SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMMUNICATION ENGINEERING

UNIT - 1
ADVANCED MICROPROCESSORS – SEC1601

UNIT 1-ADVANCED MICROPROCESSOR ARCHITECTURE

Internal Microprocessor Architecture, Real mode memory addressing, Protected mode memory addressing, Memory pagingData addressing modes - Program memory addressing modes - Stack memory addressing modes - Data movement instructions - Program control instructions Arithmetic and Logic Instructions. Intel 80186 – Architecture

UNIT 1 –INTERNAL MICROPROCESSORARCHITECTURE

On the basis of programming Microprocessor core

SINGLE CORE

1. **Single task at any time**
2. **Access whole memory at the same time by any program**

DUAL CORE

1. **Multiple task at any time**
2. **Access specific memory by any task**

PROGRAMMING MODEL

Program visible: 8086,8088-registers are used directly during programming

Program invisible: 80286 and above

1. **Registers are not accessible directly during programming**
2. **Can be used indirectly**

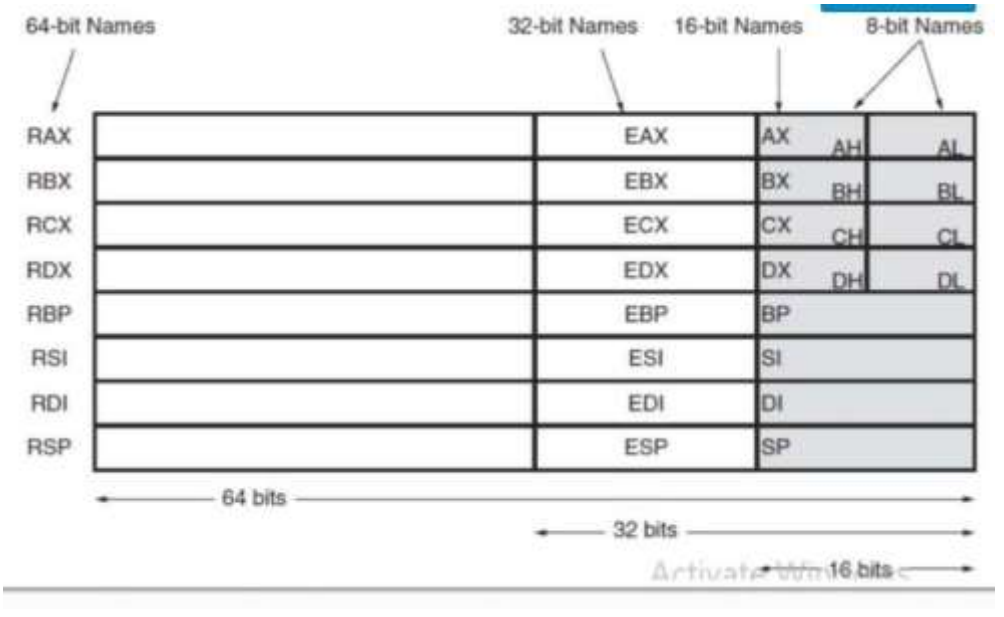


Fig 1.1 :Programming model

Fig 1.2: 64 bit registers

64 bit cannot be accessed with the instruction used to access general purpose register AX,BX Etc.,

To Access Low Order Byte Of R8 is used

To Access Low Order Word Of R10 is used

R8-R15- these registers are found only in Pentium 4 and core 2 if 64 bit extensions are enables, used as general purpose register

In most applications these registers remain unused

Register Size	*Override*	*Bits Accessed*	*Example*
8 bits	B	7–0	MOV R9B, R10B
16 bits	W	15–0	MOV R10W, AX
32 bits	D	31–0	MOV R14D, R15D
64 bits	—	63–0	MOV R13, R12

Fig 1.3: Register size with example

SEGMENT REGISTER

THERE ARE 6 SEGMENT REGISTERS

Stack segment(SS)- pointer to stack
Code segment(CS)-pointer to code
Data segment (DS)-pointer to data
Extra segment(ES)-pointer to extra data ('E' stands for extra)
F segment (FS)-pointer to more extra data('F' comes after 'E')
G segment (GS)- pointer to still more extra data('G' comes after 'F')

CS(code segment) – area of memory that hold the executable program used by microprocessor, Have the starting address of the memory that hold code

CS:IP used to access the whole 64K memory of code segment

SS(stack segment): area of memory used for the stack, stack is used to store data, memory 64K can access by SS:SP

DS(data segment): area of memory that hold all data refer by general purpose register(AX,BX etc), 64K

memory accessed by using DS and offset address register

ES(extra segment)- Additional data segment that is used by some of the string instructions to hold destination data

FS and GS- supplemental register used as most extra segment in 64 bit MP,Window use these segments for internal operation , but no definition for their usage is available POINTER AND INDEX REGISTERS
Stack pointer : Pointing the top of the stack in stack segment

Base Pointer: contain the offset address within data segment

Source Index: used as a pointer to a source in stream operations

Destination index: used as a pointer to destination in a stream operations

All these registers also used for based indexed, register indirect addressing

MULTIPURPOSE OR GENERAL PURPOSE REGISTER

Accumulator (RAX)- used for I/O operation, rotate, shift, multiplication and division

Base index(RBX)- hold the offset address of a memory location in a memory system and also address the memory data

Count register (CX)- Used as default counter for various instruction like repeated STRING, SHIFT,ROTATE(CL),LOOP etc.,

Hold the offset address of memory data and also address memory data

Data register (DX) hold the part of the result from a multiplication or division and also address memory data

SPECIAL PURPOSE REGISTERS

Instruction pointer (IP) :Point to the next instruction in a program located within the code segment
Stack pointer: pointing to the top of the stack in stack segment to store data
Flag Register: Determine the current status of the process modified after executing the instruction

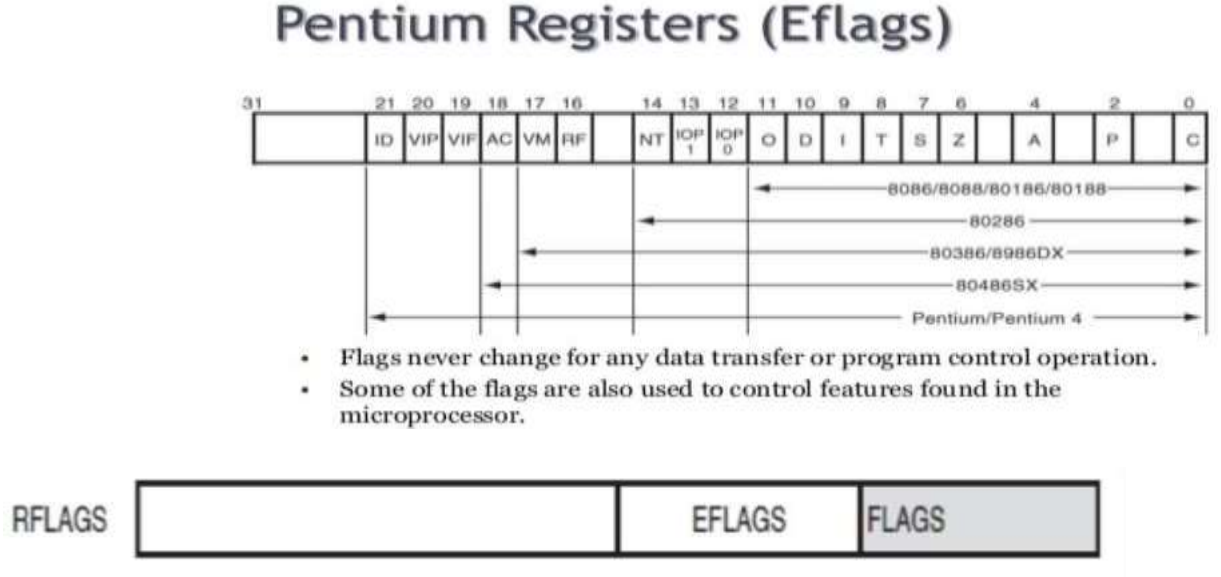


Fig 1.4: Pentium Flag registers

FLAG register for the entire 8086 and Pentium microprocessor family

FLAG REGISTERS

The **FLAG** register is the status register in Intel x86 microprocessors that contains the current state of the processor. This register is 16 bits wide. Its successors, the **EFLAGS** and **RFLAGS** registers, are **32 bits** and **64 bits** wide, respectively.

Conditional Flag
Carryflag: indication of overflow condition for unsigned integer and hold carry
Auxiliaryflag: indication of carry/borrow from lower nibble to higher nibble (D3 to D4)
Parity flag: indication of parity of the result, 0-ODD, 1-EVEN
Zero flag: set if the result is zero, Z-1 result is 0, Z-0 result is non zero
Sign flag: in signed magnitude format if the result of the operation is negative sign flag is set, S-1 result is NEGATIVE, S-0 result is POSITIVE
CONTROL FLAGS-CONTROL THE OPERATION OF THE EXECUTION UNIT
Trap flag: if set, then debugging operation is allowed

- **Interrupt flag: control the operation of interrupt request**
- **[INTR PIN-1 Enabled,0-Disabled]**
- **STI- set by this instruction, CLI- cleared by this instruction**
- **Direction flag: used in string operation-if set the string byte is accessed from higher memory address to lower memory address**
- **Flags for 32,64 bit**
- **IOPL(I/O privilege level):**
- **Used to select the privilege level for I/O devices**
- **If the current privilege level is 00 , I/O execute without hindrance**
- **If 11 then an interrupt occur and suspended the execution**
- **Nested task(NT)- indicate whether the task is nested with other or not**
- **RF(resume): resume flag is used with debugging to control the resumption of execution after the next instruction**

FLAG REGISTERS

- **VM(virtual mode)- allow the system program to execute multiple DOS operation**
- **AC(alignment check)- if the word or double word is addressed on a non-word or non-double word boundary**
- **VIF(virtual interrupt flag)-copy the interrupt flag bit (used in Pentium 4)**
- **VIP(virtual interrupt pending)-give the information about virtual mode interrupt. Set if interrupt if pending**
- **ID(identification)- The ID flag indicate that Pentium 4 microprocessor support the CPUID or not-CPUID instruction provides the system with information about the microprocessor such as its version number and manufacturer**

REAL MODE ADDRESSING

The only mode available on the 8086-8088

- **20 bit address bus- 1MB, 16 bit data bus, 16 bit registers**
- **80286 and above operate in either real or protected mode**
- **Real mode operation : allows addressing of only the first 1M byte of memory space-even in Pentium 4 or core 2 microprocessor**
- **The first 1M byte of memory is called the real memory, conventional memory or DOS memory system.**
- **Segment registers (CS,DS,SS,ES) holds the base address of where a particular segment begins in memory**

SEGMENTS AND OFFSETS

All real mode memory addresses must consist of a segment address plus an offset address

- **Segment address defines the beginning address of any 64K byte memory segment**
- **Offset address: Selects any location within the 64K byte memory segment**

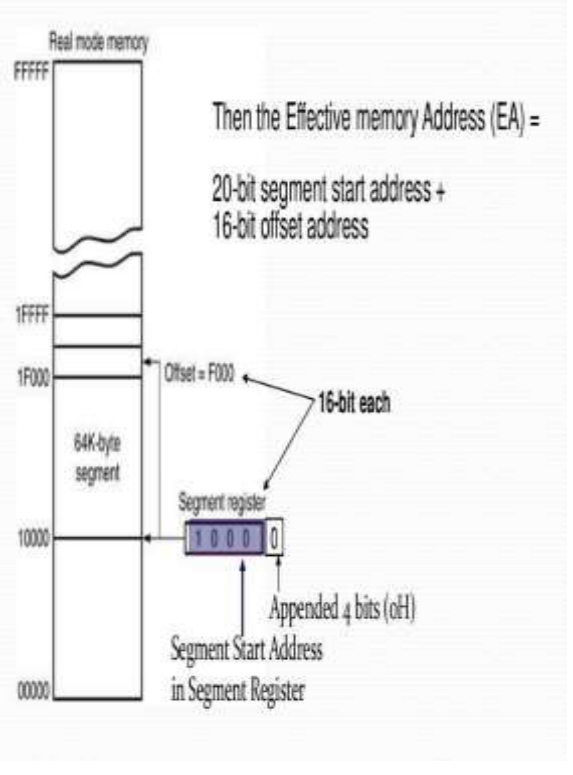


Fig 1.5: Segment and offset

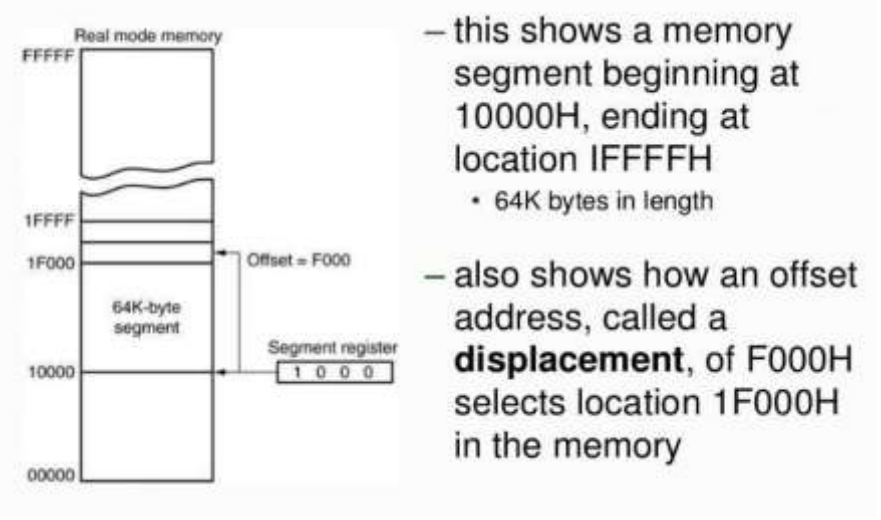


Fig 1.6: Segment and offset with displacement

- **Once the starting address is known the ending address is found by adding FFFFH**
- **Because a real mode segment of memory is 64K in length**
- **The Offset address is always added to the segment starting address to locate the data • [1000:2000H]**
- **A segment address of 1000H: an offset address of 2000H**

EFFECTIVE ADDRESS CALCULATION

- **EA= Segment register (SR) x 10H + Offset**
- **1. SR:1000H**
- **10000+0023=10023H**
- **2.SR:AAF0H**
- **AAF00 + 0134 = AB034H**
- **3.SR:1200H**
- **12000 + FFF0 =21FF0H**

Segment Register	*Starting Address*	*Ending Address*
2000H	20000H	2FFFFH
2001H	20010H	3000FH
2100H	21000H	30FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

Fig 1.7: Effective address calculation

- **segment and offset register combination [CS:IP]**
- **The code segment register defines the start of the code segment**

- **The instruction pointer locates the next instruction within the code segment**
- **Another default combination is SS:SP or SS:BP**

Segment	*Offset*	*Special Purpose*
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8- or 16-bit number	Data address
ES	DI for string instructions	String destination address

Fig 1.8: Default 16 bit segment and offset combinations

Segment	*Offset*	*Special Purpose*
CS	EIP	Instruction address
SS	ESP or EBP	Stack address
DS	EAX, EBX, ECX, EDX, ESI, EDI, an 8- or 32-bit number	Data address
ES	EDI for string instructions	String destination address
FS	No default	General address
GS	No default	General address

Fig 1.9 Default 32-bit segment and offset combinationREAL MODE MEMORY ADDRESSING

The only mode available on the 8086-8088

20 bit address bus- 1MB, 16 bit data bus, 16 bit registers

80286 and above operate in either real or protected mode

Real mode operation : allows addressing of only the first 1M byte of memory space-even in Pentium 4 or core 2 microprocessor

The first 1M byte of memory is called the real memory, conventional memory or DOS memory system.

Segment registers (CS,DS,SS,ES) holds the base address of where a particular segment begins in memory

Memory segmentation

Memory segmentation is nothing which is the methods where whole memory is divided into the smaller parts.

In 8086 microprocessor memory are divided into four parts which is known as the segments.

These segments are data segment, code segment, stack segment and extra segment.

The total memory size is divided into segments of various sizes.
A segment is just an area in memory. The process of dividing memory this way is called Segmentation.
In memory, data is stored as bytes

Each byte has a specific address.

Intel 8086 has 20 lines address bus.

With 20 address lines, the memory that can be addressed is 2 power20 bytes.

2power20= 1,048,576 bytes (1 MB).

8086 access memory with address ranging from 00000 H to FFFFF H.

SEGMENTS

In 8086, memory has four different types of segments. They are:

Code Segment

Data Segment

Stack Segment

Extra Segment

SEGMENT REGISTERS

Each of these segments are addressed by an address stored in corresponding segment register.

These registers are 16-bit in size.

Each register stores the base address (starting address) of the corresponding segment.

The segment registers cannot store 20 bits, they only store the upper 16 bits.

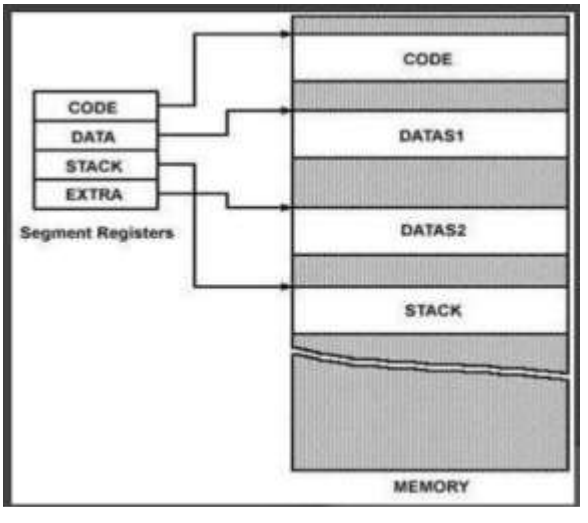


Fig1.10: segment registers

PHYSICAL ADDRESS TRANSLATION MECHANISM

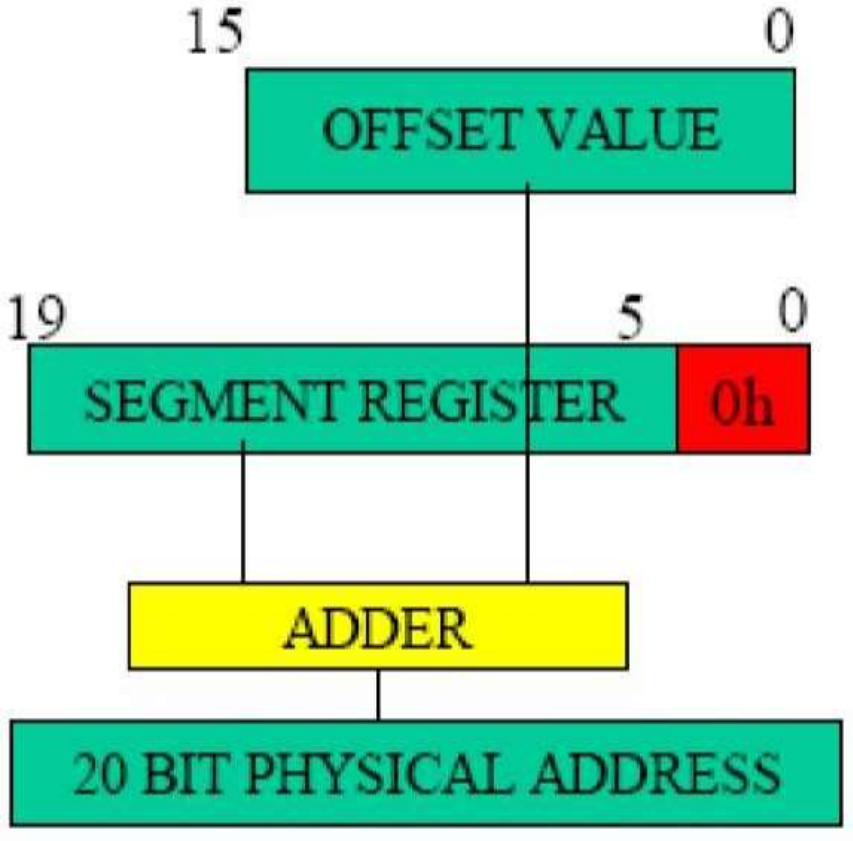


Fig 1.11Physical address translation mechanism

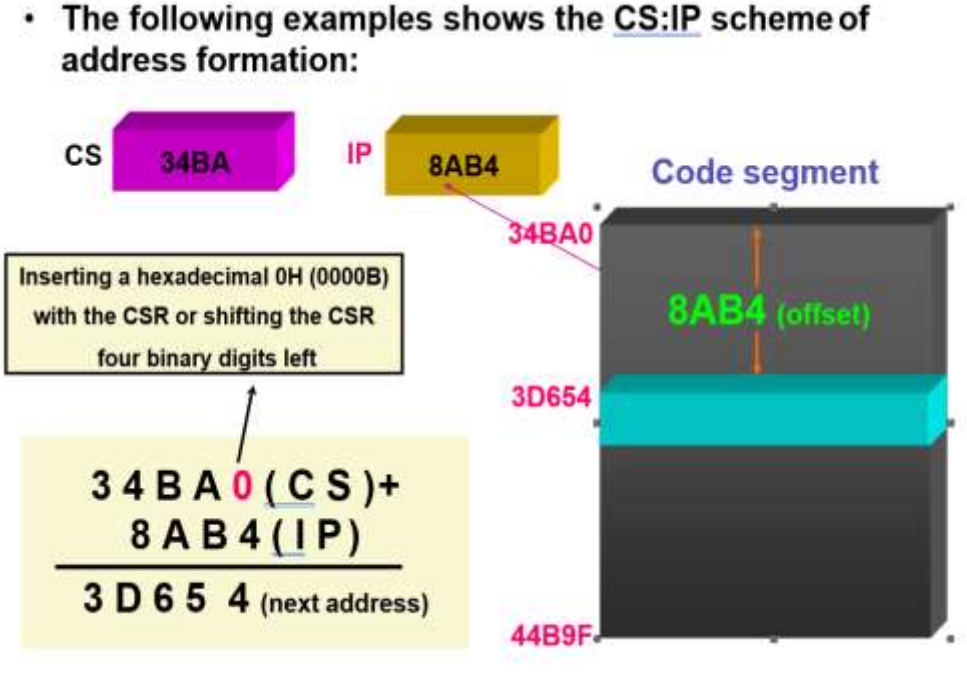


Fig1.12: Example of address formation

Example

The value of Data Segment Register (DS) is 2222H.

- **To convert this 16-bit address into 20-bit the BIU appends 0H to the LSBs of the address by address translation mechanism.**
- **After appending, the starting address of the Data Segment becomes 22220H.**

- **The 20-bit address of a byte is called its Physical Address.**
- **Logical address is in the form of: Base Address : Offset**
- **Offset is the displacement of the memory location from the starting location of the segment.**

EXAMPLE

If the data at any location has a logical address specified as:

2222 H : 0016 H

Then, the number 0016 H is the offset 2222 H is the value of DS.

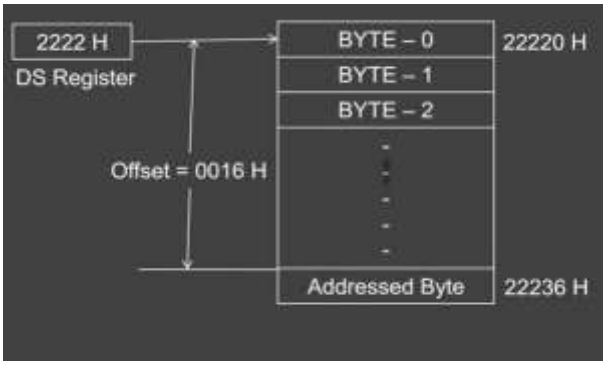


Fig 1.13: Example

Maximum size of segment

All offsets are limited to 16-bits.

It means that the maximum size possible for segment is 2 power 16 = 65,535 bytes (64KB).

The offset of the first location within the segment is 0000H.

T h e offset of the last location in the segment is FFFFH

Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP, BP	Address in the stack
ES	BX, DI, SI	Address of destination data (for string operations)

Fig:1.14 register combination to form physical address

Example
The contents of the following registers are:
CS = 1111 H DS = 3333 H SS = 2526 H IP = 1232 H SP = 1100 H DI = 0020 H
Calculate the corresponding physical addresses for the address bytes in CS, DS and SS.

Solution
1. CS = 1111 H The base address of the code segment is 11110 H.
Effective address of memory is given by 11110H + 1232H = 12342H.
2. DS = 3333 H The base address of the data segment is 33330 H.
Effective address of memory is given by 33330H + 0020H = 33350H.
3. SS = 2526 H The base address of the stack segment is 25260 H.
Effective address of memory is given by 25260H + 1100H = 26350H.

REAL MODE ADDRESSING SCHEME ALLOWRELOCATION

A relocatable program is one that can be placed into any area of memory and executed without change

Segment plus offset addressing allows DOS programs to be relocated in memory

This mainly means using relative offsets for data accesses and jump instructions. If this is easy/ possible is based on the type of architecture the size of the address space and size of the program.

Relocatable data are data that can be placed in any area of memory and used without any change to the program

Because memory is addressed within a segment by an offset address, the memory segment can be moved to any place in the memory system without changing any of the offset addresses

Only the contents of the segment register must be changed to address the program in the new area of memory

Windows programs are written assuming that the first 2G of memory are available for code and data

- **TRANSIENT PROGRAM AREA**

TPA holds the Disk OS, other programs that control the computer system
TPA is the first available area of memory above drivers and other TPA programs.
Area is indicated by a free-pointer maintained by DOS
Program loading is handled automatically by the program loader within DOS
TPA also stores any currently active or inactive DOS application programs

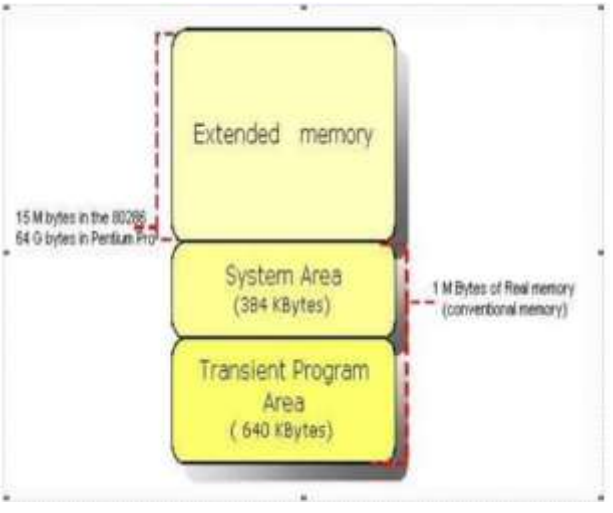


Fig 1.15: Transient program area

PROTECTED ADDRESSING MODE

It allow system software to use features such as virtual memory, paging and multi tasking designed to increase an operating system's control over application software.
Also real mode code is never in 32 bits whereas protected mode code can be 16 bits or 32 bits. Every x86 CPU starts in real mode. Protected mode starts after OS sets up several descriptor tables and enables the Protection Enable (PE) bit in the control register 0.
Allows access to data and programs located within and above the first 1M byte of memory.
Addressing this extended section of the memory system requires a change to the segment plus an offset addressing scheme used with real mode memory addressing. Protected mode is where Windows operates Used by Windows NT, 2000, XP, Linux. Protected mode memory addressing allows access to data and programs located above the first 1M byte of memory.
Addressing this extended section of the memory system requires a change to the segment plus an offset addressing scheme used with real mode memory addressing

The selector, located in the segment register, selects one of 8192 descriptors from one of two tables of descriptors (stored in memory): the global and local descriptor tables. The descriptor describes the location, length and access rights of the memory segment. Each descriptor is 8 byteslong. The 8192 descriptor table requires 8 * 8192 = 64K bytes of memory. In protected addressing mode segments can be of variable size (below or above 64 KB). Some system control instructions are only valid in the protected mode. The offset part of the memory address is still the same as in real addressing mode. However, when in the protected mode, the processor can work either with 16-bit offsets or with 32- bit offsets. A 32-bit offset allows segments of up to 4G bytes. Notice that in real-mode the only available instruction mode is the 16-bit mode. One difference between real and protected addressing mode is that the segment address, as discussed with real mode memory addressing, is no

longer present in the protected mode. In place of the segment address, the segment register contains a selector . The selector selects a descriptor from a descriptor table. descriptor that describes the starting address and length of a section of memory. The descriptor describes the memory segment's location, length, and access rights. This is similar to selecting one card from a deck of cards in one's pocket. The selector, located in the segment register, selects one of 8192 descriptors from one of two tables of descriptors (stored in memory): the global and local descriptor tables. The descriptor describes the location, length and access rights of the memory segment. Each descriptor is 8 bytes long. The 8192 descriptor table requires 8 * 8192 = 64K bytes of memory.

SELECTORS AND DESCRIPTORS

•The selector is located in the segment register. Describes the location, length, and access rights of the segment of memory. it selects one of 8192 descriptors from one of two tables of descriptors. Indirectly, the register still selects a memory segment, but not directly as in real mode.

DESCRIPTOR TABLE

Two types of Descriptor table selected by the segment register

*** global *local**

Global descriptors contain segment definitions that apply to all programs. a global descriptor might be called a system descriptor. Local descriptors are usually unique to an application. Local descriptor an application descriptor. global and local descriptor tables are a maximum of 64K bytes in length

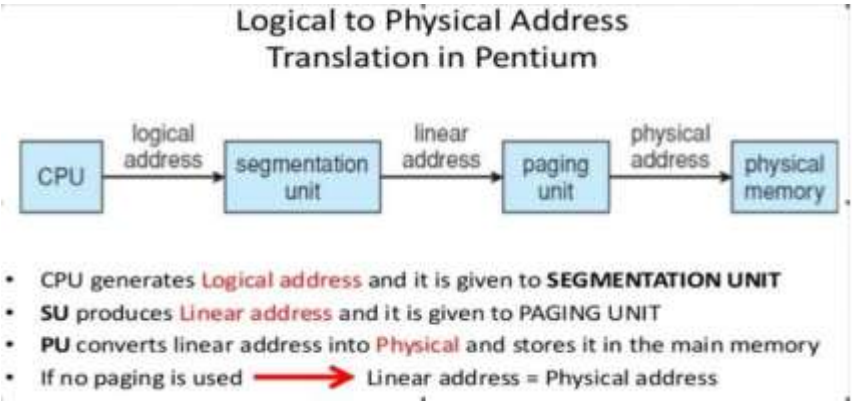


Fig 1.16: logical to physical address translation

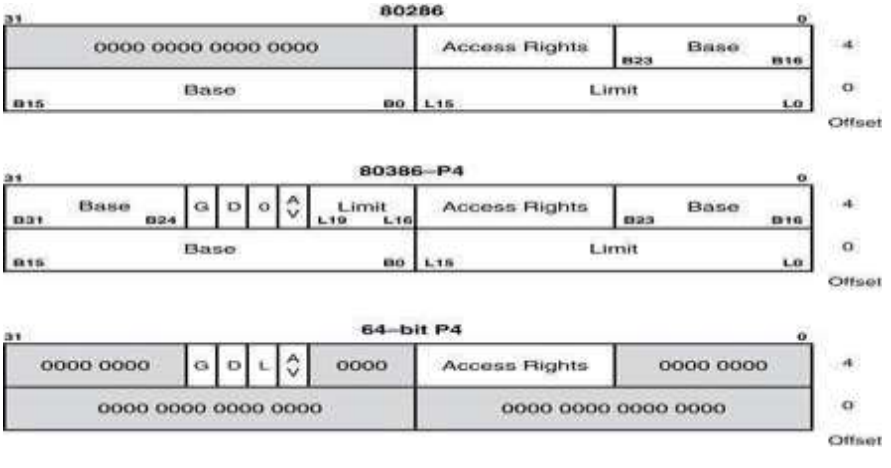


Fig 1.17 Descriptor format

The base address portion of the descriptor indicates the starting location of the memory segment.The segment limit contains the last offset address found in a segment. For example, if a segment begins at memory location F00000H and ends at location F000FFH, the base address is F00000H and the limit is FFH. D=0 real mode, D=1 protected mode.The AV bit, in the 80386 and above descriptor, is used by some operating systems to indicate that the segment is available (AV = 1) or not available (AV = 0).
Access right controls access to the protected mode memory segment

Explanation

- **each descriptor is 8 bytes in length. The base address of the descriptor indicates the starting location of the memory segment.The G, or granularity bit allows a segment length of 4K to 4G bytes in steps of 4K bytes.(granularity bit. If G=0, the limit specifies a segment limit of 00000H to FFFFH. If G = 1, the value of the limit is multiplied by 4K bytes (appended with XXXH). The limit is then 00000XXXH to FFFFFXXXH, if G=1). 32-bit offset address allows segment lengths of 4G bytes16-bit offset address allows segment lengths of 64Kbytes.**

The access rights byte controls access to theprotected mode segment. describes segment function in the system and allows complete control over the segment. if the segment is a data segment, the direction of growth is specified. If the segment grows beyond its limit, the operating system is interrupted indicating a general protection fault.

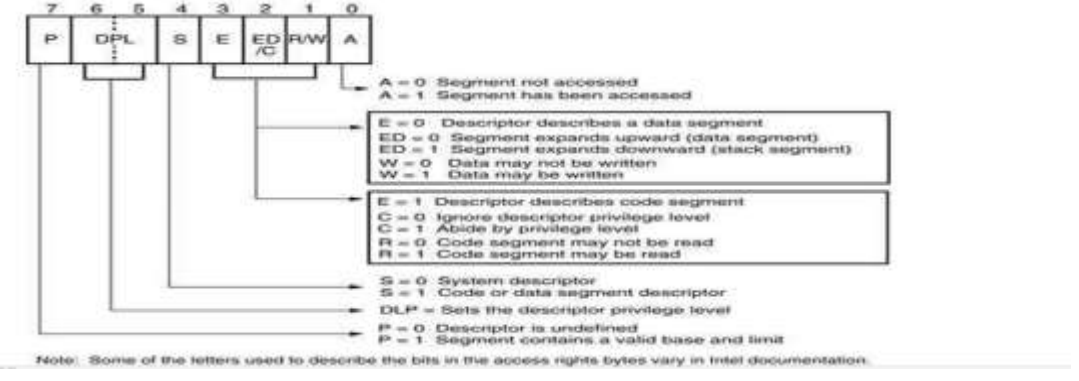


Fig 1.18 The access rights byte forthe 80286 descriptor

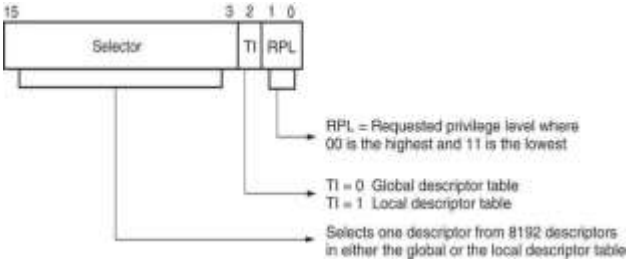


Fig 1.19. The contents of a segment register during protected mode operation of the 80286 microprocessors.

Descriptors are chosen from the descriptor table by the segment register.The segment register contains a 13-bit selector field, a table selector bit, and a requested privilege level field. The 13-bit selector chooses one of the 8192 descriptors from the descriptortable. The TI bit selects either the global descriptor table or the local descriptor table. The requested privilege level (RPL) requests the access privilege level of a memory

segment. The highest privilege level is 00 and the lowest is 11. For example, if the requested privilege level is 10 and the access rights byte sets the segment privilege level at 11, access is granted because 10 is higher in priority than privilege level 11. Privilege levels are used in multiuser environments.
Example

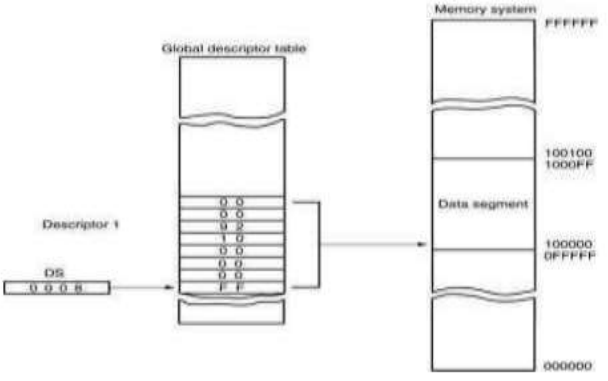


Fig 1.20 Selector operation

DS the segment register, containing a selector, chooses a descriptor from the global descriptor table. The entry in the global descriptor table selects a segment in the memory system.In this illustration, DS contains 0008H, which accesses the descriptor number 1 from the global descriptor table by using a requested privilege level of 00.Descriptor number 1 contains a descriptor that defines the base address as 00100000H with a segment limit of 000FFH.This means that a value of 0008H loaded into DS causes the microprocessor to use memory locations 00100000H—001000FFH for the data segment. Descriptor zero is called the null descriptor and may not be used for accessing memory. the segment register, containing a selector. chooses a descriptor from the global descriptor table. The entry in the global descriptor table selects a segment in the memory system. Descriptor zero is called the null descriptor, must contain all zeros, and may not be used for accessing memory.

Fig 1.21 Segment registers, Descriptor cache and Descriptor table addresses

Difference between Real and Protected Mode

Real Mode	Protected Mode (PVAM)
Memory addressing up to 1 MB physical memory	Memory addressing up to 16 MB of physical memory
No virtual memory support	Supports up tp to 64TB of virtual memory
Memory Protection mechanism is not available	Memory Protection Mechanism is avilable
Does not support virtual address space	Gives virtual and physical address space
Does not support LDT and GDT	Supports LDT and GDT
Segment descriptor cache is not available	Segment descriptor cache is available
Supports Segmentation	Supports segmentation and paging.

Fig 1.22 Difference between Real and Protected mode

Memory Paging

In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. The Memory management unit consists of Segmentation unit and Paging unit. Segmentation unit allows segments of size 4Gbytes atmax. The Paging unit organizes the physical memory in terms of pages of 4kbytes size each. Mechanism available in the 80386 andup Allows a linear address (program generated) of a program to be located in any portion of physical memory. Paging unit works under the control of the segmentation unit, i.e.each segment is further divided intopages. The memory paging mechanism located within the 80386 and above allows any physical memory location to be assigned to any linear address. The linear address is defined as the address generated by aprogram. The physical address is the actual memory location accessed by a program. With the memory paging unit, the linear address is invisibly translated to any physical address, which allows an application written to function at a specific address to be relocated through the paging mechanism.

PAGING ADVANTAGES

Complete segment of a task need not be in the physical memory at any time Only a few pages of the segment, which are required currently for execution need to be available in the physical memory. Memory requirement of the task is substantially reduced, relinquishing the available memory for othertasks.

PAGING DIRECTORY

Page directory contains the location of up to 1024 page translation table, which are each 4 bytes long. Each page translation table translates a logical address into a physical address. The page directory is stored in the memory and accessed by the page descriptor address register (CR3).Control register CR3 holds the base address of the page directory, which starts at any 4K byte boundary in the memory system. Each entry in the page directory translates the leftmost 10bits of the memory address. This 10bit portion of the linear address is used to locate different page tables for different page tableentries

PAGE TABLE ENTRY

The page table entry contain the starting address of the page and the statistical information about the page. Total entries are 1024. Each page table entry of 4 byte

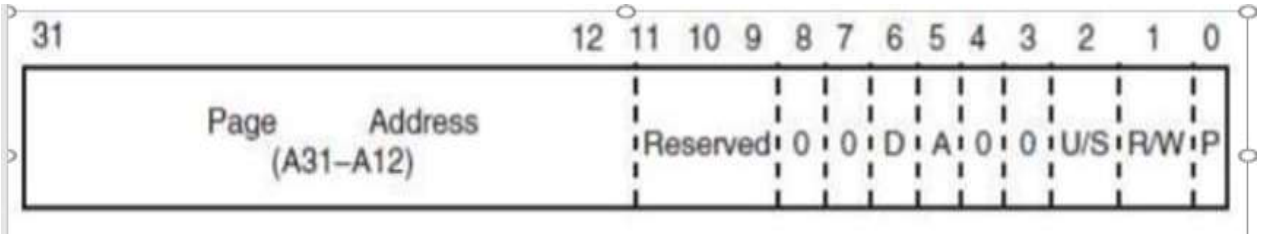


Fig 1.23 Page table entry

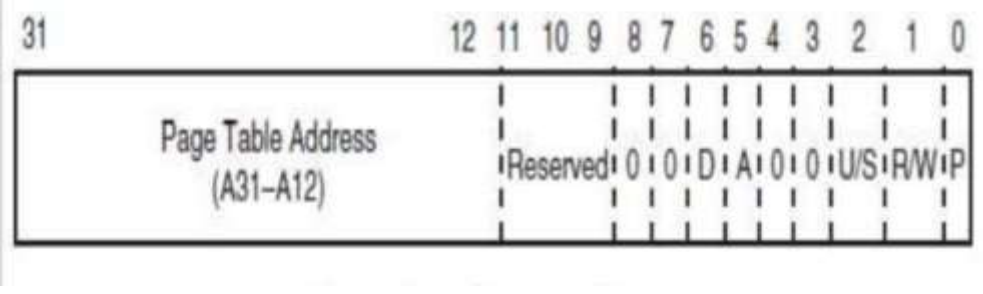


Fig 1.24 Page directory entry

Difference between page directory and page table entry

The main difference is that the page directory entry contains the physical address of a page table, while the page table entry contains the physical address of a 4K-byte physical page of memory The other difference is the D (dirty bit), which has no function in the page directory entry, bit indicates that a page has been written to in a page table entry

Page translation mechanism

A page frame is 4K byte unit of contiguous addresses of physical memory. Pages begin on byte boundaries and are fixed in size. A linear address refers indirectly to a physical address by specifying page table , a page within that table and an offset within that page

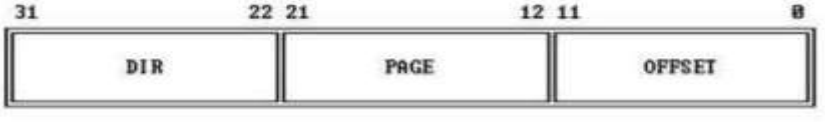


Fig 1.25 Linear address format

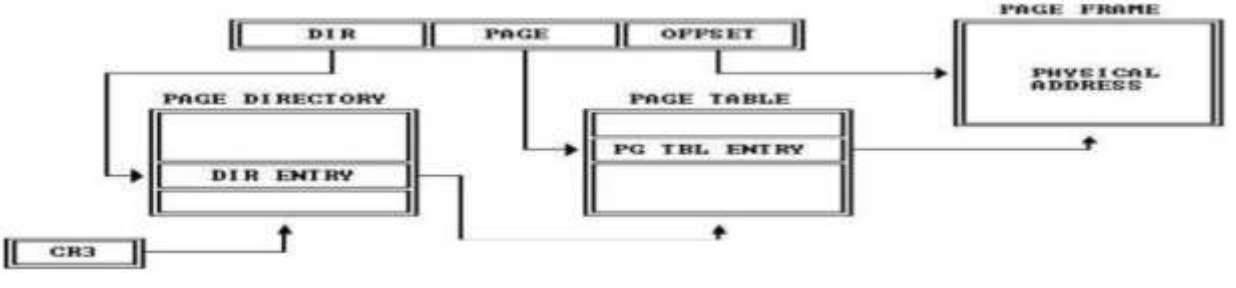


Fig 1.26: Page translation mechanism

Fig shows how processor converts the DIR, PAGE & OFFSET of a linear address into the physical address by consulting two levels of page tables. The addressing mechanism uses the DIR field as an index into a

page directory, uses the PAGE field as an index into the page table determined by the page directory, and uses the OFFSET field to address a byte within the page determined by the page table. The second phase of address transformation , linear to physical, page translation is in effect only when PG bit of CR0 is set.

Control Register

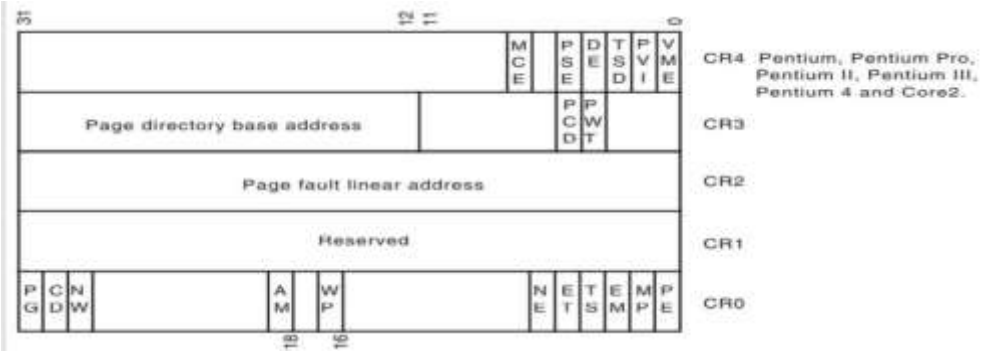


Fig 1.27: Control register structure

Paging Registers

It is enabled by setting the PG bit to 1 (left most bit in CR0). (If set to 0, linear addresses are physical addresses). CR3 contains the page directory "physical" base address. Each directory entry contain 1024 directory entries of 4 byte each. Each page directory entry addresses a page table that contains up to 1024 entries.

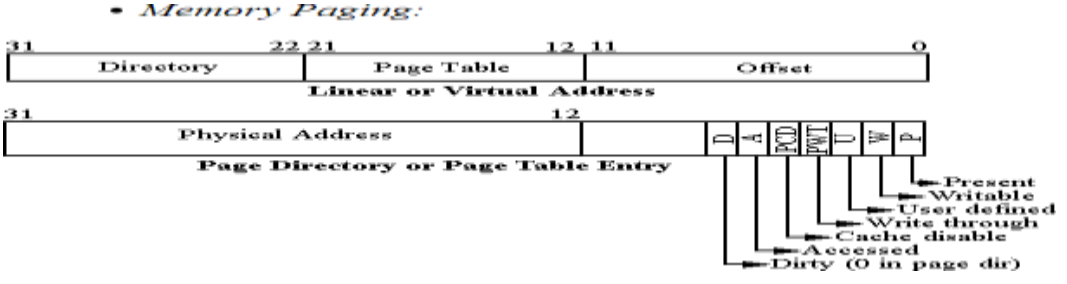


Fig 1.28 Memory paging

The linear address, as it is generated by the software, is broken into three sections that are used to access the page directory entry, page table entry, and memory page offset address. if the page table entry 0 contains address 00100000H, then the physical address is 00100000H- 00100FFFH for linear address 00000000H–00000FFFH. (4k byte of address range-000-FFF). The virtual address is broken into three pieces.

Directory : Each page directory addresses a 4MB section of main mem.

Page Table : Each page table entry addresses a 4KB section of main mem.

Offset : Specifies the byte in the page.

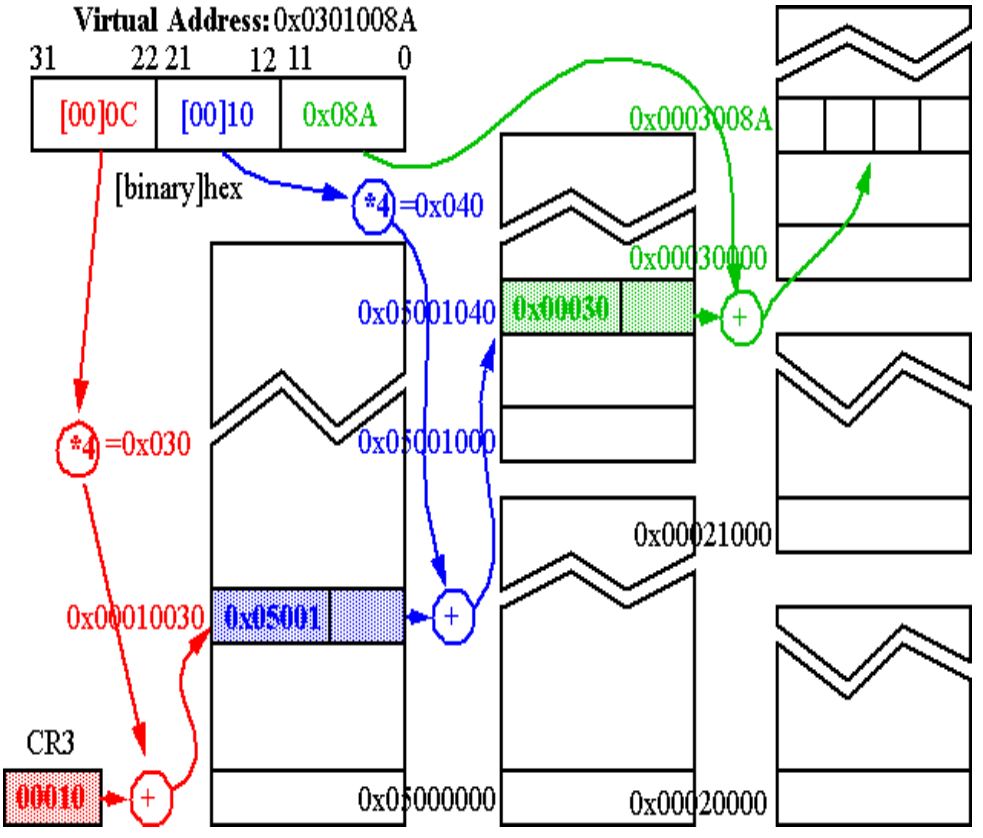


Fig 1.29 Memory Addressing

Block diagram of 8086

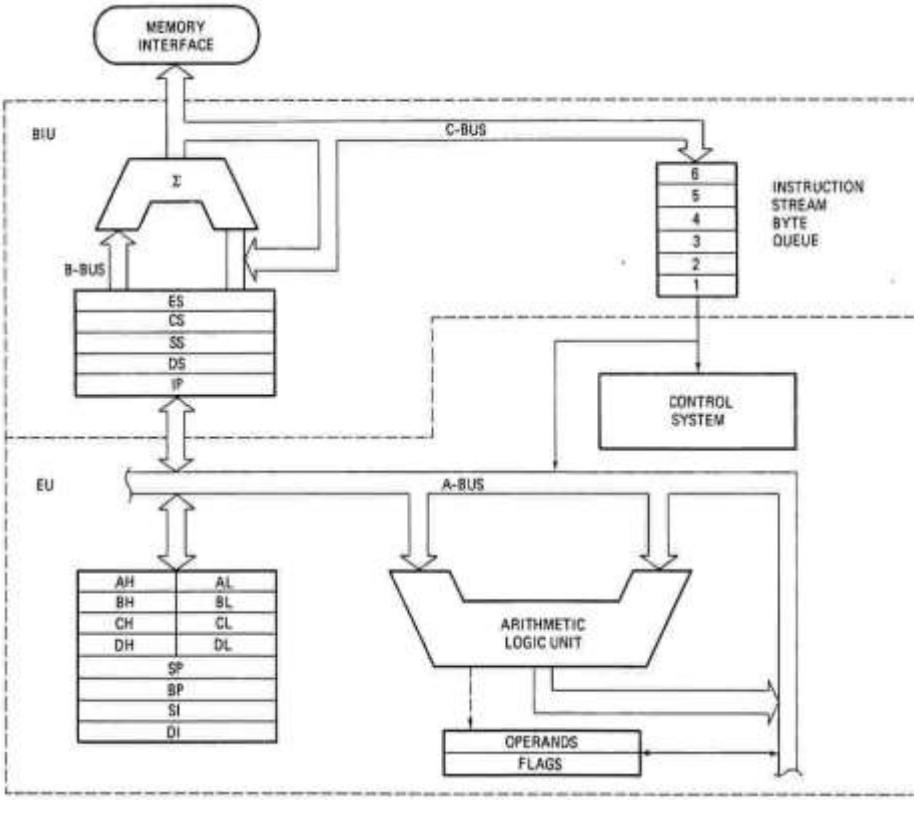


Fig 1.30 Block diagram of 8086 microprocessor

Software model of 8086

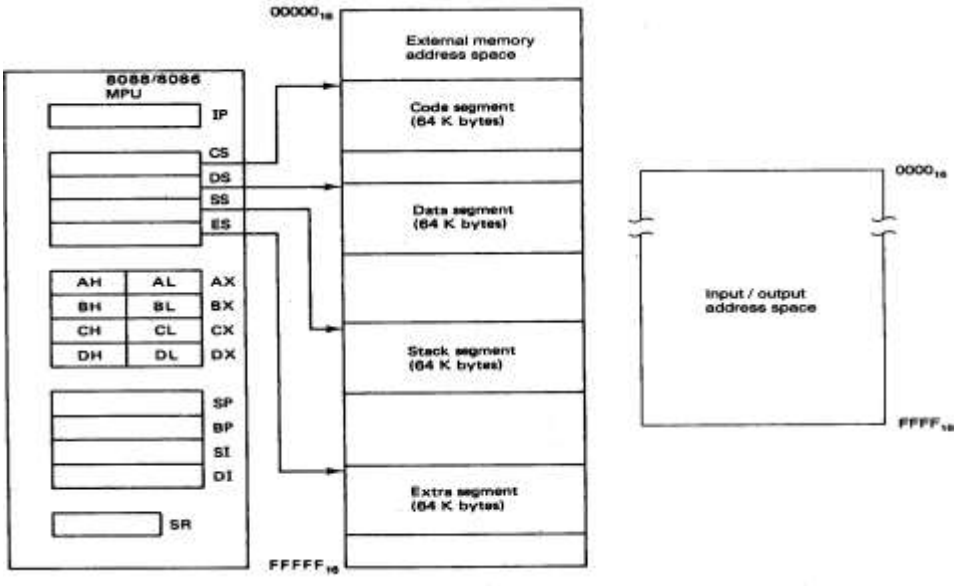


Fig 1.31 Software model-8086

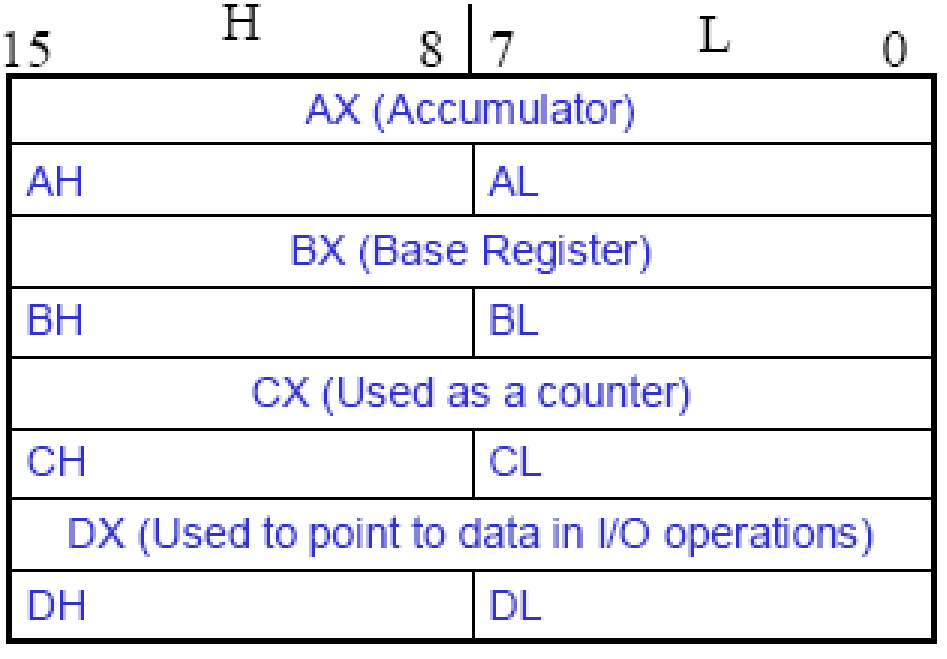


Fig 1.32: General purpose registers

AX - the Accumulator BX - the Base Register CX - the Count Register DX - the Data Register
Normally used for storing temporary results. Each of the registers is 16 bits wide (AX, BX, CX, DX). Can be accessed as either 16 or 8 bits AX, AH, AL

AX-Accumulator Register. Preferred register to use in arithmetic, logic and data transfer instructions because it generates the shortest Machine Language Code. Must be used in multiplication and division operations.Must also be used in I/O operations.

BX-Base Register.Also serves as an address register

CX- Count register. Used as a loop counter. Used in shift and rotate operations

DX- Data register. Used in multiplication and division. Also used in I/O operations

Pointer and Index Registers

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index
IP	Instruction Pointer

Fig 1.33 Pointers and index registers

- **All 16 bits wide, L/H bytes are not accessible. Used as memory pointers**
- **Example: MOV AH, [SI]**
- ***Move the byte stored in memory location whose address is contained in register SI to register AH*. IP is not under direct control of the programmer**

The Stack

The stack is used for temporary storage of information such as data or addresses. When a CALL is executed, the 8086 automatically PUSH es the current value of CS and IP onto the stack. Other registers can also be pushed. Before return from the subroutine, POP instructions can be used to pop values back from the stack into the corresponding registers.

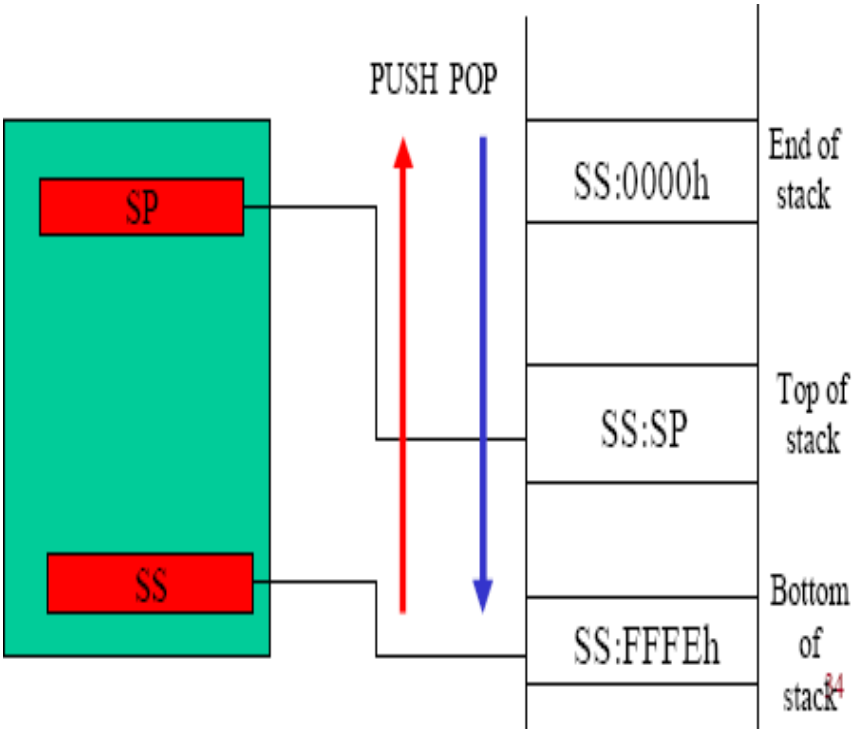


Fig 1.35 stack operation

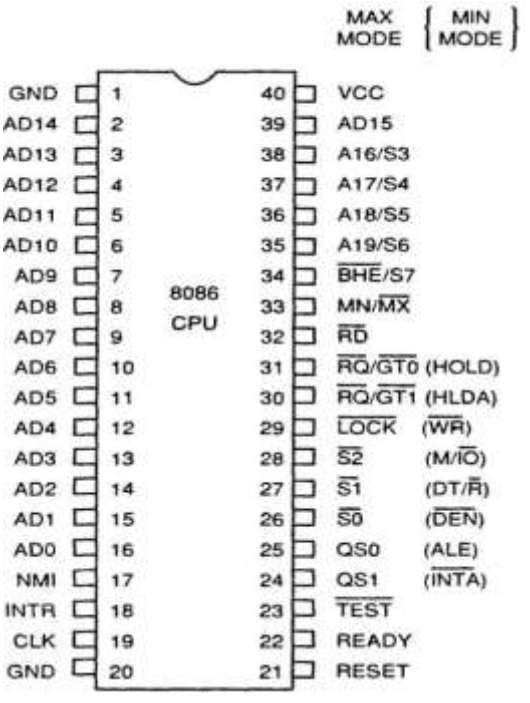


Fig.1.36 Pin Diagram

Minimum Mode 8086 System

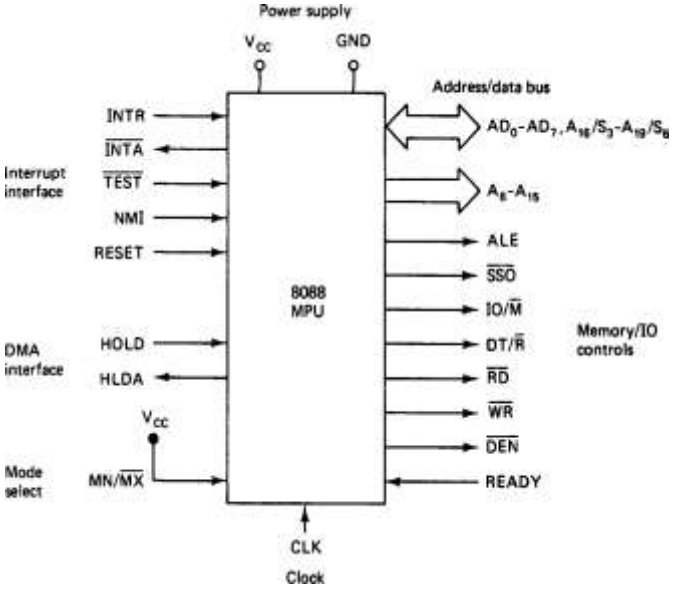


Fig 1.37. Minimum mode signals

Maximum mode signals

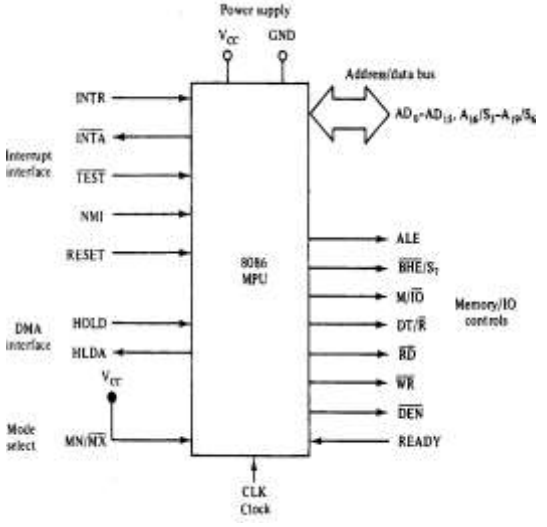


Fig 1.38. Maximum mode signals

Test signals in 8086

TEST is an input pin and is only used by the wait instruction .the 8086 enter a wait state after execution of the wait instruction until a low is Seen on the test pin. Used in conjunction with the WAIT instruction in multiprocessing environments. This is input from the 8087 coprocessor. During execution of a wait instruction, the CPU checks this signal. If it is low, execution of the signal will continue; if not, it will stop executing.

Coprocessor Execution

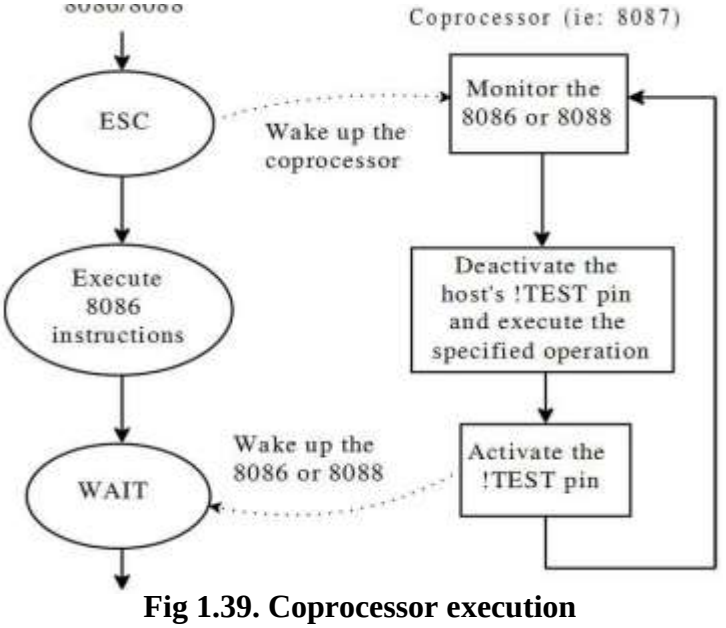


Fig 1.39. Coprocessor execution

Multiprocessor configuration Advantages

High system throughput can be achieved by having more than one CPU. The system can be expanded in modular form. Each bus master module is an independent unit and normally resides on a separate PC board. One can be added or removed without affecting the others in the system. A failure in one module normally does not affect the breakdown of the entire system and the faulty module can be easily detected and replaced. Each bus master has its own local bus to access dedicated memory or IO devices. So a greater degree of parallel processing can be achieved.

ADDRESSING MODES

8086/8088 provide a seven Addressing Modes:

- **Register Addressing**
- **Immediate Addressing**
- **Direct Addressing**
- **Register Indirect Addressing**
- **Base–Plus–Index Addressing**
- **Register Relative Addressing**
- **Base Relative–Plus–Index Addressing**

Register Addressing Mode

Assembly Language	*Size*	*Operation*
MOV AL,BL	8-bits	Copies BL into AL
MOV CH,CL	8-bits	Copies CL into CH
MOV AX,CX	16-bits	Copies CX into AX
MOV SP,BP	16-bits	Copies BP into SP
MOV DS,AX	16-bits	Copies AX into DS
MOV SI,DI	16-bits	Copies DI into SI
MOV BX,ES	16-bits	Copies ES into BX
MOV ECX,EBX	32-bits	Copies EBX into ECX
MOV ESP,EDX	32-bits	Copies EDX into ESP
MO / ES,DS	—	Not allowed (segment-to-segment)
MOV BL,DX	—	Not allowed (mixed sizes)
MOV CS,AX	—	Not allowed (the code segment register may not be the destination register)

Fig1.40 Example of Register addressing mode

Immediate Addressing Mode

Assembly Language	*Size*	*Operation*
MOV BL,44	8-bits	Copies a 44 decimal (2CH) into BL
MOV AX,44H	16-bits	Copies a 0044H into AX
MOV SI,0	16-bits	Copies a 0000H into SI
MOV CH,100	8-bits	Copies a 100 decimal (64H) into CH
MOV AL,'A'	8-bits	Copies an ASCII A into AL
MOV AX,'AB'	16-bits	Copies an ASCII BA* into AX
MOV CL,11001110B	8-bits	Copies a 11001110 binary into CL
MOV EBX,12340000H	32-bits	Copies a 12340000H into EBX
MOV ESI,12	32-bits	Copies a 12 decimal into ESI
MOV EAX,100Y	32-bits	Copies a 100 binary into EAX

Fig1.41 Example of Immediate addressing mode

Direct Addressing mode

Assembly Language	*Size*	*Operation*
MOV AL,NUMBER	8-bits	Copies the byte contents of data segment memory location NUMBER into AL
MOV AX,COW	16-bits	Copies the word contents of data segment memory location COW into AX
MOV EAX,WATER*	32-bits	Copies the doubleword contents of memory location WATER into EAX
MOV NEWS,AL	8-bits	Copies AL into data segment memory location NEWS
MOV THERE,AX	16-bits	Copies AX into data segment memory location THERE
MOV HOME,EAX*	32-bits	Copies EAX into data segment memory location HOME

Fig 1.42: Example of Direct addressing mode

Register Indirect addressing mode

Assembly Language	*Size*	*Operation*
MOV CX,[BX]	16-bits	Copies the word contents of the data segment memory location address by BX into CX
MOV [BP],DL*	8-bits	Copies DL into the stack segment memory location addressed by BP
MOV [DI],BH	8-bits	Copies BH into the data segment memory location addressed by DI
MOV [DI],[BX]	—	Memory-to-memory moves are not allowed except with string instructions
MOV AL,[EDX]	8-bits	Copies the byte contents of the data segment memory location addressed by EDX into AL
MOV ECX,[EBX]	32-bits	Copies the doubleword contents of the data segment memory location addressed by EBX into ECX

Fig1.43: Example of Register Indirect addressing mode

Base-Plus-Index Addressing Mode

Assembly Language	*Size*	*Operation*
MOV CX,[BX+DI]	16-bits	Copies the word contents of the data segment memory location address by BX plus DI into CX
MOV CH,[BP+SI]	8-bits	Copies the byte contents of the stack segment memory location addressed by BP plus SI into CH
MOV [BX+SI],SP	16-bits	Copies SP into the data segment memory location addresses by BX plus SI
MOV [BP+DI],AH	8-bits	Copies AH into the stack segment memory location addressed by BP plus DI
MOV CL,[EDX+EDI]	8-bits	Copies the byte contents of the data segment memory location addressed by EDX plus EDI into CL
MOV [EAX+EBX],ECX	32-bits	Copies ECX into the data segment memory location addressed by EAX plus EBX

Fig 1.44 Example Base-Plus-Index Addressing mode

Register Relative Addressing Mode

Assembly Language	*Size*	*Operation*
MOV AX,[DI+100H]	16-bits	Copies the word contents of the data segment memory location addressed by DI plus 100H into AX
MOV ARRAY[SI],BL	8-bits	Copies BL into the data segment memory location addressed by ARRAY plus SI
MOV LIST[SI+2],CL	8-bits	Copies CL into the data segment memory location addressed by sum of LIST, SI, and 2
MOV DI,SET_IT[BX]	16-bits	Copies the word contents of the data segment memory location addressed by the sum of SET_IT and BX into DI
MOV DI,[EAX+10H]	16-bits	Copies the word contents of the data segment memory location addressed by the sum of EAX and 10H into DI
MOV ARRAY[EBX],EAX	32-bits	Moves EAX into the data segment memory location addressed by the sum of ARRAY and EBX

Fig 1.45 Example Register Relative addressing mode

Base Relative-Plus-Index Addressing mode

Assembly Language	*Size*	*Operation*
MOV DH,[BX+DI+20H]	8-bits	Copies the byte contents of the data segment memory location addressed by the sum of BX, DI, and 20H into DH
MOV AX,FILE[BX+DI]	16-bits	Copies the word contents of the data segment memory location addressed by the sum of FILE, BX, and DI into AX
MOV LIST[BP+DI],CL	8-bits	Copies CL into the stack segment memory location addressed by the sum of LIST, BP, and DI
MOV LIST[BP+SI+4],DH	8-bits	Copies DH into the stack segment memory location addressed by the sum of LIST, BP, SI, and 4
MOV EAX,FILE[EBX+ECX+2]	32-bits	Copies the doubleword contents of the data segment memory location addressed by the sum of FILE, EBX, ECX, and 2 into EAX

Fig 1.46 Example Base relative plus Index addressing mode

PROGRAM MEMORY ADDRESSING MODE

The Program Memory Addressing mode is used in branch instructions. These branch instructions are instructions which are responsible for changing the regular flow of the instruction execution and shifting the control to some other location. In 8086 microprocessor, these instructions are usually JMP and CALLinstructions.

three distinct forms:

Direct Program MemoryAddressing
Indirect Program Memory Addressing
Relative Program Memory Addressing

DIRECT PROGRAM MEMORY ADDRESSING

- **In this addressing mode, the offset address where the control is to be shifted is defined within the instruction. This mode is called direct addressing mode because the required address is directly present in the instruction rather than being stored in some register.**

- **Example:**
- **JMP 4032H**
- **Here, the working of the above instruction will be as follows:**
- **The current value of IP which holds the address of next instruction to be executed will be stored in the TOP OF THE STACK.**
- **Now, the IP will be replaced by the mentioned value, i.e. IP- 4032H**
- **Now, the Memory address is calculated as:**
- **(Contents of CS) X 10H + (contents of IP)**

INDIRECT PROGRAM MEMORY ADDRESSING

- **As the name suggests, in this addressing mode, the offset address is not present directly in the instruction. It is rather stored in any of the CPU registers (Internal Register). So, the contents of the Instruction Pointer (IP) will be replaced by the contents of that register.**
- **Example: JMP BX**
- **Working:**
- **Suppose that the content of the BX register is 0003H. So, the working of the microprocessor for executing the above instruction will be as follows:**
- **IP = contents of BX**
- **i.e. IP = 0003H**
- **And the required memory address is calculated in a similar way as in Direct Addressing mode: (Contents of CS) X 10H + (contents of IP)**

RELATIVE PROGRAM MEMORY ADDRESSING

- **In this Addressing mode, the offset address is equal to the content of the Instruction Pointer (IP) plus the 8 or 16-bit displacement. For the 8 bit displacement, SHORT is used and for 16-bit displacement, LONG is used. This type of displacement will only be intra-segment, i.e. within the segment.**

- **Example:**

- **JMP SHORTOVER**
- **Here, SHORT is used to represent the 8-bit displacement and OVER is the Label defined for any particular memory location.**

STACK-MEMORY ADDRESSING MODES IN MICROPROCESSOR 8086/8088

The stack plays an important role in all microprocessors. It holds data temporarily and stores return addresses for procedures. The stack memory is a LIFO (last-in, first-out) memory, which describes the way that data are stored and removed from the stack. Data are placed onto the stack with a PUSH instruction and removed with a POP instruction. The CALL instruction also uses the stack to hold the return address for procedures and a RET (return) instruction to remove the return address from the stack.

