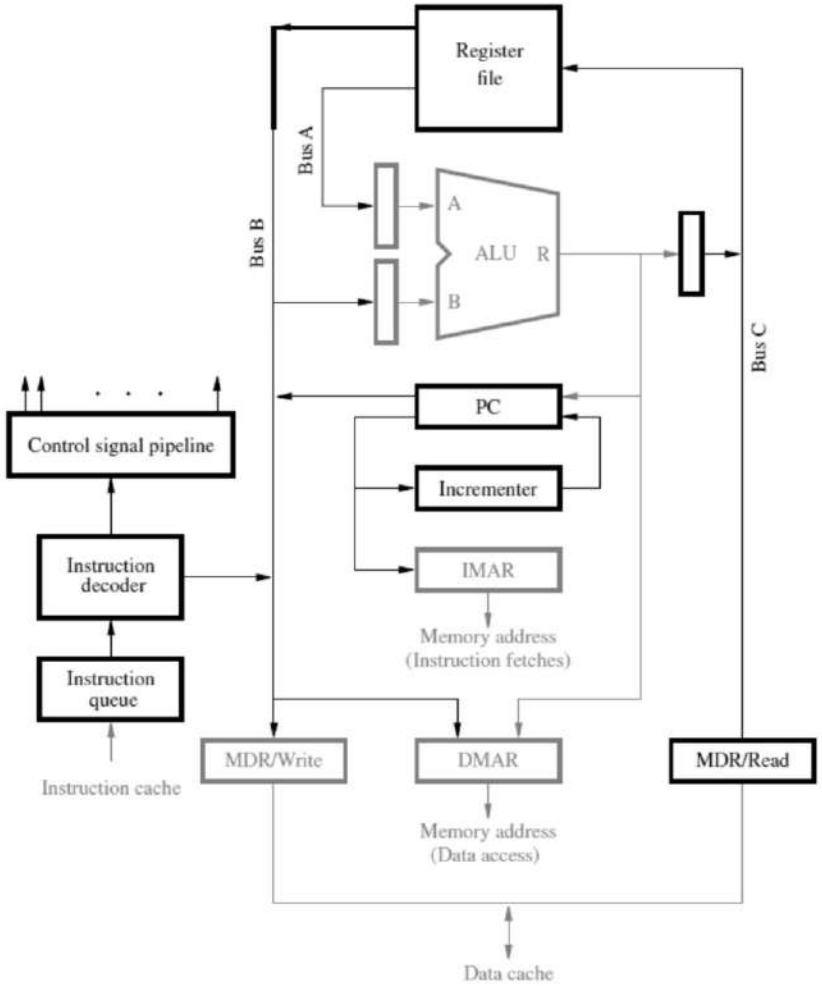
```
Compare      R3,R4
Add          R1,R2
Branch=0     . . .
```

(b) Instructions reordered

Consider the sequence of instructions in Figure a, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure. The execution time of the Branch instruction can be reduced by interchanging the Add and Compare instructions, as shown in Figure b. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes. These observations lead to two important conclusions about the way condition codes should be handled. First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility. Figure b shows the instructions reordered assuming that the condition code flags are affected only when this is explicitly stated as part of the instruction OP code. The SPARC and ARM architectures provide this flexibility.

3. DATA PATH AND CONTROL CONSIDERATIONS

Consider the three-bus structure from organization of the internal datapath of a processor suitable for pipelined execution, it can be modified as shown in Figure to support a 4-stage pipeline. The resources involved in stages F and E are shown in blue and those used in stages D and W in black. Operations in the data cache may happen during stage E or at a later stage, depending on the addressing mode and the implementation details.



Several important changes should be noted:

1. There are separate instruction and data caches that use separate address and data connections to the processor. This requires two versions of the MAR register, IMAR for accessing the instruction cache and DMAR for accessing the data cache.
2. The PC is connected directly to the IMAR, so that the contents of the PC can be transferred to IMAR at the same time that an independent ALU operation is taking place.

3. The data address in DMAR can be obtained directly from the register file or from the ALU to support the register indirect and indexed addressing modes.
4. Separate MDR registers are provided for read and write operations. Data can be transferred directly between these registers and the register file during load and store operations without the need to pass through the ALU.
5. Buffer registers have been introduced at the inputs and output of the ALU. These are registers SRC1, SRC2, and RSLT in Figure. Forwarding connections are not included in Figure. They may be added if desired.
6. The instruction register has been replaced with an instruction queue, which is loaded from the instruction cache.
7. The output of the instruction decoder is connected to the control signal pipeline. The need for buffering control signals and passing them from one stage to the next along with the instruction is discussed in Section 8.1. This pipeline holds the control signals in buffers B2 and B3.

The following operations can be performed independently in the processor of Figure:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

Because these operations do not use any shared resources, they can be performed simultaneously in any combination. The structure provides the flexibility required to implement the four-stage pipeline. For example, let I1, I2, I3, and I4 be a sequence of four instructions. As shown in Figure 8.2a, the following actions all happen during clock cycle 4:

- Write the result of instruction I1 into the register file
- Read the operands of instruction I2 from the register file
- Decode instruction I3
- Fetch instruction I4 and increment the PC.

SUPERSCALAR OPERATION

Pipelining makes it possible to execute instructions concurrently. Several instructions

are present in the pipeline at the same time, but they are in different stages of their execution. While one instruction is performing an ALU operation, another instruction is being decoded and yet another is being fetched from the memory. Instructions enter the pipeline in strict program order. In the absence of hazards, one instruction enters the pipeline and one instruction completes execution in each clock cycle. This means that the maximum throughput of a pipelined processor is one instruction per clock cycle.

A more aggressive approach is to equip the processor with multiple processing units to handle several instructions in parallel in each processing stage. With this arrangement, several instructions start execution in the same clock cycle, and the processor is said to use multiple-issue. Such processors are capable of achieving an instruction execution throughput of more than one instruction per cycle. They are known as superscalar processors. Many modern high-performance processors use this approach. We introduced the idea of an instruction queue. We pointed out that to keep the instruction queue filled, a processor should be able to fetch more than one instruction at a time from the cache. For superscalar operation, this arrangement is essential. Multiple-issue operation requires a wider path to the cache and multiple execution units. Separate execution units are provided for integer and floating-point instructions.

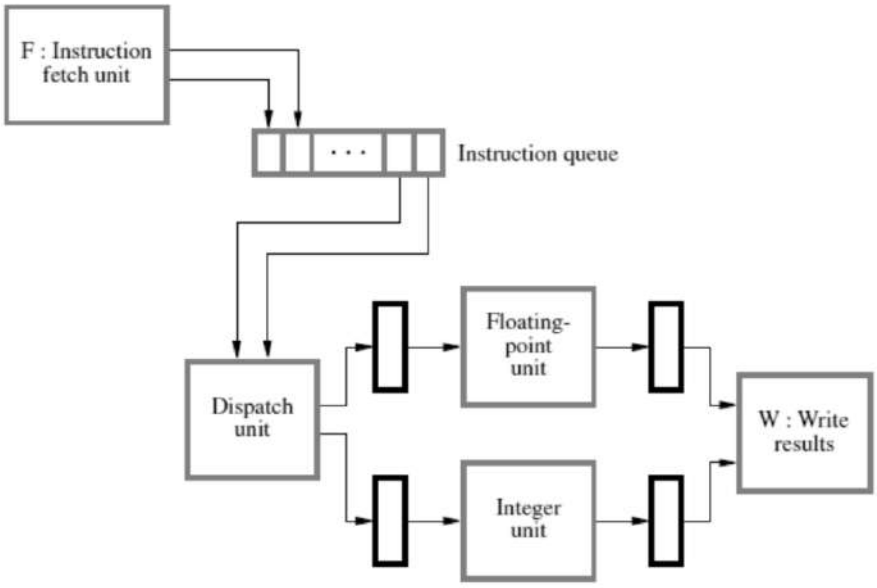
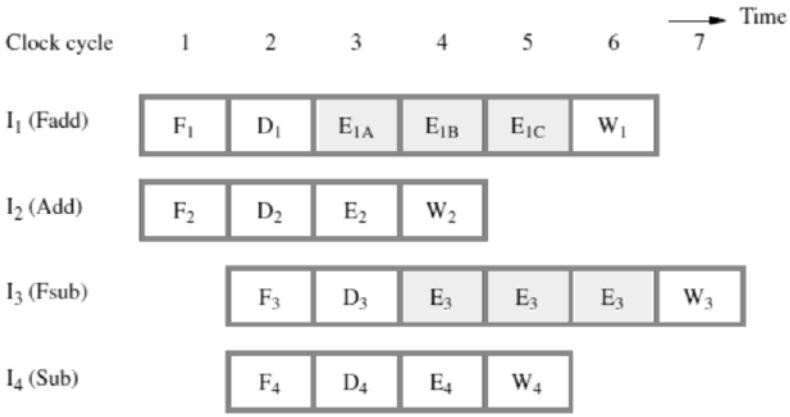


Figure shows an example of a processor with two execution units, one for integer and one for floating-point operations. The Instruction fetch unit is capable of reading two instructions at a time and storing them in the instruction queue. In each clock cycle, the Dispatch unit retrieves and decodes up to two instructions from the front of the queue. If there is one integer, one floating-point instruction, and no hazards, both instructions are dispatched in the same clock cycle.

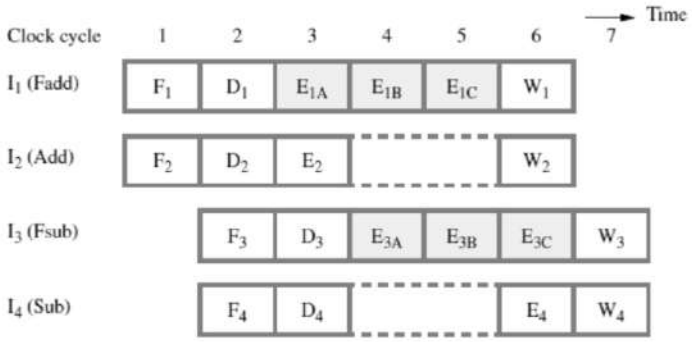


In a superscalar processor, the detrimental effect on performance of various hazards becomes even more pronounced. The compiler can avoid many hazards through judicious selection and ordering of instructions. For example, for the processor in Figure, the compiler should strive to interleave floating-point and integer instructions. This would enable the dispatch unit to keep both the integer and floating-point units busy most of the time. In general, high performance is achieved if the compiler is able to arrange program instructions to take maximum advantage of the available hardware units. Pipeline timing is shown in Figure. The blue shading indicates operations in the floating-point unit. The floating-point unit takes three clock cycles to complete the floating-point operation specified in I1. The integer unit completes execution of I2 in one clock cycle. We have also assumed that the floating-point unit is organized internally as a three-stage pipeline. Thus, it can still accept a new instruction in each clock cycle. Hence, instructions I3 and I4 enter the dispatch unit in cycle 3, and both are dispatched in cycle 4. The integer unit can receive a new instruction because instruction I2 has proceeded to the Write stage. Instruction I1 is still in the execution phase, but it has moved to the second stage of the internal pipeline in the floating-point unit. Therefore, instruction I3 can enter the first stage. Assuming that no hazards are encountered, the instructions complete execution as shown.

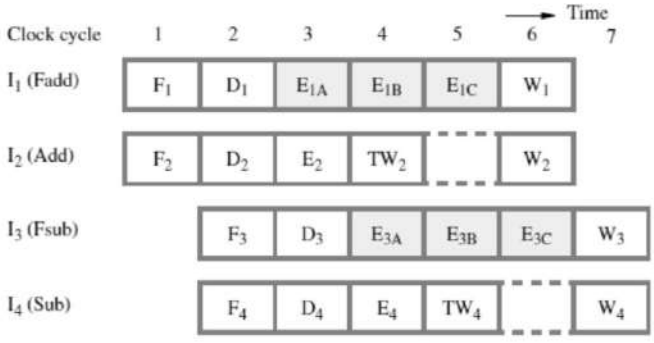
Out-of-Order Execution

In Figure, instructions are dispatched in the same order as they appear in the program. However, their execution is completed out of order. Does this lead to any problems? We have already discussed the issues arising from dependencies among instructions. For example, if instruction I2 depends on the result of I1, the execution of I2 will be delayed. As long as such dependencies are handled correctly, there is no reason to delay the execution of an instruction. However, a new complication arises when we consider the possibility of an instruction causing

an exception. **Exceptions** may be caused by a bus error during an operand fetch or by an illegal operation, such as an attempt to divide by zero. The results of I2 are written back into the register file in cycle 4. If instruction I1 causes an exception, program execution is in an inconsistent state. The program counter points to the instruction in which the exception occurred. However, one or more of the succeeding instructions have been executed to completion. If such a situation is permitted, the processor is said to have **imprecise exceptions**.



(a) Delayed write



(b) Using temporary registers

To guarantee a consistent state when exceptions occur, the results of the execution of instructions must be written into the destination locations strictly in program order. This means we must delay stepW2 in Figure 8.20 until cycle 6. In turn, the integer execution unit must retain the result of instruction I2, and hence it cannot accept instruction I4 until cycle 6, as shown in Figure 8.21a. If an exception occurs during an instruction, all subsequent instructions that may have been partially executed are discarded. This is called a **precise exception**. It is easier to provide precise exceptions in the case of external interrupts. When an external interrupt is received, the Dispatch unit stops reading new instructions from the instruction queue, and the instructions remaining in the queue are discarded. All instructions whose

execution is pending continue to completion. At this point, the processor and all its registers are in a consistent state, and interrupt processing can begin.

Execution Completion

It is desirable to use out-of-order execution, so that an execution unit is freed to execute other instructions as soon as possible. At the same time, instructions must be completed in program order to allow precise exceptions. These seemingly conflicting requirements are readily resolved if execution is allowed to proceed as shown in Figure, but the results are written into temporary registers. The contents of these registers are later transferred to the permanent registers in correct program order. This approach is illustrated in Figure b. Step TW is a write into a temporary register. Step W is the final step in which the contents of the temporary register are transferred into the appropriate permanent register. This step is often called the commitment step because the effect of the instruction cannot be reversed after that point. If an instruction causes an exception, the results of any subsequent instruction that has been executed would still be in temporary registers and can be safely discarded.

A temporary register assumes the role of the permanent register whose data it is holding and is given the same name. For example, if the destination register of I2 is R5, the temporary register used in step TW2 is treated as R5 during clock cycles 6 and 7. Its contents would be forwarded to any subsequent instruction that refers to R5 during that period. Because of this feature, this technique is called register renaming. Note that the temporary register is used only for instructions that follow I2 in program order. If an instruction that precedes I2 needs to read R5 in cycle 6 or 7, it would access the actual register R5, which still contains data that have not been modified by instruction I2. When out-of-order execution is allowed, a special control unit is needed to guarantee in-order commitment. This is called the commitment unit. It uses a queue called the reorder buffer to determine which instruction(s) should be committed next. Instructions are entered in the queue strictly in program order as they are dispatched for execution. When an instruction reaches the head of that queue and the execution of that instruction has been completed, the corresponding results are transferred from the temporary registers to the permanent registers and the instruction is removed from the queue. All resources that were assigned to the instruction, including the temporary registers, are released. The instruction is said to have been retired at this point. Because an instruction is retired only when it is at the head of the queue, all instructions that were dispatched before it must also have been retired. Hence, instructions may complete execution out of order, but they are retired in program order.

