INTRODUCTION TO MICROPROCESSORS AND PROCESSOR ORGANIZATION

Basics of 8086 and 80386(Architecture and Register Set, Descriptor Tables), Addressing Modes, Memory management- Case Study of 80386, operating modes of 80386, Interrupts
Processor Basics: CPU organization, CPU Bus Organization: Central BUS, Buses on periphery, Additional features: RISC and CISC types representative commercial, Coprocessors, Processor organization, Register Organization

1. 8086 Microprocessor Features

- It is a 16-bit Microprocessor (μp).It's ALU, internal registers works with 16bit binary word.
- 8086 has a 20 bit address bus can access up to 2^{20}= 1 MB memory locations.
- 8086 has a 16bit data bus. It can read or write data to a memory/port either 16bits or 8 bit at a time.
- It can support up to 64K I/O ports.
- It provides 14, 16 -bit registers.
- Frequency range of 8086 is 6-10 MHz
- It has multiplexed address and data bus AD0- AD15 and A16 – A19.
- It requires single phase clock with 33% duty cycle to provide internal timing.
- It can prefetch up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.
- It requires +5V power supply.
- A 40 pin dual in line package.
- 8086 is designed to operate in two modes, Minimum mode and Maximum mode.
    - The minimum mode is selected by applying logic 1 to the MN / MX# input pin. This is a single microprocessor configuration.
    - The maximum mode is selected by applying logic 0 to the MN / MX# input pin. This is a multi micro processors configuration.

2. 8086 Microprocessor Block Diagram (Internal Architecture)

8086 has two blocks Bus Interface Unit (BIU) and Execution Unit (EU).The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue. EU executes instructions from the instruction system byte queue. Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining. This results in efficient use of the system bus and system performance. BIU contains Instruction queue, Segment registers, Instruction pointer, and Address adder. EU contains Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.

Bus Interface Unit: It provides a full 16 bit bidirectional data bus and 20 bit address bus. The bus interface unit is responsible for performing all external bus operations. Specifically it has the functions like instruction fetch, instruction queuing, operand fetch and storage, address relocation and bus control. The BIU uses a mechanism known as an instruction stream queue to implement pipeline architecture. This queue permits prefetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes

and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.

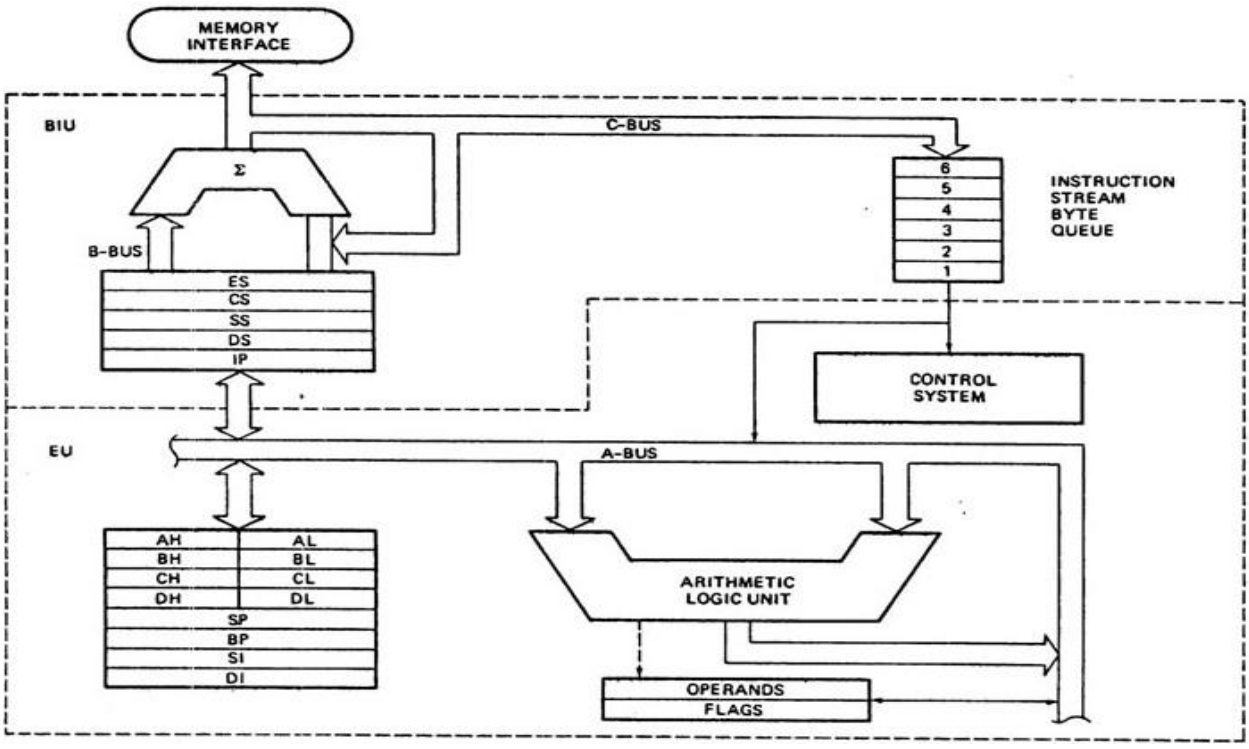


Figure 1 Block Diagram or Internal Architecture of 8086

These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle. After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output. The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to operand in memory. These intervals of no bus activity, which may occur between bus cycles, are known as idle state. If the BIU is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle. The BIU also contains a dedicated adder which is used to generate the 20bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.

For example: The physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register.

The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

Execution Unit: The Execution unit is responsible for decoding and executing all instructions. The EU extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bus cycles to memory or I/O and perform the operation specified by the instruction on the operands. During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction. If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue. When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions.

Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.

Maximum Mode System:

In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground. In this mode, the processor derives the status signal S2, S1, S0. Another chip called bus controller derives the control signal using this status information. In the maximum mode, there may be more than one microprocessor in the system configuration.

Minimum Mode System:

In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1. In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.

Memory Organization of 8086:

Segmentation is the process in which the main memory of the computer is logically divided into different segments and each segment has its own base address. It is basically used to enhance the speed of execution of the computer system, so that the processor is able to fetch and execute the data from the memory easily and fast.

Need for Segmentation –

The Bus Interface Unit (BIU) contains four 16 bit special purpose registers (mentioned below) called as Segment Registers.

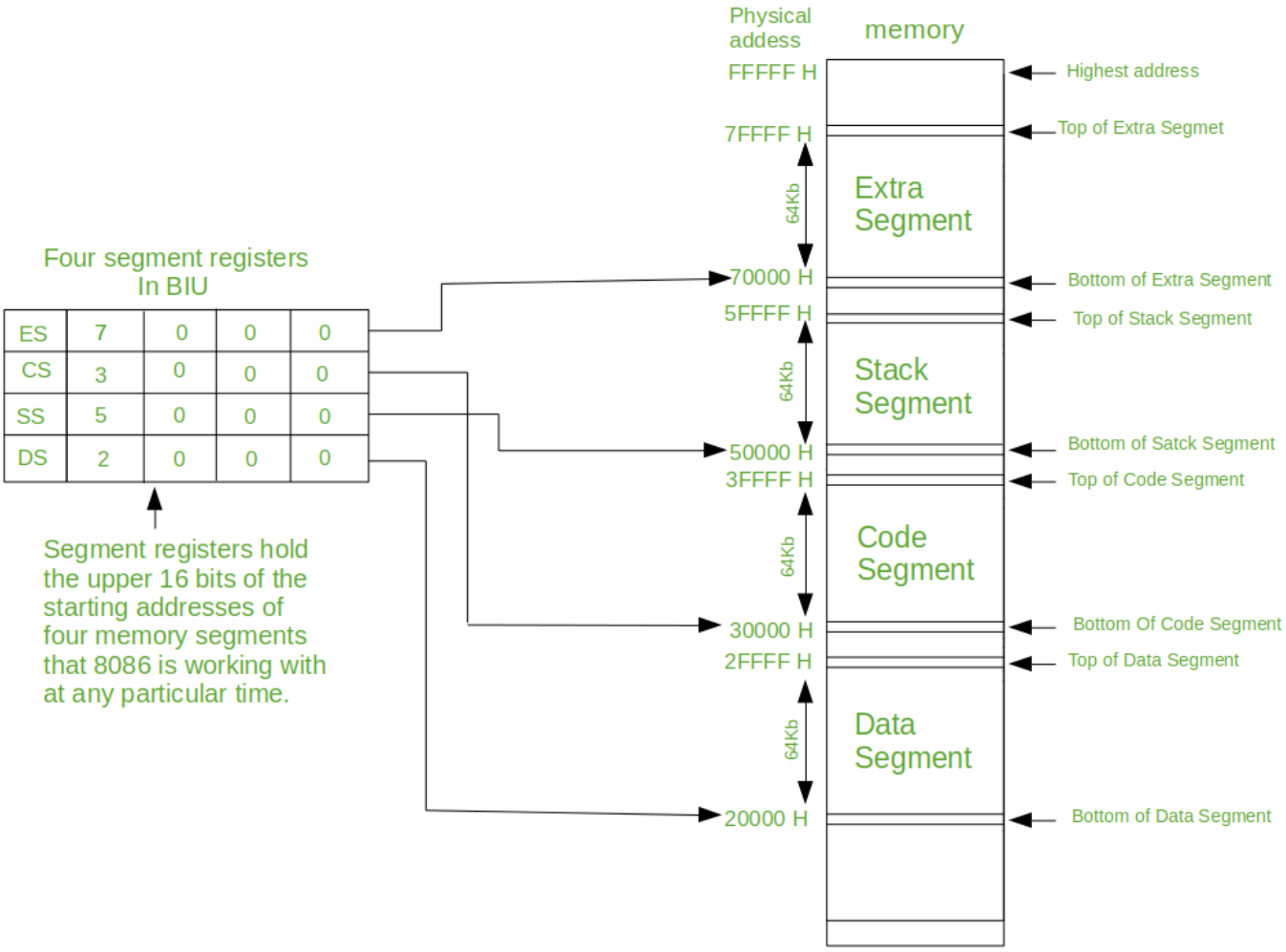
Code segment register (CS): is used for addressing memory location in the code segment of the memory, where the executable program is stored.

Data segment register (DS): points to the data segment of the memory where the data is stored.

Extra Segment Register (ES): also refers to a segment in the memory which is another data segment in the memory.

Stack Segment Register (SS): is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.

The number of address lines in 8086 is 20, 8086 BIU will send 20bit address, so as to access one of the 1MB memory locations. The four segment registers actually contain the upper 16 bits of the starting addresses of the four memory segments of 64 KB each with which the 8086 is working at that instant of time. A segment is a logical unit of memory that may be up to 64 kilobytes long. Each segment is made up of contiguous memory locations. It is an independent, separately addressable unit. Starting address will always be changing. It will not be fixed. Note that the 8086 does not work the whole 1MB memory at any given time. However, it works only with four 64KB segments within the whole 1MB memory. Below is the one way of positioning four 64 kilobyte segments within the 1M byte memory space of an 8086.



Types of Segmentation –

Overlapping Segment – A segment starts at a particular address and its maximum size can go up to 64kilobytes. But if another segment starts along with this 64kilobytes location of the first segment, then the two are said to be Overlapping Segment.

Non-Overlapped Segment – A segment starts at a particular address and its maximum size can go up to 64kilobytes. But if another segment starts before this 64kilobytes location of the first segment, then the two segments are said to be Non-Overlapped Segment.

Rules of Segmentation process follow as the starting address of a segment should be such that it can be evenly divided by 16. Minimum size of a segment can be 16 bytes & the maximum can be 64 kB.

3. 8086 Microprocessor Pin Configuration

The microprocessor 8086 is a 16-bit CPU available in different clock rates and packaged in a 40 pin plastic package. The 8086 operates in single processor or multiprocessor configuration to achieve high performance. The pins serve a particular function in minimum mode (single processor mode) and other function in maximum mode configuration (multiprocessor mode). The 8086 signals can be categorized in three groups. The first are the signal having common functions in minimum as well as maximum mode. The second are the signals which have special functions for minimum mode and third are the signals having special functions for maximum mode.

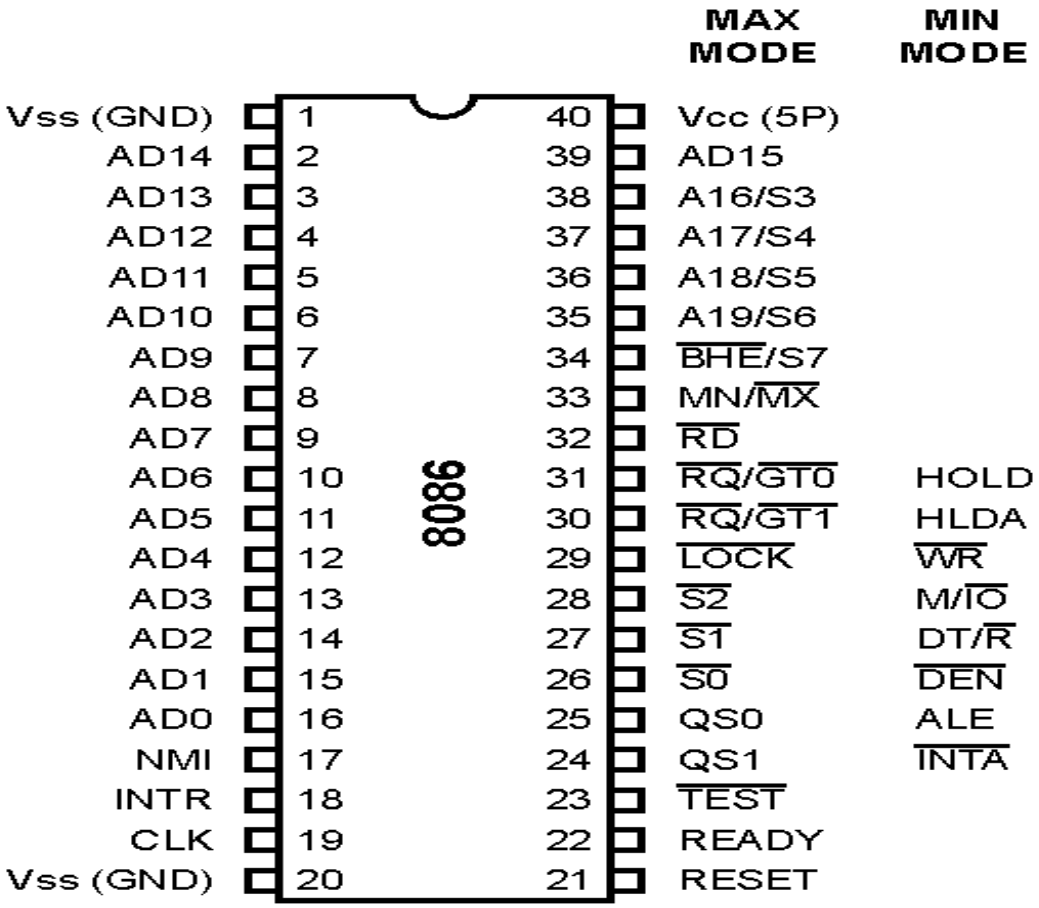


Figure 2 8086 Pin Diagram

The following signal descriptions are common for both modes.