8086 INSTRUCTION SET
DATA TRANSFER INSTRUCTIONS

MOV – MOV Destination, Source

The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location or an immediate number. The source and destination cannot both be memory locations. They must both be of the same type (bytes or words). MOV instruction does not affect any flag.

MOV CX, 037AH Put immediate number 037AH to CX
MOV BL, [437AH] Copy byte in DS at offset 437AH to BL
MOV AX, BX Copy content of register BX to AX
MOV DL, [BX] Copy byte from memory at [BX] to DL
MOV DS, BX Copy word from BX to DS register
MOV RESULT [BP], AX Copy AX to two memory locations;
AL to the first location, AH to the second;
EA of the first memory location is sum of the displacement represented by RESULTS and content of BP.
Physical address = EA + SS.
MOV ES: RESULTS [BP], AX Same as the above instruction, but physical address = EA + ES, because of the segment override prefix ES

XCHG – XCHG Destination, Source

The XCHG instruction exchanges the content of a register with the content of another register or with the content of memory location(s). It cannot directly exchange the content of two memory locations. The source and destination must both be of the same type (bytes or words). The segment registers cannot be used in this instruction. This instruction does not affect any flag.

XCHG AX, DX Exchange word in AX with word in DX
XCHG BL, CH Exchange byte in BL with byte in CH
XCHG AL, PRICES [BX] Exchange byte in AL with byte in memory at
EA = PRICE [BX] in DS.

LEA – LEA Register, Source

This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register. LEA does not affect any flag.

LEA BX, PRICES Load BX with offset of PRICE in DS
LEA BP, SS: STACK_TOP Load BP with offset of STACK_TOP in SS
LEA CX, [BX][DI] Load CX with EA = [BX] + [DI]

LDS – LDS Register, Memory address of the first word

This instruction loads new values into the specified register and into the DS register from four successive memory locations. The word from two memory locations is copied into the specified register and the

word from the next two memory locations is copied into the DS registers. LDS does not affect any flag.

LDS BX, [4326] Copy content of memory at displacement 4326H in DS to BL, content of 4327H to BH. Copy content at displacement of 4328H and 4329H in DS to DS register.
LDS SI, SPTR Copy content of memory at displacement SPTR and SPTR + 1

in DS to SI register. Copy content of memory at displacements SPTR + 2 and SPTR + 3 in DS to DS register. DS: SI now points at start of the desired string.

LES – LES Register, Memory address of the first word

This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory locations is copied into the specified register, and the word from the next two memory locations is copied into the ES register. LES does not affect any flag.

LES BX, [789AH] Copy content of memory at displacement 789AH in DS to BL, content of 789BH to BH, content of memory at displacement 789CH and 789DH in DS is copied to ES register.
LES DI, [BX] Copy content of memory at offset [BX] and offset [BX] + 1 in DS to DI register. Copy content of memory at offset [BX] + 2 and [BX] + 3 to ES register.

ARITHMETIC INSTRUCTIONS

ADD – ADD Destination, Source ADC – ADC Destination, Source

These instructions add a number from some source to a number in some destination and put the result in the specified destination. The ADC also adds the status of the carry flag to the result. The source may be an immediate number, a register, or a memory location. The destination may be a register or a memory location. The source and the destination in an instruction cannot both be memory locations. The source and the destination must be of the same type (bytes or words). If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with 0's before adding. Flags affected: AF, CF, OF, SF, ZF.

ADD AL, 74H Add immediate number 74H to content of AL. Result in AL
ADC CL, BL Add content of BL plus carry status to content of CL
ADD DX, BX Add content of BX to content of DX
ADD DX, [SI] Add word from memory at offset [SI] in DS to content of DX
ADC AL, PRICES [BX] Add byte from effective address PRICES [BX] plus carry status to content of AL
ADD AL, PRICES [BX] Add content of memory at effective address PRICES [BX] to AL

SUB – SUB Destination, Source SBB – SBB Destination, Source

These instructions subtract the number in some source from the number in some destination and put the result in the destination. The SBB instruction also subtracts the content of carry flag from the destination. The source may be an immediate number, a register or memory location. The destination

can also be a register or a memory location. However, the source and the destination cannot both be memory location. The source and the destination must both be of the same type (bytes or words). If you want to subtract a byte from a word, you must first move the byte to a word location such as a 16-bit register and fill the upper byte of the word with 0's. Flags affected: AF, CF, OF, PF, SF, ZF.

SUB CX, BX CX – BX; Result in CX
SBB CH, AL Subtract content of AL and content of CF from content of CH. Result in CH
SUB AX, 3427H Subtract immediate number 3427H from AX
SBB BX, [3427H] Subtract word at displacement 3427H in DS and content of CF from BX
SUB PRICES [BX], 04H Subtract 04 from byte at effective address PRICES [BX],
if PRICES is declared with DB; Subtract 04 from word at effective address PRICES [BX], if it is declared with DW.
SBB CX, TABLE [BX] Subtract word from effective address TABLE [BX]
and status of CF from CX.
SBB TABLE [BX], CX Subtract CX and status of CF from word in memory at
effective address TABLE[BX].

MUL – MUL Source

This instruction multiplies an unsigned byte in some source with an unsigned byte in AL register or an unsigned word in some source with an unsigned word in AX register. The source can be a register or a memory location. When a byte is multiplied by the content of AL, the result (product) is put in AX. When a word is multiplied by the content of AX, the result is put in DX and AX registers. If the most significant byte of a 16-bit result or the most significant word of a 32-bit result is 0, CF and OF will both be 0's. AF, PF, SF and ZF are undefined after a MUL instruction.

If you want to multiply a byte with a word, you must first move the byte to a word location such as an extended register and fill the upper byte of the word with all 0's. You cannot use the CBW instruction for this, because the CBW instruction fills the upper byte with copies of the most significant bit of the lower byte.

MUL BH Multiply AL with BH; result in AX
MUL CX Multiply AX with CX; result high word in DX, low word in AX
MUL BYTE PTR [BX] Multiply AL with byte in DS pointed to by [BX]
MUL FACTOR [BX] Multiply AL with byte at effective address FACTOR [BX], if it
is declared as type byte with DB. Multiply AX with word at effective address FACTOR [BX], if it is declared as type word with DW.
MOV AX, MCAND_16 Load 16-bit multiplicand into AX MOV CL, MPLIER_8 Load 8-bit multiplier into CL
MOV CH, 00H Set upper byte of CX to all 0's
MUL CX AX times CX; 32-bit result in DX and AX

IMUL – IMUL Source

This instruction multiplies a signed byte from source with a signed byte in AL or a signed word from some source with a signed word in AX. The source can be a register or a memory location. When a byte from source is multiplied with content of AL, the signed result (product) will be put in AX. When a word from source is multiplied by AX, the result is put in DX and AX. If the magnitude of the product does not require all the bits of the destination, the unused byte / word will be filled with copies of the sign bit. If the upper byte of a 16-bit result or the upper word of a 32-bit result contains only copies of the sign bit (all 0's or all 1's), then CF and the OF will both be 0; If it contains a part of the product, CF and OF

will both be 1. AF, PF, SF and ZF are undefined after IMUL.

If you want to multiply a signed byte with a signed word, you must first move the byte into a word location and fill the upper byte of the word with copies of the sign bit. If you move the byte into AL, you can use the CBW instruction to do this.

IMUL BH Multiply signed byte in AL with signed byte in BH; result in AX.
IMUL AX Multiply AX times AX; result in DX and AX
MOV CX, MULTIPLIER Load signed word in CX MOV AL, MULTIPLICAND Load signed byte in AL CBW Extend sign of AL into AH
IMUL CX Multiply CX with AX; Result in DX and AX

DIV – DIV Source

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word (32 bits) by a word. When a word is divided by a byte, the word must be in the AX register. The divisor can be in a register or a memory location. After the division, AL will contain the 8-bit quotient, and AH will contain the 8-bit remainder. When a double word is divided by a word, the most significant word of the double word must be in DX, and the least significant word of the double word must be in AX. After the division, AX will contain the 16-bit quotient and DX will contain the 16-bit remainder. If an attempt is made to divide by 0 or if the quotient is too large to fit in the destination (greater than FFH / FFFFH), the 8086 will generate a type 0 interrupt. All flags are undefined after a DIV instruction.

If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0's. Likewise, if you want to divide a word by another word, then put the dividend word in AX and fill DX with all 0's.

DIV BL Divide word in AX by byte in BL; Quotient in AL, remainder in AH
DIV CX Divide down word in DX and AX by word in CX; Quotient in AX, and remainder in DX.
DIV SCALE [BX] AX / (byte at effective address SCALE [BX]) if SCALE [BX] is of type byte; or (DX and AX) / (word at effective address SCALE[BX] if SCALE[BX] is of type word

IDIV – IDIV Source

This instruction is used to divide a signed word by a signed byte, or to divide a signed double word by a signed word.

When dividing a signed word by a signed byte, the word must be in the AX register. The divisor can be in an 8-bit register or a memory location. After the division, AL will contain the signed quotient, and AH will contain the signed remainder. The sign of the remainder will be the same as the sign of the dividend. If an attempt is made to divide by 0, the quotient is greater than 127 (7FH) or less than –127 (81H), the 8086 will automatically generate a type 0 interrupt.

When dividing a signed double word by a signed word, the most significant word of the dividend (numerator) must be in the DX register, and the least significant word of the dividend must be in the AX register. The divisor can be in any other 16-bit register or memory location. After the division, AX will contain a signed 16-bit quotient, and DX will contain a signed 16-bit remainder. The sign of the remainder will be the same as the sign of the dividend. Again, if an attempt is made to divide by 0, the quotient is greater than +32,767 (7FFFH) or less than –32,767 (8001H), the 8086 will automatically generate a type 0 interrupt.

All flags are undefined after an IDIV.

If you want to divide a signed byte by a signed byte, you must first put the dividend byte in AL and sign-extend AL into AH. The CBW instruction can be used for this purpose. Likewise, if you want to divide a signed word by a signed word, you must put the dividend word in AX and extend the sign of AX to all the bits of DX. The CWD instruction can be used for this purpose.

IDIV BL Signed word in AX/signed byte in BL
IDIV BP Signed double word in DX and AX/signed word in BP
IDIV BYTE PTR [BX] AX / byte at offset [BX] in DS

INC – INC Destination

The INC instruction adds 1 to a specified register or to a memory location. AF, OF, PF, SF, and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result will be all 0's with no carry.

INC BLAdd 1 to contains of BL register
INC CX Add 1 to contains of CX register
INC BYTE PTR [BX]Increment byte in data segment at offset contained in BX.
INC WORD PTR [BX] Increment the word at offset of [BX] and [BX + 1] in the data segment.
INC TEMP Increment byte or word named TEMP in the data segment. Increment byte if MAX_TEMP declared with DB. Increment word if MAX_TEMP is declared with DW.
INC PRICES [BX] Increment element pointed to by [BX] in array PRICES. Increment a word if PRICES is declared as an array of words; Increment a byte if PRICES is declared as an array of bytes.

DEC – DEC Destination

This instruction subtracts 1 from the destination word or byte. The destination can be a register or a memory location. AF, OF, SF, PF, and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing 00H or a 16-bit destination containing 0000H is ecremented, the result will be FFH or FFFFH with no carry (borrow).

DEC CL Subtract 1 from content of CL register
DEC BP Subtract 1 from content of BP register
DEC BYTE PTR [BX] Subtract 1 from byte at offset [BX] in DS.
DEC WORD PTR [BP] Subtract 1 from a word at offset [BP] in SS.
DEC COUNT Subtract 1 from byte or word named COUNT in DS. Decrement a byte if COUNT is declared with a DB; Decrement a word if COUNT is declared with a DW.

DAA (DECIMAL ADJUST AFTER BCD ADDITION)

This instruction is used to make sure the result of adding two packed BCD numbers is adjusted to be a legal BCD number. The result of the addition must be in AL for DAA to work correctly. If the lower nibble in AL after an addition is greater than 9 or AF was set by the addition, then the DAA instruction will add 6 to the lower nibble in AL. If the result in the upper nibble of AL in now greater than 9 or if the carry flag was set by the addition or correction, then the DAA instruction will add 60H to AL.

Let AL = 59 BCD, and BL = 35 BCD
ADD AL, BL AL = 8EH; lower nibble > 9, add 06H to AL
DAA AL = 94 BCD, CF = 0
Let AL = 88 BCD, and BL = 49 BCD
ADD AL, BL AL = D1H; AF = 1, add 06H to AL
DAA AL = D7H; upper nibble > 9, add 60H to AL AL = 37 BCD, CF = 1

The DAA instruction updates AF, CF, SF, PF, and ZF; but OF is undefined.

DAS (DECIMAL ADJUST AFTER BCD SUBTRACTION)

This instruction is used after subtracting one packed BCD number from another packed BCD number, to make sure the result is correct packed BCD. The result of the subtraction must be in AL for DAS to work correctly. If the lower nibble in AL after a subtraction is greater than 9 or the AF was set by the subtraction, then the DAS instruction will subtract 6 from the lower nibble AL. If the result in the upper nibble is now greater than 9 or if the carry flag was set, the DAS instruction will subtract 60 from AL.

Let AL = 86 BCD, and BH = 57 BCD
SUB AL, BH AL = 2FH; lower nibble > 9, subtract 06H from AL AL = 29 BCD, CF = 0

Let AL = 49 BCD, and BH = 72 BCD
SUB AL, BH AL = D7H; upper nibble > 9, subtract 60H from AL
DAS AL = 77 BCD, CF = 1 (borrow is needed) The DAS instruction updates AF, CF, SF, PF, and ZF; but OF is undefined.

CBW (CONVERT SIGNED BYTE TO SIGNED WORD)

This instruction copies the sign bit of the byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. CBW does not affect any flag.

Let AX = 00000000 10011011 (–155 decimal)
CBW Convert signed byte in AL to signed word in AX AX = 11111111 10011011 (–155 decimal)

CWD (CONVERT SIGNED WORD TO SIGNED DOUBLE WORD)

This instruction copies the sign bit of a word in AX to all the bits of the DX register. In other words, it extends the sign of AX into all of DX. CWD affects no flags.

Let DX = 00000000 00000000, and AX = 11110000 11000111 (–3897 decimal)
CWD Convert signed word in AX to signed double word in DX:AX DX = 11111111 11111111
AX = 11110000 11000111 (–3897 decimal)

AAA (ASCII ADJUST FOR ADDITION)

Numerical data coming into a computer from a terminal is usually in ASCII code. In this code, the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each. After the addition, the AAA instruction is used to make sure the result is the correct unpacked BCD.

Let AL = 0011 0101 (ASCII 5), and BL = 0011 1001 (ASCII 9)
ADD AL, BL AL = 0110 1110 (6EH, which is incorrect BCD)
AAA AL = 0000 0100 (unpacked BCD 4)
CF = 1 indicates answer is 14 decimal.

The AAA instruction works only on the AL register. The AAA instruction updates AF and CF; but OF, PF, SF and ZF are left undefined.

AAS (ASCII ADJUST FOR SUBTRACTION)

Numerical data coming into a computer from a terminal is usually in an ASCII code. In this code the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to subtract the ASCII codes for two decimal digits without masking the "3" in the upper nibble of each. The AAS instruction is then used to make sure the result is the correct unpacked BCD.

Let AL = 00111001 (39H or ASCII 9), and BL = 00110101 (35H or ASCII 5) SUB AL, BL AL = 00000100 (BCD 04), and CF = 0
AAS AL = 00000100 (BCD 04), and CF = 0 (no borrow required)
Let AL = 00110101 (35H or ASCII 5), and BL = 00111001 (39H or ASCII 9)
SUB AL, BL AL = 11111100 (– 4 in 2's complement form), and CF = 1
AAS AL = 00000100 (BCD 06), and CF = 1 (borrow required)

The AAS instruction works only on the AL register. It updates ZF and CF; but OF, PF, SF, AF are left undefined.

AAM (BCD ADJUST AFTER MULTIPLY)

Before you can multiply two ASCII digits, you must first mask the upper 4 bit of each. This leaves unpacked BCD (one BCD digit per byte) in each byte. After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX. AAM works only after the multiplication of two unpacked BCD bytes, and it works only the operand in AL. AAM updates PF, SF and ZF but AF; CF and OF are left undefined.

Let AL = 00000101 (unpacked BCD 5), and BH = 00001001 (unpacked BCD 9) MUL BH AL x BH: AX = 00000000 00101101 = 002DH
AAM AX = 00000100 00000101 = 0405H (unpacked BCD for 45)

AAD (BCD-TO-BINARY CONVERT BEFORE DIVISION)

AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. After the BCD division, AL will contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder. AAD updates PF, SF and ZF; AF, CF and OF are left undefined.

Let AX = 0607 (unpacked BCD for 67 decimal), and CH = 09H AAD AX = 0043 (43H = 67 decimal)
DIV CH AL = 07; AH = 04; Flags undefined after DIV If an attempt is made to divide by 0, the 8086 will generate a type 0 interrupt.

LOGICAL INSTRUCTIONS

AND – AND Destination, Source

This instruction ANDs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0 after AND. PF, SF, and ZF are updated by the AND instruction. AF is undefined. PF has meaning only for an 8-bit operand.

AND CX, [SI] AND word in DS at offset [SI] with word in CX register; Result in CX register
AND BH, CL AND byte in CL with byte in BH; Result in BH
AND BX, 00FFH 00FFH Masks upper byte, leaves lower byte unchanged.

OR – OR Destination, Source

This instruction ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and destination cannot both be memory locations. CF and OF are both 0 after OR. PF, SF, and ZF are updated by the OR instruction. AF is undefined. PF has meaning only for an 8-bit operand.

OR AH, CL CL ORed with AH, result in AH, CL not changed

OR BP, SI SI ORed with BP, result in BP, SI not changed
OR SI, BP BP ORed with SI, result in SI, BP not changed
OR BL, 80H BL ORed with immediate number 80H; sets MSB of BL to 1
OR CX, TABLE [SI] CX ORed with word from effective address TABLE [SI];
Content of memory is not changed.

XOR – XOR Destination, Source

This instruction Exclusive-ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and destination cannot both be memory locations. CF and OF are both 0 after XOR. PF, SF, and ZF are updated. PF has meaning only for an 8-bit operand. AF is undefined.

XOR CL, BH Byte in BH exclusive-ORed with byte in CL. Result in CL. BH not changed.
XOR BP, DI Word in DI exclusive-ORed with word in BP. Result in BP. DI not changed.

XOR WORD PTR [BX], 00FFH Exclusive-OR immediate number 00FFH with word at offset [BX] in the data segment. Result in memory location [BX]

NOT – NOT Destination

The NOT instruction inverts each bit (forms the 1's complement) of a byte or word in the specified destination. The destination can be a register or a memory location. This instruction does not affect any flag.

NOT BX Complement content or BX register
NOT BYTE PTR [BX] Complement memory byte at offset [BX] in data segment.

NEG – NEG Destination

This instruction replaces the number in a destination with its 2's complement. The destination can be a register or a memory location. It gives the same result as the invert each bit and add one algorithm. The NEG instruction updates AF, AF, PF, ZF, and OF.

NEG AL Replace number in AL with its 2's complement
NEG BX Replace number in BX with its 2's complement
NEG BYTE PTR [BX] Replace byte at offset BX in DX with its 2's complement
NEG WORD PTR [BP] Replace word at offset BP in SS with its 2's complement

CMP – CMP Destination, Source

This instruction compares a byte / word in the specified source with a byte / word in the specified destination. The source can be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, the source and the destination cannot both be memory locations. The comparison is actually done by subtracting the source byte or word from the destination byte or word. The source and the destination are not changed, but the flags are set to indicate the results of the comparison. AF, OF, SF, ZF, PF, and CF are updated by the CMP instruction. For the instruction CMP CX, BX, the values of CF, ZF, and SF will be as follows:

	CF	ZF	SF	
CX = BX	0	1	0	Result of subtraction is 0
CX > BX	0	0	0	No borrow required, so CF = 0
CX < BX	1	0	1	Subtraction requires borrow, so CF = 1

CMP AL, 01H Compare immediate number 01H with byte in AL
CMP BH, CL Compare byte in CL with byte in BH
CMP CX, TEMP Compare word in DS at displacement TEMP with word at CX
CMP PRICES [BX], 49H Compare immediate number 49H with byte at offset [BX] in array PRICES

TEST – TEST Destination, Source

This instruction ANDs the byte / word in the specified source with the byte / word in the specified destination. Flags are updated, but neither operand is changed. The test instruction is often used to set

flags before a Conditional jump instruction.

The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0's after TEST. PF, SF and ZF will be updated to show the results of the destination. AF is be undefined.

TEST AL, BH AND BH with AL. No result stored; Update PF, SF, ZF.
TEST CX, 0001H AND CX with immediate number 0001H; No result stored; Update PF, SF, ZF
TEST BP, [BX][DI] AND word are offset [BX][DI] in DS with word in BP. No result stored. Update PF, SF, and ZF

ROTATE AND SHIFT INSTRUCTIONS

RCL – RCL Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to the left. The operation circular because the MSB of the operand is rotated into the carry flag and the bit in the carry flag is rotated around into LSB of the operand.

CF MSB LSB

For multi-bit rotates, CF will contain the bit most recently rotated out of the MSB.

The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate by more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

RCL affects only CF and OF. OF will be a 1 after a single bit RCL if the MSB was changed by the rotate. OF is undefined after the multi-bit rotate.

RCL DX, 1 Word in DX 1 bit left, MSB to CF, CF to LSB
MOV CL, 4 Load the number of bit positions to rotate into CL
RCL SUM [BX], CL Rotate byte or word at effective address SUM [BX] 4 bits left
Original bit 4 now in CF, original CF now in bit 3.

RCR – RCR Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to the right. The operation circular because the LSB of the operand is rotated into the carry flag and the bit in the carry flag is rotate around into MSB of the operand.

CF MSB LSB

For multi-bit rotate, CF will contain the bit most recently rotated out of the LSB.

The destination can be a register or a memory location. If you want to rotate the operand by one bit

position, you can specify this by putting a 1 in the count position of the instruction. To rotate more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

RCR affects only CF and OF. OF will be a 1 after a single bit RCR if the MSB was changed by the rotate. OF is undefined after the multi-bit rotate.

```
        RCR BX, 1       Word in BX right 1 bit, CF to MSB, LSB to CF
        MOV CL, 4       Load CL for rotating 4 bit position
RCR BYTE PTR [BX], 4        Rotate the byte at offset [BX] in DS 4 bit positions right
CF = original bit 3, Bit 4 – original CF.
```

ROL – ROL Destination, Count

This instruction rotates all the bits in a specified word or byte to the left some number of bit positions. The data bit rotated out of MSB is circled back into the LSB. It is also copied into CF. In the case of multiple-bit rotate, CF will contain a copy of the bit most recently moved out of the MSB.

CF MSB LSB

The destination can be a register or a memory location. If you to want rotate the operand by one bit position, you can specify this by putting 1 in the count position in the instruction. To rotate more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

ROL affects only CF and OF. OF will be a 1 after a single bit ROL if the MSB was changed by the rotate.

```
        ROL AX, 1       Rotate the word in AX 1 bit position left, MSB to LSB and CF
        MOV CL, 04H Load number of bits to rotate in CL ROL BL, CL      Rotate BL 4 bit
positions
        ROL FACTOR [BX], 1      Rotate the word or byte in DS at EA = FACTOR [BX]
by 1 bit position left into CF
```

ROR – ROR Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to right. The operation is desired as a rotate rather than shift, because the bit moved out of the LSB is rotated around into the MSB. The data bit moved out of the LSB is also copied into CF. In the case of multiple bit rotates, CF will contain a copy of the bit most recently moved out of the LSB.

CF MSB LSB

The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting 1 in the count position in the instruction. To rotate by more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

ROR affects only CF and OF. OF will be a 1 after a single bit ROR if the MSB was changed by the rotate.

ROR BL, 1 Rotate all bits in BL right 1 bit position LSB to MSB and to CF
MOV CL, 08H Load CL with number of bit positions to be rotated ROR WORD PTR [BX], CL Rotate word in DS at offset [BX] 8 bit position right

SAL – SAL Destination, Count SHL – SHL Destination, Count

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination some number of bit positions to the left. As a bit is shifted out of the LSB operation, a 0 is put in the LSB position. The MSB will be shifted into CF. In the case of multi-bit shift, CF will contain the bit most recently shifted out from the MSB. Bits shifted into CF previously will be lost.

CF MSB LSB 0

The destination operand can be a byte or a word. It can be in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more than 1 bit position, load the desired number of shifts into the CL register, and put "CL" in the count position of the instruction.

The flags are affected as follow: CF contains the bit most recently shifted out from MSB. For a count of one, OF will be 1 if CF and the current MSB are not the same. For multiple-bit shifts, OF is undefined. SF and ZF will be updated to reflect the condition of the destination. PF will have meaning only for an operand in AL. AF is undefined.

SAL BX, 1 Shift word in BX 1 bit position left, 0 in LSB
MOV CL, 02h Load desired number of shifts in CL
SAL BP, CL Shift word in BP left CL bit positions, 0 in LSBs
SAL BYTE PTR [BX], 1 Shift byte in DX at offset [BX] 1 bit position left, 0 in LSB

SAR – SAR Destination, Count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. In other words, the sign bit is copied into the MSB. The LSB will be shifted into CF. In the case of multiple-bit shift, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.

MSB MSB LSB CF

The destination operand can be a byte or a word. It can be in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more than 1 bit position, load the desired number of shifts into the CL register, and put "CL" in the count position of the instruction.

The flags are affected as follow: CF contains the bit most recently shifted in from LSB. For a count of one, OF will be 1 if the two MSBs are not the same. After a multi-bit SAR, OF will be 0. SF and ZF will be updated to show the condition of the destination. PF will have meaning only for an 8- bit destination. AF will be undefined after SAR.

SAR DX, 1 Shift word in DI one bit position right, new MSB = old MSB
MOV CL, 02H Load desired number of shifts in CL

SAR WORD PTR [BP], CL Shift word at offset [BP] in stack segment right by two bit positions, the two MSBs are now copies of original LSB

SHR – SHR Destination, Count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a 0 is put in its place. The bit shifted out of the LSB position goes to CF. In the case of multi-bit shifts, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.

0 MSB LSB CF

The destination operand can be a byte or a word in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more than 1 bit position, load the desired number of shifts into the CL register, and put "CL" in the count position of the instruction.

The flags are affected by SHR as follow: CF contains the bit most recently shifted out from LSB. For a count of one, OF will be 1 if the two MSBs are not both 0's. For multiple-bit shifts, OF will be meaningless. SF and ZF will be updated to show the condition of the destination. PF will have meaning only for an 8-bit destination. AF is undefined.

SHR BP, 1 Shift word in BP one bit position right, 0 in MSB
MOV CL, 03H Load desired number of shifts into CL
SHR BYTE PTR [BX] Shift byte in DS at offset [BX] 3 bits right; 0's in 3 MSBs

TRANSFER-OF-CONTROL INSTRUCTIONS

Note: The following rules apply to the discussions presented in this section.
The terms above and below are used when referring to the magnitude of unsigned numbers. For example, the number 00000111 (7) is above the number 00000010 (2), whereas the number 00000100 (4) is below the number 00001110 (14).
The terms greater and less are used to refer to the relationship of two signed numbers. Greater means more positive. The number 00000111 (+7) is greater than the number 11111110 (-2), whereas the number 11111100 (-4) is less than the number 11110100 (-6).
In the case of Conditional jump instructions, the destination address must be in the range of –128 bytes to +127 bytes from the address of the next instruction
These instructions do not affect any flags.

JMP (UNCONDITIONAL JUMP TO SPECIFIED DESTINATION)

This instruction will fetch the next instruction from the location specified in the instruction rather than from the next location after the JMP instruction. If the destination is in the same code segment as the JMP instruction, then only the instruction pointer will be changed to get the destination location. This is referred to as a near jump. If the destination for the jump instruction is in a segment with a name different from that of the segment containing the JMP instruction, then both the instruction pointer and the code segment register content will be changed to get the destination location. This referred to as a far jump. The JMP instruction does not affect any flag.

JMP CONTINUE

This instruction fetches the next instruction from address at label CONTINUE. If the label is in the same segment, an offset coded as part of the instruction will be added to the instruction pointer to produce the new fetch address. If the label is another segment, then IP and CS will be replaced with value coded in

part of the instruction. This type of jump is referred to as direct because the displacement of the destination or the destination itself is specified directly in the instruction.

JMP BX

This instruction replaces the content of IP with the content of BX. BX must first be loaded with the offset of the destination instruction in CS. This is a near jump. It is also referred to as an indirect jump because the new value of IP comes from a register rather than from the instruction itself, as in a direct jump.

JMP WORD PTR [BX]

This instruction replaces IP with word from a memory location pointed to by BX in DX. This is an indirect near jump.

JMP DWORD PTR [SI]

This instruction replaces IP with word pointed to by SI in DS. It replaces CS with a word pointed by SI + 2 in DS. This is an indirect far jump.

JA / JNBE (JUMP IF ABOVE / JUMP IF NOT BELOW OR EQUAL)

If, after a compare or some other instructions which affect flags, the zero flag and the carry flag both are 0, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are not both 0, the instruction will have no effect on program execution.

CMP AX, 4371H Compare by subtracting 4371H from AX JA NEXT Jump to label NEXT if AX above 4371H
CMP AX, 4371H Compare (AX – 4371H)
JNBE NEXT Jump to label NEXT if AX not below or equal to 4371H
JAE / JNB / JNC
(JUMP IF ABOVE OR EQUAL / JUMP IF NOT BELOW / JUMP IF NO CARRY)

If, after a compare or some other instructions which affect flags, the carry flag is 0, this instruction will cause execution to jump to a label given in the instruction. If CF is 1, the instruction will have no effect on program execution.

CMP AX, 4371HCompare (AX – 4371H)
JAE NEXT Jump to label NEXT if AX above 4371H
CMP AX, 4371H Compare (AX – 4371H)
JNB NEXT Jump to label NEXT if AX not below 4371H
ADD AL, BL Add two bytes
JNC NEXT If the result with in acceptable range, continue
JB / JC / JNAE (JUMP IF BELOW / JUMP IF CARRY / JUMP IF NOT ABOVE OR EQUAL)

If, after a compare or some other instructions which affect flags, the carry flag is a 1, this instruction will cause execution to jump to a label given in the instruction. If CF is 0, the instruction will have no

effect on program execution.

```
CMP AX, 4371H Compare (AX – 4371H)
JB NEXT Jump to label NEXT if AX below 4371H
ADD BX, CX      Add two words
JC NEXT Jump to label NEXT if CF = 1
CMP AX, 4371H Compare (AX – 4371H)
JNAE NEXT       Jump to label NEXT if AX not above or equal to 4371H
```

JBE / JNA (JUMP IF BELOW OR EQUAL / JUMP IF NOT ABOVE)

If, after a compare or some other instructions which affect flags, either the zero flag or the carry flag is 1, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are both 0, the instruction will have no effect on program execution.

```
CMP AX, 4371H Compare (AX – 4371H)
JBE NEXT        Jump to label NEXT if AX is below or equal to 4371H
CMP AX, 4371H Compare (AX – 4371H)
JNA NEXT        Jump to label NEXT if AX not above 4371H
```

JG / JNLE (JUMP IF GREATER / JUMP IF NOT LESS THAN OR EQUAL)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction, if the zero flag is 0 and the carry flag is the same as the overflow flag.

```
CMP BL, 39H     Compare by subtracting 39H from BL
JG NEXT Jump to label NEXT if BL more positive than 39H
CMP BL, 39H     Compare by subtracting 39H from BL
JNLE NEXT       Jump to label NEXT if BL is not less than or equal to 39H
```

JGE / JNL (JUMP IF GREATER THAN OR EQUAL / JUMP IF NOT LESS THAN)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction, if the sign flag is equal to the overflow flag.

```
 CMP BL, 39H    Compare by subtracting 39H from BL
JGE NEXT        Jump to label NEXT if BL more positive than or equal to 39H
CMP BL, 39H     Compare by subtracting 39H from BL
JNL NEXT        Jump to label NEXT if BL not less than 39H
```

JL / JNGE (JUMP IF LESS THAN / JUMP IF NOT GREATER THAN OR EQUAL)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the sign flag is not equal to the overflow flag.

```
        CMP BL, 39H Compare by subtracting 39H from BL
JL AGAIN        Jump to label AGAIN if BL more negative than 39H
        CMP BL, 39H Compare by subtracting 39H from BL
JNGE AGAIN      Jump to label AGAIN if BL not more positive than or equal to 39H
```

JLE / JNG (JUMP IF LESS THAN OR EQUAL / JUMP IF NOT GREATER)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the

label given in the instruction if the zero flag is set, or if the sign flag not equal to the overflow flag.

```
CMP BL, 39H     Compare by subtracting 39H from BL
JLE NEXT        Jump to label NEXT if BL more negative than or equal to 39H
CMP BL, 39H     Compare by subtracting 39H from BL
JNG NEXT        Jump to label NEXT if BL not more positive than 39H
```

JE / JZ (JUMP IF EQUAL / JUMP IF ZERO)

This instruction is usually used after a Compare instruction. If the zero flag is set, then this instruction will cause a jump to the label given in the instruction.

```
CMP BX, DX      Compare (BX-DX)
JE DONE Jump to DONE if BX = DX
IN AL, 30H      Read data from port 8FH SUB AL, 30H    Subtract the minimum value.
JZ START        Jump to label START if the result of subtraction is 0
```

JNE / JNZ (JUMP NOT EQUAL / JUMP IF NOT ZERO)

This instruction is usually used after a Compare instruction. If the zero flag is 0, then this instruction will cause a jump to the label given in the instruction.

```
IN AL, 0F8H     Read data value from port
CMP AL, 72      Compare (AL –72)
JNE NEXT        Jump to label NEXT if AL  72
ADD AX, 0002H Add count factor 0002H to AX DEC BX    Decrement BX
JNZ NEXT        Jump to label NEXT if BX  0
```

JS (JUMP IF SIGNED / JUMP IF NEGATIVE)

This instruction will cause a jump to the specified destination address if the sign flag is set. Since a 1 in the sign flag indicates a negative signed number, you can think of this instruction as saying "jump if negative".

```
ADD BL, DH      Add signed byte in DH to signed byte in DL
JS NEXT  Jump to label NEXT if result of addition is negative number
```

JNS (JUMP IF NOT SIGNED / JUMP IF POSITIVE)

This instruction will cause a jump to the specified destination address if the sign flag is 0. Since a 0 in the sign flag indicate a positive signed number, you can think to this instruction as saying "jump if positive".

```
DEC AL Decrement AL
JNS NEXT        Jump to label NEXT if AL has not decremented to FFH
```

JP / JPE (JUMP IF PARITY / JUMP IF PARITY EVEN)

If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is even, then the parity flag will be set. If the parity flag is set, the JP / JPE instruction will cause a jump to the specified destination address.

```
IN AL, 0F8H     Read ASCII character from Port F8H OR AL, AL          Set flags
```

JPE ERROR Odd parity expected, send error message if parity found even
JNP / JPO (JUMP IF NO PARITY / JUMP IF PARITY ODD)

If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is odd, then the parity flag is 0. The JNP / JPO instruction will cause a jump to the specified destination address, if the parity flag is 0.

IN AL, 0F8H Read ASCII character from Port F8H OR AL, AL Set flags
JPO ERROR Even parity expected, send error message if parity found odd
JO (JUMP IF OVERFLOW)

The overflow flag will be set if the magnitude of the result produced by some signed arithmetic operation is too large to fit in the destination register or memory location. The JO instruction will cause a jump to the destination given in the instruction, if the overflow flag is set.

ADD AL, BL Add signed bytes in AL and BL
JO ERROR Jump to label ERROR if overflow from add

JNO (JUMP IF NO OVERFLOW)

The overflow flag will be set if some signed arithmetic operation is too large to fit in the destination register or memory location. The JNO instruction will cause a jump to the destination given in the instruction, if the overflow flag is not set.

ADD AL, BL Add signed byte in AL and BL
JNO DONE Process DONE if no overflow
JCXZ (JUMP IF THE CX REGISTER IS ZERO)

This instruction will cause a jump to the label to a given in the instruction, if the CX register contains all 0's. The instruction does not look at the zero flag when it decides whether to jump or not.

JCXZ SKIP If CX = 0, skip the process SUB [BX], 07H Subtract 7 from data value
SKIP: ADD C Next instruction
LOOP (JUMP TO SPECIFIED LABEL IF CX □ 0 AFTER AUTO DECREMENT)

This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX is not 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the auto decrement, execution will simply go on to the next instruction after LOOP. The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOP instruction. This instruction does not affect any flag.

MOV BX, OFFSET PRICES Point BX at first element in array
MOV CX, 40 Load CX with number of elements in array
NEXT: MOV AL, [BX] Get element from array
INC AL Increment the content of AL
MOV [BX], AL Put result back in array
INC BX Increment BX to point to next location
LOOP NEXT Repeat until all elements adjusted

LOOPE / LOOPZ (LOOP WHILE CX □ 0 AND ZF = 1)

This instruction is used to repeat a group of instructions some number of times, or until the zero flag becomes 0. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX 0 and ZF = 1, execution will jump to a destination specified by a label in the instruction. If CX = 0, execution simply go on the next instruction after LOOPE / LOOPZ. In other words, the two ways to exit the loop are CX = 0 or ZF =
The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOPE / LOOPZ instruction. This instruction does not affect any flag.

MOV BX, OFFSET ARRAY Point BX to address of ARRAY before start of array DEC BX Decrement BX
MOV CX, 100 Put number of array elements in CX
NEXT: INC BX Point to next element in array
CMP [BX], OFFH Compare array element with FFH LOOPE NEXT

LOOPNE / LOOPNZ (LOOP WHILE CX NOT EQUAL TO 0 AND ZF = 0)

This instruction is used to repeat a group of instructions some number of times, or until the zero flag becomes a 1. The number of times the instruction sequence is to be repeated is loaded into the count register CX. Each time the LOOPNE / LOOPNZ instruction executes, CX is automatically decremented by 1. If CX NOT EQUAL TO 0 and ZF = 0, execution will jump to a destination specified by a label in the instruction. If CX = 0, after the auto decrement or if ZF = 1, execution simply go on the next instruction after LOOPNE
/ LOOPNZ. In other words, the two ways to exit the loop are CX = 0 or ZF = 1. The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOPNE / LOOPZ instruction. This instruction does not affect any flags.

MOV BX, OFFSET ARRAY Point BX to adjust before start of array DEC BX Decrement BX
MOV CX, 100 Put number of array in CX
NEXT: INC BX Point to next element in array
CMP [BX], ODH Compare array element with 0DH LOOPNZ NEXT
CALL (CALL A PROCEDURE)

The CALL instruction is used to transfer execution to a subprogram or a procedure. There two basic type of calls near and far.
A near call is a call to a procedure, which is in the same code segment as the CALL instruction. When the 8086 executes a near CALL instruction, it decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL into the stack. This offset saved in the stack is referred to as the return address, because this is the address that execution will return to after the procedure is executed. A near CALL instruction will also load the instruction pointer with the offset of the first instruction in the procedure. A RET instruction at the end of the procedure will return execution to the offset saved on the stack which is copied back to IP. A far call is a call to a procedure, which is in a different segment from the one that contains the CALL instruction. When the 8086 executes a far call, it decrements the stack pointer by 2 and copies the content of the CS register to the stack. It then decrements the stack pointer by 2 again and copies the offset of the instruction after the CALL instruction to the stack. Finally, it loads CS with the segment base of the segment that contains the procedure, and loads IP with the offset of the first instruction of the procedure in that segment. A RET instruction at the end of the procedure will return execution to the next instruction after the CALL by restoring the saved values of

CS and IP from the stack.

CALL MULT
This is a direct within segment (near or intra segment) call. MULT is the name of the procedure. The assembler determines the displacement of MULT from the instruction after the CALL and codes this displacement in as part of the instruction.

CALL BX
This is an indirect within-segment (near or intra-segment) call. BX contains the offset of the first instruction of the procedure. It replaces content of IP with content of register BX.

CALL WORD PTR [BX]
This is an indirect within-segment (near or intra-segment) call. Offset of the first instruction of the procedure is in two memory addresses in DS. Replaces content of IP with content of word memory location in DS pointed to by BX.

CALL DIVIDE
This is a direct call to another segment (far or inter-segment call). DIVIDE is the name of the procedure. The procedure must be declared far with DIVIDE PROC FAR at its start. The assembler will determine the code segment base for the segment that contains the procedure and the offset of the start of the procedure. It will put these values in as part of the instruction code.

CALL DWORD PTR [BX]
This is an indirect call to another segment (far or inter-segment call). New values for CS and IP are fetched from four-memory location in DS. The new value for CS is fetched from [BX] and [BX + 1]; the new IP is fetched from [BX + 2] and [BX +3].

RET (RETURN EXECUTION FROM PROCEDURE TO CALLING PROGRAM)

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction which was used to call the procedure. If the procedure is near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the IP with a word from the top of the stack. The word from the top of the stack is the offset of the next instruction after the CALL. This offset was pushed into the stack as part of the operation of the CALL instruction. The stack pointer will be incremented by 2 after the return address is popped off the stack.

If the procedure is a far procedure (in a code segment other than the one from which it is called), then the instruction pointer will be replaced by the word at the top of the stack. This word is the offset part of the return address put there by the CALL instruction. The stack pointer will then be incremented by 2. The CS register is then replaced with a word from the new top of the stack. This word is the segment base part of the return address that was pushed onto the stack by a far call operation. After this, the stack pointer is again incremented by 2.

A RET instruction can be followed by a number, for example, RET 6. In this case, the stack pointer will be incremented by an additional six addresses after the IP when the IP and CS are popped off the stack. This form is used to increment the stack pointer over parameters passed to the procedure on the stack.

The RET instruction does not affect any flag.

STRING MANIPULATION INSTRUCTIONS

MOVS – MOVS Destination String Name, Source String Name MOVSB – MOVSB Destination String Name, Source String Name MOVSW – MOVSW Destination String Name, Source String Name

This instruction copies a byte or a word from location in the data segment to a location in the extra segment. The offset of the source in the data segment must be in the SI register. The offset of the destination in the extra segment must be in the DI register. For multiple-byte or multiple-word moves, the

number of elements to be moved is put in the CX register so that it can function as a counter. After the byte or a word is moved, SI and DI are automatically adjusted to point to the next source element and the next destination element. If DF is 0, then SI and DI will incremented by 1 after a byte move and by 2 after a word move. If DF is 1, then SI and DI will be decremented by 1 after a byte move and by 2 after a word move. MOVS does not affect any flag.

When using the MOVS instruction, you must in some way tell the assembler whether you want to move a string as bytes or as word. There are two ways to do this. The first way is to indicate the name of the source and destination strings in the instruction, as, for example. MOVS DEST, SRC. The assembler will code the instruction for a byte / word move if they were declared with a DB / DW. The second way is to add a "B" or a "W" to the MOVS mnemonic. MOVSB says move a string as bytes; MOVSW says move a string as words.

MOV SI, OFFSET SOURCE Load offset of start of source string in DS into SI MOV DI, OFFSET DESTINATION Load offset of start of destination string in ES into DI CLD Clear DF to auto increment SI and DI after move
MOV CX, 04H Load length of string into CX as counter
REP MOVSB Move string byte until CX = 0

LODS / LODSB / LODSW (LOAD STRING BYTE INTO AL OR STRING WORD INTO AX)

This instruction copies a byte from a string location pointed to by SI to AL, or a word from a string location pointed to by SI to AX. If DF is 0, SI will be automatically incremented (by 1 for a byte string, and 2 for a word string) to point to the next element of the string. If DF is 1, SI will be automatically decremented (by 1 for a byte string, and 2 for a word string) to point to the previous element of the string. LODS does not affect any flag.

CLD Clear direction flag so that SI is auto-incremented MOV SI, OFFSET SOURCE Point SI to start of string
LODS SOURCE Copy a byte or a word from string to AL or AX

Note: The assembler uses the name of the string to determine whether the string is of type bye or type word. Instead of using the string name to do this, you can use the mnemonic LODSB to tell the assembler that the string is type byte or the mnemonic LODSW to tell the assembler that the string is of type word.

STOS / STOSB / STOSW (STORE STRING BYTE OR STRING WORD)

This instruction copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI. In effect, it replaces a string element with a byte from AL or a word from AX. After the copy, DI is automatically incremented or decremented to point to next or previous element of the string. If DF is cleared, then DI will automatically incremented by 1 for a byte string and by 2 for a word string. If DI is set, DI will be automatically decremented by 1 for a byte string and by 2 for a word string. STOS does not affect any flag.

MOV DI, OFFSET TARGET STOS TARGET

Note: The assembler uses the string name to determine whether the string is of type byte or type word. If it is a byte string, then string byte is replaced with content of AL. If it is a word string, then string word is replaced with content of AX.

MOV DI, OFFSET TARGET STOSB

"B" added to STOSB mnemonic tells assembler to replace byte in string with byte from AL. STOSW would tell assembler directly to replace a word in the string with a word from AX.

CMPS / CMPSB / CMPSW (COMPARE STRING BYTES OR STRING WORDS)

This instruction can be used to compare a byte / word in one string with a byte / word in another string. SI is used to hold the offset of the byte or word in the source string, and DI is used to hold the offset of the byte or word in the destination string.

The AF, CF, OF, PF, SF, and ZF flags are affected by the comparison, but the two operands are not affected. After the comparison, SI and DI will automatically be incremented or decremented to point to the next or previous element in the two strings. If DF is set, then SI and DI will automatically be decremented by 1 for a byte string and by 2 for a word string. If DF is reset, then SI and DI will automatically be incremented by 1 for byte strings and by 2 for word strings. The string pointed to by SI must be in the data segment. The string pointed to by DI must be in the extra segment.

The CMPS instruction can be used with a REPE or REPNE prefix to compare all the elements of a string.

MOV SI, OFFSET FIRST Point SI to source string MOV DI, OFFSET SECOND Point DI to destination string
CLD DF cleared, SI and DI will auto-increment after compare
MOV CX, 100 Put number of string elements in CX
REPE CMPSB Repeat the comparison of string bytes until end of string or until compared bytes are not equal

CX functions as a counter, which the REPE prefix will cause CX to be decremented after each compare. The B attached to CMPS tells the assembler that the strings are of type byte. If you want to tell the assembler that strings are of type word, write the instruction as CMPSW. The REPE CMPSW instruction will cause the pointers in SI and DI to be incremented by 2 after each compare, if the direction flag is set.

SCAS / SCASB / SCASW (SCAN A STRING BYTE OR A STRING WORD)

SCAS compares a byte in AL or a word in AX with a byte or a word in ES pointed to by DI. Therefore, the string to be scanned must be in the extra segment, and DI must contain the offset of the byte or the word to be compared. If DF is cleared, then DI will be incremented by 1 for byte strings and by 2 for

word strings. If DF is set, then DI will be decremented by 1 for byte strings and by 2 for word strings. SCAS affects AF, CF, OF, PF, SF, and ZF, but it does not change either the operand in AL (AX) or the operand in the string.

The following program segment scans a text string of 80 characters for a carriage return, 0DH, and puts the offset of string into DI:

MOV DI, OFFSET STRING
MOV AL, 0DH Byte to be scanned for into AL
MOV CX, 80 CX used as element counter
CLD Clear DF, so that DI auto increments REPNE SCAS STRING Compare byte in string with byte in AL

REP / REPE / REPZ / REPNE / REPNZ (PREFIX)
(REPEAT STRING INSTRUCTION UNTIL SPECIFIED CONDITIONS EXIST)

REP is a prefix, which is written before one of the string instructions. It will cause the CX register to be decremented and the string instruction to be repeated until CX = 0. The instruction REP MOVSB, for example, will continue to copy string bytes until the number of bytes loaded into CX has been copied.

REPE and REPZ are two mnemonics for the same prefix. They stand for repeat if equal and repeat if zero, respectively. They are often used with the Compare String instruction or with the Scan String instruction. They will cause the string instruction to be repeated as long as the compared bytes or words are equal (ZF = 1) and CX is not yet counted down to zero. In other words, there are two conditions that will stop the repetition: CX = 0 or string bytes or words not equal.

REPE CMPSB Compare string bytes until end of string or until string bytes not equal.

REPNE and REPNZ are also two mnemonics for the same prefix. They stand for repeat if not equal and repeat if not zero, respectively. They are often used with the Compare String instruction or with the Scan String instruction. They will cause the string instruction to be repeated as long as the compared bytes or words are not equal (ZF = 0) and CX is not yet counted down to zero.

REPNE SCASW Scan a string of word until a word in the string matches the word in AX or until all of the string has been scanned.

The string instruction used with the prefix determines which flags are affected.

FLAG MANIPULATION INSTRUCTIONS

STC (SET CARRY FLAG) This instruction sets the carry flag to 1. It does not affect any other flag.
CLC (CLEAR CARRY FLAG) This instruction resets the carry flag to 0. It does not affect any other flag.
CMC (COMPLEMENT CARRY FLAG) This instruction complements the carry flag. It does not affect any other flag.
STD (SET DIRECTION FLAG) This instruction sets the direction flag to 1. It does not affect any other flag.
CLD (CLEAR DIRECTION FLAG)
This instruction resets the direction flag to 0. It does not affect any other flag.

STI (SET INTERRUPT FLAG)

Setting the interrupt flag to a 1 enables the INTR interrupt input of the 8086. The instruction will not take affect until the next instruction after STI. When the INTR input is enabled, an interrupt signal on this input will then cause the 8086 to interrupt program execution, push the return address and flags on the stack, and execute an interrupt service procedure. An IRET instruction at the end of the interrupt service procedure will restore the return address and flags that were pushed onto the stack and return execution to the interrupted program. STI does not affect any other flag.

CLI (CLEAR INTERRUPT FLAG)

This instruction resets the interrupt flag to 0. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input. The CLI instructions, however, has no effect on the non-maskable interrupt input, NMI. It does not affect any other flag.

LAHF (COPY LOW BYTE OF FLAG REGISTER TO AH REGISTER)

The LAHF instruction copies the low-byte of the 8086 flag register to AH register. It can then be pushed onto the stack along with AL by a PUSH AX instruction. LAHF does not affect any flag.

SAHF (COPY AH REGISTER TO LOW BYTE OF FLAG REGISTER)

The SAHF instruction replaces the low-byte of the 8086 flag register with a byte from the AH register. SAHF changes the flags in lower byte of the flag register.

STACK RELATED INSTRUCTIONS

PUSH – PUSH Source

The PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment to which the stack pointer points. The source of the word can be general- purpose register, segment register, or memory. The stack segment register and the stack pointer must be initialized before this instruction can be used. PUSH can be used to save data on the stack so that it will not destroyed by a procedure. This instruction does not affect any flag.

PUSH BX Decrement SP by 2, copy BX to stack.
PUSH DS Decrement SP by 2, copy DS to stack.
PUSH BL Illegal; must push a word
PUSH TABLE [BX] Decrement SP by 2, and copy word from memory in DS at
EA = TABLE + [BX] to stack

POP – POP Destination

The POP instruction copies a word from the stack location pointed to by the stack pointer to a destination specified in the instruction. The destination can be a general-purpose register, a segment register or a memory location. The data in the stack is not changed. After the word is copied to the specified destination, the stack pointer is automatically incremented by 2 to point to the next word on the stack. The POP instruction does not affect any flag.

POP DX Copy a word from top of stack to DX; increment SP by 2
POP DS Copy a word from top of stack to DS; increment SP by 2
POP TABLE [DX] Copy a word from top of stack to memory in DS with EA = TABLE + [BX]; increment SP by 2.

PUSHF (PUSH FLAG REGISTER TO STACK)

The PUSHF instruction decrements the stack pointer by 2 and copies a word in the flag register to two memory locations in stack pointed to by the stack pointer. The stack segment register is not affected. This instruction does to affect any flag.

POPF (POP WORD FROM TOP OF STACK TO FLAG REGISTER)

The POPF instruction copies a word from two memory locations at the top of the stack to the flag register and increments the stack pointer by 2. The stack segment register and word on the stack are not affected. This instruction does to affect any flag.

INPUT-OUTPUT INSTRUCTIONS

IN – IN Accumulator, Port

The IN instruction copies data from a port to the AL or AX register. If an 8-bit port is read, the data will go to AL. If a 16-bit port is read, the data will go to AX.

The IN instruction has two possible formats, fixed port and variable port. For fixed port type, the 8-bit address of a port is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

IN AL, OC8H Input a byte from port OC8H to AL
IN AX, 34H Input a word from port 34H to AX

For the variable-port form of the IN instruction, the port address is loaded into the DX register before the IN instruction. Since DX is a 16-bit register, the port address can be any number between 0000H and FFFFH. Therefore, up to 65,536 ports are addressable in this mode.

MOV DX, 0FF78H Initialize DX to point to port
IN AL, DX Input a byte from 8-bit port 0FF78H to AL
IN AX, DX Input a word from 16-bit port 0FF78H to AX

The variable-port IN instruction has advantage that the port address can be computed or dynamically determined in the program. Suppose, for example, that an 8086-based computer needs to input data from 10 terminals, each having its own port address. Instead of having a separate procedure to input data from each port, you can write one generalized input procedure and simply pass the address of the desired port to the procedure in DX.

The IN instruction does not change any flag.
OUT – OUT Port, Accumulator

The OUT instruction copies a byte from AL or a word from AX to the specified port. The OUT instruction has two possible forms, fixed port and variable port.

For the fixed port form, the 8-bit port address is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

OUT 3BH, AL Copy the content of AL to port 3BH
OUT 2CH, AX Copy the content of AX to port 2CH

For variable port form of the OUT instruction, the content of AL or AX will be copied to the port at an address contained in DX. Therefore, the DX register must be loaded with the desired port address before this form of the OUT instruction is used.

MOV DX, 0FFF8H Load desired port address in DX OUT DX, AL Copy content of AL to port FFF8H
OUT DX, AX Copy content of AX to port FFF8H The OUT instruction does not affect any flag.

MISCELLANEOUS INSTRUCTIONS

HLT (HALT PROCESSING)

The HLT instruction causes the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The different ways to get the processor out of the halt state are with an interrupt signal on the INTR pin, an interrupt signal on the NMI pin, or a reset signal on the RESET input.

NOP (PERFORM NO OPERATION)

This instruction simply uses up three clock cycles and increments the instruction pointer to point to the next instruction. The NOP instruction can be used to increase the delay of a delay loop. When hand coding, a NOP can also be used to hold a place in a program for an instruction that will be added later. NOP does not affect any flag.

ESC (ESCAPE)

This instruction is used to pass instructions to a coprocessor, such as the 8087 Math coprocessor, which shares the address and data bus with 8086. Instructions for the coprocessor are represented by a 6-bit code embedded in the ESC instruction. As the 8086 fetches instruction bytes, the coprocessor also fetches these bytes from the data bus and puts them in its queue. However, the coprocessor treats all the normal 8086 instructions as NOPs. When 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6-bit code specified in the instruction. In most cases, the 8086 treats the ESC instruction as a NOP. In some cases, the 8086 will access a data item in memory for the coprocessor.

INT – INT TYPE

The term type in the instruction format refers to a number between 0 and 255, which identify the interrupt. When an 8086 executes an INT instruction, it will

- **Decrement the stack pointer by 2 and push the flags on to the stack.**
- **Decrement the stack pointer by 2 and push the content of CS onto the stack.**
- **Decrement the stack pointer by 2 and push the offset of the next instruction after the INT number**

instruction on the stack.
Get a new value for IP from an absolute memory address of 4 times the type specified in the instruction. For an INT 8 instruction, for example, the new IP will be read from address 00020H.
Get a new for value for CS from an absolute memory address of 4 times the type specified in the instruction plus 2, for an INT 8 instruction, for example, the new value of CS will be read from address 00022H.
Reset both IF and TF. Other flags are not affected.

INT 35 New IP from 0008CH, new CS from 0008Eh
INT 3 This is a special form, which has the single-byte code of CCH;
Many systems use this as a break point instruction (Get new IP from 0000CH new CS from 0000EH).

INTO (INTERRUPT ON OVERFLOW)

If the overflow flag (OF) is set, this instruction causes the 8086 to do an indirect far call to a procedure you write to handle the overflow condition. Before doing the call, the 8086 will
Decrement the stack pointer by 2 and push the flags on to the stack.
Decrement the stack pointer by 2 and push CS on to the stack.
Decrement the stack pointer by 2 and push the offset of the next instruction after INTO instruction onto the stack.

Reset TF and IF. Other flags are not affected. To do the call, the 8086 will read a new value for IP from address 00010H and a new value of CS from address 00012H.

IRET (INTERRUPT RETURN)

When the 8086 responds to an interrupt signal or to an interrupt instruction, it pushes the flags, the current value of CS, and the current value of IP onto the stack. It then loads CS and IP with the starting address of the procedure, which you write for the response to that interrupt. The IRET instruction is used at the end of the interrupt service procedure to return execution to the interrupted program. To do this return, the 8086 copies the saved value of IP from the stack to IP, the stored value of CS from the stack to CS, and the stored value of the flags back to the flag register. Flags will have the values they had before the interrupt, so any flag settings from the procedure will be lost unless they are specifically saved in some way.

LOCK – ASSERT BUS LOCK SIGNAL

Many microcomputer systems contain several microprocessors. Each microprocessor has its own local buses and memory. The individual microprocessors are connected together by a system bus so that each can access system resources such as disk drive or memory. Each microprocessor takes control of the system bus only when it needs to access some system resources. The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus while it is in the middle of a critical instruction, which uses the system bus. The LOCK prefix is put in front of the critical instruction. When an instruction with a LOCK prefix executes, the 8086 will assert its external bus controller device, which then prevents any other processor from taking over the system bus. LOCK instruction does not affect any flag.

LOCK XCHG SAMAPHORE, AL
The XCHG instruction requires two bus accesses. The LOCK prefix prevents another processor from

taking control of the system bus between the two accesses.

WAIT – WAIT FOR SIGNAL OR INTERRUPT SIGNAL

When this instruction is executed, the 8086 enters an idle condition in which it is doing no processing. The 8086 will stay in this idle state until the 8086 test input pin is made low or until an interrupt signal is received on the INTR or the NMI interrupt input pins. If a valid interrupt occurs while the 8086 is in this idle state, the 8086 will return to the idle state after the interrupt service procedure executes. It returns to the idle state because the address of the WAIT instruction is the address pushed on the stack when the 8086 responds to the interrupt request. WAIT does not affect any flag. The WAIT instruction is used to synchronize the 8086 with external hardware such as the 8087 Math coprocessor.

XLAT / XLATB – TRANSLATE A BYTE IN AL

The XLATB instruction is used to translate a byte from one code (8 bits or less) to another code (8 bits or less). The instruction replaces a byte in AL register with a byte pointed to by BX in a lookup table in the memory. Before the XLATB instruction can be executed, the lookup table containing the values for a new code must be put in memory, and the offset of the starting address of the lookup table must be loaded in BX. The code byte to be translated is put in AL. The XLATB instruction adds the byte in AL to the offset of the start of the table in BX. It then copies the byte from the address pointed to by (BX + AL) back into AL. XLATB instruction does not affect any flag.

8086 routine to convert ASCII code byte to EBCDIC equivalent: ASCII code byte is in AL at the start, EBCDIC code in AL after conversion.

```
MOV BX, OFFSET EBCDIC      Point BX to the start of EBCDIC table in DS XLATB
Replace ASCII in AL with EBCDIC from table.
```

8086 ASSEMBLER DIRECTIVES

SEGMENT

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment. For example, the statement CODE SEGMENT indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directive are used to "bracket" a logical segment containing code of data.

Additional terms are often added to a SEGMENT directive statement to indicate some special way in which we want the assembler to treat the segment. The statement CODE SEGMENT WORD tells the assembler that we want the content of this segment located on the next available word (even address) when segments ate combined and given absolute addresses. Without this WORD addition, the segment will be located on the next available paragraph (16-byte) address, which might waste as much as 15 bytes of memory. The statement CODE SEGMENT PUBLIC tells the assembler that the segment may be put together with other segments named CODE from other assembly modules when the modules are linked together.

ENDS (END SEGMENT)

This directive is used with the name of a segment to indicate the end of that logical segment.

CODE SEGMENT Start of logical segment containing code instruction statements CODE ENDS End of segment named CODE

END (END PROCEDURE)

The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statements after an END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive.

ASSUME

The ASSUME directive is used tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE. The statement ASSUME DS: DATA tells the assembler that for any program instruction, which refers to the data segment, it should use the logical segment called DATA.

DB (DEFINE BYTE)

The DB directive is used to declare a byte type variable, or a set aside one or more storage locations of type byte in memory.

PRICES DB 49H, 98H, 29H Declare array of 3 bytes named PRICE and initialize them with specified values.

NAMES DB "THOMAS" Declare array of 6 bytes and initialize with ASCII codes for the letters in THOMAS.

TEMP DB 100 DUP (?) Set aside 100 bytes of storage in memory and give it the name TEMP. But leave the 100 bytes un-initialized.

PRESSURE DB 20H DUP (0) Set aside 20H bytes of storage in memory, give it the name PRESSURE and put 0 in all 20H locations.

DD (DEFINE DOUBLE WORD)

The DD directive is used to declare a variable of type double word or to reserve memory locations, which can be accessed as type double word. The statement ARRAY DD 25629261H, for example, will define a double word named ARRAY and initialize the double word with the specified value when the program is loaded into memory to be run. The low word, 9261H, will be put in memory at a lower address than the high word.

DQ (DEFINE QUADWORD)

The DQ directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory. The statement BIG_NUMBER DQ 243598740192A92BH, for example, will declare a variable named BIG_NUMBER and initialize the 4 words set aside with the specified number when the program is loaded into memory to be run.

DT (DEFINE TEN BYTES)

The DT directive is used to tell the assembler to declare a variable, which is 10 bytes in length or to reserve 10 bytes of storage in memory. The statement PACKED_BCD DT 11223344556677889900 will declare an array named PACKED_BCD, which is 10 bytes in length. It will initialize the 10 bytes with the values 11, 22, 33, 44, 55, 66, 77, 88, 99, and 00 when the program is loaded into memory to be run. The statement RESULT DT 20H DUP (0) will declare an array of 20H blocks of 10 bytes each and initialize all 320 bytes to 00 when the program is loaded into memory to be run.

DW (DEFINE WORD)

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement MULTIPLIER DW 437AH, for example, declares a variable of type word named MULTIPLIER, and initialized with the value 437AH when the program is loaded into memory to be run.

WORDS DW 1234H, 3456H Declare an array of 2 words and initialize them with the specified values.
STORAGE DW 100 DUP (0) Reserve an array of 100 words of memory and initialize all 100 words with 0000. Array is named as STORAGE.
STORAGE DW 100 DUP (?) Reserve 100 word of storage in memory and give it the name STORAGE, but leave the words un-initialized.

EQU (EQUATE)

EQU is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it replaces the name with the value or symbol you equated with that name. Suppose, for example, you write the statement FACTOR EQU 03H at the start of your program, and later in the program you write the instruction statement ADD AL, FACTOR. When the assembler codes this instruction statement, it will code it as if you had written the instruction ADD AL, 03H.

CONTROL EQU 11000110 B Replacement MOV AL, CONTROL Assignment
DECIMAL_ADJUST EQU DAA Create clearer mnemonic for DAA ADD AL, BL Add BCD numbers
DECIMAL_ADJUST Keep result in BCD format

LENGTH

LENGTH is an operator, which tells the assembler to determine the number of elements in some named data item, such as a string or an array. When the assembler reads the statement MOV CX, LENGTH STRING1, for example, will determine the number of elements in STRING1 and load it into CX. If the string was declared as a string of bytes, LENGTH will produce the number of bytes in the string. If the string was declared as a word string, LENGTH will produce the number of words in the string.

OFFSET

OFFSET is an operator, which tells the assembler to determine the offset or displacement of a named data item (variable), a procedure from the start of the segment, which contains it. When the assembler reads the statement MOV BX, OFFSET PRICES, for example, it will determine the offset of the variable PRICES from the start of the segment in which PRICES is defined and will load this value into BX.

PTR (POINTER)

The PTR operator is used to assign a specific type to a variable or a label. It is necessary to do this in any instruction where the type of the operand is not clear. When the assembler reads the instruction INC [BX], for example, it will not know whether to increment the byte pointed to by BX. We use the PTR operator to clarify how we want the assembler to code the instruction. The statement INC BYTE PTR [BX] tells the assembler that we want to increment the byte pointed to by BX. The statement INC WORD PTR [BX] tells the assembler that we want to increment the word pointed to by BX. The PTR operator assigns the type specified before PTR to the variable specified after PTR.

We can also use the PTR operator to clarify our intentions when we use indirect Jump instructions. The statement JMP [BX], for example, does not tell the assembler whether to code the instruction for a near jump. If we want to do a near jump, we write the instruction as JMP WORD PTR [BX]. If we want to do a far jump, we write the instruction as JMP DWORD PTR [BX].

EVEN (ALIGN ON EVEN MEMORY ADDRESS)

As an assembler assembles a section of data declaration or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The EVEN directive tells the assembler to increment the location counter to the next even address, if it is not already at an even address. A NOP instruction will be inserted in the location incremented over.

```
DATA SEGMENT
SALES DB 9 DUP (?)    Location counter will point to 0009 after this instruction. EVEN
                      Increment location counter to 000AH
INVENTORY DW 100 DUP (0)    Array of 100 words starting on even address for quicker read
DATA ENDS
```

PROC (PROCEDURE)

The PROC directive is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive, the term near or the term far is used to specify the type of the procedure. The statement DIVIDE PROC FAR, for example, identifies the start of a procedure named DIVIDE and tells the assembler that the procedure is far (in a segment with different name from the one that contains the instructions which calls the procedure). The PROC directive is used with the ENDP directive to "bracket" a procedure.

ENDP (END PROCEDURE)

The directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The directive, together with the procedure directive, PROC, is used to "bracket" a procedure.

```
SQUARE_ROOT PROC    Start of procedure. SQUARE_ROOT ENDP    End of procedure.
```

ORG (ORIGIN)

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The location counter is automatically set to 0000 when assembler starts reading a segment. The ORG directive allows you to set the location counter to a desired value at any point in the program. The statement ORG 2000H tells the assembler to set the location counter to 2000H, for example.

A "$" it often used to symbolically represent the current value of the location counter, the $ actually represents the next available byte location where the assembler can put a data or code byte. The $ is often used in ORG statements to tell the assembler to make some change in the location counter relative to its current value. The statement ORG $ + 100 tells the assembler increment the value of the location counter by 100 from its current value.

NAME

The NAME directive is used to give a specific name to each assembly module when programs consisting of several modules are written.

LABEL

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to be keep track of how many bytes it is from the start of a segment at any time. The LABEL directive is used to give a name to the current value in the location counter. The LABEL directive must be followed by a term that specifics the type you want to associate with that name. If the label is going to be used as the destination for a jump or a call, then the label must be specified as type near or type far. If the label is going to be used to reference a data item, then the label must be specified as type byte, type word, or type double word. Here's how we use the LABEL directive for a jump address.

ENTRY_POINT LABEL FAR Can jump to here from another segment
NEXT: MOV AL, BL Can not do a far jump directly to a label with a colon The following example shows how we use the label directive for a data reference.
STACK_SEG SEGMENT STACK
DW 100 DUP (0) Set aside 100 words for stack
STACK_TOP LABEL WORD Give name to next location after last word in stack STACK_SEG ENDS

To initialize stack pointer, use MOV SP, OFFSET STACK_TOP.