Dispatch Operation

We now return to the dispatch operation. When dispatching decisions are made, the dispatch unit must ensure that all the resources needed for the execution of an instruction are available. For example, since the results of an instruction may have to be written in a temporary register, the required register must be free, and it is reserved for use by that instruction as a part of the dispatch operation. A location in the reorder buffer must also be available for the instruction. When all the resources needed are assigned, including an appropriate execution unit, the instruction is dispatched. Should instructions be dispatched out of order? For example, if instruction I2 in Figure b is delayed because of a cache miss for a source operand, the integer unit will be busy in cycle 4, and I4 cannot be dispatched. Should I5 be dispatched instead? In principle this is possible, provided that a place is reserved in the reorder buffer for instruction I4 to ensure that all instructions are retired in the correct order. Dispatching instructions out of order requires considerable care. If I5 is dispatched while I4 is still waiting for some resource, we must ensure that there is no possibility of a deadlock occurring. A deadlock is a situation that can arise when two units, A and B, use a shared resource. Suppose that unit B cannot complete its task until unit A completes its task. At the same time, unit B has been assigned a resource that unit A needs. If this happens, neither unit can complete its task. Unit A is waiting for the resource it needs, which is being held by unit B. At the same time, unit B is waiting for unit A to finish before it can release that resource.

If instructions are dispatched out of order, a deadlock can arise as follows. Suppose that the processor has only one temporary register, and that when I5 is dispatched, that register is reserved for it. Instruction I4 cannot be dispatched because it is waiting for the temporary register, which, in turn, will not become free until instruction I5 is retired. Since instruction I5 cannot be retired before I4, we have a deadlock. To prevent deadlocks, the dispatcher must take many factors into account. Hence, issuing instructions out of order is likely to increase the complexity of the Dispatch unit significantly. It may also mean that more time is required to make dispatching decisions. For these reasons, most processors use only in-order dispatching. Thus, the program order of instructions is enforced at the time instructions are dispatched and again at the time instructions are retired. Between these two events, the execution of several instructions can proceed at their own speed, subject only to any interdependencies that may exist among instructions.

4. PERFORMANCE CONSIDERATIONS

We pointed out in Section 1.6 that the execution time, T , of a program that has a dynamic instruction count N is given by

$$T = (N \times S) / R$$

where S is the average number of clock cycles it takes to fetch and execute one instruction, and R is the clock rate. This simple model assumes that instructions are executed one after the other, with no overlap. A useful performance indicator is the instruction throughput, which is the number of instructions executed per second. For sequential execution, the throughput, Ps is given by

$$Ps = R/S$$

We examine the extent to which pipelining increases instruction throughput. The only real measure of performance is the total execution time of a program. Higher instruction throughput will not necessarily lead to higher performance if a larger number of instructions is needed to implement the desired task. For this reason, the SPEC ratings described in Chapter 1 provide a much better indicator when comparing two processors.

The four-stage pipeline may increase instruction throughput by a factor of four. In general, an n-stage pipeline has the potential to increase throughput n times. Thus, it would appear that the higher the value of n, the larger the performance gain. This leads to two questions:

• How much of this potential increase in instruction throughput can be realized in practice?

• What is a good value for n?

Any time a pipeline is stalled, the instruction throughput is reduced. Hence, the performance of a pipeline is highly influenced by factors such as branch and cache miss penalties. First, we discuss the effect of these factors on performance, and then we return to the question of how many pipeline stages should be used.

Effect of Instruction Hazards

The effects of various hazards have been examined qualitatively in the previous sections. We now assess the impact of cache misses and branch penalties in quantitative terms. Consider a processor that uses the four-stage pipeline. The clock rate, hence the time allocated to each step in the pipeline, is determined by the longest step. Let the delay through the ALU be the critical parameter. This is the time needed to add two integers. Thus, if the ALU delay is 2 ns, a clock of 500 MHz can be used. The on-chip instruction and data caches for this

processor should also be designed to have an access time of 2 ns. Under ideal conditions, this pipelined processor will have an instruction throughput, Pp, given by

$$Pp = R = 500 \text{ MIPS (million instructions per second)}$$

To evaluate the effect of cache misses, we use the same parameters. The cache miss penalty, Mp, in that system is computed to be 17 clock cycles. Let TI be the time between two successive instruction completions. For sequential execution, TI = S. However, in the absence of hazards, a pipelined processor completes the execution of one instruction each clock cycle, thus, TI = 1 cycle. A cache miss stalls the pipeline by an amount equal to the cache miss penalty. This means that the value of TI increases by an amount equal to the cache miss penalty for the instruction in which the miss occurs. A cache miss can occur for either instructions or data. Consider a computer that has a shared cache for both instructions and data, and let d be the percentage of instructions that refer to data operands in the memory. The average increase in the value of TI as a result of cache misses is given by

$$\delta miss = ((1 - hi) + d(1 - hd)) \times Mp$$

where hi and hd are the hit ratios for instructions and data, respectively. Assume that 30 percent of the instructions access data in memory. With a 95-percent instruction hit rate and a 90-percent data hit rate, δmiss is given by

$$\delta miss = (0.05 + 0.3 \times 0.1) \times 17 = 1.36 \text{ cycles}$$

Taking this delay into account, the processor's throughput would be

$$Pp = R/TI = R/(1 + \delta miss) = 0.42R$$

Note that with R expressed in MHz, the throughput is obtained directly in millions of instructions per second. For R = 500 MHz, Pp = 210 MIPS. Let us compare this value to the throughput obtainable without pipelining. A processor that uses sequential execution requires four cycles per instruction. Its throughput would be

$$Ps = R / (4 + \delta miss) = 0.19R$$

For R = 500 MHz, Ps = 95 MIPS. Clearly, pipelining leads to significantly higher throughput. But the performance gain of 0.42/0.19 = 2.2 is only slightly better than one-half the ideal case. Reducing the cache miss penalty is particularly worthwhile in a pipelined processor. This can be achieved by introducing a secondary cache between the primary, on-chip cache and the memory. Assume that the time needed to transfer an 8-word block from the secondary cache is 10 ns. Hence, a miss in the primary cache for which the required block is found in the secondary cache introduces a penalty, Ms, of 5 cycles. In the case of a miss in the secondary cache, the full 17-cycle penalty (Mp) is still incurred. Hence, assuming a hit rate hs of 94 percent in the secondary cache, the average increase in TI is

$$\delta miss = ((1 - hi) + d(1 - hd)) \times (hs \times Ms + (1 - hs) \times Mp) = 0.46 \text{ cycle}$$

The instruction throughput in this case is 0.68R, or 340 MIPS. An equivalent nonpipelined processor would have a throughput of 0.22R, or 110 MIPS. Thus, pipelining provides a performance gain of 0.68/0.22 = 3.1. The values of 1.36 and 0.46 are, in fact, somewhat pessimistic, because we have assumed that every time a data miss occurs, the entire miss penalty is incurred. This is the case only if the instruction immediately following the instruction that references memory is delayed while the processor waits for the memory access to be completed. However, an optimizing compiler attempts to increase the distance between two instructions that create a dependency by placing other instructions between them whenever possible. Also, in a processor that uses an instruction queue, the cache miss penalty during instruction fetches may have a much reduced effect as the processor is able to dispatch instructions from the queue.

Number of Pipeline Stages

The fact that an n-stage pipeline may increase instruction throughput by a factor of *n* suggests that we should use a large number of stages. However, as the number of pipeline stages increases, so does the probability of the pipeline being stalled, because more instructions are being executed concurrently. Thus, dependencies between instructions that are far apart may still cause the pipeline to stall. Also, branch penalties may become more significant. For these reasons, the gain from increasing the value of n begins to diminish, and the associated cost is not justified.

Another important factor is the inherent delay in the basic operations performed by the processor. The most important among these is the ALU delay. In many processors, the cycle time of the processor clock is chosen such that one ALU operation can be completed in one cycle. Other operations are divided into steps that take about the same time as an add operation. It is also possible to use a pipelined ALU. For example, the ALU of the Compaq Alpha 21064 processor consists of a two-stage pipeline, in which each stage completes its operation in 5 ns. Many pipelined processors use four to six stages. Others divide instruction execution into smaller steps and use more pipeline stages and a faster clock. For example, the UltraSPARC II uses a 9-stage pipeline and Intel's Pentium Pro uses a 12-stage pipeline. The latest Intel processor, Pentium 4, has a 20-stage pipeline and uses a clock speed in the range 1.3 to 1.5 GHz. For fast operations, there are two pipeline stages in one clock cycle.

HANDLING EXCEPTIONS

The general actions needed to switch from User mode to the appropriate exception mode. The actions vary in detail, depending upon the exception and the exception mode entered. Here, we consider some of those details.

Pipelined Execution, the Program Counter, and the Status Register

The ARM processor overlaps the fetching and execution of successive instructions in order to increase instruction throughput. This technique is called pipelined instruction execution. During pipelined execution of instructions, updating of the program counter is done as follows. Suppose that the processor fetches instruction I1 from address A. The contents of PC are incremented toA+4, then execution of I1 is begun. Before the execution of I1 is completed, the processor fetches instruction I2 from address A+4, then increments PC to A+8.

Now assume that at the end of execution of I1 the processor detects that an ordinary interrupt request (IRQ) has been received. The processor performs the actions to enter the IRQ exception mode to service the interrupt. It copies the contents of CPSR into SPSR_irq and copies the contents of PC, which are now A+8, into the link register R14_irq. Instruction I2, which has been fetched but not yet fully executed, is discarded. This is the instruction to which the interrupt-service routine must return. The interrupt-service routine must subtract 4 from R14_irq before using its contents as the return address. The saved copy of the Status register must also be restored. The required actions are carried out by the single instruction

SUBS PC, R14_irq, #4

which subtracts 4 from R14_irq and stores the result into PC. The suffix S in the OP code normally means "set condition codes." But when the target register of the instruction is PC, the S suffix causes the processor to copy the contents of SPSR_irq into CPSR, thus completing the actions needed to return to the interrupted program.

Exception	Saved address*	Desired return address	Return instruction
Undefined instruction	PC+4	PC+4	MOVS PC, R14_und
Software interrupt	PC+4	PC+4	MOVS PC, R14_svc
Instruction Abort	PC+4	PC	SUBS PC, R14_abt, #4
Data Abort	PC+8	PC	SUBS PC, R14_abt, #8
IRQ	PC+4	PC	SUBS PC, R14_irq, #4
FIQ	PC+4	PC	SUBS PC, R14_fiq, #4

*PC is the address of the instruction that caused the exception. For IRQ and FIQ, it is the address of the first instruction not executed because of the interrupt.

In the case of a software interrupt triggered by execution of the SWI instruction, the value saved in R14_svc is the correct return address. Return from a software interrupt can be accomplished using the instruction

MOVS PC, R14_svc

that also copies the contents of SPSR_svc into CPSR. Table gives the correct return-address value and the instruction that can be used to return to the interrupted program for each of the exceptions in Table, except for powerup/reset, which abandons any currently executing program. Note that for a data access or instruction access violation, the return address is the address of the instruction that caused the exception, because it must be re-executed after the cause of the violation has been resolved.

Manipulating Status Register Bits

When the processor is running in a privileged mode, special Move instructions, MRS and MSR, can be used to transfer the contents of the current or saved processor status registers to or from a general-purpose register. For example,

MRS Rd, CPSR

copies the contents of CPSR into register Rd. Similarly,

MSR SPSR, Rm

copies the contents of register Rm into SPSR_mode. After status register contents have been loaded into a register, logic instructions can be used to manipulate individual bits. Then, the register contents can be copied back into the status register to effect the desired changes. For example, these steps can be used to set or clear interrupt-disable bits in an exception-service routine.

Nesting Exception-Service Routines

Recall that nesting of subroutines is facilitated by storing the contents of the link register in the stack frame associated with a subroutine that calls another subroutine. This action is not needed when an exception-service routine is interrupted by a higher-priority exception whose service routine runs in a different processor mode. This is because each mode has its own banked link register.

For example, suppose that an ordinary interrupt is being serviced by an IRQ-mode routine when an interrupt that requires fast servicing is received. The first routine is interrupted and the FIQ mode is entered to service the second interrupt. The return address for the program that was interrupted to service the IRQ interrupt remains unchanged in link register R14_irq.

The return address for the IRQ routine is stored in R14_fiq. Hence, the use of banked registers avoids overwriting saved return addresses, and these addresses do not need to be placed on the stack when nesting of exception routines occurs. However, if different exceptions are serviced in the same processor mode, then their return addresses will need to be saved if nesting is allowed.

CONCLUSION:

Two important features have been introduced in this chapter, pipelining and multiple issue. Pipelining enables us to build processors with instruction throughput approaching one instruction per clock cycle. Multiple issue makes possible superscalar operation, with instruction throughput of several instructions per clock cycle. The potential gain in performance can only be realized by careful attention to three aspects:

- The instruction set of the processor
- The design of the pipeline hardware
- The design of the associated compiler

It is important to appreciate that there are strong interactions among all three. High performance is critically dependent on the extent to which these interactions are taken into account in the design of a processor. Instruction sets that are particularly well-suited for pipelined execution are key features of modern processors.

Unit 4 - Memory System

Dr.A.Parivazhagan,
Associate Professor, CSE ,KARE

==========

OBJECTIVES:

In this lesson, you will be explained about the concept of various memories like Basic memory circuits, Organization of the main memory, Memory technology, Direct memory access as an I/O mechanism, Cache memory, which reduces the effective memory access time, Virtual memory, which increases the apparent size of the main memory and Magnetic and optical disks used for secondary storage.

CONTENTS:

1. Basic Concepts
    - Semiconductor RAM
2. ROM
    - Speed, Size and cost
3. Cache memories
    - Improving cache performance, Virtual memory, Memory management requirements
4. Associative memories, Secondary storage devices

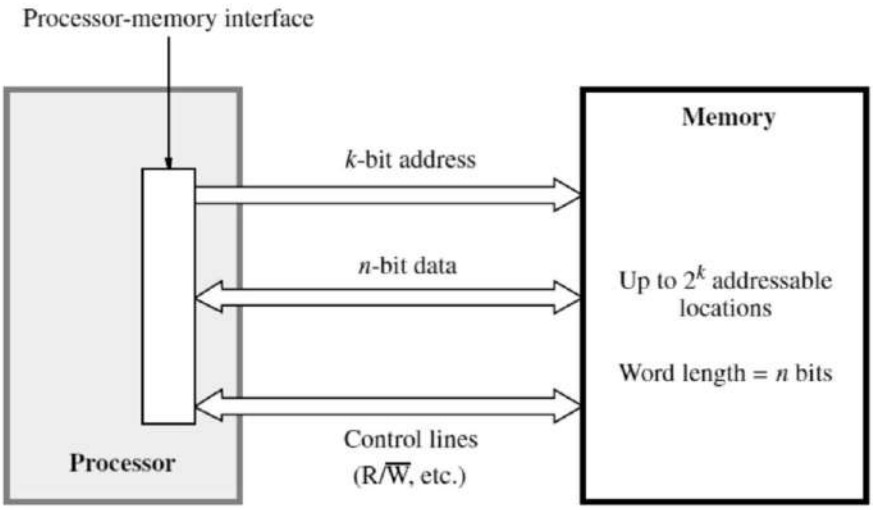
Introduction

Programs and the data they operate on are held in the memory of the computer. In this chapter, we discuss how this vital part of the computer operates. By now, the reader appreciates that the execution speed of programs is highly dependent on the speed with which instructions and data can be transferred between the processor and the memory. It is also important to have sufficient memory to facilitate execution of large programs having large amounts of data.