- AD15-AD0: These are the time multiplexed memory I/O address and data lines. Address remains on the lines during T1 state, while the data is available on the data bus during T2, T3, Tw and T4. These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.
- A19/S6,A18/S5,A17/S4,A16/S3: These are the time multiplexed address and status lines. During T1 these are the most significant address lines for memory operations. During I/O operations, these lines are low. During memory or I/O operations, status information is available on those lines for T2,T3,Tw and T4. The status of the interrupt enable flag bit is updated at the beginning of each clock cycle. The S4 and S3 combine indicate which segment registers is presently being used for memory accesses. These lines float to tri-state off during the local bus hold acknowledge. The status line S6 is always low.
- The address bits are separated from the status bit using latches controlled by the ALE signal.

S4	S3	Indication/Registers
0	0	Alternate Data / ES
0	1	Stack / SS
1	0	Code / CS or none
1	1	Data / DS

- BHE /S7: The bus high enable is used to indicate the transfer of data over the higher order (D15-D8) data bus as shown in table. It goes low for the data transfer over D15- D8 and is used to derive chip selects of odd address memory bank or peripherals. BHE is low during T1 for read, write and interrupt acknowledge cycles, whenever a byte is to be transferred on higher byte of data bus. The status information is available during T2, T3 and T4. The signal is active low and tristated during hold. It is low during T1 for the first pulse of the interrupt acknowledges cycle.

BHE	A0	Indication
0	0	Word
0	1	Upper byte from odd address
1	0	Lower byte from even address
1	1	None

- RD Read: This signal on low indicates the peripheral that the processor is performing memory or I/O read operation. RD is active low and shows the state for T2, T3, Tw of any read cycle. The signal remains tristated during the hold acknowledge.
- READY: This is the acknowledgement from the slow device or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. The signal is active high.
- INTR-Interrupt Request: This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. This can be internally masked by resulting the interrupt enable flag. This signal is active high and internally synchronized.
- TEST: This input is examined by a 'WAIT' instruction. If the TEST pin goes low, execution will continue, else the processor remains in an idle state. The input is synchronized internally during each clock cycle on leading edge of clock.
- CLK- Clock Input: The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle.
- MN/MX : The logic level at this pin decides whether the processor is to operate in either minimum or maximum mode.

The following pin functions are for the minimum mode operation of 8086.

- M/ IO – Memory/IO: This is a status line logically equivalent to S2 in maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active high in the previous T4 and remains active till final T4 of the current cycle. It is tristated during local bus "hold acknowledge".
- INTA Interrupt Acknowledge: This signal is used as a read strobe for interrupt acknowledge cycles. i.e. when it goes low, the processor has accepted the interrupt.
- ALE – Address Latch Enable: This output signal indicates the availability of the valid address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tristated.

- DT/R – Data Transmit/Receive: This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low.
- DEN – Data Enable: This signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers (bidirectional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle of T2 until the middle of T4. HOLD, HLDA-Acknowledge: When the HOLD line goes high; it indicates to the processor that another master is requesting the bus access.
- The processor, after receiving the HOLD request, issues the hold acknowledge signal on HLDA pin, in the middle of the next clock cycle after completing the current bus cycle. At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal. HOLD is an asynchronous input, and is should be externally synchronized. If the DMA request is made while the CPU is performing a memory or I/O cycle.

The following pin functions are applicable for maximum mode operation of 8086.

- S2, S1, S0 – Status Lines: These are the status lines which reflect the type of operation, being carried out by the processor. These become activity during T4 of the previous cycle and active during T1 and T2 of the current bus cycles.

- LOCK: This output pin indicates that other system bus master will be prevented from gaining the system bus, while the LOCK signal is low. The LOCK signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus.

S2	S1	S0	Machine Cycle
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Instruction Fetch
1	0	1	Memory Red
1	1	0	Memory Write
1	1	1	Inactive-Passive

- QS1, QS0 – Queue Status: These lines give information about the status of the code-prefetch queue. These are active during the CLK cycle after while the queue operation is performed.
  This modification in a simple fetch and execute architecture of a conventional microprocessor offers an added advantage of pipelined processing of the instructions. The 8086 architecture has 6-byte instruction prefetch queue. Thus even the largest (6- bytes) instruction can be perfected from the memory and stored in the prefetch. This results in a faster execution of the instructions. In 8085 an instruction is fetched, decoded and executed and only after the execution of this instruction, the next one is fetched. By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This is known as instruction pipelining. At the starting the CS:IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS:IP address is odd or two bytes at a time, if the CS:IP address is even. The first byte is a complete opcode in case of some instruction (one byte opcode instruction) and is a part of opcode, in case of some instructions (two byte opcode instructions), the remaining part of code lie in second byte. The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions. The next byte after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program. The fetch operation of the next instruction is overlapped with the execution of the current instruction. As in the architecture, there are two separate units, namely Execution unit and Bus interface unit. While the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes of the next instruction from memory, depending upon the queue status.

QS0	QS1	Status
0	0	No operation (queue is idle)
0	1	First byte of an op-code
1	0	Queue is empty
1	1	Subsequent byte of an op-code

- RQ / GT0, RQ / GT1 – Request/Grant: These pins are used by the other local bus master in maximum mode, to force the processor to release the local bus at the end of the processor current bus cycle. Each of the pin is bidirectional with RQ/GT0 having higher priority than RQ/GT1. RQ/GT pins have internal pull-up resistors and may be left unconnected.

4. 8080 Addressing Modes

Register Addressing

The instruction will specify the name of the register which holds the data to be operated by the instruction.
Example: **MOV CL, DH**
The content of 8-bit register DH is moved to another 8-bit register CL
(CL) ← (DH)

Immediate Addressing

In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example: **MOV DL, 08H**

The 8-bit data (08_H) given in the instruction is moved to DL

(DL) ← 08_H

MOV AX, 0A9FH

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX register

(AX) ← $0A9F_H$

Direct Addressing

Here, the effective address of the memory location at which the data operand is stored is given in the instruction. The effective address is just a 16-bit number written directly in the instruction.

Example: **MOV BX, [1354H], MOV BL, [0400H]**

The square bracket around the 1354H denotes the contents of the memory location. When executed, this instruction will copy the contents of the memory location into BX register.

This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction**.**

Register Indirect Addressing

In Register indirect addressing, name of the register which holds the effective address (EA) will be specified in the instruction. Registers used to hold EA are any of the following registers: BX, BP, DI and SI. Content of the DS register is used for base address calculation.

Example: **MOV CX, [BX]**

Operations:

EA = (BX)

BA = (DS) x 16_{10}

MA = BA + EA

Based Addressing

In Based Addressing, BX or BP is used to hold the base value for effective address and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction. In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

When BX holds the base value of EA, 20-bit physical address is calculated from BX and DS. When BP holds the base value of EA, BP and SS is used.

Example: **MOV AX, [BX + 08H]**

Operations:

$0008_H \leftarrow 08_H$ (Sign extended)

$EA = (BX) + 0008_H$

$BA = (DS) \times 16_{10}$

$MA = BA + EA$

Indexed Addressing

SI or DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

Displacement is added to the index value in SI or DI register to obtain the EA. In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

Example: **MOV CX, [SI + FA2H]**

Operations:

$FFA2_H \leftarrow FA2_H$ (Sign extended)

$EA = (SI) + FFA2_H$

$BA = (DS) \times 16_{10}$

$MA = BA + EA$

Based Index Addressing

In Based Index Addressing, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI) and a displacement.

Example: **MOV DX, [BX + SI + 0AH]**

Operations:

$000A_H \leftarrow 0A_H$ (Sign extended)

$EA = (BX) + (SI) + 000A_H$

$BA = (DS) \times 16_{10}$

$MA = BA + EA$

String Addressing

It is used in string operations to operate on string data. The effective address (EA) of source data is stored in SI register and the EA of destination is stored in DI register.

Segment register for calculating base address of source data is DS and that of the destination data is ES

Example: MOVS BYTE

Operations:

Calculation of source memory location:

$EA = (SI)$ $BA = (DS) \times 16_{10}$ $MA = BA + EA$

Calculation of destination memory location:

$EA_E = (DI)$ $BA_E = (ES) \times 16_{10}$ $MA_E = BA_E + EA_E$

Direct I/O & Indirect I/O port Addressing

These addressing modes are used to access data from standard I/O mapped devices or ports.

In direct port addressing mode, an 8-bit port address is directly specified in the instruction.

Example: **IN AL, [09H]**

Operations: $PORT_{addr} = 09_H$

(AL) ← (PORT)

Content of port with address 09_H is moved to AL register

In indirect port addressing mode, the instruction will specify the name of the register which holds the port address. In 8086, the 16-bit port address is stored in the DX register.

Example: **OUT [DX], AX**

Operations: $PORT_{addr} = (DX)$

(PORT) ← (AX)

Content of AX is moved to port whose address is specified by DX register.

Relative Addressing

In this addressing mode, the effective address of a program instruction is specified relative to Instruction Pointer (IP) by an 8-bit signed displacement.

Example: **JZ 0AH**

Operations:

$000A_H \leftarrow 0A_H$ (sign extend)

$EA = (IP) + 000A_H$

$BA = (CS) \times 16_{10}$

$MA = BA + EA$

Implied Addressing

Instructions using this mode have no operands. The instruction itself will specify the data to be operated by the instruction.

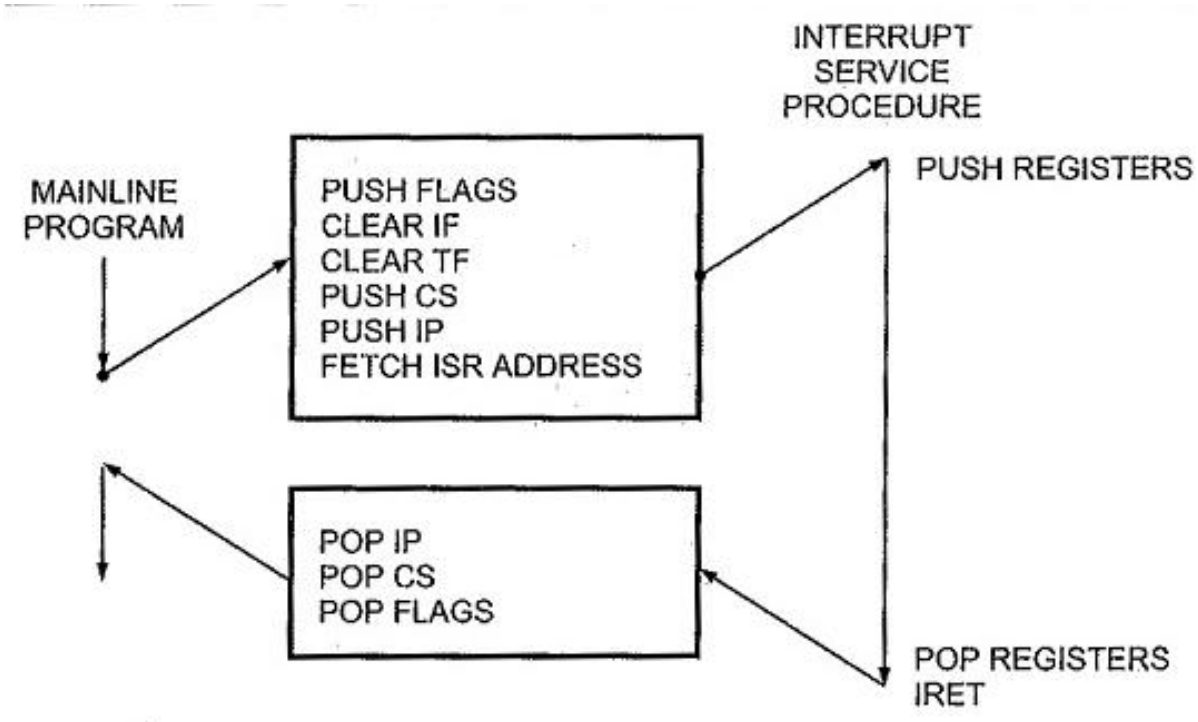
Example: **CLC**

This clears the carry flag to zero.