EXTRN

The EXTRN directive is used to tell the assembler that the name or labels following the directive are in some other assembly module. For example, if you want to call a procedure, which in a program module assembled at a different time from that which contains the CALL instruction, you must tell the assembler

that the procedure is external. The assembler will then put this information in the object code file so that the linker can connect the two modules together. For a reference to externally named variable, you must specify the type of the variable, as in the statement EXTRN DIVISOR: WORD. The statement EXTRN DIVIDE: FAR tells the assembler that DIVIDE is a label of type FAR in another assembler module. Name or labels referred to as external in one module must be declared public with the PUBLIC

directive in the module in which they are defined.

PROCEDURE SEGMENT
EXTRN DIVIDE: FAR Found in segment PROCEDURES PROCEDURE ENDS

PUBLIC

Large program are usually written as several separate modules. Each module is individually assembled, tested, and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared PUBLIC in the module in which it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules. An example is the statement PUBLIC DIVISOR, DIVIDEND, which makes the two variables DIVISOR and DIVIDEND available to other assembly modules.

SHORT

The SHORT operator is used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction in the program. The destination must in the range of –128 bytes to +127 bytes from the address of the instruction after the jump. The statement JMP SHORT NEARBY_LABEL is an example of the use of SHORT.

TYPE

The TYPE operator tells the assembler to determine the type of a specified variable. The assembler actually determines the number of bytes in the type of the variable. For a byte-type variable, the assembler will give a value of 1, for a word-type variable, the assembler will give a value of 2, and for a double word-type variable, it will give a value of 4. It can be used in instruction such as ADD BX, TYPE-WORD-ARRAY, where we want to increment BX to point to the next word in an array of words.

GLOBAL (DECLARE SYMBOLS AS PUBLIC OR EXTRN)

The GLOBAL directive can be used in place of a PUBLIC directive or in place of an EXTRN directive. For a name or symbol defined in the current assembly module, the GLOBAL directive is used to make the symbol available to other modules. The statement GLOBAL DIVISOR, for example, makes the variable DIVISOR public so that it can be accessed from other assembly modules.

INCLUDE (INCLUDE SOURCE CODE FROM FILE)

This directive is used to tell the assembler to insert a block of source code from the named file into the current source module.

80186 architecture

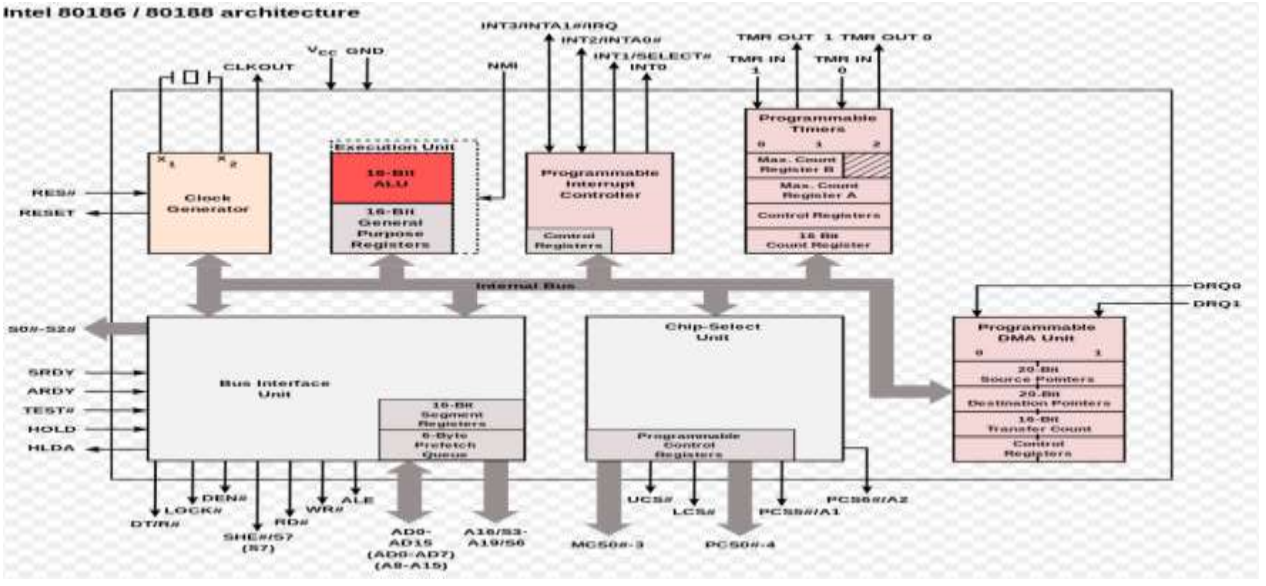


Fig 1.47 80186 block diagram

80186 Features
The 80186 double the performance of 8086.
80186 = 8086 + several additional chips
contains 16 bit data bus and 20 bit address bus
The total addressable memory size is 1MB.
The instruction set is upward compatible with the 8086/88 family.

Functional Blocks are
Clock generator
Execution unit
Two independent high-speed DMA channels
Programmable interrupt controller
Three programmable 16-bit timers

Bus interface unit
Programmable memory and peripheral chip-select logic
Execution Unit
- **Contains 16 bit ALU and a set of general purpose registers**

Chip selection unit
- **Used to select memory and I/O**

Bus Interface Unit
- **It provides various functions, including generation of the memory and I/O addresses for the transfer of data between outside the CPU, and the Execution Unit.**
- **Prefetch queue is used for prefetching**

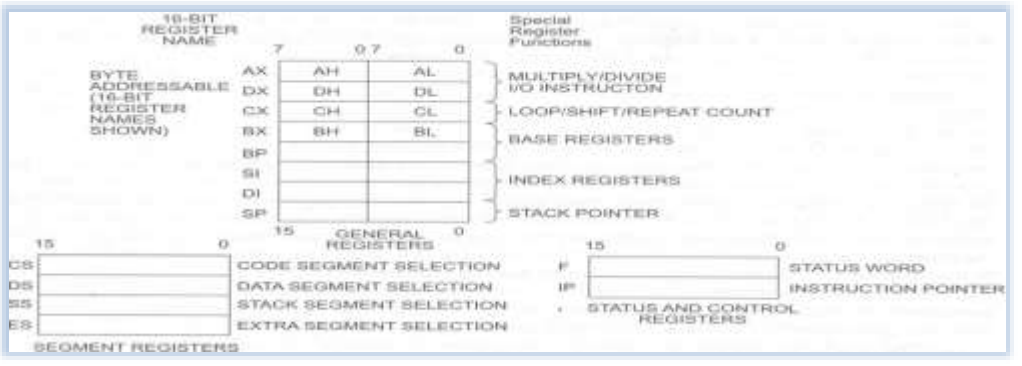


Fig 1.48 programming model-80186

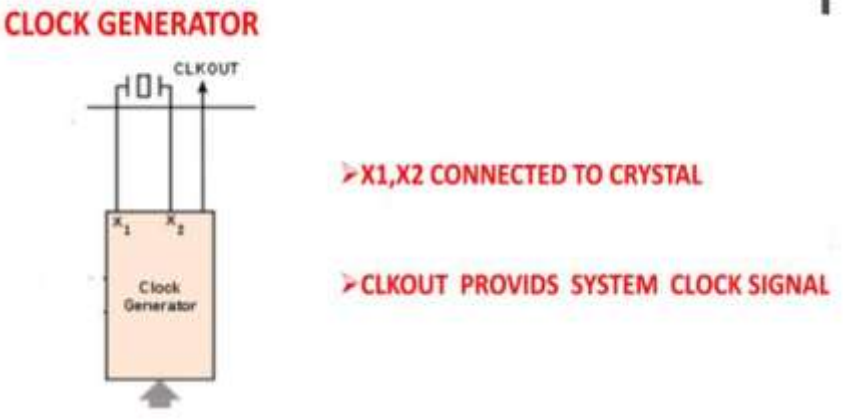


Fig 1.49 clock generator

Clock Generator

- On-chip clock generator / crystal oscillator circuit.
- a crystal connected at the 80186 X1 ,X2 pins is divided by 2 internally

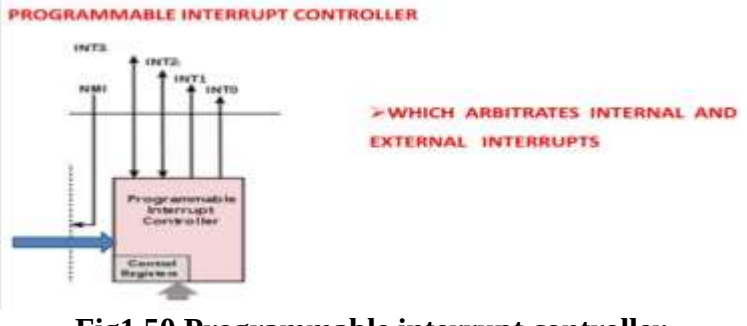


Fig1.50 Programmable interrupt controller

Allows internal and external interrupts and controls up to two external 8259A PICs.
Accepts interrupts only in the master mode. control register is used for controlling the interrupts

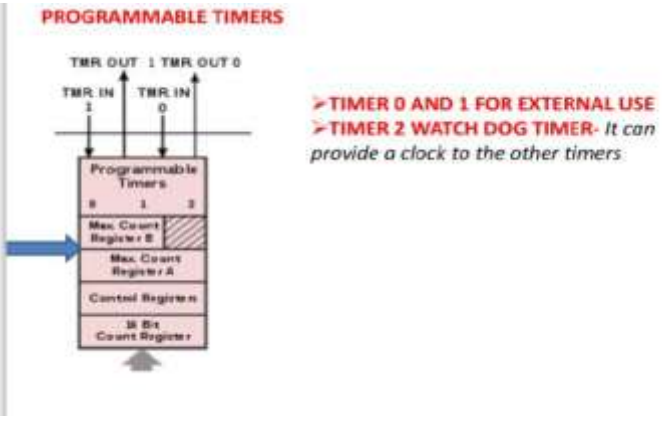


Fig 1.51 Programmable timers

Timers

- **contains three fully programmable 16-bit timers**
- **The timers 0 and 1 programmed to count external events and driven by either the master clock of the 80186 or by an external clock**
- **Timer 2 is for internal events and clocked by the master clock.**

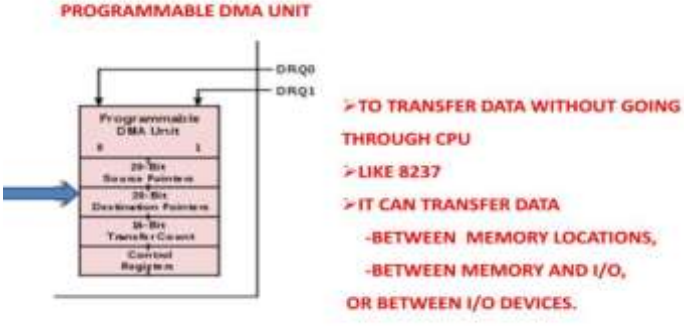


Fig 1.52 Programmable DMA unit

Programmable DMA Unit

- **contains two DMA channels,**
- **Data can be transferred either by the byte or by 16-bit words.**
- **Each DMA channel contains**

i. **20-bit source and destination pointers: used to address the source and destination of the data transferred.**
ii. **16-bit transfer count register: contains a number of DMA transfers to be performed.**
iii. **16-bit control register: specifies information such as data rate, MA operation etc.:**

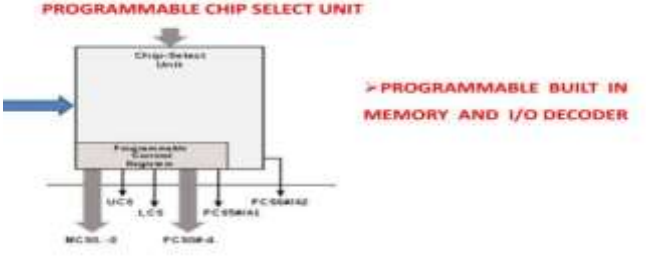


Fig 1.53 Programmable chip select unit

The 10 additional instructions that the 80186 has are as follows:

ENTER	— Enter a procedure
LEAVE	— Leave a procedure
BOUND	— Check if an array index in a register is in range of array
INS	— Input string byte or string word
OUTS	— Output string byte or string word
PUSHA	— Push all registers on stack
POPA	— Pop all registers off stack
PUSH immediate	— Push immediate number on stack
IMUL destination register,	— Immediate x source to destination source, immediate
SHIFT/ROTATE	— Shift register or memory contents specified immediate destination, immediate number of times

Fig 1.54 Additional instructions

TEXT / REFERENCE BOOKS

1. **Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing", 2nd Edition, Pearson Education, Inc., 2004**
2. **Barry B.Brey , "The Intel Microprocessors 8086/8088, 8086, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, Architecture,**
3. **Programming and interfacing", Prentice Hall of India Private Limited, New Delhi, 2003**
4. **Alan Clements, "The Principles of computer Hardware", Oxford University Press, 3rd Edition, 2003**
5. **John Paul Shen, Mikko H.Lipasti, "Modern Processor Design", Tata McGraw Hill, 2006**

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMMUNICATION ENGINEERING

UNIT - 2
ADVANCED MICROPROCESSORS – SEC1601

- **UNIT 2: INTRODUCTION TO 80286,80386 & 80486**

- **Introduction to 80286 – Architecture, Real address mode & protected virtual address mode.**
- **80386 Microprocessor - Architecture, Pins & Signals, Memory System Registers, 80386 operating modes -Paging Technique, Protected Mode Operation**
- **Intel 80486 Architecture**
- **Comparison of Microprocessors (8086 - 80286 - 80386 - 80486)**

INTRODUCTION

We have used the 8088/8086 microprocessor, because it is the simplest member of this family of Intel processors and is therefore for a good starting point. now it the time to looking at the evolutionary offspring of 8086 . to give you an overview, here are a few brief notes about the members of the family.

the 80286 , another 16-bit enhancement of the 8086 we introduced as the same time as the 80186 instead of the integrated peripherals of the 80186, it has virtual memory management circuitry, production circuitry, and a 16MB addressing capability. the 80286 was the first family member designed specifically for use as the CPU in the multi-user microcomputer.

MULTIUSER/MULTITASKING OPERATING SYSTEM CONCEPTS

The basic principle of a timeshare system is that the CPU runs one user's program for a few milliseconds, that runs the next users program for a few milliseconds, and so on until if the user have had to turn. it cycles through the users over and over, fast enough that each users seem to have the complete attention of the CPU . an operating system which coordinates the action of the intersystem such as this is referred to as a multi-user operating system. the program or section of a program of each user is revered to as a task or process, so a multi-user operating system's also commonly referred to as multitasking . multitasking operating system are also used to control the operating system of the machine in industrial manufacturing environments. networking is an example of the multitasking/multi-user operating system .

SCHEDULING TSR PROGRAMS AND DOS

MS Dos is designed as a single user, single-tasking operating system. this means that dos can usually execute only one program at the time . the only exception to this in the basic this dos in the print program. print.com you may have noticed that when you execute the print command , dos returned a prompt and allow you to enter another command before the printing is complete . the print program starts printing the specified file and the returned execution to dos. however, the print program continue to monitor dos execution when dos is sitting in a loop sitting in a loop waiting for a user command or some other event , the print program borrows the CPU for a short time and sends more data to the printer. if then return execution to the interrupted dos loop .

The dos print command then is a limited form of multitasking. products sunk as Orlando's sidekick use this same technique in dos system to provide popup menus of useful function such as a calculator. the first time you run in a such as sidekick. it is loaded into memory as other programs are. however , unlike other programs sidekick is designed so that when you terminate the program. it says "resident" in memory . you can executed the program and popup the menu again by a simply pressing some hot key combination such as Ctrl+Alt program which work in this way are called terminate and-stay-resident or TSR programs , because TSRs are so common in the PC world , we through you might find it interesting to see hoe they work before we get into discussions of the scheduling technique used in full fledged multitasking operating system.

When you boot up dos, the basic 640 KB of ram are set up as shown in figure a starting from absolute address 00000 , the first section of the ram is reserved for interrupt vector . the main part of the dos program is loaded into the next -higher section of ram .

After this come device drivers such as ANSI.SYS,MOUSE.SYS etc the dos command processor program , command.com , gets loaded into RAM at boot time. this program , with processes user command and executes programs , has two parts . the resident part the command processor is loaded in memory just above the device drivers and the transient part is loaded in at the very top of RAM . when you tell DOS to execute a exeprogram, the program will be loaded into the transient program area of RAM and , if necessary into the RAM where the transient part of the command processor was loaded . (the transient part of the command processor will be reloaded when the program terminate).

Normally ,when a program terminates all the transient program area is dealocated , so that another program can be loaded in it to be run . TSR program , however, are terminated in a special way so that they are left resident in memory as shown in figure b . the terminated program . when another program is loaded to be run , it is put in RAM above the TSRs.

The need of a multitasking multi-user operating system include environment preservation during task switches , operating system and user protection, and virtual memory

management. The Intel 80286 was the first 8086 family processor designed to make the implementation of these features relatively closer.

80286 ARCHITECTURE, SIGNALS AND SYSTEM CONNECTIONS

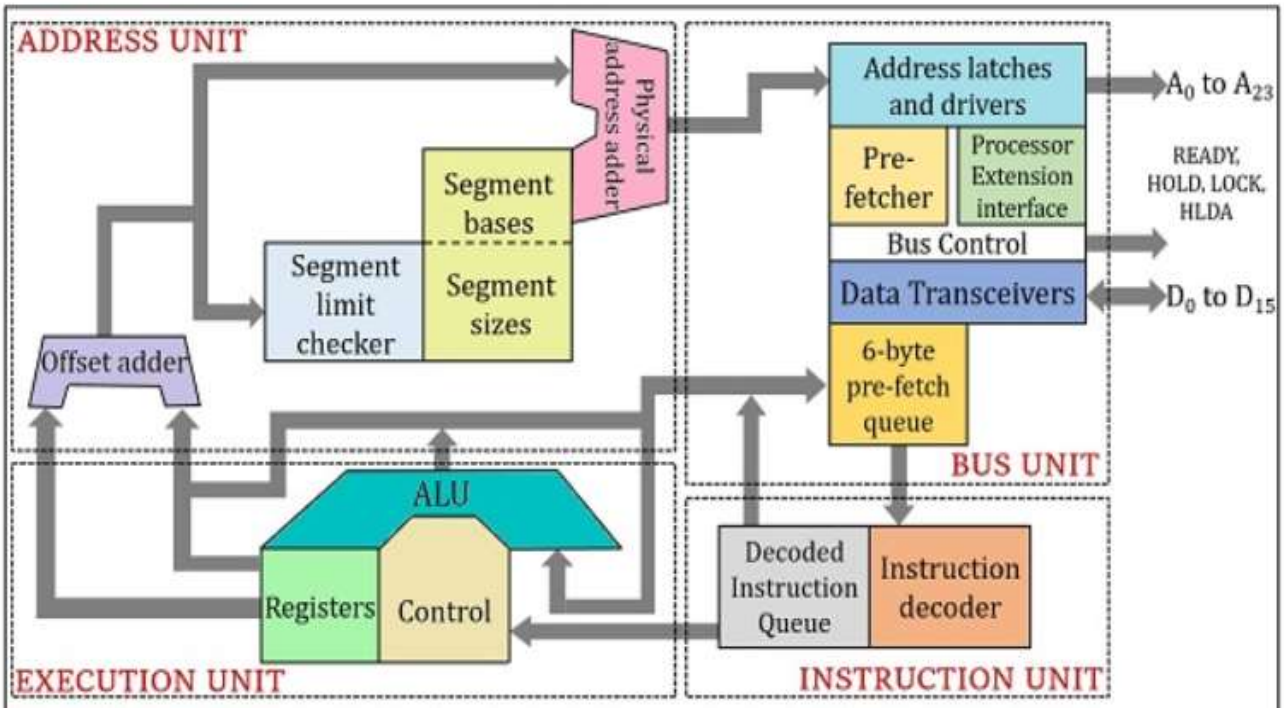


Fig 2.1: Architecture diagram -80286

BUS UNIT

The bus unit (BU) in the device performs all necessary memory and I/O reads and writes, prefects instruction bytes, and controls transfer of data to and from the microprocessor extension devices such as 80287 math coprocessor.

INSTRUCTION UNIT

The instruction Unit(IU) fully decodes up to three perfected instructions and holds them in a queue, where the execution unit can access them. This is a further example of how

modern processors keeps several instructions 'in the pipeline' instead of waiting to finish one instruction before fetching the next.

EXECUTION UNIT

The execution unit(EU) uses its 16-bit ALU to execute instructions it receives from the instruction unit. When operating in its real address mode, the 80286 register set is the same as that of an 8086 except for the addition of a 16-bit machine status word (MSW) register.

ADDRESS UNIT

The address unit (AU) computes the physical address what will be sent out to memory or I/ by the BU. The 80286 can operate in one of two memory address modes, real address or protected virtual address mode. If the 80286 is operating in the real address mode, the address unit computes addresses using a segment base and an offset just as the 80286 does. The familiar CS,DS,SS and ES registers are used to hold the base addresses for the segment currently in use. The maximum physical address space in this mode is 1MB, just as 8086.

If an 80286 is operating in its protected virtual address mode(protected mode), the address unit functions as a complete MMU. In this address mode the 80286 uses all the 24 address lines to access up to 16 MB of physical memory. In protected mode it also provides up to a gigabyte of virtual memory.

80286 SIGNALS AND SYSTEM CONNECTIONS

The 80286 has a 16-bit data bus and a 24-bit no multiplexed address bus. The 24 bit address bus allows the processor to access 16 MB of physical memory when operating in the protected mode. Memory hardware for the 80286 is set up as an odd bank and an even bank, just as in the 8086.

The even bank will be enabled when AO is low, and the odd bank will be enabled when BHE will be Low. External buffers are used in both the address and data bus. From a control standpoint, the 80286 functions similarly to an 8086 operating in maximum mode. Status signals SO, SI and M/IO are decoded by an external 82288 bus controller to produce the control bus, read, write and interrupt-acknowledge signals.

The HOLD, HLDA, INTR, INTA, (NMI), READY, LOCK and RESET pins functions basically the same as they do in 8086. An external 82284 clock generator is used to produce a clock signal for the 80286 and to synchronize RESET and READY signals.

The final four signal pins are used to interface with processor Extensions(coprocessors) such as the 80287 math coprocessor. The processor extension request (PEREQ) input pin

will be asserted by a coprocessor to tell the 80286 to perform a data transfer to or from memory for it.

When the 80286 gets around to do the transfer, it asserts the process extension acknowledge (PEACK) signal to the coprocessor to let it know the data transfer has started. Data transfers are done through the 80286 in this way so that the coprocessor uses the protection and virtual memory capability of the MMU in the 80286.

The BUSY signal input does on the 80286 functions the same as the TEST input does on the 8086. When the 80286 executes a WAIT instruction , it will remain in a WAIT loop until it finds the BUSY signal from the coprocessor high. If a coprocessor finds some error during processing, it will assert the ERROR input of the 80286. This will cause the 80286 automatically da a type 16H interrupt call.

80286 REAL ADDRESS MODE OPERATIONS

After 80286 is reset, it starts executing in its real address mode. This mode is referred to as real because physical memory addresses are produced by directly adding an offset to a segment base. In this mode, the 80286 can address 1MB of physical address and functions essentially as a "souped-up" 8086.

When operating in real address mode, the interrupt vector table of the 80286 is located in the first 1KB of memory and the response to an interrupt is same as 8086.The 80286 has several additional built in interrupt types.

The 80186 and the later processors separate interrupts in two categories, interrupts and exceptions. Asynchronous external events which affect the processor through the INRT and NMI inputs are referred to as interrupts. An exception type interrupt is generated by some error condition that occurred during the execution of an instruction. Software interrupts are produced by the INT n instruction are classified as exceptions, because they are synchronous with the processor.

Exceptions are further divided into faults and traps. Faults are exceptions that are Exceptions that detected and signaled before the faulting instructions is executed. Traps are exceptions which are reported after the instruction which caused the exception executes.

80286 PROTECTED –MODE OPERATION

On an 80286 based system running under MSDOS or a similar OS, the 80286 is left in real address mode because current versions of DOS are not designed to take advantage of the protected mode features of 80286. If an 80286 based system is running OS such as Microsoft OS/2, which uses the protected mode, the real mode will be used to initialize

peripheral devices, load the main part of the OS into the memory from the disk, load some registers , enable interrupts set up descriptor tables and switch the processor to the protected mode.

The first step is switching to the protected mode is to set the protection enable bit in the machine status word (MSW) register in the 80286.Bits 1,2 and 3 of the MSW are for the most part to indicate whether a processor extension (coprocessor) is present in the system or not. Bit 0 of the MSW is used to switch the 80286 into protected mode. To change bits in the MSW you load the desired word in a register or memory location and execute the load machine status word(LMSW) instruction .The initial step to get the 80286 operating in protected mode is to execute an intersegment jump to the start of the main system program.

ADDITIONAL INSTRUCTIONS

The 80286 has even more instructions than its predecessors. These extra instructions Control the virtual memory system through manager of the 80286.These instructions are as follows

INSTRUCTIONS PURPOSE

CLTS Clears the task-switched flag bit

LDGT Loads the global descriptor table register

SGDT Stores the global descriptor table register

LIDT Loads the interrupt descriptor table register

SIDT Stores the interrupt descriptor table register

LLDT Loads the local descriptor table register

SLDT Stores the interrupt descriptor table register

LMSW Loads the machine status word

SMSW Stores the machine status word

LAR Loads the access rights

LSL Loads the segment limit

SAR Stores the access rights

ARPL Adjusts the requested privilege level

VERR Verifies a read access

VERW Verifies a write access

From a software standpoint the 80286 was designed to be upward compatible from the 8086 so that the huge amount of software developed for the 8086/8088 could be easily transported to the 80286.The instruction set of the 80286 and later processors are "supersets" of the 8086 instructions. The new and enhanced instructions in the 80286 instructions is as follows

Real or protected mode instructions

INS Input String

OUTS Output String

PUSHA Push eight general –purpose

registers on the stack.

POPA Pop eight general –purpose registers

on the stack.

PUSH immediate Push immediate number on stack

SHIFT/ROTATE destination, immediate Shift or rotate destination register or

memory location specified number of

bit positions.

IMUL destination, immediate Signed multiply destination by

immediate number.

IMUL destination, multiplicand, immediate Signed multiply, result in specified

Multiplier destination.

ENTER Set up stack in procedure.Saves BP,

points BP to TOS and allocates

stack space for local variables.

THE VIRTUAL MEMORY MACHINE

A virtual memory machine is a machine that maps a larger memory space(1 G byte for the 80286) into a much smaller physical memory space(16M bytes for 80286). This allows a very large system to execute in smaller physical memory systems. This is accomplished by spooling the data and programs between the fixed disk memory system & the physical memory. Addressing a 1G-byte memory system is accomplished by the description in the 80286 microprocessor. Each 80286 description describes a 64 k-byte memory segment and the 80286 allows 16k descriptions. This (64k x 16k) allows a maximum of 1G byte of memory to be addressed by the system.

ARCHITECTURE OF 80386

- **The Internal Architecture of 80386 is divided into 3 sections.**
- **Central processing unit**
- **Memory management unit**
- **Bus interface unit**
- **Central processing unit is further divided into Execution unit and Instruction unit**
- **Execution unit has 8 General purpose and 8 Special purpose registers which are either used for handling data or calculating offset addresses.**

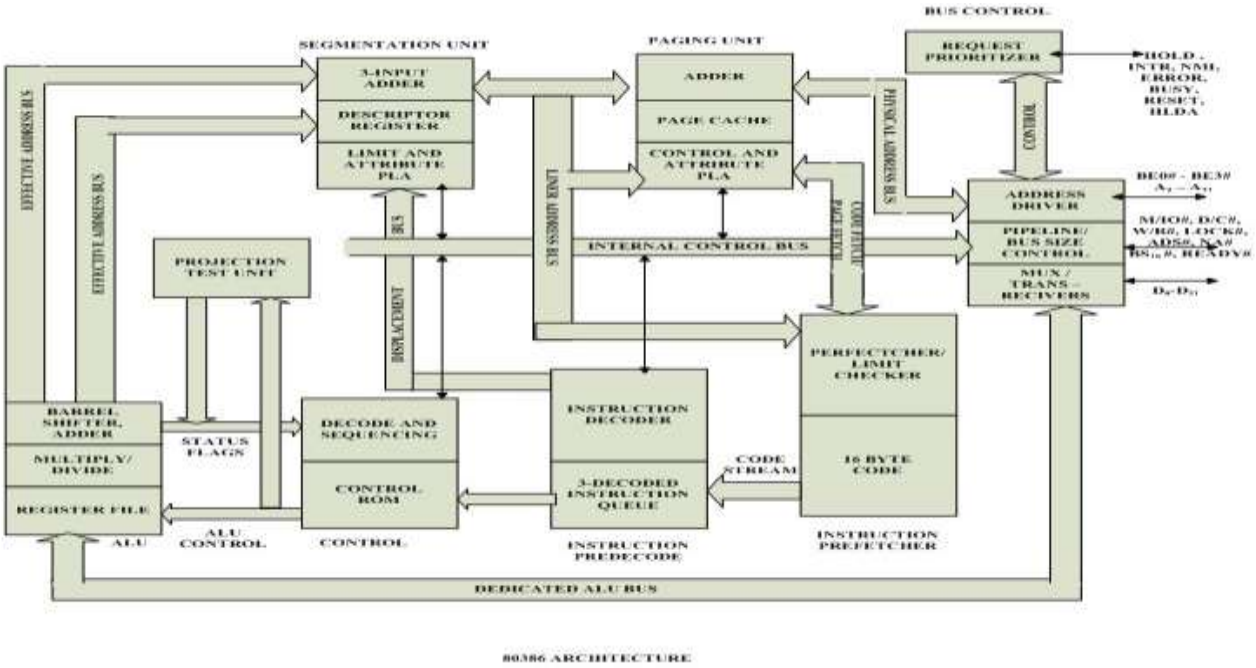


Fig 2.2. 80386 block diagram

- **The Instruction unit decodes the opcode bytes received from the 16-byte instruction code queue and arranges them in a 3- instruction decoded instruction queue. After decoding them pass it to the control section for deriving the necessary control signals. The barrel shifter increases the speed of all shift and rotate operations. The multiply / divide logic implements the bit-shift-rotate algorithms to complete the operations in minimum time. Even 32- bit multiplications can be executed within one microsecond by the multiply / divide logic. The Memory management unit consists of a Segmentation unit and a Paging unit. Segmentation unit allows the use of two address components, viz. segment and offset for relocability and sharing of code and data.**

- **Segmentation unit allows segments of size 4Gbytes at max.**
- **The Paging unit organizes the physical memory in terms of pages of 4kbytes size each.**
- **Paging unit works under the control of the segmentation unit, i.e. each segment is further divided into pages. The virtual memory is also organizes in terms of segments and pages by the memory management unit.**
- **The Segmentation unit provides a 4 level protection mechanism for protecting and isolating the system code and data from those of the application program.**
- **Paging unit converts linear addresses into physical addresses.The control and attribute PLA checks the privileges at the page level. Each of the pages maintains the paging information of the task. The limit and attribute PLA checks segment limits and attributes at segment level to avoid invalid accesses to code and data in the memory segments. The Bus control unit has a prioritizer to resolve the priority of the various bus requests. This controls the access of the bus. The address driver drives the bus enable and address signal $A_0 - A_{31}$. The pipeline and dynamic bus sizing unit handle the related control signals. The data buffers interface the internal data bus with the system bus.**

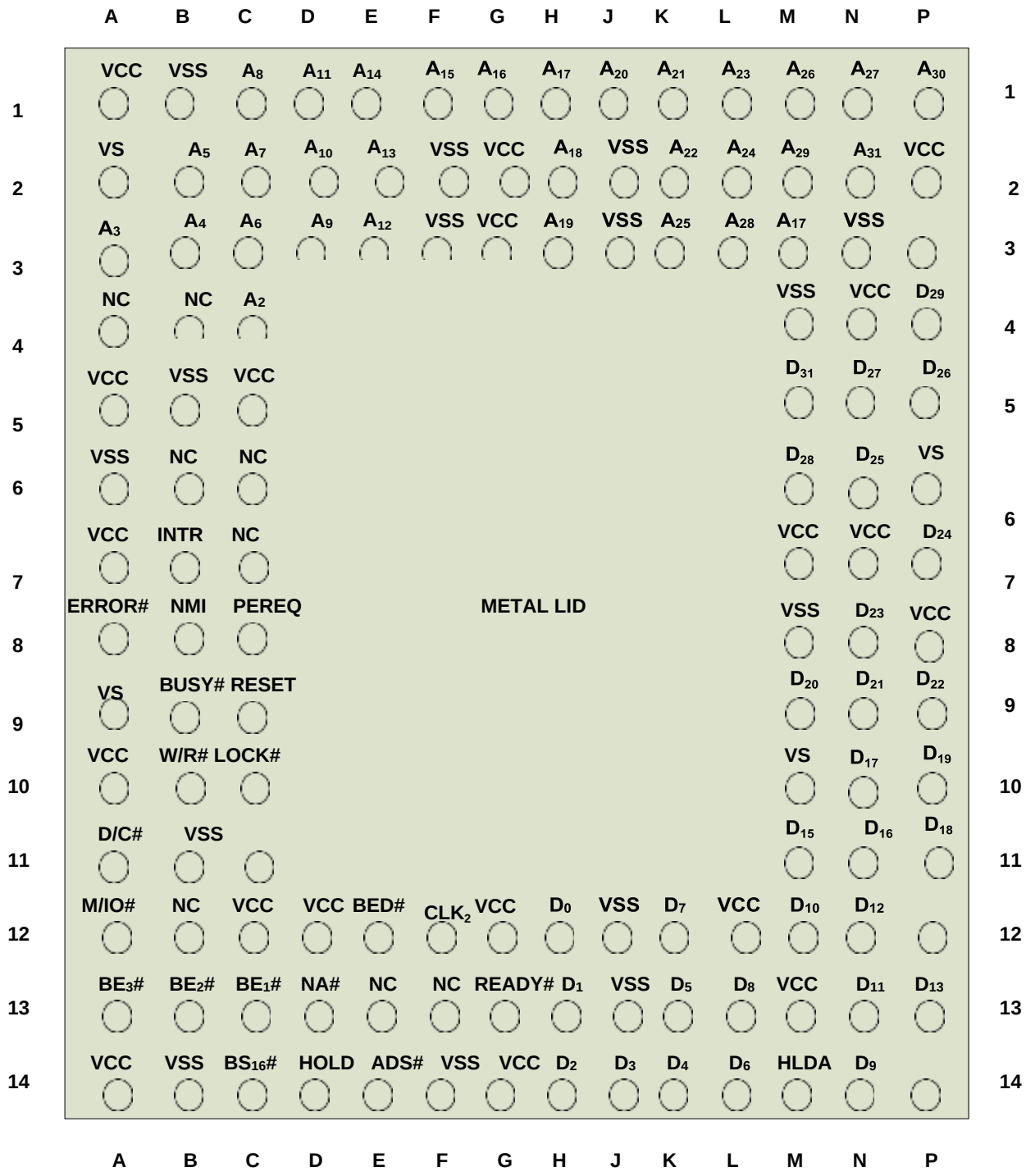


Fig 2.3: Pin diagram

Signal Descriptions of 80386

•CLK_2 :The input pin provides the basic system clock timing for the operation of 80386.

•D_0 – D_{31}:These 32 lines act as bidirectional data bus during different access cycles.

•A_{31} – A_2: These are upper 30 bit of the 32- bit address bus.

- **$\overline{BE0}$ to $\overline{BE3}$: The 32- bit data bus supported by 80386 and the memory system of 80386 can be viewed as a 4- byte wide memory access mechanism. The 4 byte enable lines $\overline{BE0}$ to $\overline{BE3}$, may be used for enabling these 4 blanks. Using these 4 enable signal lines, the CPU may transfer 1 byte / 2 / 3 / 4 byte of data simultaneously.**
- **ADS#: The address status output pin indicates that the address bus and bus cycle definition pins(W/R#, D/C#, M/IO#, BE_0# to BE_3#) are carrying the respective valid signals. The 80383 does not have any ALE signals and so this signals may be used for latching the address to external latches.**
- **READY#: The ready signals indicates to the CPU that the previous bus cycle has been terminated and the bus is ready for the next cycle. The signal is used to insert WAIT states in a bus cycle and is useful for interfacing of slow devices with CPU.**

•VCC: These are system power supply lines.

•VSS: These return lines for the power supply.

- **BS_{16}#: The bus size – 16 input pin allows the interfacing of 16 bit devices with the 32 bit wide 80386 data bus. Successive 16 bit bus cycles may be executed to read a 32 bit data from a peripheral.**
- **HOLD: The bus hold input pin enables the other bus masters to gain control of the system bus if it is asserted.**
- **HLDA: The bus hold acknowledge output indicates that a valid bus hold request has been received and the bus has been relinquished by the CPU.**
- **BUSY#: The busy input signal indicates to the CPU that the coprocessor is busy with the allocated task.**
- **ERROR#: The error input pin indicates to the CPU that the coprocessor has encountered an error while executing its instruction.**
- **PEREQ: The processor extension request output signal indicates to the CPU to fetch a data word for the coprocessor.**
- **INTR: This interrupt pin is a maskable interrupt, that can be masked using the IF of the flag register.**
- **NMI: A valid request signal at the non-maskable interrupt request input pin internally generates a non- maskable interrupt of type2.**
- **RESET: A high at this input pin suspends the current operation and restart the execution from the starting location.**
- **N / C : No connection pins are expected to be left open while connecting the 80386 in the circuit.**

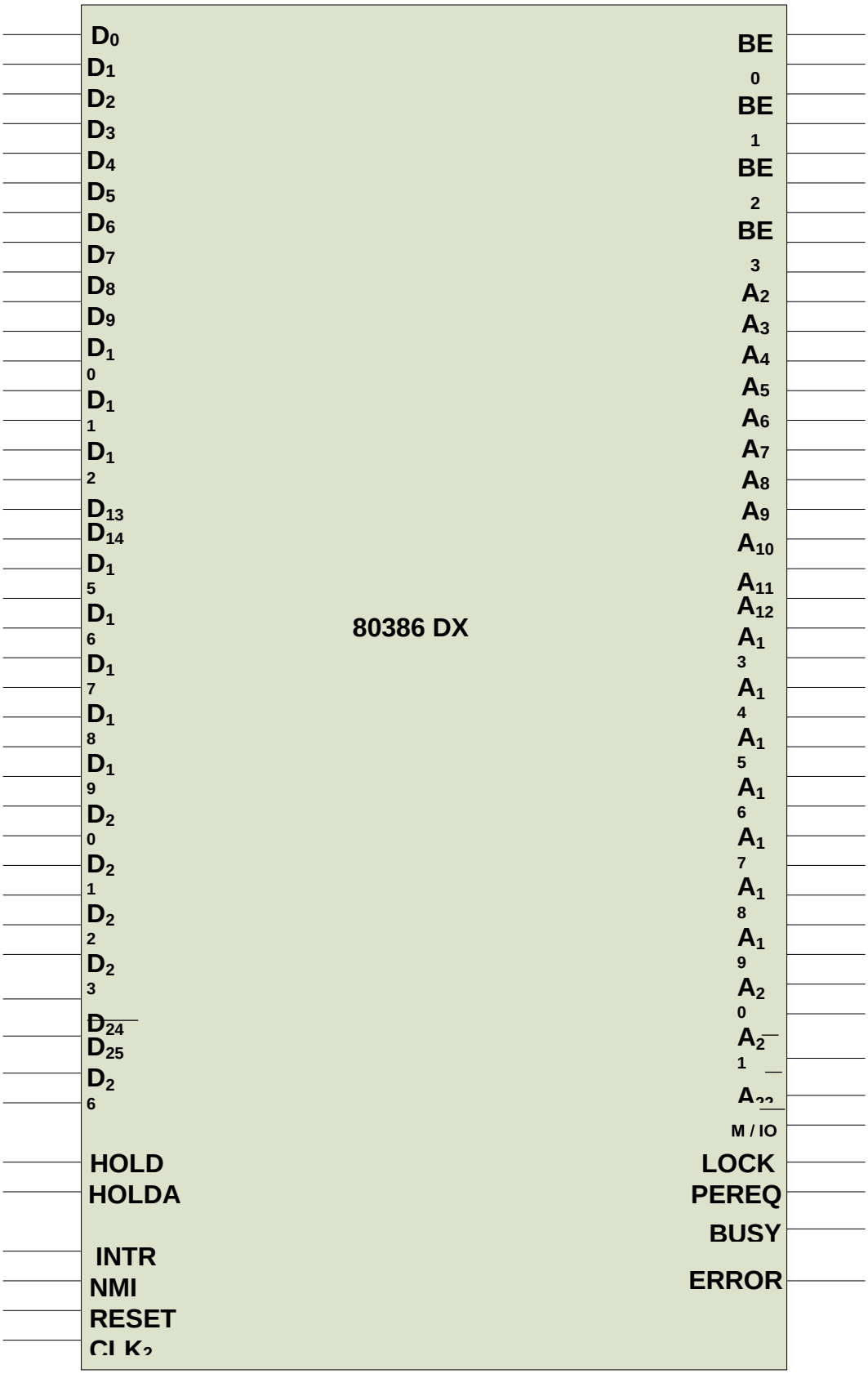


Fig 2.4 PIN DIAGRAM-80386

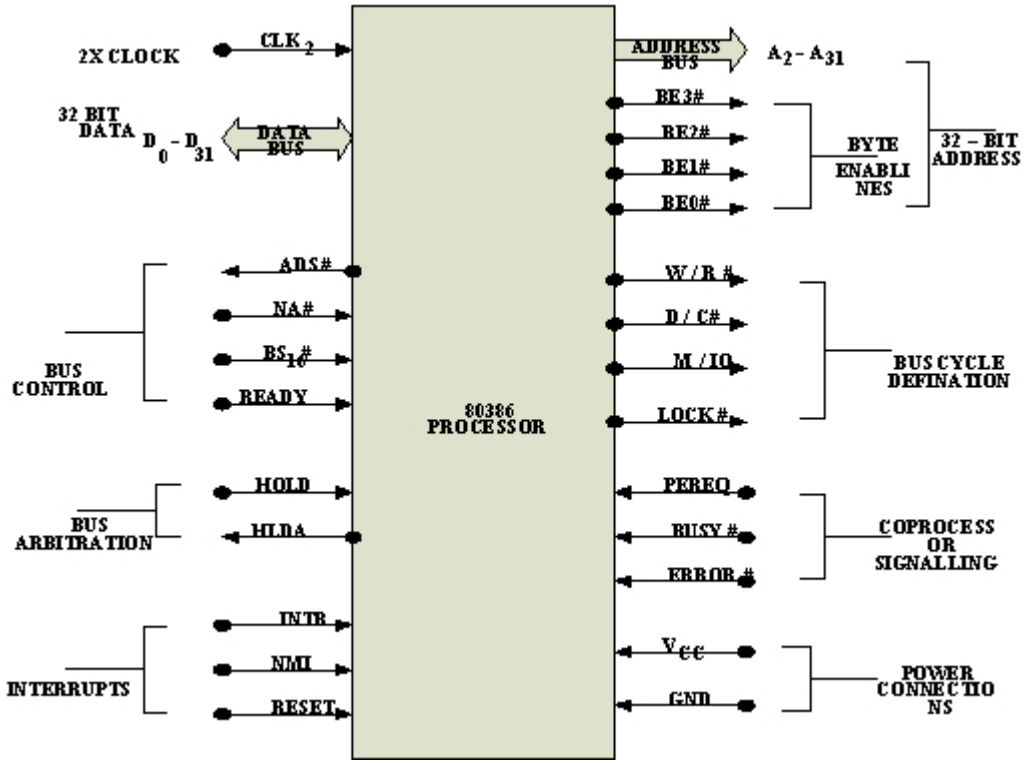


Fig 2.5: grouping of signals

Register Organisation

- **The 80386 has eight 32 - bit general purpose registers which may be used as either 8 bit or 16 bit registers.**
- **A 32 - bit register known as an extended register, is represented by the register name with prefix E.**

•Example : A 32 bit register corresponding to AX is EAX, similarly BX is EBX etc.

- **The 16 bit registers BP, SP, SI and DI in 8086 are now available with their extended size of 32 bit and are names as EBP,ESP,ESI and EDI.**

•AX represents the lower 16 bit of the 32 bit register EAX.

- **BP, SP, SI, DI represents the lower 16 bit of their 32 bit counterparts, and can be used as independent 16 bit registers.**

•The six segment registers available in 80386 are CS, SS, DS, ES, FS and GS.

- **The CS and SS are the code and the stack segment registers respectively, while DS, ES, FS, GS are 4 data segment registers.**

•A 16 bit instruction pointer IP is available along with 32 bit counterpart EIP.

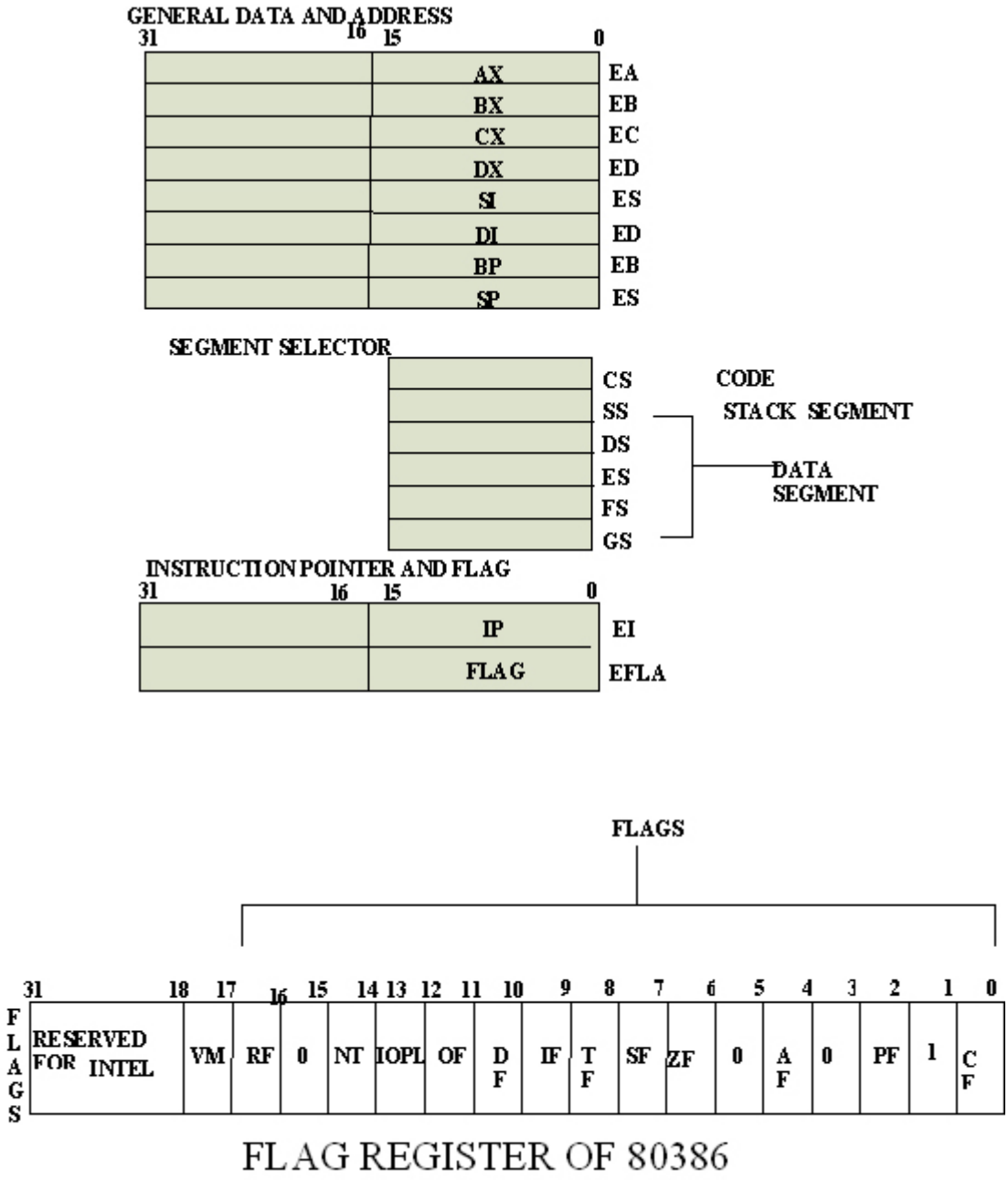


Fig 2.6- Register organization and Flag register

- ***Flag Register of 80386*: The Flag register of 80386 is a 32 bit register. Out of the 32 bits, Intel has reserved bits D_{18} to D_{31}, D_5 and D_3, while D_1 is always set at 1.Two extra new flags are added to the 80286 flag to derive the flag register of 80386. They are VM and RF flags.**
- **VM - *Virtual Mode Flag*: If this flag is set, the 80386 enters the virtual 8086 mode within the protection mode. This is to be set only when the 80386 is in protected mode. In this mode, if any privileged instruction is executed an exception 13 is generated. This bit can be set using IRET instruction or any task switch operation only in the protected mode.**

- ***RF- Resume Flag*: This flag is used with the debug register breakpoints. It is checked at the starting of every instruction cycle and if it is set, any debug fault is ignored during the instruction cycle. The RF is automatically reset after successful execution of every instruction, except for IRET and POPF instructions.**
- **Also, it is not automatically cleared after the successful execution of JMP, CALL and INT instruction causing a task switch. These instruction are used to set the RF to the value specified by the memory data available at the stack.**
- ***Segment Descriptor Registers*: This registers are not available for programmers, rather they are internally used to store the descriptor information, like attributes, limit and base addresses of segments.**
- **The six segment registers have corresponding six 73 bit descriptor registers. Each of them contains 32 bit base address, 32 bit base limit and 9 bit attributes. These are automatically loaded when the corresponding segments are loaded with selectors.**
- ***Control Registers*: The 80386 has three 32 bit control registers CR_0, CR_2 and CR_3 to hold global machine status independent of the executed task. Load and store instructions are available to access these registers.**
- ***System Address Registers*: Four special registers are defined to refer to the descriptor tables supported by 80386.**
- **The 80386 supports four types of descriptor table, viz. global descriptor table (GDT), interrupt descriptor table (IDT), local descriptor table (LDT) and task state segment descriptor (TSS).**
- ***Debug and Test Registers*: Intel has provide a set of 8 debug registers for hardware debugging. Out of these eight registers DR_0 to DR_7, two registers DR_4 and DR_5 are Intel reserved.**
- **The initial four registers DR_0 to DR_3 store four program controllable breakpoint addresses, while DR_6 and DR_7 respectively hold breakpoint status and breakpoint control information.**
- **Two more test register are provided by 80386 for page cacheing namely test control and test status register.**
- ***ADDRESSING MODES*: The 80386 supports overall eleven addressing modes to facilitate efficient execution of higher level language programs.**
- **In case of all those modes, the 80386 can now have 32-bit immediate or 32- bit register operands or displacements.**
- **The 80386 has a family of scaled modes. In case of scaled modes, any of the index register values can be multiplied by a valid scale factor to obtain the displacement.**
- **The valid scale factor are 1, 2, 4 and 8.**
- **The different scaled modes are as follows.**
- ***Scaled Indexed Mode*: Contents of the an index register are multiplied by a scale factor that may be added further to get the operand offset.**
- ***Based Scaled Indexed Mode*: Contents of the an index register are multiplied by a scale factor and then added to base register to obtain the offset.**
- ***Based Scaled Indexed Mode with Displacement*: The Contents of the an index register are multiplied by a scaling factor and the result is added to a base register and a displacement to get the offset of an operand.**

Real Address Mode of 80386

•After reset, the 80386 starts from memory location FFFFFFF0H under the real address mode. In the real mode, 80386 works as a fast 8086 with 32-bit registers and data types.

- **In real mode, the default operand size is 16 bit but 32- bit operands and addressing modes may be used with the help of override prefixes.**
- **The segment size in real mode is 64k, hence the 32-bit effective addressing must be less than 0000FFFFFH. The real mode initializes the 80386 and prepares it for protected mode.**

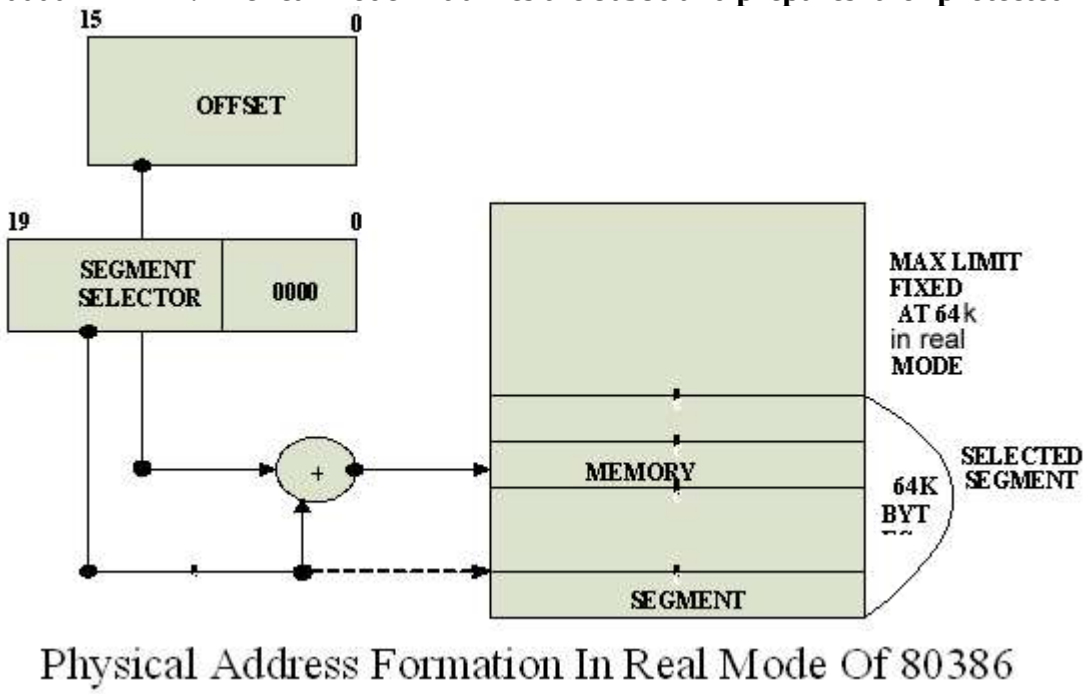


Fig 2.7 Physical address formation in Real mode

- ***Memory Addressing in Real Mode*: In the real mode, the 80386 can address at the most 1Mbytes of physical memory using address lines A_0-A_{19}.**
- **Paging unit is disabled in real addressing mode, and hence the real addresses are the same as the physical addresses.**
- **To form a physical memory address, appropriate segment registers contents (16-bits) are shifted left by four positions and then added to the 16-bit offset address formed using one of the addressing modes, in the same way as in the 80386 real address mode.**
- **The segment in 80386 real mode can be read, write or executed, i.e. no protection is available.**
- **Any fetch or access past the end of the segment limit generate exception 13 in real address mode.**

•The segments in 80386 real mode may be overlapped or non-overlapped.

- **The interrupt vector table of 80386 has been allocated 1Kbyte space starting from 00000H to 003FFH.**

Protected Mode of 80386

- **All the capabilities of 80386 are available for utilization in its protected mode of operation.**
- **The 80386 in protected mode support all the software written for 80286 and 8086 to be executed under the control of memory management and protection abilities of 80386.**
- **The protected mode allows the use of additional instruction, addressing modes and capabilities of 80386.**
-

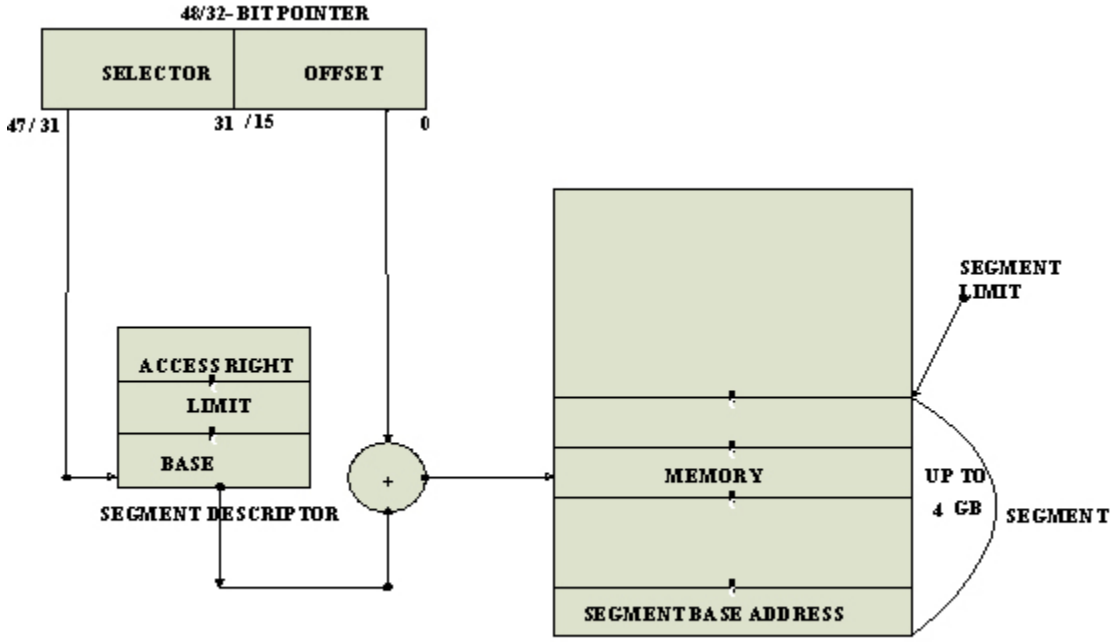


Fig 2.8 Protected mode addressing without paging

- ***ADDRESSING IN PROTECTED MODE*: In this mode, the contents of segment registers are used as selectors to address descriptors which contain the segment limit, base address and access rights byte of the segment.**
- **The effective address (offset) is added with segment base address to calculate linear address. This linear address is further used as physical address, if the paging unit is disabled, otherwise the paging unit converts the linear address into physical address.**
- **The paging unit is a memory management unit enabled only in protected mode. The paging mechanism allows handling of large segments of memory in terms of pages of 4Kbyte size.**
- **The paging unit operates under the control of segmentation unit. The paging unit if enabled converts linear addresses into physical address, in protected mode.**

Segmentation

- ***DESCRIPTOR TABLES*: These descriptor tables and registers are manipulated by the operating system to ensure the correct operation of the processor, and hence the correct execution of the program.**

•Three types of the 80386 descriptor tables are listed as follows:

•GLOBAL DESCRIPTOR TABLE (GDT)

•LOCAL DESCRIPTOR TABLE (LDT)

•INTERRUPT DESCRIPTOR TABLE (IDT)

- ***DESCRIPTORS*: The 80386 descriptors have a 20-bit segment limit and 32-bit segment address. The descriptor of 80386 are 8-byte quantities access right or attribute bits along with the base and limit of the segments.**
- ***Descriptor Attribute Bits*: The A (accessed) attributed bit indicates whether the segment has been accessed by the CPU or not.**

•The TYPE field decides the descriptor type and hence the segment type.

- **The S bit decides whether it is a system descriptor (S=0) or code/data segment descriptor (S=1).**

•The DPL field specifies the descriptor privilege level.

- **The D bit specifies the code segment operation size. If D=1, the segment is a 32-bit operand segment, else, it is a 16-bit operand segment.**
- **The P bit (present) signifies whether the segment is present in the physical memory or not. If P=1, the segment is present in the physical memory.**
- **The G (granularity) bit indicates whether the segment is page addressable. The zero bit must remain zero for compatibility with future process.**
- **The AVL (available) field specifies whether the descriptor is for user or for operating system.**

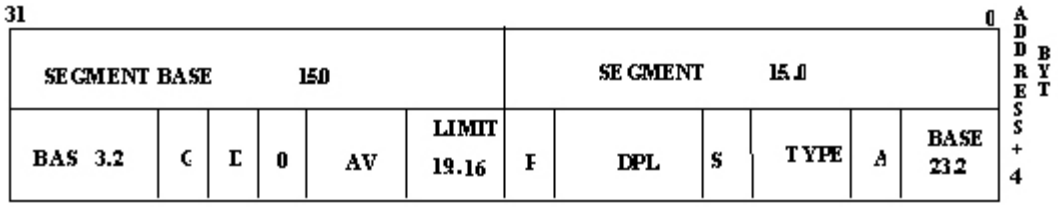
•The 80386 has five types of descriptors listed as follows:

1.Code or Data Segment Descriptors. 2.System Descriptors.

3.Local descriptors.

4. TSS (Task State Segment) Descriptors. 5.GATE Descriptors.

- **The 80386 provides a four level protection mechanism exactly in the same way as the 80286 does.**



Structure of a Descriptor

BASE Base Address of the segment
LIMIT The length of the segment
P Present Bit - 1=Present ,0 =not present
S Segment Descriptor -0 = System Descriptor,
1 = Code or data segment descriptor
TYPE Type of segment
G Granularity Bit - 1 = Segment length is page granular ,
0 = Segment length is byte granular
D Default Operation size
0 Bit must be zero
AVL Available field for user or OS

Fig 2.9 Structure of Descriptor

Paging

- ***PAGING OPERATION*: Paging is one of the memory management techniques used for virtual memory multitasking operating system.**
- **The segmentation scheme may divide the physical memory into a variable size segments but the paging divides the memory into a fixed size pages.**
- **The segments are supposed to be the logical segments of the program, but the pages do not have any logical relation with the program.**
- **The pages are just fixed size portions of the program module or data.**
- **The advantage of paging scheme is that the complete segment of a task need not be in the physical memory at any time.**
- **Only a few pages of the segments, which are required currently for the execution need to be available in the physical memory. Thus the memory requirement of the task is substantially reduced, relinquishing the available memory for other tasks.**
- **Whenever the other pages of task are required for execution, they may be fetched from the secondary storage.**
- **The previous page which are executed, need not be available in the memory, and hence the space occupied by them may be relinquished for other tasks.**
- **Thus paging mechanism provides an effective technique to manage the physical memory for multitasking systems.**

- ***Paging Unit*: The paging unit of 80386 uses a two level table mechanism to convert a linear address provided by segmentation unit into physical addresses.**
- **The paging unit converts the complete map of a task into pages, each of size 4K. The task is further handled in terms of its page, rather than segments.**
- **The paging unit handles every task in terms of three components namely page directory, page tables and page itself.**

- ***Paging Descriptor Base Register*: The control register CR_2 isused to store the 32-bit linear address at which the previous page fault was detected.**
- **The CR_3 is used as page directory physical base address register, to store the physical starting address of the page directory.**
- **The lower 12 bit of the CR_3 are always zero to ensure the page size aligned directory. A move operation to CR_3 automatically loads the page table entry caches and a task switch operation, to load CR_0 suitably.**

- ***Page Directory* : This is at the most 4Kbytes in size. Each directory entry is of 4 bytes, thus a total of 1024 entries are allowed in a directory.**
- **The upper 10 bits of the linear address are used as an index to the corresponding page directory entry. The page directory entries point to page tables.**

- ***Page Tables*: Each page table is of 4Kbytes in size and many contain a maximum of 1024 entries. The page table entries contain the starting address of the page and the statistical information about the page.**
- **The upper 20 bit page frame address is combined with the lower 12 bit of the linear address. The address bits A_{12}- A_{21} are used to select the 1024 page table entries. The page table can be shared between the tasks.**
- **The P bit of the above entries indicate, if the entry can be used in address translation.**
- **If P=1, the entry can be used in address translation, otherwise it cannot be used.**
- **The P bit of the currently executed page is always high.**
- **The accessed bit A is set by 80386 before any access to the page. If A=1, the page is accessed, else unaccessed.**
- **The D bit (Dirty bit) is set before a write operation to the page is carried out. The D-bit is undefined for page director entries.**
- **The OS reserved bits are defined by the operating system software.**
- **The User / Supervisor (U/S) bit and read/write bit are used to provide protection. These bits are decoded to provide protection under the 4 level protection model.**
- **The level 0 is supposed to have the highest privilege, while the level 3 is supposed to have the least privilege.**
- **This protection provide by the paging unit is transparent to the segmentation unit.**

PAGE TABLE 31..1	0 RESERV	0	0	D	A	0	0	U/S	R/W	P

PAGE DIRECTORY ENTRY

PAGE FRAME ADDRESS 31..1	OS RESERVE	0	0	D	A	0	0	U/S	R/W	P

PAGE TABLE ENTRY

U/S	R/W	PERMITTED FOR LEVEL 3	PERMITTED LEVEL2 ,1 OR 0
0	0	NONE	READ /WRITE
0	1	NONE	READ/ WRIT
1	0	READ	READ/ WRITE
1	1	READ-WRITE	READ/ WRITE

Fig 2.10 Page directory and page table entry

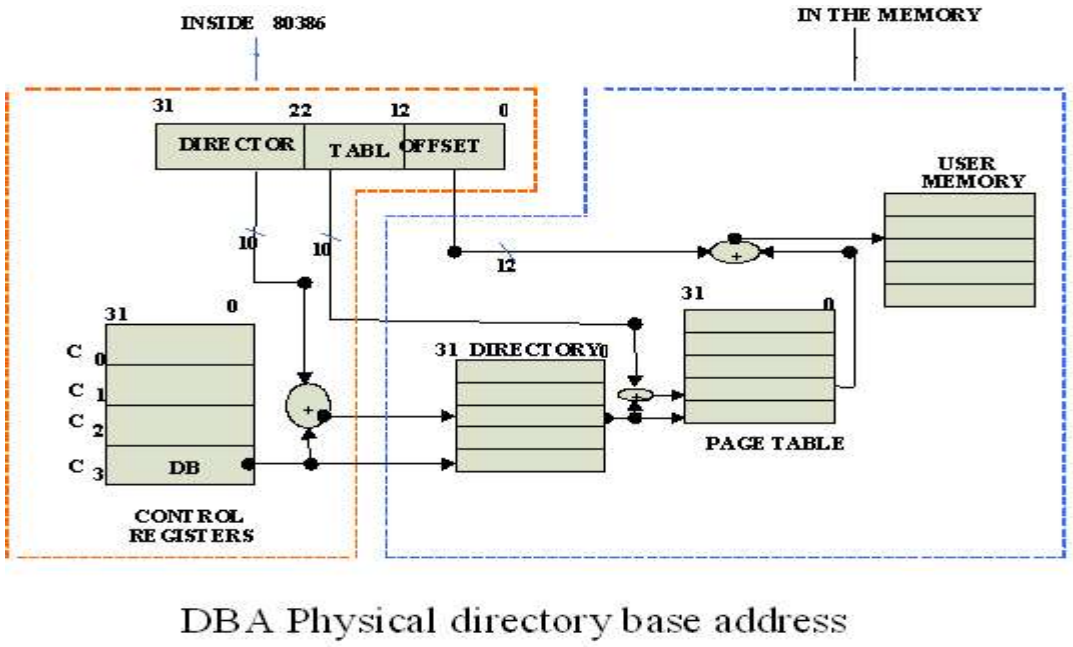


Fig 2.11 DBA physical director base address

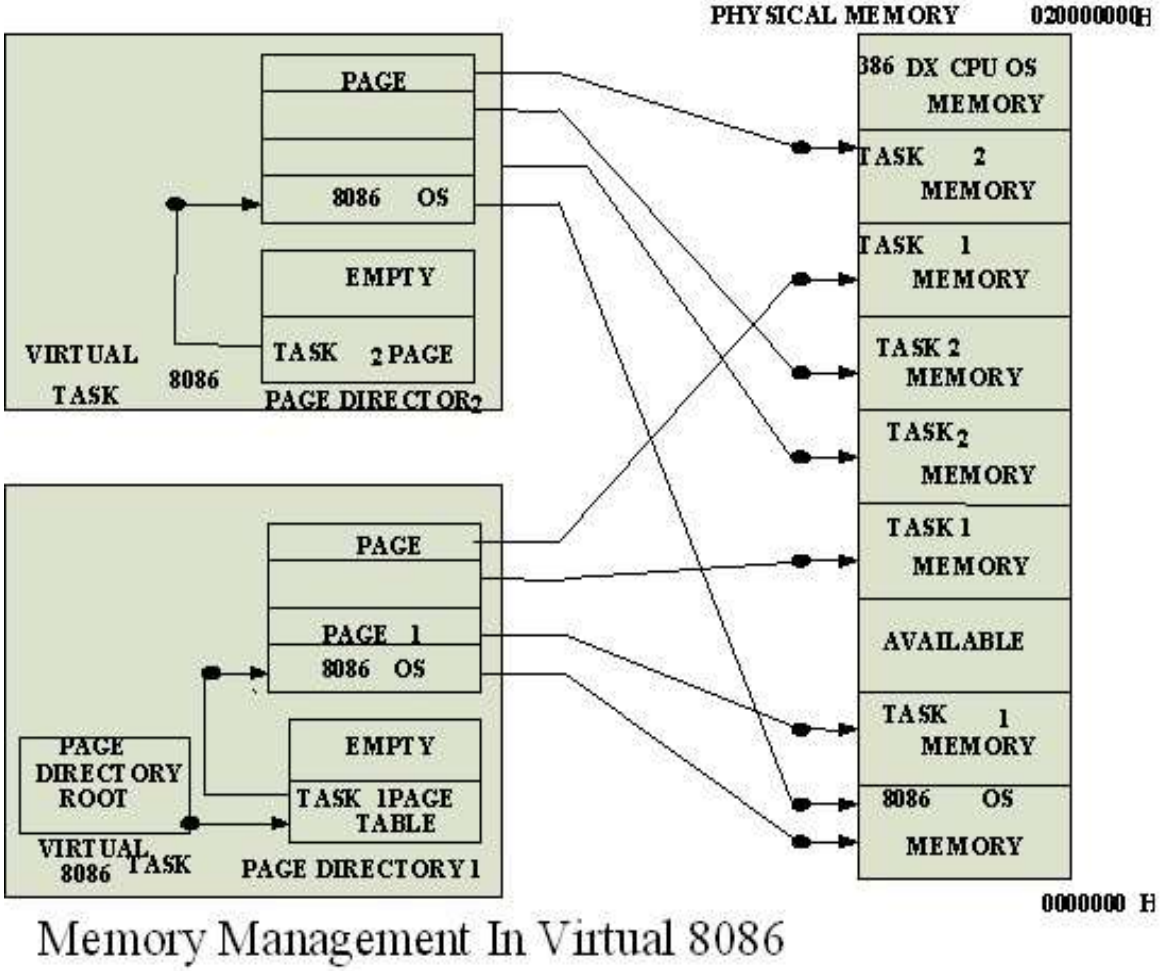
Virtual 8086 Mode

- **In its protected mode of operation, 80386DX provides a virtual 8086 operating environment to execute the 8086 programs.**
- **The real mode can also used to execute the 8086 programs along with the capabilities of 80386, like protection and a few additional instructions.**
- **Once the 80386 enters the protected mode from the real mode, it cannot return back to the real mode without a reset operation.**
- **Thus, the virtual 8086 mode of operation of 80386, offers an advantage of executing 8086 programs while in protected mode.**
- **The address forming mechanism in virtual 8086 mode is exactly identical with that of 8086 real mode.**
- **In virtual mode, 8086 can address 1Mbytes of physical memory that may be anywhere in the 4Gbytes address space of the protected mode of 80386.**
- **Like 80386 real mode, the addresses in virtual 8086 mode lie within 1Mbytes of memory.**
- **In virtual mode, the paging mechanism and protection capabilities are available at the service of the programmers.**
- **The 80386 supports multiprogramming, hence more than one programmer may be use the CPU at a time.**

- **Paging unit may not be necessarily enable in virtual mode, but may be needed to run the 8086 programs which require more than 1Mbyts of memory for memory management function.**
- **In virtual mode, the paging unit allows only 256 pages, each of 4Kbytes size.**
- **Each of the pages may be located anywhere in the maximum 4Gbytes physical memory. The virtual mode allows the multiprogramming of 8086 applications.**

- **The virtual 8086 mode executes all the programs at privilege level 3.Any of the other programmes may deny access to the virtual mode programs or data.**

- **However, the real mode programs are executed at the highest privilege level, i.e. level 0.**
- **The virtual mode may be entered using an IRET instruction at CPL=0 or a task switch at any CPL, executing any task whose TSS is having a flag image with VM flag set to 1.**
- **The IRET instruction may be used to set the VM flag and consequently enter the virtual mode.**
- **The PUSHF and POPF instructions are unable to read or set the VM bit, as they do not access it.**
- **Even in the virtual mode, all the interrupts and exceptions are handled by the protected mode interrupt handler.**
- **To return to the protected mode from the virtual mode, any interrupt or execution may be used.**
- **As a part of interrupt service routine, the VM bit may be reset to zero to pull back the 80386 into protected mode.**



Memory Management In Virtual 8086

Fig 2.12 Memory management in Virtual 8086 mode

Summary of 80386

- **This 80386 is a 32bit processor that supports, 8bit/32bit data operands.**
- **The 80386 instruction set is upward compatible with all its predecessors.**
- **The 80386 can run 8086 applications under protected mode in its virtual 8086 mode of operation.**
- **With the 32 bit address bus, the 80386 can address upto 4Gbytes of physical memory. The physical memory is organised in terms of segments of 4Gbytes at maximum.**
- **The 80386 CPU supports 16K number of segments and thus the total virtual space of 4Gbytes * 16K = 64 Terrabytes.**
- **The memory management section of 80386 supports the virtual memory, paging and four levels of protection, maintaining full compatibility with 80286.**
- **The 80386 offers a set of 8 debug registers DR_0-DR_7 for hardware debugging and control. The 80386 has on-chip address translation cache.**
- **The concept of paging is introduced in 80386 that enables it to organise the available physical memory in terms of pages of size 4Kbytes each, under the segmented memory.**
- **The 80386 can be supported by 80387 for mathematical data processing.**

80486 MICROPROCESSOR

- **The 32-bit 80486 is the next evolutionary step up from the 80386.**
- **One of the most obvious feature included in a 80486 is a built in math coprocessor. This coprocessor is essentially the same as the 80387 processor used with a 80386, but being integrated on the chip allows it to execute math instructions about three times as fast as a 80386/387 combination.**
- **80486 is an 8Kbyte code and data cache.**
- **To make room for the additional signals, the 80486 is packaged in a 168 pin, pin grid array package instead of the 132 pin PGA used for the 80386.**

Pin Definitions

- **A_{31}-A_2 : Address outputs A_{31}-A_2 provide the memory and I/O with the address during normal operation. During a cache line invalidation A_{31}-A_4 are used to drive the microprocessor.**

- **$A_{20}M_3$: The address bit 20 mask causes the 80486 to wrap its address around from location 000FFFFFH to 00000000H as in 8086. This provides a memory system that functions like the 1M byte real memory system in the 8086 processors.**

- **$\overline{ADS}$: The address data strobe become logic zero to indicate that the address bus contains a valid memory address.**
- **AHOLD: The address hold input causes the microprocessor to place its address bus connections at their high-impedance state, with the remainder of the buses staying active. It is often used by another bus master to gain access for a cache invalidation cycle.**

- **BREQ: This bus request output indicates that the 486 has generated an internal bus**

request.