Ideally, the memory would be fast, large, and inexpensive. Unfortunately, it is impossible to meet all three of these requirements simultaneously. Increased speed and size are achieved at increased cost.

1. BASIC CONCEPTS

The maximum size of the memory that can be used in any computer is determined by the addressing scheme. For example, a computer that generates 16-bit addresses is capable of addressing up to 216 = 64K (kilo) memory locations. Machines whose instructions generate 32-bit addresses can utilize a memory that contains up to 232 = 4G (giga) locations, whereas machines with 64-bit addresses can access up to 264 = 16E (exa) ≈ 16 × 1018 locations. The number of locations represents the size of the address space of the computer. The memory is usually designed to store and retrieve data in word-length quantities. Consider, for example, a byte-addressable computer whose instructions generate 32-bit addresses. When a 32-bit address is sent from the processor to the memory unit, the high order 30 bits determine which word will be accessed. If a byte quantity is specified, the low-order 2 bits of the address specify which byte location is involved.



The connection between the processor and its memory consists of address, data, and control lines, as shown in Figure. The processor uses the address lines to specify the memory location involved in a data transfer operation, and uses the data lines to transfer the data. At the same time, the control lines carry the command indicating a Read or a Write operation and whether a byte or a word is to be transferred. The control lines also provide the necessary timing information and are used by the memory to indicate when it has completed the requested operation. When the processor-memory interface receives the memory's response, it asserts the MFC signal shown in Figure. This is the processor's internal control signal that indicates that

the requested memory operation has been completed. When asserted, the processor proceeds to the next step in its execution sequence.

A useful measure of the speed of memory units is the time that elapses between the initiation of an operation to transfer a word of data and the completion of that operation. This is referred to as the memory access time. Another important measure is the memory cycle time, which is the minimum time delay required between the initiation of two successive memory operations, for example, the time between two successive Read operations. The cycle time is usually slightly longer than the access time, depending on the implementation details of the memory unit. A memory unit is called a random-access memory (RAM) if the access time to any location is the same, independent of the location's address. This distinguishes such memory units from serial, or partly serial, access storage devices such as magnetic and optical disks. Access time of the latter devices depends on the address or position of the data. The technology for implementing computer memories uses semiconductor integrated circuits. The sections that follow present some basic facts about the internal structure and operation of such memories. We then discuss some of the techniques used to increase the effective speed and size of the memory.

Cache and Virtual Memory

The processor of a computer can usually process instructions and data faster than they can be fetched from the main memory. Hence, the memory access time is the bottleneck in the system. One way to reduce the memory access time is to use a cache memory. This is a small, fast memory inserted between the larger, slower main memory and the processor. It holds the currently active portions of a program and their data.

Virtual memory is another important concept related to memory organization. With this technique, only the active portions of a program are stored in the main memory, and the remainder is stored on the much larger secondary storage device. Sections of the program are transferred back and forth between the main memory and the secondary storage device in a manner that is transparent to the application program. As a result, the application program sees a memory that is much larger than the computer's physical main memory.

Block Transfers

The discussion above shows that data move frequently between the main memory and the cache and between the main memory and the disk. These transfers do not occur one word at a time. Data are always transferred in contiguous blocks involving tens, hundreds, or thousands of words. Data transfers between the main memory and high-speed devices such as a graphic display or an Ethernet interface also involve large blocks of data. Hence, a critical

parameter for the performance of the main memory is its ability to read or write blocks of data at high speed. This is an important consideration that we will encounter repeatedly as we discuss memory technology and the organization of the memory system.

SEMICONDUCTOR RAM MEMORIES

Semiconductor random-access memories (RAMs) are available in a wide range of speeds. Their cycle times range from 100 ns to less than 10 ns. In this section, we discuss the main characteristics of these memories. We start by introducing the way that memory cells are organized inside a chip.

Internal Organization of Memory Chips

Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. A possible organization is illustrated in Figure. Each row of cells constitutes a memory word, and all cells of a row are connected to a common line referred to as the word line, which is driven by the address decoder on the chip. The cells in each column are connected to a Sense/Write circuit by two bit lines, and the Sense/Write circuits are connected to the data input/output lines of the chip. During a Read operation, these circuits sense, or read, the information stored in the cells selected by a word line and place this information on the output data lines. During a Write operation, the Sense/Write circuits receive input data and store them in the cells of the selected word.

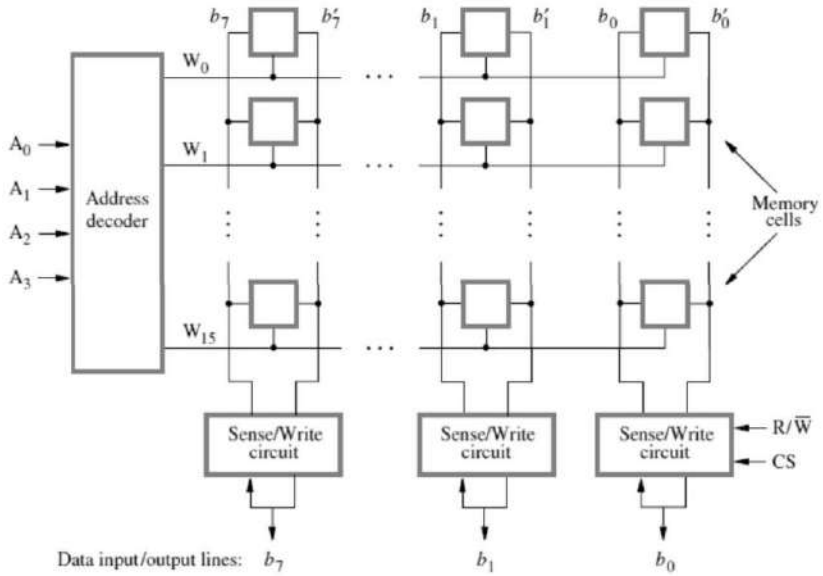


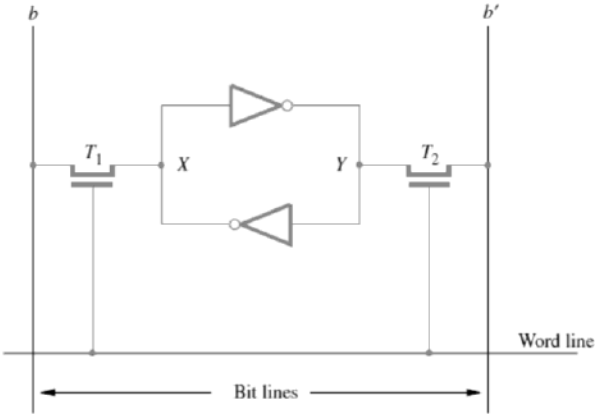
Figure is an example of a very small memory circuit consisting of 16 words of 8 bits each. This is referred to as a 16 × 8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that can be connected to

the data lines of a computer. Two control lines, R/W and CS, are provided. The R/W (Read/Write) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

The memory circuit in Figure stores 128 bits and requires 14 external connections for address, data, and control lines. It also needs two lines for power supply and ground connections. Consider now a slightly larger memory circuit, one that has 1K (1024) memory cells. This circuit can be organized as a 128 × 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a 1K×1 format. In this case, a 10-bit address is needed, but there is only one data line, resulting in 15 external connections.

STATIC MEMORIES

Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memories. Figure illustrates how a static RAM (SRAM) cell may be implemented. Two inverters are cross-connected to form a latch. The latch is connected to two bit lines by transistors T1 and T2. These transistors act as switches that can be opened or closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state. For example, if the logic value at point X is 1 and at point Y is 0, this state is maintained as long as the signal on the word line is at ground level. Assume that this state represents the value 1.



Read Operation

In order to read the state of the SRAM cell, the word line is activated to close switches T1 and T2. If the cell is in state 1, the signal on bit line b is high and the signal on bit line b' is low. The opposite is true if the cell is in state 0. Thus, b and b' are always complements of each

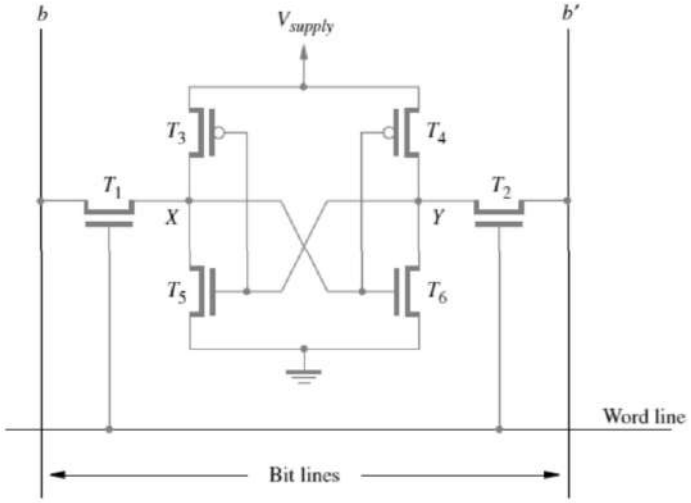
other. The Sense/Write circuit at the end of the two bit lines monitors their state and sets the corresponding output accordingly.

Write Operation

During a Write operation, the Sense/Write circuit drives bit lines b and b', instead of sensing their state. It places the appropriate value on bit line b and its complement on b' and activates the word line. This forces the cell into the corresponding state, which the cell retains when the word line is deactivated.

CMOS Cell

A CMOS realization of the cell in Figure is given in Figure. Transistor pairs (T3, T5) and (T4, T6) form the inverters in the latch (see Appendix A). The state of the cell is read or written as just explained. For example, in state 1, the voltage at point X is maintained high by having transistors T3 and T6 on, while T4 and T5 are off. If T1 and T2 are turned on, bit lines b and bwill have high and low signals, respectively.

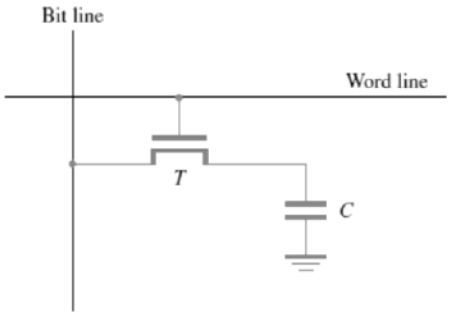


Continuous power is needed for the cell to retain its state. If power is interrupted, the cell's contents are lost. When power is restored, the latch settles into a stable state, but not necessarily the same state the cell was in before the interruption. Hence, SRAMs are said to be volatile memories because their contents are lost when power is interrupted. A major advantage of CMOS SRAMs is their very low power consumption, because current flows in the cell only when the cell is being accessed. Otherwise, T1, T2, and one transistor in each inverter are turned off, ensuring that there is no continuous electrical path between Vsupply and ground.

Static RAMs can be accessed very quickly. Access times on the order of a few nanoseconds are found in commercially available chips. SRAMs are used in applications where speed is of critical concern.

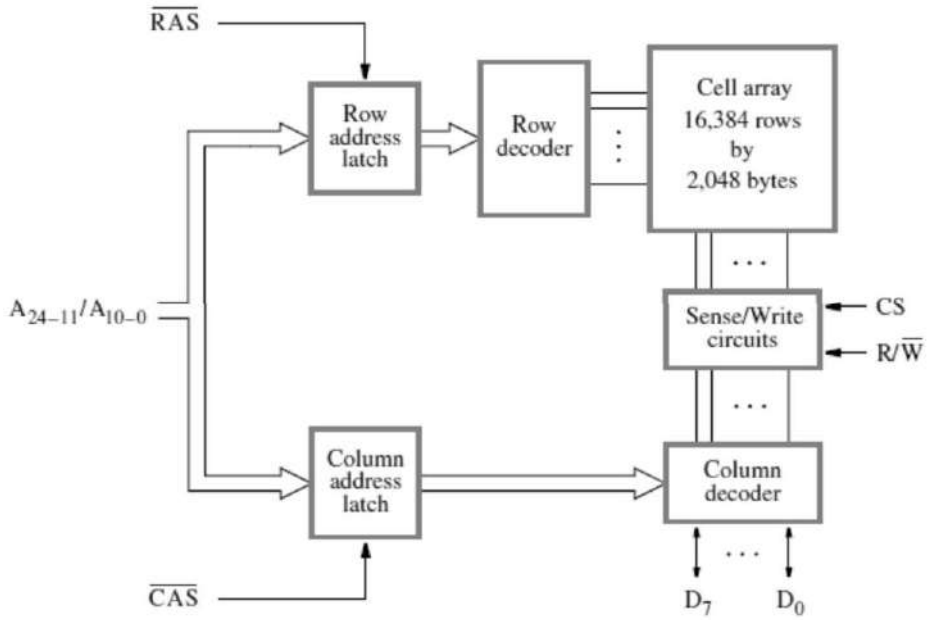
DYNAMIC RAMS

Static RAMs are fast, but their cells require several transistors. Less expensive and higher density RAMs can be implemented with simpler cells. But, these simpler cells do not retain their state for a long period, unless they are accessed frequently for Read or Write operations. Memories that use such cells are called dynamic RAMs (DRAMs). Information is stored in a dynamic memory cell in the form of a charge on a capacitor, but this charge can be maintained for only tens of milliseconds. Since the cell is required to store information for a much longer time, its contents must be periodically refreshed by restoring the capacitor charge to its full value. This occurs when the contents of the cell are read or when new information is written into it. An example of a dynamic memory cell that consists of a capacitor, C, and a transistor, T, is shown in Figure. To store information in this cell, transistor T is turned on and an appropriate voltage is applied to the bit line. This causes a known amount of charge to be stored in the capacitor.



After the transistor is turned off, the charge remains stored in the capacitor, but not for long. The capacitor begins to discharge. This is because the transistor continues to conduct a tiny amount of current, measured in picoamperes, after it is turned off. Hence, the information stored in the cell can be retrieved correctly only if it is read before the charge in the capacitor drops below some threshold value. During a Read operation, the transistor in a selected cell is turned on. A sense amplifier connected to the bit line detects whether the charge stored in the capacitor is above or below the threshold value. If the charge is above the threshold, the sense amplifier drives the bit line to the full voltage representing the logic value 1. As a result, the capacitor is recharged to the full charge corresponding to the logic value 1. If the sense amplifier detects that the charge in the capacitor is below the threshold value, it pulls the bit line to ground level to discharge the capacitor fully. Thus, reading the contents of a cell automatically refreshes its contents. Since the word line is common to all cells in a row, all cells in a selected row are read and refreshed at the same time.