INTERRUPT

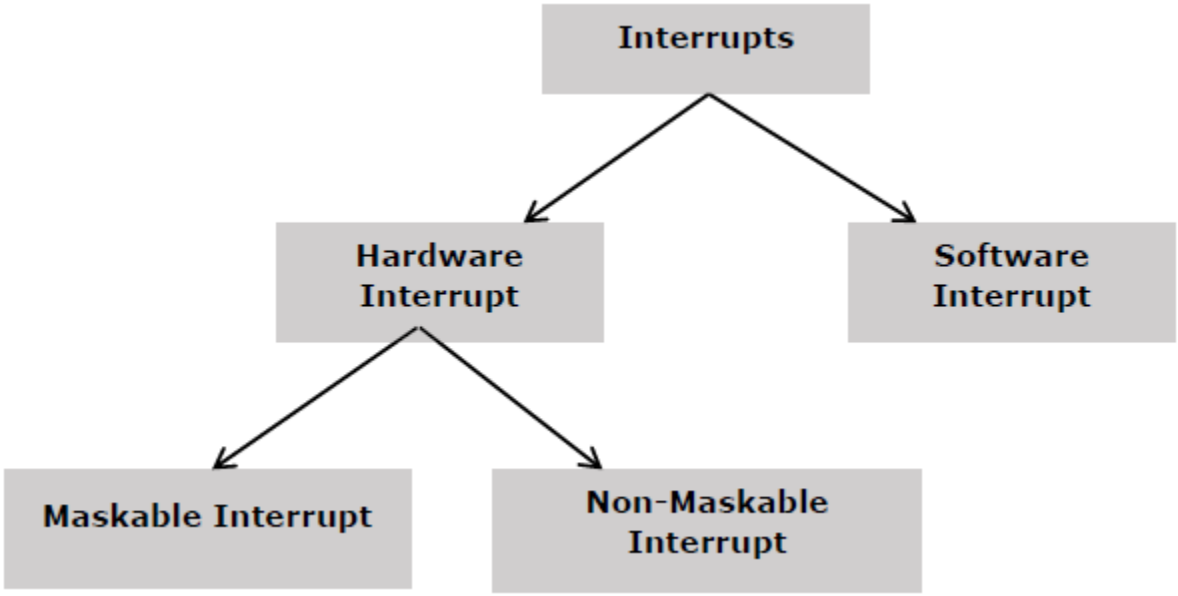
Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an **ISR** (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.

An 8086 is interrupted by some condition produced in the 8086 by the execution of an instruction. For example divide by zero: Program execution will automatically be interrupted if you attempt to divide an operand by zero. At the end of each instruction cycle 8086 interrupts checks to see if there is any interrupt request. If so, 8086 responds to the interrupt by performing series of actions



- It decrements stack pointer by 2 and pushes the flag register on the stack.
- It disables the INTR interrupt input by clearing the interrupt flag in the flag
- It resets the trap flag in the flag register.
- It decrements stack pointer by 2 and pushes the current code segment register contents on the stack.
- It decrements stack pointer by 2 and pushes the current instruction pointer contents on the stack.
- It does an indirect far jump at the start of the procedure by loading the CS and IP values for the start of the interrupt service routine (ISR).

The following image shows the types of interrupts we have in an 8086 microprocessor −



Hardware Interrupts

Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor.

The 8086 has two hardware interrupt pins, i.e. NMI and INTR. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority. One more interrupt pin associated is INTA called interrupt acknowledge.

NMI

It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR) and it is of type 2 interrupt.

When this interrupt is activated, these actions take place −

- Completes the current instruction that is in progress.
- Pushes the Flag register values on to the stack.

- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
- IP is loaded from the contents of the word location 00008H.
- CS is loaded from the contents of the next word location 0000AH.
- Interrupt flag and trap flag are reset to 0.

INTR

The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction. It should not be enabled using clear interrupt Flag instruction.

The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends '0' on INTA pin twice. The first '0' means INTA informs the external device to get ready and during the second '0' the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.

These actions are taken by the microprocessor −

- First completes the current instruction.
- Activates INTA output and receives the interrupt type, say X.
- Flag register value; CS value of the return address and IP value of the return address are pushed on to the stack.
- IP value is loaded from the contents of word location X × 4
- CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0

Software Interrupts

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes −

INT- Interrupt instruction with type number

It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 'type number' × 4
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

The starting address for type0 interrupt is 000000H, for type1 interrupt is 00004H similarly for type2 is 00008H and ……so on. The first five pointers are dedicated interrupt pointers. i.e. −

- **TYPE 0** interrupt represents division by zero situation.
- **TYPE 1** interrupt represents single-step execution during the debugging of a program.
- **TYPE 2** interrupt represents non-maskable NMI interrupt.
- **TYPE 3** interrupt represents break-point interrupt.
- **TYPE 4** interrupt represents overflow interrupt.

The interrupts from Type 5 to Type 31 are reserved for other advanced microprocessors, and interrupts from 32 to Type 255 are available for hardware and software interrupts.

INT 3-Break Point Interrupt Instruction

It is a 1-byte instruction having op-code is CCH. These instructions are inserted into the program so that when the processor reaches there, then it stops the normal execution of program and follows the break-point procedure.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 3×4 = 0000CH
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

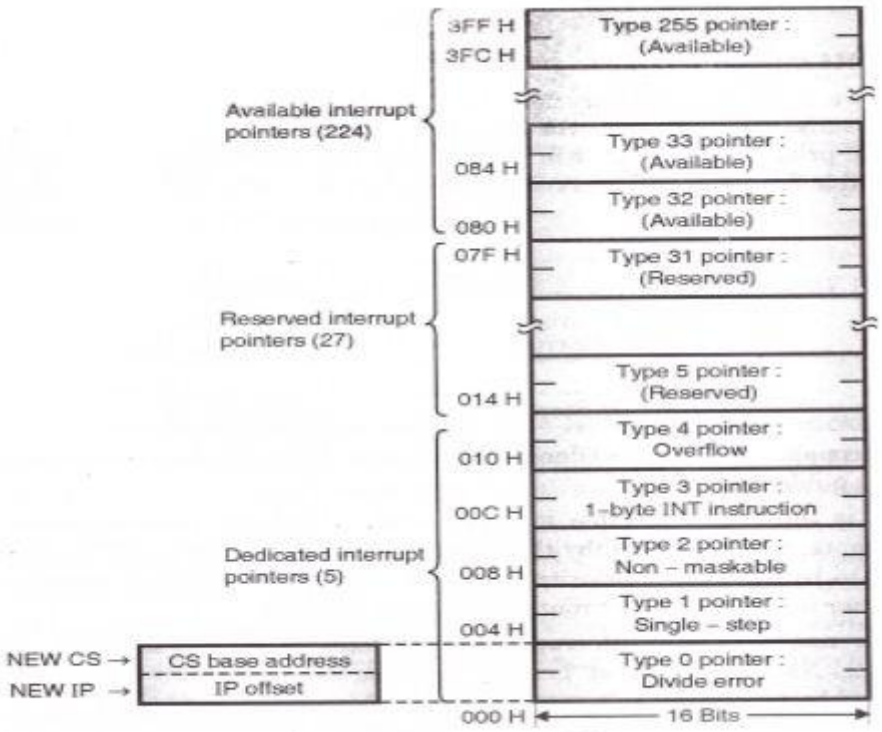
INT 4 - Interrupt on overflow instruction

It is a 1-byte instruction and their mnemonic INT 4. The op-code for this instruction is CEH. As the name suggests it is a conditional interrupt instruction, i.e. it is active only when the overflow flag is set to 1 and branches to the interrupt handler whose interrupt type number is 4. If the overflow flag is reset then, the execution continues to the next instruction.

Its execution includes the following steps −

- Flag register values are pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of word location 4×4 = 00010H
- CS is loaded from the contents of the next word location.
- Interrupt flag and Trap flag are reset to 0

Interrupt Vector Table (IVT):



Interrupt vector table

The interrupt vector (or interrupt pointer) table is the link between an interrupt type code and the procedure that has been designated to service interrupts associated with that code. 8086 supports total 256 types i.e. 00H to FFH.

- For each type it has to reserve four bytes i.e. double word. This double word pointer contains the address of the procedure that is to service interrupts of that type.
- The higher addressed word of the pointer contains the base address of the segment containing the procedure. This base address of the segment is normally referred as NEW CS.
- The lower addressed word contains the procedure's offset from the beginning of the segment. This offset is normally referred as NEW IP.
- Thus NEW CS: NEW IP provides NEW physical address from where user ISR routine will start.
- As for each type, four bytes (2 for NEW CS and 2 for NEW IP) are required; therefore interrupt pointer table occupies up to the first 1k bytes (i.e. 256 x 4 = 1024 bytes) of low memory.
- The total interrupt vector table is divided into three groups namely,
    - Dedicated interrupts (INT 0…..INT 4)
    - Reserved interrupts (INT 5…..INT 31)
    - Available interrupts (INT 32…..INT 225)

80386

80386 Microprocessor is a 32-bit processor that holds the ability to carry out 32-bit operation in one cycle. It has data and address bus of 32-bit each. Thus has the ability to address 4 GB (or 232) of physical memory.

Multitasking and protection capability are the two key characteristics of 80386 microprocessor. 80386 have an internal dedicated hardware that permits multitasking.

We know 8086 is a 16-bit microprocessor and 80286 was an advancement of 8086 with some additional characteristics. But with the advent of technology Intel introduced a 32-bit microprocessor whose processing speed was twice as that of 80286 microprocessor. This was 80386 microprocessor that was designed by Intel in October 1985 and was an upgraded version of 80286 microprocessor

Features of 80386

- As it is a 32-bit microprocessor. Thus has 32-bit ALU.
- 80386 have data bus of 32-bit.
- It holds address bus of 32 bit.
- It supports physical memory addressability of 4 GB and virtual memory addressability of 64 TB.
- 80386 support variety of operating clock frequency, which are 16 MHz, 20 MHz, 25 MHz and 33 MHz.
- It offers 3 stage pipeline: fetch, decode and execute. As it supports simultaneous fetching, decoding and execution inside the system.

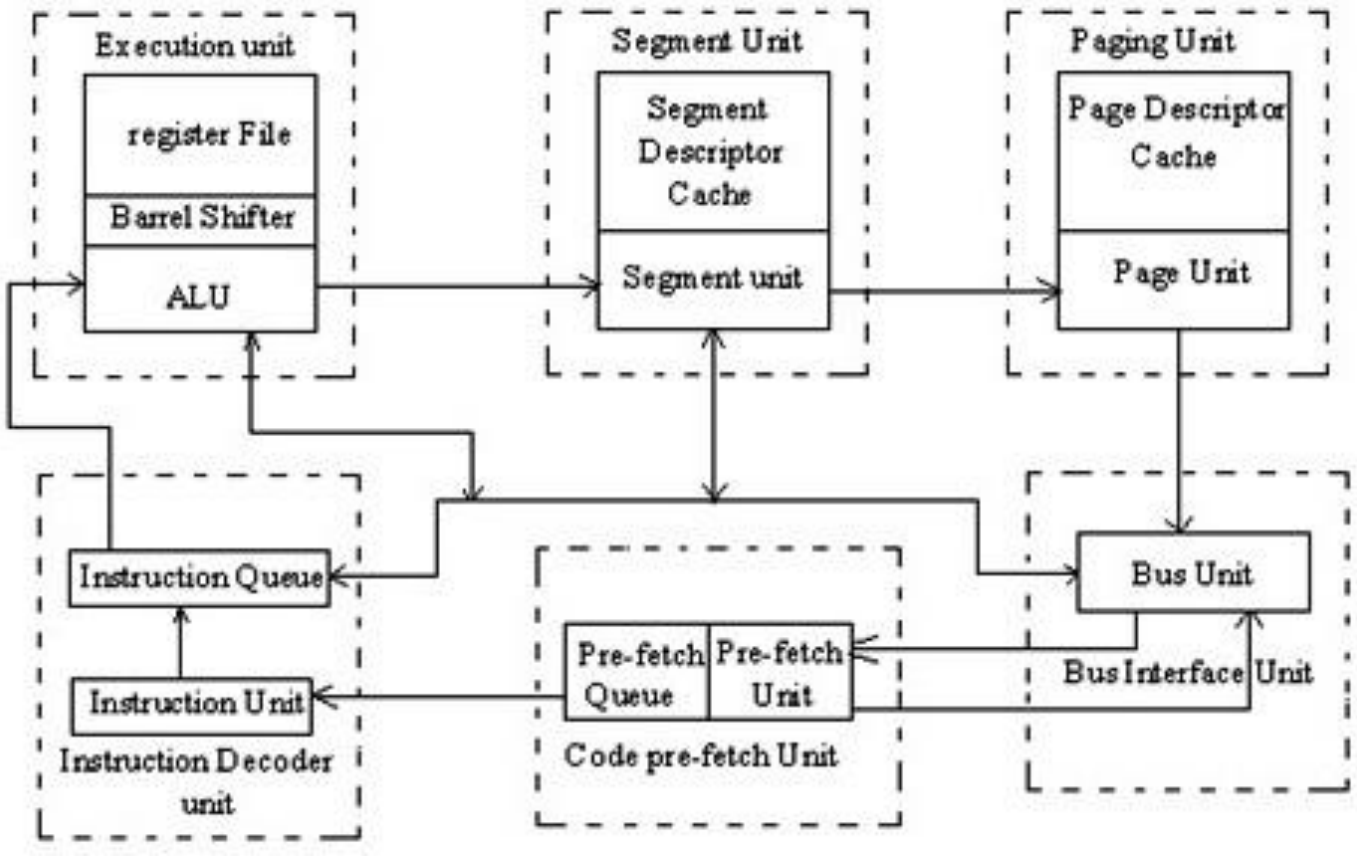
Operating modes of 80386

80386 support 3 operating modes: real, protected and virtual real mode. Of the two modes of 80286 microprocessor, initially the 80286 was booted in real mode. However, to have better operating performance, separate software command is used to switch from the real mode to the protected mode.

But it requires the resetting of microprocessor in order to switch to real mode from protected mode. This drawback was eliminated in 80386 that allow the switching between the modes using software commands.

In the **protected mode**, 80386 microprocessor operates in similar way like 80286, but offers higher memory addressing ability. In **virtual mode**, the overall memory of 80386 can be divided into various virtual machines. And all of them act as a separate computer with 8086 microprocessor. This mode is also called virtual 8086 mode or V86 mode. The other one is the **virtual real mode**, this mode allows the system to execute multiple programs in the protected memory. And in case a program at a particular memory gets crashed then it will not cause any adverse effect on the other part of the memory.