- **BE3 - BE0 : Byte enable outputs select a bank of the memory system when information is transferred between the microprocessor and its memory and I/O.**

  The BE_3 signal enables $D_{31} – D_{24}$, BE_2 enables D_{23}-D_{16}, BE_1 enables $D_{15} – D_8$ and BE_0 enables D_7-D_0.

- **BLAST : The burst last output shows that the burst bus cycle is complete on the next activation of BRDY# signal.**

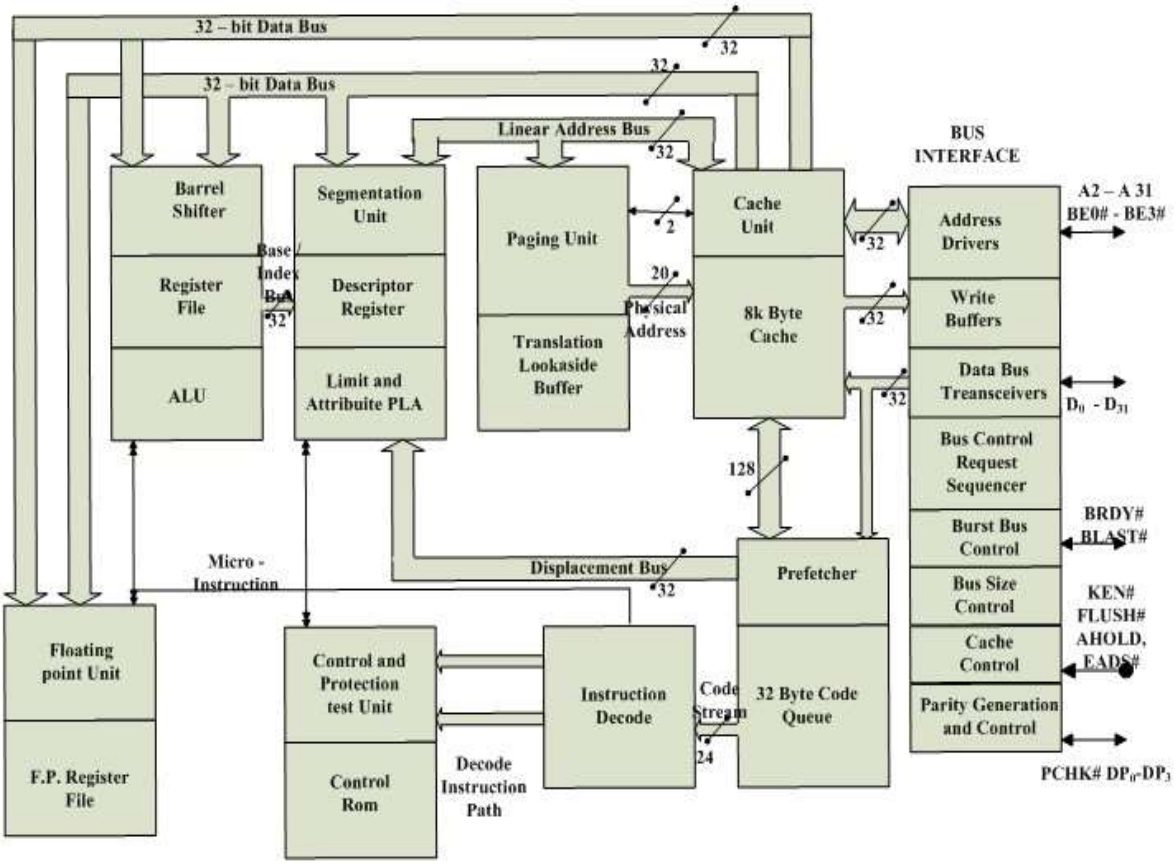


Fig 2.13 Block diagram-80486

- **BOFF : The Back-off input causes the microprocessor to place its buses at their high impedance state during the next cycle. The microprocessor remains in the bus hold state until the BOFF# pin is placed at a logic 1 level.**

- **NMI : The non-maskable interrupt input requests a type 2 interrupt.**

- **BRDY : The burst ready input is used to signal the microprocessor that a burst cycle is complete.**

- **$\overline{\text{KEN}}$: The cache enable input causes the current bus to be stored in the internal.**

- **$\overline{\text{LOCK}}$: The lock output becomes a logic 0 for any instruction that is prefixed with the lock prefix.**

- **W / $\overline{\text{R}}$: current bus cycle is either a read or a write.**

- **$\overline{\text{IGNNE}}$: The ignore numeric error input causes the coprocessor to ignore floating point error and to continue processing data. The signal does not affect the state of the FERR pin.**

- **$\overline{\text{FLUSH}}$: The cache flush input forces the microprocessor to erase the contents of its 8K byte internal cache.**

- **$\overline{\text{EADS}}$: The external address strobe input is used with AHOLD to signal that an external address is used to perform a cache invalidation cycle.**

- **$\overline{\text{FERR}}$: The floating point error output indicates that the floating point coprocessor has detected an error condition. It is used to maintain compatibility with DOS software.**

- **$\overline{\text{BS8}}$: The bus size 8, input causes the 80486 to structure itself with an 8-bit data bus to access byte-wide memory and I/O components.**

- **$\overline{\text{BS16}}$: The bus size 16, input causes the 80486 to structure itself with an 16-bit data bus to access word-wide memory and I/O components.**

- **$\overline{\text{PCHK}}$: The parity check output indicates that a parity error was detected during a read operation on the $DP_3 – DP_0$ pin.**

- **$\overline{\text{PLOCK}}$: The pseudo-lock output indicates that current operation requires more than one bus cycle to perform. This signal becomes a logic 0 for arithmetic coprocessor operations that access 64 or 80 bit memory data.**

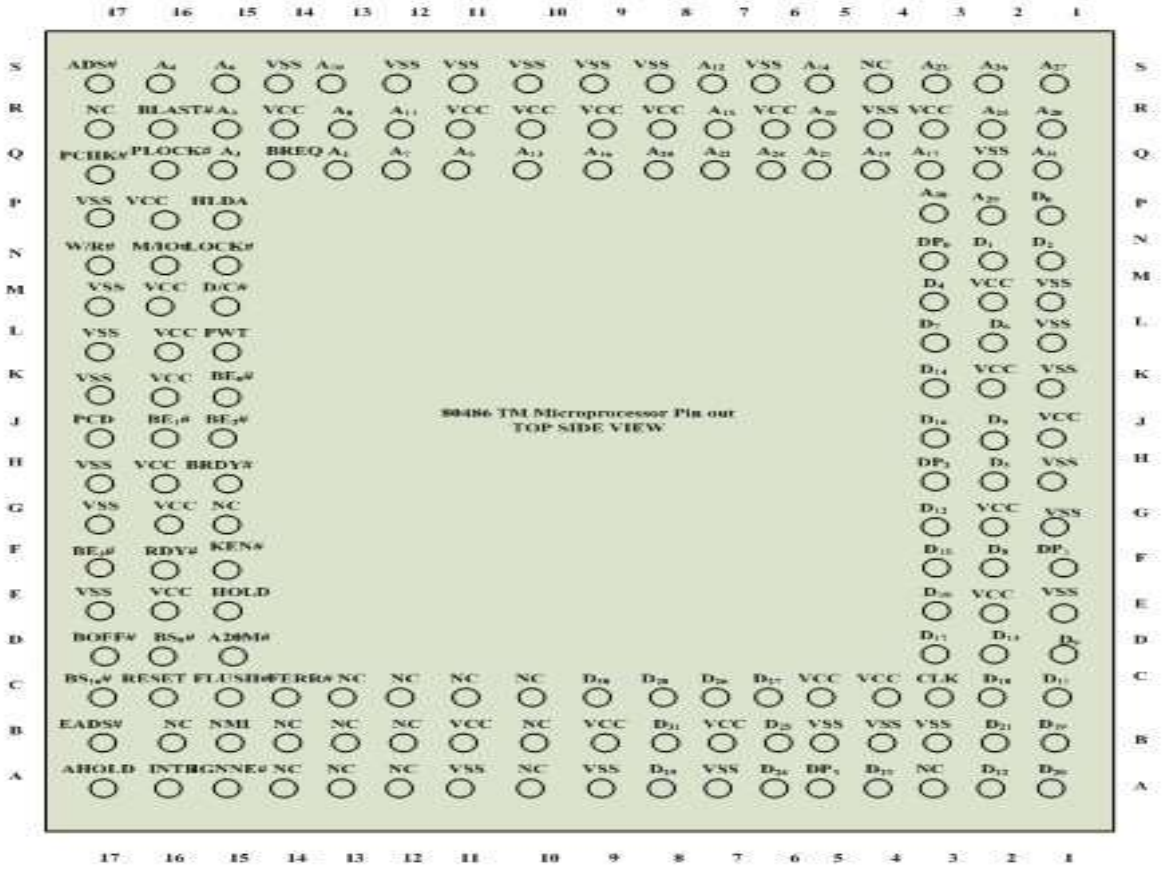


Fig 2.14 Pin diagram -80486

- **PWT: The page write through output indicates the state of the PWT attribute bit in the page table entry or the page directory entry.**

- **$\overline{\text{RDY}}$: The ready input indicates that a non-burst bus cycle is complete. The RDY signal must be returned or the microprocessor places wait states into its timing until RDY is asserted.**

- **M / $\overline{\text{IO}}$: Memory / IO# defines whether the address bus contains a memory address or an I/O port number. It is also combined with the W/ R signal to generate memory and I/O read and write control signals.**

80486 Signal Group

- **The 80486 data bus, address bus, byte enable, ADS#, RDY#, INTR, RESET, NMI, M/IO#, D/C#, W/R#, LOCK#, HOLD, HLDA and BS_{16}# signals function as we described for 80386.**

- **The 80486 requires 1 clock instead of 2 clock required by 80386.**
- **A new signal group on the 486 is the PARITY group DP_0-DP_3 and PCHK#.**
- **These signals allow the 80486 to implement parity detection / generation for memory reads and memory writes.**

- **During a memory write operation, the 80486 generates an even parity bit for each byte and outputs these bits on the DP_0-DP_3 lines.**

- **These bits will store in a separate parity memory bank.**
- **During a read operation the stored parity bits will be read from the parity memory and applied to the DP_0-DP_3 pins.**
- **The 80486 checks the parities of the data bytes read and compares them with the DP_0-DP_3 signals. If a parity error is found, the 80486 asserts the PCHK# signal.**
- **Another new signals group consists of the BURST ready signal BRDY# and BURST last signal BLAST#.**
- **These signals are used to control burst-mode memory reads and writes.**
- **A normal 80486 memory read operation to read a line into the cache requires 2 clock cycles. However, if a series of reads is being done from successive memory locations, the reads can be done in burst mode with only 1 clock cycle per read.**
- **To start the process the 80486 sends out the first address and asserts the BLAST# signal high. When the external DRAM controller has the first data bus, it asserts the BRDY# signal.**
- **The 80486 reads the data word and outputs the next address. Since the data words are at successive addresses, only the lower address bits need to be changed. If the DRAM controller is operating in the page or the static column modes then it will only have to output a new column address to the DRAM.**
- **In this mode the DRAM will be able to output the new data word within 1 clock cycle.**

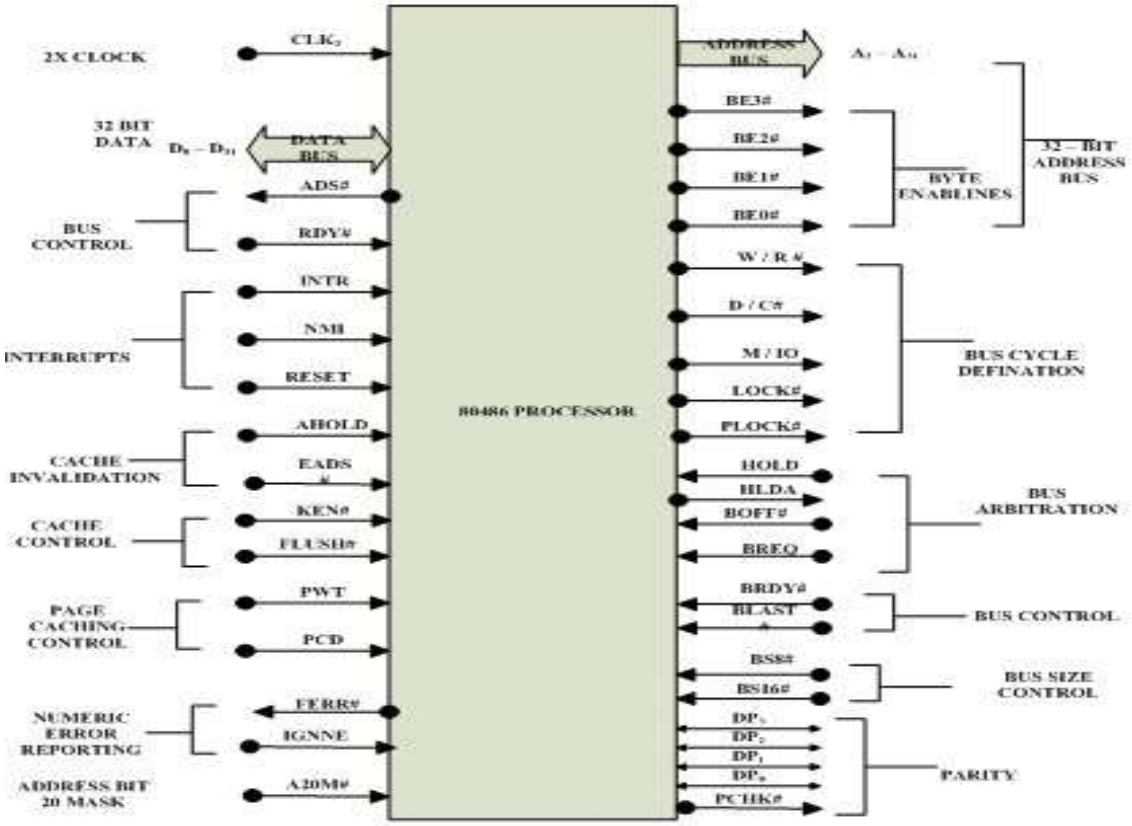


Fig 2.15- Grouping of signals-80486

When the processor has read the required number of data words, it asserts the BLAST# signal low to terminate the burst mode.

- **The final signal we want to discuss here are the bus request output signal BREQ, the back-off input signal BOFF#, the HOLD signal and the hold-acknowledge signal HLDA.**

- **These signals are used to control sharing the local 486 bus by multiple processors (bus master).**

•When a master on the bus need to use the bus, it asserts its BERQ signal .

- **An external parity circuit will evaluate requests to use the bus and grant bus use to the highest – priority master. To ask the 486 to release the bus , the bus controller asserts the 486 HOLD input or BOFF# input.**

- **If the HOLD input is asserted, the 486 will finish the current bus cycle, float its buses and assert the HLDA signal.**

- **To prevent another master from taking over the bus during a critical operation, the 486 can assert its LOCK# or PLOCK# signal.**

EFLAG Register of The 80486

- **The extended flag register EFLAG is illustrated in the figure. The only new flag bit is the AC alignment check, used to indicate that the microprocessor has accessed a word at an odd address or a double word boundary.Efficient software and execution require that data be stored at word or doubleword boundaries.**

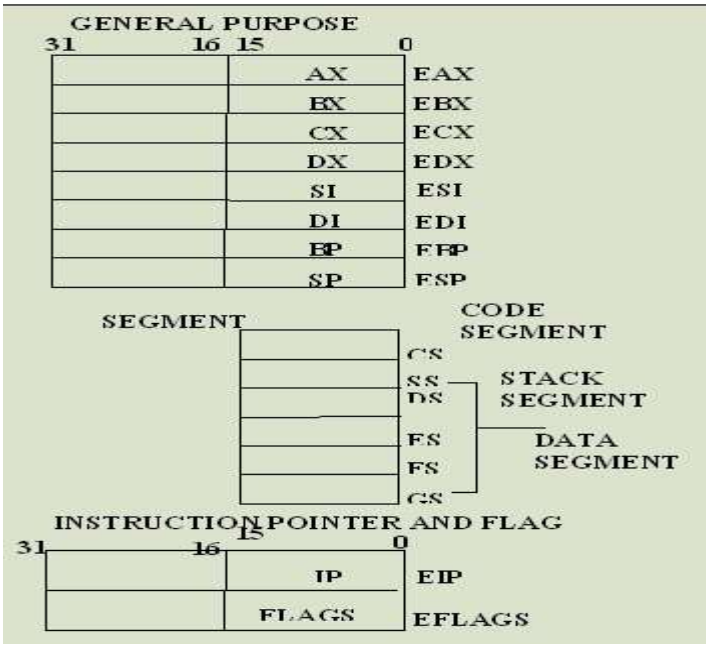


Fig 2.16 Register organization -80486

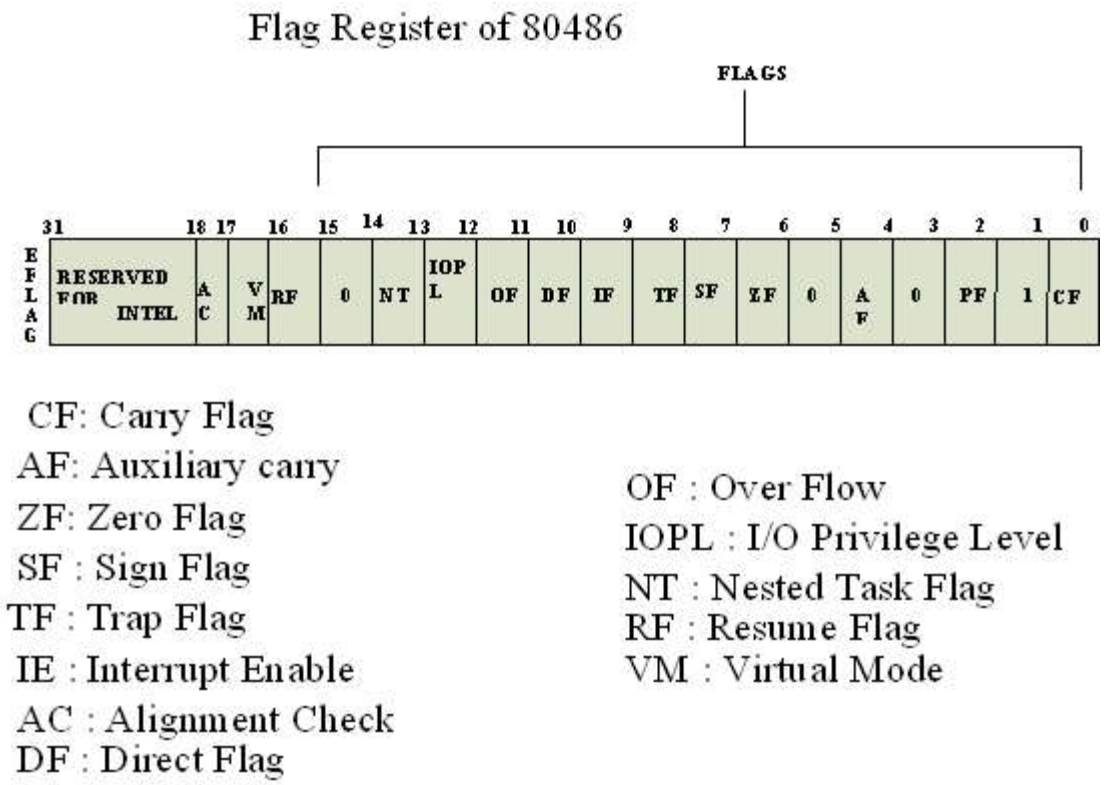


Fig 2.17- 80486 flag register

80486 Memory System

- **The memory system for the 486 is identical to 386 microprocessor. The 486 contains 4G bytes of memory beginning at location 00000000H and ending at FFFFFFFFH.**
- **The major change to the memory system is internal to 486 in the form of 8K byte cache memory, which speeds the execution of instructions and the acquisition of data.**
- **Another addition is the parity checker/ generator built into the 80486 microprocessor.**
- ***Parity Checker / Generator* : Parity is often used to determine if data are correctly read from a memory location. INTEL has incorporated an internal parity generator / decoder.**

- **Parity is generated by the 80486 during each write cycle. Parity is generated as even parity and a parity bit is provided for each byte of memory. The parity check bits appear on pins DP0-DP3, which are also parity inputs as well as parity outputs.**
- **These are typically stored in memory during each write cycle and read from memory during each read cycle.**
- **On a read, the microprocessor checks parity and generates a parity check error, if it occurs on the PCHK# pin. A parity error causes no change in processing unless the user applies the PCHK signal to an interrupt input.**
- **Interrupts are often used to signal a parity error in DS-based computer systems. This is same as 80386, except the parity bit storage.**
- **If parity is not used, Intel recommends that the DP0 – DP3 pins be pulled up to +5v.**

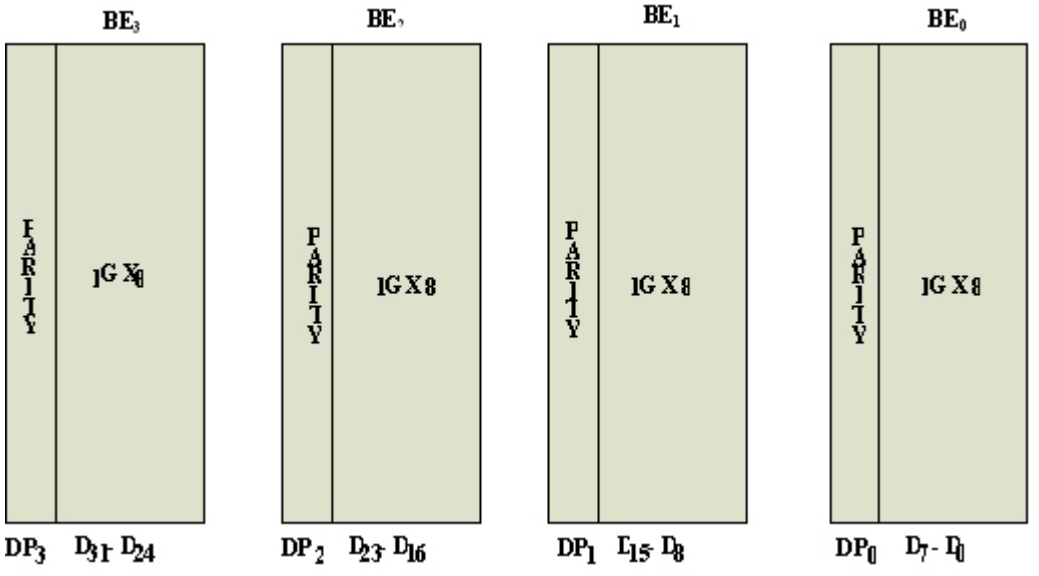


Fig 2.18- Memory system

- ***CACHE MEMORY:*** **The cache memory system stores data used by a program and also the instructions of the program. The cache is organised as a 4 way set associative cache with each location containing 16 bytes or 4 doublewords of data.**
- **Control register CR0 is used to control the cache with two new control bits not present in the 80386 microprocessor.**
- **The CD (cache disable) , NW (non-cache write through) bits are new to the 80486 and are used to control the 8K byte cache.**
- **If the CD bit is a logic 1, all cache operations are inhibited. This setting is only used for debugging software and normally remains cleared. The NW bit is used to inhibit cache write-through operation. As with CD, cache write through is inhibited only for testing. For normal operations CD = 0 and NW = 0.**
- **Because the cache is new to 80486 microprocessor and the cache is filled using burst cycle not present on the 386.**

80486 Memory Management

- **The 80486 contains the same memory-management system as the 80386. This includes a paging unit to allow any 4K byte block of physical memory to be assigned to any 4K byte block of linear memory. The only difference between 80386 and 80486 memory- management system is paging.**
- **The 80486 paging system can disabled caching for section of translation memory pages, while the 80386 could not.**

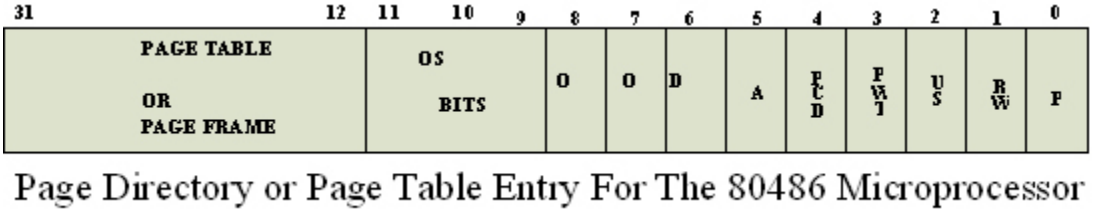


Fig 2.19-Page directory or page table entry

- **If these are compared with 80386 entries, the addition of two new control bits is observed (PWT and PCD).**

•The page write through and page cache disable bits control caching.

- **The PWT controls how the cache functions for a write operation of the external cache memory. It does not control writing to the internal cache. The logic level of this bit is found on the PWT pin of the 80486 microprocessor. Externally, it can be used to dictate the write through policy of the external caching.**
- **The PCD bit controls the on-chip cache. If the PCD = 0, the on-chip cache is enabled for the current page of memory.**
- **Note that 80386 page table entries place a logic 0 in the PCD bit position, enabling caching. If PCD = 1, the on-chip cache is disable. Caching is disable regard less of condition of KEN#, CD, and NW.**

Cache Test Registers

•The 80486 cache test registers are TR3, TR4, TR5.

- **Cache data register (TR3) is used to access either the cache fill buffer for a write test operation or the cache read buffer for a cache read test operation.**
- **In order to fill or read a cache line (128 bits wide), TR3 must be written or read four times.**

- **The contents of the set select field in TR5 determine which internal cache line is written or read through TR3. The 7 bit test field selects one of the 128 different 16 byte wide cache lines. The entry select bits of TR5 select an entry in the set or the 32 bit location in the read buffer.**

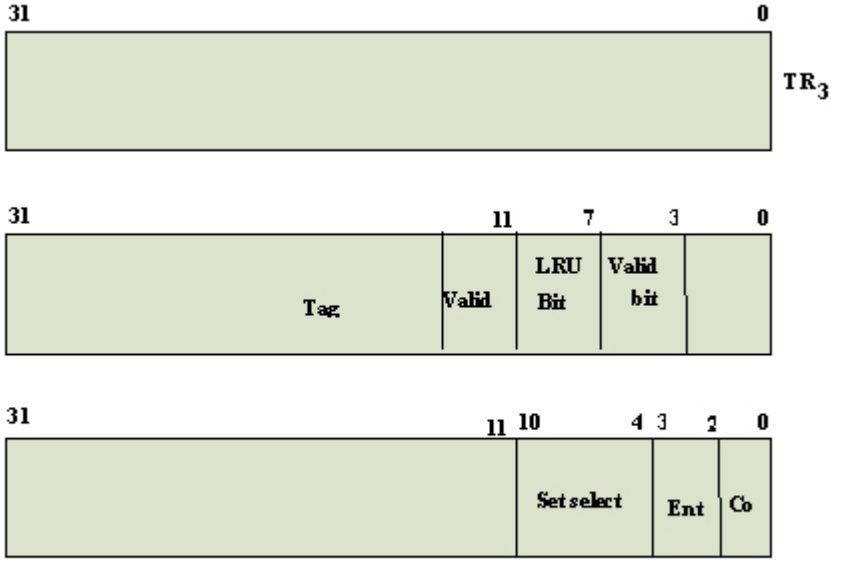
•The control bits in TR5 enable the fill buffer or read buffer operation (00)
•Perform a cache write (01), Perform a cache read (10)
•Flush the cache (11).

- **The cache status register (TR4) hold the cache tag, LRU bits and a valid bit. This register is loaded with the tag and valid bit before a cache a cache write operation and contains the tag, valid bit, LRU bits, and 4 valid bits on a cache test read.**

•Cache is tested each time that the microprocessor is reset if the AHOLD pin is high for 2 clocks prior to the RESET pin going low. This causes the 486 to completely test itself with a built in self test or BIST.

- **The BIST uses TR3, TR4, TR5 to completely test the internal cache. Its outcome is reported in register EAX. If EAX is a zero, the microprocessor, the coprocessor and cache have passed the self test.**

• **The value of EAX can be tested after reset to determine if an error is detected. In most of the cases we do not directly access the test register unless we wish to perform our own tests on the cache or TLB.**



Cache test register of the 80486 microprocessor

Fig 2.20 Cache test register

S.No	Description	8086	80286	80386	80486
1	Data bus	16	16	32	32
2	Address Bus	20	24	32	32
		Address and Data Lines are MULTIPLEXED		Address and Data Lines are NOT MULTIPLEXED	
3	Physical Address	1 MB	16 MB	4 GB	4 GB
4	Speed	8 MHz	8 MHz	50 MHz	66 MHz
5	Internal data word	16 Bits	16 Bits	32 Bits	32 Bits
6	Math Co-processor	8087 External	80287 External	80387 External	Internal
7	Pins	40(DIP)	68(Flat Package)	132 (PGA)	168 (PGA)
8	Execution Speed	2.5 MIPS	8 MIPS	27 MIPS	54 MIPS

Fig 2.21 Comparison

S.No	Description	8086	80286	80386	80486
9	Memory Management	External	Internal	Internal	Internal
10	Instruction cache	******	******	16 Bytes External	32 Bytes External
11	Data Cache	******	******	256 Bytes External	8KB Bytes External
12	FLAGS	O-D-I-T-S-Z-AF-P-C	12 NT-IOP-IOP are New	14 RF-VM are New	15 AC New
13	Segments	CS,DS,SS,ES	CS,DS,SS,ES	FS,GS are New	FS,GS are New
14	Offset Registers	SI,DI,BX	SI,DI,BX	Any Registers	Any Registers
15	Registers	AX,BX,CX,DX, SI,DI,SP,BP,IP (8,16)	AX,BX,CX,DX,SI,DI ,SP,BP,IP (8,16)	EAX,EBX,ECX,EDX,ESI,EDI,ESP, EBP,EIP (8,16,32)	EAX,EBX,ECX, EDX,ESI,EDI, ESP,EBP,EIP (8,16,32)
16	Year Introduction	1978	1982	1985	1989

Fig 2.22 Comparison(contd)

TEXT / REFERENCE BOOKS

1. **Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing", 2nd Edition,Pearson Education, Inc., 2004**
2. **Barry B.Brey , "The Intel Microprocessors 8086/8088, 8086, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, Architecture, Programming and interfacing", Prentice Hall of India Private Limited, New Delhi, 2003**
3. **Alan Clements, "The Principles of computer Hardware", Oxford University Press, 3rd Edition, 2003**
4. **John Paul Shen, Mikko H.Lipasti, "Modern Processor Design", Tata McGraw Hill, 2006**

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMMUNICATION ENGINEERING

UNIT - 3
ADVANCED MICROPROCESSORS – SEC1601

UNIT 3- PENTIUM PROCESSORS

Introduction to Pentium Microprocessor - Special Pentium registers - Branch Prediction Logic, Floating Point Module, Cache Structure, and Superscalar Architecture. Pentium memory management - New Pentium Instructions - Pentium Processor -Special Pentium pro features - Pentium 4 processor

Pentium family history
Pentium processor details
Pentium registers
Data
Pointer and index
Control
Segment
Real mode memory architecture

- **Protected mode memory architecture**
  - **Segment registers**
  - **Segment descriptors**
  - **Segment descriptor tables**
  - **Segmentation models**

PENTIUM FAMILY HISTORY

- **Mixed-mode operation**
- **Default segment registers used**
- **Intel introduced microprocessors in 1969**
- **4-bit microprocessor 4004**
- **8-bit microprocessors**
- **8080**
- **8085**
- **16-bit processors**
- **8086 introduced in 1979**
- **20-bit address bus, 16-bit data bus**
- **8088 is a less expensive version**
- **Uses 8-bit data bus**
- **Can address up to 4 segments of 64 KB**
- **Referred to as the real mode**
  - **80186**
    - **A faster version of 8086**
    - **16-bit data bus and 20-bit address bus**
    - **Improved instruction set**
  - **80286 was introduced in 1982**
    - **24-bit address bus**
    - **16 MB address space**
    - **Enhanced with memory protection capabilities**
    - **Introduced protected mode**
      - **Segmentation in protected mode is different from the real mode**
    - **Backwards compatible**
- **80386 was introduced 1985**
- **First 32-bit processor**
- **32-bit data bus and 32-bit address bus**

- **4 GB address space**
- **Segmentation can be turned off (flat model)**
- **Introduced paging**
- **80486 was introduced 1989**
- **Improved version of 386**
- **Combined coprocessor functions for performing floating-point arithmetic**
- **Added parallel execution capability to instruction decode and execution units**
- **Achieves scalar execution of 1 instruction/clock**
    - **Later versions introduced energy savings for laptops**
    -

Pentium (80586) was introduced in 1993

- **Similar to 486 but with 64-bit data bus**
- **Wider internal datapaths**
    - **128- and 256-bit wide**
- **Added second execution pipeline**
    - **Superscalar performance-Superscalar means that the CPU can execute two (or more) instructions per cycle.**
    - **Two instructions/clock**
- **Doubled on-chip L1 cache**
    - **8 KB data**
    - **8 KB instruction**
- **Added branch prediction-a prediction is made for the branch instruction currently in the pipeline**
- **Pentium Pro was introduced in 1995**
- **Three-way superscalar-The Pentium has what is known as a "superscalar pipelined architecture." Superscalar means that the CPU can execute two (or more) instructions per cycle.**
- **3 instructions/clock**
- **36-bit address bus**
- **64 GB address space**
- **Introduced dynamic execution**
- **Out-of-order execution-is a technique used in most high-performance microprocessors to make use of cycles that would otherwise be wasted by a certain type of costly delay.**
- **Speculative execution- is an optimization technique where a computer system performs some task that may not be needed. Work is done before it is known whether it is actually needed, so as to prevent a delay that would have to be incurred by doing the work after it is known that it is needed.**
- **In addition to the L1 cache**
- **Has 256 KB L2 cache**
- **L1 is "level-1" cache memory, usually built onto the microprocessor chip itself. L2 (that is, level-2) cache memory is on a separate chip (possibly on an expansion card) that can be accessed more quickly than the larger "main" memory. A popular L2 cache memory size is 1,024 kilobytes (one megabyte)**
- **Pentium II was introduced in 1997**
- **Introduced multimedia (MMX) instructions**
- **Doubled on-chip L1 cache**
- **16 KB data**
- **16 KB instruction**
- **Introduced comprehensive power management features**
- **Sleep**
- **Deep sleep**

- **Sleep and Deep Sleep to conserve power during idle times-done by stopping the clock to internal sections of the processor**
- **In addition to the L1 cache**
- **Has 256 KB L2 cache**
- **Pentium III, Pentium IV,…**

- **Itanium processor**
  - **RISC design**
    - **Previous designs were CISC**
  - **64-bit processor**
  - **Uses 64-bit address bus**
  - **128-bit data bus**
  - **Introduced several advanced features**
    - **Speculative execution**
    - **Predication to eliminate branches**
    - **Branch prediction**

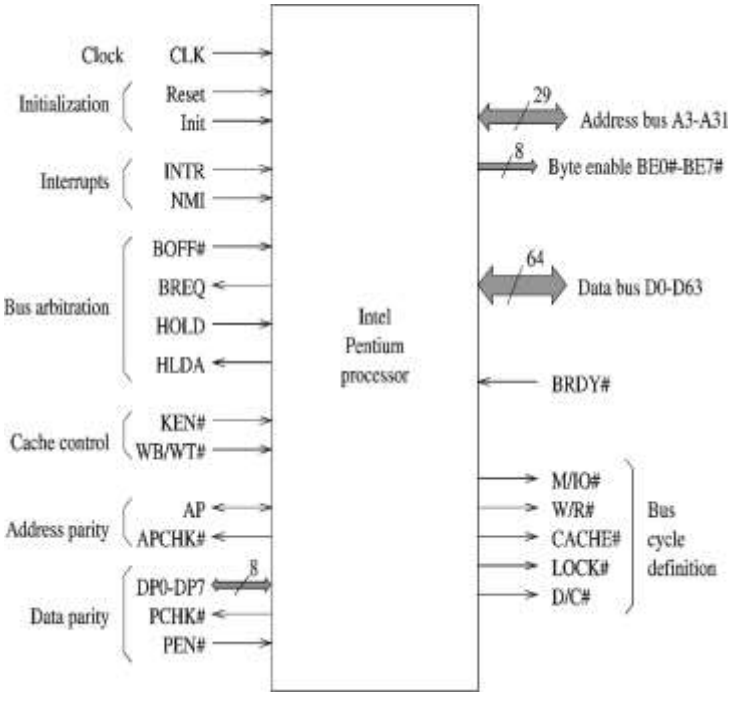


Fig 3.1. Pin diagram-Pentium

- **Data bus (D0 – D 63)**
  - **64-bit data bus**
- **Address bus (A3 – A31)**
  - **Only 29 lines**
    - **No A0-A2 (due to 8-byte wide data bus)**
- **Byte enable (BE0# - BE7#)**
  - **Identifies the set of bytes to read or write**
    - **BE0# : least significant byte (D0 – D7)**
    - **BE1# : next byte (D8 – D15)**
    - **…**
    - **BE7# : most significant byte (D56 – D63)**

Any combination of bytes can be specified

- **Data parity (DP0 – DP7)**
  - **Even parity for 8 bytes of data**
    - **DP0 : D0 – D7**

        - **DP1 : D8 – D15**
        - **…**
        - **DP7 : D56 – D63**
- **Parity check (PCHK#)**
    - **Indicates the parity check result on data read**
    - **Parity is checked only for valid bytes**
        - **Indicated by BE# signals**
- **Parity enable (PEN#)**
    - **Determines whether parity check should be used**
- **Address parity (AP)**
    - **Bad address parity during inquire cycles**
- **Memory/IO (M/IO#)**
    - **Defines bus cycle: memory or I/O**
- **Write/Read (W/R#)**
    - **Distinguishes between write and read cycles**
- **Data/Code (D/C#)**
    - **Distinguishes between data and code**
- **Cacheability (CACHE#)**
    - **Read cycle: indicates internal cacheability**
    - **Write cycle: burst write-back**
- **Bus lock (LOCK#)**
    - **Used in read-modify-write cycle**
    - **Useful in implementing semaphores**
- **Interrupt (INTR)**
    - **External interrupt signal**
- **Non maskable interrupt (NMI)**
    - **External NMI signal**
- **Clock (CLK)**
    - **System clock signal**
- **Bus ready (BRDY#)**
    - **Used to extend the bus cycle**
        - **Introduces wait states**
- **Bus request (BREQ)**
    - **Used in bus arbitration**
- **Backoff (BOFF#)**
    - **Aborts all pending bus cycles and floats the bus**
    - **Useful to resolve deadlock between two bus masters**
- **Bus hold (HOLD)**
    - **Completes outstanding bus cycles and floats bus**
    - **Asserts HLDA to give control of bus to another master**
- **Bus hold acknowledge (HLDA)**
    - **Indicates the Pentium has given control to another local master**
    - **Pentium continues execution from its internal caches**
- **Cache enable (KEN#)**
    - **If asserted, the current cycle is transformed into cache line fill**
- **Write-back/Write-through (WB/WT#)**
    - **Determines the cache write policy to be used**
- **Reset (RESET)**
    - **Resets the processor**
    - **Starts execution at FFFFFFF0H**
    - **Invalidates all internal caches**
- **Initialization (INIT)**
    - **Similar to RESET but internal caches and FP registers are not flushed**
    - **After powerup, use RESET (not INIT)**

- **Four 32-bit registers can be used as**
  - **Four 32-bit register (EAX, EBX, ECX, EDX)**
  - **Four 16-bit register (AX, BX, CX, DX)**
  - **Eight 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL)**
- **Some registers have special use**
  - **ECX for count in loop instructions**
  -

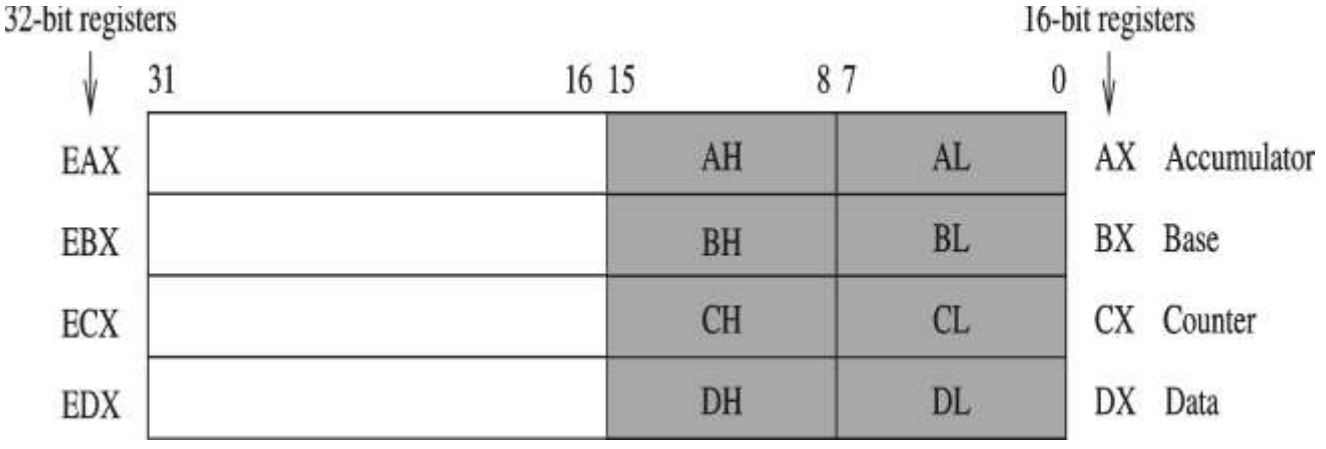


Fig 3.2: General purpose registers

- **Two index registers**
  - **16- or 32-bit registers**
  - **Used in string instructions**
    - **Source (SI) and destination (DI)**
  - **Can be used as general-purpose data registers**
  - **Two pointer registers**
  - **16- or 32-bit registers**
  - **Used exclusively to maintain the stack**

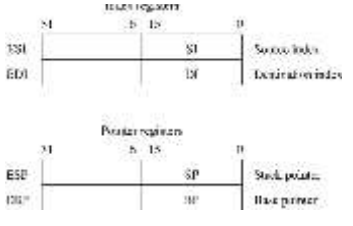


Fig 3.3. Index and pointer registers

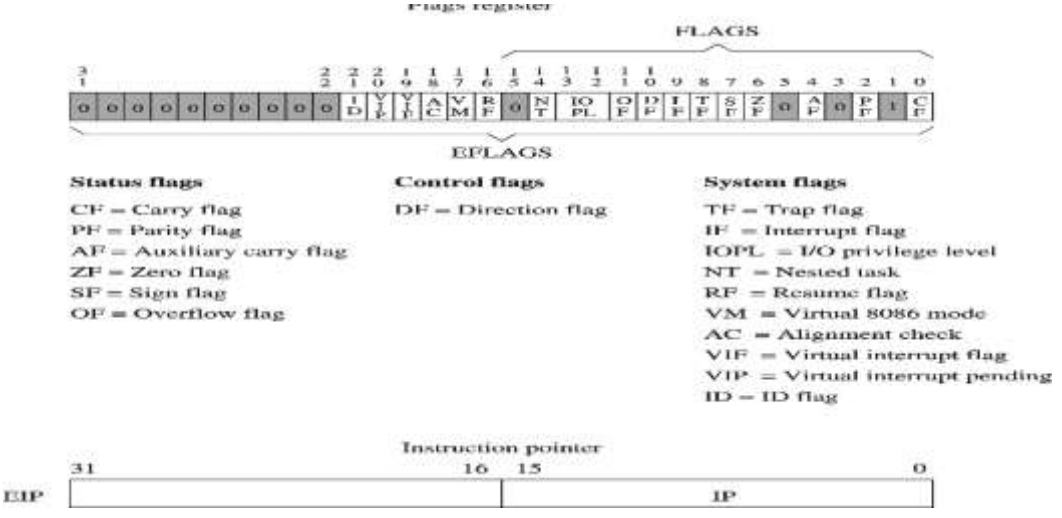


Fig 3.4: Flag register and instruction pointer

- **Control registers**
  - **(E)IP**
    - **Program counter**
  - **(E) FLAGS**
    - **Status flags**
      - **Record status information about the result of the last arithmetic/logical instruction**
    - **Direction flag**
      - **Forward/backward direction for data copy**
    - **System flags**
      - **IF : interrupt enable**

TF : Trap flag (useful in single-stepping)

The FLAG register is the status register in Intel x86 microprocessors that contains the current state of the processor. This register is 16 bits wide. Its successors, the EFLAGS and RFLAGS registers, are 32 bits and 64 bits wide, respectively.
Conditional Flag
Carry flag: indication of overflow condition for unsigned integer and hold carry
Auxiliary flag: indication of carry/borrow from lower nibble to higher nibble (D3 to D4)
Parity flag: indication of parity of the result, 0-ODD, 1-EVEN
Zero flag: set if the result is zero, Z-1 result is 0, Z-0 result is non zero
Sign flag: in signed magnitude format if the result of the operation is negative sign flag is set, S-1 result is NEGATIVE, S-0 result is POSITIVE

- **Trap flag: if set, then debugging operation is allowed**
- **Interrupt flag: control the operation of interrupt request**
- **[INTR PIN-1 Enabled,0-Disabled]**
- **STI- set by this instruction, CLI- cleared by this instruction**
- **Direction flag: used in string operation-if set the string byte is accessed from higher memory address to lower memory address**
- **Flags for 32,64 bit**
- **IOPL(I/O privilege level):**
- **Used to select the privilege level for I/O devices**
- **If the current privilege level is 00 , I/O execute without hindrance**
- **If 11 then an interrupt occur and suspended the execution**
- **Nested task(NT)- indicate whether the task is nested with other or not**
- **RF(resume): resume flag is used with debugging to control the resumption of execution after the next instruction**

- **VM(virtual mode)- allow the system program to execute multiple DOS operation**
- **AC(alignment check)- if the word or double word is addressed on a non-word or non-double word boundary**
- **VIF(virtual interrupt flag)-copy the interrupt flag bit (used in Pentium 4)**
- **VIP(virtual interrupt pending)-give the information about virtual mode interrupt. Set if interrupt if pending**
- **ID(identification)- The ID flag indicate that Pentium 4 microprocessor support the CPUID or not- CPUID instruction provides the system with information about the microprocessor such as its version number and manufacturer**
- **Segment register**
    - **Six 16-bit registers**
    - **Support segmented memory architecture**
    - **At any time, only six segments are accessible**
    - **Segments contain distinct contents**
        - **Code**
        - **Data**
        - **Stack**

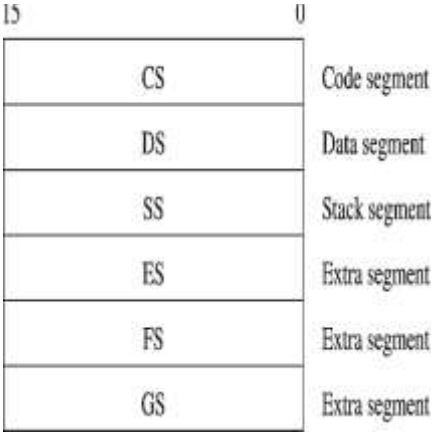
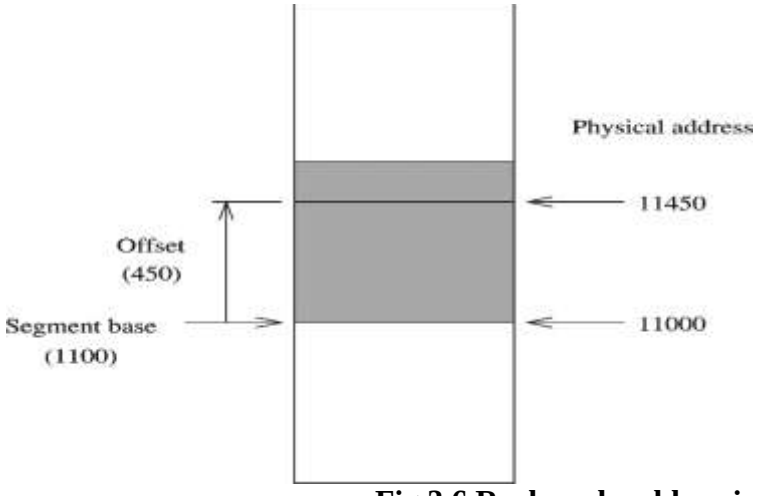


Fig 3.5 Segment Register

- **Pentium supports two modes**
- **Real mode**
- **Uses 16-bit addresses**
- **Runs 8086 programs**
- **Pentium acts as a faster 8086**
- **Protected mode**
- **32-bit mode**
- **Native mode of Pentium**
- **Supports segmentation and paging**
- **Segmented organization**
- **16-bit wide segments**
- **Two components**
- **Base (16 bits)**
- **Offset (16 bits)**
- **Two-component specification is called *logical address***
- **Also called *effective address***
- **20-bit *physical address***
-



- **Fig 3.6 Real mode addressing**

Conversion from logical to physical addresses

11000 (add 0 to base)
+ 450 (offset)
11450 (physical address)

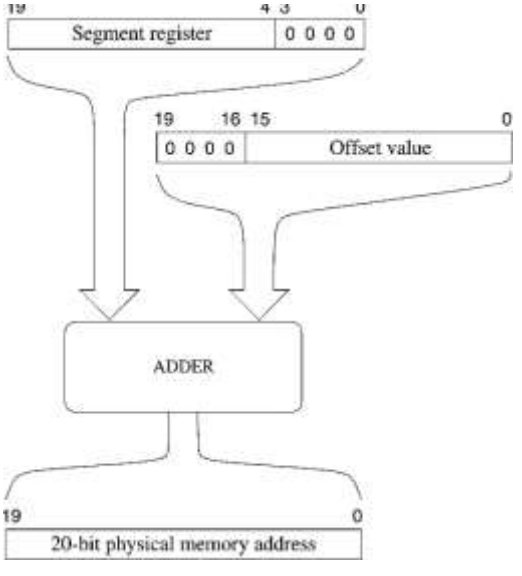


Fig 3.7 Physical address calculation

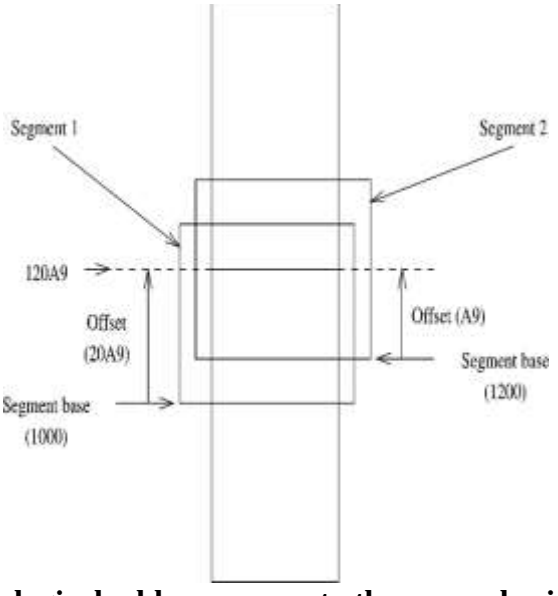


Fig 3.8 Two logical addresses map to the same physical address

- **Programs can access up to six segments at any time**
- **Two of these are for**
  - **Data**
  - **Code**
- **Another segment is typically used for**
  - **Stack**
- **Other segments can be used for**

data, code,..

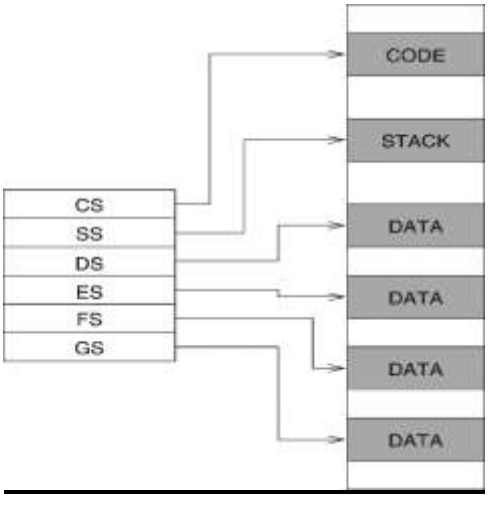


Fig 3.9 Segment register

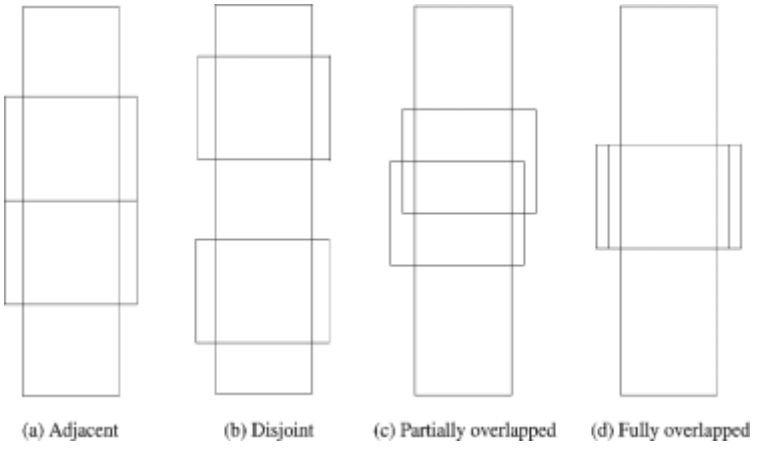


Fig 3.10 Types of Segment register

- **Supports sophisticated segmentation**
- **Segment unit translates 32-bit logical address to 32-bit linear address**
- **Paging unit translates 32-bit linear address to 32-bit physical address**
    - **If no paging is used**
        - **Linear address = physical address**

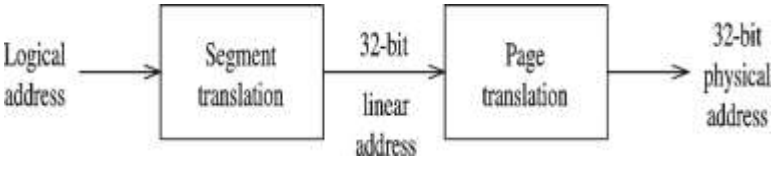


Fig 3.11 Linear to physical address

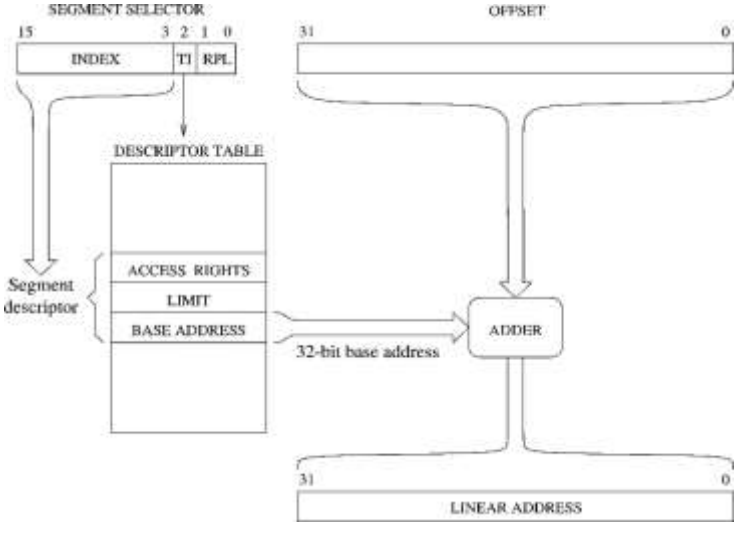


Fig 3.12 Address translation

- **Index**
    - **Selects a descriptor from one of two descriptor tables**
        - **Local**
        - **Global**
- **Table Indicator (TI)**
    - **Select the descriptor table to be used**
        - **0 = Local descriptor table**

- 1 = Global descriptor table

- **Requestor Privilege Level (RPL)**
  - **Privilege level to provide protected access to data**
    - **Smaller the RPL, higher the privilege level**
  - **Visible part**
    - **Instructions to load segment selector**
      - **mov, pop, lds, les, lss, lgs, lfs**
  - **Invisible**
    - **Automatically loaded when the visible part is loaded from a descriptor table**

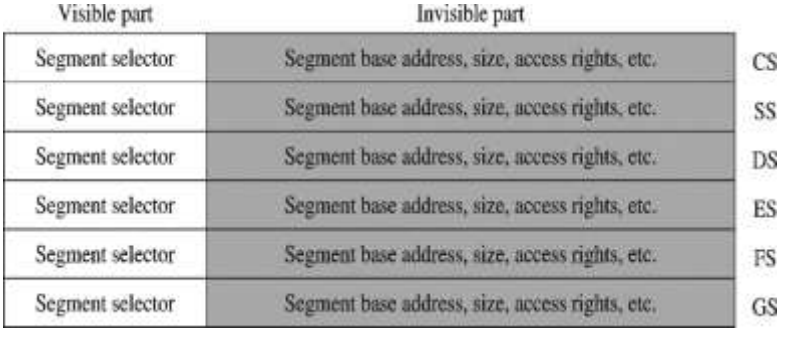


Fig 3.13 visible and invisible part

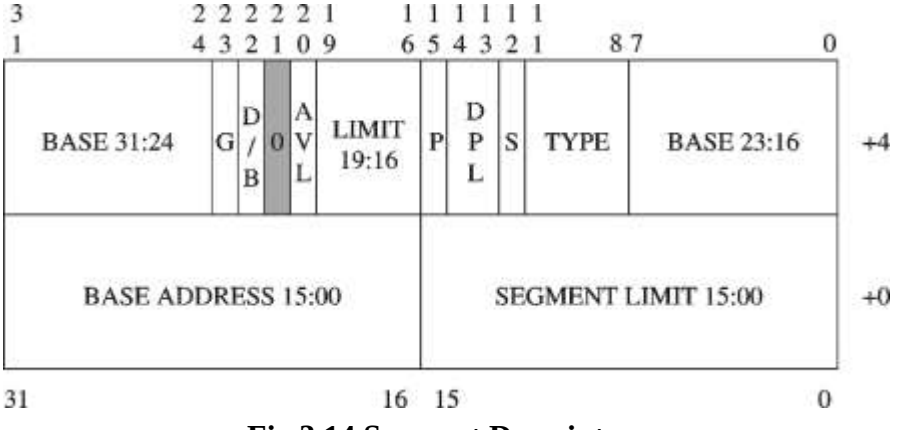


Fig 3.14 Segment Descriptors

- **Base address**
  - **32-bit segment starting address**
- **Granularity (G)**
  - **Indicates whether the segment size is in**
    - **0 = bytes, or**
    - **1 = 4KB**
- **Segment Limit**
  - **20-bit value specifies the segment size**
    - **G = 0: 1byte to 1 MB**

  G = 1: 4KB to 4GB, in increments of 4KB
- **D/B bit**

- **Code segment**
    - **D bit: default size operands and offset value**
        - **D = 0: 16-bit values**
        - **D = 1: 32-bit values**
- **Data segment**
    - **B bit: controls the size of the stack and stack pointer**
        - **B = 0: SP is used with an upper bound of FFFFH**
        - **B = 1: ESP is used with an upper bound of FFFFFFFFH**
- **Cleared for real mode**

Set for protected mode

- **S bit**
    - **Identifies whether**
        - **System segment, or**
        - **Application segment**
- **Descriptor privilege level (DPL)**
    - **Defines segment privilege level**
- **Type**
    - **Identifies type of segment**
        - **Data segment: read-only, read-write, …**
        - **Code segment: execute-only, execute/read-only, …**
- **P bit**
    - **Indicates whether the segment is present**
- **Three types of segment descriptor tables**
    - **Global descriptor table (GDT)**
        - **Only one in the system**
        - **Contains OS code and data**
        - **Available to all tasks**
    - **Local descriptor table (LDT)**
        - **Several LDTs**
        - **Contains descriptors of a program**
    - **Interrupt descriptor table (IDT**

Used in interrupt processing

- **Segmentation Models**
    - **Pentium can turn off segmentation**
    - **Flat model**
        - **Consists of one segment of 4GB**
        - **E.g. used by UNIX**
    - **Multi segment model**
        - **Up to six active segments**
        - **Can have more than six segments**

- **Descriptors must be in the descriptor table**
- **A segment becomes active by loading its descriptor into one of the segment registers**

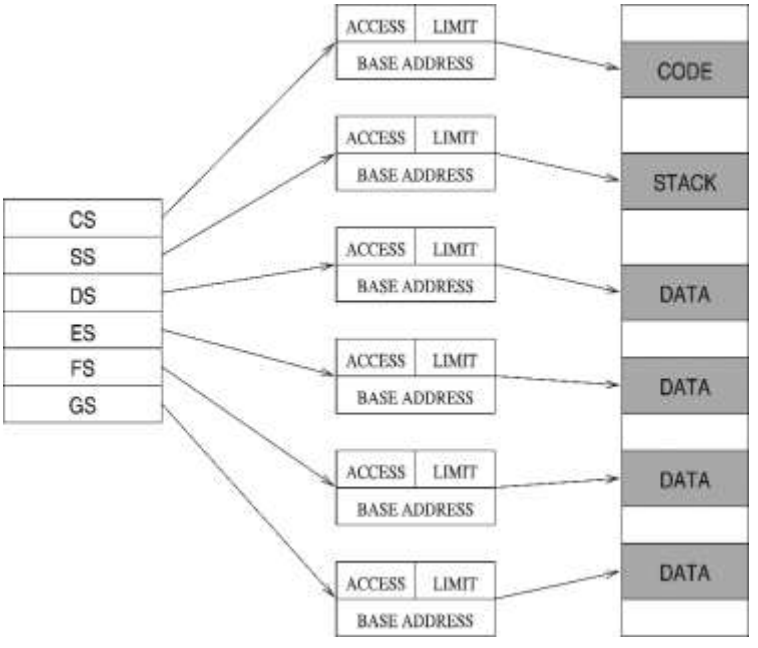


Fig 3.15 segmentation models

Mixed mode operation

- **Pentium allows mixed-mode operation**
  - **Possible to combine 16-bit and 32-bit operands and addresses**
  - **D/B bit indicates the default size**
    - **0 = 16 bit mode**
    - **1 = 32-bit mode**
  - **Pentium provides two override prefixes-to use some other segment register than the default segment for a particular code, then it is possible.**
    - **One for operands**
    - **One for addresses**
- **Pentium uses default segments depending on the purpose of the memory reference**
  - **Instruction fetch**
    - **CS register**
  - **Stack operations**
    - **16-bit mode: SP**
    - **32-bit mode: ESP**
  - **Accessing data**
    - **DS register**

Offset depends on the addressing mode

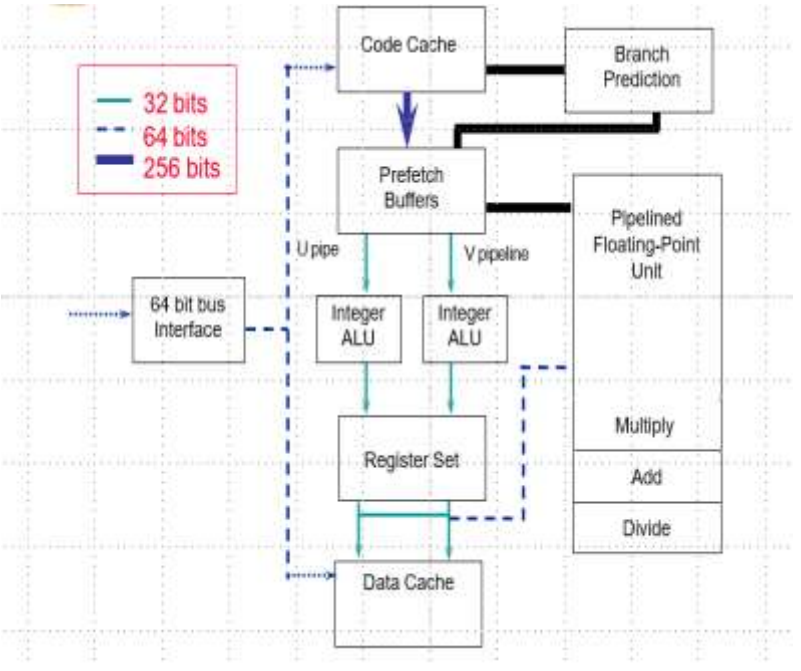


Fig 3.16 Super scalar architecture

- **Separate instruction and Data caches.**
- **Dual integer pipelines i.e. U-pipeline and V-Pipeline.**
- **Branch prediction using the branch target buffer (BTB).**
- **Pipelined floating point unit.**
- **64- bit external data bus.**
- **Even-parity checking is implemented for data bus, caches and TLBs.**
- **32- bit Superscalar and super-pipelined architecture CISC processor.**
- **32-bit address bus can address up to 4GB of physical memory.**
- **64- bit data bus so arithmetic and logical operation can be perform on 64-bit operand.**
- **Two integer pipeline U and V with two ALU's provide one-clock execution for core instructions which Improved Instructions to execute Time.**
- **Support five stage pipeline enables multiple instructions to execute in parallel with high efficiency.**
- **Two 8 KB caches memories, one for data and other for code.**
- **On- chip pipelined floating point coprocessor.**
- **Support power management feature i.e. System Management mode and Clock Control.**
- **Enhanced branch prediction buffer.**
- **On-chip memory management unit.**
- **Support multiprogramming and multitasking.**
- **Internal error Detection features.**
- **4MB pages for increased TLB Hit Rate.**
- **Support for Second level cache and Write Back MESI.**
- **Protocol in the data cache.**
- **Supports Bus cycle Pipelining, Address Parity and Internal Parity Checking, Functional Redundancy checking, Execution Tracing, Performance Monitoring.**

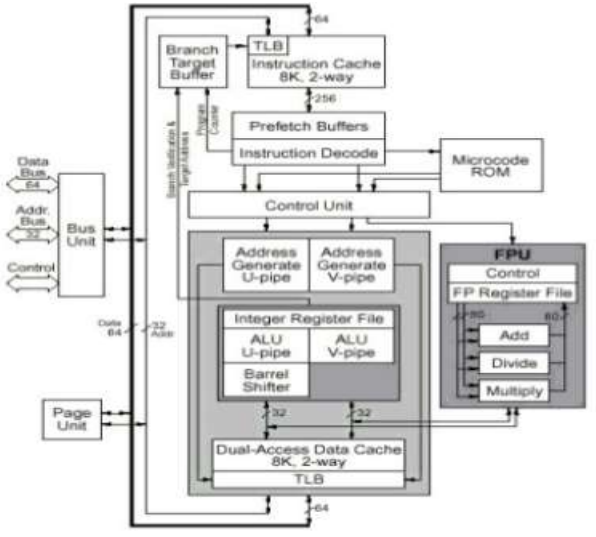


Fig 3.17 Detailed block diagram

- **It supports superscalar pipeline architecture.**
- **The Pentium processor sends two instructions in parallel to the two independent integer pipeline known as U and V pipelines for execution of multiple instructions concurrently.**
- **Thus, processor capable of parallel instruction execution of multiple instructions is known as Superscalar Machine.**
- **Each of these pipeline i.e. U and V, has 5 stages of execution i.e.**
    - **Prefetch**
    - **First Decode**
    - **Second Decode**
    - **Execute**
    - **Write Back**

1. **Pre-fetch (PF): In this stage, instructions are prefetched by prefetch buffer through U and V pipelene from the on-chip instruction cache.**
2. **First Decode (D1): In this stage, two decoders decode the instructions to generate a control word and try to pair them together so they can run in parallel.**
3. **Second Decode (D2): In this stage, the CPU decodes the control word and calculates the address of memory operand.**
4. **Execute (EX): The instruction is executed in ALU and data cache is accessed at this stage. For both ALU and data cache access requires more than one clock.**
5. **Write Back (WB): In this stage, the CPU stores result and update the flags.**

Separate data and code cache

Pentium has two separate 8KB caches for code and data as the superscalar design and branch prediction need more bandwidth than a unified cache.