A256-Megabit DRAM chip, configured as 32M × 8, is shown in Figure. The cells are organized in the form of a 16K × 16K array. The 16,384 cells in each row are divided into 2,048 groups of 8, forming 2,048 bytes of data. Therefore, 14 address bits are needed to select a row, and another 11 bits are needed to specify a group of 8 bits in the selected row. In total, a 25-bit address is needed to access a byte in this memory. The high-order 14 bits and the low-order 11 bits of the address constitute the row and column addresses of a byte, respectively. To reduce the number of pins needed for external connections, the row and column addresses are multiplexed on 14 pins. During a Read or a Write operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on an input control line called the Row Address Strobe (RAS). This causes a Read operation to be initiated, in which all cells in the selected row are read and refreshed.



Shortly after the row address is loaded, the column address is applied to the address pins and loaded into the column address latch under control of a second control line called the Column Address Strobe (CAS). The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits is selected. If the R/W control signal indicates a Read operation, the output values of the selected circuits are transferred to the data lines, D7−0. For a Write operation, the information on theD7−0 lines is transferred to the selected circuits, then used to overwrite the contents of the selected cells in the corresponding 8 columns. We should note that in commercial DRAM chips, the RAS and CAS control signals are active when low. Hence, addresses are latched when these signals change from high to low. The signals are shown in diagrams as RAS and CAS to indicate this fact. The timing of the operation of the

DRAM described above is controlled by the RAS and CAS signals. These signals are generated by a memory controller circuit external to the chip when the processor issues a Read or a Write command. During a Read operation, the output data are transferred to the processor after a delay equivalent to the memory's access time. Such memories are referred to as asynchronous DRAMs. The memory controller is also responsible for refreshing the data stored in the memory chips, as we describe later.

Fast Page Mode

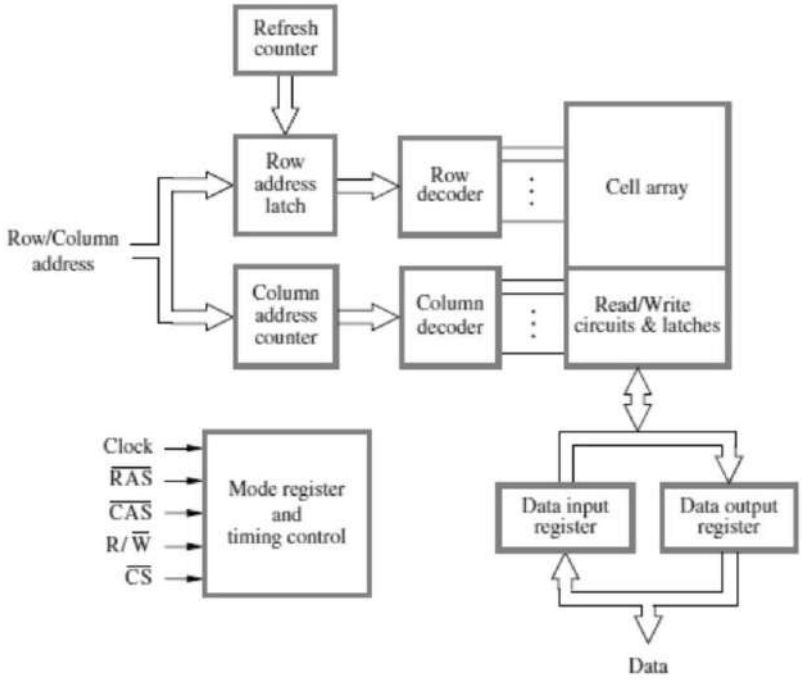
When the DRAM in Figure is accessed, the contents of all 16,384 cells in the selected row are sensed, but only 8 bits are placed on the data lines, D7−0. This byte is selected by the column address, bitsA10−0. A simple addition to the circuit makes it possible to access the other bytes in the same row without having to reselect the row. Each sense amplifier also acts as a latch. When a row address is applied, the contents of all cells in the selected row are loaded into the corresponding latches. Then, it is only necessary to apply different column addresses to place the different bytes on the data lines.

This arrangement leads to a very useful feature. All bytes in the selected row can be transferred in sequential order by applying a consecutive sequence of column addresses under the control of successive CAS signals. Thus, a block of data can be transferred at a much faster rate than can be achieved for transfers involving random addresses. The block transfer capability is referred to as the fast page mode feature. (A large block of data is often called a page.) It was pointed out earlier that the vast majority of main memory transactions involve block transfers. The faster rate attainable in the fast page mode makes dynamic RAMs particularly well suited to this environment.

SYNCHRONOUS DRAMS

In the early 1990s, developments in memory technology resulted in DRAMs whose operation is synchronized with a clock signal. Such memories are known as synchronous DRAMs (SDRAMs). Their structure is shown in Figure. The cell array is the same as in asynchronous DRAMs. The distinguishing feature of an SDRAM is the use of a clock signal, the availability of which makes it possible to incorporate control circuitry on the chip that provides many useful features. For example, SDRAMs have built-in refresh circuitry, with a refresh counter to provide the addresses of the rows to be selected for refreshing. As a result, the dynamic nature of these memory chips is almost invisible to the user. The address and data connections of an SDRAM may be buffered by means of registers, as shown in the figure. Internally, the Sense/Write amplifiers function as latches, as in asynchronous DRAMs. A Read operation causes the contents of all cells in the selected row to be loaded into these latches. The

data in the latches of the selected column are transferred into the data register, thus becoming available on the data output pins. The buffer registers are useful when transferring large blocks of data at very high speed. By isolating external connections from the chip's internal circuitry, it becomes possible to start a new access operation while data are being transferred to or from the registers.

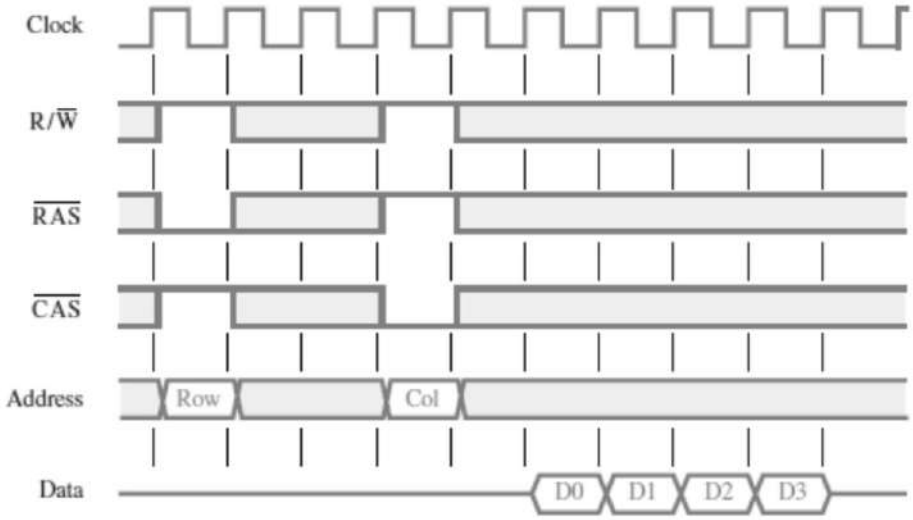


SDRAMs have several different modes of operation, which can be selected by writing control information into a mode register. For example, burst operations of different lengths can be specified. It is not necessary to provide externally-generated pulses on the CAS line to select successive columns. The necessary control signals are generated internally using a column counter and the clock signal. New data are placed on the data lines at the rising edge of each clock pulse.

Figure shows a timing diagram for a typical burst read of length 4. First, the row address is latched under control of the RAS signal. The memory typically takes 5 or 6 clock cycles (we use 2 in the figure for simplicity) to activate the selected row. Then, the column address is latched under control of the CAS signal. After a delay of one clock cycle, the first set of data bits is placed on the data lines. The SDRAM automatically increments the column address to access the next three sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

Synchronous DRAMs can deliver data at a very high rate, because all the control signals needed are generated inside the chip. The initial commercial SDRAMs in the 1990s were

designed for clock speeds of up to 133 MHz. As technology evolved, much faster SDRAM chips were developed. Today's SDRAMs operate with clock speeds that can exceed 1 GHz.



Latency and Bandwidth

Data transfers to and from the main memory often involve blocks of data. The speed of these transfers has a large impact on the performance of a computer system. The memory access time defined earlier is not sufficient for describing the memory's performance when transferring blocks of data. During block transfers, memory latency is the amount of time it takes to transfer the first word of a block. The time required to transfer a complete block depends also on the rate at which successive words can be transferred and on the size of the block. The time between successive words of a block is much shorter than the time needed to transfer the first word. For instance, in the timing diagram in Figure, the access cycle begins with the assertion of the RAS signal. The first word of data is transferred five clock cycles later. Thus, the latency is five clock cycles. If the clock rate is 500 MHz, then the latency is 10 ns. The remaining three words are transferred in consecutive clock cycles, at the rate of one word every 2 ns.

The example above illustrates that we need a parameter other than memory latency to describe the memory's performance during block transfers. A useful performance measure is the number of bits or bytes that can be transferred in one second. This measure is often referred to as the memory bandwidth. It depends on the speed of access to the stored data and on the number of bits that can be accessed in parallel. The rate at which data can be transferred to or from the memory depends on the bandwidth of the system interconnections. For this reason, the interconnections used always ensure that the bandwidth available for data transfers between the processor and the memory is very high.

Double-Data-Rate SDRAM

In the continuous quest for improved performance, faster versions of SDRAMs have been developed. In addition to faster circuits, new organizational and operational features make it possible to achieve high data rates during block transfers. The key idea is to take advantage of the fact that a large number of bits are accessed at the same time inside the chip when a row address is applied. Various techniques are used to transfer these bits quickly to the pins of the chip. To make the best use of the available clock speed, data are transferred externally on both the rising and falling edges of the clock. For this reason, memories that use this technique are called double-data-rate SDRAMs (DDR SDRAMs). Several versions of DDR chips have been developed. The earliest version is known as DDR. Later versions, called DDR2, DDR3, and DDR4, have enhanced capabilities. They offer increased storage capacity, lower power, and faster clock speeds. For example, DDR2 and DDR3 can operate at clock frequencies of 400 and 800 MHz, respectively. Therefore, they transfer data using the effective clock speeds of 800 and 1600 MHz, respectively.

Rambus Memory

The rate of transferring data between the memory and the processor is a function of both the bandwidth of the memory and the bandwidth of its connection to the processor. Rambus is a memory technology that achieves a high data transfer rate by providing a high-speed interface between the memory and the processor. One way for increasing the bandwidth of this connection is to use a wider data path. However, this requires more space and more pins, increasing system cost. The alternative is to use fewer wires with a higher clock speed. This is the approach taken by Rambus. Rambus technology competes directly with the DDR SDRAM technology. Each has certain advantages and disadvantages. Anontechnical consideration is that the specification of DDR SDRAM is an open standard that can be used free of charge. Rambus, on the other hand, is a proprietary scheme that must be licensed by chip manufacturers.

Refresh Overhead

A dynamic RAM cannot respond to read or write requests while an internal refresh operation is taking place. Such requests are delayed until the refresh cycle is completed. However, the time lost to accommodate refresh operations is very small. For example, consider an SDRAM in which each row needs to be refreshed once every 64 ms. Suppose that the minimum time between two row accesses is 50 ns and that refresh operations are arranged such that all rows of the chip are refreshed in 8K (8192) refresh cycles. Thus, it takes 8192 × 0.050

= 0.41 ms to refresh all rows. The refresh overhead is 0.41/64 = 0.0064, which is less than 1 percent of the total time available for accessing the memory.

Choice of Technology

The choice of a RAM chip for a given application depends on several factors. Foremost among these are the cost, speed, power dissipation, and size of the chip. Static RAMs are characterized by their very fast operation. However, their cost and bit density are adversely affected by the complexity of the circuit that realizes the basic cell. They are used mostly where a small but very fast memory is needed. Dynamic RAMs, on the other hand, have high bit densities and a low cost per bit. Synchronous DRAMs are the predominant choice for implementing the main memory.

2. ROM

Both static and dynamic RAM chips are volatile, which means that they retain information only while power is turned on. There are many applications requiring memory devices that retain the stored information when power is turned off. For example, the need to store a small program in such a memory, to be used to start the bootstrap process of loading the operating system from a hard disk into the main memory. The embedded applications are another important example. Many embedded applications do not use a hard disk and require non-volatile memories to store their software.

Different types of non-volatile memories have been developed. Generally, their contents can be read in the same way as for their volatile counterparts discussed above. But, a special writing process is needed to place the information into a non-volatile memory. Since its normal operation involves only reading the stored data, a memory of this type is called a read-only memory (ROM).

ROM

A memory is called a read-only memory, or ROM, when information can be written into it only once at the time of manufacture. Figure 8.11 shows a possible configuration for a ROM cell. A logic value 0 is stored in the cell if the transistor is connected to ground at point P; otherwise, a 1 is stored. The bit line is connected through a resistor to the power supply. To read the state of the cell, the word line is activated to close the transistor switch.

As a result, the voltage on the bit line drops to near zero if there is a connection between the transistor and ground. If there is no connection to ground, the bit line remains at the high voltage level, indicating a 1. A sense circuit at the end of the bit line generates the proper output value. The state of the connection to ground in each cell is determined when the chip is manufactured, using a mask with a pattern that represents the information to be stored.

PROM

Some ROM designs allow the data to be loaded by the user, thus providing a programmable ROM (PROM). Programmability is achieved by inserting a fuse at point P, in Figure. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible. PROMs provide flexibility and convenience not available with ROMs. The cost of preparing the masks needed for storing a particular information pattern makes ROMs cost effective only in large volumes. The alternative technology of PROMs

provides a more convenient and considerably less expensive approach, because memory chips can be programmed directly by the user.

EPROM

Another type of ROM chip provides an even higher level of convenience. It allows the stored data to be erased and new data to be written into it. Such an erasable, reprogrammable ROM is usually called an EPROM. It provides considerable flexibility during the development phase of digital systems. Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs or PROMs while software is being developed.

In this way, memory changes and updates can be easily made. An EPROM cell has a structure similar to the ROM cell in Figure. However, the connection to ground at point P is made through a special transistor. The transistor is normally turned off, creating an open switch. It can be turned on by injecting charge into it that becomes trapped inside. Thus, an EPROM cell can be used to construct a memory in the same way as the previously discussed ROM cell. Erasure requires dissipating the charge trapped in the transistors that form the memory cells. This can be done by exposing the chip to ultraviolet light, which erases the entire contents of the chip. To make this possible, EPROM chips are mounted in packages that have transparent windows.