Architecture of 80386 Microprocessor



- The internal architecture of the 80386 includes six functional units that operate in parallel. The parallel operation is called as pipeline processing.
- Fetching, decoding execution, memory management, and bus access for several instructions are performed simultaneously.
- The six functional units of the 80386 are
  1) Bus Interface Unit
  2) Code Pre-fetch Unit
  3) Instruction Decoder Unit
  4) Execution Unit
  5) Segmentation Unit
  6) Paging Unit

- The Bus Interface Unit connects the 80386 with memory and I/O. Based on internal requests for fetching instructions and transferring data from the code pre-fetch unit, the 80386 generates the address, data and control signals for the current bus cycles.
- The code pre-fetch unit pre-fetches instructions when the bus interface unit is not executing the bus cycles. It then stores them in a 16-byte instruction queue for decoding by the instruction decode unit.
- The instruction decode unit translates instructions from the pre-fetch queue into micro-codes. The decoded instructions are then stored in an instruction queue (FIFO) for processing by the execution unit.
- The execution unit processes the instructions from the instruction queue. It contains a control unit, a data unit and a protection test unit.
- The control unit contains microcode and parallel hardware for fast multiply, divide and effective address calculation. The unit includes a 32-bit ALU, 8 general purpose registers and a 64-bit barrel shifter for performing multiple bit shifts in one clock. The data unit carries out data operations requested by the control unit.
- The protection test unit checks for segmentation violations under the control of microcode.
- The segmentation unit calculates and translates the logical address into linear addresses at the request of the execution unit.
- The translated linear address is sent to the paging unit. Upon enabling the paging mechanism, the 80386 translates these linear addresses into physical addresses.
- If paging is not enabled, the physical address is identical to the linear address and no translation is necessary.

5. Processor Organization

To understand the organization of processor it is necessary to understand the functions of the processor. The processor has to do following main functions.

Fetch instruction: The processor reads an instruction from memory (register, cache, main memory).

Interpret instruction: The instruction is decoded to determine what action is required.

Fetch data: The execution of an instruction may require reading data from memory or an I/O module.

Process data: The execution of an instruction may require performing some arithmetic or logical operation on data.

Write data: The results of an execution may require writing data to memory or I/O module.

To do these things, it should be clear that the processor needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the processor needs a small internal memory.

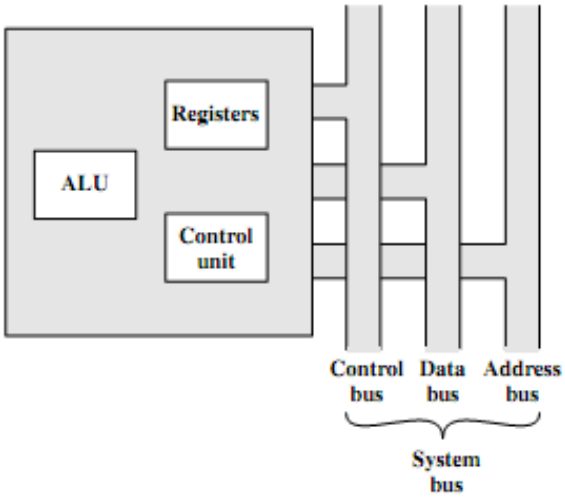


Figure 3 General Organization of Processor

Figure 7 is a simplified view of a processor, indicating its connection to the rest of the system via the system bus. The major components of the processor are an arithmetic and logic unit (ALU) and a control unit (CU).The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. The figure shows a minimal internal memory, consisting of a set of storage locations, called registers.

Figure 8 is a slightly more detailed view of the processor. The data transfer and logic control paths are indicated, including an element labeled internal processor bus. This element is needed to transfer data between the various registers and the ALU because the ALU in fact operates only on data in the internal processor memory. The figure also shows typical basic elements of the ALU. Note the similarity between the internal structure of the computer as a whole and the internal structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers) connected by data paths.

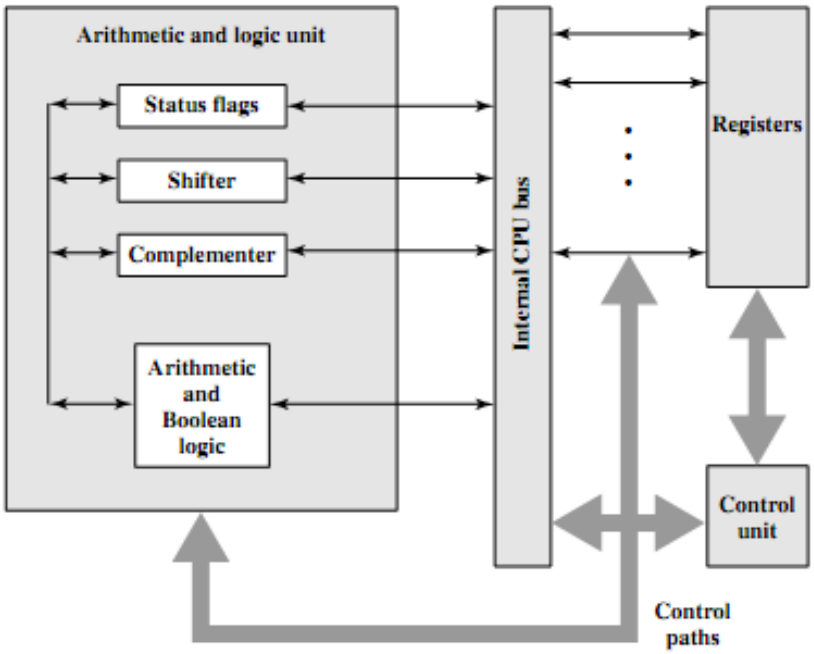


Figure 4 Detail Organization of Processor

6. Machine Instruction Characteristics

The operation of the processor is determined by the instructions it executes, referredto as machine instructions.The collection of different instructions that the processor can execute is referred to as the processor's instruction set.

Each instruction must contain the information required by the CPU for execution. Figure 1 shows the instruction cycle state diagram. The elements of an instruction are as follows:

Operation Code: Specifies the operation to be performed (e.g., add, move etc.). The operation is specified by a binary code, known as the operation code or opcode.

Source operand reference: The operation may involve one or more source operands; that is, operands that are inputs for the operation.

Result operand reference: The operation may produce a result.

Next instruction reference: This tells the CPU where to fetch the next instruction after the execution of this instruction is complete.

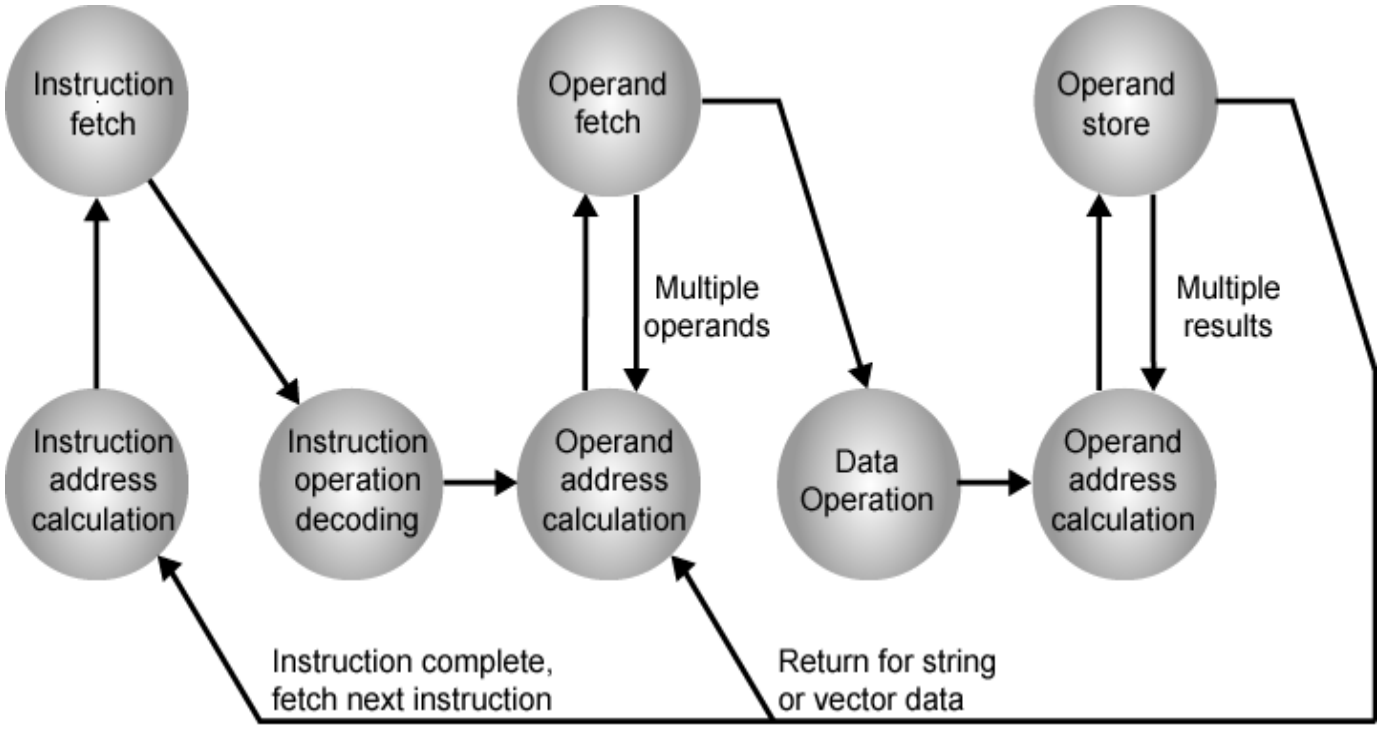


Figure 5 Instruction Cycle State Diagram

The address of the next instruction to be fetched could be either a real addressor a virtual address, depending on the architecture. Generally, the distinction istransparent to the instruction set architecture. In most cases, the next instruction tobe fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction.When an explicit reference is needed, thenthe main memory or virtual memory address must be supplied.

Instruction Representation

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. The instruction format is highly machine specific and it mainly depends on the machine architecture. A simple example of an instruction format is shown in the Figure 2. It is assume that it is a 16-bit CPU.4 bits are used to provide the operation code. So, we may have to

16 (2^4 = 16) different set of instructions. With each instruction, there are two operands. To specify each operand, 6 bits are used. It is possible to provide 64 (2^6 = 64) different operands for each operand reference.It is difficult to deal with binary representation of machine instructions. Thus, it has become common practice to use a symbolic representation of machine instructions.Opcodes are represented by abbreviations, called mnemonics, which indicate the operations.Common examples include: ADD (Addition), SUB (Subtraction), MULT (Multiply), DIV (DIVISION).

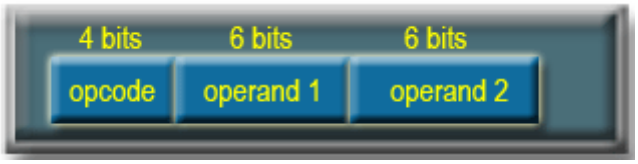


Figure 6 Instruction Representation

Instruction Types

The instruction set of a CPU can be categorized as follows:

Data Processing (Arithmetic and Logic instructions): Arithmetic instructions provide computational capabilities for processing numeric data. Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers. Logic instructions thus provide capabilities for processing any other type of data. There operations are performed primarily on data in CPU registers.

Data Storage: Memory instructions are used for moving data between memory and CPU registers.

Data Movement:I/O instructions are needed to transfer program and data into memory from storage device or input device and the results of computation back to the user.

Control (Test and branch instructions): Test instructions are used to test the value of a data word or the status of a computation. Branch instructions are then used to branch to a different set of instructions depending on the decision made.

Number of Addresses

Most of the arithmetic and logic operations are either unary (one operand) or binary (two operands). Thus we need a maximum of two addresses to reference operands. The result of an operation must be stored, suggesting a third address. Finally after completion of an instruction, the next instruction must be fetched, and its address is needed.

This reasoning suggests that an instruction may require to contain four address references: two operands, one result, and the address of the next instruction. In practice, four address instructions are rare. Most instructions have one, two or three operands addresses, with the address of the next instruction being implicit (obtained from the program counter).

Instruction Set Design

One of the most interesting, and most analyzed, aspects of computer design is instruction set design. The instruction set defines the functions performed by the CPU. The instruction set is the programmer's means of controlling the CPU. Thus programmer requirements must be considered in designing the instruction set.

Most important and fundamental design issues:

Operation repertoire: How many and which operations to provide, and how complex operations should be.

Data Types: The various type of data upon which operations are performed.
Instruction format: Instruction length (in bits), number of addresses, size of various fields and so on.
Registers: Number of CPU registers that can be referenced by instructions and their use.
Addressing: The mode or modes by which the address of an operand is specified.

Types of Operands

Machine instructions operate on data. Data can be categorized as follows:

Addresses: It basically indicates the address of a memory location. Addresses are nothing but the unsigned integer, but treated in a special way to indicate the address of a memory location. Address arithmetic is somewhat different from normal arithmetic and it is related to machine architecture.

Numbers: All machine languages include numeric data types. Numeric data are classified into two broad categories: integer or fixed point and floating point.

Characters: A common form of data is text or character strings. Since computer works with bits, so characters are represented by a sequence of bits. The most commonly used coding scheme is ASCII (American Standard Code for Information Interchange) code.

Logical Data: Normally each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometime useful to consider an *n*-bit unit as consisting of *n* 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be logical data. Generally 1 is treated as true and 0 is treated as false.

Types of Operations

The number of different opcodes and their types varies widely from machine to machine. However, some general types of operations are found in most of the machine architecture. Those operations can be categorized as follows:

- Data Transfer
- Arithmetic
- Logical
- Conversion
- Input Output [I/O]
- System Control
- Transfer Control

Data Transfer: The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things. First, the locationof the source and destination operands must be specified. Each location could bememory, a register, or the top of the stack. Second, the length of data to be transferred must be indicated. Third, as with all instructions with operands, the modeof addressing for each operand must be specified.

Examples: Move, Store, Load, Push, Pop

Arithmetic: The basic arithmetic operations of add, subtract, multiply,and divide are supported by processors. These are invariably provided for signed integer (fixed-point) numbers.Often they are also provided for floating-point and packed decimal numbers.