The data cache has a dedicated TLB to translate linear address into physical address used by data cache.

The code cache is responsible for getting raw instructions into execution units of the Pentium processor and hence instructions are fetched from the code cache.

ADVANTAGES OF HAVING SEPARATE DATA AND CODE CACHE

- **Separate caches efficiently execute the branch prediction.**
- **Caches raise system performance i.e. an internal read request is performed more quickly than a bus cycle to memory.**
- **Separate caches also reduce the processor's use of the external bus when the same location are accessed multiple times.**
- **Separate caches for instructions and data allow simultaneous cache look-up.**
- **Up to two data references and up to 32 bytes of raw op-codes can be accessed In one clock**
- **BRANCH PREDICTION**
- **Branch prediction is used to predict the most likely set of instruction to be executed and prefetch to make them available to the pipeline as and when they are called.**
- **Hence the Pentium processor incorporate a branch target buffer (BTB), which is an associative memory used to improve the performance if it takes the branch instrction.**
- **Branch instructions of Pentium processor change the normal sequential control flow of the program execution and may stall the pipelined execution in the pentium system.**
- **Branches instruction is of two types i.e. conditional and unconditional branch.**
- **During the conditional branching, the CPU has two wait till the execution stage to determine whether the condition is satisfied or not.**
- **When the condition satisfies, a branching is to be taken using branch prediction algorithm for speed up of the instruction execution.**
- **In Pentium Processor, BTB can have 256 entries which contains the branch target address for previously executed branches.**
- **The BTB is four ways set associative on-chip memory.**
- **Whenever the branching occurs, the CPU checks the branch instruction address and the destination address in the BTB.**
- **When the instruction is decoded, the CPU searches the branch target buffer to decide whether any entry exists for a corresponding branch instruction.**
- **If BTB is hit, i.e. if BTB exist such entries, then the CPU use the history to decide whether the branch will be taken or not.**
- **If the entry exist in its previous history in BTB to take the branch, the CPU fetches the instruction from the target address and decodes them.**
- **If the branch prediction is correct, the process continue else the CPU clears the pipeline and fetches from the appropriate target address.**
- **Floating point unit:**
- **The Pentium contains an on chip floating point unit that provides significant floating point performance advantage over previous generations of processors.**
- **Providing the coprocessor onto the same chip as the processor, pentium allows faster communication and quicker execution.**

Thus many floating point instructions requires fewer clock cycles that the previous 80X87

Floating point pipeline

- **The floating point pipeline of Pentium consists of eight stages which are used to speedup the execution of floating point unit. Hence, it is necessary floating point pipeline.**
- **These are prefetch, first decode, second decode, operand fetch, first execute, second execute, write float and error reporting.**

- **Floating point unit has eight stage pipeline which gives a single cycle execution for many of floating point instructions such as floating adds, subtract, multiply and compare**
- **The stages and their functions are given below:**

1. **PF: Instructions are prefetched from the on chip instruction cache.**
2. **D1: Instruction decode to generate control word. A single control word causes direct execution of an instruction and complex instruction require micro-coded control sequence.**
3. **D2: Address of memory resident operand are calculated.**
4. **Ex: In this stage, register read, memory read or memory write operation is performed to access an operand as required by the instruction.**
5. **X1: In this stage, the floating point data from register or memory is written into floating point register or memory is written into floating point register and converted to floating point format before loaded into the floating point unit.**
6. **X2: In this stage, the floating point operation is performed by floating point unit.**
7. **WF: In this stage, results or floating point operation are rounded and the written to the destination floating point register.**
8. **ER: In this stage, if any error is occurred during floating point execution, then it is reported and FPU status word is updated.**

- **There are six floating point exception condition while execution floating point instructions are available in status word register of Pentium FPU.**
- **Floating point exceptions:**
- **These exceptions are:**

1. **Invalid Operations:**
   - **Stack overflow or underflow**
   - **Invalid arithmatic operation**

   This exception occurs when stack fault flag (SF) of the FPU status word indicates the type of operation i.e. stack overflow or Underflow for SF=1 and an arithmetic instruction has encountered an invalid operand for SF=1.
   - **Divide by zero: This exception occurs whenever an instruction attempts to divide a finite non-zero operand by 0.**
   - **De-normalized operand exception: The de-normal operand exception occurs if an arithmetic instruction attempts to operate on a de-normal operand or if an attempt is made to load de-normal single or double real value into an FPU register.**
   - **Numeric flow exception: This exception occurs whenever the rounded result of an arithmetic instruction is less than the smallest possible normalized, finite value that will fit into the real format of the destination operand.**
   - **Inexact result (Precision) Exception: This exception occurs if the result of an operation is not exactly representable in the destination format.**

80386	Pentium
32- bit integer core CPU with 32 bit Data bus	32 bit CPU with 64-bit data bus
No superscalar architecture and single cycle execution	Superscalar architecture i.e. two pipelined Integer Units are capable of 2 Instructions per clock
Ni internal cache available for data and code	Separate 8KB code and 8KB data cache available
80386 does not support branch prediction	Advanced design feature i.e. Dynamic branch Prediction
One integer Alu	Two integer ALU
Operating frequency are 20 MHz to 66 MHz	Operating frequency 60 MHz and more
FPU is non- pipelined as it is an external device 80387	FPU is pipelined as it is in built in pentium

Fig 3.18 Difference between 80386 and Pentium

- **Need for Branch prediction**
- **The gain produced by Pipelining can be reduced by the presence of program transfer instructions eg JMP, CALL, RET etc**
- **They change the sequence causing all the instructions that entered the pipeline after program transfer instructions invalid**
- **Thus no work is done as the pipeline stages are reloaded.**
- **Branch prediction logic:**
- **To avoid this problem, Pentium uses a scheme called Dynamic Branch Prediction. In this scheme, a prediction is made for the branch instruction currently in the pipeline. The prediction will either be taken or not taken. If the prediction is true then the pipeline will not be flushed and no clock cycles will be lost. If the prediction is false then the pipeline is flushed and starts over with the current instruction.**
- **It is implemented using 4 way set associated cache with 256 entries. This is called Branch Target Buffer (BTB). The directory entry for each line consists of:**
- **Valid bit: Indicates whether the entry is valid or not.**
- **History bit: Track how often bit has been taken.**
- **Source memory address is from where the branch instruction was fetched. If the directory entry is valid then the target address of the branch is stored in corresponding data entry in BTB.**
- **BTB is a lookaside cache that sits to the side of Decode Instruction(DI) stage of 2 pipelines and monitors for branch instructions.**
- **The first time that a branch instruction enters the pipeline, the BTB uses its source memory to perform a lookup in the cache.**
- **Since the instruction was never seen before, it is BTB miss. It predicts that the branch will not be taken even though it is unconditional jump instruction.**
- **When the instruction reaches the EU(execution unit), the branch will either be taken or not taken. If taken, the next instruction to be executed will be fetched from the branch**

target address. If not taken, there will be a sequential fetch of instructions.

- **When a branch is taken for the first time, the execution unit provides feedback to the branch prediction. The branch target address is sent back which is recorded in BTB.**
- **A directory entry is made containing the source memory address and history bit is set as strongly taken.**
-

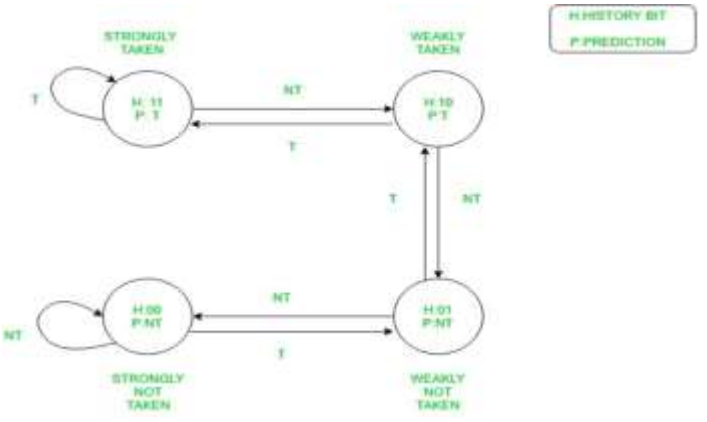


Fig 3.19 Branch prediction logic

HISTORY BITS	RESULTING DESCRIPTION	PREDICTION MADE	IF BRANCH TAKEN	IF BRANCH NOT TAKEN
11	Strongly Taken	Branch Taken	Remains in same state	Downgraded to weakly taken
10	Weakly Taken	Branch Taken	Upgraded to strongly taken	Downgraded to weakly not taken
01	Weakly Not Taken	Branch Not Taken	Upgraded to weakly taken	Downgraded to strongly not taken
00	Strongly Not Taken	Branch Not Taken	Upgraded to weakly not taken	Remains in same state

Fig 3.20 Branch prediction logic upgradation

PENTIUM DEBUG REGISTER

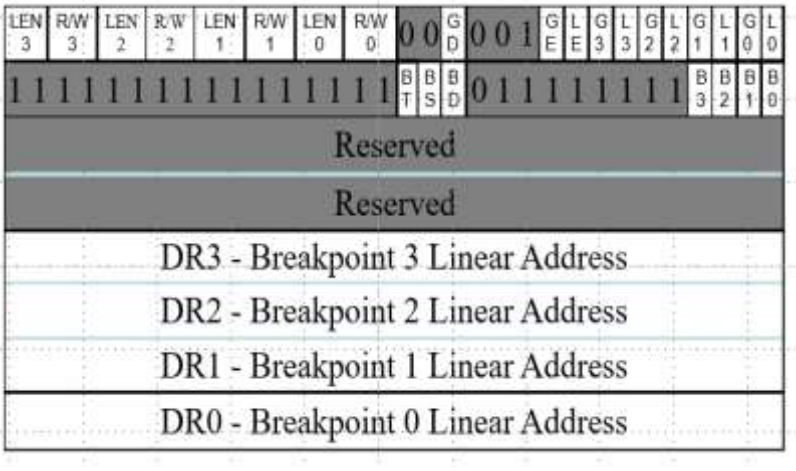


Fig 3.21 Debug Register

For each address in registers DR0-DR3, the corresponding fields R/W0 through R/W3 specify the type of action that should cause a breakpoint. The processor interprets these bits as follows:

00 -- Break on instruction execution only

01 -- Break on data writes only

10 -- undefined

11 -- Break on data reads or writes but not instruction fetches

Fields LEN0 through LEN3 specify the length of data item to be monitored. A length of 1, 2, or 4 bytes may be specified. The values of the length fields are interpreted as follows:

00 -- one-byte length

01 -- two-byte length

10 -- undefined

11 -- four-byte length

CR4

- **The Pentium uses Control Register 4 to activate certain extended features of the processor, while still allowing for backward compatibility of software written for earlier Intel x86 processors**
- **An example: Debug Extensions (DE-bit)**
- **DEBUG REGISTER DR0-DR3**
- **Each of these registers contains the linear address associated with one of four breakpoint conditions. Each breakpoint condition is further defined by bits in DR7.**
- **The debug address registers are effective whether or not paging is enabled. The addresses in these registers are linear addresses. If paging is enabled, the linear addresses are translated into physical addresses by the processor's paging mechanism. If paging is not enabled, these linear addresses are the same as physical addresses.**
- **Note that when paging is enabled, different tasks may have different linear-to-physical address mappings. When this is the case, an address in a debug address register may be relevant to one task but not to another. For this reason it has both global and local enable bits in DR7. These bits indicate whether a given debug address has a global (all tasks) or local (current task only) relevance.**
- **DEBUG REGISTER DR7**

For each address in registers DR0-DR3, the corresponding fields R/W0 through R/W3 specify the type of action that should cause a breakpoint. The processor interprets these bits as follows:

- **00 -- Break on instruction execution only**
- **01 -- Break on data writes only**
- **10 -- undefined**
- **11 -- Break on data reads or writes but not instruction fetches**
- **Fields LEN0 through LEN3 specify the length of data item to be monitored. A length of 1, 2, or 4 bytes may be specified. The values of the length fields are interpreted as follows:**
- **00 -- one-byte length**
- **01 -- two-byte length**

- **10 -- undefined**
- **11 -- four-byte length**
- **If RWn is 00 (instruction execution), then LENn should also be 00. Any other length is undefined.The low-order eight bits of DR7 (L0 through L3 and G0 through G3) selectively enable the four address breakpoint conditions. There are two levels of enabling: the local (L0 through L3) and global (G0 through G3) levels. The local enable bits are automatically reset by the processor at every task switch to avoid unwanted breakpoint conditions in the new task. The global enable bits are not reset by a task switch; therefore, they can be used for conditions that are global to all tasks.**
- **The LE and GE bits control the "exact data breakpoint match" feature of the processor. If either LE or GE is set, the processor slows execution so that data breakpoints are reported on the instruction that causes them. It is recommended that one of these bits be set whenever data breakpoints are armed. The processor clears LE at a task switch but does not clear GE.**
- **DEBUG REGISTER DR6**
- **When the processor detects an enabled debug exception, it sets the low-order bits of this register (B0 thru B3) before entering the debug exception handler. Bn is set if the condition described by DRn, LENn, and R/Wn occurs. (Note that the processor sets Bn regardless of whether Gn or Ln is set. If more than one breakpoint condition occurs at one time and if the breakpoint trap occurs due to an enabled condition other than n, Bn may be set, even though neither Gn nor Ln is set.)**
- **The BT bit is associated with the T-bit (debug trap bit) of the TSS (refer to 7 for the location of the T-bit). The processor sets the BT bit before entering the debug handler if a task switch has occurred and the T-bit of the new TSS is set. There is no corresponding bit in DR7 that enables and disables this trap; the T-bit of the TSS is the sole enabling bit.**
- **The BS bit is associated with the TF (trap flag) bit of the EFLAGS register. The BS bit is set if the debug handler is entered due to the occurrence of a single-step exception. The single-step trap is the highest-priority debug exception; therefore, when BS is set, any of the other debug status bits may also be set.**
- **The BD bit is set if the next instruction will read or write one of the eight debug registers and ICE-386 is also using the debug registers at the same time.**
- **Note that the bits of DR6 are never cleared by the processor. To avoid any confusion in identifying the next debug exception, the debug handler should move zeros to DR6 immediately before returning.**
- **I/O ADDRESS SPACE**
- **I/O Address Space is limited to 64 Kbytes (0000H-**

FFFFH).

- **This limit is imposed by a 16 bit CPU Register.**

– **A 16 bit register can store up to FFFFH (1111 1111 1111 1111 y).**

- **Which CPU Register limits I/O space to 64K**

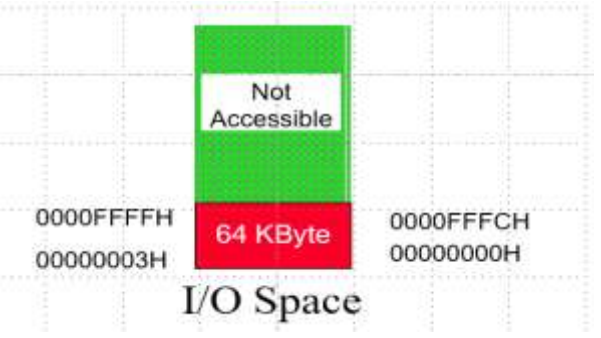


Fig 3.22 I/O address space

- **Address bus: The microprocessor provides an address to the memory & I/O chips.**
  - **The number of address lines determines the amount of memory supported by the processor.**
  - **A31:A3 Address bus lines (output except for cache snooping) determines where in the 4GB memory space or 64K I/O space the processor is accessing.**
- **The Pentium address consists of two sets of signals:**
  - **Address Bus (A31:3)**
  - **Byte Enables (BE7#:0#)**
- **Since address lines A2:0 do not exist on the Pentium, the CPU uses A31:3 to identify a group of 8 locations known as a Quadword (8 bytes -- also know as a "chunk").**
  - **Without A2:0, the CPU is only capable of outputting every 8th address. (e.g. 00H, 08H, 10H, 18H, 20H, 28H, etc.)**

A2:0 could address from 000 to 111 in binary (0-7H)

Addr to be Output (in Hex)	Addr Placed on CPU Addr Bus	Addr to be Output (in Hex)	Addr Placedon CPU Addr Bus
0000 0000	0000 0000	0000 0008	0000 0008
0000 0001	0000 0000	0000 0009	0000 0008
0000 0002	0000 0000	0000 000a	0000 0008
0000 0003	0000 0000	0000 000b	0000 0008
0000 0004	0000 0000	0000 000c	0000 0008
0000 0005	0000 0000	0000 000d	0000 0008
0000 0006	0000 0000	0000 000e	0000 0008
0000 0007	0000 0000	0000 000f	0000 0008

Fig 3.23 Output on Address Lines for addresses within a Quadword Addresses 00000000 - 000000F

Addr to be Output (in Hex)	Addr Placed on CPU Addr Bus	Addr to be Output (in Hex)	Addr Placed on CPU Addr Bus
0000 0010	0000 0010	0000 0018	0000 0018
0000 0011	0000 0010	0000 0019	0000 0018
0000 0012	0000 0010	0000 001a	0000 0018
0000 0013	0000 0010	0000 001b	0000 0018
0000 0014	0000 0010	0000 001c	0000 0018
0000 0015	0000 0010	0000 001d	0000 0018
0000 0016	0000 0010	0000 001e	0000 0018
0000 0017	0000 0010	0000 001f	0000 0018

Fig 3.24 Output on Address Lines for addresses within a Quadword Addresses 00000000 - 000000F

- **The Pentium uses Byte Enables to address locations within a QWORD.**
  - **BE7#:BEO# (outputs): Byte enable lines - to enable each of the 8 bytes in the 64-bit data path.**
  - **In effect a decode of the address lines A2-A0 which the Pentium does not generate.**

Which lines go active depends on the address, and whether a byte, word, double word or quad word is required

Byte Enable	Data Path Used	Location in Qword
BE0#	D07:00	FIRST
BE1#	D15:08	SECOND
BE2#	D23:16	THIRD
BE3#	D31:24	FOURTH
BE4#	D39:32	FIFTH
BE5#	D47:40	SIXTH
BE6#	D55:48	SEVENTH
BE7#	D63:56	EIGHT

Fig 3.25. Relationship of Byte Enables to Locations Addressed within a QWORD

Data bus: The data bus provides a path for data to flow.

- **The data can flow to/from the microprocessor during a memory or I/O operation.**
  - **D63:DO (bi-directional): The 64-bit data path to or from the processor. The signal W/R# distinguishes direction.**
- **Control bus: The control bus is used by the CPU to tell the memory and I/O chips what the CPU is doing.**
  - **Typical control bus signals are these:**
    - **ADS# (output): Signals that the processor is beginning a bus cycle:**
    - **BRDY# (input): This signal ends the current bus cycle and is**

used to extend bus cycles.

PENTIUM MEMORY MANAGEMENT

- **Pentium General Memory Management**
- **Pentium Paging**
- **Segmentation**
- **Pentium Segmentation**

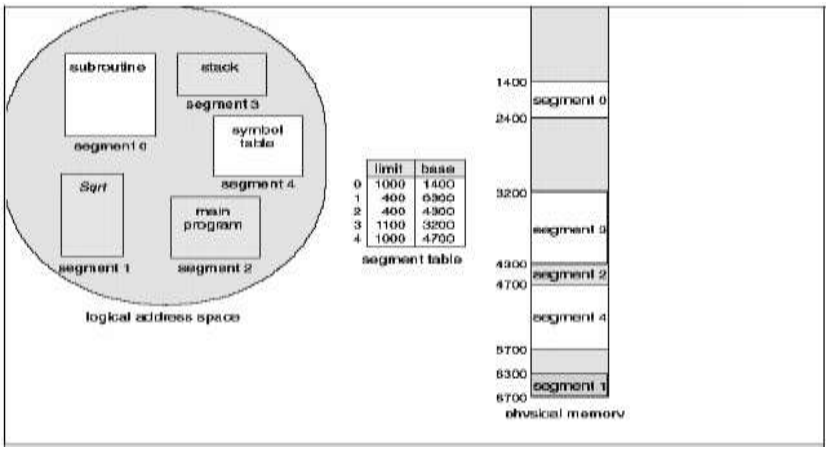


Fig 3.26 Segmentation

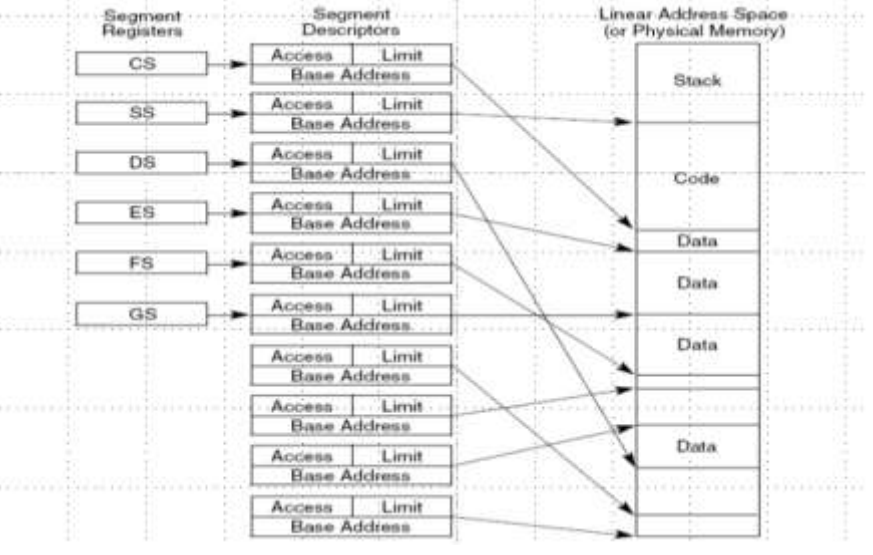


Fig 3.27 Protected mode memory management

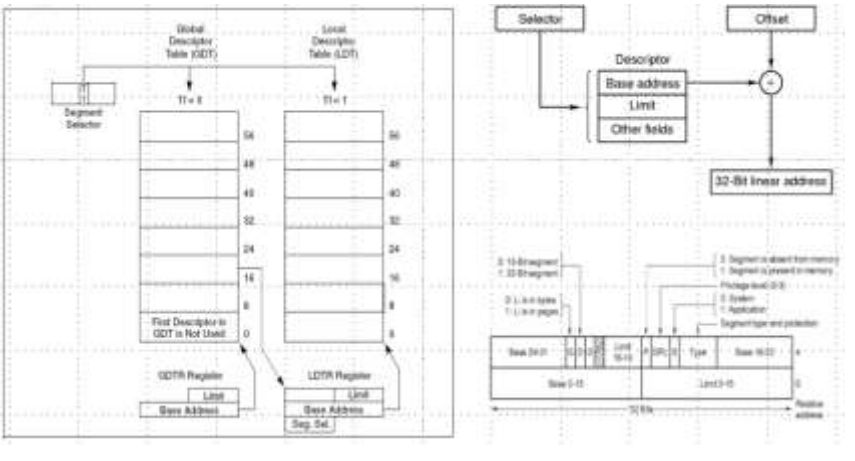


Fig 3.28 Pentium segmentation

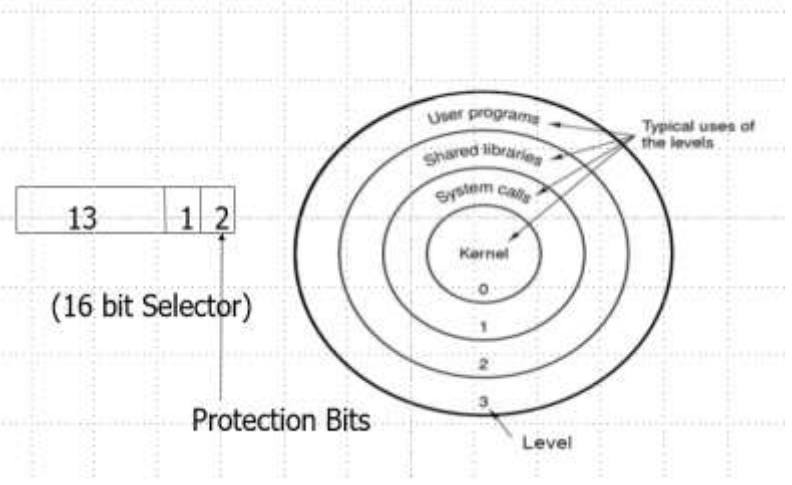


Fig 3.29 protection level mechanism

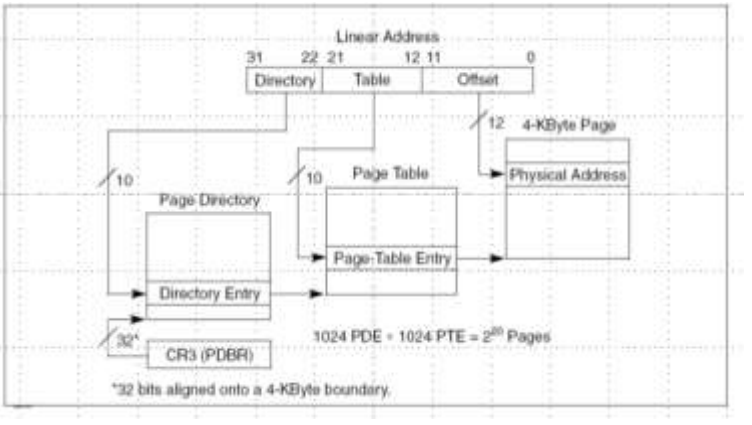


Fig 3.30 Pentium paging

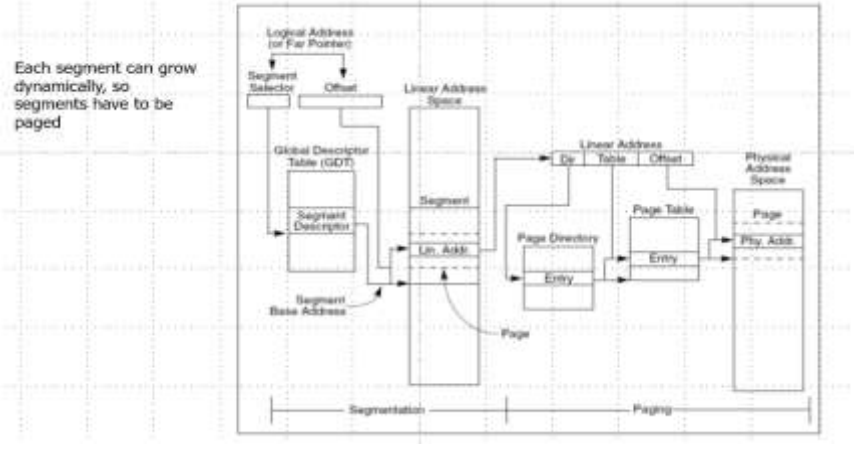


Fig 3.31 Pentium general memory management

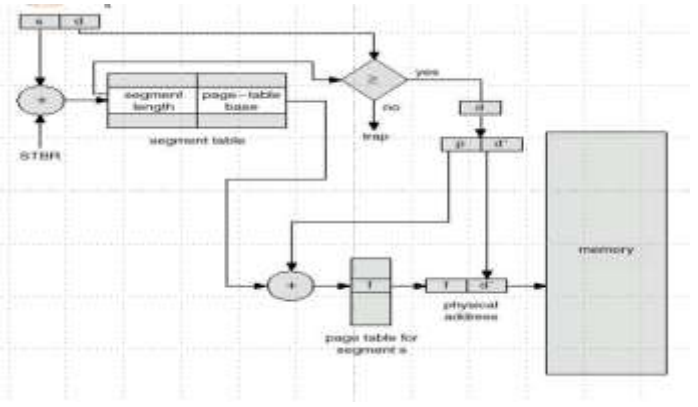


Fig 3.32 Segmentation with paging example

PENTIUM INSTRUCTION SET

Generalities:

-- Most of the instructions have exactly 2 operands.

If there are 2 operands, then one of them will be required to use register mode, and the

other will have no restrictions on its addressing mode.

There are most often ways of specifying the same instruction for 8-, 16-, or 32-bit operands

Meanings of the operand specifications:

reg - register mode operand, 32-bit register

reg8 - register mode operand, 8-bit register

r/m - general addressing mode, 32-bit

r/m8 - general addressing mode, 8-bit

immed - 32-bit immediate is in the instruction

immed8 - 8-bit immediate is in the instruction

m - symbol (label) in the instruction is the effective address

DATA MOVEMENT INSTRUCTIONS

Generalities:

-- Most of the instructions have exactly 2 operands.

If there are 2 operands, then one of them will be required to use register mode, and the

other will have no restrictions on its addressing mode.

There are most often ways of specifying the same instruction for 8-, 16-, or 32-bit operands

Meanings of the operand specifications:

reg - register mode operand, 32-bit register

reg8 - register mode operand, 8-bit register

r/m - general addressing mode, 32-bit

r/m8 - general addressing mode, 8-bit

immed - 32-bit immediate is in the instruction

immed8 - 8-bit immediate is in the instruction

```
  m - symbol (label) in the instruction is the effective address
ARITHMETIC INSTRUCTIONS
add   reg, r/m              ; two's complement addition
            r/m, reg
            reg, immed
            r/m, immed
    inc   reg                ; add 1 to operand
          r/m
    sub   reg, r/m             ; two's complement subtraction
            r/m, reg
            reg, immed
            r/m, immed
    dec   reg                ; subtract 1 from operand
          r/m
    neg   r/m                ; get additive inverse of operand
mul   eax, r/m             ; unsigned multiplication
                           ; edx||eax <- eax * r/m
    imul   r/m               ; 2's comp. multiplication
                           ; edx||eax <- eax * r/m
          reg, r/m           ; reg <- reg * r/m
          reg, immed           ; reg <- reg * immed
    div   r/m                ; unsigned division
                                   ; does edx||eax / r/m
                                   ; eax <- quotient
                                   ; edx <- remainder
    idiv   r/m               ; 2's complement division
                                   ; does edx||eax / r/m
                                   ; eax <- quotient
                                   ; edx <- remainder
cmp   reg, r/m             ; sets EFLAGS based on
          r/m, immed           ; second operand - first operand
          r/m8, immed8
          r/m, immed8          ; sign extends immed8 before subtract
           EXAMPLES:
           neg [eax + 4]   ; takes doubleword at address eax+4
                           ;   and finds its additive inverse, then places
                           ;   the additive inverse back at that address
                           ;   the instruction should probably be
                           ;      neg  dword ptr [eax + 4]
           inc ecx         ; adds one to contents of register ecx, and
```

```
                        ;   result goes back to ecx
LOGICAL INSTRUCTION
not   r/m               ; logical not
    and   reg, r/m          ; logical and
          reg8, r/m8
            r/m, reg
          r/m8, reg8
            r/m, immed
            r/m8, immed8
    or    reg, r/m          ; logical or
          reg8, r/m8
            r/m, reg
          r/m8, reg8
            r/m, immed
            r/m8, immed8
xor   reg, r/m          ; logical exclusive or
          reg8, r/m8
            r/m, reg
          r/m8, reg8
            r/m, immed
            r/m8, immed8
    test  r/m, reg          ; logical and to set EFLAGS
          r/m8, reg8
            r/m, immed
            r/m8, immed8
     EXAMPLES:
         and edx, 00330000h   ; logical and of contents of register
                              ;   edx (bitwise) with 0x00330000,
                              ;   result goes back to edx
     finit                   ; initialize the FPU
    fld   m32                ; load floating point value
          m64
          ST(i)
    fldz                    ; load floating point value 0.0
    fst   m32                ; store floating point value
          m64
          ST(i)
    fstp  m32                ; store floating point value
          m64               ;   and pop ST
          ST(i)
```

fadd m32 ; floating point addition
m64
ST, ST(i)
ST(i), ST
faddp ST(i), ST ; floating point addition and pop ST

ST refers to the stack top. ST(1) refers to the register within the stack that is next to ST.

PENTIUM I/O INSTRUCTION

The only instructions which actually allow the reading and writing of I/O devices are priviledged. The OS must handle these things. But, in writing programs that do something useful, we need input and output. Therefore, there are some simple macros defined to help us do I/O.

These are used just like instructions.

put_ch r/m ; print character in the least significant
; byte of 32-bit operand
get_ch r/m ; character will be in AL
put_str m ; print null terminated string given
; by label m

CONTROL INSTRUCTION

jmp m ; unconditional jump
jg m ; jump if greater than 0
jge m ; jump if greater than or equal to 0
jl m ; jump if less than 0
jle m ; jump if less than or equal to 0

PENTIUM PRO PROCESSOR

INTRODUCTION

- **Sixth gen x86 microprocessor, introduced in November 1,1995.**
- **Successor of Intel Pentium microprocessor.**
- **Capable of dual and quad-processor configuration.**
- **36-bit address bus, supports up to 64GB memory.**
- **Has Error Correction Circuitry, can fix 1-bit and detect 2-bits of error.**
- **256KB or 512KB L2 cache, 16 KB L1 cache.**
- **The most important change in this processor was the use of dynamic execution instead of the superscalar architecture.**
- **Pipeline is divided in 3 sections.**
    - **Fetch and decode unit**

- **dispatch and execution unit**
- **retire unit**

- **Intel Pentium Pro microprocessor takes CISC instructions and converts them into RISC micro-operations.**

PENTIUM PRO FEATURES

- **The Pentium pro has a performance near about 50% higher than a Pentium of the same clock speed.**
- **Super-pipelining: 14 stages pipelining as compare to 5 stage of pentium Processor.**
- **Integrated Level 2 Cache: 256-KB static Ram on- chip coupled to the core processor through a full clock speed, 64- bit, cache bus.**
- **32- bit Optimization: Optimized for running, 32-bit code used in Windows NT.**
- **Wider Address Bus: 36 bit address bus which is used to address 2^{36}=64GB of physical address space**
- **Greater Multiprocessing: Multi-processor systems of up to 4 Pentium Pro processors.**
- **Out of order completion: Out of order execution mechanism called as dynamic execution.**
- **Superior branch prediction Unit: The branch target buffer (BTB) is double the size as compare to Pentium processor which increases its accuracy.**
- **Register renaming: Improves parallel performance of the pipelines.**
- **Speculative execution: Speculative execution reduces pipeline stall time in its RISC core.**
- **Dynamic data flow analysis: Real time analysis of the flow of data trough the processor to determine data and register dependencies and to detect opportunities for out of order instruction execution.**

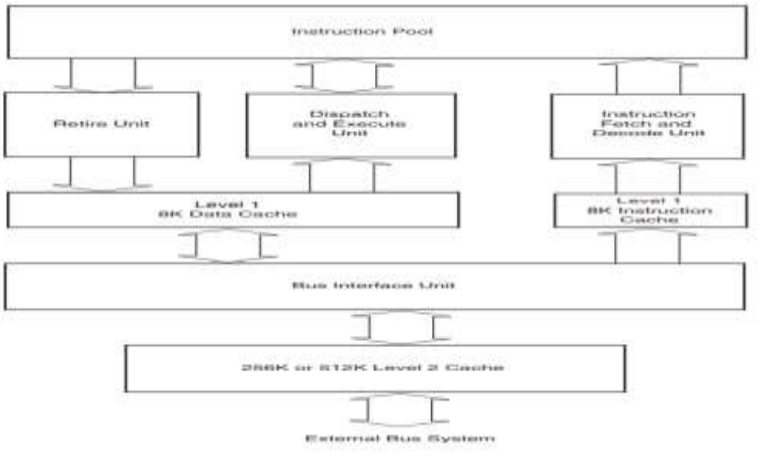


Fig 3.33 Pentium Pro block diagram

- **The level 2 cache in the Pentium Pro is either 256K bytes or 512K bytes. The integration of the level 2 cache speeds processing and reduces the number of components in a system.**
- **The bus interface unit (BIU) controls the access to the system buses through the level 2 cache. Again, the difference is that the level 2 cache is integrated. The BIU generates the memory address and control signals, and passes and fetches data or instructions to either a level 1 data cache or a level 1 instruction cache. Each cache is 8K bytes in size at**

present and may be made larger in future versions of the microprocessor. The implementation of separate caches improves performance.

- **The instruction cache is connected to the instruction fetch and decode unit (IFDU). The IFDU contains three separate instruction decoders that decode three instructions simultaneously. Once decoded, the outputs of the three decoders are passed to the instruction pool, where they remain until the dispatch and execution unit or retire unit obtains them.**
- **Also included within the IFDU is a branch prediction logic section that looks ahead in code sequences that contain conditional jump instructions. If a conditional jump is located, the branch prediction logic tries to determine the next instruction in the flow of a program.**
- **Internal structure**
- **The system bus connects to L2 cache.**
- **BIU controls system bus access via L2 cache.**
- **L2 cache is integrated in Intel Pentium Pro.**
- **BIU generates control signals and memory address.**
- **BIU fetches or passes data or instruction via L1 cache.**
- **The IFDU can decode three instructions simultaneously, and passes it to instruction pool.**
- **The IFDU has branch prediction logic.**

The DEU then executes the instructions.

- **DEU contains three execution units. Two for processing integer instruction and one for processing floating point instruction simultaneously.**
- **Lastly RU checks the instruction pool and removes decoded instructions that have been executed.**
- **RU can remove three decoded instructions per clock pulse.**
- **DYNAMIC EXECUTION**
- **Intel Pro microprocessor can anticipate jumps in the flow of instructions.**
- **While the program is being executed processor looks at instructions further ahead in the program and decides in which order to execute the instructions.**
- **If the instructions can be executed independently it will execute these instructions in optimal order rather than in the original program order.**

MEMORY SYSTEM

- **The Pentium Pro uses a 64-bit data bus to address memory organized in eight banks each contain 8G bytes of data.**
- **Pentium pro also has a built in error-correction-circuit. For that the memory system must have room for extra 8 bit number that is stored with each 64 bit number.**
- **These 8 bit numbers are used to store an error-correction code that allows the Pentium Pro to auto correct any single-bit error.**
- **A 1M X 72 is an SDRAM with ECC(error correcting code) support whereas 1M X 64 is an SDRAM without ECC.**
- **I/O SYSTEM**
- **Input-output system of Intel Pentium pro is completely portable with earlier Intel**

microprocessors.

- **A15-A3 address lines with bank enable signals are used to select actual memory banks used for I/O transfer.**
- **DRAWBACKS OF PENTIUM PRO**
- **As Intel Pentium Pro uses RISC approach the first drawback is converting instructions from CISC to RISC. It takes time to do so. So Pentium pro inevitably takes performance hit when processing instructions.**
- **Second is that out of order design can be particularly affected by 16 bit code resulting in eventual stop of process.**
- **ECC scheme causes additional cost of SDRAM that is 72 bits wide.**
- **DIFFERENCE BETWEEN PENTIUM AND PENTIUM PRO**
- **Level-2 cache is integrated in Intel Pentium Pro and not in Pentium microprocessor. This speeds up processing and reduces number of components.**
- **In Pentium unified cache holds both instructions and data, but in Pentium Pro separate cache is used for instruction and data which speeds up performance.**
- **Pentium microprocessor doesn't have jump execution unit or address generation unit as Pentium Pro has. It's one of the major changes.**
- **2M paging isn't available in Pentium microprocessor. In Pentium Pro 2M paging allows memory above 4G to be accessed.**
- **Pentium doesn't have built in error correction circuit. But in Pentium Pro ECC allows correction of one bit error and detection of two bit error.**
- **PENTIUM 4**
- **INTRODUCTION**
- **The Pentium 4 processor is Intel's new microprocessor that was introduced in November of 2000**
- **The Pentium 4 processor**
- **Has 42 million transistors implemented on Intel's 0.18μ CMOS process, with six levels of aluminum interconnect**
- **Has a die size of 217 mm^2 -A die is a small block of semiconducting material on which a given functional circuit is fabricated.**
- **Consumes 55 watts of power at 1.5 GHz**
- **3.2 GB/second system bus helps provide the high data bandwidths needed to supply data for demanding applications**
- **Implements a new Intel NetBurst microarchitecture-The technology used in Intel's Pentium 4 chips. It doubles the instruction pipeline to 20 stages, runs the ALU at twice the core frequency and improves performance in the Level 1 and 2 caches.**
- **Extends Single Instruction Multiple Data (SIMD) computational model with the introduction of Streaming SIMD Extension 2 (SSE2) and Streaming SIMD Extension 3 (SSE3) that improve performance for multi-media, content creation, scientific, and engineering applications**
- **Supports Hyper-Threading (HT) Technology**
- **Has Deeper pipeline – (20 pipeline stages)**
- **OVERVIEW**

- **Product review**
- **Specialized architectural features (NetBurst)**
- **SIMD instructional capabilities (MMX, SSE2)**
- **SHARC 2106x comparison- Super Harvard Architecture Computer is a high-performance 32-bit digital signal processor for speech, sound, graphics, and imaging applications.**
- **Reworked micro-architecture for high-bandwidth applications**
- **Internet audio and streaming video, image processing, video content creation, speech, 3D, CAD, games, multi-media, and multi-tasking user environments**
- **These are used for DSP intensive applications**

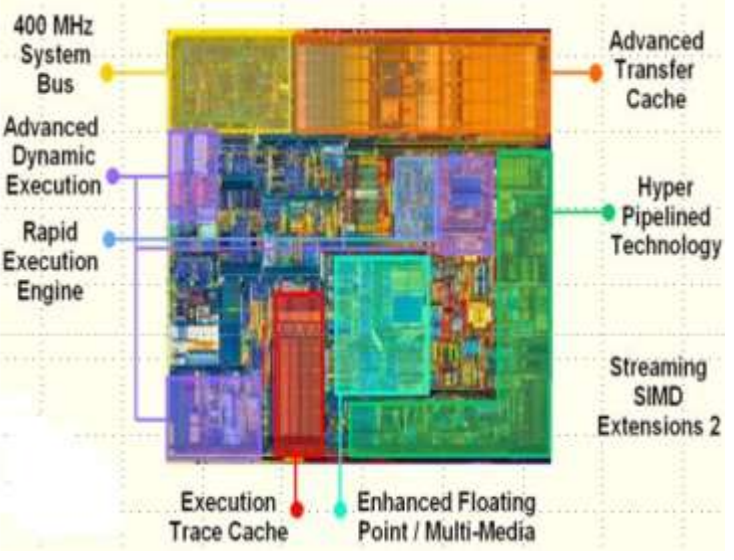


Fig 3.34 pentium 4 block diagram

HARDWARE FEATURES

- **Hyper pipelined technology**
- **Advanced dynamic execution**
- **Cache (data, L1, L2)**
- **Rapid ALU execution engines**
- **400 MHz system bus**
- **OOE(out of order Execution)**
- **Microcode ROM**
- **Advanced Transfer Cache**
- **Execution Trace Cache**

SCALAR VS SIMD

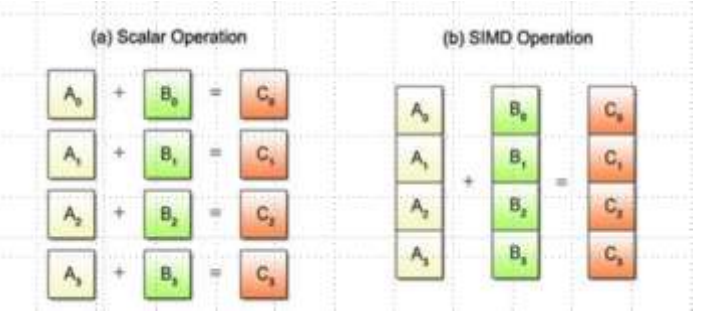


Fig 3.35 Scalar vs SIMD

PENTIUM 4 CHIP LAYOUT

- **400 MHz System Bus**
- **Advanced Transfer Cache**
- **Hyper Pipelined Technology**
- **Enhanced Floating Point/Multi-Media**
- **Execution Trace Cache**
- **Rapid Execution Engine**
- **Advanced Dynamic Execution**

400MHZ SYSTEM BUS

- **Quad Pump - On every latch, four addresses from the L2 cache are decoded into µops (micro-operations) and stored in the trace cache.**
  - **100 MHz System Bus yields 400 MHz data transfers into and out of the processor**
  - **200 MHz System Bus yields 800 MHz data transfers into and out of the processor**
- **Overall, the P4 has a data rate of 3.2 GB/s in and out of the processor.**

Which compares to the 1.06 GB/s in the PIII 133MHz system bus

ADVANCED TRANSFER CACHE

- **Handles the first 5 stages of the Hyper Pipeline**
- **Located on the die with the processor core**
- **Includes data pre-fetching**
- **256-bit interface that transfers data on each core clock**
- **256KB - Unified L2 cache (instruction + data)**
  - **8-way set associative**
  - **128 bit cache line**
    - **2 64 bit pieces**
      reads 64 bytes in one go
- **For a P4 @ 1.4 GHz the data bandwidth between the ATC and the core is 44.8 GB/s**

HYPER PIPELINED TECHNOLOGY

- **Deep 20 stage pipeline**
  - **Allows for signals to propagate quickly through the circuits**
- **Allows 126 “in-flight” instructions**
  - **Up to 48 load and 24 store instructions at one time**
- **However, if a branch is mispredicted it takes a long time to refill the pipeline and**

continue execution.

The improved (Trace Cache) branch prediction unit is supposed to make pipeline flushes rare

ENHANCED FLOATING POINT

- **Extended Instruction Set of 144 New Instructions**
- **Designed to enhance Internet and computing applications**
- **New Instructions Types**
- **128-bit SIMD integer arithmetic operations**
- **64-bit MMX technology**
- **Accelerates video, speech, encryption, imaging and photo processing**
- **128-bit SIMD double-precision floating-point operations**
- **Accelerates 3D rendering, financial calculations and scientific applications**

EXECUTION TRACE CACHE

- **Basically, the execution trace cache is a L1 instruction cache that lies direction behind the decoders.**
- **Holds the µops for the most recently decoded instructions**
- **Integrates results of branches in the code into the same cache line**
- **Stores decoded IA-32 instructions**

Removes latency associated with the CISC decoder from the main execution loops

RAPID EXECUTION ENGINE

- **Execution Core of the NetBurst microarchitecture**
- **Facilitates parallel execution of the µops by using 2 Double Pumped ALUs and AGUs**
- **D.P. ALUs handle Simple Instructions**
- **D.P. AGUs (Address Generation Unit) handles Loading/Storing of Addresses**
- **Clocked with double the processors clock.**
- **Can receive a µop every half clock**
- **1 "Slow" ALU**
- **Not double pumped**

1 MMX and 1 SSE unit

ADVANCED DYNAMIC ENGINE

- **Deep, Out-of-Order Speculative Execution Engine**
- **Ensures execution units are busy**
- **Enhanced Branch Prediction Algorithm**
- **Reduces mispredictions by 33% from previous versions**
- **Significantly improves performance of processor**

INTEL NETBURST MICROARCHITECTURE OVERVIEW

- **Designed to achieve high performance for integer and floating point computations at high clock rates**
- **Features:**
- **hyper-pipelined technology that enables high clock rates and frequency headroom (up to 10 GHz)**

- **a high-performance, quad-pumped bus interface to the Intel NetBurst microarchitecture system bus**
- **a rapid execution engine to reduce the latency of basic integer instructions**
- **out-of-order speculative execution to enable parallelism**
- **superscalar issue to enable parallelism**
- **INTEL NETBURST MICROARCHITECTURE FEATURES**
- **Hardware register renaming to avoid register name space limitations**
- **Cache line sizes of 64 bytes**
- **Hardware pre-fetch**
- **A pipeline that optimizes for the common case of frequently executed instructions**
- **Employment of techniques to hide stall penalties such as parallel execution, buffering, and speculation**

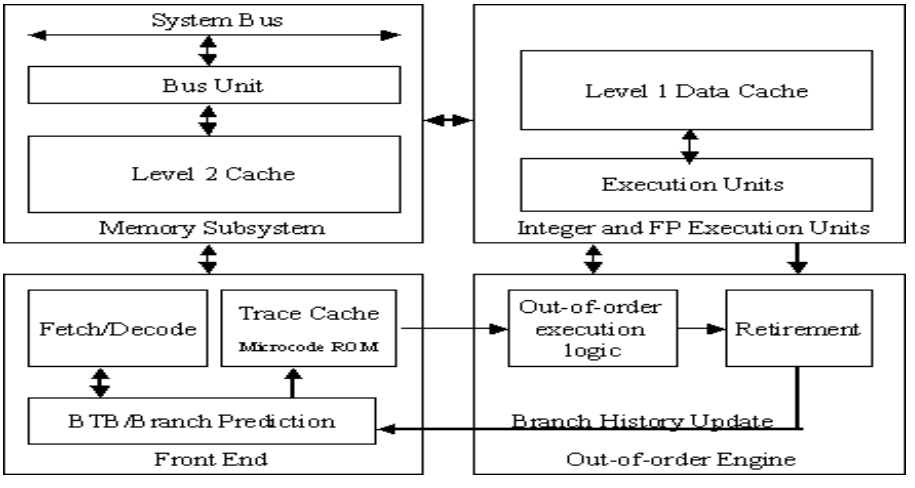


Fig 3.36 Pentium 4 basic block diagram

Four main sections:

- **The In-Order Front End**
- **The Out-Of-Order Execution Engine**
- **The Integer and Floating-Point Execution Units**

The Memory Subsystem

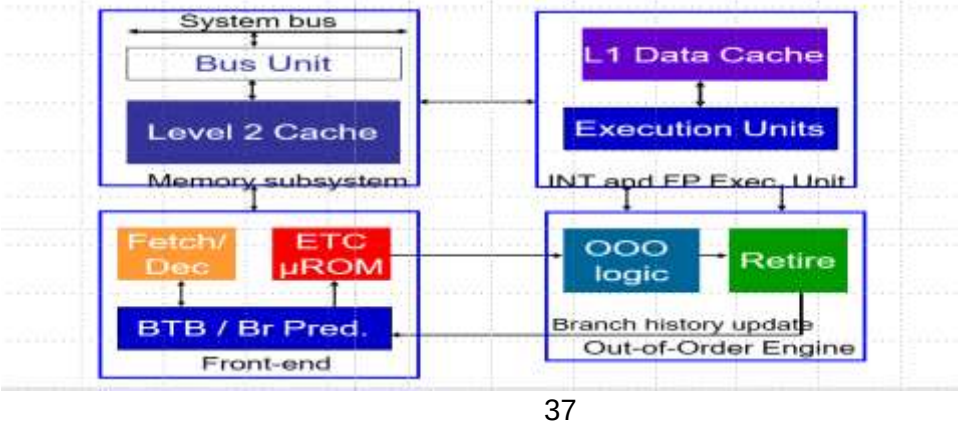


Fig 3.37 Pentium 4 Netburst

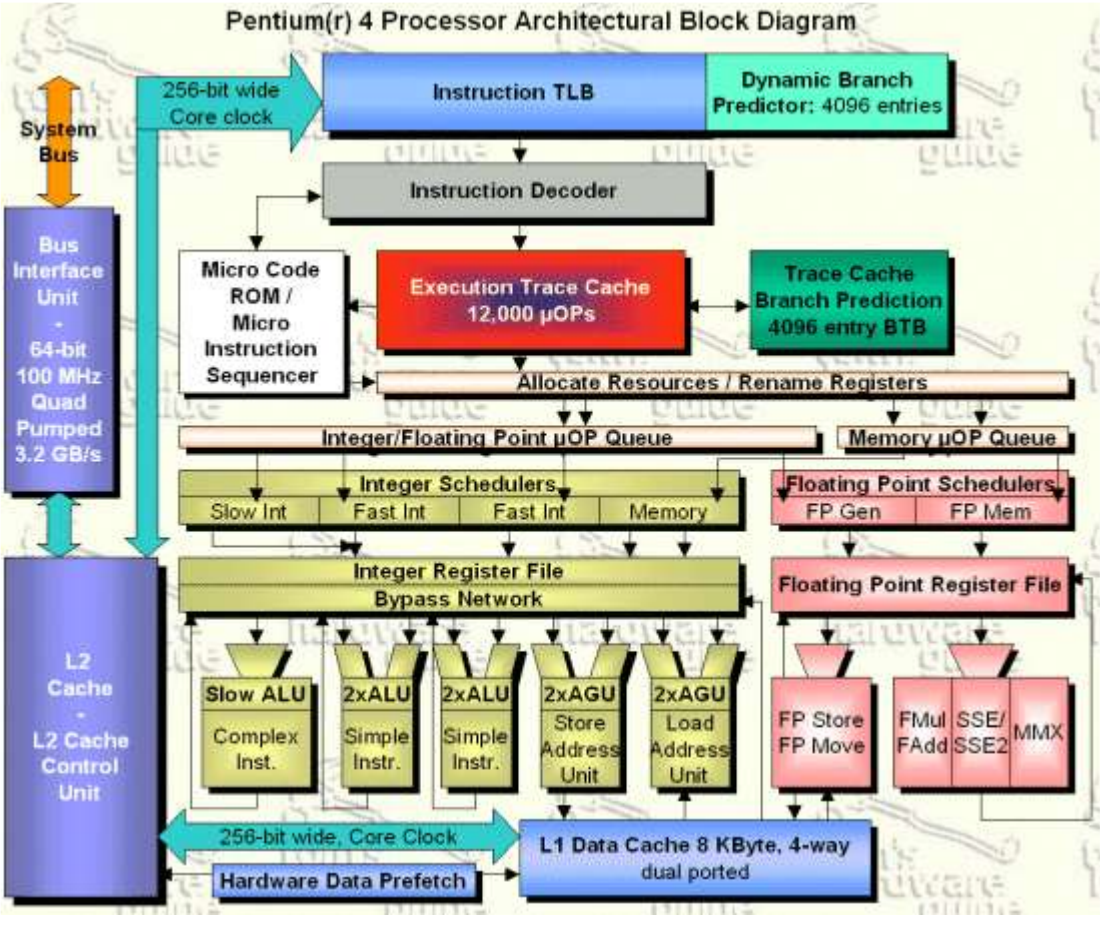


Fig 3.38Pentium 4 block diagram

INORDER FRONT END

- **Consists of:**
  - **The Instruction TLB/Pre-fetcher**
  - **The Instruction Decoder**
  - **The Trace Cache**
  - **The Microcode ROM**
  - **The Front-End Branch Predictor (BTB)**
- **Performs the following functions:**
  - **Pre-fetches instructions that are likely to be executed**
  - **Fetches required instructions that have not been pre-fetched**
  - **Decodes instructions into µops**
  - **Generates microcode for complex instructions and special purpose code**
  - **Delivers decoded instructions from the execution trace cache**
  - **Predicts branches (uses the past history of program execution to speculate where the program is going to execute next)**

INSTRUCTION TLB PREFETCHER

- **The Instruction TLB/Pre-fetcher translates the linear instruction pointer addresses**

given to it into physical addresses needed to access the L2 cache, and performs page-level protection checking

- **Intel NetBurst microarchitecture supports three pre-fetching mechanisms:**
  - **A hardware instruction fetcher that automatically pre-fetches instructions**
  - **A hardware mechanism that automatically fetches data and instructions into the unified L2 cache**
  - **A mechanism fetches data only and includes two components:**
    - **A hardware mechanism to fetch the adjacent cache line within an 128-byte sector that contains the data needed due to a cache line miss**
    - **A software controlled mechanism that fetches data into the caches using the pre-fetch instructions**

INORDER FRONTEND INSTRUCTION DECODER

- **The instruction decoder receives instruction bytes from L2 cache 64-bits at a time and decodes them into μops**
- **Decoding rate is one instruction per clock cycle**
- **Some complex instructions need the help of the Microcode ROM**
- **The decoder operation is connected to the Trace Cache**

INORDER FRONTEND BRANCH PREDICTOR

- **Instruction pre-fetcher is guided by the branch prediction logic (branch history table and branch target buffer BTB)**
- **Branch prediction allows the processor to begin fetching and executing instructions long before the previous branch outcomes**
- **The front-end branch predictor has 4K branch target entries to capture most of the branch history information for the program**
- **If a branch is not found in the BTB, the branch prediction hardware statically predicts the outcome of the branch based on the direction of the branch displacement (forward or backward)**
- **Backward branches are assumed to be taken and forward branches are assumed to not be taken**

INORDER FRONTEND TRACE CACHE

- **The Trace Cache is L1 instruction cache of the Pentium 4 processor**
- **Stores decoded instructions (μops)**
- **Holds up to 12K μops**
- **Delivers up to 3 μops per clock cycle to the out-of-order execution logic**
- **Hit rate to an 8K to 16K byte conventional instruction cache**
- **Takes decoded μops from instruction decoder and assembles them into program-ordered sequences of μops called traces**
  - **Can be many trace lines in a single trace**
  - **μops are packed into groups of 6 μops per trace line**
  - **Traces consist of μops running sequentially down the predicted path of the program execution**

  - **Target of branch is included in the same trace cache line as the branch itself**
- **Has its own branch predictor that directs where instruction fetching needs to go next in the Trace Cache**
- **INORDER FRONT END MICROCODE ROM**
- **Microcode ROM is used for complex IA-32 instructions (string move, for fault and interrupt handling)**
- **Issues the μops needed to complete complex instruction**
- **The μops that come from the Trace Cache and the microcode ROM are buffered in a single in-order queue to smooth the flow of μops going to the out-of-order execution engine**

OUT OF ORDER EXECUTION ENGINE

- **Consists of:**
  - **Out-of-Order Execution Logic**
    - **Allocator Logic**
    - **Register Renaming Logic**
    - **Scheduling Logic**
  - **Retirement Logic**
- **Out-of-Order Execution Logic is where instructions are prepared for execution**
  - **Has several buffers to smooth and re-order the flow of the instructions**
  - **Instructions reordering allows to execute them as quickly as their input operands are ready**
  - **Executes as many ready instructions as possible each clock cycle, even if they are not in the original program order**
  - **Allows instructions in the program following delayed instructions to proceed around them as long as they do not depend on those delayed instructions**
  - **Allows the execution resources to be kept as busy as possible**

OUT OF ORDER ALLOCATION LOGIC

- **The Allocator Logic allocates many of the key machine buffers needed by each μop to execute**
- **Stalls if a needed resource is unavailable for one of the three μops coming to the allocator in clock cycle**
- **Assigns available resources to the requesting μops and allows these μops to flow down the pipeline to be executed**
- **Allocates a Reorder Buffer (ROB) entry, which tracks the completion status of one of the 126 μops that could be in flight simultaneously in the machine**
- **Allocates one of the 128 integer or floating-point register entries for the result data value of the μop, and possibly a load or store buffer used to track one of the 48 loads or 24 stores in the machine pipeline**
- **Allocates an entry in one of the two μop queues in front of the instruction schedulers**

OUT OF ORDER REGISTER RENAMING LOGIC

- **The Register Renaming Logic renames the logical IA-32 registers such as EAX (extended accumulator) onto the processor's 128-entry physical register file**

- **Advantages:**
  - **Allows the small, 8-entry, architecturally defined IA-32 register file to be dynamically expanded to use the 128 physical registers in the Pentium 4 processor**
  - **Removes false conflicts caused by multiple instructions creating their simultaneous, but unique versions of a register such as EAX**

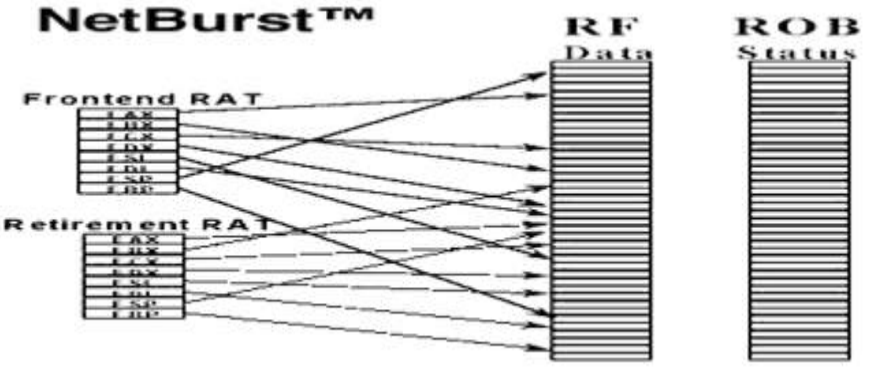


Fig 3.39 Register renaming logic

- **Pentium 4**
  - **Allocates the ROB entries and the result data Register File (RF) entries separately**
  - **ROB entries consist only of the status field that are allocated and de-allocated sequentially**
  - **Sequence number assigned to each μop indicates its relative age**
  - **Sequence number points to the μop's entry in the ROB array, which is similar to the P6 microarchitecture**
  - **Register File entry is allocated from a list of available registers in the 128-entry RF not sequentially like the ROB entries**
  - **No result data values are actually moved from one physical structure to another upon retirement**

OUT OF ORDER SCHEDULING LOGIC

- **The μop Scheduling Logic allow the instructions to be reordered to execute as soon as they are ready**
- **Two sets of structures:**
  - **μop queues**
  - **Actual μop schedulers**
- **μop queues:**
  - **For memory operations (loads and stores)**
  - **For non-memory operations**
- **Queues store the μops in first-in, first-out (FIFO) order with respect to the μops in its own queue**
- **Queue can be read out-of-order with respect to the other queue (this provides dynamic**

out-of-order scheduling window to be larger than just having the μop schedulers do all the reordering work)

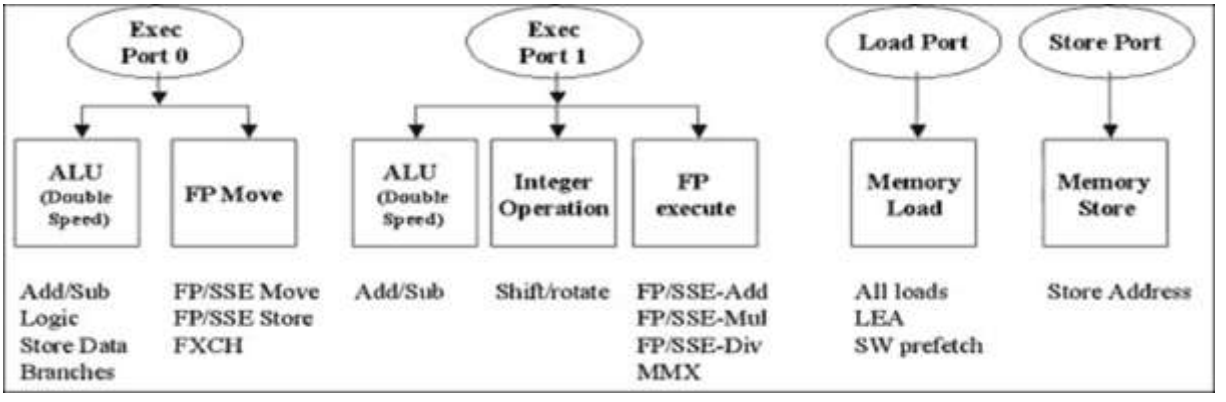


Fig 3.40 Scheduling Logic

- **Schedulers are tied to four dispatch ports**
- **Two execution unit dispatch ports labeled Port 0 and Port 1 (dispatch up to two operations each main processor clock cycle)**
  - **Port 0 dispatches either one floating-point move μop (a floating-point stack move, floating-point exchange or floating-point store data) or one ALU μop (arithmetic, logic or store data) in the first half of the cycle. In the second half of the cycle, dispatches one similar ALU μop**
  - **Port 1 dispatches either one floating-point execution μop or one integer μop (multiply, shift and rotate) or one ALU (arithmetic, logic or branch) μop in the first half of the cycle. In the second half of the cycle, dispatches one similar ALU μop**
- **Multiple schedulers share each of two dispatch ports**
- **ALU schedulers can schedule on each half of the main clock cycle**
- **Other schedulers can only schedule once per main processor clock cycle**
- **Schedulers compete for access to dispatch ports**
- **Loads and stores have dedicated ports**
  - **Load port supports the dispatch of one load operation per cycle**
  - **Store port supports the dispatch of one store address operation per cycle**
- **Peak bandwidth is of 6 μops per cycle**

RETIREMENT LOGIC

- **The Retirement Logic reorders the instructions executed out-of-order back to the original program order**
  - **Receives the completion status of the executed instructions from the execution units**
  - **Processes the results so the proper architectural state is committed according to the program order**
  - **Ensures the exceptions occur only if the operation causing the exception is not-retired operation**
  - **Reports branch history information to the branch predictors at the front end**

INTEGER AND FLOATING POINT EXECUTION UNITS

- **Consists of:**
  - **Execution units**
  - **Level 1 (L1) data cache**
- **Execution units are where the instructions are executed**
  - **Units used to execute integer operations:**
    - **Low-latency integer ALU**
    - **Complex integer instruction unit**
    - **Load and store address generation units**
  - **Floating-point/SSE execution units**
    - **FP Adder**
    - **FP Multiplier**
    - **FP Divide**
    - **Shuffle/Unpack**
- **L1 data cache is used for most load and store operations**

LOW LATENCY INTEGER ALU

- **ALU operations can be performed at twice the clock rate**
  - **Improves the performance for most integer applications**
- **ALU-bypass loop**
  - **A key closed loop in the processor pipeline**
- **High-speed ALU core is kept as small as possible**
  - **Minimizes: Metal length and Loading**
- **Only the essential hardware necessary to perform the frequent ALU operations is included in this high-speed ALU execution loop**
- **Functions that are not used very frequently are put elsewhere**
  - **Multiplier, Shifts, Flag logic, and Branch processing**

LOW LATENCY INTEGER ALU STAGGERED ADD

- **ALU operations are performed in a sequence of three fast clock cycles (the fast clock runs at 2x the main clock rate)**
  - **First fast clock cycle - The low order 16-bits are computed and are immediately available to feed the low 16-bits of a dependent operation the very next fast clock cycle**
  - **Second fast clock cycle - The high-order 16 bits are processed, using the carry out just generated by the low 16-bit operation**
  - **Third fast clock cycle - The ALU flags are processed**
- **Staggered add means that only a 16-bit adder and its input mux-es need to be completed in a fast clock cycle**

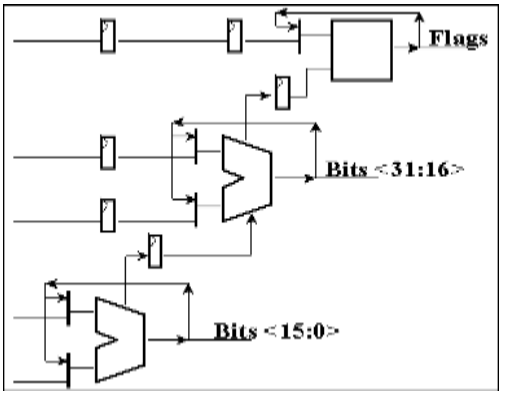


Fig 3.41 Low latency integer ALU

COMPLEX INTEGER OPERATION

- **Integer operations that are more complex go to separate hardware for completion**
- **Integer shift or rotate operations go to the complex integer dispatch port**
- **Shift operations have a latency of four clocks**
- **Integer multiply and divide operations have a latency of about 14 and 60 clocks, respectively.**

FLOATING POINT/SSE EXECUTION UNIT

- **The Floating-Point (FP) execution unit is where the floating-point, MMX, SSE, and SSE2 instructions are executed**
- **This execution unit has two 128-bit execution ports that can each begin a new operation every clock cycle**
  - **One execution port is for 128-bit general execution**
  - **Another is for 128-bit register-to-register moves and memory stores**
- **FP/SSE unit can complete a full 128-bit load each clock cycle**
- **FP adder can execute one Extended-Precision (EP) addition, one Double-Precision (DP) addition, or two Single-Precision (SP) additions every clock cycle**
- **Single precision-32 bits are used to represent floating-point number. It uses 8 bits for exponent**
- **Double precision-64 bits are used to represent floating-point number. It uses 11 bits for exponent**
- **128-bit SSE/SSE2 packed SP or DP add μops can be completed every two clock cycles**
- **FP multiplier can execute either**
- **One EP multiply every two clocks**
- **Or it can execute one DP multiply**
- **Or two SP multiplies every clock cycle**
- **128-bit SSE/SSE2 packed SP or DP multiply μop can be completed every two clock cycles**
- **Peak GFLOPS: (GIGAFLoating point OPerations per Second) One billion floating point operations per second)**
- **Single precision - 6 GFLOPS at 1.5 GHz**
- **Double precision - 3 GFLOPS at 1.5 GHz**