EEPROM

An EPROM must be physically removed from the circuit for reprogramming. Also, the stored information cannot be erased selectively. The entire contents of the chip are erased when exposed to ultraviolet light. Another type of erasable PROM can be programmed, erased, and reprogrammed electrically. Such a chip is called an electrically erasable PROM, or EEPROM. It does not have to be removed for erasure. Moreover, it is possible to erase the cell contents selectively. One disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data, which increases circuit complexity. However, this disadvantage is outweighed by the many advantages of EEPROMs. They have replaced EPROMs in practice.

Flash Memory

An approach similar to EEPROM technology has given rise to flash memory devices. A flash cell is based on a single transistor controlled by trapped charge, much like an EEPROM cell. Also like an EEPROM, it is possible to read the contents of a single cell. The key difference is that, in a flash device, it is only possible to write an entire block of cells. Prior to

writing, the previous contents of the block are erased. Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and consume less power in their operation.

The low power consumption of flash memories makes them attractive for use in portable, battery-powered equipment. Typical applications include hand-held computers, cell phones, digital cameras, and MP3 music players. In hand-held computers and cell phones, a flash memory holds the software needed to operate the equipment, thus obviating the need for a disk drive. A flash memory is used in digital cameras to store picture data. In MP3 players, flash memories store the data that represent sound. Cell phones, digital cameras, and MP3 players are good examples of embedded systems. Single flash chips may not provide sufficient storage capacity for the applications. Larger memory modules consisting of a number of chips are used where needed. There are two popular choices for the implementation of such modules: flash cards and flash drives.

Flash Cards

One way of constructing a larger module is to mount flash chips on a small card. Such flash cards have a standard interface that makes them usable in a variety of products. A card is simply plugged into a conveniently accessible slot. Flash cards with a USB interface are widely used and are commonly known as memory keys. They come in a variety of memory sizes. Larger cards may hold as much as 32 Gbytes. A minute of music can be stored in about 1 Mbyte of memory, using the MP3 encoding format. Hence, a 32-Gbyte flash card can store approximately 500 hours of music.

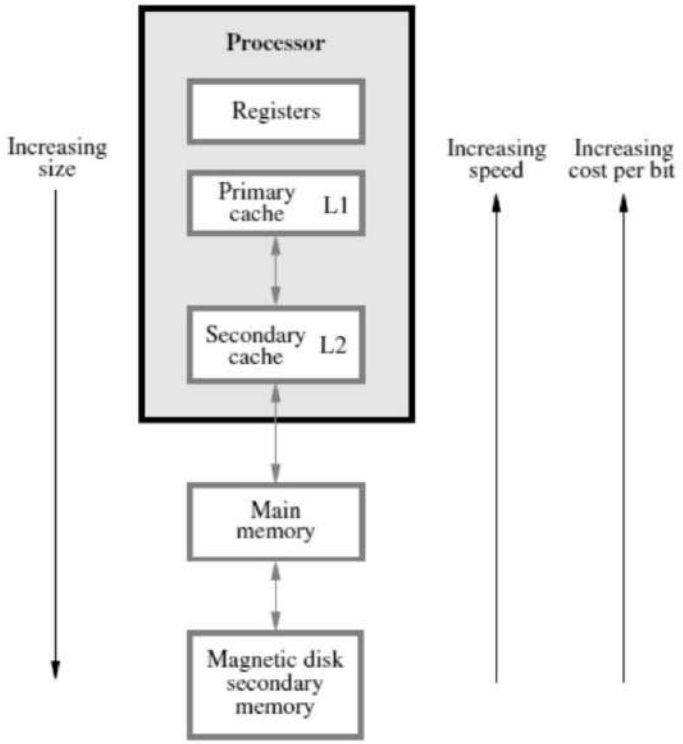
Flash Drives

Larger flash memory modules have been developed to replace hard disk drives, and hence are called flash drives. They are designed to fully emulate hard disks, to the point that they can be fitted into standard disk drive bays. However, the storage capacity of flash drives is significantly lower. Currently, the capacity of flash drives is on the order of 64 to 128 Gbytes. In contrast, hard disks have capacities exceeding a terabyte. Also, disk drives have a very low cost per bit.

The fact that flash drives are solid state electronic devices with no moving parts provides important advantages over disk drives. They have shorter access times, which result in a faster response. They are insensitive to vibration and they have lower power consumption, which makes them attractive for portable, battery-driven applications.

SPEED, SIZE AND COST – REVIEW ABOUT MEMORY HIERARCHY

An ideal memory would be fast, large, and inexpensive. It is clear that a very fast memory can be implemented using static RAM chips. But, these chips are not suitable for implementing large memories, because their basic cells are larger and consume more power than dynamic RAM cells. Although dynamic memory units with gigabyte capacities can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data. A solution is provided by using secondary storage, mainly magnetic disks, to provide the required memory space. Disks are available at a reasonable cost, and they are used extensively in computer systems. However, they are much slower than semiconductor memory units.



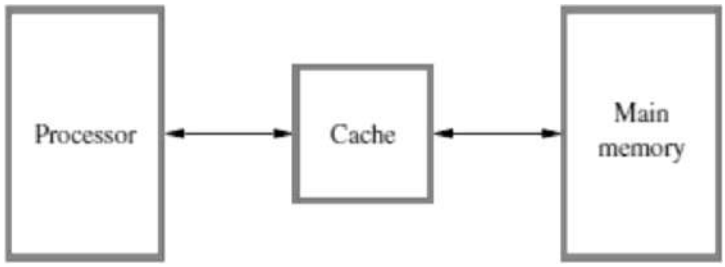
Characteristics	SRAM	DRAM	Magnetis Disk
Speed	Very Fast	Slower	Much slower than DRAM
Size	Large	Small	Small
Cost	Expensive	Less Expensive	Low price

In summary, a very large amount of cost-effective storage can be provided by magnetic disks, and a large and considerably faster, yet affordable, main memory can be built with dynamic RAM technology. This leaves the more expensive and much faster static RAM technology to be used in smaller units where speed is of the essence, such as in cache memories. huge amount of cost effective storage can be provided by magnetic disk; The main memory can be built with DRAM which leaves SRAM‟s to be used in smaller units where speed is of essence

Memory	Speed	Size	Cost
Registers	Very high	Lower	Very Lower
Primary cache	High	Lower	Low
Secondary cache	Low	Low	Low
Main memory	Lower than Seconadry cache	High	High
Secondary Memory	Very low	Very High	Very High

3. CACHE MEMORIES

The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference. Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other. The actual detailed pattern of instruction sequencing is not important—the point is that many instructions in localized areas of the program are executed repeatedly during some time period. This behaviour manifests itself in two ways: temporal and spatial. The first means that a recently executed instruction is likely to be executed again very soon. The spatial aspect means that instructions close to a recently executed instruction are also likely to be executed soon.



Conceptually, operation of a cache memory is very simple. The memory control circuitry is designed to take advantage of the property of locality of reference. Temporal locality suggests that whenever an information item, instruction or data, is first needed, this item should be brought into the cache, because it is likely to be needed again soon. Spatial locality suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that are located at adjacent addresses as well. The term cache block refers to a set of contiguous address locations of some size. Another term that is often used to refer to a cache block is a cache line. Consider the arrangement in Figure. When the processor issues a Read request, the contents of a block of memory words containing the location specified are transferred into the cache. Subsequently, when the program references any of the locations in this block, the desired contents are read directly from the cache. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function. When the cache is full and a memory word (instruction or data) that is not in the cache is

referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache's replacement algorithm.

Cache Hits

The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a read or write hit is said to have occurred. The main memory is not involved when there is a cache hit in a Read operation. For a Write operation, the system can proceed in one of two ways. In the first technique, called the write-through protocol, both the cache location and the main memory location are updated. The second technique is to update only the cache location and to mark the block containing it with an associated flag bit, often called the dirty or modified bit. The main memory location of the word is updated later, when the block containing this marked word is removed from the cache to make room for a new block. This technique is known as the write-back, or copy-back, protocol. The write-through protocol is simpler than the write-back protocol, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. The write-back protocol also involves unnecessary Write operations, because all words of the block are eventually written back, even if only a single word has been changed while the block was in the cache. The write-back protocol is used most often, to take advantage of the high speed with which data blocks can be transferred to memory chips.

Cache Misses

A Read operation for a word that is not in the cache constitutes a Read miss. It causes the block of words containing the requested word to be copied from the main memory into the cache. After the entire block is loaded into the cache, the particular word requested is forwarded to the processor. Alternatively, this word may be sent to the processor as soon as it is read from the main memory. The latter approach, which is called load-through, or early restart, reduces the processor's waiting time somewhat, at the expense of more complex circuitry.

When a Write miss occurs in a computer that uses the write-through protocol, the information is written directly into the main memory. For the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information. Resource limitations in a pipelined processor

can cause instruction execution to stall for one or more cycles. This can occur if a Load or Store instruction requests access to data in the memory at the same time that a subsequent instruction is being fetched. When this happens, instruction fetch is delayed until the data access operation is completed. To avoid stalling the pipeline, many processors use separate caches for instructions and data, making it possible for the two operations to proceed in parallel.

MAPPING FUNCTIONS

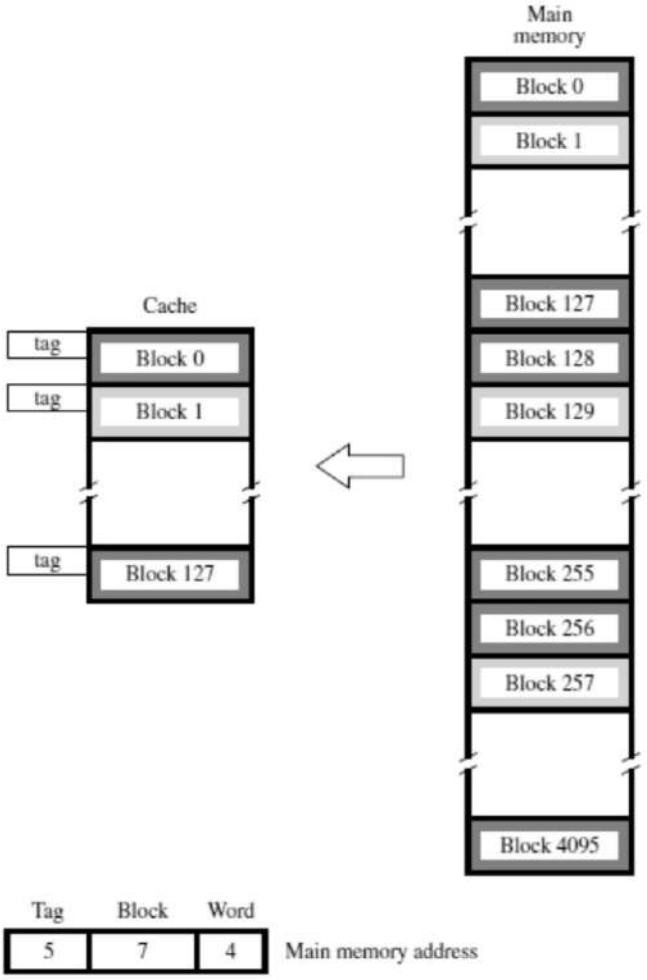
There are several possible methods for determining where memory blocks are placed in the cache. It is instructive to describe these methods using a specific small example. Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by a 16-bit address. The main memory has 64K words, which we will view as 4K blocks of 16 words each. For simplicity, we have assumed that consecutive addresses refer to consecutive words.

Direct Mapping

The simplest way to determine cache locations in which to store memory blocks is the direct-mapping technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure. Thus, whenever one of the main memory blocks 0, 128, 256, . . . is loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full. For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure. The low-order 4 bits select one of 16 words in a block. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The high-order 5 bits of the memory address of the block are stored in 5 tag bits associated with its location in the cache. The tag bits identify which of the 32 main memory blocks mapped into this cache position is currently resident in the cache. As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache. The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that

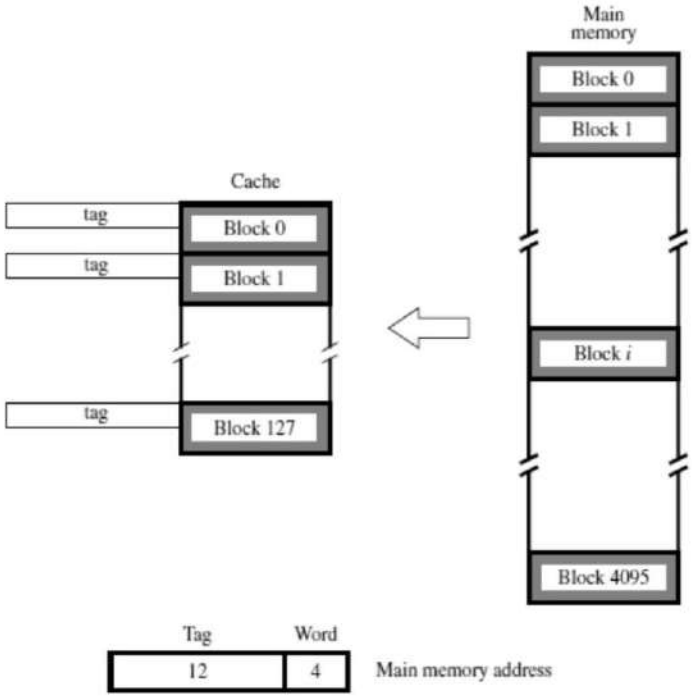
block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. The direct-mapping technique is easy to implement, but it is not very flexible.



Associative Mapping

Figure shows the most flexible mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the associative-mapping technique. It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced. Many replacement algorithms are possible. The complexity of an associative

cache is higher than that of a direct-mapped cache, because of the need to search all 128 tag patterns to determine whether a given block is in the cache. To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an associative search.

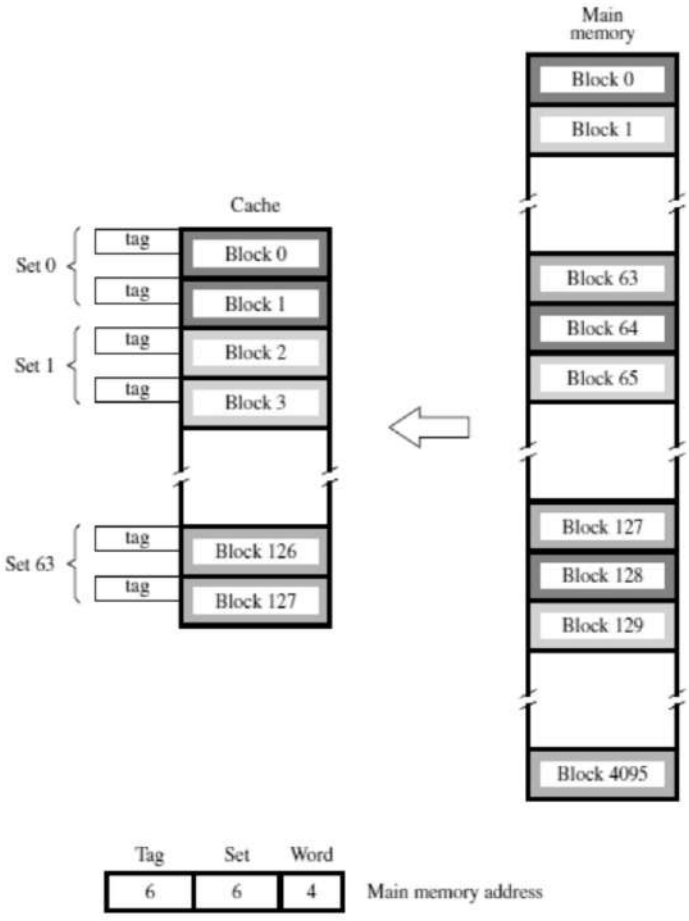


Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this set-associative-mapping technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-

associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping method. A cache that has k blocks per set is referred to as a k-way set-associative cache.