Examples: Add, Subtract, Increment etc.

Logical: Most machines also provide a variety of operations for manipulating individual bitsof a word or other addressable units, often referred to as “bit twiddling.” They arebased upon Boolean operations. Examples: AND, OR, NOT, XOR etc.

Conversion: Conversion instructions are those that change the format or operate on the formatof data. An example is converting from decimal to binary. Examples: Translate, Convert.

Input Output: Input and output instructions are necessary to provide input output functionality to the machine.Example: Read, Write etc.

System Control: System control instructions are those that can be executed only while the processoris in a certain privileged state or is executing a program in a special privileged areaof memory. Typically, these instructions are reserved for the use of the operatingsystem.

Transfer Control: A significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the processor is to update the program counter to contain the address ofsome instruction in memory. Examples: Jump, Return etc.

Addressing Modes

The address field or fields in a typical instruction format are relatively small. Wewould like to be able to reference a large range of locations in main memory or, forsome systems, virtual memory. To achieve this objective, a variety of addressingtechniques has been employed.The most common addressing techniques are:

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement
- Stack

Immediate Addressing: The simplest form of addressing is immediate addressing, in which the operandvalue is present in the instructionOperand = A. This mode can be used to define and use constants or set initial values ofvariables.

The advantage of immediate addressing is that no memory reference otherthan the instruction fetch is required to obtain the operand, thus saving one memoryor cache cycle in the instruction cycle.The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, issmall compared with the word length.

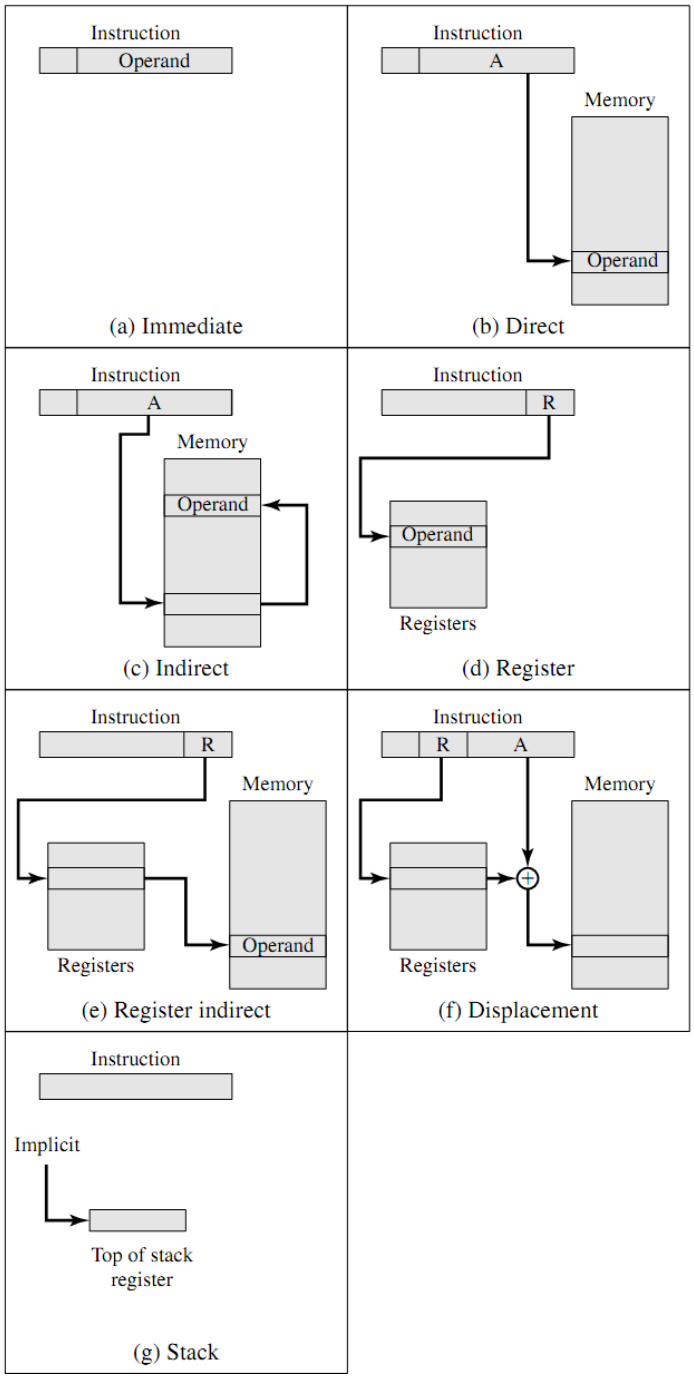


Figure 7 Addressing Modes

Direct Addressing: A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand EA = A. The technique was common in earlier generations of computers but is not commonon contemporary architectures. It requires only one memory reference and nospecial calculation. The obvious limitation is that it provides only a limited addressspace.

Indirect Addressing: With direct addressing, the length of the address field is usually less than the wordlength, thus limiting the address range. One solution is to have the address fieldrefer to the address of a word in memory, which in turn contains a full-lengthaddress of the operand.This is known as indirect addressing: EA = (A). Its disadvantage isthat three or more memory references could be required to fetch an operand.

Register Addressing: Register addressing is similar to direct addressing. The only difference is that theaddress field refers to a register rather than a main memory address: EA = R. The advantages of register addressing are that (1) only a small address field isneeded in the instruction, and (2) no time-consuming memory references are required.The disadvantage of register addressing is that the address space is very limited.

Register Indirect Addressing: Register indirect addressingis analogous to indirect addressing. In both cases, the only difference is whether theaddress field refers to a memory location or a register.Thus, for register indirect address, EA = (R). The advantages and limitations of register indirect addressing are basically the sameas for indirect addressing. In both cases, the address space limitation (limited rangeof addresses) of the address field is overcome by having that field refer to a word length location containing an address. In addition, register indirect addressing usesone less memory reference than indirect addressing.

Displacement Addressing: A very powerful mode of addressing combines the capabilities of direct addressing andregister indirect addressing. It is known by a variety of names depending on the contextof its use, but the basic mechanism is the same. The displacementaddressing is referred as: EA = A + (R). Displacement addressing requires that the instruction have two address fields, atleast one of which is explicit.The value contained in one address field (value = A)is used directly. The other address field, or an implicit reference based on opcode,refers to a register whose contents are added to A to produce the effectiveaddress.

Stack Addressing: The stack is a reserved block of locations.Items are appended to the top of the stack so that, at any given time, the block ispartially filled.Associated with the stack is a pointer whose value is the address ofthe top of the stack. Alternatively, the top two elements of the stack may be inprocessor registers, in which case the stack pointer references the third element ofthe stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.The stack mode of addressing is a form of implied addressing. The machineinstructions need not include a memory reference but implicitly operate on the topof the stack.

Instruction Format

An instruction format defines the layout of the bits of an instruction, in terms of its constituent's parts. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes that is available for that machine. The format must, implicitly or explicitly, indicate the addressing mode of each operand. For most instruction sets, more than one instruction format is used. Four common instruction formats are shown in the Figure 4.

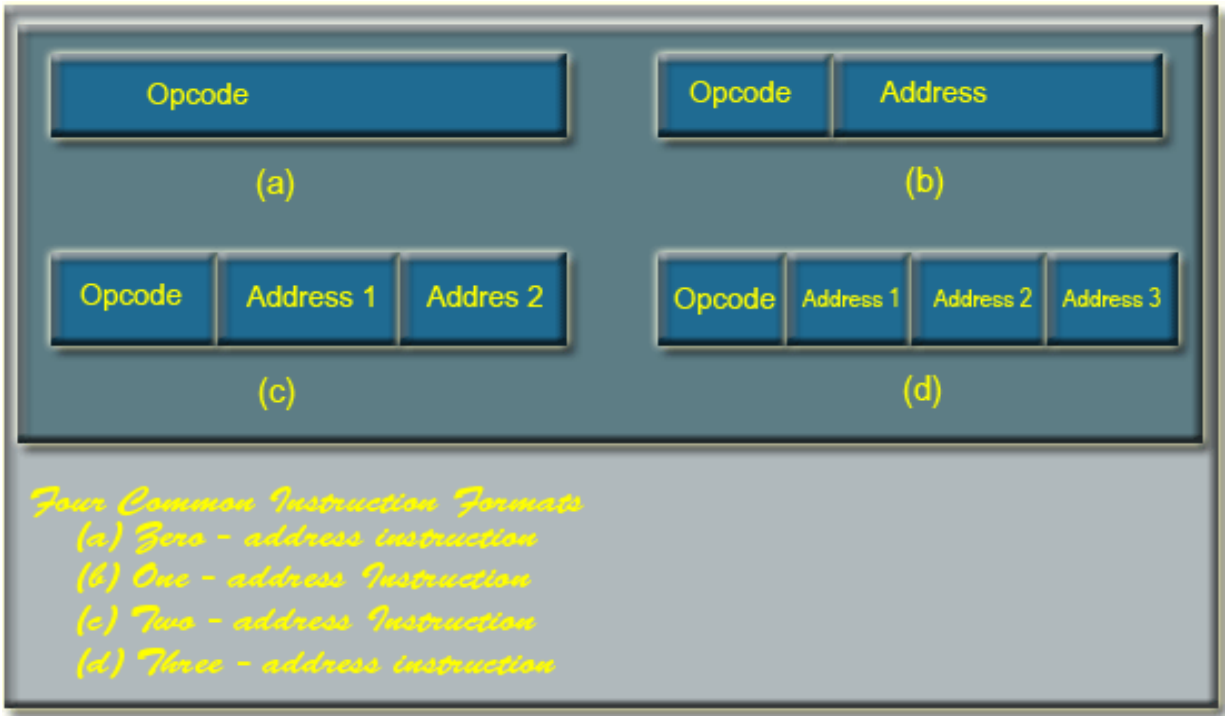


Figure 8 Four Instruction Formats

Instruction Length: On some machines, all instructions have the same length; on others there may be many different lengths. Instructions may be shorter than, the same length as, or more than the word length. Having all the instructions be the same length is simpler and make decoding easier but often wastes space, since all instructions then have to be as long as the longest one. Possible relationship between instruction length and word length is shown in the Figure 5.

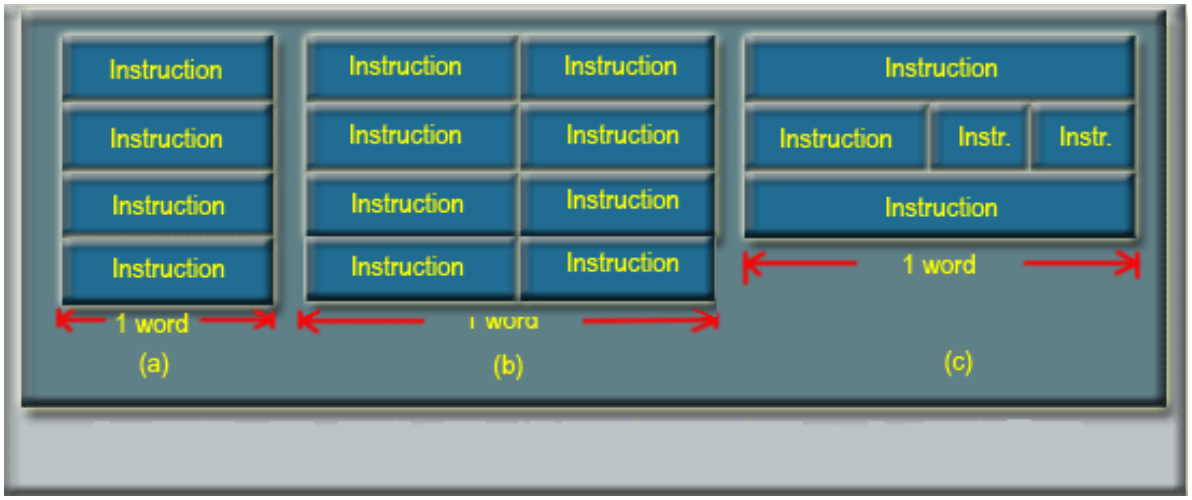


Figure 9 Relation between Instruction and Word Length

Generally there is a correlation between memory transfer length and the instruction length. Either the instruction length should be equal to the memory transfer length or one should be a multiple of the other. Also in most of the case there is a correlation between memory transfer length and word length of the machine.

Allocation of Bits: For a given instruction length, there is a clearly a trade-off between the number of opcodes and the power of the addressing capabilities. More opcodes obviously mean more bits in the opcode field. For

an instruction format of a given length, this reduces the number of bits available for addressing. The following interrelated factors go into determining the use of the addressing bits.

- **Number of Addressing modes:**Sometimes as addressing mode can be indicated implicitly. In other cases, the addressing mode must be explicit, and one or more bits will be needed.
- **Number of Operands:** Typical instructions on today's machines provide for two operands. Each operand address in the instruction might require its own mode indicator, or the use of a mode indicator could be limited to just one of the address field.
- **Register versus memory:** A machine must have registers so that data can be brought into the CPU for processing. With a single user-visible register (usually called the accumulator), one operand address is implicit and consumes no instruction bits. Even with multiple registers, only a few bits are needed to specify the register. The more that registers can be used for operand references, the fewer bits are needed.
- **Number of register sets:** A number of machines have one set of general purpose registers, with typically 8 or 16 registers in the set. These registers can be used to store data and can be used to store addresses for displacement addressing. The trend recently has been away from one bank of general purpose registers and toward a collection of two or more specialized sets (such as data and displacement).
- **Address range:** For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits. With displacement addressing, the range is opened up to the length of the address register.
- **Address granularity**: In a system with 16- or 32-bit words, an address canreference a word or a byte at the designer's choice. Byte addressing is convenient for character manipulation but requires, for a fixed size memory, more address bits.

Variable-Length Instructions: Instead of looking for fixed length instruction format, designer may choose to provide a variety of instructions formats of different lengths. This tactic makes it easy to provide a large repertoire of opcodes, with different opcode lengths. Addressing can be more flexible, with various combinations of register and memory references plus addressing modes. With variable length instructions, many variations can be provided efficiently and compactly. The principal price to pay for variable length instructions is an increase in the complexity of the CPU.