- **For integer SIMD operations there are three execution units that can run in parallel:**
- **SIMD integer ALU execution hardware can process 64 SIMD integer bits per clock cycle**
- **Shuffle/Unpack execution unit can also process 64 SIMD integer bits per clock cycle allowing it to do a full 128-bit shuffle/unpack μop operation each two clock cycles**
- **MMX/SSE2 SIMD integer multiply instructions use the FP multiply hardware to also do a 128-bit packed integer multiply μop every two clock cycles**
- **The FP divider executes all divide, square root, and remainder μops, and is based on a double-pumped SRT(square root) radix-2 algorithm, producing two bits of quotient (or square root) every clock cycle**
- **SSE-streaming SIMD extension**
- **PENTIUM 4 MEMORY SUBSYSTEM**
- **The Pentium 4 processor has a highly capable memory subsystem to enable the high-bandwidth stream-oriented applications such as 3D, video, and content creation**
- **This subsystem consists of:**
  - **Level 2 (L2) Unified Cache**
  - **400 MHz System Bus**
- **L2 cache stores instructions and data that cannot fit in the Trace Cache and L1 data cache**
- **System bus is used to access main memory when L2 cache has a cache miss, and to access the system I/O resources**
  - **System bus bandwidth is 3.2 GB per second**
  - **Uses a source-synchronous protocol that quad-pumps the 100 MHz bus to give 400 million data transfers per second**
  - **Has a split-transaction, deeply pipelined protocol to provide high memory bandwidths in a real system**
  - **Bus protocol has a 64-byte access length**

TRACE CACHE

- **Level 1 Execution Trace Cache is the primary or L1 instruction cache**
- **Most frequently executed instructions in a program come from the Trace Cache**
- **Only when there is a Trace Cache miss fetching and decoding instructions are performed from L2 cache**
- **Trace Cache has a capacity to hold up to 12K μops in the order of program execution**
- **Performance is increased by removing the decoder from the main execution loop**
- **Usage of the cache storage space is more efficient since instructions that are branched around are not stored**

DATA CACHE

- **Level 1 (L1) data cache is an 8KB cache that is used for both integer and floating-point/SSE loads and stores**
  - **Organized as a 4-way set-associative cache**
  - **Has 64 bytes per cache line**
  - **Write-through cache (writes to it are always copied into the L2 cache)**
  - **Can do one load and one store per clock cycle**

- **L1 data cache operates with a 2-clock load-use latency for integer loads and a 6-clock load-use latency for floating-point/SSE loads**
- **L1 cache uses new access algorithms to enable very low load-access latency (almost all accesses hit the first-level data cache and the data TLB)**

L2 CACHE

- **Level 2 (L2) cache is a 256KB cache that holds both instructions that miss the Trace Cache and data that miss the L1 data cache**
  - **Non-blocking, full speed**
  - **Organized as an 8-way set-associative cache**
  - **128 bytes per cache line**
  - **128-byte cache lines consist of two 64-byte sectors**
  - **Write-back cache that allocates new cache lines on load or store misses**
  - **256-bit data bus to the level 2 cache**
  - **Data clocked into and out of the cache every clock cycle**
- **A miss in the L2 cache typically initiates two 64-byte access requests to the system bus to fill both halves of the cache line**
- **New cache operation can begin every two processor clock cycles**
  - **For a peak bandwidth of 48Gbytes per second, when running at 1.5 GHz**
- **Hardware pre-fetcher**
  - **Monitors data access patterns and pre-fetches data automatically into the L2 cache**
  - **Remembers the history of cache misses to detect concurrent, independent streams of data that it tries to pre-fetch ahead of use in the program.**
  - **Tries to minimize pre-fetching unwanted data that can cause over utilization of the memory system and delay the real accesses the program needs**

L3 CACHE

- **Integrated 2-MB Level 3 (L3) Cache is coupled with the 800MHz system bus to provide a high bandwidth path to memory**
- **The efficient design of the integrated L3 cache provides a faster path to large data sets stored in cache on the processor**
- **Average memory latency is reduced and throughput is increased for larger workloads**
- **Available only on the Pentium 4 Extreme Edition**
- **Level 3 cache can preload a graphics frame buffer or a video frame before it is required by the processor, enabling higher throughput and faster frame rates when accessing memory and I/O devices**

BRANCH PREDICTION

- **2 Branch Prediction Units present on the Pentium 4**
  - **Front End Unit – 4KB Entries**
  - **Trace Cache – 512 Entries**
- **Allows the processor to begin execution of instructions before the actual outcome of the branch is known**
- **The Pentium 4 has an advanced branch predictor. It is comprised of three different**

components:

- **Static Predictor**
- **Branch Target Buffer**
- **Return Stack**

- **Branch delay penalty for a correctly predicted branch can be as few as zero clock cycles.**
- **However, the penalty can be as many as the pipeline depth.**
- **Also, the predictor allows a branch and its target to coexist in a signal trace cache line. Thus maximizing instruction delivery from the front end**

STATIC PREDICTOR

- **As soon as a branch is decoded, the direction of the branch is known.**
- **If there is no entry in the Branch History Table (BHT) then the Static Predictor makes a prediction based on the direction of the branch.**
- **The Static Predictor predicts two types of branches:**
  - **Backward – (negative displacement branches) always predicted as taken**
  - **Foreword – (positive displacement branches) always predicted not taken**

BRANCH TARGET BUFFER

- **In the Pentium 4 processor the Branch Target Buffer consists of both the Branch History Table as well as the Branch Target Buffer.**
  - **8 times larger than the BTB in the PIII**
  - **Intel claims this can eliminate 33% of the mispredictions found in the PIII.**
- **Once a branch history is available the processor can predict the branch outcome before the branch instruction is even decoded.**
- **The processor uses the BTB to predict the direction and target of branches based on an instruction's linear address.**
- **When a branch is retired the BTB is updated with the target address.**

RETURN STACK

- **Functionality:**
  - **Holds return addresses**
  - **Predicts return addresses for a series of procedure calls**
- **Increases benefit of unrolling loops containing function calls**
- **The need to put certain procedures inline (because of the return penalty portion of the procedure call overhead) is reduced.**

PIPELINE

- **The Pentium 4 has a 20 stage pipeline**
- **This deep pipeline increases**
  - **Performance of the processor**
  - **Frequency of the clock**
  - **Scalability of the processor**
- **Also, it provides**

- **High Clock Rates**
- **Frequency headroom to above 1GHz**

- **TC Nxt IP**
- **TC Fetch**
- **Drive**
- **Allocate**
- **Rename**
- **Que**
- **Schedule**
- **Dispatch**
- **Retire**
- **Execution**
- **Flags**
- **Branch Check**

TX NXT IP

- **"Trace Cache: Next Instruction Pointer"**
- **Held in the BTB (branch target buffer)**
- **And specifies the position of the next instruction to be processed**
- **Branch Prediction takes over**
  - **Previously executed branch: BHT has entry**
  - **Not previously executed or Trace Cache has invalidated the location: Calculate Branch Address and send to L2 cache and/or system bus**

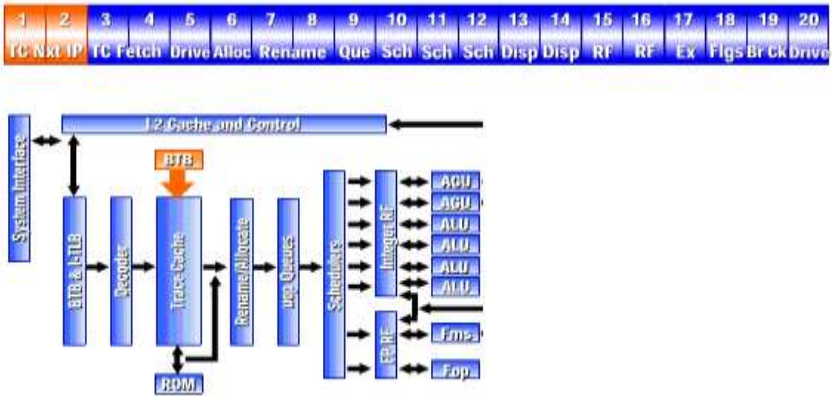


Fig3.42 Pipeline stages

TRACE CACHE FETCH

- **Reading µops (from Execution TC) requires two clock cycles**
- **The TC holds up to 12K µops and can output up to three µops per cycle to the Rename/Allocator**
- **Storing µops in the TC removes:**
  - **Decode-costs on frequently used instructions**
  - **Extra latency to recover on a branch misprediction**

WIRE DRIVE

- **This stage of the pipeline occurs multiple times**
- **WD only requires one clock cycle**
- **During this stage, up to three µops are moved to the Rename/Allocator**
  - **One load**
  - **One store**
  - **One manipulate instruction**

ALLOCATE

- **This stage determines what resources are needed by the µops.**
- **Decoded µops go through a one-stage Register Allocation Table (RAT)**
- **IA-32 instruction register references are renamed during the RAT stage**

RENAMING REGISTERS

- **This stage renames logical registers to the physical register space**
- **In the MicroBurst Architecture there are 128 registers with unique names**
- **Basically, any references to original IA-32 general purpose registers are renamed to one of the internal physical registers.**
- **Also, it removes false register name dependencies between instructions allowing the processor to execute more instructions in parallel.**
- **Parallel execution helps keep all resources busy**

QUE

- **Also known as the µops "pool".**
- **µops are put in the queue before they are sent to the proper execution unit.**
- **Provides record keeping of order commitment/retirement to ensure that µops are retired correctly.**
- **The queue combined with the schedulers provides a function similar to that of a reservation station.**

SCHEDULERS

- **Ensures µops execute in the correct sequence**
- **Disperses µops in the queue (or pool) to the proper execution units.**
- **The scheduler looks to the pool for requests, and checks the functional units to see if the necessary resources are available.**

DISPATCH

- **This stage takes two clock cycles to send each µops to the proper execution unit.**
- **Logical functions are allowed to execute in parallel, which takes half the time, and thus executes them out of order.**
- **The dispatcher can also store results back into the queue (pool) when it executes out of order.**

RETIREMENT

- **During this stage results are written back to memory or actual IA-32 registers that were referred to before renaming took place.**
- **This unit retires all instructions in their original order, taking all branches into account.**

- **Three µops may be retired in one clock cycle**
- **The processor detects and recovers from mispredictions in this stage.**
- **Also, a reorder buffer (ROB) is used:**
  - **Updates the architectural state**
  - **Manages the ordering of exceptions**

EXECUTION

- **µops will be executed on the proper execution engine by the processor**
- **The number of execution engines limits the amount of execution that can be performed.**
- **Integer and floating point unites comprise this limiting factor**

FLAGS, BRANCH CHECK &WIREDRIVE

- **Flags**
  - **One clock cycle is required to set or reset any flags that might have been affected.**
- **Branch Check**
  - **Brach operations compares the result of the branch to the prediction,The P4 uses a BHT and a BTB**
- **Wire Drive**
  - **One clock cycle moves the result of the branch check into the BTB and updates the target address after the branch has been retired.**

HYPER PIPELINE

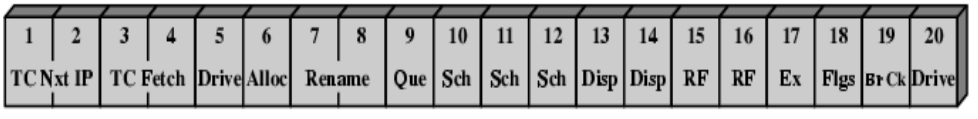


Fig 3.43 Hyper pipelining

TC Nxt IP-Trace Cache: Next Instruction Pointer"-Held in the BTB (branch target buffer) and specifies the position of the next instruction to be processed

TC Fetch-Trace Cache (TC) Fetch-Reading µops (from Execution TC) requires two clock cycles

Drive-Wire Drive- this stages of pipeline occurs multiple times, 3 microoperations are moved to rename/allocator, one load, one store and one manipulate instruction

Allocate- This stage determines what resources are needed by the µops

Rename-This stage renames logical registers to the physical register space

Que-Also known as the µops "pool"- µops are put in the queue before they are sent to the proper execution unit

Schedule-Ensures µops execute in the correct sequence, Disperses µops in the queue (or pool) to the proper execution units

Dispatch- This stage takes two clock cycles to send each µops to the proper execution unit

Retire- During this stage results are written back to memory or actual IA-32 registers (Intel Architecture 32 bit)

Execution- µops will be executed on the proper execution engine by the processor

Flags-One clock cycle is required to set or reset any flags that might have been affected.

Branch Check-Branch operations compares the result of the branch to the prediction. The P4 uses a BHT and a BTB

Wire drive- One clock cycle moves the result of the branch check into the BTB and updates the target address after the branch has been retired

HYPER THREADING

Enables software to take advantage of both task-level and thread-level parallelism by providing multiple logical processors within a physical processor package.

Two logical units in one processor

Each one contains a full set of architectural registers

But, they both share one physical processor's resources

Appears to software (including operating systems and application code) as having two processors.

Provides a boost in throughput in actual multiprocessor machines.

Each of the two logical processors can execute one software thread.

Allows for two threads (max) to be executed simultaneously on one physical processor

SMP-symmetric multiprocessing

thread of execution is the smallest sequence of programmed instructions that can be managed independently , a task is a unit of execution or a unit of work

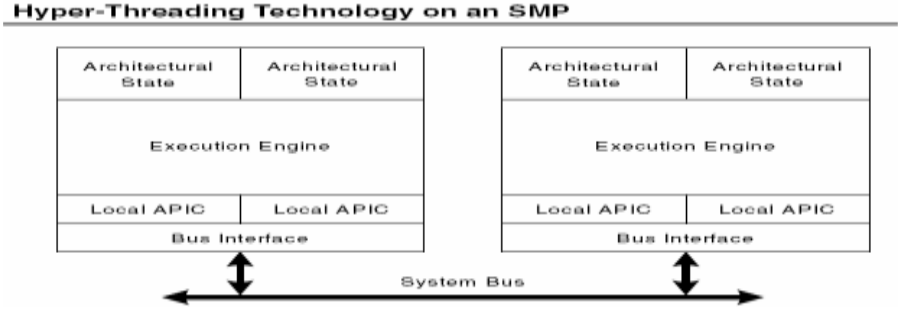


Fig 3.44 Hyper threading

Replicated Resources

Architectural State is replicated for each logical processor. The state registers control program behavior as well as store data. General Purpose Registers (8),Control Registers, Machine State Registers, Debug Registers

Instruction pointers and register renaming tables are replicated to track execution and state changes.

Return Stack is replicated to improve branch prediction of return instructions

Finally, Buffers were replicated to reduce complexity

Partitioned Resources

Buffers are shared by limiting the use of each logical processor to half the buffer entries.

By partitioning these buffers the physical processor achieves: Operational fairness

Allows operations from one logical processor to continue on while the other logical processor may be stalled.

Example: cache miss – partitioning prevents the stalled logical processor from blocking forward progress. Generally speaking, the partitioned buffers are located between the major pipeline stages

Shared Resources

Most resources in a physical processor are fully shared:

Caches

All execution units

Some shared resources (like the DTLB) include an identification bit to determine which logical processor the information belongs too.

INSTRUCTION SET

- **Pentium 4 instructions divided into the following groups:**
  - **General-purpose instructions**
  - **x87 Floating Point Unit (FPU) instructions**
  - **x87 FPU and SIMD state management instructions**
  - **Intel (MMX) technology instructions**
  - **Streaming SIMD Extensions (SSE) extensions instructions**
  - **SSE2 extensions instructions**
  - **SSE3 extensions instructions**
  - **System instructions**
- **MMX is a Pentium microprocessor that is designed to run faster when playing multimedia applications**
- **The MMX technology consists of three improvements over the non-MMX Pentium microprocessor:**
  - **57 new microprocessor instructions have been added to handle video, audio, and graphical data more efficiently**
  - **Single Instruction Multiple Data (SIMD), makes it possible for one instruction to perform the same operation on multiple data items**
  - **The memory cache on the microprocessor has increased to 32KB, meaning fewer accesses to memory that is off chip**
- **MMX instructions operate on packet byte, word, double-word, or quad-word integer operands contained in either memory, MMX registers, and/or in general-purpose registers**
- **MMX instructions are divided into the following subgroups:**
  - **Data transfer instructions**
  - **Conversion instructions**
  - **Packed arithmetic instructions**
  - **Comparison instructions**
  - **Logical instructions**
  - **Shift and rotate instructions**
  - **State management instructions**

- **Example: Logical AND – PAND**
  - **Source can be any of these:**
    MMX technology register or 64-bit memory location
    MMX technology register or 128-bit memory location
  - **Destination must be:**
    MMX technology register or XMM register
- **SSE2 add the following:**
  - **128-bit data type with two packed double-precision floating-point operands**
  - **128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-double-word, or 2-quad-word integers**
  - **Support for SIMD arithmetic on 64-bit integer operands**
  - **Instructions for converting between new existing data types**
  - **Extended support for data shuffling**
  - **Extended support for data cache ability and memory ordering operations**
- **SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption**
- **SSE3 instructions are divided into following groups:**
  - **Data movement**
  - **Arithmetic**
  - **Comparison**
  - **Conversion**
  - **Logical**

Shuffle operations

- **SSE3 add the following:**
  - **SIMD floating-point instructions for asymmetric and horizontal computation**
  - **A special-purpose 128-bit load instruction to avoid cache line splits**
  - **An x87 floating-point unit instruction to convert to integer independent of the floating-point control word**
  - **Instructions to support thread synchronization**
- **SSE3 instructions are useful for scientific, video and multi-threaded applications**
- **SSE3 instructions can be grouped into the following categories:**
  - **One x87FPU instruction used in integer conversion**
  - **One SIMD integer instruction that addresses unaligned data loads**
  - **Two SIMD floating-point packed ADD/SUB instructions**
  - **Four SIMD floating-point horizontal ADD/SUB instructions**
  - **Three SIMD floating-point LOAD/MOVE/DUPLICATE instructions**
  - **Two thread synchronization instructions**
- **NEW P4 INSTRUCTIONS**

- **WBINVD – Write Back and Invalidate Cache**
  - **System Instruction**
  - **Writes back all modified cache lines to main memory and invalidates (flushes) the**

internal caches.

- **CLFLUSH – Flush Cache Line**
  - **SSE2 Instruction**
  - **Flushes and invalidates a memory operand and its associated cache line from all levels of the processor's cache hierarchy**
- **LDDQU – Load Unaligned Integer 128-bits**
  - **SSE3 Instruction**
  - **Special 128-bit unaligned load designed to avoid cache line splits**

COMPARISON OF PENTIUM, PENTIUM PRO AND PENTIUM 4

SL.NO	DESCRIPTIION	PENTIUM	PENTIUM PRO	PENTIUM 4
1.	DATA BUS	64 BIT	64 BIT	64 BIT
2.	ADDRESS BUS	32 BIT	36-bit	32 BIT
3.	PHYSICAL ADDRESS	4GB	64GB	4GB
4.	SPEED	50MHZ-66MHZ	60 MHz, 66 MHz	400MHZ
7.	PINS	296	387	478
9.	MEMORY MANAGEMENT	PAGING AND SEGMENTATION	PAGING AND SEGMENTATION	PAGING AND SEGMENTATION
10.	CACHE	L1 CODE-8KB L1 DATA-8KB L2 CACHE-NIL	16 KB -L1 256KB or 512KB -L2 CACHE	8KB-L1 L1-12KB-EXECUTION TRACE CACHE L2- 256K for data 2-MB L3
12.	FLAGS	**Status Flags**-S,Z,C,AC,OV,P **Control Flags**-AC (Alignment check), ID (Identification flag), RF (Resume flag), IOPL (I/O privilege level), DF (Direction flag), IF (Interrupt enable flag), TF (Trap flag) **System Flags**-NT (Nested task flag), VM (Virtual 8086 mode), VIP (Virtual interrupt pending), VIF (Virtual interrupt flag)	Status Flags, Control Flags, System Flags	Status Flags, Control Flags, System Flags
13.	SEGMENT REGISTERS	CS,DS,ES,SS,FS,GS	CS,DS,ES,SS,FS,GS	CS,DS,ES,SS,FS,GS
14.	GENERAL PURPOSE REGISTERS	EAX,EBX,ECX,EDX, EBP,ESP,ESI,EDI	EAX,EBX,ECX,EDX, EBP,ESP,ESI,EDI	EAX,EBX,ECX,EDX, EBP,ESP,ESI,EDI
16.	YEAR	1993	1995	2000
17.	ALU	2 INTEGER ALU 1 FP ALU	1 DEU(DISPATCH AND EXECUTION UNIT)	2 FAST ALU 1 SLOW ALU 1FP ALU
18.	PIPELINING STAGES	INTEGER-5 FP-8	14 STAGES SUPER PIPELINING	20 STAGES PIPELINING
19.	PAGE SIZE	4MB	2MB	4MB
20.	ARCHITECTURE	SUPER SCALAR AND SUPER-PIPELINED CISC	SUPER PIPELINING	NETBURST ARCH SUPER PIPELINING HYPER THREADING
21.	CLOCK RATE	65 MHz - 250 MHz	150 MHz - 200 MHz	1.3 GHz - 3.8 GHz
22.	DATA TYPES	BYTE, WORD, DOUBLE WORD, QUAD WORD	BYTE, WORD, DOUBLE WORD, QUAD WORD	BYTE, WORD, DOUBLE WORD, QUAD WORD
23	BRANCH PREDICTION LOGIC	YES	YES	YES
24	ERROR CORRECTION	YES-PARITY CHECK	YES-ERROR CORRECTING CODE	NO

TEXT / REFERENCE BOOKS

1. **Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing", 2nd Edition,Pearson Education, Inc., 2004**
2. **Barry B.Brey , "The Intel Microprocessors 8086/8088, 8086, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, Architecture, Programming and interfacing", Prentice Hall of India Private Limited, New Delhi, 2003**
3. **Alan Clements, "The Principles of computer Hardware", Oxford University Press, 3rd Edition, 2003**
4. **John Paul Shen, Mikko H.Lipasti, "Modern Processor Design", Tata McGraw Hill, 2006**

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMMUNICATION ENGINEERING

UNIT - 4
ADVANCED MICROPROCESSORS – SEC1601

UNIT 4-RISC PROCESSORS I

- **PowerPC-620 - Instruction fetching - Branch Prediction - Fetching - Speculation, Instruction dispatching - dispatch stalls - Instruction Execution - Issue stalls- Execution Parallelism - Instruction completion –**
- **Basics of P6 micro architecture - Pipelining - Memory subsystem**

POWERPC 620 MICROPROCESSOR OVERVIEW

- **PowerPC (*Performance Optimization With Enhanced RISC – Performance Computing*) is a RISC architecture created by (AIM) Apple–IBM–Motorola alliance in 1991.**
- **The original idea for the PowerPC architecture came from IBM's *Power architecture (introduced in the* RISC/6000) and retains a high level of compatibility**
- **The intention was to build a high-performance, superscalar low-cost processor.**
- **To facilitate parallel instruction execution and to scale well with advancing technology**
- **The PowerPC alliance has released and announced a number of chips**
    - **The fourth chip was the 64-bit 620**

The 620 was the first 64-bit superscalar processor.

The features are

- **Out-of-order execution,**
- **aggressive branch prediction**
- **distributed multi entry reservation stations**
- **dynamic renaming for all register files**
- **six pipelined execution units**

a completion buffer to ensure precise exceptions

The 620 is an implementation of the PowerPC™ family of reduced instruction set computer (RISC) microprocessors. The 620 implements the PowerPC architecture as it is specified for 64-bit addressing, which provides 64-bit effective addresses, integer data types of 8, 16, 32, and 64 bits, and floating-point data types of 32 and 64 bits (single-precision and double-precision). The 620 is software compatible with the 32- bit versions of the PowerPC microprocessor family.

The 620 is a superscalar processor capable of issuing four instructions simultaneously. As many as four instructions can finish execution in parallel. The 620 has six execution units that can operate in parallel:

- **Floating-point unit (FPU)**
- **Branch processing unit (BPU)**
- **Load/store unit (LSU)**
- **Three integer units (IUs):**
    - **Two single-cycle integer units (SCIUs)**
    - **One multiple-cycle integer unit (MCIU)**

This parallel design, combined with the PowerPC architecture's specification of uniform instructions that allows for rapid execution times, yields high efficiency and throughput. The 620's rename buffers, reservation stations, dynamic branch prediction, and completion unit increase instruction throughput, guarantee in-order completion, and ensure a precise exception model. (Note that the PowerPC architecture specification refers to all exceptions as interrupts.)

The 620 has separate memory management units (MMUs) and separate 32-Kbyte on-chip caches for instructions and data. The 620 implements a 128-entry, two-way set-associative translation lookaside buffer (TLB) for instructions and data, and provides support for demand-paged virtual memory address translation and variable-sized block translation. The TLB and the cache use least-recently used (LRU) replacement algorithms.

The 620 has a 40-bit address bus, and can be configured with either a 64- or 128-bit data bus. The 620 interface protocol allows multiple masters to compete for system resources through a central external arbiter. Additionally, on-chip snooping logic maintains data cache coherency for multiprocessor applications. The 620 supports single-beat and burst data transfers for memory accesses and memory- mapped I/O accesses.

The 620 processor core uses an advanced, 2.5-V CMOS process technology, and is compatible with 3.3-V CMOS devices.

PowerPC 620 Microprocessor Features

This section summarizes features of the 620's implementation of the PowerPC architecture. Major features of the 620 are as follows:

- **High-performance, superscalar microprocessor**
  - **As many as four instructions can be issued per clock**
  - **As many as six instructions can start executing per clock (including three integer instructions)**
- **Single clock cycle execution for most instructions**

Six independent execution units and two register files

- **BPU featuring dynamic branch prediction**
  - **Speculative execution through four branches**
  - **256-entry fully-associative branch target address cache (BTAC)**
  - **2048-entry branch history table (BHT) with two bits per entry indicating four levels of prediction—not-taken, strongly not-taken, taken, strongly taken**
- **Two single-cycle IUs (SCIUs) and one multiple-cycle IU (MCIU)**
  - **Instructions that execute in the SCIU take one cycle to execute; most instructions that execute in the MCIU take multiple cycles to execute.**
  - **Each SCIU has a two-entry reservation station to minimize stalls.**
  - **The MCIU has a two-entry reservation station and provides early exit (three cycles) for 16 x 32-bit and overflow operations**
  - **Thirty-two GPRs for integer operands**
  - **Eight rename buffers for GPRs**
- **Three-stage floating-point unit (FPU)**
  - **Fully IEEE 754-1985 compliant FPU for both single- and double-precision operations**
  - **Supports non-IEEE mode for time-critical operations**
  - **Fully pipelined, single-pass double-precision design**
  - **Hardware support for denormalized numbers**

  - **Two-entry reservation station to minimize stalls**
  - **Thirty-two 64-bit FPRs for single- or double-precision operands**
  - **Eight rename buffers for FPRs**
- **Load/store unit (LSU)**
  - **Three-entry reservation station to minimize stalls**
  - **Single-cycle, pipelined cache access**
  - **Dedicated adder that performs EA calculations**
  - **Performs alignment and precision conversion for floating-point data**
  - **Performs alignment and sign extension for integer data**
  - **Five-entry pending load queue that provides load/store address collision detection**
  - **Five-entry finished store queue**
  - **Six-entry completed store queue**
  - **Supports both big- and little-endian modes**

- **Rename buffers**
  - **Eight GPR rename buffers**
  - **Eight FPR rename buffers**
  - **Sixteen condition register (CR) rename buffers**
- **Completion unit**
  - **Retires an instruction from the 16-entry reorder buffer when all instructions ahead of it have been completed and the instruction has finished execution**
  - **Guarantees sequential programming model (precise exception model)**
  - **Monitors all dispatched instructions and retires them in order**
  - **Tracks unresolved branches and removes speculatively executed, dispatched, and fetched instructions if branch is mispredicted**
  - **Retires as many as four instructions per clock**
- **Separate on-chip instruction and data caches (Harvard architecture)**
  - **32-Kbyte, eight-way set-associative instruction and data caches; data cache is 2-way interleaved.**
  - **LRU replacement algorithm**
  - **64-byte (sixteen word) cache block size**
  - **Physically indexed; physical tags**
  - **Cache write-back or write-through operation programmable on a per page or per block basis**
  - **Instruction cache can provide four instructions per clock; data cache can provide two words per clock.**
  - **Caches can be disabled in software**
  - **Parity checking performed on both caches**
  - **Data cache coherency (MESI) maintained in hardware**

  - **Interprocessor broadcast of cache control instructions**
  - **Instruction cache coherency maintained in software**
- **On-chip L2 cache interface**
  - **L2 cache is a unified instruction and data secondary cache with ECC.**
  - **L2 cache is direct-mapped, physically-indexed, and physically-tagged.**
  - **L2 data cache is inclusive of L1; L2 instruction cache is not inclusive of L1.**
  - **L2 cache capacity is configurable from 1 Mbyte to 128 Mbyte.**
  - **Independent user-configurable PLL provides L2 interface clock.**
  - **L2 cache interface supports single-, double-, triple-, and quad-register synchronous SRAMs.**
  - **L2 cache interface supports CMOS and HSTL SRAMs.**
  - **Supports direct connection of two SRAM banks**
  - **Supports direct connection of coprocessor**
- **Separate memory management units (MMUs) for instructions and data**
  - **Address translation facilities for 4-Kbyte page size, variable block size, and 256-Mbyte segment size**
  - **Independent 64-entry fully-associative effective-to-real address translation (ERAT) cache with invalid-first replacement algorithm for instructions and data**
  - **Unified instruction and data translation lookaside buffer (TLB)**
  - **TLB is 128-entry and two-way set-associative**
  - **20-entry CAM segment lookaside buffer (SLB) with FIFO replacement algorithm**
  - **Sixteen segment registers that provide support for 32-bit memory management**
  - **SLB, TLB, and ERAT cache miss handling performed by 620 hardware**
  - **Hardware update of page frame table referenced and changed bits**
  - **Hardware broadcast of TLB and control instructions**
  - **Separate IBATs and DBATs (four each) also defined as SPRs**
  - **64-bit effective addressing**
  - **80-bit virtual addressing**
  - **40-bit physical memory address for up to one terabyte**
- **Bus interface**
  - **Selectable processor-to-bus clock frequency ratios (2:1, 3:1, and 4:1)**
  - **A 64- and 128-bit split-transaction external data bus with burst transfers**
  - **Explicit address and data bus tagging**
  - **Pended (split) read protocol**
  - **Pipelined snoop response, fixed boot-time latency**
  - **620 bus is crossbar compatible**
  - **Additional signals and signal redefinition for direct-store operations**
- **Multiprocessing support**

- **Hardware-enforced, four-state cache coherency protocol (MESI) for data cache. Bits are provided in the instruction cache to indicate only whether a cache block is valid or invalid.**
- **Data cache coherence for L1 and L2, and external L3 cache is fully supported by 620 hardware.**
- **Snoop operations take priority over processor access to L1 and L2 cache.**
- **Instruction cache coherence is software controlled.**
- **Load/store with reservation instruction pair is provided for atomic memory references, semaphores, and other multiprocessor operations.**

- **Power requirements**
  - **Operating voltage is 2.5 V for the processor core, and 3.3 V for I/O drivers.**
- **Performance monitor can be used to help in debugging system designs and improving software efficiency, especially in multiprocessor systems.**
- **In-system testability and debugging features are provided through JTAG boundary-scan capability.**

Block Diagram

Figure provides a block diagram showing features of the 620. Note that this is a conceptual block diagram intended to show the basic features rather than an attempt to show how these features are physically implemented on the chip.

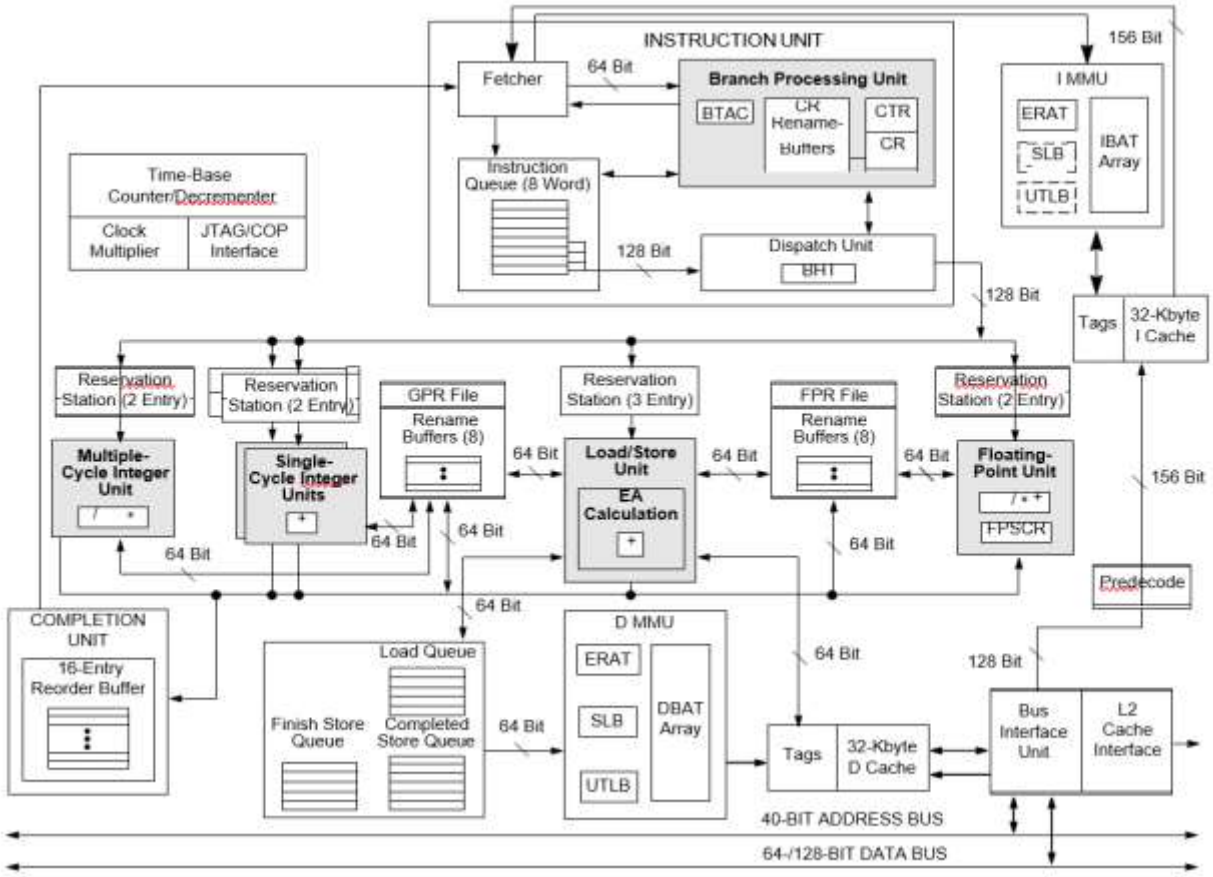


Fig 4.1 Block diagram power pc 620

PowerPC 620 Microprocessor Hardware Implementation

This section provides an overview of the 620's hardware implementation, including descriptions of the functional units, shown in Figure 2, the cache implementation, MMU, and the system interface.

Note that Figure 2 provides a more detailed block diagram than that presented in Figure 1—showing the additional data paths that contribute to the improved efficiency in instruction execution and more clearly indicating the relationships between execution units and their associated register files.

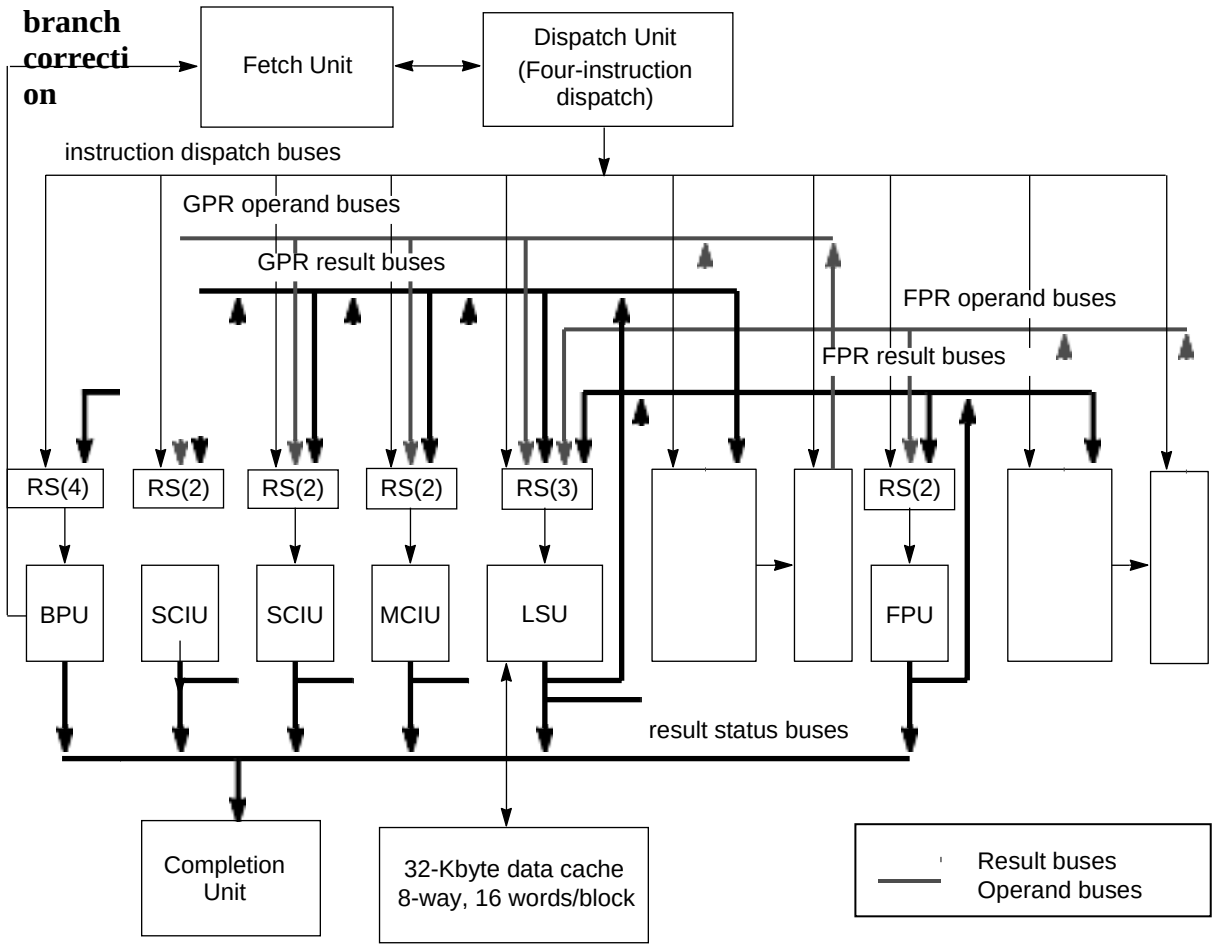


Figure 4.2. Block Diagram—Internal Data Paths

Instruction Flow

Several units on the 620 ensure the proper flow of instructions and operands and guarantee the correct update of the architectural machine state. These units include the following:

- **Predecode unit—Provides logic to decode instructions and determine what resources are required for execution.**
- **Fetch unit—Using the next sequential address or the address supplied by the BPU when a branch is predicted or resolved, the fetch unit supplies instructions to the eight-word instruction buffer.**

- **Dispatch unit—The dispatch unit dispatches instructions to the appropriate execution unit. During dispatch, operands are provided to the execution unit (or reservation station) from the register files, rename buffers, and result buses.**
- **Branch processing unit (BPU)—In addition to providing the fetcher with predicted target instructions when a branch is predicted (and a mispredict-recovery address if a branch is incorrectly predicted), the BPU executes all condition register logical and flow control instructions.**
- **Completion unit—The completion unit retires executed instructions in program order and controls the updating of the architectural machine state.**

Predecode Unit

The instruction predecode unit provides the logic to decode the instructions and categorize the resources that will be used, source operands, destination registers, execution registers, and other resources required for execution. The instruction stream is predecoded on its way from the bus interface unit to the instruction cache.

Fetch Unit

The fetch unit provides instructions to the four-entry instruction queue by accessing the on-chip instruction cache. Typically, the fetch unit continues fetching sequentially as many as four instructions at a time.

The address of the next instruction to be fetched is determined by several conditions, which are prioritized as follows:

1. **Detection of an exception. Instruction fetching begins at the exception vector.**
2. **The BPU recovers from an incorrect prediction when a branch instruction is in the execute stage. Undispatched instructions are flushed and fetching begins at the correct target address.**
3. **The BPU recovers from an incorrect prediction when a branch instruction is in the dispatch stage. Undispatched instructions are flushed and fetching begins at the correct target address.**
4. **A fetch address is found in the BTAC. As a cache block is fetched, the branch target address cache (BTAC) and the branch history table (BHT) are searched with the fetch address. If it is found in the BTAC, the target address from the BTAC is the first candidate for being the next fetch address.**
5. **If none of the previous conditions exist, the instruction is fetched from the next sequential address.**

Dispatch Unit

The dispatch unit provides the logic for dispatching the predecoded instructions to the appropriate execution unit. For many branch instructions, these decoded instructions along with the bits in the BHT, are used during the decode stage for branch correction. The dispatch logic also resolves unconditional branch instructions and predicts conditional branch instructions using the branch decode logic, BHT, and values in the count register (CTR).

The 2048-entry BHT provides two bits per entry, indicating four levels of dynamic

prediction—strongly not-taken, not-taken, taken, and strongly taken. The history of a branch's direction is maintained in these two bits. Each time a branch is taken, the value is incremented (with a maximum value of three meaning strongly-taken); when it is not taken, the bit value is decremented (with a minimum value of zero meaning strongly not-taken). If the current value predicts taken and the next branch is taken again, the BHT entry then predicts strongly taken. If the next branch is not taken, the BHT then predicts taken.

The dispatch logic also allocates each instruction to the appropriate execution unit. A reorder buffer entry in the completion unit is allocated for each instruction, and dependency checking is done between the instructions in the dispatch queue. The rename buffers are searched for the operands as the operands are fetched from the register file. Operands that are written by other instructions ahead of this one in the dispatch queue are given the tag of that instruction's rename buffer; otherwise, the rename buffer or register file supplies either the operand or a tag. As instructions are dispatched, the fetch unit is notified that the dispatch queue can be updated with more instructions.

Branch Processing Unit (BPU)

The BPU is used for branch instructions and condition register logical operations. All branches, including unconditional branches, are placed in reservation stations until conditions are resolved and they can be executed. At that point, branch instructions are executed in order—the completion unit is notified whether the prediction was correct.

The BPU also executes condition register logical instructions, which flow through the reservation station like the branch instructions.

Completion Unit

The completion unit retires executed instructions from the reorder buffer in the completion unit and updates register files and control registers. The completion unit recognizes exception conditions and discards any operations being performed on subsequent instructions in program order. The completion unit can quickly remove instructions from a mispredicted branch, and the dispatch unit begins dispatching from the correct path.

The instruction is retired from the reorder buffer when it has finished execution and all instructions ahead of it have been completed. The instruction's result is written into the appropriate register file and is removed from the rename buffers at or after completion. At completion, the 620 also updates any other resource affected by this instruction. Several instructions can complete simultaneously. Most exception conditions are recognized at completion time.

Rename Buffers

To avoid contention for a given register location, the 620 provides rename registers for storing instruction results before the completion unit commits them to the architected register. Eight rename registers are provided for the GPRs, eight for the FPRs, and sixteen for the condition register. GPRs, FPRs, and the condition register

"Registers and Programming Model,"

When the dispatch unit dispatches an instruction to its execution unit, it allocates a rename register for the results of that instruction. The dispatch unit also provides a tag to the

execution unit identifying the result that should be used as the operand. When the proper result is returned to the rename buffer, it is latched into the reservation station. When all operands are available in the reservation station, execution can begin.

The completion unit does not transfer instruction results from the rename registers to the registers until any speculative branch conditions preceding it in the completion queue are resolved and the instruction itself is retired from the completion queue without exceptions. If a speculatively executed branch is found to have been incorrectly predicted, the speculatively executed instructions following the branch are flushed from the completion queue and the results of those instructions are flushed from the rename registers.

Execution Units

The following sections describe the 620's arithmetic execution units— two single-cycle IUs, multiple-cycle IU, and FPU. When the reservation station sees the proper result being written back, it will grab it directly from one of the result buses. Once all operands are in the reservation station for an instruction, it is eligible to be executed. Reservation stations temporarily store dispatched instructions that cannot be executed until all of the source operands are valid.

Integer Units (IUs)

The two single-cycle IUs (SCIUs) and one multiple-cycle IU (MCIU) execute all integer instructions. These are shown in Figure 1 and Figure 2. The results generated by the IUs are put on the result buses that are connected to the appropriate reservation stations and rename buffers. Each IU has a two-entry reservation station to reduce stalls. The reservation station can receive instructions from the dispatch unit and operands from the GPRs, the rename buffers, or the result buses.

Each SCIU consists of three single-cycle subunits—a fast adder/comparator, a subunit for logical operations, and a subunit for performing rotates, shifts, and count-leading-zero operations. These subunits handle all one-cycle arithmetic instructions; only one subunit can execute an instruction at a time.

The MCIU consists of a 64-bit integer multiplier/divider. The MCIU executes mfspr and mtspr instructions, which are used to read and write special-purpose registers. The MCIU can execute an mtspr or mfspr instruction at the same time that it executes a multiply or divide instruction. These instructions are allowed to complete out of order.

Floating-Point Unit (FPU)

The FPU, shown in Figure 1 and Figure 2, is a single-pass, double-precision execution unit; that is, both single- and double-precision operations require only a single pass, with a latency of three cycles.

As the dispatch unit issues instructions to the FPU's two reservation stations, source operand data may be accessed from the FPRs, the floating-point rename buffers, or the result buses. Results in turn are written to the floating-point rename buffers and to the reservation stations and are made available to subsequent instructions. The three reservation stations provided by the FPU support out-of-order execution of floating- point instructions.

Load/Store Unit (LSU)

The LSU transfers data between the data cache and the result buses, which route data to other execution units. The LSU supports the address generation and handles any alignment for transfers to and from system memory. The LSU also supports cache control instructions and load/store multiple/string instructions.

The LSU includes a 64-bit adder dedicated for EA calculation. Data alignment logic manipulates data to support aligned or misaligned transfers with the data cache. The LSU's load and store queues are used to buffer instructions that have been executed and are waiting to be completed. The queues are used to monitor data dependencies generated by data forwarding and out-of-order instruction execution ensuring a sequential model.

The LSU allows load instructions to precede store instructions in the reservation stations. Data dependencies resulting from the out-of-order execution of loads before stores to addresses with the same low-order 12 bits in the effective address are resolved when the store instruction is completed. If an out-of-order load operation is found to have an address that matches a previous store, the instruction pipeline is flushed, and the load instruction will be refetched and re-executed.

The LSU does not allow the following operations to be speculatively performed on unresolved branches:

- **Store operations**
- **Loading of noncacheable data or cache miss operations**
- **Loading from direct-store segments**

Memory Management Units (MMUs)

The primary functions of the MMUs are to translate logical (effective) addresses to physical addresses for memory accesses, I/O accesses (most I/O accesses are assumed to be memory-mapped), and direct-store accesses, and to provide access protection on blocks and pages of memory.

The PowerPC MMUs and exception model support demand-paged virtual memory. Virtual memory management permits execution of programs larger than the size of physical memory; demand-paged implies that individual pages are loaded into physical memory from system memory only when they are first accessed by an executing program.

The hashed page table is a variable-sized data structure that defines the mapping between virtual page numbers and physical page numbers. The page table size is a power of 2, and its starting address is a multiple of its size.

Address translations are enabled by setting bits in the MSR—MSR[IR] enables instruction address translations and MSR[DR] enables data address translations.

The 620's MMUs support up to one heptabyte (2^{80}) of virtual memory and one terabyte (2^{40}) of physical memory. The MMUs support block address translations, direct-store segments, and page translation of memory segments. Referenced and changed status are maintained by the processor for each page to assist implementation of a demand-paged virtual memory system.

Separate but identical translation logic is implemented for data accesses and for instruction

accesses. The 620 implements a two-stage translation mechanism; the first stage consists of independent 64-entry content- addressable address translation caches for instructions and data, and the second stage consists of a shared 128-entry, two-way set-associative translation lookaside buffer (TLB). If a TLB miss occurs during the second-stage address translation, memory segment lookup is assisted by a 20-entry content-addressable segment lookaside buffer (SLB). Sixteen segment registers are provided by binary decode of 16 of the 20 SLBs for the execution of software compiled for 32-bit PowerPC microprocessors.

Cache Implementation

The PowerPC architecture does not define hardware aspects of cache implementations. For example, the 620 implements separate data and instruction caches (Harvard architecture), while other processors may use a unified cache, or no cache at all. The PowerPC architecture defines the unit of coherency as a cache block, which for the 620 is a 64-byte (sixteen-word) line.

PowerPC implementations can control the following memory access modes on a page or block basis:

- **Write-back/write-through mode**
- **Caching-inhibited mode**
- **Memory coherency**
- **Guarded memory (prevents access for out-of-order execution)**

Instruction Cache

The 620's 32-Kbyte, eight-way set-associative instruction cache is physically indexed. Within a single cycle, the instruction cache provides up to four instructions. Instruction cache coherency is not maintained by hardware.

The PowerPC architecture defines a special set of instructions for managing the instruction cache. The instruction cache can be invalidated entirely or on a cache-block basis. The instruction cache can be disabled and invalidated by setting the HID0[16] and HID0[20] bits, respectively.

Data Cache

The 620's data cache is a 32-Kbyte, eight-way set-associative cache. It is a physically-indexed, nonblocking, write-back cache with hardware support for reloading on cache misses. Within one cycle, the data cache provides double-word access to the LSU.

The data cache tags are dual-ported, so the process of snooping does not affect other transactions on the system interface. If a snoop hit occurs in the same cache set as a load or store access, the LSU is blocked internally for one cycle to allow the 16-word block of data to be copied to the write-back buffer.

The 620 data cache supports the four-state MESI (modified/exclusive/shared/invalid) protocol to ensure cache coherency.

These four states indicate the state of the cache block as follows:

- **Modified (M)—The cache block is modified with respect to system memory; that is, data for this address is valid only in the cache and not in system memory.**

- **Exclusive (E)—This cache block holds valid data that is identical to the data at this address in system memory. No other cache has this data.**
- **Shared (S)—This cache block holds valid data that is identical to this address in system memory and at least one other caching device.**
- **Invalid (I)—This cache block does not hold valid data.**

Like the instruction cache, the data cache can be invalidated all at once or on a per cache block basis. The data cache can be disabled and invalidated by setting the HID0[17] and HID0[21] bits, respectively.

Each cache line contains 16 contiguous words from memory that are loaded from a 16-word boundary (that is, bits A[58–63] of the logical addresses are zero); thus, a cache line never crosses a page boundary. Accesses that cross a page boundary can incur a performance penalty.

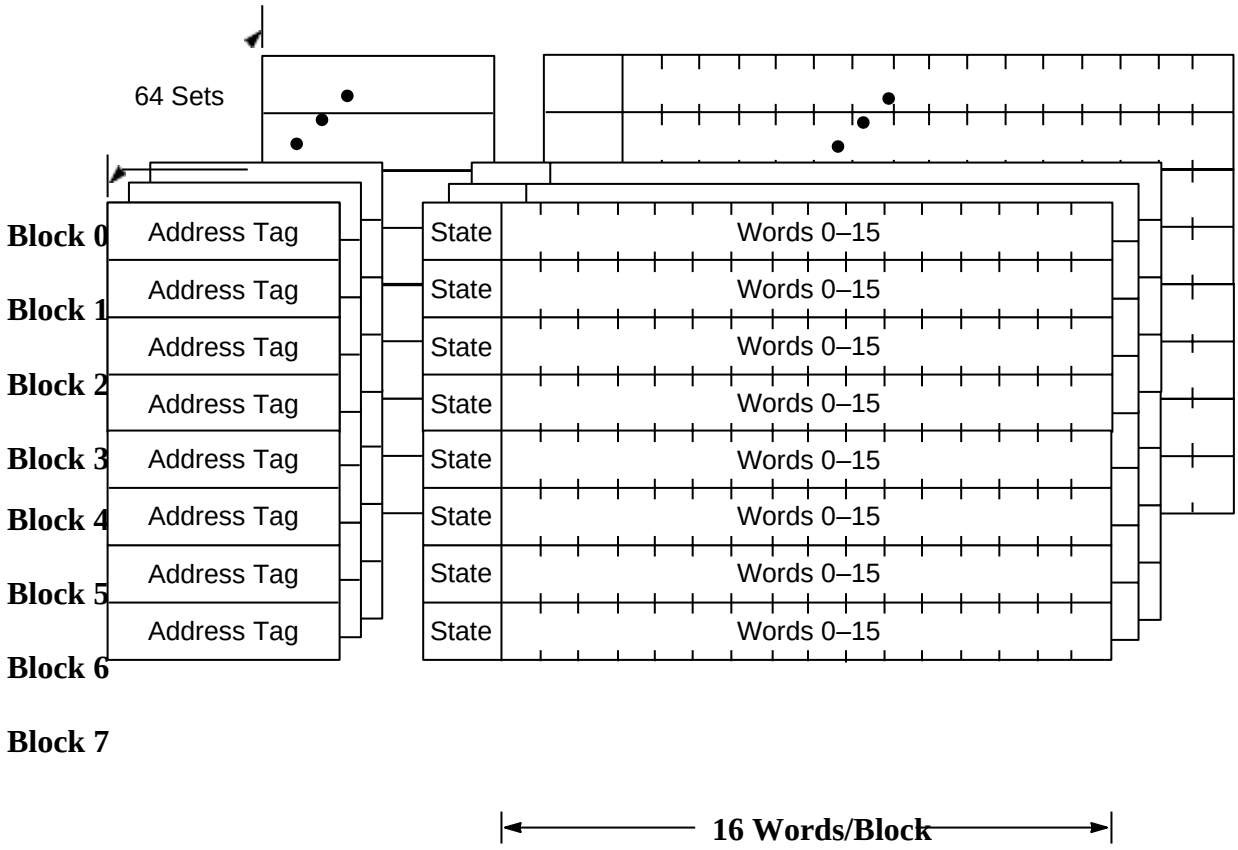


Figure 4.3. Cache Unit Organization

Level 2 (L2) Cache Interface

The 620 provides an integrated L2 cache controller that supports L2 configurations from 1 Mbyte to 128 Mbyte, using the same block size (64 bytes) as the internal L1 caches. The 620's L2 cache is a direct- mapped, error-correction-code (ECC) protected, unified instruction and data secondary cache that supports the use of single- and double-register synchronous static RAMs. The L2 cache interface supports a wide variety of static RAM access speeds by means of a boot-time configurable subsynchronous interface that is

configurable for either CMOS or HSTL logic levels. An external coprocessor can also be connected to the 620 through the L2 cache interface.

The L2 cache interface generates 9 bits of ECC for the 128 bits of data in a cache block, and 6 bits of ECC for the tag and coherency state of the block. The ECC allows the correction of single-bit errors, and the detection of double-bit errors. Uncorrectable errors detected by the L2 cache interface will generate a machine check exception. The ECC capability of the L2 cache interface can be configured in three modes— always-corrected mode, never-corrected mode, and automatic mode. In always-corrected mode, ECC is generated for write operations, and always corrected on read operations, resulting in constant L2 read access latency. In never-corrected mode, ECC generation, checking, and correction are disabled. In the automatic mode, ECC is generated during write operations, and read operations are corrected only when errors are detected, thereby increasing read latency only when correctable errors are detected.

System Interface/Bus Interface Unit (BIU)

The 620 provides a versatile bus interface that allows a wide variety of system design options. The interface includes a 144-bit data bus (128 bits of data and 16 bits of parity), a 43-bit address bus (40 bits of address and 3 bits of parity), and sufficient control signals to allow for a variety of system-level optimizations. The 620 uses one-beat, four-beat, and eight-beat data transactions (depending on whether the 620 is configured with a 64- or 128-bit data bus), although it is possible for other bus participants to perform longer data transfers. The 620 clocking structure supports processor-to-bus clock ratios of 2:1, 3:1, and 4:1

The system interface is specific for each PowerPC processor implementation. The 620 system interface is shown in Figure.

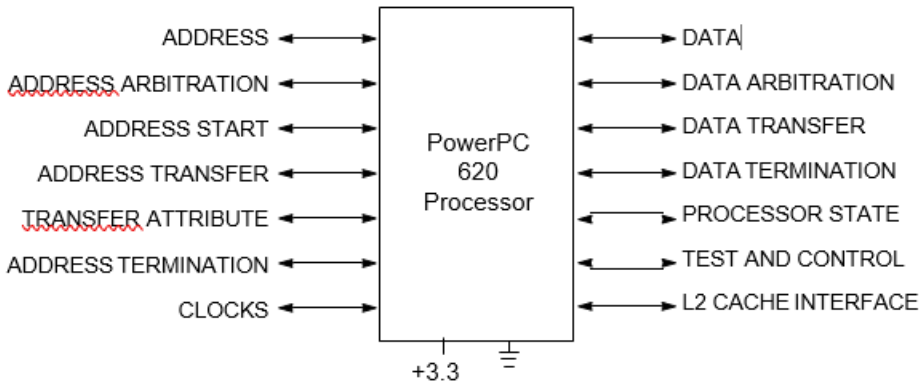


Figure 4.4. System Interface

Four-beat (or eight-beat, if in 64-bit data bus mode) burst-read memory operations that load a 16-word cache block into one of the on-chip caches are the most common bus transactions in typical systems, followed by burst-write memory operations, direct-store operations, and single-beat (noncacheable or write-through) memory read and write operations. Additionally, there can be address-only operations, data-only operations, variants of the burst and single-beat operations (global memory operations that are snooped and atomic memory operations, for example), and address retry activity.

Memory accesses can occur in single-beat or four-beat burst data transfers. The address and

data buses are independent for memory accesses to support pipelining and split transactions, and all bus operations are explicitly tagged through the use of 8-bit tags for addresses and data. The 620 supports bus pipelining and out-of-order split-bus transactions.

Typically, memory accesses are weakly-ordered. Sequences of operations, including load and store string/ multiple instructions, do not necessarily complete in the same order in which they began—maximizing the efficiency of the bus without sacrificing coherency of the data. The 620 allows load operations to precede store operations (except when a dependency exists). In addition, the 620 provides a separate queue for snoop push operations so these operations can access the bus ahead of previously queued operations. The 620 dynamically optimizes run-time ordering of load/store traffic to improve overall performance.

Access to the system interface is granted through an external arbitration mechanism that allows devices to compete for bus mastership. This arbitration mechanism is flexible, allowing the 620 to be integrated into systems that use various fairness and bus-parking procedures to avoid arbitration overhead. Additional multiprocessor support is provided through coherency mechanisms that provide snooping, external control of the on-chip caches and TLBs, and support for a secondary cache. The PowerPC architecture provides the load/store with reservation instruction pair (lwarx/stwcx.) for atomic memory references and other operations useful in multiprocessor implementations.

The following sections describe the 620 bus support for memory and direct-store operations. Note that some signals perform different functions depending upon the addressing protocol used.

Memory Accesses

Memory accesses allow transfer sizes of 8, 16, 24, 32, 64, or 128 bits in one bus clock cycle. Data transfers occur in either single-beat, four-beat, or eight-beat burst transactions. A single-beat transaction transfers as much as 128 bits. Single-beat transactions are caused by noncached accesses that access memory directly (that is, reads and writes when caching is disabled, caching-inhibited accesses, and stores in write-through mode). Burst transactions, which always transfer an entire cache block (64 bytes), are initiated when a block in the cache is read from or written to memory. Additionally, the 620 supports address-only transactions used to invalidate entries in other processors' TLBs and caches, and data-only transactions in which modified data is provided by a snooping device during a read operation to both the bus master and the memory system.

Typically I/O accesses are performed using the same protocol as memory accesses.

Signals

The 620's signals are grouped as follows:

- **Address arbitration signals—The 620 uses these signals to arbitrate for address bus mastership.**
- **Address transfer start signals—These signals indicate that a bus master has begun a transaction on the address bus.**
- **Address transfer signals—These signals, which consist of the address bus, address parity, and address parity error signals, are used to transfer the address and to ensure the integrity of the transfer.**

- **Transfer attribute signals—These signals provide information about the type of transfer, such as the transfer size and whether the transaction is bursted, write-through, or caching-inhibited.**
- **Address transfer termination signals—These signals are used to acknowledge the end of the address phase of the transaction. They also indicate whether a condition exists that requires the address phase to be repeated.**
- **Data arbitration signals—The 620 uses these signals to arbitrate for data bus mastership.**
- **Data transfer signals—These signals, which consist of the data bus, data parity, and data parity error signals, are used to transfer the data and to ensure the integrity of the transfer.**
- **Data transfer termination signals—Data termination signals are required after each data beat in a data transfer. In a single-beat transaction, the data termination signals also indicate the end of the tenure, while in burst accesses, the data termination signals apply to individual beats and indicate the end of the tenure only after the final data beat. They also indicate whether a condition exists that requires the data phase to be repeated.**
- **System status signals—These signals include the interrupt signal, checkstop signals, and both soft reset and hard reset signals. These signals are used to interrupt and, under various conditions, to reset the processor.**
- **Processor state signal—This signal is used to indicate the state of the reservation coherency bit.**
- **Miscellaneous signals—These signals are used in conjunction with such resources as secondary caches and the time base facility.**
- **COP interface signals—The common on-chip processor (COP) unit is the master clock control unit and it provides a serial interface to the system for performing built-in self test (BIST) on all internal memory arrays.**
- **Clock signals—These signals determine the system clock frequency. These signals can also be used to synchronize multiprocessor systems.**

Signal Configuration

Figure illustrates the logical pin configuration of the 620, showing how the signals are grouped.

Clocking

The 620 has a phase-locked loop (PLL) that generates the internal processor clock. The input, or reference signal, to the PLL is the bus clock. The feedback in the PLL guarantees that the processor clock is phase locked to the bus clock, regardless of process variations, temperature changes, or parasitic capacitances. The PLL also ensures a 50% duty cycle for the processor clock.

The 620 supports the following processor-to-bus clock frequency ratios—2:1, 3:1, and 4:1, although not all ratios are available for all frequencies. For more information about the configuration of the PLL, refer to the 620 hardware specifications.

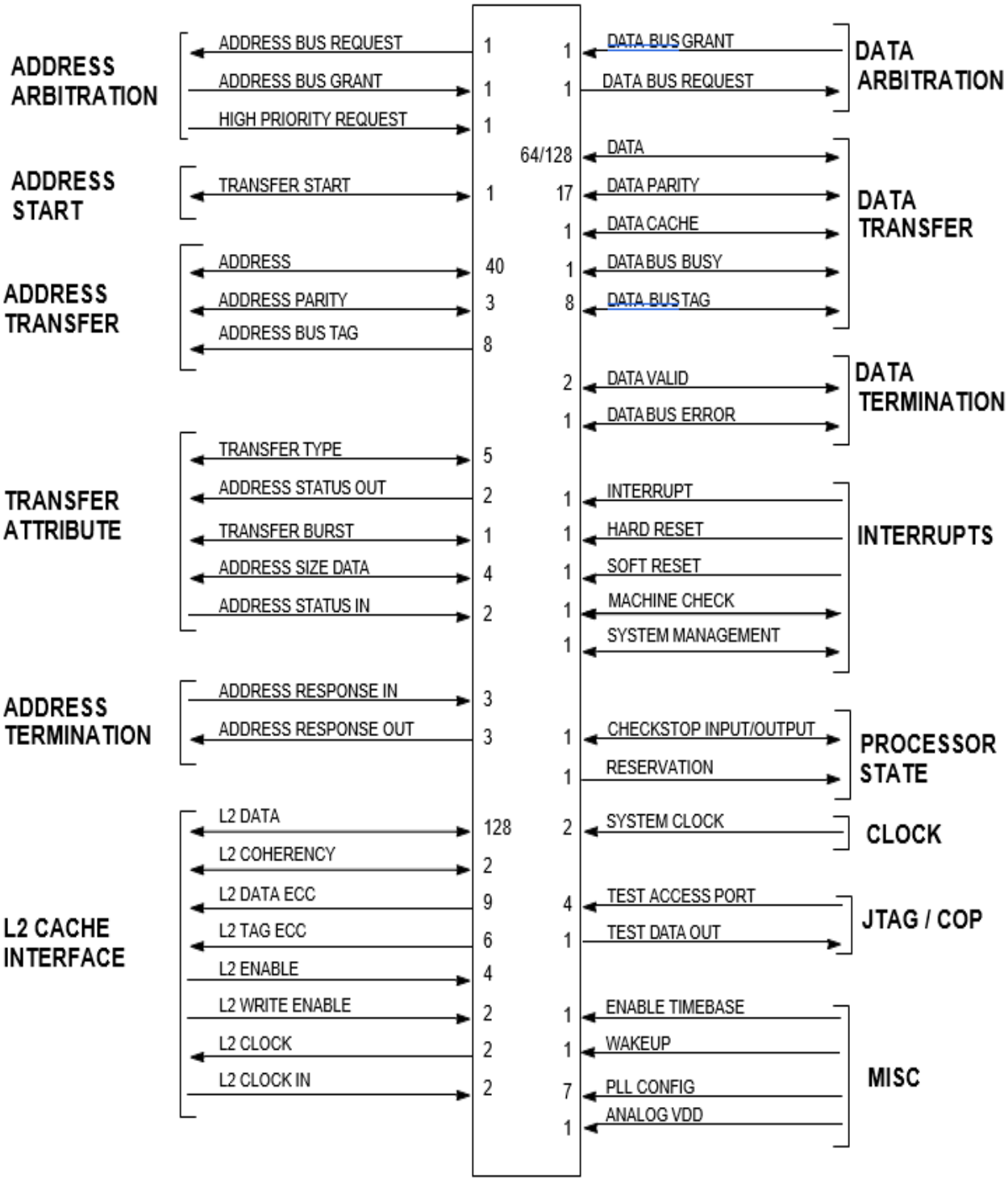


Figure 4.5. PowerPC 620 Microprocessor Signal Groups

PowerPC 620 Microprocessor Execution Model

This section describes the following characteristics of the 620's execution model:

- **The PowerPC architecture**
- **The 620 register set and programming model**
- **The 620 instruction set**
- **The 620 exception model**

- **Instruction timing on the 620**
- **The 620 is a high-performance, superscalar PowerPC implementation of the PowerPC architecture. Like other PowerPC processors, it adheres to the PowerPC architecture specifications but also has additional features not defined by the architecture. These features do not affect software compatibility. The PowerPC architecture allows optimizing compilers to schedule instructions to maximize performance through efficient use of the PowerPC instruction set and register model. The multiple, independent execution units in the 620 allow compilers to maximize parallelism and instruction throughput. Compilers that take advantage of the flexibility of the PowerPC architecture can additionally optimize instruction processing of the PowerPC processors.**

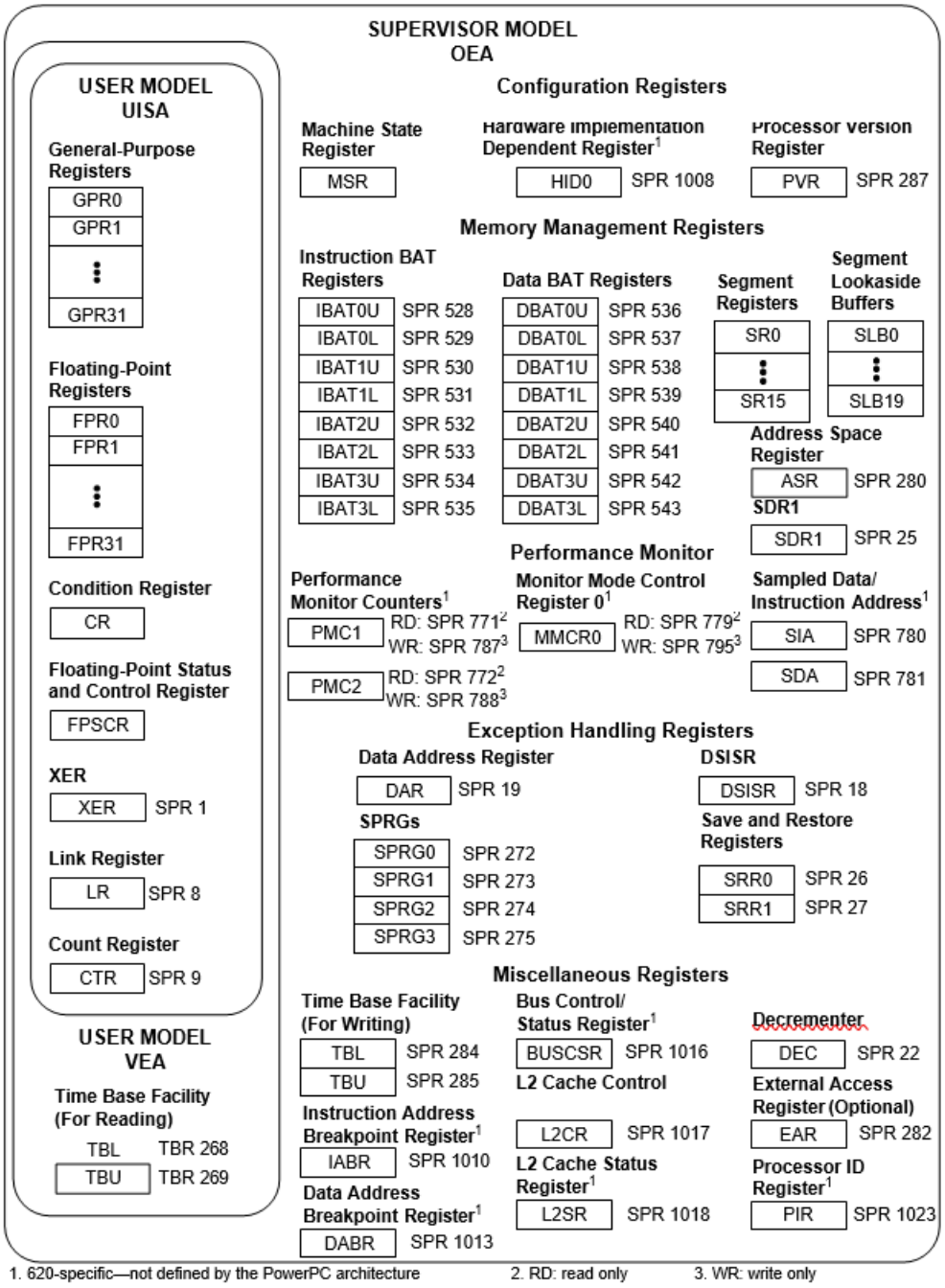
Registers and Programming Model

The PowerPC architecture defines register-to-register operations for most computational instructions. Source operands for these instructions are accessed from the registers or are provided as immediate values embedded in the instruction opcode. The three-register instruction format allows specification of a target register distinct from the two source operands. Load and store instructions transfer data between registers and memory.