Stale Data

When power is first turned on, the cache contains no valid data. A control bit, usually called the valid bit, must be provided for each cache block to indicate whether the data in that block are valid. This bit should not be confused with the modified, or dirty, bit mentioned earlier. The valid bits of all cache blocks are set to 0 when power is initially applied to the system. Some valid bits may also be set to 0 when new programs or data are loaded from the disk into the main memory. Data transferred from the disk to the main memory using the DMA mechanism are usually loaded directly into the main memory, bypassing the cache. If the memory blocks being updated are currently in the cache, the valid bits of the corresponding cache blocks are set to 0. As program execution proceeds, the valid bit of a given cache block is set to 1 when a memory block is loaded into that location. The processor fetches data from

a cache block only if its valid bit is equal to 1. The use of the valid bit in this manner ensures that the processor will not fetch stale data from the cache.

A similar precaution is needed in a system that uses the write-back protocol. Under this protocol, new data written into the cache are not written to the memory at the same time. Hence, data in the memory do not always reflect the changes that may have been made in the cached copy. It is important to ensure that such stale data in the memory are not transferred to the disk. One solution is to flush the cache, by forcing all dirty blocks to be written back to the memory before performing the transfer. The operating system can do this by issuing a command to the cache before initiating the DMA operation that transfers the data to the disk. Flushing the cache does not affect performance greatly, because such disk transfers do not occur often. The need to ensure that two different entities (the processor and the DMA subsystems in this case) use identical copies of the data is referred to as a cache-coherence problem.

REPLACEMENT ALGORITHMS

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced. The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the least recently used (LRU) block, and the technique is called the LRU replacement algorithm.

To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds. Suppose it is required to track the LRU block of a four-block set in a set-associative cache. A 2-bit counter can be used for each block. When a hit occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged. When a miss occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one. When a miss occurs and

the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0. The other three block counters are incremented by one. It can be easily verified that the counter values of occupied blocks are always distinct.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache. Performance of the LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.

Several other replacement algorithms are also used in practice. An intuitively reasonable rule would be to remove the "oldest" block from a full set when a new block must be brought in. However, because this algorithm does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU algorithm in choosing the best blocks to remove. The simplest algorithm is to randomly choose the block to be overwritten. Interestingly enough, this simple algorithm has been found to be quite effective in practice.

IMPROVING CACHE PERFORMANCE

Two key factors in the commercial success of a computer are performance and cost; the best possible performance for a given cost is the objective. A common measure of success is the price/performance ratio. Performance depends on how fast machine instructions can be brought into the processor and how fast they can be executed. The main purpose of this hierarchy is to create a memory that the processor sees as having a short access time and a large capacity. When a cache is used, the processor is able to access instructions and data more quickly when the data from the referenced memory locations are in the cache. Therefore, the extent to which caches improve performance is dependent on how frequently the requested instructions and data are found in the cache. In this section, we examine this issue quantitatively.

Hit Rate and Miss Penalty

An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various levels of the hierarchy. Recall that a successful access to data in a cache is called a hit. The number of hits stated as a fraction of all attempted accesses is called the hit rate, and the miss rate is the number of misses stated as a fraction of attempted accesses. Ideally, the entire memory hierarchy would appear to the processor as a single memory unit that has the access time of the cache on the processor

chip and the size of the magnetic disk. How close we get to this ideal depends largely on the hit rate at different levels of the hierarchy. High hit rates well over 0.9 are essential for high-performance computers.

Performance is adversely affected by the actions that need to be taken when a miss occurs. A performance penalty is incurred because of the extra time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. During that period, the processor is stalled waiting for instructions or data. The waiting time depends on the details of the operation of the cache. For example, it depends on whether or not the load-through approach is used. We refer to the total access time seen by the processor when a miss occurs as the miss penalty.

Consider a system with only one level of cache. In this case, the miss penalty consists almost entirely of the time to access a block of data in the main memory. Let h be the hit rate, M the miss penalty, and C the time to access information in the cache. Thus, the average access time experienced by the processor is

$$t_{avg} = hC + (1 - h) M$$

How can the hit rate be improved? One possibility is to make the cache larger, but this entails increased cost. Another possibility is to increase the cache block size while keeping the total cache size constant, to take advantage of spatial locality. If all items in a larger block are needed in a computation, then it is better to load these items into the cache in a single miss, rather than loading several smaller blocks as a result of several misses. The high data rate achievable during block transfers is the main reason for this advantage. But larger blocks are effective only up to a certain size, beyond which the improvement in the hit rate is offset by the fact that some items may not be referenced before the block is ejected (replaced). Also, larger blocks take longer to transfer, and hence increase the miss penalty. Since the performance of a computer is affected positively by increased hit rate and negatively by increased miss penalty, block size should be neither too small nor too large. In practice, block sizes in the range of 16 to 128 bytes are the most popular choices. Finally, we note that the miss penalty can be reduced if the load-through approach is used when loading new blocks into the cache. Then, instead of waiting for an entire block to be transferred, the processor resumes execution as soon as the required word is loaded into the cache.

Caches on the Processor Chip

When information is transferred between different chips, considerable delays occur in driver and receiver gates on the chips. Thus, it is best to implement the cache on the processor chip. Most processor chips include at least one L1 cache. Often there are two separate L1

caches, one for instructions and another for data. In high-performance processors, two levels of caches are normally used, separate L1 caches for instructions and data and a larger L2 cache. These caches are often implemented on the processor chip. In this case, the L1 caches must be very fast, as they determine the memory access time seen by the processor. The L2 cache can be slower, but it should be much larger than the L1 caches to ensure a high hit rate. Its speed is less critical because it only affects the miss penalty of the L1 caches. A typical computer may have L1 caches with capacities of tens of kilobytes and an L2 cache of hundreds of kilobytes or possibly several megabytes.

Other Enhancements

In addition to the main design issues just discussed, several other possibilities exist for enhancing performance. We discuss three of them in this section.

Write Buffer

When the write-through protocol is used, each Write operation results in writing a new value into the main memory. If the processor must wait for the memory function to be completed, as we have assumed until now, then the processor is slowed down by all Write requests. Yet the processor typically does not need immediate access to the result of a Write operation; so it is not necessary for it to wait for the Write request to be completed To improve performance, a Write buffer can be included for temporary storage of Write requests. The processor places each Write request into this buffer and continues execution of the next instruction. The Write requests stored in the Write buffer are sent to the main memory whenever the memory is not responding to Read requests. It is important that the Read requests be serviced quickly, because the processor usually cannot proceed before receiving the data being read from the memory. Hence, these requests are given priority over Write requests.

The Write buffer may hold a number of Write requests. Thus, it is possible that a subsequent Read request may refer to data that are still in the Write buffer. To ensure correct operation, the addresses of data to be read from the memory are always compared with the addresses of the data in the Write buffer. In the case of a match, the data in the Write buffer are used. A similar situation occurs with the write-back protocol. In this case, Write commands issued by the processor are performed on the word in the cache. When a new block of data is to be brought into the cache as a result of a Read miss, it may replace an existing block that has some dirty data. The dirty block has to be written into the main memory. If the required write-back is performed first, then the processor has to wait for this operation to be completed before

the new block is read into the cache. It is more prudent to read the new block first. The dirty block being ejected from the cache is temporarily stored in the Write buffer and held there while the new block is being read. Afterwards, the contents of the buffer are written into the main memory. Thus, theWrite buffer also works well for the write-back protocol.

Prefetching

In the previous discussion of the cache mechanism, we assumed that new data are brought into the cache when they are first needed. Following a Read miss, the processor has to pause until the new data arrive, thus incurring a miss penalty. To avoid stalling the processor, it is possible to prefetch the data into the cache before they are needed. The simplest way to do this is through software. A special prefetch instruction may be provided in the instruction set of the processor. Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a Read miss. Aprefetch instruction is inserted in a program to cause the data to be loaded in the cache shortly before they are needed in the program. Then, the processor will not have to wait for the referenced data as in the case of a Read miss. The hope is that prefetching will take place while the processor is busy executing instructions that do not result in a Read miss, thus allowing accesses to the main memory to be overlapped with computation in the processor.

Prefetch instructions can be inserted into a program either by the programmer or by the compiler. Compilers are able to insert these instructions with good success for many applications. Software prefetching entails a certain overhead because inclusion of prefetch instructions increases the length of programs. Moreover, some prefetches may load into the cache data that will not be used by the instructions that follow. This can happen if the prefetched data are ejected from the cache by a Read miss involving other data. However, the overall effect of software prefetching on performance is positive, and many processors have machine instructions to support this feature. Prefetching can also be done in hardware, using circuitry that attempts to discover a pattern in memory references and prefetches data according to this pattern.

Lockup-Free Cache

Software prefetching does not work well if it interferes significantly with the normal execution of instructions. This is the case if the action of prefetching stops other accesses to the cache until the prefetch is completed. While servicing a miss, the cache is said to be locked. This problem can be solved by modifying the basic cache structure to allow the processor to

access the cache while a miss is being serviced. In this case, it is possible to have more than one outstanding miss, and the hardware must accommodate such occurrences.

A cache that can support multiple outstanding misses is called lockup-free. Such a cache must include circuitry that keeps track of all outstanding misses. This may be done with special registers that hold the pertinent information about these misses. We have used software prefetching to motivate the need for a cache that is not locked by a Read miss. A much more important reason is that in a pipelined processor, which overlaps the execution of several instructions, a Read miss caused by one instruction could stall the execution of other instructions. A lockup-free cache reduces the likelihood of such stalls.

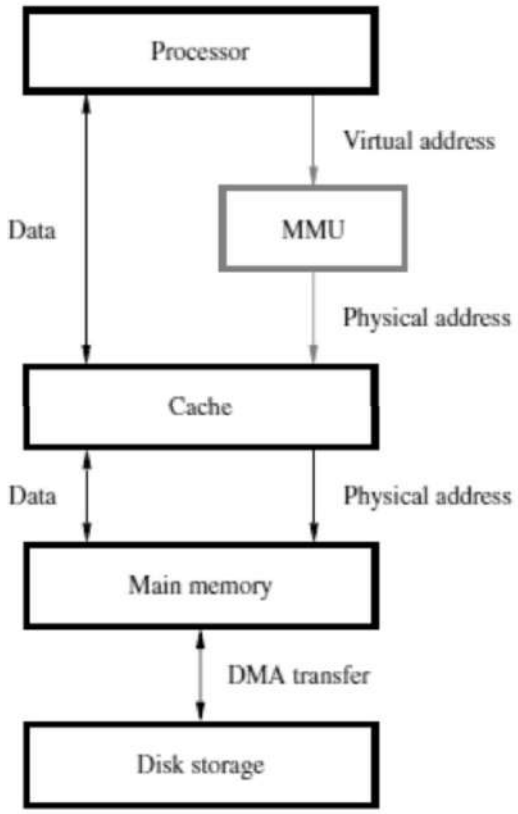
VIRTUAL MEMORY

In most modern computer systems, the physical main memory is not as large as the address space of the processor. For example, a processor that issues 32-bit addresses has an addressable space of 4G bytes. The size of the main memory in a typical computer with a 32-bit processor may range from 1G to 4G bytes. If a program does not completely fit into the main memory, the parts of it not currently being executed are stored on a secondary storage device, typically a magnetic disk. As these parts are needed for execution, they must first be brought into the main memory, possibly replacing other parts that are already in the memory. These actions are performed automatically by the operating system, using a scheme known as virtual memory. Application programmers need not be aware of the limitations imposed by the available main memory. They prepare programs using the entire address space of the processor.

Under a virtual memory system, programs, and hence the processor, reference instructions and data in an address space that is independent of the available physical main memory space. The binary addresses that the processor issues for either instructions or data are called virtual or logical addresses. These addresses are translated into physical addresses by a combination of hardware and software actions. If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. Otherwise, the contents of the referenced address must be brought into a suitable location in the memory before they can be used.

Figure shows a typical organization that implements virtual memory. A special hardware unit, called the Memory Management Unit (MMU), keeps track of which parts of the virtual address space are in the physical memory. When the desired data or instructions are in

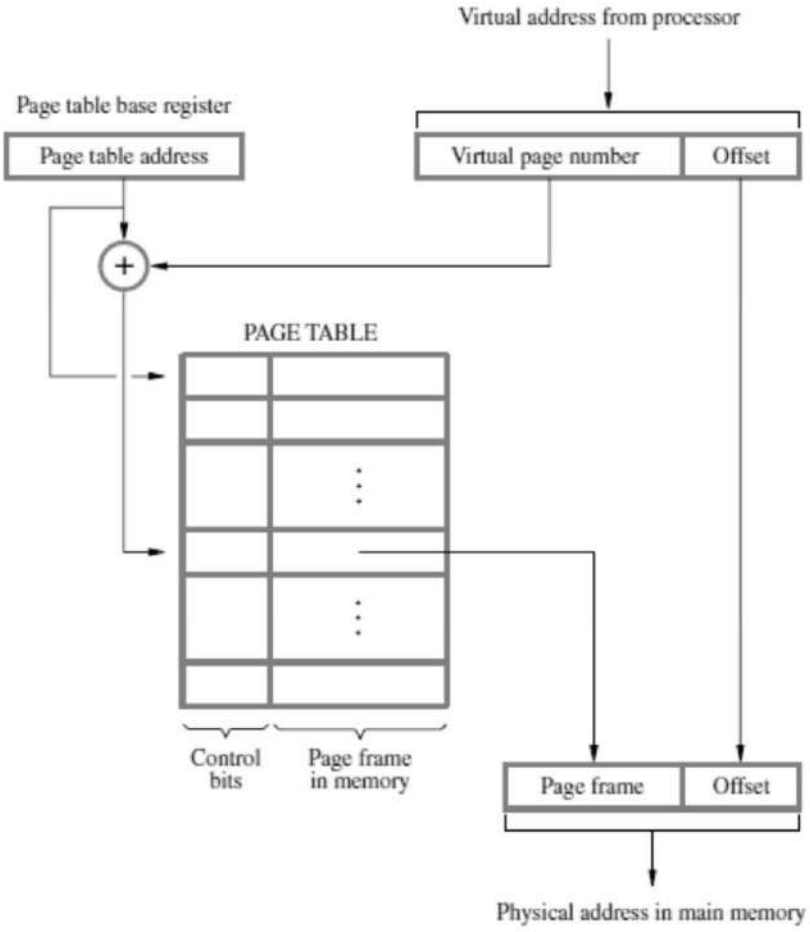
the main memory, the MMU translates the virtual address into the corresponding physical address. Then, the requested memory access proceeds in the usual manner. If the data are not in the main memory, the MMU causes the operating system to transfer the data from the disk to the memory. Such transfers are performed using the DMA scheme.