RISC and CISC Processors

Instruction set architecture is a part of processor architecture, which is necessary for creating machine level programs to perform any mathematical or logical operations. Instruction set architecture acts as an interface between hardware and software. It prepares the processor to respond to the commands like execution, deleting etc given by the user.

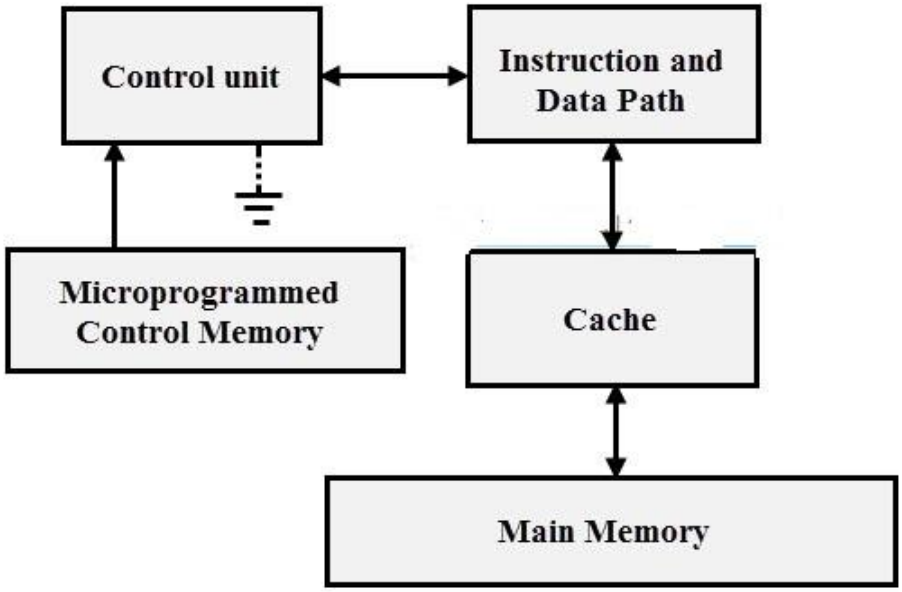
The performance of the processor is defined by the instruction set architecture designed in it. As both software and hardware are required for functioning of a processor, there is dilemma in deciding which should play a major role. Major firms like Intel argues that hardware should play a major role than software. While, Apple’s argument is that software should play a major role in processors architecture.

The two major instruction sets architectures are
1) CISC (Complex Instruction Set Computer)

2) RISC (Reduced Instruction Set Computer)

CISC Architecture

In the early days machines were programmed in assembly language and the memory access is also slow. To calculate complex arithmetic operations, compilers have to create long sequence of machine code. This made the designers to build architecture, which access memory less frequently and reduce burden to compiler. Thus this lead to very power full but complex instruction set. CISC architectures directly use the memory, instead of a register file. The above figure shows the architecture of CISC with micro programmed control and cache memory.



This architecture uses cache memory for holding both data and instructions. Thus, they share the same path for both instructions and data. CISC has instructions with variable length format. Thus, the number of clock cycles required to execute the instructions may be varied. Instructions in CISC are executed by micro program which has sequence of microinstructions.

Advantages of CISC Architecture

- Microprogramming is easy to implement and much less expensive than hard wiring a control unit.
- It is easy to add new commands into the chip without changing the structure of the instruction set as the architecture uses general-purpose hardware to carry out commands.
- This architecture makes the efficient use of main memory since the complexity (or more capability) of instruction allows to use less number of instructions to achieve a given task.
- The compiler need not be very complicated, as the micro program instruction sets can be written to match the constructs of high level languages.

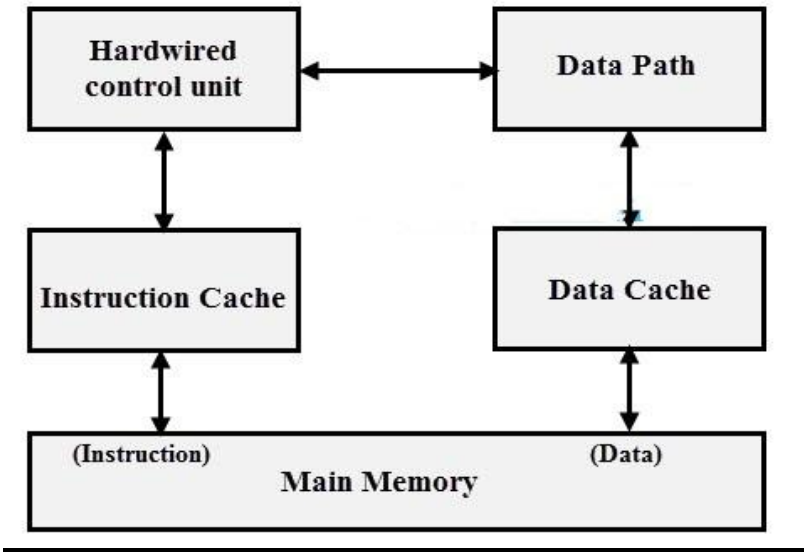
Disadvantages of CISC Architecture

- A new or succeeding versions of CISC processors consists early generation processors in their subsets (succeeding version). Therefore, chip hardware and instruction set became complex with each generation of the processor.
- The overall performance of the machine is reduced because of slower clock speed.
- The complexity of hardware and on-chip software included in CISC design to perform many functions.

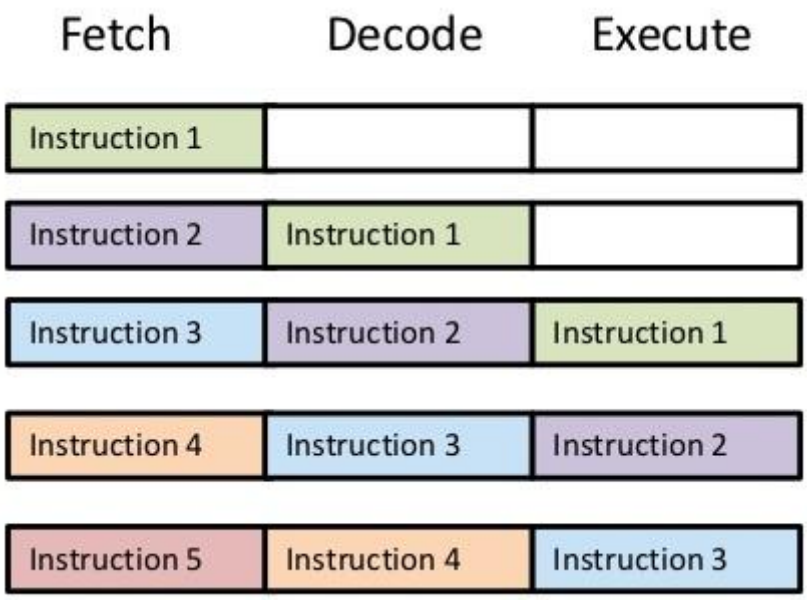
Examples of CISC processor

- IBM 370/168
- Intel 80486
- VAX 11/780

RISC (Reduced Instruction Set Computer) Architecture



Although CISC reduces usage of memory and compiler, it requires more complex hardware to implement the complex instructions. In RISC architecture, the instruction set of processor is simplified to reduce the execution time. It uses small and highly optimized set of instructions which are generally register to register operations. The speed of the execution is increased by using smaller number of instructions .This uses pipeline technique for execution of any instruction. The figure shown below is the architecture of RISC processor, which uses separate instruction and data caches and their access paths also different. There is one instruction per machine cycle in RISC processor.



The pipelining technique allows the processor to work on different steps of instruction like fetch, decode and execute instructions at the same time. Below is image showing execution of instructions in pipelining technique. Generally, execution of second instruction is started, only after the completion of the first instruction. But in pipeline technique, each instruction is executed in number of stages simultaneously. When the first stage of first instruction is completed, next instruction is enters into the first stage. This process continuous until all the instructions is executed.

Advantages of RISC Architecture

- The performance of RISC processors is often two to four times than that of CISC processors because of simplified instruction set.
- This architecture uses less chip space due to reduced instruction set. This makes to place extra functions like floating point arithmetic units or memory management units on the same chip.
- The per-chip cost is reduced by this architecture that uses smaller chips consisting of more components on a single silicon wafer.
- RISC processors can be designed more quickly than CISC processors due to its simple architecture.
- The execution of instructions in RISC processors is high due to the use of many registers for holding and passing the instructions as compared to CISC processors.

Disadvantages of RISC Architecture

- The performance of a RISC processor depends on the code that is being executed. The processor spends much time waiting for first instruction result before it proceeds with next subsequent instruction, when a compiler makes a poor job of scheduling instruction execution.
- RISC processors require very fast memory systems to feed various instructions. Typically, a large memory cache is provided on the chip in most RISC based systems.

Examples of RISC processors

This architecture includes alpha, AVR, ARM, PIC, PA-RISC, and power architecture.

RISC vs. CISC

RISC	CISC
Multiple register sets, often consisting of more than 256 registers	Single register set, typically 6 to 16 registers total
Three register operands allowed per instruction (e.g., `add R1, R2, R3`)	One or two register operands allowed per instruction (e.g., `add R1, R2`)
Parameter passing through efficient on-chip register windows	Parameter passing through inefficient off-chip memory
Single-cycle instructions (except for `load` and `store`)	Multiple-cycle instructions
Hardwired control	Microprogrammed control
Highly pipelined	Less pipelined
Simple instructions that are few in number	Many complex instructions
Fixed length instructions	Variable length instructions
Complexity in compiler	Complexity in microcode
Only load and store instructions can access memory	Many instructions can access memory
Few addressing modes	Many addressing modes

Design Principles for Modern Computers

Now that more than two decades have passed since the first RISC machines were introduced, certain design principles have come to be accepted as a good way to design computers given the current state of the hardware technology. If a major change in technology occurs (e.g., a new manufacturing process suddenly makes memory cycle time 10 times faster than CPU cycle time), all bets are off. Thus machine designers should always keep an eye out for technological changes that may affect the balance among the components.

That said, there is a set of design principles, sometimes called the RISC design principles that architects of general-purpose CPUs do their best to follow. External constraints, such as the requirement of being backward compatible with some existing architecture, often require compromises from time to time, but these principles are goals that most designers strive to meet. Below we will discuss the major ones.

All Instructions Are Directly Executed by Hardware

All common instructions are directly executed by the hardware. They are not interpreted by microinstructions. Eliminating a level of interpretation provides high speed for most instructions. For computers that implement CISC instruction sets, the more complex instructions may be broken into separate parts, which can then be executed as a sequence of microinstructions. This extra step slows the machine down, but for less frequently occurring instructions it may be acceptable.

Maximize the Rate at Which Instructions Are Issued

Modern computers resort to many tricks to maximize their performance, chief among which is trying to start as many instructions per second as possible. After all, if you can issue 500 million instructions/sec, you have built a 500-MIPS processor, no matter how long the instructions actually take to complete. (MIPS stand for Millions of Instructions Per Second; the MIPS processor was so-named as to be a pun on this acronym.) This principle suggests that parallelism can play a major role in improving performance, since issuing large numbers of slow instructions in a short time interval is only possible if multiple instructions can execute at once. Although instructions are always encountered in program order, they are not always issued in program order (because some needed resource might be busy) and they need not finish in program order. Of course, if instruction 1 sets a register and instruction 2 uses that register, great care must be taken to make sure that instruction 2 does not read the register until it contains the correct value. Getting this right requires a lot of bookkeeping but has the potential for performance gains by executing multiple instructions at once.

Instructions should be Easy to Decode

A critical limit on the rate of issue of instructions is decoding individual instructions to determine what resources they need. Anything that can aid this process is useful. That includes making instructions regular, fixed length, with a small number of fields. The fewer different formats for instructions, the better.

Only Loads and Stores Should Reference Memory

One of the simplest ways to break operations into separate steps is to require that operands for most instructions come from—and return to—CPU registers. The operation of moving operands from memory into registers can be performed in separate instructions. Since access to memory can take a long time, and the delay is unpredictable, these instructions can best be overlapped with other instructions if they do nothing but move operands between registers and memory. This observation means that only LOAD and STORE instructions should reference memory. All other instructions should operate only on registers.

Provide Plenty of Registers

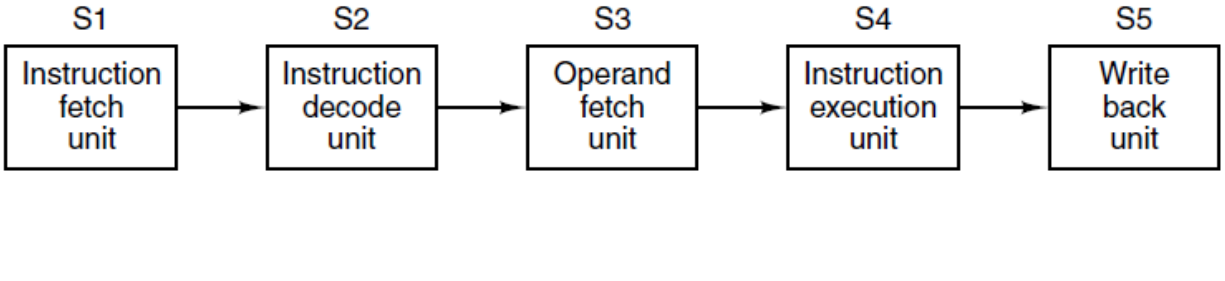
Since accessing memory is relatively slow, many register (at least 32) need to be provided, so that once a word is fetched, it can be kept in a register until it is no longer needed. Running out of registers and having to flush them back to memory only to later reload them is undesirable and should be avoided as much as possible. The best way to accomplish this is to have enough registers.

Instruction-Level Parallelism

Computer architects are constantly striving to improve performance of the machines they design. Making the chips run faster by increasing their clock speed is one way, but for every new design, there is a limit to what is possible by brute force at that moment in history. Consequently, most computer architects look to parallelism (doing two or more things at once) as a way to get even more performance for a given clock speed.
Parallelism comes in two general forms, namely, instruction-level parallelism and processor-level parallelism. In the former, parallelism is exploited within individual instructions to get more instructions/sec out of the machine. In the latter, multiple CPUs work together on the same problem. Each approach has its own merits.