During normal execution, a program can access the registers, shown in Figure 6, depending on the program's access privilege (supervisor or user, determined by the privilege-level (PR) bit in the machine state register (MSR)). Note that registers such as the general-purpose registers (GPRs) and floating-point registers (FPRs) are accessed through operands that are part of the instructions. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (mtspr) and Move from Special-Purpose Register (mfspr) instructions) or implicitly as the part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

The numbers to the right of the SPRs indicate the number that is used in the syntax of the instruction operands to access the register.

Figure shows the registers implemented in the 620, indicating those that are defined by the PowerPC architecture and those that are 620-specific.



1. 620-specific—not defined by the PowerPC architecture 2. RD: read only 3. WR: write only

Fig 4.6 Register implementation

PowerPC processors have two levels of privilege—supervisor mode of operation (typically used by the operating environment) and one that corresponds to the user mode of operation (used by application software). As shown in Figure 6, the programming model incorporates 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Note that each PowerPC implementation has its own unique set of implementation-dependent registers that are typically used for debugging, configuration, and other implementation-specific operations.

Some registers are accessible only by supervisor-level software. This division allows the operating system to control the application environment (providing virtual memory and protecting operating-system and critical machine resources). Instructions that control the state of the processor, the address translation mechanism, and supervisor registers can be executed only when the processor is in supervisor mode.

The PowerPC registers implemented in the 620 are summarized as follows:

- **General-purpose registers (GPRs)—The PowerPC architecture defines 32 user-level, general- purpose registers (GPRs). These registers are 32 bits wide in 32-bit PowerPC implementations and 64 bits wide in 64-bit PowerPC implementations. The 620 also has eight GPR rename buffers, which provide a way to buffer data intended for the GPRs, reducing stalls when the results of one instruction are required by a subsequent instruction. The use of rename buffers is not defined by the PowerPC architecture, and they are transparent to the user with respect to the architecture. The GPRs and their associated rename buffers serve as the data source or destination for instructions executed in the IUs.**
- **Floating-point registers (FPRs)—The PowerPC architecture also defines 32 floating-point registers (FPRs). These 64-bit registers typically are used to provide source and target operands for user- level, floating-point instructions. The 620 has eight FPR rename buffers that provide a way to buffer data intended for the FPRs, reducing stalls when the results of one instruction are required by a subsequent instruction. The rename buffers are not defined by the PowerPC architecture. The FPRs and their associated rename buffers can contain data objects of either single- or double-precision floating-point formats.**
- **Condition register (CR)—The CR is a 32-bit user-level register that consists of eight 4-bit fields that reflect the results of certain operations, such as move, integer and floating-point compare, arithmetic, and logical instructions, and provide a mechanism for testing and branching. The 620 also has 16 CR rename buffers, which provide a way to buffer data intended for the CR. The rename buffers are not defined by the PowerPC architecture.**
- **Floating-point status and control register (FPSCR)—The floating-point status and control register (FPSCR) is a user-level register that contains all exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE-754 standard.**
- **Machine state register (MSR)—The machine state register (MSR) is a supervisor-level register that defines the state of the processor. The contents of this register are saved when an exception is taken and restored when the exception handling completes. The 620 implements the MSR as a 64-bit register that provides a superset of the 32-bit functionality.**
- **Segment registers (SRs)—For memory management, 32-bit PowerPC implementations**

use sixteen 32-bit segment registers (SRs). The 620 provides 16 segment registers for use when executing programs compiled for 32-bit PowerPC microprocessors.

- **Special-purpose registers (SPRs)—The PowerPC operating environment architecture defines numerous special-purpose registers that serve a variety of functions, such as providing controls, indicating status, configuring the processor, and performing special operations. Some SPRs are accessed implicitly as part of executing certain instructions. All SPRs can be accessed by using the move to/from SPR instructions, mtspr and mfspr.**
  - **User-level SPRs—The following SPRs are accessible by user-level software:**
    - **Link register (LR)—The link register can be used to provide the branch target address and to hold the return address after branch and link instructions. The LR is 64 bits wide.**
    - **Count register (CTR)—The CTR is decremented and tested automatically as a result of branch and count instructions. The CTR is 64 bits wide.**
    - **XER—The 32-bit XER contains the integer carry and overflow bits.**
    - **Time base registers (TBL and TBU)—The TBL and TBU can be read by user-level software, but can be written to only by supervisor-level software.**
  - **Supervisor-level SPRs—The 620 also contains SPRs that can be accessed only by supervisor- level software. These registers consist of the following:**
    - **The 32-bit data DSISR defines the cause of DSI and alignment exceptions.**
    - **The data address register (DAR) is a 64-bit register that holds the address of an access after an alignment or DSI exception.**
    - **The decrementer register (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. In the 620, the decrementer frequency is equal to the bus clock frequency (as is the time base frequency).**
    - **The 32-bit SDR1 register specifies the page table format used in logical-to-physical address translation for pages.**
    - **The machine status save/restore register 0 (SRR0) is a 64-bit register that is used by the 620 for saving the address of the instruction that caused the exception, and the address to return to when a Return from Interrupt (rfi) instruction is executed.**
    - **The machine status save/restore register 1 (SRR1) is a 64-bit register used to save machine status on exceptions and to restore machine status when an rfi instruction is executed.**
    - **SPRG[0–3] registers are 64-bit registers provided for operating system use.**
    - **The external access register (EAR) is a 32-bit register that controls access to the external control facility through the External Control In Word Indexed (eciwx) and External Control Out Word Indexed (ecowx) instructions.**
    - **The processor version register (PVR) is a 32-bit, read-only register that identifies the version (model) and revision level of the PowerPC processor.**
    - **The time base registers (TBL and TBU) together provide a 64-bit time base register. The registers are implemented as a 64-bit counter, with the least-significant bit being the most frequently incremented. The PowerPC architecture**

defines that the time base frequency be provided as a subdivision of the processor clock frequency. In the 620. the time base frequency is equal to the bus clock frequency (as is the decrementer frequency). Counting is enabled by the Time Base Enable (TBE) signal.

- **The address space register (ASR) is a 64-bit register that holds the physical address of the segment table. The segment table defines the set of memory segments that can be addressed.**
- **Block address translation (BAT) registers—The PowerPC architecture defines 16 BAT registers, divided into four pairs of data BATs (DBATs) and four pairs of instruction BATs (IBATs).**

The 620 includes the following registers not defined by the PowerPC architecture:

- **Instruction address breakpoint register (IABR)—This register can be used to cause a breakpoint exception to occur if a specified instruction address is encountered.**
- **Data address breakpoint register (DABR)—This register can be used to cause a breakpoint exception to occur if a specified data address is encountered.**
- **Hardware implementation-dependent register 0 (HID0)—This register is used to control various functions within the 620, such as enabling checkstop conditions, and locking, enabling, and invalidating the instruction and data caches.**
- **Bus control and status register (BUSCSR)—This register controls the setting of various bus operational parameters, and provides read-only access to bus control values set at system boot time.**
- **L2 cache control register (L2CR)—The L2 cache control register provides controls for the operation of the L2 cache interface, including the ECC mode desired, size of the L2 cache, and the selection of GTL or CMOS interface logic.**
- **L2 cache status register (L2SR)—The L2 cache status register contains all ECC error information for the L2 cache interface.**
- **Processor identification register (PIR)—The PIR is a supervisor-level register that has a right- justified, 4-bit field that holds a processor identification tag used to identify a particular 620. This tag is used to identify the processor in multiple-master implementations.**
- **Performance monitor counter registers (PMC1 and PMC2). The counters are used to record the number of times a certain event has occurred.**
- **Monitor mode control register 0 and 1 (MMCR0 and MMCR1)—These registers are used for enabling various performance monitoring interrupt conditions and establishing the function of the counters.**
- **Sampled instruction address and sampled data address registers (SIA and SDA)—These registers hold the addresses for instruction and data used by the performance monitoring interrupt.**

Note that while it is not guaranteed that the implementation of HID registers is consistent among PowerPC processors, other processors may be implemented with similar or identical HID registers.

Instruction Set and Addressing Modes

The following subsections describe the PowerPC instruction set and addressing modes.

PowerPC Instruction Set and Addressing Modes

All PowerPC instructions are encoded as single-word (32-bit) opcodes. Instruction formats are consistent among all instruction types, permitting efficient decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplifies instruction pipelining.

Instruction Set

The 620 implements the entire PowerPC instruction set (for 64-bit implementations) and most optional PowerPC instructions. The PowerPC instructions can be loosely grouped into the following general categories:

- **Integer instructions—These include computational and logical instructions.**
  - **Integer arithmetic instructions**
  - **Integer compare instructions**
  - **Logical instructions**
  - **Integer rotate and shift instructions**
- **Floating-point instructions—These include floating-point computational instructions, as well as instructions that affect the FPSCR. Floating-point instructions include the following:**
  - **Floating-point arithmetic instructions**
  - **Floating-point multiply/add instructions**
  - **Floating-point rounding and conversion instructions**
  - **Floating-point compare instructions**
  - **Floating-point move instructions**
  - **Floating-point status and control instructions**
  - **Optional floating-point instructions (listed with the optional instructions below)**

  The 620 supports all IEEE 754-1985 floating-point data types (normalized, denormalized, NaN, zero, and infinity) in hardware, eliminating the latency incurred by software exception routines.

  The PowerPC architecture also supports a non-IEEE mode, controlled by a bit in the FPSCR. In this mode, denormalized numbers, NaNs, and some IEEE invalid operations are not required to conform to IEEE standards and can execute faster. Note that all single-precision arithmetic instructions are performed using a double-precision format. The floating-point pipeline is a single-pass implementation for double-precision products. A single-precision instruction using only single- precision operands in double-precision format performs the same as its double-precision equivalent.
- **Load/store instructions—These include integer and floating-point load and store instructions.**
  - **Integer load and store instructions**
  - **Integer load and store multiple instructions**
  - **Integer load and store string instructions**
  - **Floating-point load and store**
- **Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect**

the instruction flow.

- **Branch and trap instructions**
- **System call and rfi instructions**
- **Condition register logical instructions**

- **Synchronization instructions—The instructions are used for memory synchronizing, especially useful for multiprocessing.**
  - **Load and store with reservation instructions—These UISA-defined instructions provide primitives for synchronization operations such as test and set, compare and swap, and compare memory.**
  - **The Synchronize instruction (sync)—This UISA-defined instruction is useful for synchronizing load and store operations on a memory bus that is shared by multiple devices.**
  - **The Enforce In-Order Execution of I/O instruction (eieio)—The eieio instruction, defined by the VEA, can be used instead of the sync instruction when only memory references seen by I/ O devices need to be ordered. The 620 implements eieio as a barrier for all storage accesses to the BIU, but not as a barrier for all instructions like the implementation of the sync instruction.**
- **Processor control instructions—These instructions are used for synchronizing memory accesses and managing caches, TLBs, segment registers and SLBs. These instructions include move to/from special-purpose register instructions (mtspr and mfspr).**
- **Memory/cache control instructions—These instructions provide control of caches, TLBs, segment registers, and SLBs.**
  - **User- and supervisor-level cache instructions**
  - **Segment lookaside buffer management instructions**
  - **Segment register manipulation instructions**
  - **Translation lookaside buffer management instructions**
- **Optional instructions—The 620 implements the following optional instructions:**
  - **The eciwx/ecowx instruction pair**
  - **TLB invalidate entry instruction (tlbie)**
  - **TLB synchronize instruction (tlbsync)**
  - **SLB invalidate entry instruction (slbie)**
  - **SLB invalidate all instruction (slbia)**
  - **Optional graphics instructions:**
    - **Store Floating-Point as Integer Word Indexed (stfiwx)**
    - **Floating Reciprocal Estimate Single (fres)**
    - **Floating Reciprocal Square Root Estimate (frsqrte)**
    - **Floating Square Root Single (fsqrts)**
    - **Floating Square Root Double (fsqrt)**
    - **Floating Select (fsel)**

Note that this grouping of the instructions does not indicate which execution unit executes a particular instruction or group of instructions.

Integer instructions operate on byte, half-word, word, and double-word operands. Floating-point instructions operate on single-precision (one word) and double-precision (one double word) floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, word, and double-word operand loads and stores between memory and a set of 32 GPRs. It also provides for word and double-word operand loads and stores between memory and a set of 32 FPRs.

Computational instructions do not modify memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location with specific store instructions.

PowerPC processors follow the program flow when they are in the normal execution state. However, the flow of instructions can be interrupted directly by the execution of an instruction or by an asynchronous event. Either kind of exception may cause one of several components of the system software to be invoked.

Exception handlers should save the information stored in SRR0 and SRR1 early to prevent the program state from being lost due to a system reset or machine check exception or to an instruction-caused exception in the exception handler.

The PowerPC architecture supports four types of exceptions:

- **Synchronous, precise—These are caused by instructions. All instruction-caused exceptions are handled precisely; that is, the machine state at the time the exception occurs is known and can be completely restored.**
- **Synchronous, imprecise—The PowerPC architecture defines two imprecise floating-point exception modes, recoverable and nonrecoverable. The 620 treats the imprecise, recoverable and imprecise, nonrecoverable modes as the precise mode.**
- **Asynchronous—The OEA portion of the PowerPC architecture defines two types of asynchronous exceptions:**
  - **Asynchronous, maskable—The PowerPC architecture defines the external interrupt and decrementer interrupt which are maskable and asynchronous exceptions. In the 620, and in many PowerPC processors, the hardware interrupt is generated by the assertion of the Interrupt (INT) signal, which is not defined by the architecture. In addition, the 620 implements one additional interrupt, the system management interrupt, which performs similarly to the external interrupt, and is generated by the assertion of the System Management Interrupt (SMI) signal.**

    When these exceptions occur, their handling is postponed until all instructions, and any exceptions associated with those instructions, complete execution.
  - **Asynchronous, nonmaskable—There are two nonmaskable asynchronous exceptions that are imprecise—system reset and machine check exceptions. Note that the OEA portion of the PowerPC architecture, which defines how these exceptions work, does not define the causes or the signals used to cause these exceptions. These exceptions may not be recoverable, or may provide a limited degree of recoverability for diagnostic purposes.**

The PowerPC architecture defines two bits in the MSR—FE0 and FE1—that determine how floating-point exceptions are handled. There are four combinations of bit settings, of which the 620 implements two, which are as follows:

- **Ignore exceptions mode—In this mode, the instruction dispatch logic feeds the FPU as fast as possible and the FPU uses an internal pipeline to allow overlapped execution of instructions. In this mode, floating-point exception conditions return a predefined value instead of causing an exception.**
- **Precise interrupt mode—This mode includes both the precise mode and imprecise recoverable and nonrecoverable modes defined in the PowerPC architecture. In this mode, a floating-point instruction that causes a floating-point exception brings the machine to a precise state. In doing so, the 620 takes floating-point exceptions as defined by the PowerPC architecture.**

Instruction Timing

As shown in Figure , the common pipeline of the 620 has five stages through which all instructions must pass. Some instructions occupy multiple stages simultaneously and some individual execution units have additional stages. For example, the floating-point pipeline consists of three stages through which all floating-point instructions must pass.

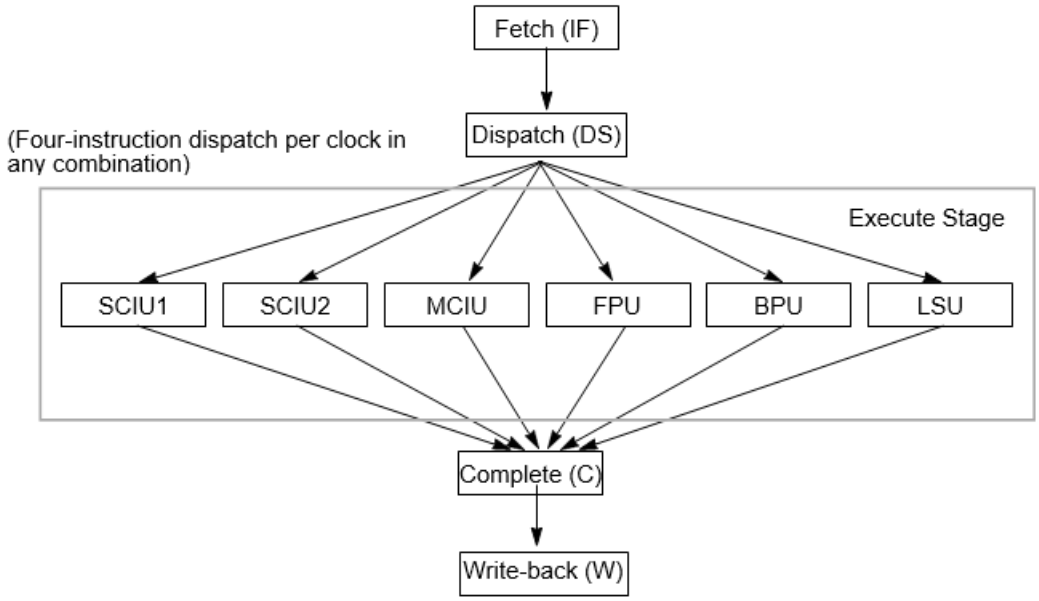


Figure 4.7. Pipeline Diagram

The common pipeline stages are as follows:

- **Instruction fetch (IF)—During the IF stage, the fetch unit loads the decode queue (DEQ) with instructions from the instruction cache and determines from what address the next instruction should be fetched.**
- **Instruction dispatch (DS)—During the dispatch stage, the decoding that is not time-critical is performed on the instructions provided by the previous IF stage. Logic associated with this stage determines when an instruction can be dispatched to the appropriate execution unit. At the end of the DS stage, instructions and their operands**

are latched into the execution input latches or into the unit's reservation station. Logic in this stage allocates resources such as the rename registers and reorder buffer entries.

- **Execute (E)—While the execution stage is viewed as a common stage in the 620 instruction pipeline, the instruction flow is split among the six execution units, some of which consist of multiple pipelines. An instruction may enter the execute stage from either the dispatch stage or the execution unit's dedicated reservation station.**

  At the end of the execute stage, the execution unit writes the results into the appropriate rename buffer entry and notifies the completion stage that the instruction has finished execution.

  The execution unit reports any internal exceptions to the completion stage and continues execution, regardless of the exception. Under some circumstances, results can be written directly to the target registers, bypassing the rename buffers.

- **Complete (C)—The completion stage ensures that the correct machine state is maintained by monitoring instructions in the completion buffer and the status of instruction in the execute stage.**

  When instructions complete, they are removed from the reorder buffer. Results may be written back from the rename buffers to the register as early as the complete stage. If the completion logic detects an instruction containing exception status or if a branch has been mispredicted, all subsequent instructions are cancelled, any results in rename buffers are discarded, and instructions are fetched from the correct instruction stream.

- **Write-back (W)—The write-back stage is used to write back any information from the rename buffers that was not written back during the complete stage. The CR, CTR, and LR are updated during the write-back stage.**

All instructions are fully pipelined except for divide operations and some integer multiply operations. The integer multiplier is a three-stage pipeline. SPR and divide operations can execute in the MCIU in parallel with multiply operations. The floating-point pipeline has three stages. All floating-point instructions are fully pipelined except for divide and square root operations.

PENTIUM 6 (P6)

The P6 family of processors use a dynamic execution micro-architecture. This three-way superscalar, pipelined micro-architecture features a decoupled, multi-stage superpipeline, which trades less work per pipestage for more stages. A P6 family processor, for example, has twelve stages with a pipestage time 33 percent less than the Pentium processor, which helps achieve a higher clock rate on any given manufacturing process.The approach used in the P6 family micro-architecture removes the constraint of linear instruction sequencing between the traditional "fetch" and "execute" phases, and opens up a wide instruction window using an instruction pool. This approach allows the "execute" phase of the processor to have much more visibility into the program instruction stream so that better scheduling may take place. It requires the instruction "fetch/decode" phase of the processor to be much more efficient in terms of predicting program flow. Optimized scheduling requires the fundamental "execute" phase to be replaced by decoupled "dispatch/execute" and "retire" phases. This allows instructions to be started in any order but always be completed in the original program order. Processors in the P6 family may be thought of as three independent engines coupled with an instruction pool as shown in

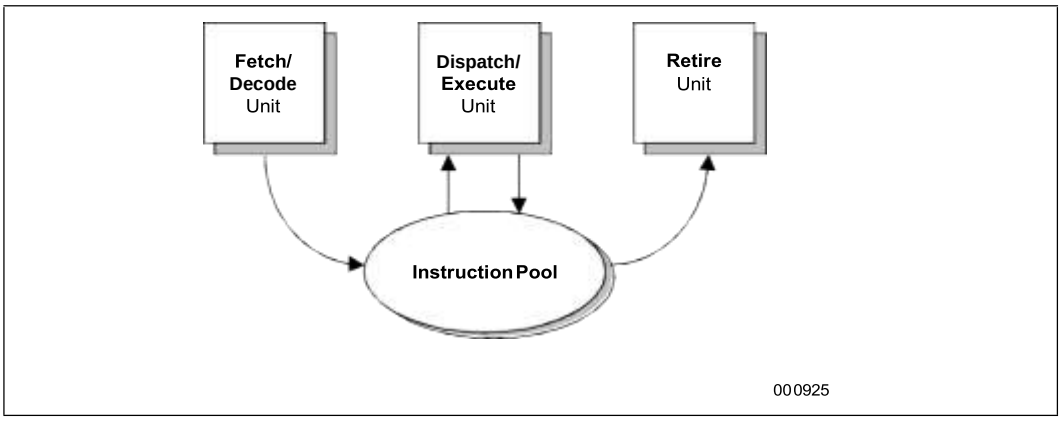


Figure 4.8. Three Engines Communicating Using an Instruction Pool

FULL CORE UTILIZATION

```
r1 <= mem [r0] /* Instruction 1 */
r2 <= r1 + r2 /* Instruction 2 */
r5 <= r5 + 1 /* Instruction 3 */
r6 <= r6 - r3 /* Instruction 4 */
```

Fig 4.9 A typical pseudo code fragment

The three independent-engine approach was taken to more fully utilize the processor core.The first instruction in this example is a load of r1 that, at run time, causes a cache miss. A traditional processor core must wait for its bus interface unit to read this data from main memory and return it before moving on to instruction 2. This processor stalls while waiting for this data and is thus being under-utilized.To avoid this memory latency problem, a P6 family processor "looks-ahead" into the instruction pool at subsequent instructions and does useful work rather than stalling. In the example in Figure, instruction 2 is not executable since it depends upon the result of instruction 1; however both instructions 3 and 4 have no prior dependencies and are therefore executable. The processor executes instructions 3 and 4 out-of-order. The results of this out-of-order execution cannot be committed to permanent machine state (i.e., the programmer-visible registers) immediately since the original program order must be maintained. The results are instead stored back in the instruction pool awaiting in-order retirement. The core executes instructions depending upon their readiness to execute, and not on their original program order, and is therefore a true dataflow engine. This approach has the side effect that instructions are typically executed out-of-order.The cache miss on instruction 1 will take many internal clocks, so the core continues to look ahead for other instructions that could be speculatively executed, and is typically looking 20 to 30 instructions in front of the instruction pointer. Within this 20 to 30 instruction window there will be, on average, five branches that the fetch/decode unit must correctly predict if the dispatch/ execute unit is to do useful work. The sparse register set of an Intel Architecture (IA) processor will create many false dependencies on registers so the dispatch/execute unit will rename the Intel Architecture registers into a larger register set to enable additional forward progress. The Retire Unit owns the programmer's Intel Architecture register set and results are only committed to permanent machine state in these registers when it removes completed

instructions from the pool in original program order.Dynamic Execution technology can be summarized as optimally adjusting instruction execution by predicting program flow, having the ability to speculatively execute instructions in any order, and then analyzing the program's dataflow graph to choose the best order to execute the instructions.

THE P6 FAMILY PROCESSOR PIPELINE

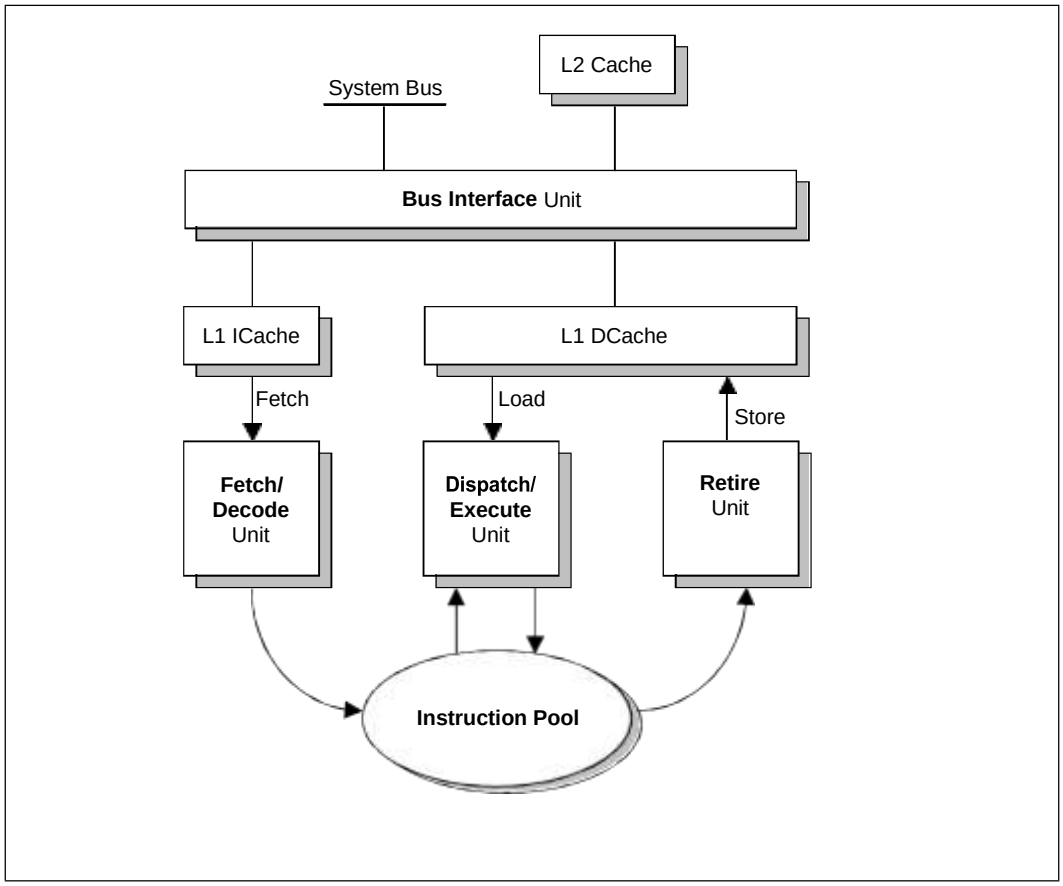


Figure 4.10. The Three Core Engines Interface with Memory via Unified Caches

In order to get a closer look at how the P6 family micro-architecture implements Dynamic Execution, Figure shows a block diagram of the P6 family of processor products including cache and memory interfaces. The "Units" shown in Figure 2-3 represent stages of the P6 family of processors pipeline.

The FETCH/DECODE unit: An in-order unit that takes as input the user program instruction stream from the instruction cache, and decodes them into a series of µoperations (µops) that represent the dataflow of that instruction stream. The pre-fetch is speculative.

The DISPATCH/EXECUTE unit: An out-of-order unit that accepts the dataflow stream, schedules execution of the µops subject to data dependencies and resource availability and temporarily stores the results of these speculative executions.

The RETIRE unit: An in-order unit that knows how and when to commit ("retire") the temporary, speculative results to permanent architectural state.

The BUS INTERFACE unit: A partially ordered unit responsible for connecting the three internal units to the real world. The bus interface unit communicates directly with the L2 (second level) cache supporting up to four concurrent cache accesses. The bus interface unit also controls a transaction bus, with MESI snooping protocol, to system memory.

The Fetch/Decode Unit

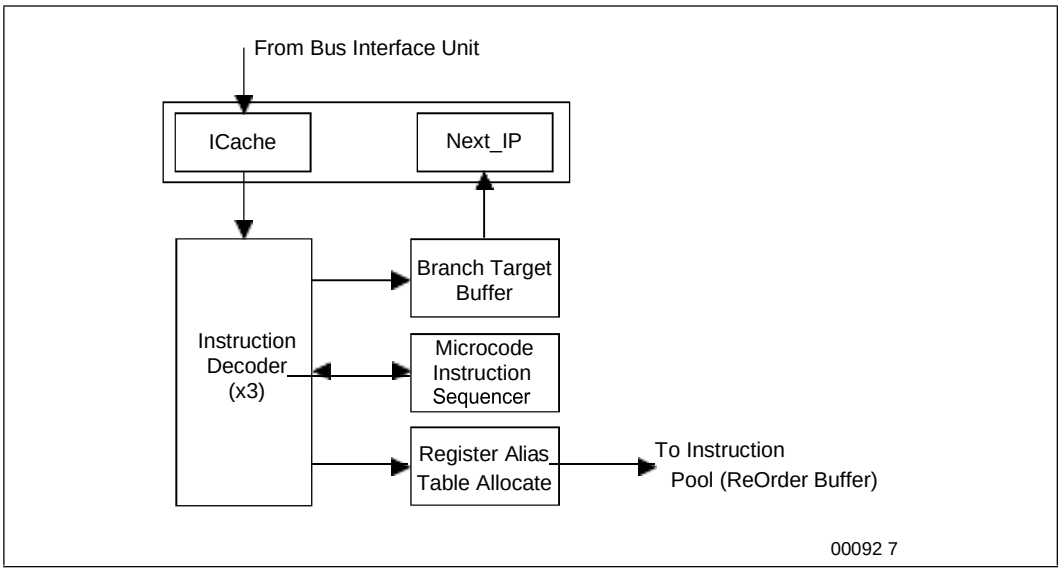


Figure 4.11. Inside the Fetch/Decode Unit

The L1 Instruction Cache is a local instruction cache. The Next_IP unit provides the L1 Instruction Cache index, based on inputs from the Branch Target Buffer (BTB), trap/interrupt status, and branch-misprediction indications from the integer execution section.The L1 Instruction Cache fetches the cache line corresponding to the index from the Next_IP, and the next line, and presents 16 aligned bytes to the decoder. The prefetched bytes are rotated so that they are justified for the instruction decoders (ID). The beginning and end of the Intel Architecture instructions are marked.Three parallel decoders accept this stream of marked bytes, and proceed to find and decode the Intel Architecture instructions contained therein. The decoder converts the Intel Architecture

instructions into triadic µops (two logical sources, one logical destination per µop). Most Intel Architecture instructions are converted directly into single µops, some instructions are decoded into one-to-four µops and the complex instructions require microcode (the box labeled Microcode Instruction Sequencer in Figure 2-4). This microcode is just a set of preprogrammed sequences of normal µops. The µops are queued, and sent to the Register Alias Table (RAT) unit, where the logical Intel Architecture-based register references are converted into references to physical registers in P6 family processors physical register references, and to the Allocator stage, which adds status information to the µops and enters them into the instruction pool. The instruction pool is implemented as an array of Content Addressable Memory called the ReOrder Buffer (ROB).

The Dispatch/Execute Unit

The Dispatch unit selects µops from the instruction pool depending upon their status. If the status indicates that a µop has all of its operands then the dispatch unit checks to see if the execution resource needed by that µop is also available. If both are true, the Reservation Station removes that µop and sends it to the resource where it is executed. The results of the µop are later returned to the pool. There are five ports on the Reservation Station, and the multiple resources are accessed as shown in Figure

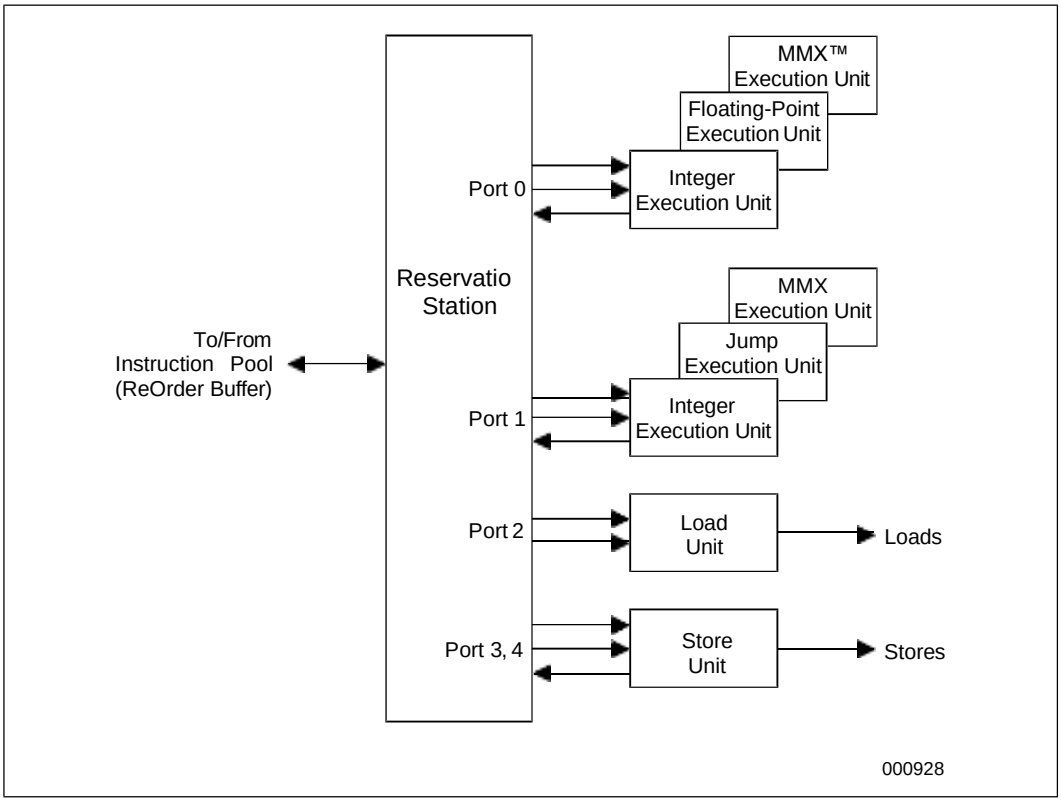


Figure 4.12. Inside the Dispatch/Execute Unit

The P6 family of processors can schedule at a peak rate of 5 µops per clock, one to each resource port, but a sustained rate of 3 µops per clock is more typical. The activity of this scheduling process is the out-of-order process; µops are dispatched to the execution resources strictly according to dataflow constraints and resource availability, without regard to the original ordering of the program.

Note that the actual algorithm employed by this execution-scheduling process is vitally important to performance. If only one µop per resource becomes data-ready per clock cycle, then there is no choice. But if several are available, it must choose. The P6 family micro-architecture uses a pseudo FIFO scheduling algorithm favoring back-to-back µops. Note that many of the µops are branches. The Branch Target Buffer (BTB) will correctly predict most of these branches but it can't correctly predict them all. Consider a BTB that is correctly predicting the backward branch at the bottom of a loop; eventually that loop is going to terminate, and when it does, that branch will be mispredicted. Branch µops are tagged (in the in-order pipeline) with their fall-through address and the destination that was predicted for them. When the branch executes, what the branch actually did is compared against what the prediction hardware said it would do. If those coincide, then the branch eventually retires and the speculatively executed work between it and the next branch instruction in the instruction pool is good. But if they do not coincide, then the Jump Execution Unit (JEU) changes the status of all of the µops behind the branch to remove them from the instruction pool. In that case the proper branch destination is provided to the BTB which restarts the whole pipeline from the new target address.The Retire Unit Figure shows a more detailed view of the Retire Unit.

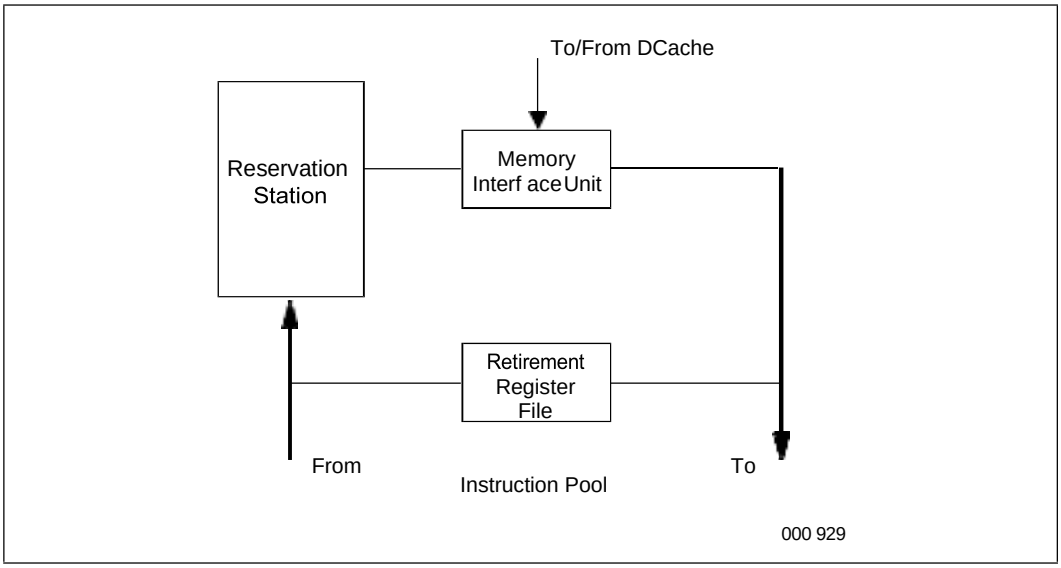


Figure 4.13. Inside the Retire Unit

The Retire Unit is also checking the status of µops in the instruction pool. It is looking for µops that have executed and can be removed from the pool. Once removed, the original architectural target of the µops is written as per the original Intel Architecture instruction. The Retire Unit must not only notice which µops are complete, it must also re-impose the original program order on them. It must also do this in the face of interrupts, traps, faults, breakpoints and mispredictions. The Retire Unit must first read the instruction pool to find the potential candidates for retirement and determine which of these candidates are next in the original program order. Then it writes the results of this cycle's retirements to the Retirement Register File (RRF). The Retire Unit is capable of retiring 3 µops per clock.

The Bus Interface Unit Figure shows a more detailed view of the Bus Interface Unit.

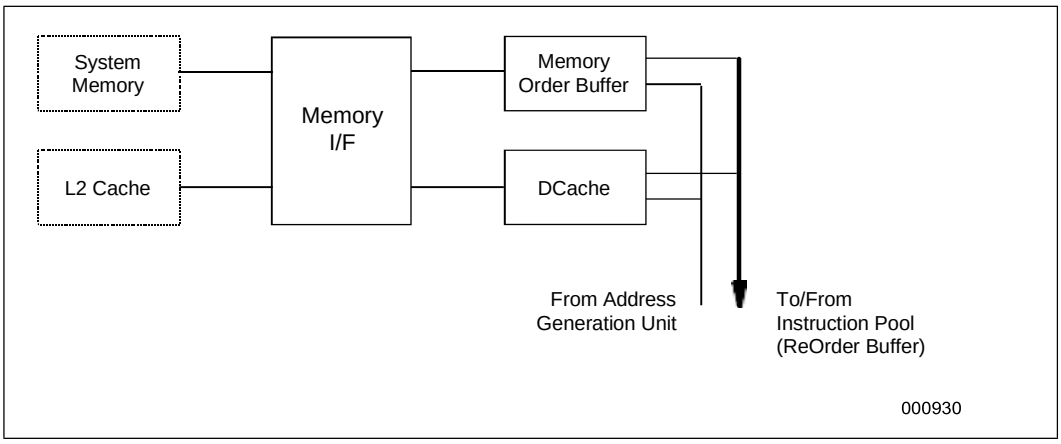


Figure 4.14. Inside the Bus Interface Unit

There are two types of memory access: loads and stores. Loads only need to specify the memory address to be accessed, the width of the data being retrieved, and the destination register. Loads are encoded into a single µop.Stores need to provide a memory address, a data width, and the data to be written. Stores therefore require two µops, one to generate the address and one to generate the data. These µops must later re-combine for the store to complete.Stores are never performed speculatively since there is no transparent way to undo them. Stores are also never re-ordered among themselves. A store is dispatched only when both the address and the data are available and there are no older stores awaiting dispatch. A study of the importance of memory access reordering concluded:

- **Stores must be constrained from passing other stores, for only a small impact on performance.**
- **Stores can be constrained from passing loads, for an inconsequential performance loss.**
- **Constraining loads from passing other loads or stores has a significant impact on performance.**

The Memory Order Buffer (MOB) allows loads to pass other loads and stores by acting like a reservation station and re-order buffer. It holds suspended loads and stores and re-dispatches them when a blocking condition (dependency or resource) disappears.

ERROR CLASSIFICATION

The P6 family processor system bus architecture uses the following error classification. An implementation may always choose to report an error in a more severe category to simplify its logic.

- **Recoverable Error (RE): The error can be corrected by a retry or by using ECC information. The error is logged in the MCA hardware.**
- **Unrecoverable Error (UE): The error cannot be corrected, but it only affects one agent. The memory interface logic and bus pipeline are intact, and can be used to report the error via an exception handler.**
- **Fatal Error (FE): The error cannot be corrected and may affect more than one agent. The memory interface logic and bus pipeline integrity may have been violated, and**

cannot be reliably used to report the error via an exception handler. A bus pipeline reset is required of all bus agents before operation can continue. An exception handler may then proceed.

P6 ARCHITECTURE

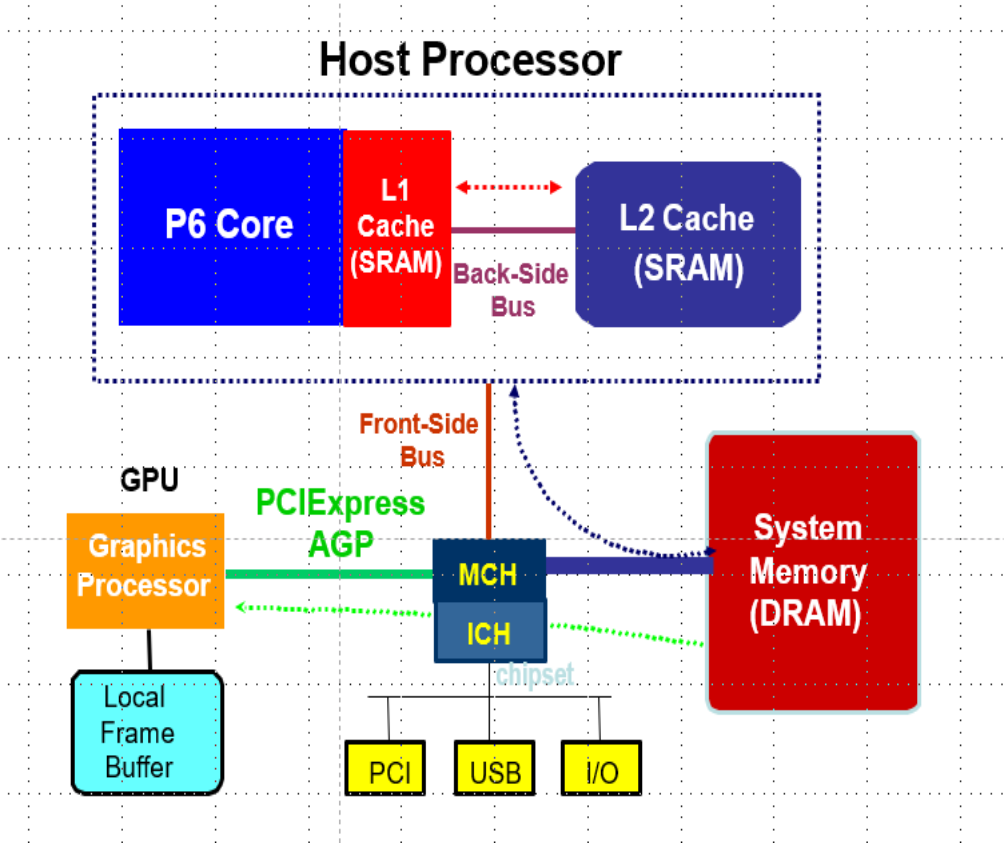


Fig 4.15 P6 system architecture

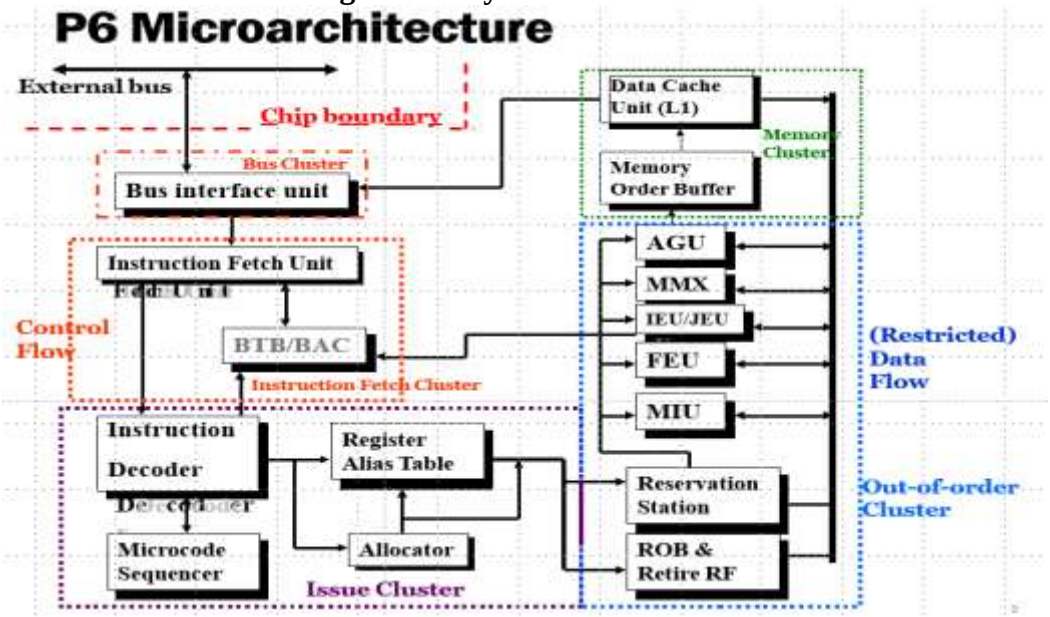


Fig 4.16. P6 micro architecture

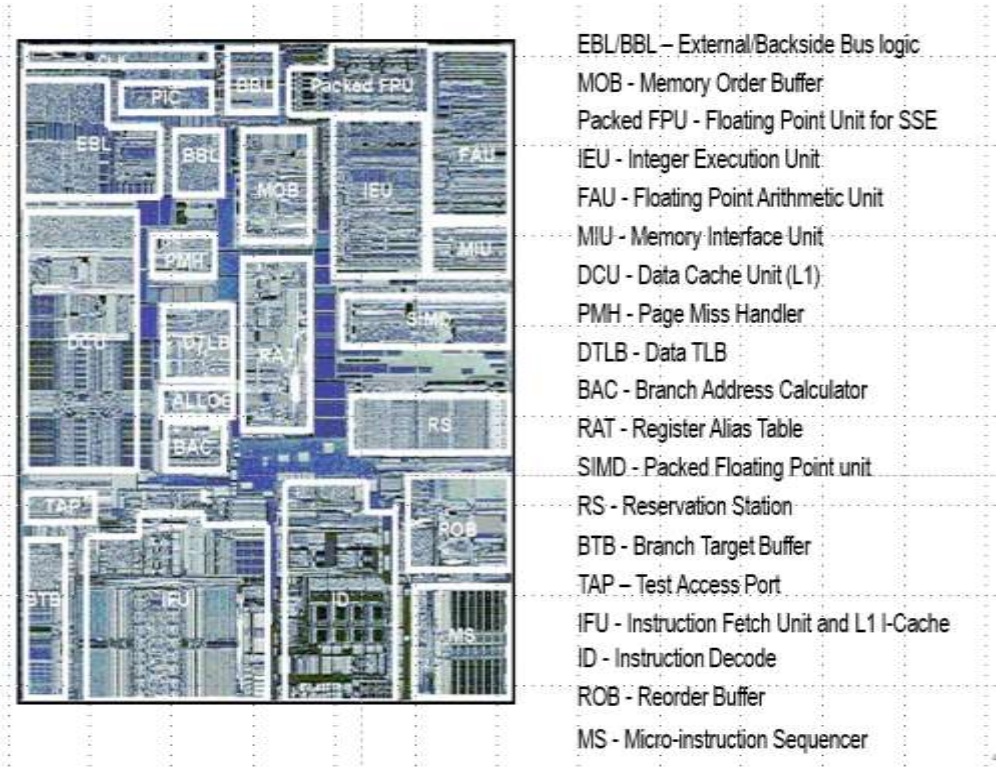


Fig 4.17 P6 Die map

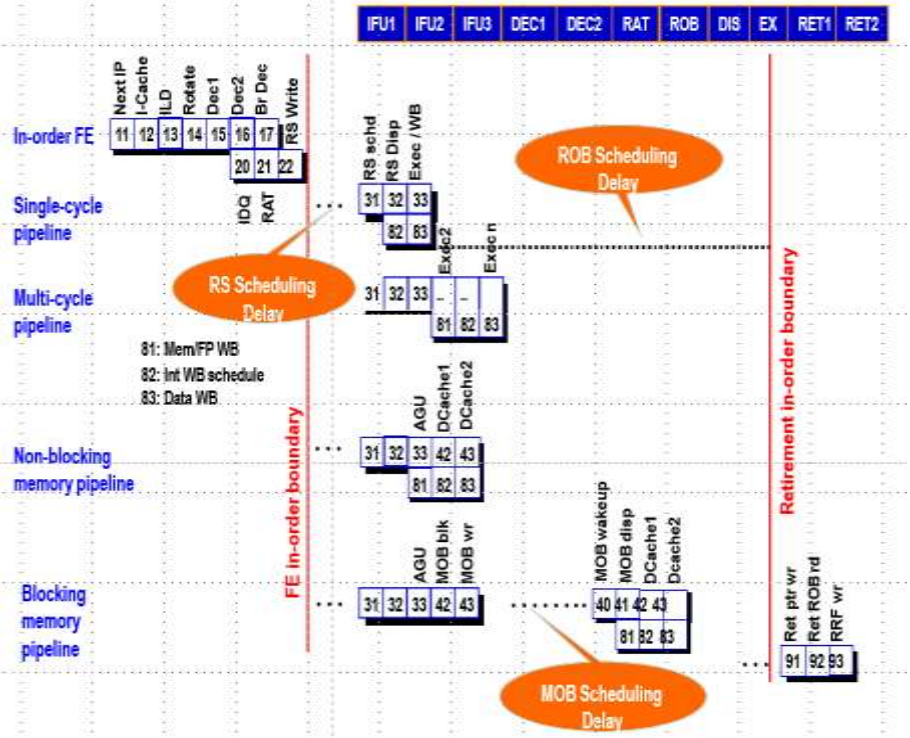
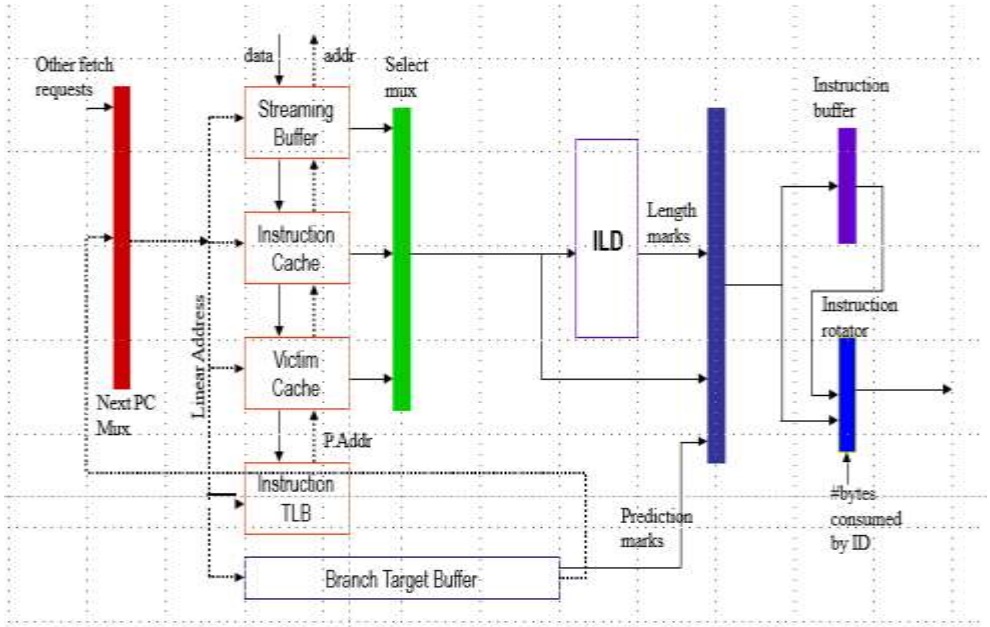
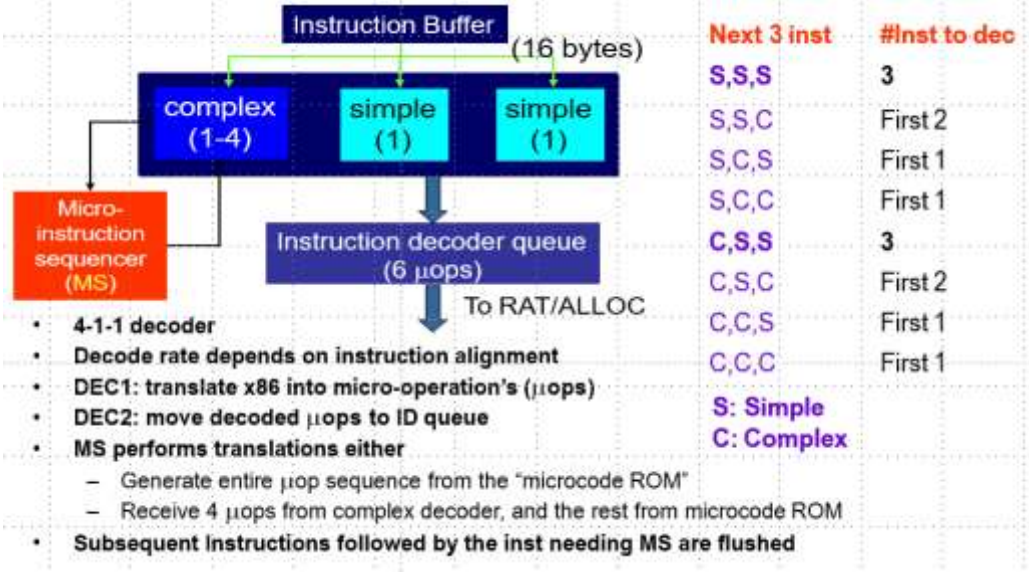


Fig 4.18 P6 pipelining



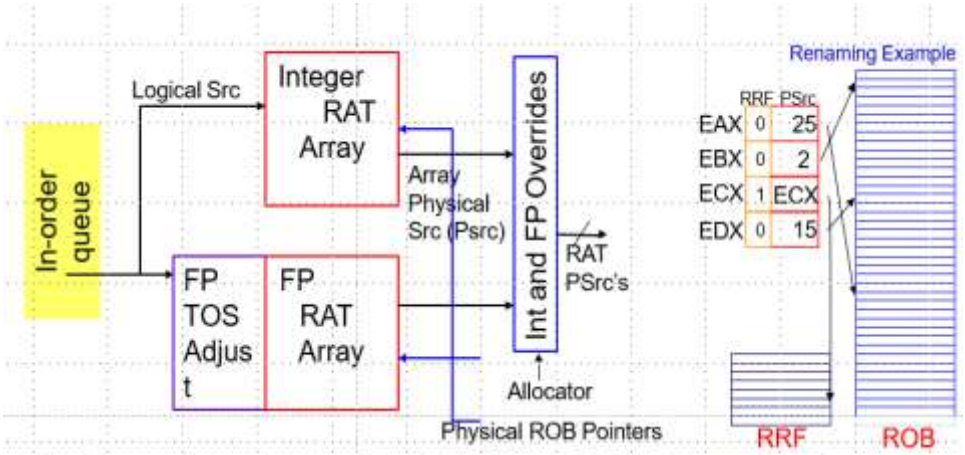
- IFU1: Initiate fetch, requesting 16 bytes at a time
- IFU2: Instruction length decoder, mark instruction boundaries, BTB makes prediction (2 cycles)
- IFU3: Align instructions to 3 decoders in 4-1-1 format

Fig 4.19 P6 Instruction fetch unit



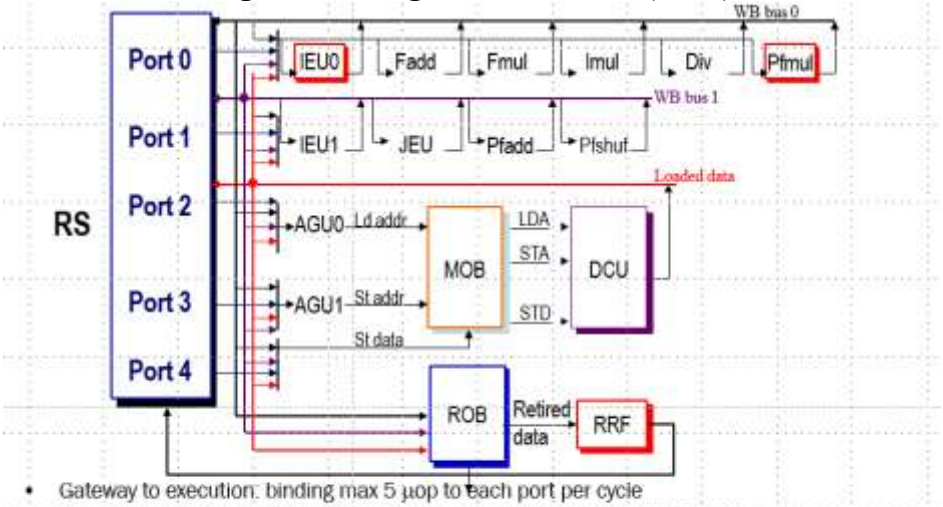
- **4-1-1 decoder**
- **Decode rate depends on instruction alignment**
- **DEC1: translate x86 into micro-operation's (µops)**
- **DEC2: move decoded µops to ID queue**
- **MS performs translations either**
  - Generate entire µop sequence from the "microcode ROM"
  - Receive 4 µops from complex decoder, and the rest from microcode ROM
- **Subsequent instructions followed by the inst needing MS are flushed**

Fig 4.20 P6 Instruction Decode



- Register renaming for 8 integer registers, 8 floating point (stack) registers and flags: 3 µop per cycle
- 40 80-bit physical registers embedded in the ROB (thereby, 6 bit to specify PSrc)
- RAT looks up physical ROB locations for renamed sources based on RRF bit
- Override logic is for dependent µops decoded at the same cycle
- Misprediction will revert all pointers to point to Retirement Register File (RRF)

Fig 4.21 P6 Register Alias table (RAT)



- Gateway to execution: binding max 5 µop to each port per cycle
- Port binding at dispatch time (certain µop can only be bound to one port)
- 20 µop entry buffer bridging the In-order and Out-of-order engine (32 entries in Core)
- RS fields include µop opcode, data valid bits, Pdst, Psrc, source data, BrPred, etc.
- Oldest first FIFO scheduling when multiple µops are ready at the same cycle

Fig 4.22 P6 Reservation station

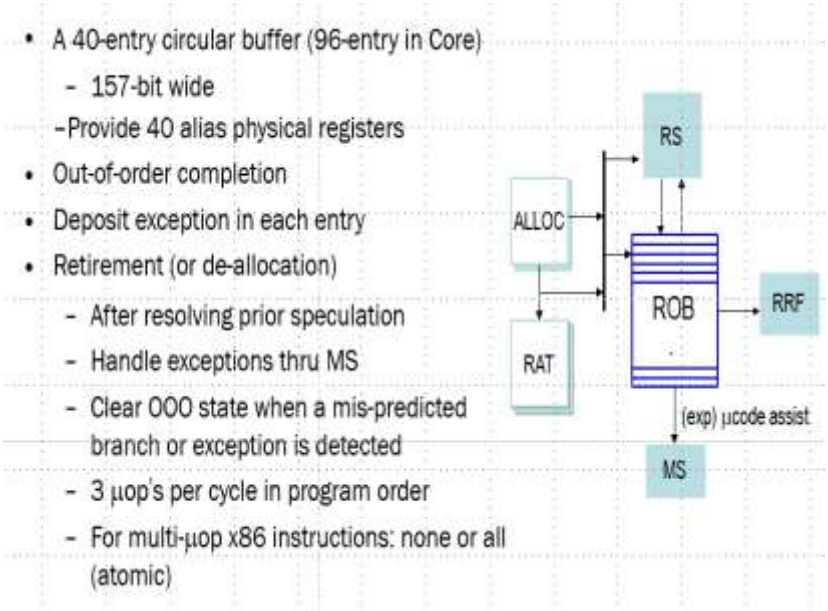


Fig 4.23 P6 Reorder buffer

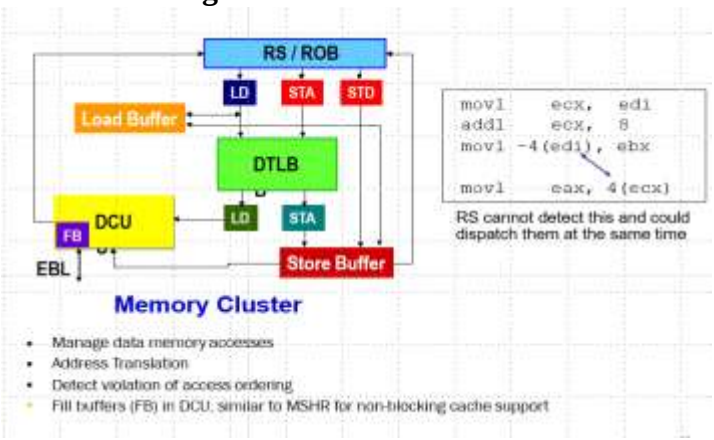


Fig 4.24 P6 Memory execution cluster

MEMORY TYPE RANGE REGISTERS (MTRR)

- **Control registers written by the system (OS)**
- **Supporting Memory Types**
  - **UnCacheable (UC)**
  - **Uncacheable Speculative Write-combining (USWC or WC)**
    - **Use a fill buffer entry as WC buffer**
  - **WriteBack (WB)**
  - **Write-Through (WT)**
  - **Write-Protected (WP)**
    - **E.g. Support copy-on-write in UNIX, save memory space by allowing child processes to share with their parents. Only create new memory pages when child processes attempt to write.**
- **Page Miss Handler (PMH)**
  - **Look up MTRR while supplying physical addresses**

Return memory types and physical address to DTLB

TEXT / REFERENCE BOOKS

1. **Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing", 2nd Edition,Pearson Education, Inc., 2004**
2. **Barry B.Brey , "The Intel Microprocessors 8086/8088, 8086, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, Architecture, Programming and interfacing", Prentice Hall of India Private Limited, New Delhi, 2003**
3. **Alan Clements, "The Principles of computer Hardware", Oxford University Press, 3rd Edition, 2003**
4. **John Paul Shen, Mikko H.Lipasti, "Modern Processor Design", Tata McGraw Hill, 2006**

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMMUNICATION ENGINEERING

UNIT - 5
ADVANCED MICROPROCESSORS – SEC1601

UNIT 5 RISC PROCESSORS II(SUPERSCALAR PROCESSORS)
Conversion of Color Models; Basic of Full-Color Image Processing; Color Transformations; Smoothing; Sharpening; Segmentation; Applications of Image Processing - Motion Analysis, Image Fusion, Image Classification

INTRODUCTION

Image processing is a very important topic in computer science due to the large number of applications being used in our everyday life. The common challenge for most of the image processing applications is the large number of operations required. Microprocessors are often used in image processing. Although today's microprocessors are much more powerful than the ones ten years ago, the amount of data acquired by image sensors has also increased rapidly. Furthermore, many microprocessors are designed for general purpose with the ability to process audio, image, video, and network packet data. However, the increased number of functionalities results in a very complicated internal circuit with a large instructions set. In terms of image processing, they are inefficient compared to a dedicated image processor. Therefore, there is a need to design a simple, highly efficient and low power image processor for embedded systems. After an intensive research, it suggested that Single Instruction Multiple Data (SIMD) architectures could significantly increase the processing speed. This will provide an on-board fast image processing solution for real time image processing applications.

This image processor will not replace any general-purpose microprocessor. It, will however, enhance the overall system performance by offloading the image processing task from the general-purpose microprocessor. This can be used in a large number of embedded systems, such as Raspberry Pi, Arduino, BeagleBone, and more. Figure shows the setup of such system.

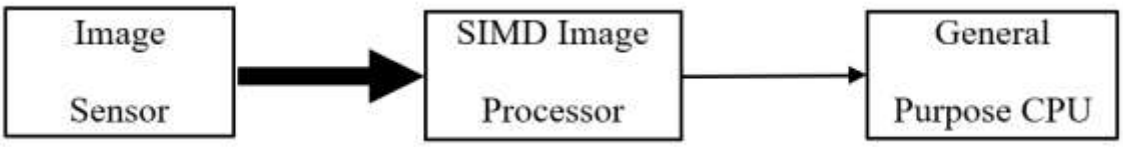


Fig 5.1 Image Data Flow in an Embedded System

In this setup, the raw digit image captured by the image sensor is first sent to our image processor. The image is processed by our SIMD image processor. The final result is then delivered to the on-board general-purpose CPU. This can significantly reduce the work load of the on-board CPU by shifting the image processing task away from the CPU. Also, it reduces the bandwidth usage by sending results instead of raw data into the on-board CPU. The on-board CPU can spend more computational power and allocate more bandwidth to other tasks, which results in an increase in the performance of the overall system.

In many image processing applications, multiple pixels can be processed

simultaneously. Focuses on the development of an image processor using a parallel architecture called Single Instruction Multiple Data (SIMD).

It takes a three step approach:

- **Improve and further develop Retro**
- **Design SIMD circuit**
- **Verify the correctness by simulating a number of image processing operations**

Digital Image

Images you see on your television and capture by your smartphone are digital images. Unlike a continuous picture drawn using a pencil, a digital image is a discrete image. It is made up of many small square elements termed pixel. Each pixel is filled with a colour. A digital image is represented as a matrix in digital world. Each number in the matrix specifies the colour of each pixel in the image.

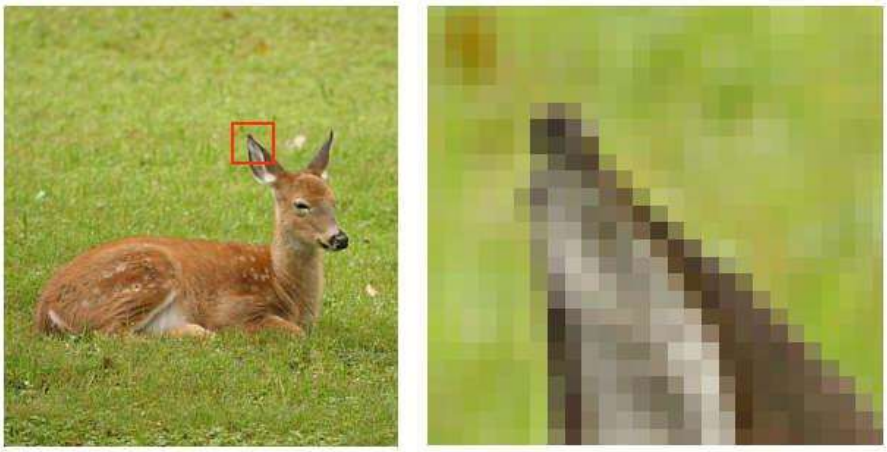


Fig 5.2 Pixel

There are three types of digital images – binary image, greyscale image and colour image. A binary image is a 1-bit (monochrome) image. A pixel of a binary (black and white) image has only 2 possible values - 1 for black and 0 for white

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

Fig 5.3 Binary Image

In addition to binary image, a greyscale image use 8-bit of data to represent each pixel. In other words, a pixel in a greyscale image has 256 possible values. These values represent different darkness levels ranging from 0 for pure black to 255 for pure white. All other values give grey colours with different intensities.

256	256	256	256	256	256	256	256	256	256
256	256	256	256	256	256	256	256	256	256
256	256	128	128	128	128	128	128	256	256
256	256	128	128	128	128	128	128	256	256
256	256	128	128	0	0	128	128	256	256
256	256	128	128	0	0	128	128	256	256
256	256	128	128	128	128	128	128	256	256
256	256	128	128	128	128	128	128	256	256
256	256	256	256	256	256	256	256	256	256
256	256	256	256	256	256	256	256	256	256

Fig 5.4 Greyscale Image

Unlike binary and greyscale images, there are a number of colour systems used to represent a colour image. The most widely used system is called RGB. A colour image is made up of three separate layers – Red, Green, and Blue. Each colour layer is a greyscale image. A pixel in a colour image carries three intensity values, and each one corresponds to each colour component. An artist can create lots of different colours by mixing them differently.

Similarly, the combination of different intensity of red, green and blue results in millions of different colours. Although there are some other colour systems currently being used (such as CMYK for printing), focus on RGB images

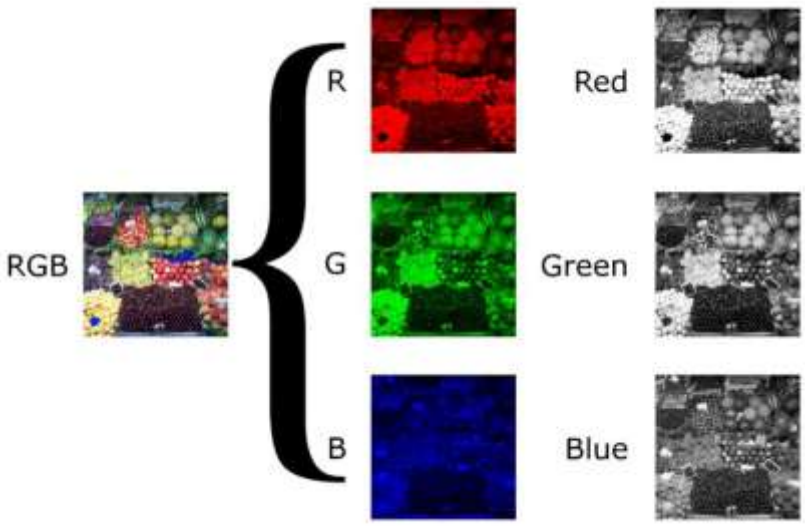


Fig 5.5 RGB Image

The term colour depth is used to describe the number of possible colours a pixel can choose from. A binary image is also known as 1-bit monochrome image since each pixel has 2^1 possible values. As mention before, a greyscale image is an 8-bit image with 2^8 possible grey intensity levels. On the other hand, colour image has many different colour depth values. Some common colour depth values are: 8-bit, 15/16-bit (High Colour), 24-bit (True

Colour), and 30/36/48-bit (Deep Colour). A higher colour depth gives a pixel more colours to choose from, which gives a vivid image. However, this also generates a much larger amount of data. For instance, an 8 megapixel 24-bit (8 bits per each RGB component) colour image capture by a consumer digital camera has the size of [4]:

8 * 10^6 (pixels) * 24 (bits/colour) ~ 24Mbytes

This is a lot of data for an embedded system to process.