Address Translation

A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called pages, each of which consists of a block of words that occupy contiguous locations in the main memory. Pages commonly range from 2K to 16K bytes in length. They constitute the basic unit of information that is transferred between the main memory and the disk whenever the MMU determines that a transfer is required. Pages should not be too small, because the access time of a magnetic disk is much longer (several milliseconds) than the access time of the main memory. The reason for this is that it takes a considerable amount of time to locate the data on the disk, but once located, the data can be transferred at a rate of several megabytes per second. On the other hand, if pages are too large, it is possible that a substantial portion of a page may not be used, yet this unnecessary data will occupy valuable space in the main memory. This discussion clearly parallels the concepts introduced on cache memory. The cache bridges the speed gap between

the processor and the main memory and is implemented in hardware. The virtual-memory mechanism bridges the size and speed gaps between the main memory and secondary storage and is usually implemented in part by software techniques. Conceptually, cache techniques and virtual-memory techniques are very similar. They differ mainly in the details of their implementation.



A virtual-memory address-translation method based on the concept of fixed-length pages is shown schematically in Figure. Each virtual address generated by the processor, whether it is for an instruction fetch or an operand load/store operation, is interpreted- as a virtual page number (high-order bits) followed by an offset (low-order bits) that specifies the location of a particular byte (or word) within a page. Information about the main memory location of each page is kept in a page table. This information includes the main memory address where the page is stored and the current status of the page. An area in the main memory that can hold one page is called a page frame. The starting address of the page table is kept in a page table base register. By adding the virtual page number to the contents of this register,

the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the main memory.

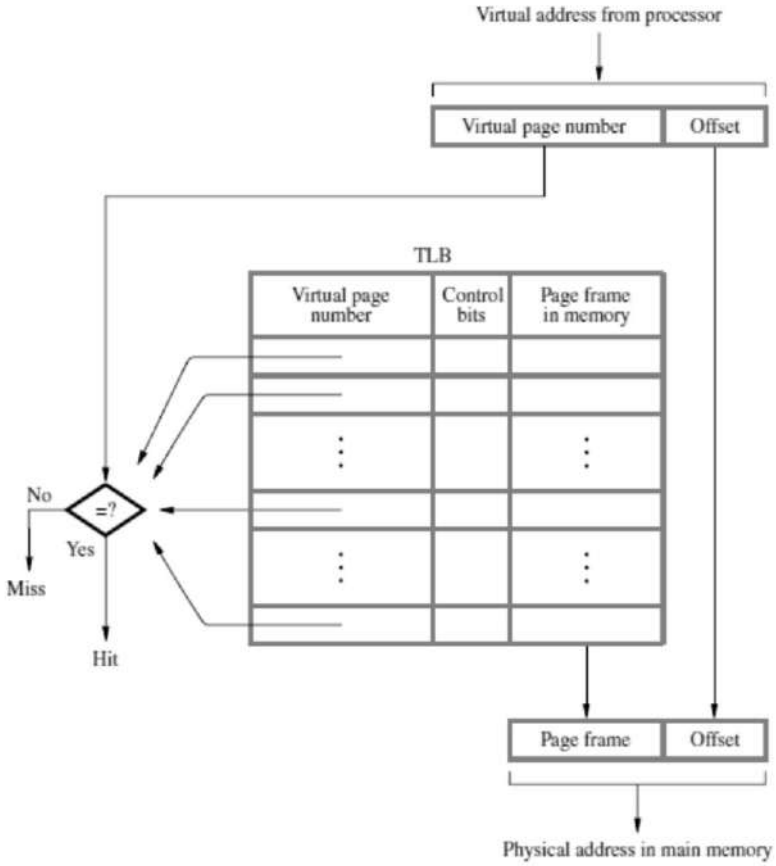
Each entry in the page table also includes some control bits that describe the status of the page while it is in the main memory. One bit indicates the validity of the page, that is, whether the page is actually loaded in the main memory. It allows the operating system to invalidate the page without actually removing it. Another bit indicates whether the page has been modified during its residency in the memory. As in cache memories, this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page. Other control bits indicate various restrictions that may be imposed on accessing the page. For example, a program may be given full read and write permission, or it may be restricted to read accesses only.

Translation Lookaside Buffer

The page table information is used by the MMU for every read and write access. Ideally, the page table should be situated within the MMU. Unfortunately, the page table may be rather large. Since the MMU is normally implemented as part of the processor chip, it is impossible to include the complete table within the MMU. Instead, a copy of only a small portion of the table is accommodated within the MMU, and the complete table is kept in the main memory. The portion maintained within the MMU consists of the entries corresponding to the most recently accessed pages. They are stored in a small table, usually called the Translation Lookaside Buffer (TLB). The TLB functions as a cache for the page table in the main memory. Each entry in the TLB includes a copy of the information in the corresponding entry in the page table. In addition, it includes the virtual address of the page, which is needed to search the TLB for a particular page. Figure shows a possible organization of a TLB that uses the associative-mapping technique. Set-associative mapped TLBs are also found in commercial products.

Address translation proceeds as follows. Given a virtual address, the MMU looks in the TLB for the referenced page. If the page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated. It is essential to ensure that the contents of the TLB are always the same as the contents of page tables in the memory. When the operating system changes the contents of a page table, it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose. When an entry is invalidated, the TLB acquires the new

information from the page table in the memory as part of the MMU's normal response to access misses.



Page Faults

When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred. The entire page must be brought from the disk into the memory before access can proceed. When it detects a page fault, the MMU asks the operating system to intervene by raising an exception (interrupt). Processing of the program that generated the page fault is interrupted, and control is transferred to the operating system. The operating system copies the requested page from the disk into the main memory. Since this process involves a long delay, the operating system may begin execution of another program whose pages are in the main memory. When page transfer is completed, the execution of the interrupted program is resumed.

When the MMU raises an interrupt to indicate a page fault, the instruction that requested the memory access may have been partially executed. It is essential to ensure that the interrupted program continues correctly when it resumes execution. There are two options. Either the execution of the interrupted instruction continues from the point of interruption, or

the instruction must be restarted. The design of a particular processor dictates which of these two options is used. If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. The problem of choosing which page to remove is just as critical here as it is in a cache, and the observation that programs spend most of their time in a few localized areas also applies. Because main memories are considerably larger than cache memories, it should be possible to keep relatively larger portions of a program in the main memory. This reduces the frequency of transfers to and from the disk. Concepts similar to the LRU replacement algorithm can be applied to page replacement, and the control bits in the page table entries can be used to record usage history. One simple scheme is based on a control bit that is set to 1 whenever the corresponding page is referenced (accessed). The operating system periodically clears this bit in all page table entries, thus providing a simple way of determining which pages have not been used recently.

A modified page has to be written back to the disk before it is removed from the main memory. It is important to note that the write-through protocol, which is useful in the framework of cache memories, is not suitable for virtual memory. The access time of the disk is so long that it does not make sense to access it frequently to write small amounts of data. Looking up entries in the TLB introduces some delay, slowing down the operation of the MMU. Here again we can take advantage of the property of locality of reference. It is likely that many successive TLB translations involve addresses on the same program page. This is particularly likely when fetching instructions. Thus, address translation time can be reduced by keeping the most recently used TLB entries in a few special registers that can be accessed quickly.

MEMORY MANAGEMENT REQUIREMENTS

In our discussion of virtual-memory concepts, we have tacitly assumed that only one large program is being executed. If all of the program does not fit into the available physical memory, parts of it (pages) are moved from the disk into the main memory when they are to be executed. Although we have alluded to software routines that are needed to manage this movement of program segments, we have not been specific about the details. Memory management routines are part of the operating system of the computer. It is convenient to assemble the operating system routines into a virtual address space, called the system space, that is separate from the virtual space in which user application programs reside. The latter space is called the user space. In fact, there may be a number of user spaces, one for each user. This is arranged by providing a separate page table for each user program. The MMU uses a

page table base register to determine the address of the table to be used in the translation process. Hence, by changing the contents of this register, the operating system can switch from one space to another. The physical main memory is thus shared by the active pages of the system space and several user spaces. However, only the pages that belong to one of these spaces are accessible at any given time.

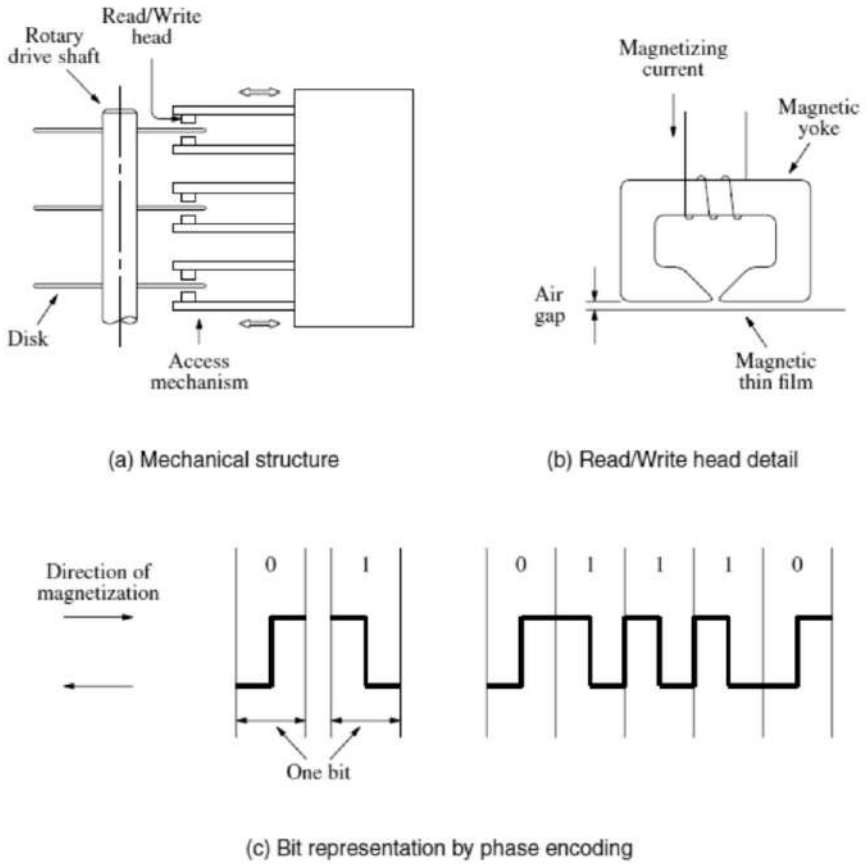
In any computer system in which independent user programs coexist in the main memory, the notion of protection must be addressed. No program should be allowed to destroy either the data or instructions of other programs in the memory. The needed protection can be provided in several ways. Let us first consider the most basic form of protection. Most processors can operate in one of two modes, the supervisor mode and the user mode. The processor is usually placed in the supervisor mode when operating system routines are being executed and in the user mode to execute user programs. In the user mode, some machine instructions cannot be executed. These are privileged instructions. They include instructions that modify the page table base register, which can only be executed while the processor is in the supervisor mode. Since a user program is executed in the user mode, it is prevented from accessing the page tables of other users or of the system space. It is sometimes desirable for one application program to have access to certain pages belonging to another program. The operating system can arrange this by causing these pages to appear in both spaces. The shared pages will therefore have entries in two different page tables. The control bits in each table entry can be set to control the access privileges granted to each program. For example, one program may be allowed to read and write a given page, while the other program may be given only read access.

4. ASSOCIATIVE MEMORIES & SECONDARY STORAGE DEVICES

The semiconductor memories discussed in the previous sections cannot be used to provide all of the storage capability needed in computers. Their main limitation is the cost per bit of stored information. The large storage requirements of most computer systems are economically realized in the form of magnetic and optical disks, which are usually referred to as secondary storage devices.

Magnetic Hard Disks

The storage medium in a magnetic-disk system consists of one or more disk platters mounted on a common spindle. A thin magnetic film is deposited on each platter, usually on both sides. The assembly is placed in a drive that causes it to rotate at a constant speed. The magnetized surfaces move in close proximity to read/write heads, as shown in Figure a. Data are stored on concentric tracks, and the read/write heads move radially to access different tracks.



Each read/write head consists of a magnetic yoke and a magnetizing coil, as indicated in Figure b. Digital information can be stored on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area

immediately underneath the head to switch to a direction parallel to the applied field. The same head can be used for reading the stored information. In this case, changes in the magnetic field in the vicinity of the head caused by the movement of the film relative to the yoke induce a voltage in the coil, which now serves as a sense coil. The polarity of this voltage is monitored by the control circuitry to determine the state of magnetization of the film. Only changes in the magnetic field under the head can be sensed during the Read operation. Therefore, if the binary states 0 and 1 are represented by two opposite states of magnetization, a voltage is induced in the head only at 0-to-1 and at 1-to-0 transitions in the bit stream. A long string of 0s or 1s causes an induced voltage only at the beginning and end of the string. Therefore, to determine the number of consecutive 0s or 1s stored, a clock must provide information for synchronization.

In some early designs, a clock was stored on a separate track, on which a change in magnetization is forced for each bit period. Using the clock signal as a reference, the data stored on other tracks can be read correctly. The modern approach is to combine the clocking information with the data. Several different techniques have been developed for such encoding. One simple scheme, depicted in Figure c, is known as phase encoding or Manchester encoding. In this scheme, changes in magnetization occur for each data bit, as shown in the figure. Clocking information is provided by the change in magnetization at the midpoint of each bit period. The drawback of Manchester encoding is its poor bit-storage density. The space required to represent each bit must be large enough to accommodate two changes in magnetization. We use the Manchester encoding example to illustrate how a self-clocking scheme may be implemented, because it is easy to understand. Other, more compact codes have been developed. They are much more efficient and provide better storage density. They also require more complex control circuitry. The discussion of such codes is beyond the scope of this book.