It has been known for years that the actual fetching of instructions from memory is a major bottleneck in instruction execution speed. To alleviate this problem, computers going back at least as far as the IBM Stretch (1959) have had the ability to fetch instructions from memory in advance, so they would be there when they were needed. These instructions were stored in a set of registers called the prefetch buffer. This way, when an instruction was needed, it could usually be taken from the prefetch buffer rather than waiting for a memory read to complete.

In effect, prefetching divides instruction execution up into two parts: fetching and actual execution. The concept of a pipeline carries this strategy much further. Instead of dividing instruction execution into only two parts, it is often divided into many (often a dozen or more) parts, each one handled by a dedicated piece of hardware, all of which can run in parallel.



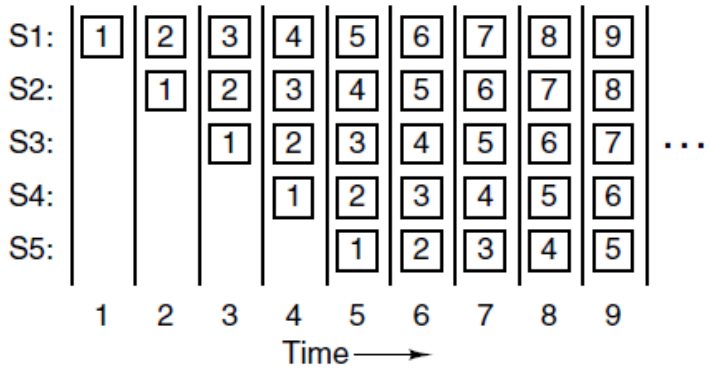
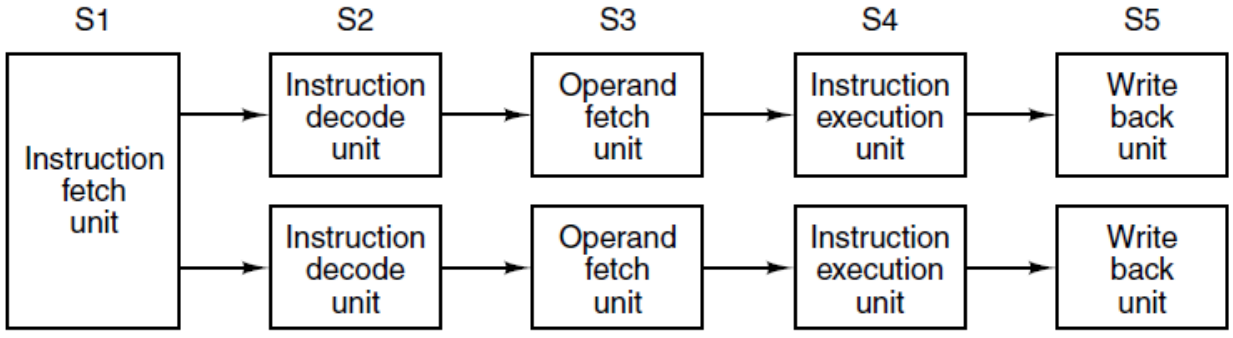


Figure above illustrates a pipeline with five units, also called stages. Stage 1 fetches the instruction from memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its type and what operands it needs. Stage 3 locates and fetches the operands, either from registers or from memory.
Stage 4 actually does the work of carrying out the instruction, typically by running the operands through the data path of Figure below. Finally, stage 5 writes the result back to the proper register.

We see how the pipeline operates as a function of time. During clock cycle 1, stage S1 is working on instruction 1, fetching it from memory. During cycle 2, stage S2 decodes instruction 1, while stage S1 fetches instruction 2. During cycle 3, stage S3 fetches the operands for instruction 1, stage S2 decodes instruction 2, and stage S1 fetches the third instruction. During cycle 4, stage S4 executes instruction 1, S3 fetches the operands for instruction 2, S2 decodes instruction 3, and S1 fetches instruction 4. Finally, during cycle 5, S5 writes the result of instruction 1 back, while the other stages work on the following instructions.

Superscalar Architectures

If one pipeline is good, then surely two pipelines are better. One possible design for a dual pipeline CPU, Here a single instruction fetch unit fetches pairs of instructions together and puts each one into its own pipeline, complete with its own ALU for parallel operation. To be able to run in parallel, the two instructions must not conflict over resource usage (e.g., registers), and neither must depend on the result of the other. As with a single pipeline, either the compiler must guarantee this situation to hold (i.e., the hardware does not check and gives incorrect results if the instructions are not compatible), or conflicts are detected and eliminated during execution using extra hardware.
Although pipelines, single or double, are mostly used on RISC machines (the 386 and its predecessors did not have any), starting with the 486 Intel began introducing data pipelines into its CPUs. The 486 had one pipeline and the original Pentium had two five-stage pipelines roughly, although the exact division of work between stages 2 and 3 (called decode-1 and decode-2) was slightly different than in our example. The main pipeline, called the u pipeline, could execute an arbitrary Pentium instruction. The second pipeline, called the v pipeline, could execute only simple integer instructions.



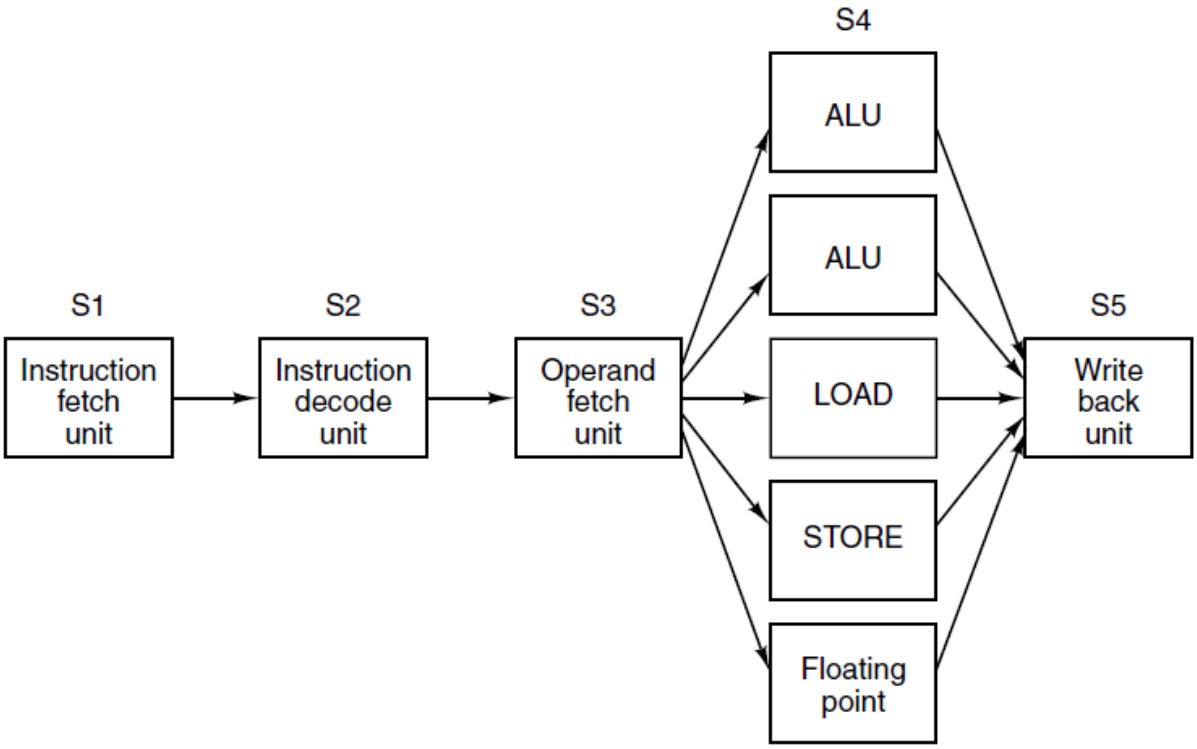
Fixed rules determined whether a pair of instructions were compatible so they could be executed in parallel. If the instructions in a pair were not simple enough or incompatible, only the first one was executed (in the u pipeline). The second one was then held and paired with the instruction following it. Instructions were always executed in order. Thus Pentium-specific compilers that produced compatible pairs could produce faster-running programs than older compilers. Measurements showed that a Pentium running code optimized for it was exactly twice as fast on integer programs as a 486 running at the same clock rate.

This gain could be attributed entirely to the second pipeline. Going to four pipelines is conceivable, but doing so duplicates too much hardware (computer scientists, unlike folklore specialists, do not believe in the

number three). Instead, a different approach is used on high-end CPUs. The basic idea is to have just a single pipeline but give it multiple functional units, as shown in Fig.

For example, the Pentium II has a structure similar to this figure. The term superscalar architecture was coined for this approach in 1987. Its roots, however, go back more than 40 years to the CDC 6600 computer. The 6600 fetched an instruction every 100 nsec and passed it off to one of 10 functional units for parallel execution while the CPU went off to get the next instruction.

The definition of ‘‘superscalar’’ has evolved somewhat over time. It is now used to describe processors that issue multiple instructions—often four or six—in a single clock cycle. Of course, a superscalar CPU must have multiple functional units to hand all these instructions to. Since superscalar processors generally have one pipeline, they tend to look like.



Implicit in the idea of a superscalar processor is that the S3 stage can issue instructions considerably faster than the S4 stage is able to execute them. If the S3 stage issued an instruction every 10 nsec and all the functional units could do their work in 10 nsec, no more than one would ever be busy at once, negating the whole idea. In reality, most of the functional units in stage 4 take appreciably longer than one clock cycle to execute, certainly the ones that access memory or do floating-point arithmetic. As can be seen from the figure, it is possible to have multiple ALUs in stage S4.

Processor-Level Parallelism

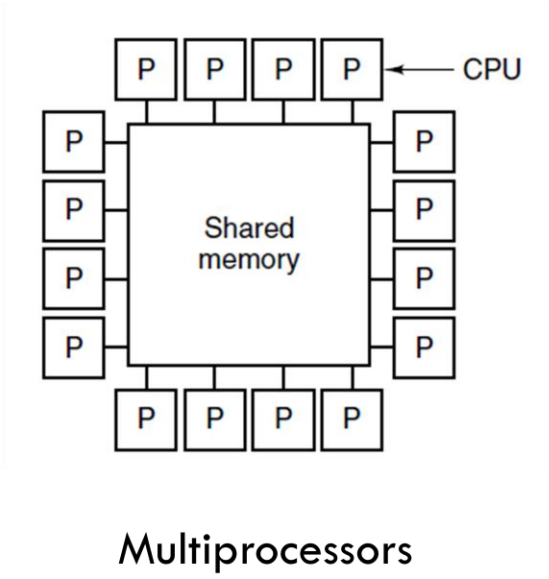
The demand for ever faster computers seems to be insatiable. Astronomers want to simulate what happened in the first microsecond after the big bang, economists want to model the world economy, and teenagers want to play 3D interactive multimedia games over the Internet with their virtual friends. While CPUs keep getting faster, eventually they are going to run into the problems with the speed of light, which is likely to stay at 20 cm/nanosecond in copper wire or optical fiber, no matter how clever Intel’s engineers are. Faster chips also produce more heat, whose dissipation is a problem. Instruction-level parallelism helps a little, but pipelining and

superscalar operation rarely win more than a factor of five or ten. To get gains of 50, 100, or more, the only way are to design computers with multiple CPUs, so we will now take a look at how some of these are organized.

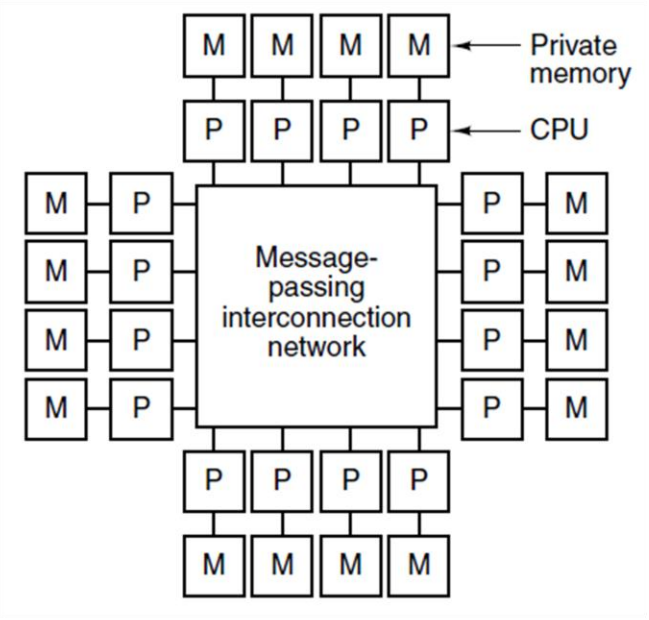
Multiprocessors

The processing elements in an array processor are not independent CPUs, since there is only one control unit shared among all of them. Our first parallel system with multiple full-blown CPUs is the multiprocessor, a system with more than one CPU sharing a common memory, like a group of people in a room sharing a common blackboard. Since each CPU can read or write any part of memory, they must co-ordinate (in software) to avoid getting in each other's way.

When two or more CPUs have the ability to interact closely, as is the case with multiprocessors, they are said to be tightly coupled. Various implementation schemes are possible. The simplest one is to have a single bus with multiple CPUs and one memory all plugged into it.



Multiprocessors



Multicomputers

Multicomputers

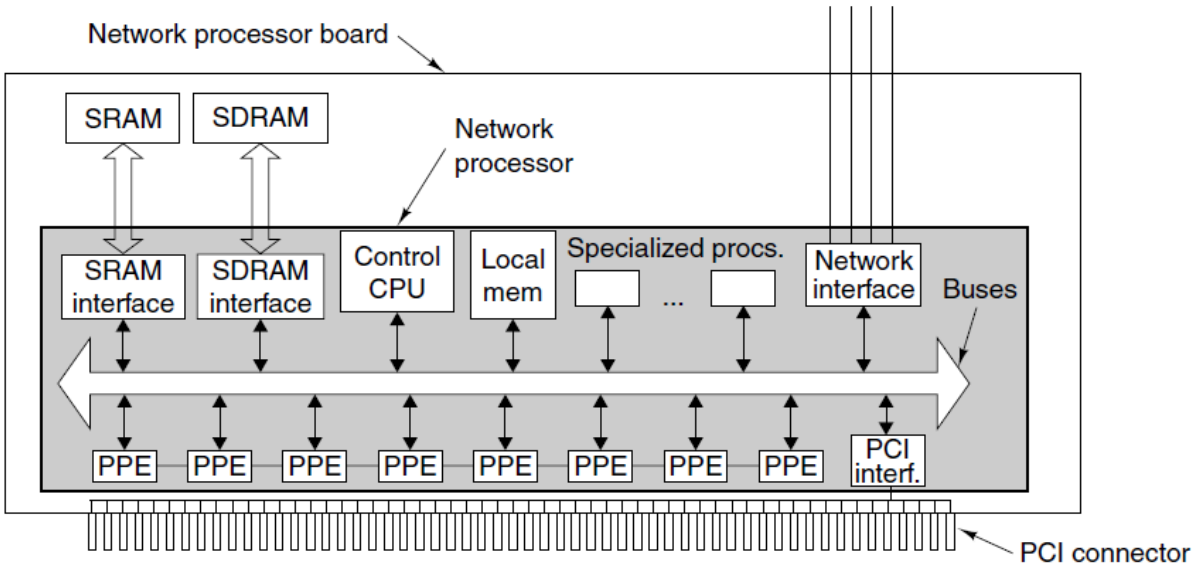
Although multiprocessors with a modest number of processors (≤ 256) are relatively easy to build, large ones are surprisingly difficult to construct. The difficulty is in connecting all the processors to the memory. To get around these problems, many designers have simply abandoned the idea of having a shared memory and just build systems consisting of large numbers of interconnected computers, each having its own private memory, but no common memory. These systems are called multicomputers. The CPUs in a multicomputer are sometimes said to be loosely coupled, to contrast them with the tightly-coupled CPUs in a multiprocessor. The CPUs in a multicomputer communicate by sending each other messages, something like e-mail, but much faster. For large systems, having every computer connected to every other computer is impractical, so topologies such as 2D and 3D grids, trees, and rings are used. As a result, messages from one computer to another often must pass through one or more intermediate computers or switches to get from the source to the destination. Nevertheless, message-passing times on the order of a few microseconds can be achieved without much difficulty. Multicomputers with nearly 10,000 CPUs have been built and put into operation.

COPROCESSORS

Having examined some of the ways of achieving on-chip parallelism, let us now move up a step and look at how the computer can be speeded up by adding a second, specialized processor. These coprocessors some in many varieties, from small to large. On the IBM 360 mainframes and all of its successors, independent I/O channels exist for doing input/output. Similarly, the CDC 6600 had 10 independent processors for doing I/O. Graphics and floating-point arithmetic are other areas where coprocessors have been used. Even a DMA chip can be seen as a coprocessor. In some cases, the CPU gives the coprocessor an instruction or set of instructions and tells it to execute them; in other cases, the coprocessor is more independent and runs pretty much on its own. Physically, coprocessors can range from a separate cabinet (the 360 I/O channels) to a plug-in board (network processors) to an area on the main chip (floating-point). In all cases, what distinguishes them is the fact that some other processor is the main processor and the coprocessors are there to help it. We will now examine the areas where speedups are possible: network processing, multimedia, and cryptography.

Network Processors

Most computers nowadays are connected to a network or to the Internet. As a result of technological progress in network hardware, networks are now so fast that it has become increasingly difficult to process all the incoming and outgoing data in software. As a consequence, special network processors have been developed to handle the traffic, and many high-end computers now have one of these processors.



Finally, we come to network processors, programmable devices that can handle incoming and outgoing packets at wire speed (i.e., in real time). A common design is a plug-in board containing a network processor on a chip along with memory and support logic. One or more network lines connect to the board and are routed to the network processor. There packets are extracted, processed, and either sent out on a different network line (e.g., for a router) or are sent out onto the main system bus (e.g., the PCI bus) in the case of end-user device such as a PC. A typical network processor board and chip are illustrated in Fig Both SRAM and SDRAM are provided on the board and typically used in different ways. SRAM is faster, but more expensive, than SDRAM, so there is only a small amount of it. SRAM is used to hold routing tables and other key data structures, whereas SDRAM holds the actual packets being processed.

By making the SRAM and SDRAM external to the network processor chip, the board designers are given the flexibility to determine how much of each to supply. In this way, low-end boards with a single network line (e.g., for a PC or server) can be equipped with a small amount of memory whereas a high-end board for a large router can be equipped with much more. Network processor chips are optimized for processing large numbers of incoming and outgoing packets quickly. Millions of packets per second per network line, that is, and a router could easily have half a dozen lines. The only way to achieve such processing rates is to build network processors that are highly parallel inside. And indeed, all network processors consist of multiple PPEs, variously called Protocol/Programmable/Packet Processing Engines.

Each one is a (possibly modified) RISC core and a small amount of internal memory for holding the program and some variables. The PPEs can be organized in two different ways. The simplest organization is having all the PPEs being identical. When a packet arrives at the network processor, either an incoming packet from a network line or an outgoing packet from the bus, it is handed to an idle PPE for processing. If no PPE is idle, the packet is queued in the on-board SDRAM until a PPE frees up. When this organization is used, the horizontal connections shown between the PPEs in Fig. do not exist because the PPEs have no need to communicate with one another. The other PPE organization is the pipeline. In this one, each PPE performs one processing step and then feeds a pointer to its output packet to the next PPE in the pipeline. In this way, the PPE pipeline acts very much like the CPU pipelines. In both organizations, the PPEs are completely programmable.

In addition to the PPEs, all network processors contain a control processor, usually just a standard general-purpose RISC CPU, for performing all work not related to packet processing, such as updating the routing tables. Its program and data are in the local on-chip memory. Furthermore, many network processor chips also contain one or more specialized processors for doing pattern matching or other critical operations. These processors are really small ASICs that are good at doing one simple operation, such as looking up a destination address in the routing table. All the components of the network processor communicate over one or more on-chip, parallel buses that run at multigigabit/sec speeds.

Media Processors

A second area in which coprocessors are used is for handling high-resolution photographic images, audio, and video streams. Ordinary CPUs are not especially good at the massive computations needed to process the large amounts of data required in these applications. For this reason, some current PCs and most future PCs will be equipped with media coprocessors to which they can offload large portions of the work.

The Nexperia Media Processor

We will study this increasingly important area by means of an example: the Philips Nexperia, a family of chips that is available at several clock frequencies. The Nexperia is a self-contained single-chip heterogeneous multiprocessor in the sense of Fig. It contains multiple cores, including a TriMedia VLIW CPU for control, but also numerous cores for image, audio, video, and networking processing.

It can be used either as a stand-alone main processor in a CD, DVD, or MP3 player or recorder, TV set or set-top box, still or video camera, etc., or as a coprocessor in a PC for processing images and media streams. In both configurations, it runs its own small real-time operating system.

The Nexperia has three functions: capturing input streams and converting them to data structures in memory, processing these data structures, and finally, outputting them in forms suitable for the various output devices attached. For example, when a PC is used as a DVD player, the Nexperia can be programmed to read the encrypted, compressed video stream from the DVD disc, decrypt and decompress it, and then output it at a

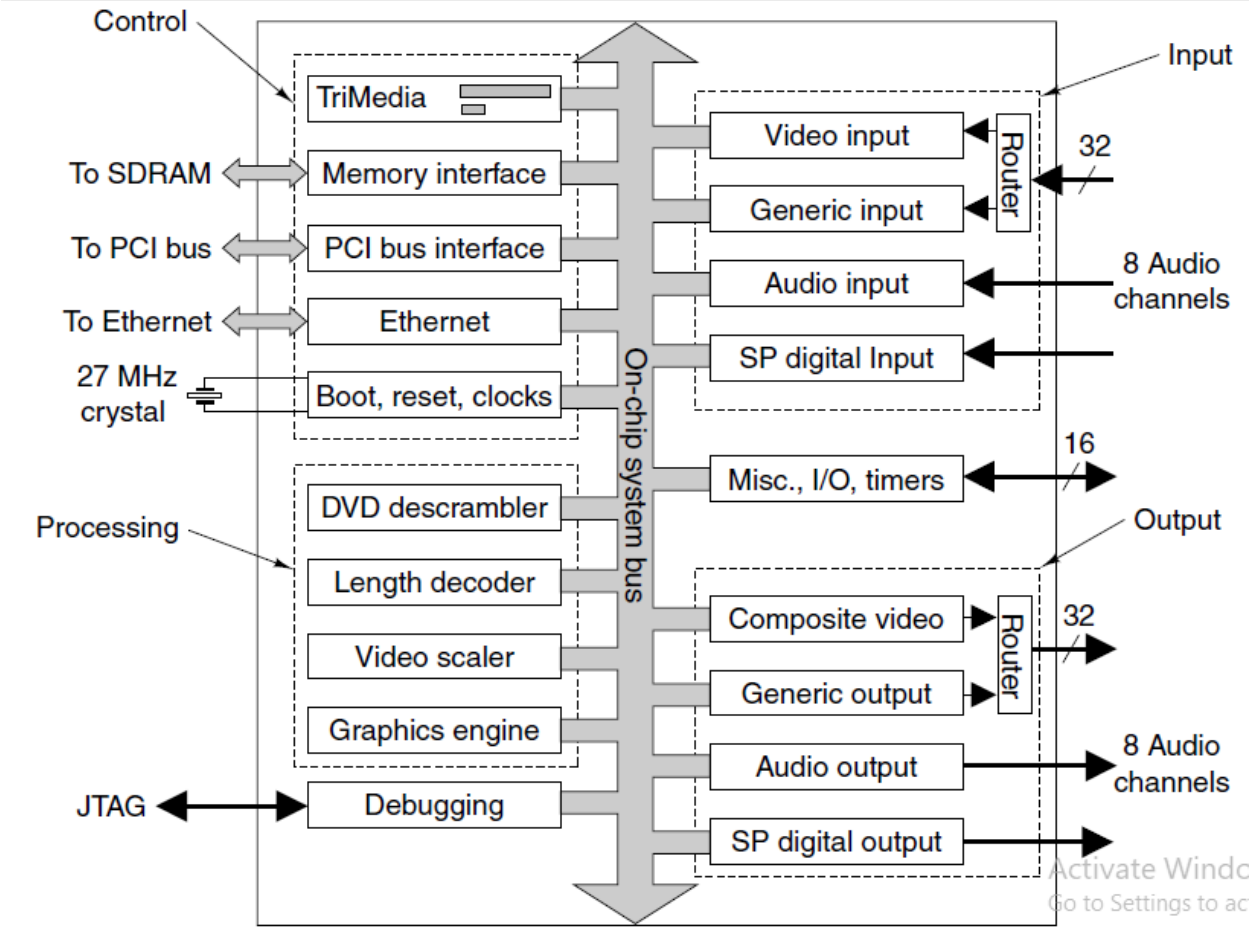
size appropriate to the window it is being displayed in. And all of this can be done in the background, without involving the computer's main CPU at all once the DVD player program has been loaded into the Nexperia.

All incoming data are first stored in memory for processing; there is no direct connection between input devices and output devices. Capturing input includes decoding from a wide variety of video sizes and formats (including MPEG-1, MPEG-2, and MPEG-4), audio formats (including AAC, Dolby, and MP3) and converting to appropriate data structures for storage and processing. Input can come from the PCI bus, Ethernet, or dedicated input lines (e.g., a microphone or stereo system plugged directly into the chip). The Nexperia chip has 456 pins, some of which are available for direct input and output of media (and other) streams.

Data processing is controlled by the software in the TriMedia CPU, which can be programmed for whatever is needed. Typical tasks include deinterlacing video to improve its sharpness, correcting the brightness, contrast, and color of images, scaling image size, converting between different video formats, and reducing noise. Usually, the CPU acts as the general contractor, subcontracting out much of the work to the specialized cores on the chip.

Output functionality includes coding the data structures in a form suitable for the output device, merging multiple (video, audio, image, 2D-graphics) data sources, and controlling the output devices. As with input, output can go to the PCI bus, Ethernet, or dedicated output lines (e.g., a loudspeaker or amplifier).

A block diagram of the Nexperia PNX 1500 chip is given in Fig. Other versions are slightly different, so to be consistent, throughout this section when we say ''Nexperia'' we mean the PNX 1500 implementation. It has four major sections: control, input, processing, and output. The CPU is the 32-bit TriMedia VLIW processor discussed running at 300 MHz. Its program, usually written in C or C++, determines the Nexperia's functionality.



A full PCI interface is also included on chip, with 8-, 16-, and 32-bit transfers at 33 MHz. When used as the main CPU inside a consumer electronics device (e.g., a DVD player), the PCI interface can also act as bus arbiter. This interface can be used, for example, to talk to a DVD drive. Direct ethernet connectivity is provided by a dedicated core that can handle 10- and 100-Mbps Ethernet connections. Consequently, a Nexperia-based camcorder can output a digital video stream over an Ethernet to a remote capture or display device.

The next core handles booting, resetting, clocks, and some other minor features. If a certain pin on the Nexperia is asserted, a reset is initiated. The core can also be programmed as a dead man's switch. If the CPU fails to ping it for a certain period of time, it assumes the system has hung and initiates a reboot on its own. In stand-alone devices, rebooting can be done from a flash memory.

The core is driven by an external 27 MHz crystal oscillator, which is multiplied internally by 64 to give a 1.728 GHz signal used throughout the chip. Power management is also handled here. Normally, the CPU runs at full speed and the other components run at whatever speed they have to to get their work done.

However, it is possible for the CPU to slow down the clock to save energy. A sleep mode is also provided to shut down most functions when there is no work to do, thus conserving battery charge on mobile devices