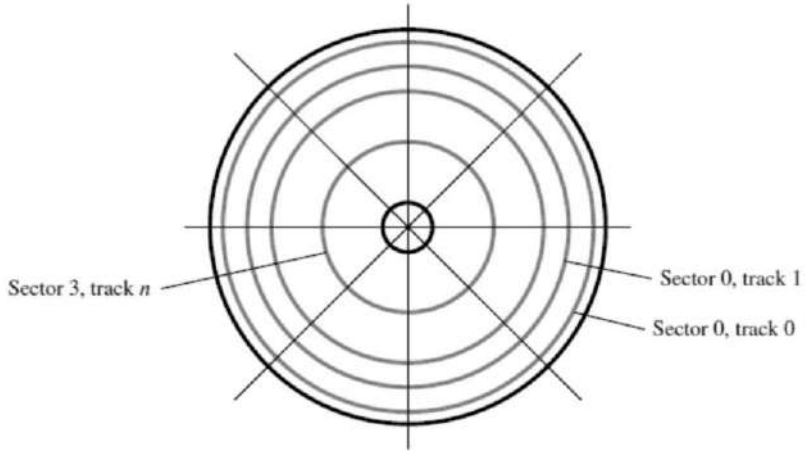
Read/write heads must be maintained at a very small distance from the moving disk surfaces in order to achieve high bit densities and reliable Read and Write operations. When the disks are moving at their steady rate, air pressure develops between the disk surface and the head and forces the head away from the surface. This force is counterbalanced by a spring-loaded mounting arrangement that presses the head toward the surface. The flexible spring connection between the head and its arm mounting permits the head to fly at the desired distance away from the surface in spite of any small variations in the flatness of the surface. In most modern disk units, the disks and the read/write heads are placed in a sealed, air-filtered enclosure. This approach is known as Winchester technology. In such units, the read/write

heads can operate closer to the magnetized track surfaces, because dust particles, which are a problem in unsealed assemblies, are absent. The closer the heads are to a track surface, the more densely the data can be packed along the track, and the closer the tracks can be to each other. Thus, Winchester disks have a larger capacity for a given physical size compared to unsealed units. Another advantage of Winchester technology is that data integrity tends to be greater in sealed units, where the storage medium is not exposed to contaminating elements.

The read/write heads of a disk system are movable. There is one head per surface. All heads are mounted on a comb-like arm that can move radially across the stack of disks to provide access to individual tracks, as shown in Figure a. To read or write data on a given track, the read/write heads must first be positioned over that track. The disk system consists of three key parts. One part is the assembly of disk platters, which is usually referred to as the disk. The second part comprises the electromechanical mechanism that spins the disk and moves the read/write heads; it is called the disk drive. The third part is the disk controller, which is the electronic circuitry that controls the operation of the system. The disk controller may be implemented as a separate module, or it may be incorporated into the enclosure that contains the entire disk system. We should note that the term disk is often used to refer to the combined package of the disk drive and the disk it contains. We will do so in the sections that follow, when there is no ambiguity in the meaning of the term.

Organization and Accessing of Data on a Disk

The organization of data on a disk is illustrated in Figure. Each surface is divided into concentric tracks, and each track is divided into sectors. The set of corresponding tracks on all surfaces of a stack of disks forms a logical cylinder. All tracks of a cylinder can be accessed without moving the read/write heads. Data are accessed by specifying the surface number, the track number, and the sector number. Read and Write operations always start at sector boundaries.



Data bits are stored serially on each track. Each sector may contain 512 or more bytes. The data are preceded by a sector header that contains identification (addressing) information used to find the desired sector on the selected track. Following the data, there are additional bits that constitute an error-correcting code (ECC). The ECC bits are used to detect and correct errors that may have occurred in writing or reading the data bytes. There is a small inter-sector gap that enables the disk control circuitry to distinguish easily between two consecutive sectors.

An unformatted disk has no information on its tracks. The formatting process writes markers that divide the disk into tracks and sectors. During this process, the disk controller may discover some sectors or even whole tracks that are defective. The disk controller keeps a record of such defects and excludes them from use. The formatting information comprises sector headers, ECC bits, and inter-sector gaps. The capacity of a formatted disk, after accounting for the formatting information overhead, is the proper indicator of the disk's storage capability. After formatting, the disk is divided into logical partitions. Figure indicates that each track has the same number of sectors, which means that all tracks have the same storage capacity. In this case, the stored information is packed more densely on inner tracks than on outer tracks. It is also possible to increase the storage density by placing more sectors on the outer tracks, which have longer circumference. This would be at the expense of more complicated access circuitry.

Access Time

There are two components involved in the time delay between the disk receiving an address and the beginning of the actual data transfer. The first, called the seek time, is the time required to move the read/write head to the proper track. This time depends on the initial position of the head relative to the track specified in the address. Average values are in the 5- to 8-ms range. The second component is the rotational delay, also called latency time, which is the time taken to reach the addressed sector after the read/write head is positioned over the correct track. On average, this is the time for half a rotation of the disk. The sum of these two delays is called the disk access time. If only a few sectors of data are accessed in a single operation, the access time is at least an order of magnitude longer than the time it takes to transfer the data.

Data Buffer/Cache

A disk drive is connected to the rest of a computer system using some standard interconnection scheme, such as SCSI or SATA. The interconnection hardware is usually capable of transferring data at much higher rates than the rate at which data can be read from disk tracks. An efficient way to deal with the possible differences in transfer rates is to include

a data buffer in the disk unit. The buffer is a semiconductor memory, capable of storing a few megabytes of data. The requested data are transferred between the disk tracks and the buffer at a rate dependent on the rotational speed of the disk. Transfers between the data buffer and the main memory can then take place at the maximum rate allowed by the interconnect between them.

The data buffer in the disk controller can also be used to provide a caching mechanism for the disk. When a Read request arrives at the disk, the controller can first check to see if the desired data are already available in the buffer. If so, the data are transferred to the memory in microseconds instead of milliseconds. Otherwise, the data are read from a disk track in the usual way, stored in the buffer, then transferred to the memory. Because of locality of reference, a subsequent request is likely to refer to data that sequentially follow the data specified in the current request. In anticipation of future requests, the disk controller may read more data than needed and place them into the buffer. When used as a cache, the buffer is typically large enough to store entire tracks of data. So, a possible strategy is to begin transferring the contents of the track into the data buffer as soon as the read/write head is positioned over the desired track.

Disk Controller

Operation of a disk drive is controlled by a disk controller circuit, which also provides an interface between the disk drive and the rest of the computer system. One disk controller may be used to control more than one drive. A disk controller that communicates directly with the processor contains a number of registers that can be read and written by the operating system. Thus, communication between the OS and the disk controller is achieved in the same manner as with any I/O interface. The disk controller uses the DMA scheme to transfer data between the disk and the main memory. Actually, these transfers are from/to the data buffer, which is implemented as a part of the disk controller module. The OS initiates the transfers by issuing Read and Write requests, which entail loading the controller's registers with the necessary addressing and control information. Typically, this information includes:

Main memory address—The address of the first main memory location of the block of words involved in the transfer.

Disk address—The location of the sector containing the beginning of the desired block of words.

Word count—The number of words in the block to be transferred. The disk address issued by the OS is a logical address. The corresponding physical address on the disk may be different. For example, bad sectors may be detected when the disk is formatted. The disk controller keeps

track of such sectors and maintains the mapping between logical and physical addresses. Normally, a few spare sectors are kept on each track, or on another track in the same cylinder, to be used as substitutes for the bad sectors. On the disk drive side, the controller's major functions are:

Seek—Causes the disk drive to move the read/write head from its current position to the desired track.

Read—Initiates a Read operation, starting at the address specified in the disk address register. Data read serially from the disk are assembled into words and placed into the data buffer for transfer to the main memory. The number of words is determined by the word count register.

Write—Transfers data to the disk, using a control method similar to that for Read operations.

Error checking—Computes the error correcting code (ECC) value for the data read from a given sector and compares it with the corresponding ECC value read from the disk. In the case of a mismatch, it corrects the error if possible; otherwise, it raises an interrupt to inform the OS that an error has occurred. During a Write operation, the controller computes the ECC value for the data to be written and stores this value on the disk.

Floppy Disks

The disks discussed above are known as hard or rigid disk units. *Floppy disks* are smaller, simpler, and cheaper disk units that consist of a flexible, removable, plastic *diskette* coated with magnetic material. The diskette is enclosed in a plastic jacket, which has an opening where the read/write head can be positioned. A hole in the centre of the diskette allows a spindle mechanism in the disk drive to position and rotate the diskette. The main feature of floppy disks is their low cost and shipping convenience. However, they have much smaller storage capacities, longer access times, and higher failure rates than hard disks. In recent years, they have largely been replaced by CDs, DVDs, and flash cards as portable storage media.

RAID Disk Arrays

Processor speeds have increased dramatically. At the same time, access times to disk drives are still on the order of milliseconds, because of the limitations of the mechanical motion involved. One way to reduce access time is to use multiple disks operating in parallel. They called it RAID, for Redundant Array of Inexpensive Disks. (Since all disks are now inexpensive, the acronym was later reinterpreted as Redundant Array of Independent Disks.) Using multiple disks also makes it possible to improve the reliability of the overall system. Different configurations were proposed, and many more have been developed since.

The basic configuration, known as RAID 0, is simple. A single large file is stored in several separate disk units by dividing the file into a number of smaller pieces and storing these

pieces on different disks. This is called data striping. When the file is accessed for a Read operation, all disks access their portions of the data in parallel. As a result, the rate at which the data can be transferred is equal to the data rate of individual disks times the number of disks. However, access time, that is, the seek and rotational delay needed to locate the beginning of the data on each disk, is not reduced. Since each disk operates independently, access times vary. Individual pieces of the data are buffered, so that the complete file can be reassembled and transferred to the memory as a single entity.

Various RAID configurations form a hierarchy, with each level in the hierarchy providing additional features. For example, RAID 1 is intended to provide better reliability by storing identical copies of the data on two disks rather than just one. The two disks are said to be mirrors of each other. If one disk drive fails, all Read and Write operations are directed to its mirror drive. Other levels of the hierarchy achieve increased reliability through various parity-checking schemes, without requiring a full duplication of disks. Some also have error-recovery capability. The RAID concept has gained commercial acceptance. RAID systems are available from many manufacturers for use with a variety of operating systems.

Optical Disks

Storage devices can also be implemented using optical means. The familiar compact disk (CD), used in audio systems, was the first practical application of this technology. Soon after, the optical technology was adapted to the computer environment to provide a high capacity read-only storage medium known as a CD-ROM.

The first generation of CDs was developed in the mid-1980s by the Sony and Philips companies. The technology exploited the possibility of using a digital representation for analog sound signals. To provide high-quality sound recording and reproduction, 16-bit samples of the analog signal are taken at a rate of 44,100 samples per second. Initially, CDs were designed to hold up to 75 minutes, requiring a total of about 3×109 bits (3 gigabits) of storage. Since then, higher-capacity devices have been developed.

CD Technology

The optical technology that is used for CD systems makes use of the fact that laser light can be focused on a very small spot. A laser beam is directed onto a spinning disk, with tiny indentations arranged to form a long spiral track on its surface. The indentations reflect the focused beam toward a photodetector, which detects the stored binary patterns. The laser emits a coherent light beam that is sharply focused on the surface of the disk. Coherent light consists of synchronized waves that have the same wavelength. If a coherent light beam is combined with another beam of the same kind, and the two beams are in phase, the result is a brighter

beam. But, if the waves of the two beams are 180 degrees out of phase, they cancel each other. Thus, a photodetector can be used to detect the beams. It will see a bright spot in the first case and a dark spot in the second case. Across-section of a small portion of a CD is shown in Figure a. The bottom layer is made of transparent polycarbonate plastic, which serves as a clear glass base. The surface of this plastic is programmed to store data by indenting it with pits. The unintended parts are called lands. A thin layer of reflecting aluminium material is placed on top of a programmed disk. The aluminium is then covered by a protective acrylic. Finally, the topmost layer is deposited and stamped with a label. The total thickness of the disk is 1.2 mm, almost all of it contributed by the polycarbonate plastic. The other layers are very thin. The laser source and the photodetector are positioned below the polycarbonate plastic. The emitted beam travels through the plastic layer, reflects off the aluminium layer, and travels back toward the photodetector. Note that from the laser side, the pits actually appear as bumps rising above the lands.

CD-Rewritable

The most flexible CDs are those that can be written multiple times by the user. They are known as CD-RWs (CD-ReWritables). The basic structure of CD-RWs is similar to the structure of CD-Rs. Instead of using an organic dye in the recording layer, an alloy of silver, indium, antimony, and tellurium is used. This alloy has interesting and useful behavior when it is heated and cooled. If it is heated above its melting point (500 degrees C) and then cooled down, it goes into an amorphous state in which it absorbs light. But, if it is heated only to about 200 degrees C and this temperature is maintained for an extended period, a process known as annealing takes place, which leaves the alloy in a crystalline state that allows light to pass through. If the crystalline state represents land area, pits can be created by heating selected spots past the melting point. The stored data can be erased using the annealing process, which returns the alloy to a uniform crystalline state. A reflective material is placed above the recording layer to reflect the light when the disk is read.

DVD Technology

The success of CD technology and the continuing quest for greater storage capability has led to the development of DVD (Digital Versatile Disk) technology. The first DVD standard was defined in 1996 by a consortium of companies, with the objective of being able to store a full-length movie on one side of a DVD disk. The physical size of a DVD disk is the same as that of CDs. The disk is 1.2 mm thick, and it is 120 mm in diameter. Its storage capacity is made much larger than that of CDs by several design changes:

- A red-light laser with a wavelength of 635 nm is used instead of the infrared light laser used in CDs, which has a wavelength of 780 nm. The shorter wavelength makes it possible to focus the light to a smaller spot.
- Pits are smaller, having a minimum length of 0.4 micron.
- Tracks are placed closer together; the distance between tracks is 0.74 micron.

Using these improvements leads to a DVD capacity of 4.7 Gbytes.

Magnetic Tape Systems

Magnetic tapes are suited for off-line storage of large amounts of data. They are typically used for backup purposes and for archival storage. Magnetic-tape recording uses the same principle as magnetic disks. The main difference is that the magnetic film is deposited on a very thin 0.5- or 0.25-inch wide plastic tape. Seven or nine bits (corresponding to one character) are recorded in parallel across the width of the tape, perpendicular to the direction of motion. A separate read/write head is provided for each bit position on the tape, so that all bits of a character can be read or written in parallel. One of the character bits is used as a parity bit. Data on the tape are organized in the form of records separated by gaps. Tape motion is stopped only when a record gap is underneath the read/write heads. The record gaps are long enough to allow the tape to attain its normal speed before the beginning of the next record is reached. If a coding scheme such as that is used for recording data on the tape, record gaps are identified as areas where there is no change in magnetization. This allows record gaps to be detected independently of the recorded data. To help users organize large amounts of data, a group of related records is called a file. The beginning of a file is identified by a file mark. The file mark is a special single- or multiple-character record, usually preceded by a gap longer than the inter-record gap. The first record following a file mark can be used as a header or identifier for the file. This allows the user to search a tape containing a large number of files for a particular file.

CONCLUSION:

The design of the memory hierarchy is critical to the performance of a computer system. Modern operating systems and application programs place heavy demands on both the capacity and speed of the memory. In this chapter, we presented the most important technological and organizational details of memory systems and how they have evolved to meet these demands. Developments in semiconductor technology have led to significant improvements in the speed

and capacity of memory chips, accompanied by a large decrease in the cost per bit. The performance of computer memories is enhanced further by the use of a memory hierarchy.

Today, a large yet affordable main memory is implemented with dynamic memory chips. One or more levels of cache memory are always provided. The introduction of the cache memory reduces significantly the effective memory access time seen by the processor. Virtual memory makes the main memory appear larger than the physical memory. Magnetic disks continue to be the primary technology for secondary storage. They provide enormous storage capacity, reaching and exceeding a trillion bytes on a single drive, with a very low cost per bit. But, flash semiconductor technology is beginning to compete effectively in some applications..