Image processing

There are millions of image processing applications that have been developed in the past few decades. These applications are used in every industry, such as quality control systems found in an assembly line, optical character recognition (OCR) system found in scanners, and motion detection system found in burglar alarms. They all follow the same procedure –

capturing and processing images. The capturing stage is normally done by image sensors, while the processing stage is done by microprocessors.

Image processing can also be separated into two distinguished levels. The lower level provides hardware infrastructure support. In this level, digital images are treated as a collection of data. The actual content of the image is meaningless. On the other hand, the higher level provides methods and algorithms on how to process digital images. This level is also known as Computer Vision. Starting from a 2D image, computer vision attempts to extract information of a 3D scene

The core of image processing is matrix manipulation. As mentioned previously, a digital image is nothing more than a colour intensity matrix. Operations in both space domain and frequency domain produce various visual effects on the image. We can also generate lots of different images by combining a number of operations differently. Here is a list of some basic matrix operations and their corresponding effects:

- **Transpose – rotation**
- **Amplification – changing the brightness and contrast**
- **Addition – Colour offset**
- **Finding the standard arithmetic mean of RGB component – convert into greyscale image**
- **Convolution/filtering – blurring, sharpening, edge detection Etc**

Most of the image processing applications require a lot of processing power. For instance, if we blur an eight megapixel example using a 15 x 15 Gaussian Filter, the number of operations required roughly equals:

8 x 10^6 (pixels) x 15 x 15 ~ 5.4 billion operations

General purpose microprocessors used in many embedded systems are inefficient when performing such tasks. For instance, a Cortex A8 microprocessor takes roughly 3 seconds to process the entire image. This can be critical for a lot of real time image processing applications.

Microprocessor

Computer vision provides the algorithms on how to manipulate a digital image, but this will not be possible without the support from its underlying hardware. Microprocessors are commonly used for image processing. In fact, because today's microprocessors are so cheap and capable of performing any sort of task, they have occupied every corner of your digital life.

There are two widely used computer architecture: Harvard Architecture and von Neumann Architecture. Both architectures consist of following four components

- **Control Unit (CU)**
- **Arithmetic Logic Unit (ALU)**
- **Memory**
- **Input / Output**

The main difference between these two architectures is the way instructions and data are stored in the memory. In von Neumann Architecture, both instruction and data are stored in the same memory and hence the microprocessor cannot fetch both during the same cycle. This bottleneck restricts the throughput of the data bus between microprocessor and memory. In order to overcome this performance bottleneck, Harvard Architecture implies two memories, one for instructions and the other for data. Hence the microprocessor can obtain both instruction and data via two separate buses within the same clock cycle

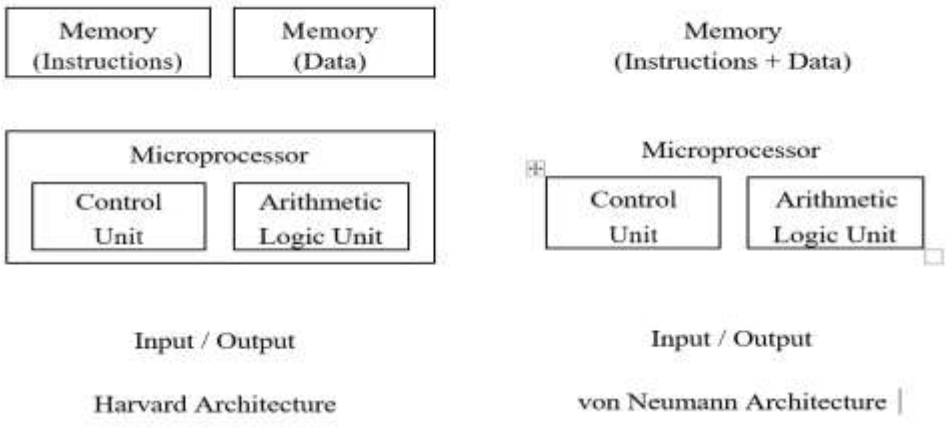


Fig 5.6 Computer Architectures

The first two components, Control Unit and Arithmetic Logic Unit, together form a microprocessor, or Central processing Unit (CPU). Control Unit is the command centre of the entire system. It directs other parts of the system according to the instructions stored in the memory. It does not perform any arithmetic operations on the data. The Arithmetic Logic Unit, on the other hand, performs either arithmetic or logic operations on data according to the instructions received from the Control Unit. Arithmetic operations are mathematical calculations, these include:

- **Additions**
- **Subtraction**
- **Multiplication**
- **Division**
- **Bit shifting**

Logical operations are basically comparisons. These include:

- **Equal to**
- **Less than**
- **Greater than**
- **AND**
- **OR**
- **NOT**
- **XOR**
- **NAND**
- **NOR**

Design based on von Neumann architecture.

Microarchitecture

According to Flynn's taxonomy, microarchitectures can be divided into four classifications

- **Single instruction, single data (SISD)**
- **Single instruction, multiple data (SIMD)**
- **Multiple instruction, single data (MISD)**
- **Multiple instruction, multiple data (MIMD)**

	Instruction	
Data	**SISD** **Single processor Computer**	**SIMD** **Vector/Array Computer**
	MISD **Pipeline Computer**	**MIMD** **Multiprocessor Distributed Computer System**

Table 5.1 Microarchitecture Classifications

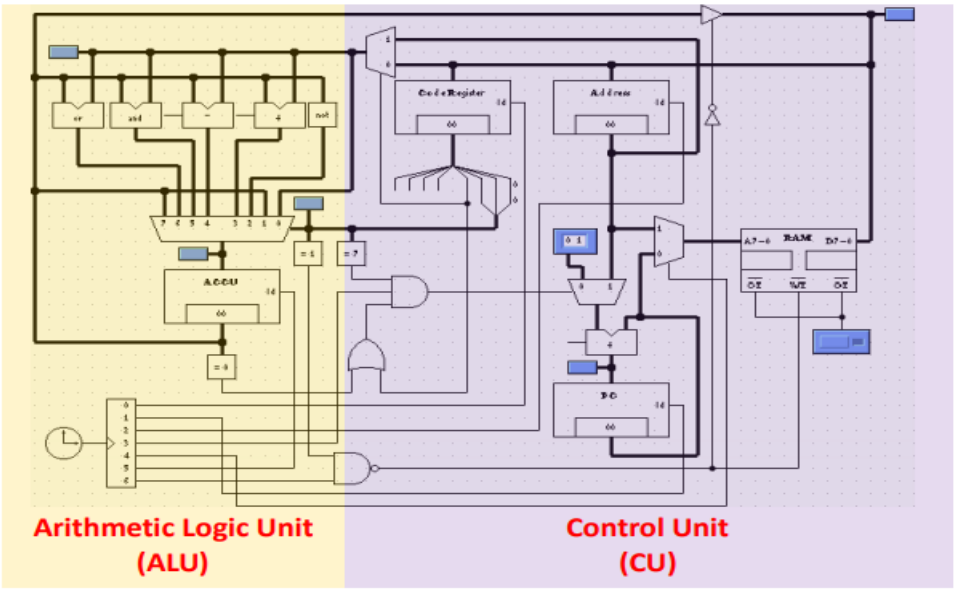


Figure 5.7 Single Instruction Single Data (SISD)

A SISD microprocessor is the most basic architecture. It is made up of one control unit (CU) and one arithmetic logic unit (ALU). A single instruction is executed for every clock cycle, and since there is one ALU, it can store and process only one piece of data. This is very slow for processing a large amount of data because the data has to be queued and processed one by one. A digital image contains a large number of pixels (eg 8 megapixel image contains 8 million pixel). It is a big challenge to process all the pixels in a split second, especially for real time image processing applications. It is clear that SISD architecture is not suitable for image processing. Imagine there is a one-lane street. Cars travelling down the street have to stay in the same lane, one after another. Congestion occurs as the number of cars increases. The common solution is to widen the street with additional lanes. Cars can now travel in different lanes. More cars can pass through for the same time period, or in other words, it takes less time for the same number of cars to go from one end to the other. Also, in both cases, only one traffic light is used to control the traffic flow, regardless the number of lanes.

Figure 5.8 Single Lane Traffic

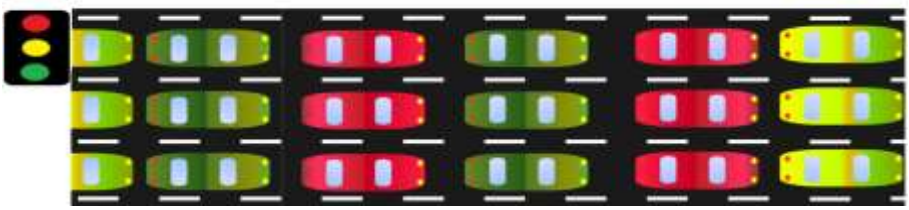


Figure 5.9 Multi Lane Traffic

Similarly, in many image processing applications, the same operations apply to all pixels, and different pixels may have different values. We therefore would like to have a system, which can send the instruction once and process multiple pixels at the same time. There is only one architecture suitable for this design idea – Single Instruction Multiple Data (SIMD) architecture. "Single Instruction" means there is only one CU in the microprocessor and "Multiple Data" means there are multiple ALUs used for multiple data.

Single Instruction Multiple Data (SIMD)

Unlike a SISD architecture, a SIMD microprocessor is made up of one CU and multiple ALUs (also known as Process Element, PE). In a microprocessor, the CU provides both data and instructions to ALU. The ALU then processes the data according to the instruction. These two parts are connected via two buses – a data bus and an instruction bus.The following figure shows the overall layout of our SIMD design.

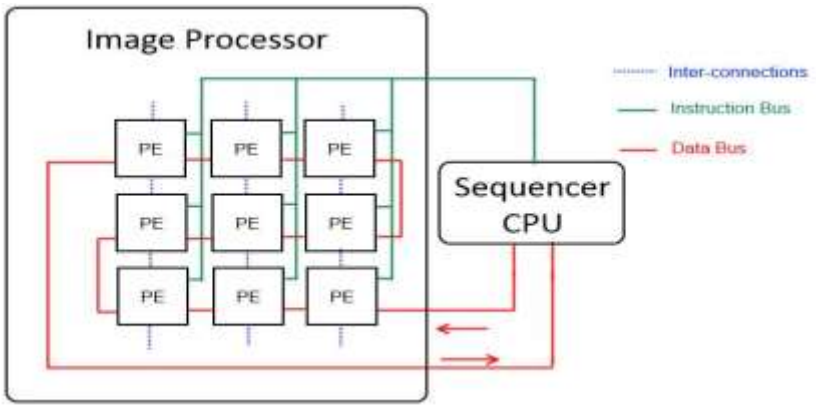


Figure 5.10 SIMD Design Concept

There are two regions in our design – Sequencer CPU and PE network. The Sequencer CPU is based on von Neumann architecture – both data and instructions are stored in the same memory. The role of the sequencer CPU is to provide input data and instructions to all PEs, and collect results from PEs. On the other hand, the role of a PE is much simpler. It is in fact an ALU (sometimes with a small local memory). It processes data according to the instructions received from the sequencer CPU. There are two buses coming out from the sequencer CPU. All PEs are connected to the data bus in series. This data bus is a two way bus. Data coming from the sequencer CPU can be fed into each PE one by one. Data can also be sent to all the PEs in parallel using the instruction bus. This instruction bus is a one way bus and all PEs receive the same instructions and operand from the sequencer CPU at the same time. PEs are also interconnected with their neighbours to form a network grid of PEs. These interconnects are two way buses and they allow PEs to exchange information with their neighbours without going back to the Sequencer CPU.

INSTRUCTION SET

Microprocessor architectures can also be classified by their instruction set. Some widely used architectures are X86, ARM (RISC) and SPARC. Each has its own territory: X86 microprocessors are used in every personal computer; ARM (based of RISC) microprocessors dominate the embedded world; and finally SPARC microprocessors play an important role in many supercomputers. Over the past few decades, the size of the instruction sets of theses microprocessors has grown rapidly. This can be quite challenge for developers to use. In this project, we have decided to create our own instruction set. It is significantly smaller compared to those big names. Simple circuit design and small instruction set are the key features of this image processor.

SIMD Image Microprocessor

SIMD picks up a lot of popularities in recent years. This chapter covers the actual SIMD implementations. There are three parts we have designed in this stage: the instruction set used by the image microprocessor, the internal circuit of a Process Element (PE), and the internal circuit of a Sequencer CPU. Various functions of Retro have also been verified by using it to design the circuits.

Requirements

A number of factors need to be considered when designing a SIMD microprocessor

- **Choice of Process Element (PE)**
- **Communications/network topology**
- **Instruction Issue**

There is always a trade-off in simplicity and functionality for any circuit design. A decision need be made based on the specifications and the purpose of the design. SIMD PEs are normally interconnected to form a network grid. The network topology also needs to be defined before the circuit design process. Last but not least, also need to develop a method by which the instructions coming from Sequencer CPU can be delivered to the PEs without errors.

Since this SIMD microprocessor is designed for image processing applications, it should have the following features:

- **Simple, sufficient and highly efficient minimum waste on transistors**
- **Interconnected to form a 2D network, and ideally, every PE receives one pixel value from the image sensor**
- **PEs are connected the same instruction bus and clock signal in order to achieve a synchronous system**

Instruction Set

This SIMD microprocessor is designed solely for image processing applications, and hence the instruction set can be trimmed down to a very small table. An assembly instruction consists of two parts. The first part is called Op-Code, and the second part is called Operand. Operand can be either a numerical value, or an address location. Different semiconductor companies have different methods to identify the data type. Atmel microprocessors use separate Op-Code for address and numerical value. Motorola microprocessors, on the other hand, use the same Op-Code for both data and address value. A single selection bit is used to differentiate between address and numerical value. The later has a cleaner structure and smaller instruction set with less confusion. There are two different circuits used in the SIMD microprocessor – one for the PE and the other one for the Sequencer CPU.

PE Instruction Set

Each instruction is 15 bit long. It has the following format:

MSB **LSB**

14 to 10	9 to 8	7 to 0
4bit	2bit	8bit
Op-Code	Type: 00 Constant (Seq + PE) 01 Memory (Seq + PE) 10 Neighbour (PE only) 11 Neighbour + Seq (PE only)	Data

Table 5.2 PE Instruction Format

The following table is a list of available Op-Codes for PE:

PE OPCODES

OPCODE	INSTRUCTION	INPUT	DESCRIPTION	OPERATION
0	NOP	-	-	-
1	LOAD	imm, addr or PE	Load constant from memory address into ACCU	ACCU <- RAM(imm)
2	ADD	imm, addr or PE	ADD to ACCU (without Carry in)	ACCU <- ACCU + <data>
3	AND	imm, addr or PE	Invert ACCU	ACCU <- $\overline{ACCU}$
4	NOT	-	AND with ACCU	ACCU <- ACCU · <data>
5	OR	imm, addr or PE	OR with ACCU	ACCU <- ACCU + <data>
6	ADDC	imm, addr or PE	ADD to ACCU (with Carry)	ACCU <- ACCU + <data> + 1
7	-	-	-	-
8	STORE	imm	Store ACCU at input address	RAM(imm) <- ACCU
11	LESSTHAN	imm, addr or PE	Compare <data> to ACCU, result stored in Activity register	ACTIV(0) <- ACCU < <data>
12	EQUAL	imm, addr or PE	Compare <data> to ACCU, result stored in Activity register	ACTIV(0) <- ACCU == <data>
13	ACTIV_INVERT	-	Invert the LSB of the Activity register	ACTIV(0) <- $\overline{ACTIV(0)}$

Table 5.3 PE Opcodes

14	ACTIV_CARRY	-	If Carry set LSB of Activity register to 1	ACTIV(0) <- C
15	-	-	Currently unused	
16	ACTIV_ZERO	-	If Zero set LSB of Activity register to 1	ACTIV(0) <- Z
17	ACTIV_NEGATIVE	-	If Negative set LSB of Activity register to 1	ACTIV(0) <- N
18	STATUS_SHIFTLEFT (STATUS)	-	Bit-shift Activity register one place to the left, bringing in a 1 from the left	ACTIV(i) <- ACTIV(i-1)
19	SHIFT RIGHT (STATUS)	-	Bit-shift Activity register one place to the right, bringing in a 1 from the right	ACTIV(i) <- ACTIV(i+1)

Table 5.4 PE Opcodes(Contd)

Sequencer CPU Instruction Set

The Sequencer CPU also has its own instruction set. Previously we have looked at the instruction set used by PE. A PE does not have CU and it has to obtain instructions from the Sequencer CPU. Therefore the sequencer CPU stores two sets of instructions – instructions for PEs and instructions for Sequencer CPU. To differentiate between these two, a number of flag bits are used in front of each Op-Code. The following table show the format of the instruction.

Op-Code								Data		
7	6	5	4	3	2	1	0	7	to	0

Op-Code

Bit 7 **Sequencer/PE Switch**

0 Sequencer

1 PE

Bit 6 – 5 **Input type**

00 Constant (Seq + PE)

01 Memory (Seq + PE)

10 Neighbour (PE only)

11 Neighbour + Seq (PE only)

Bit 4 – 0 **Op-Code**

Sequencer: 4 bit Op-Code (3-0), bit 4 is unused

PE: Bit 4

0 ACCU

1 STATUS/Logic

Bit 3 – 0

3 bit Op-Code

Table 5.4 Sequencer CPU Instruction Format

For the Op-Code part of the instruction:

Bit 7 indicates whether this instruction is for Sequencer CPU or PE.

Bit 6 – 5 indicates the input data type and where the data is from.

This is same as Bit 9 – 8 used in PE's instruction.

Bit 4 – 0 is the "actual" Op-Code. If this instruction is for Sequencer CPU, then only Bit 3 – 0 are used. If this instruction is for PE, then all 5 bits are used with Bit 4 being a flag. This is same as Bit 14 – 10 used in PE's instruction.

The following table is the instruction set used by the Sequencer CPU.

SEQUENCER CPU OPCODES

OPCODE	INSTRUCTION	INPUT	DESCRIPTION	OPERATION
0	NOP	-	No Operation	-
1	STORE	imm	Store ACCU at input address	RAM(imm) <- ACCU
2	LOAD	imm	Load constant from memory address into ACCU	ACCU <- RAM(imm)
3	ADD	imm or addr	ADD to ACCU (without Carry in)	ACCU <- ACCU + <data>
4	NOT	-	Invert ACCU	ACCU <- $\overline{ACCU}$
5	AND	imm or addr	AND with ACCU	ACCU <- ACCU · <data>
6	OR	imm or addr	OR with ACCU	ACCU <- ACCU + <data>
7	BRA	imm	Branch Ahead	pc <- pc + K
8	-	-	-	-
9	-	-	-	-
A	-	-	-	-
B	BRR	imm	Branch on Ready	IF (rdy==1) pc <- K
C	BRC	imm	Branch on Carry	IF (C==1) pc <- K
D	-	-	-	-
E	BRZ	imm	Branch on zero	IF (Z==1) pc <- K
F	BRN	imm	Branch on negative	IF (N==1) pc <- K

Table 5.5 Sequencer CPU Opcodes

Processes Element (PE)

Each PE is an execution unit and corresponds to each pixel of the image. Multiple PEs are connected to form a PE network and they are controlled by a single sequencer CPU. The same PE internal circuit is repeated multiple times. In reality, a silicon chip has a finite number of transistors, and we would like to fit as many PEs into the chip as possible. It is therefore important to have a simple but highly efficient circuit design.

Figure shows the schematic of the PE internal circuit. It is made up of five regions – ALU, Input/Output, Instruction Bus, Internal Memory, and Status Register. A PE does not have a CU. It relies on the sequencer CPU to provide both instruction and data. The following section covers each of these five regions.

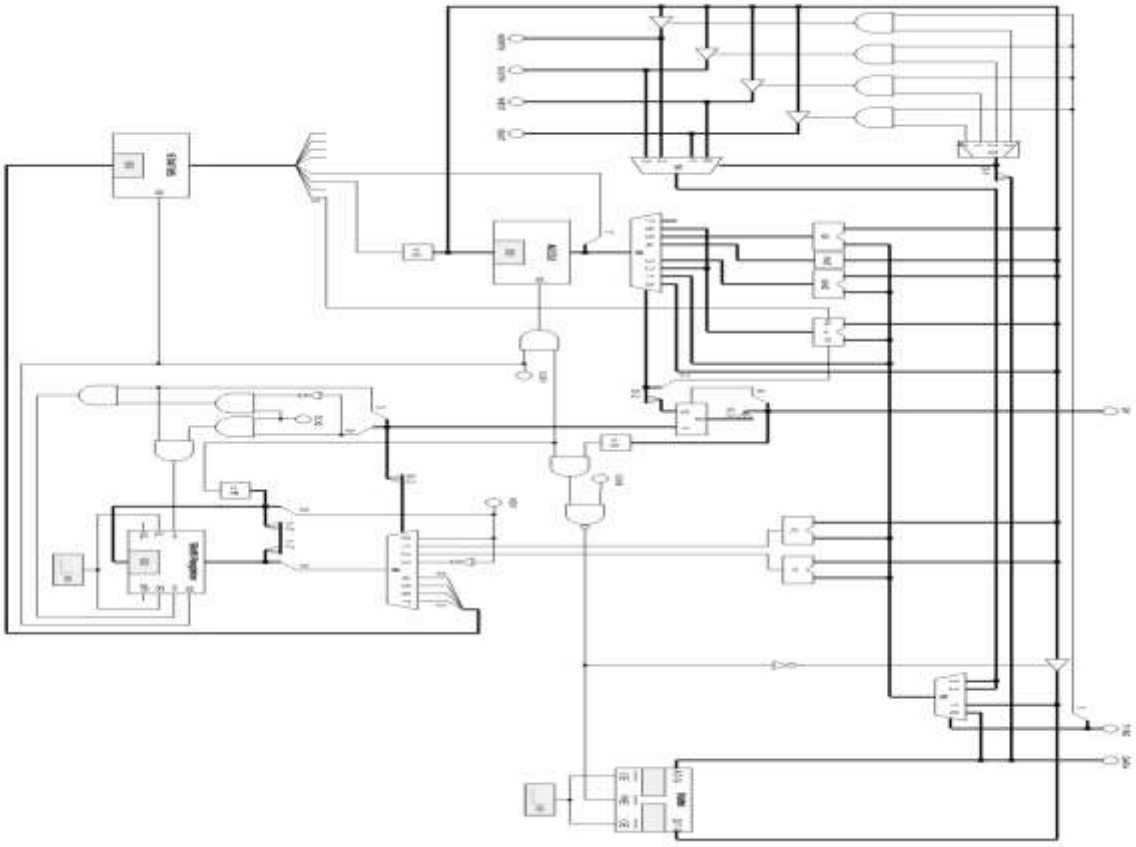


Fig 5.11 PE Internal Circuit

Fig 5.12 PE Internal Circuit

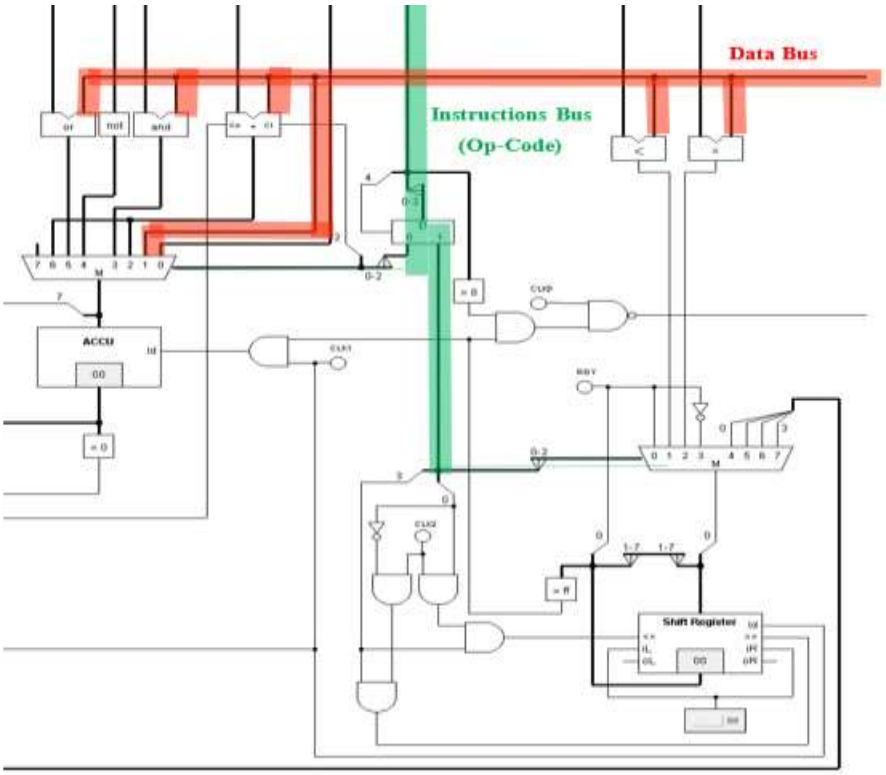


Fig 5.13 ALU

The ALU is the heart of a PE and it is made up of two multiplexers, an accumulator and a shift register. The select lines of the both multiplexers are connected to the instruction bus

Data coming from both external world and accumulator passes through different function units, which then feeds into the multiplexer. It has six most basic function units:

- **Full Adder**
- **AND**
- **NOT**
- **OR**
- **Less Than**
- **Equal**

Unlike a general-purpose microprocessor, this PE does not have some complicated function units such as multiplications. There are three major reasons behind this. First of all, these complicated function units take up a large amount of transistors, which results in less PEs fitting on a single silicon chip. Additionally, this microprocessor is built for image processing, and it does not require these complicated function units. Finally, a lot of functions can be achieved by combining these basic functions. Here are some examples.

INPUT/OUPUT-DATA BUS AND INTERCONNECTION

The I/O control is important for data shifting. Every PE has two way communications with its four surrounding neighbours. In the horizontal direction, a PE exchanges data with its neighbour using the data bus. In the vertical direction, PE transfers data between its neighbours using the interconnection bus. The data flow direction of each bus is controlled by the combined action of tri-state gates, a decoder, AND gates and a multiplexer. In SIMD, a data shifting command causes all PEs to shift data into their neighbours. For example, if we want to shift data to the right (east), then all PEs enable their "east" tri-state gate. Data can flow out of PEs via "east" bus to their east

neighbours. At the same time, every PE's multiplexer is switched to the "west" bus, which can now accept data coming from their west neighbours.

Instruction Bus

Fig 5.14- Instruction Bus

All PEs are connected to the same instruction bus in parallel. PEs do not hold any instructions internally. They receive instructions from the external world via the instructions bus. As shown in figure 3.8, there are three pins connected to the Sequencer CPU. The "OP" pin is used for receiving Op-Code. The "TYPE" pin gets a 2-bit value, which indicates the type of the data and where the data is from – either from its neighbour or from Sequencer CPU. And finally the "DATA" pin accepts the data from the Sequencer CPU.

Internal Memory

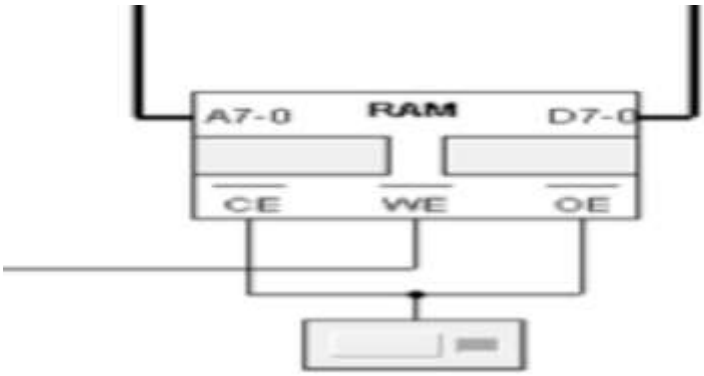


Fig 5.15 PE Internal Memory

Each PE has a small memory space for temporary data storage. This memory can be used to store intermediate results, address pointers, as well as various data structures such as queue and stack.

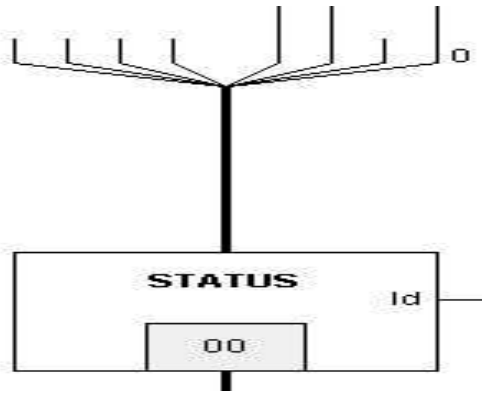


Fig 5.16 PE Status Register

Status Register

Status Register is another important component of a PE. This is an 8-bit status register, which can hold up to eight status flags. The definition of each status bit is stated in Table
Currently there are only 3 flags are used – Negative, Zero, and Carry bit.

Bit	7	6	5	4	3	2	1	0
Flag	-	-	-	-	N	Z	V	C

C	Carry bit
V	Overflow (Currently not implemented)
Z	Zero
N	Negative

Table 5.6 PE – Flags

Having a dedicated status register for each PE is very important in SIMD design. A PE can now record and determine its state by setting and checking its status flags.

Image Processing

Image processing is an important topic in computer science with a large number of applications. Applications such as contrast enhancement, lightening, image blurring, corner detection and edge detection are the most basic image processing applications. These applications are the building blocks of other advanced applications, such as artificial intelligence, 3D object reconstructions, and object recognitions.

DIGITAL IMAGE FUNDAMENTALS:

The field of digital image processing refers to processing digital images by means of digital computer. Digital image is composed of a finite number of elements, each of which has a particular location and value. These elements are called picture elements, image elements, pels and pixels. Pixel is the term used most widely to denote the elements of digital image.

An image is a two-dimensional function that represents a measure of some characteristic such as brightness or color of a viewed scene. An image is a projection of a 3-D scene into a 2D projection plane.

An image may be defined as a two-dimensional function *f(x,y)*, where x and y are spatial (plane) coordinates, and the amplitude of *f* at any pair of coordinates (x,y) is called the intensity of the image at that point.

Fig 5.17 Digital Image processing

The term *gray level* is used often to refer to the intensity of monochrome images.

Color images are formed by a combination of individual 2-D images. For example: The RGB color

system, a color image consists of three (red, green and blue) individual component images. For this reason many of the techniques developed for monochrome images can be extended to color images by processing the three component images individually. An image may be continuous with respect to the x- and y- coordinates and also in amplitude. Converting such an image to digital form requires that the coordinates, as well as the amplitude, be digitized.

APPLICATIONS OF DIGITAL IMAGE PROCESSING

Since digital image processing has very wide applications and almost all of the technical fields are impacted by DIP, we will just discuss some of the major applications of DIP.

Digital image processing has a broad spectrum of applications, such as

- **Remote sensing via satellites and other spacecrafts**
- **Image transmission and storage for business applications**
- **Medical processing,**
- **RADAR (Radio Detection and Ranging)**
- **SONAR(Sound Navigation and Ranging) and**
- **Acoustic image processing (The study of underwater sound is known as underwater acoustics or hydro acoustics.)**
- **Robotics and automated inspection of industrial parts. Images acquired by satellites are useful in tracking of**
    - **Earth resources;**
    - **Geographical mapping;**
    - **Prediction of agricultural crops,**
    - **Urban growth and weather monitoring**
    - **Flood and fire control and many other environmental applications.**

Space image applications include:

- **Recognition and analysis of objects contained in images obtained from deep space-probe missions.**
- **Image transmission and storage applications occur in broadcast television**
- **Teleconferencing**
- **Transmission of facsimile images(Printed documents and graphics) for office automation**

Communication over computer networks

- **Closed-circuit television based security monitoring systems and**
- **In military Communications.**

Medical applications:

- **Processing of chest X- rays**
- **Cineangiograms**
- **Projection images of transaxial tomography and**
- **Medical images that occur in radiology nuclear magnetic resonance(NMR)**
- **Ultrasonic scanning**

Components of Image processing System:

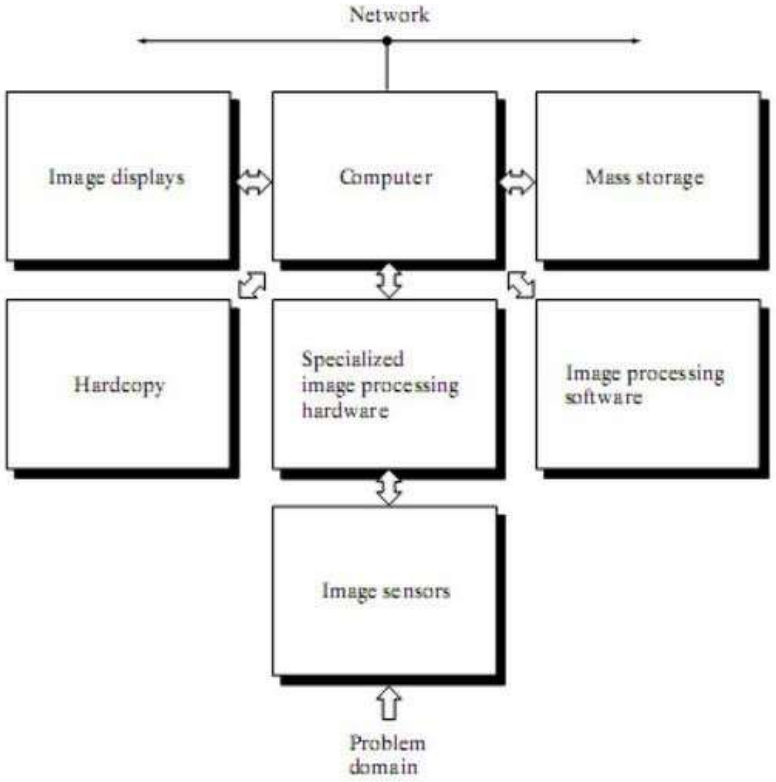


Figure 5.18 : Components of Image processing System

Image Sensors:

With reference to sensing, two elements are required to acquire digital image. The first is a physical device that is sensitive to the energy radiated by the object we wish to image and second is specialized image processing hardware. Specialize image processing hardware: It consists of the digitizer just mentioned, plus hardware that performs other primitive operations such as an arithmetic logic unit, which performs arithmetic such addition and subtraction and logical operations in parallel on images.

Computer:

It is a general purpose computer and can range from a PC to a supercomputer depending on the application. In dedicated applications, sometimes specially designed computer are used to achieve a required level of performance

Software:

It consists of specialized modules that perform specific tasks a well designed package also includes capability for the user to write code, as a minimum, utilizes the specialized module. More sophisticated software packages allow the integration of these modules.

Mass storage:

This capability is a must in image processing applications. An image of size 1024 x1024 pixels, in which the intensity of each pixel is an 8- bit quantity requires one Megabytes of storage space if the image is not compressed .Image processing applications falls into three principal categories of storage

i) Short term storage for use during processing

ii) On line storage for relatively fast retrieval

iii) Archival storage such as magnetic tapes and disks

Image display:

Image displays in use today are mainly color TV monitors. These monitors are driven by the outputs of image and graphics displays cards that are an integral part of computer system.

Hardcopy devices: The devices for recording image includes laser printers, film cameras, heat sensitive devices inkjet units and digital units such as optical and CD ROM disk. Films provide the highest possible resolution, but paper is the obvious medium of choice for written applications.

Networking:

It is almost a default function in any computer system in use today because of the large amount of data inherent in image processing applications. The key consideration in image transmission bandwidth.

Fundamental Steps in Digital Image Processing:

There are two categories of the steps involved in the image processing –

1. **Methods whose outputs are input are images.**
2. **Methods whose outputs are attributes extracted from those images.**

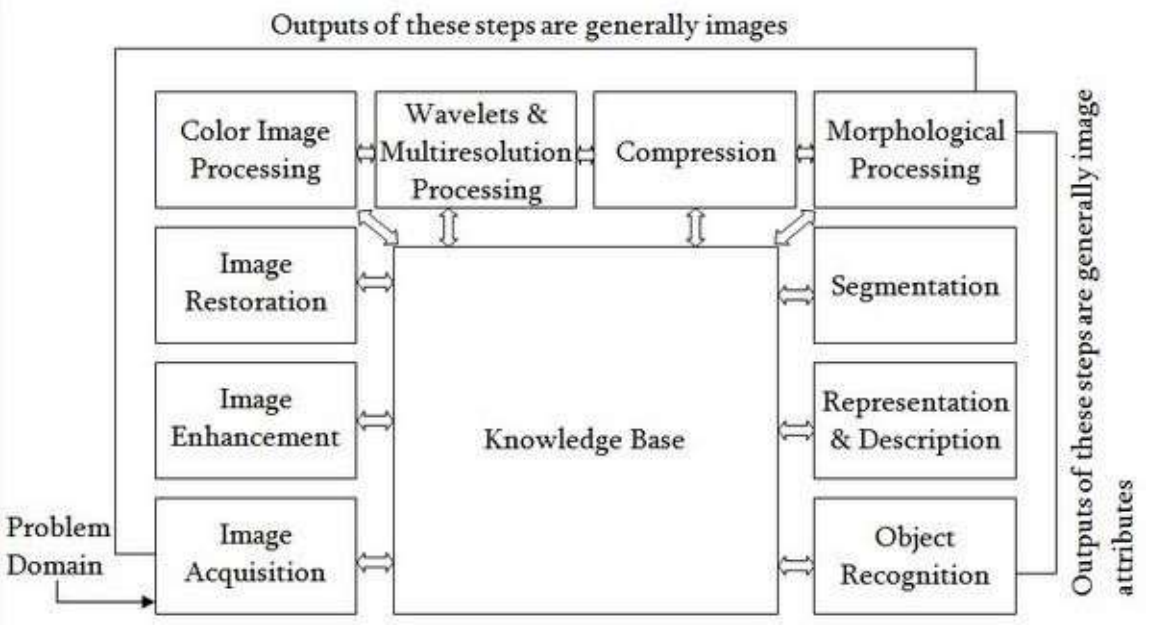


Fig 5.19 Fundamental Steps in Digital Image Processing

Image acquisition:

It could be as simple as being given an image that is already in digital form. Generally the image acquisition stage involves processing such scaling.

Image Enhancement:

It is among the simplest and most appealing areas of digital image processing. The idea behind this is to bring out details that are obscured or simply to highlight certain features of interest in image. Image enhancement is a very subjective area of image processing.

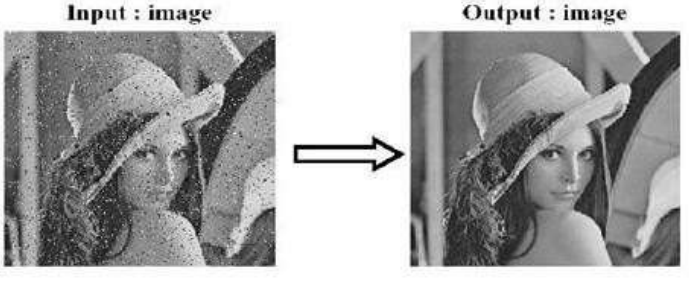


Fig 5.20 Image enhancement

Image Restoration:

It deals with improving the appearance of an image. It is an objective approach, in the sense that restoration techniques tend to be based on mathematical or probabilistic models of image processing. Enhancement, on the other hand is based on human subjective preferences regarding what constitutes a "good" enhancement result.

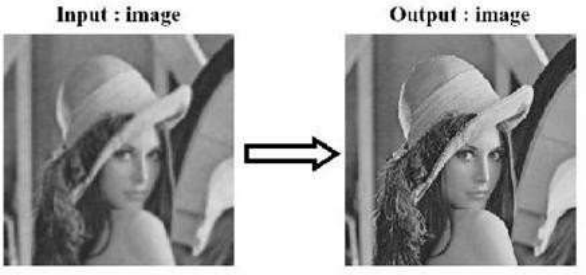


Fig 5.21 Image restoration

Color image processing:

It is an area that is been gaining importance because of the use of digital images over the internet. Color image processing deals with basically color models and their implementation in image processing applications.

Wavelets and Multiresolution Processing:

These are the foundation for representing image in various degrees of resolution.

Compression:

It deals with techniques reducing the storage required to save an image, or the bandwidth required to transmit it over the network. It has to major approaches

a) Lossless Compression b) Lossy Compression

Morphological processing:

It deals with tools for extracting image components that are useful in the representation and description of shape and boundary of objects. It is majorly used in automated inspection applications.

Representation and Description:

It always follows the output of segmentation step that is, raw pixel data, constituting either the boundary of an image or points in the region itself. In either case converting the data to a form suitable for computer processing is necessary.

Recognition:

It is the process that assigns label to an object based on its descriptors. It is the last step of image processing which use artificial intelligence of software.

Knowledge base:

Knowledge about a problem domain is coded into an image processing system in the form of a knowledge base. This knowledge may be as simple as detailing regions of an image where the information of the interest in known to be located. Thus limiting search that has to be conducted in seeking the information. The knowledge base also can be quite complex

such interrelated list of all major possible defects in a materials inspection problem or an image database containing high resolution satellite images of a region in connection with change detection application.

There are three types of computerized processes in the processing of image

1) Low level process -these involve primitive operations such as image processing to reduce noise, contrast enhancement and image sharpening. These kind of processes are characterized by fact the both inputs and output are images.

2) Mid level image processing - it involves tasks like segmentation, description of those objects to reduce them to a form suitable for computer processing, and classification of individual objects. The inputs to the process are generally images but outputs are attributes extracted from images.

3) High level processing – It involves "making sense" of an ensemble of recognized objects, as in image analysis, and performing the cognitive functions normally associated with vision.

Digital Image representation:

Digital image is a finite collection of discrete samples (pixels) of any observable object. The pixels represent a two- or higher dimensional "view" of the object, each pixel having its own discrete value in a finite range. The pixel values may represent the amount of visible light, infra red light, absortation of x-rays, electrons, or any other measurable value such as ultrasound wave impulses. The image does not need to have any visual sense; it is sufficient that the samples form a two-dimensional spatial structure that may be illustrated as an image. The images may be obtained by a digital camera, scanner, electron microscope, ultrasound stethoscope, or any other optical or non-optical sensor. Examples of digital image are:

- **digital photographs**
- **satellite images**
- **radiological images (x-rays, mammograms)**
- **binary images, fax images, engineering drawings**

Computer graphics, CAD drawings, and vector graphics in general are not considered in this course even though their reproduction is a possible source of an image. In fact, one goal

of intermediate level image processing may be to reconstruct a model (e.g. vector representation) for a given digital image.

FULL COLOR IMAGE PROCESSING

Full-color image processing techniques falls into two major categories • Processing each color channel individually and compile the results all gray level techniques are immediately available • Processing the entire color image directly pixels are viewed as vectors and scalar methods are (if possible) extended to operate on vector fields Two criteria must be met for extending a scalar method to vector space • The process has to be applicable to vectors • The operation on each component of a vector has to be independent of the other components

Color transformations Extension of the gray level transformations to color space In theory any transformation can be done in any color space In practice some transformations are better suited for specific color spaces The cost of color space transformations must be considered

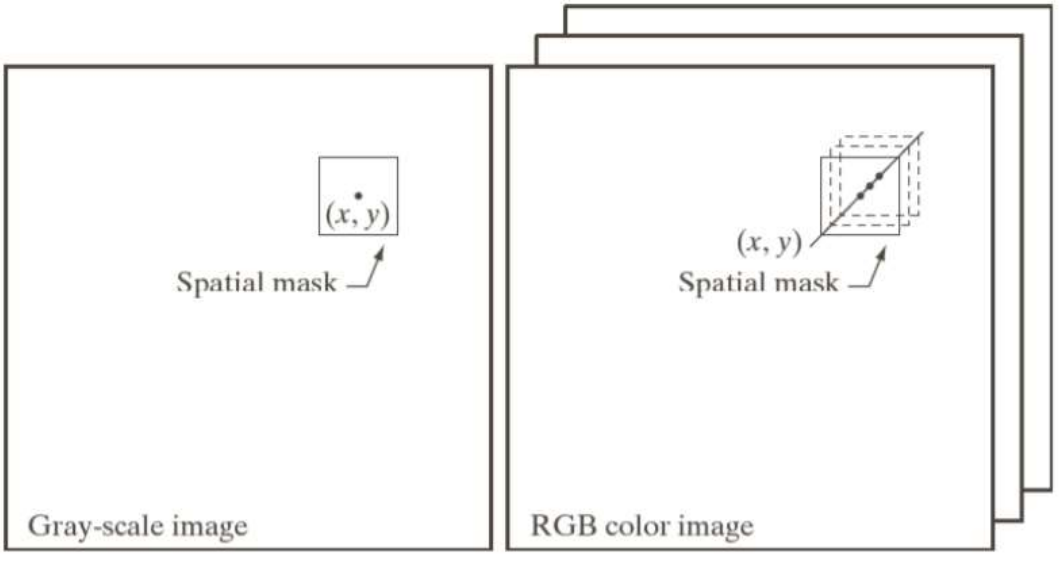


Fig 5.22 Spatial mask for gray scale and RGB images

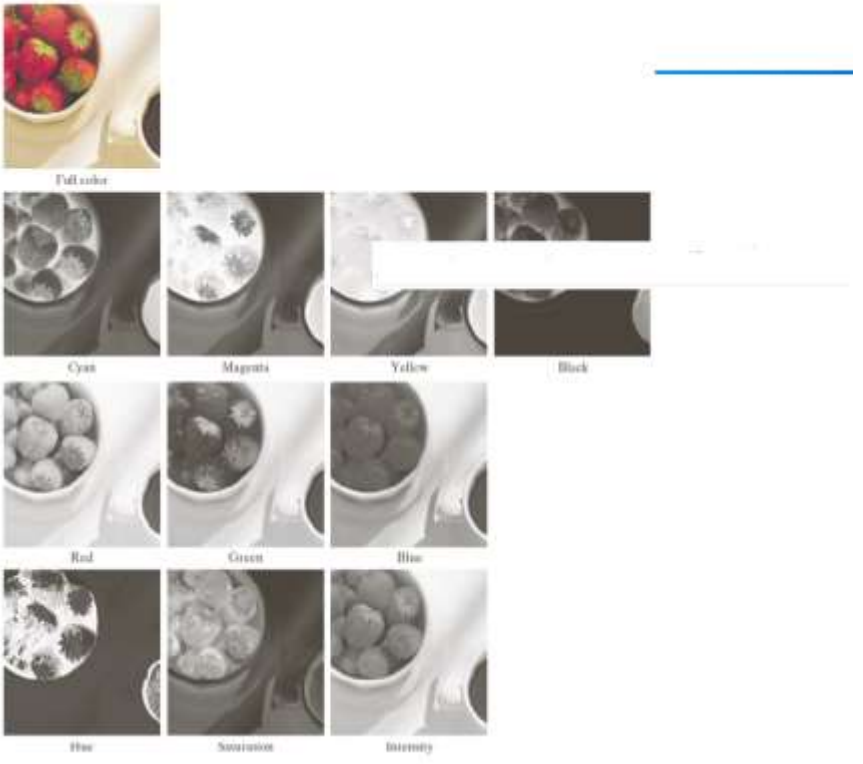


Fig 5.23 Full color and its various color spaces

Fig 5.24 a. RGB image with green plane corrupted by salt and pepper noise, b. Hue component of HSI (Hue, saturation, intensity) image, c. Saturation component d. Intensity component

SMOOTHING

Smoothing is often used to reduce noise within an image. • Image smoothing is a key technology of image enhancement, which can remove noise in images. So, it is a necessary functional module in various image-processing software. • Image smoothing is a method of improving the quality of images. • Smoothing is performed by spatial and frequency filters

Image smoothing techniques have the goal of preserving image quality. In other words, to

remove noise without losing the principal features of the image. However, there are several types of noise. The main three types are: impulsive, additive, and multiplicative. Impulsive noise is usually characterized by some portion of image pixels that are corrupted, leaving the others unchanged. Additive noise appears when the values of the original image have been modified by adding random values which follow a certain probability distribution. Finally, multiplicative noise is more difficult to be removed from images than additive noise, because in this case intensities vary along with signal intensity (e.g., speckle noise).

There are different sources of noise and plenty of denoising methods for each kind of noise. The most common one is probably the so-called thermal noise. This impulsive noise is due to CCD sensor malfunction in the image acquisition process.

Another interesting case is Gaussian noise, in which each pixel of the image will be changed from its original value by some small amount that follows a Gaussian distribution. This kind of noise is modelled as an additive white Gaussian noise. So that, its presence can be simulated by adding random values from a zero-mean Gaussian distribution to the original pixel intensities in each image channel independently, where the standard deviation σ of the Gaussian distribution characterizes the noise intensity .

The elimination of this type of noise is known as smoothing, and this will be the type of noise elimination considered in this work. There are plenty of nonlinear methods for smoothing.

Fig 5.25 Image smoothing

SHARPENING

Before getting into the act of sharpening an image, we need to consider what sharpness actually is. The biggest problem is that, in large part, sharpness is subjective. Sharpness is a combination of two factors: *resolution* and *acutance*. Resolution is straightforward and not subjective. It's just the size, in pixels, of the image file. All other factors equal, the higher the resolution of the image—the more pixels it has—the sharper it can be. Acutance is a little more complicated. It's a subjective measure of the contrast at an edge. There's no unit for acutance—you either think an edge has contrast or think it doesn't. Edges that have more contrast appear to have a more defined edge to the human visual system.

Fig 5.26 Identify the image acutance

Both Ts **have the same resolution but different acutance. Sharpness comes down to how defined the details in an image are—especially the small details. For example, if a subject's eyelashes are an indistinct black blur they won't appear sharp. If, on the other hand, you can pick out each one then most people will consider the image sharp.**

Sharpening then, is a technique for increasing the *apparent* sharpness of an image. Once an image is captured, Photoshop can't magically any more details: the actual resolution remains fixed. Yes, you can increase the file's size but the algorithms any image editor uses to do so will decrease the sharpness of the details.

In other words, the only way to increase apparent sharpness is by increasing acutance. If you want your image to look sharper, you need to add edge contrast.

Fig 5.27 Image sharpening

Here mimicked the Unsharp Mask **effect in Photoshop. The first image is the original file, the second is a blurred copy, the third is the one subtracted from the other so as to detect the edges, and the fourth is the original image sharpened using the edge layer. Unsharp masking is the oldest sharpening technique. It subtracts a blurred (unsharp) copy from the original image to detect any edges. A mask is made with this edge detail. Contrast is then increased at the edges and the effect is applied to the original image. While unsharp masking was originally a film technique, it's now the basis of digital sharpening.**

SEGMENTATION

In digital image processing and computer vision, image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as image objects). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.

The result of image segmentation is a set of segments that collectively cover the entire image, or a set of contours extracted from the image (see edge detection). Each of the pixels in a region are similar with respect to some characteristic or computed property, such as color, intensity, or texture. Adjacent regions are significantly different with respect to the same characteristic(s). When applied to a stack of images, typical in medical imaging, the resulting contours after image segmentation can be used to create 3D reconstructions with the help of interpolation algorithms like marching cubes.

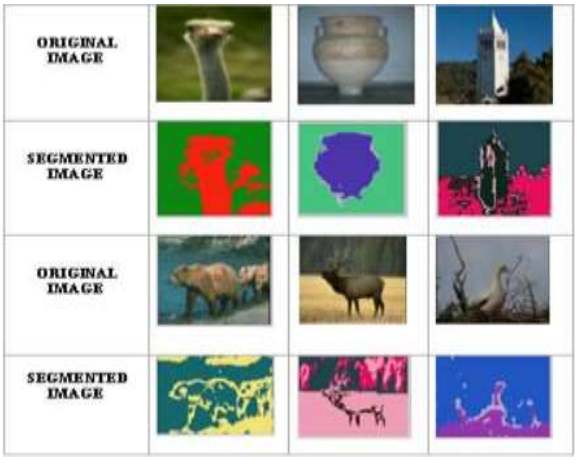


Fig 5.28 Segmented image

MOTION ANALYSIS

Motion analysis is used in computer vision, image processing, high-speed photography and machine vision that studies methods and applications in which two or more consecutive images from an image sequences, e.g., produced by a video camera or high-speed camera, are processed to produce information based on the apparent motion in the images. In some applications, the camera is fixed relative to the scene and objects are moving around in the scene, in some applications the scene is more or less fixed and the camera is moving, and in some cases both the camera and the scene are moving.

The motion analysis processing can in the simplest case be to detect motion, i.e., find the points in the image where something is moving. More complex types of processing can be to track a specific object in the image over time, to group points that belong to the same rigid object that is moving in the scene, or to determine the magnitude and direction of the motion of every point in the image. The information that is produced is often related to a specific image in the sequence, corresponding to a specific time-point, but then depends also on the neighboring images. This means that motion analysis can produce time-dependent information about motion.

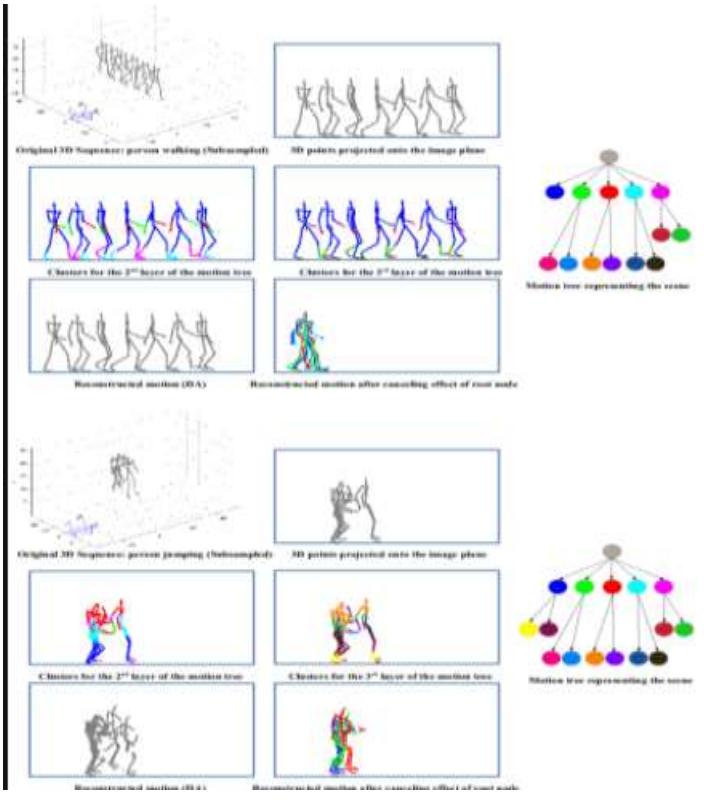


Fig 5.29 motion analysis

IMAGE FUSION

The image fusion process is defined as gathering all the important information from multiple images, and their inclusion into fewer images, usually a single one. This single image is more informative and accurate than any single source image, and it consists of all the necessary information. The purpose of image fusion is not only to reduce the amount of data but also to construct images that are more appropriate and understandable for the human and machine perception. In computer vision, multisensor image fusion is the process of combining relevant information from two or more images into a single image. The resulting image will be more informative than any of the input images.

In remote sensing applications, the increasing availability of space borne sensors gives a motivation for different image fusion algorithms. Several situations in image processing require high spatial and high spectral resolution in a single image. Most of the available equipment is not capable of providing such data convincingly. Image fusion techniques allow the integration of different information sources. The fused image can have complementary spatial and spectral resolution characteristics. However, the standard image fusion techniques can distort the spectral information of the multispectral data while merging.

In satellite imaging, two types of images are available. The panchromatic image acquired by satellites is transmitted with the maximum resolution available and the multispectral data are transmitted with coarser resolution. This will usually be two or four times lower. At the receiver station, the panchromatic image is merged with the multispectral data to convey more information.

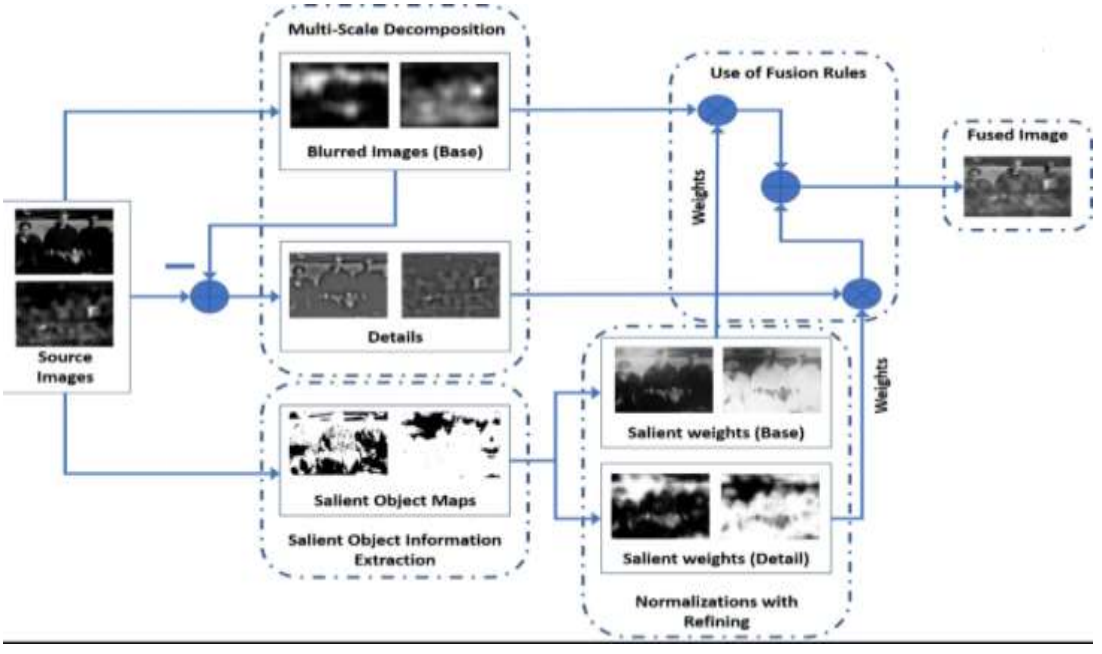


Fig 5.30 Image fusion

IMAGE CLASSIFICATION

Image classification refers to the task of extracting information classes from a multiband raster image. The resulting raster from image classification can be used to create thematic maps. Depending on the interaction between the analyst and the computer during classification, there are two types of classification: supervised and unsupervised.The classification process is a multi-step workflow, therefore, the Image Classification toolbar has been developed to provided an integrated environment to perform classifications with the tools. Not only does the toolbar help with the workflow for performing unsupervised and supervised classification, it also contains additional functionality for analyzing input data, creating training samples and signature files, and determining the quality of the training samples and signature files. The recommended way to perform classification and multivariate analysis is through the Image Classification toolbar.

Supervised classification

Supervised classification uses the spectral signatures obtained from training samples to classify an image. With the assistance of the Image Classification toolbar, you can easily create training samples to represent the classes you want to extract. You can also easily create a signature file from the training samples, which is then used by the multivariate classification tools to classify the image.

Unsupervised classification

Unsupervised classification finds spectral classes (or clusters) in a multiband image without the analyst's intervention. The Image Classification toolbar aids in unsupervised classification by providing access to the tools to create the clusters, capability to analyze the quality of the clusters, and access to classification tools.

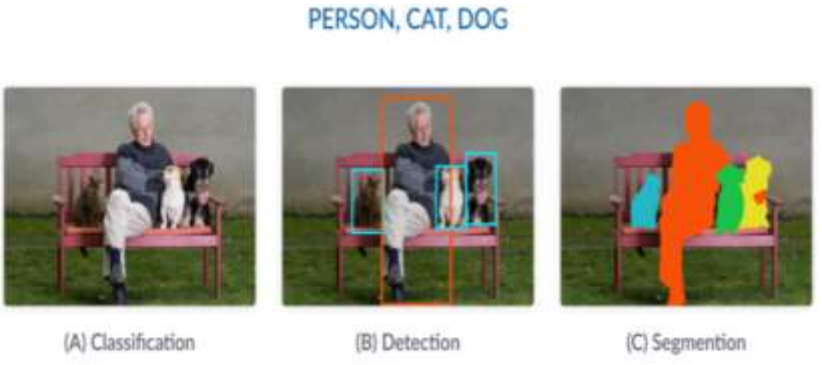


Fig 5.31 Image classification, detection and segmentation

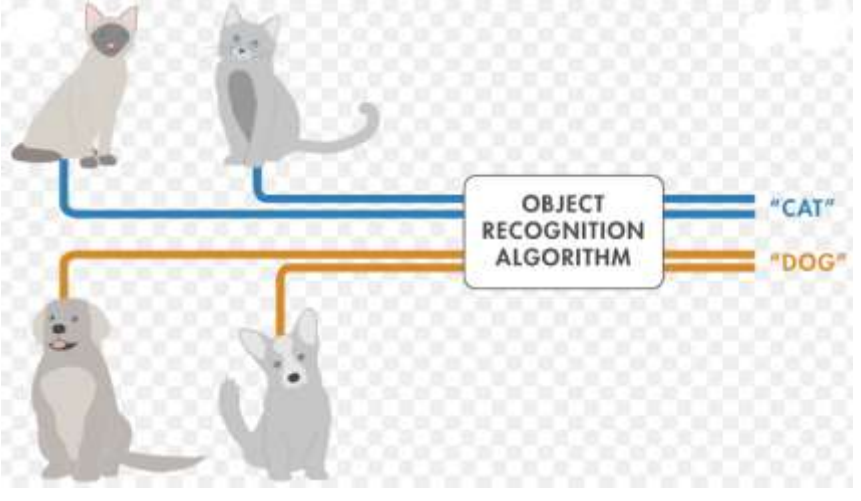


Fig 5.32 Object recognition

Color transformations

Background

- **Humans can perceive thousands of colors, and only about a couple of dozen gray shades (cones/rods)**

- **Divide into two major areas: full color and pseudo color processing**
- **Full color – Image is acquired with a full-color sensor like TV camera or color scanner**
- **Pseudo color – Assign a color to a range of monochrome intensities**
- **The availability of inexpensive and powerful hardware has resulted in the proliferation of applications based on full color processing**
- **8-bit color vs 24-bit color**
- **Color quantization**

Some of the gray scale image processing methods are directly applicable to color processing but others will need reformulation

Color characterized by three quantities

Hue: Dominant color as perceived by an observer (red, orange, or yellow)
Saturation Relative purity of color; pure spectrum colors are fully saturated
- **Saturation is inversely proportional to the amount of white light added**

Brightness Achromatic notion of intensity

- **Chromaticity**
  - **Combination of hue and saturation**
  - **Allows a color to be expressed as its brightness and chromaticity**
- **Tristimulus values**
  - **Three types of cones in the eye require three components for each color, using appropriate spectral weighting functions**
  - **Based on standard curves/functions defined by CIE – Commission Internationale de L'E´ clairage**
  - **Curves specify the transformation of spectral power distribution for each color into three numbers**
  - **Amount of red, green, and blue to express a color**
  - **Denoted by *X*, *Y* , and *Z***
  - **Color specified by its tristimulus coefficients**

Luminance Measure of amount of energy as perceived by an observer

Applications of Digital Image Processing
Some of the major fields in which digital image processing is widely used are mentioned below

- **Image sharpening and restoration**
- **Medical field**
- **Remote sensing**
- **Transmission and encoding**
- **Machine/Robot vision**
- **Color processing**
- **Pattern recognition**
- **Video processing**
- **Microscopic Imaging**
- **Others**

Image sharpening and restoration
Image sharpening and restoration refers here to process images that have been captured from the modern camera to make them a better image or to manipulate those images in way to achieve desired result. It refers to do what Photoshop usually does.
This includes Zooming, blurring , sharpening , gray scale to color conversion, detecting edges

and vice versa , Image retrieval and Image recognition. The common examples are:

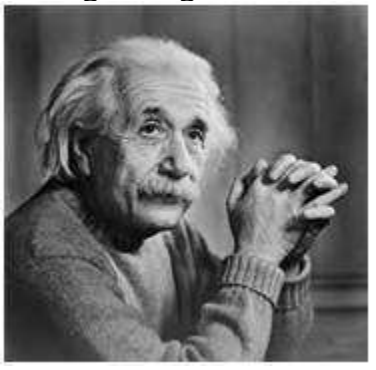


Fig 5.33 The original image

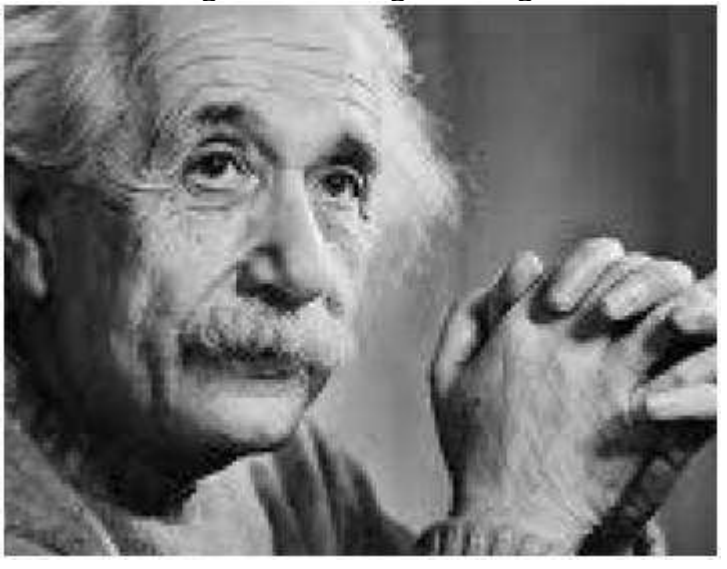


Fig 5.34 The zoomed image

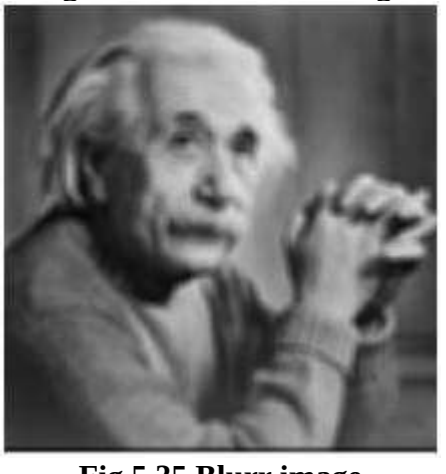


Fig 5.35 Blurr image

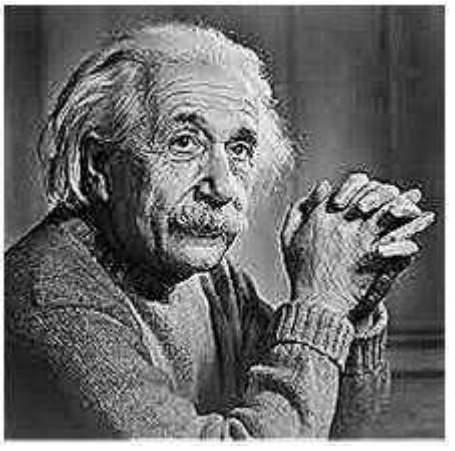


Fig 5.36 Sharp image

Fig 5.37 Edges

Medical field

The common applications of DIP in the field of medical is

- **Gamma ray imaging**
- **PET scan**
- **X Ray Imaging**
- **Medical CT**
- **UV imaging**

UV imaging

In the field of remote sensing , the area of the earth is scanned by a satellite or from a very high ground and then it is analyzed to obtain information about it. One particular application of digital image processing in the field of remote sensing is to detect infrastructure damages caused by an earthquake.

As it takes longer time to grasp damage, even if serious damages are focused on. Since the area effected by the earthquake is sometimes so wide , that it not possible to examine it with human eye in order to estimate damages. Even if it is , then it is very hectic and time consuming procedure. So a solution to this is found in digital image processing. An image of the effected area is captured from the above ground and then it is analyzed to detect the various types of damage done by the earthquake.

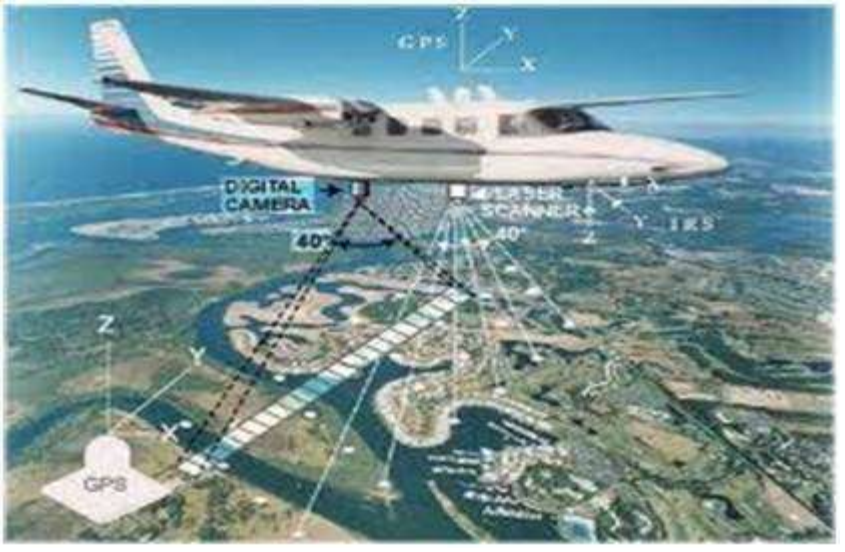


Fig 5.38 Earth quake damage image capture

The key steps include in the analysis are

- **The extraction of edges**
- **Analysis and enhancement of various types of edges**

Transmission and encoding

The very first image that has been transmitted over the wire was from London to New York via a submarine cable. The picture that was sent is shown below.

Fig 5.39 First image

The picture that was sent took three hours to reach from one place to another.

Now just imagine , that today we are able to see live video feed , or live cctv footage from one continent to another with just a delay of seconds. It means that a lot of work has been done in this field too. This field doesnot only focus on transmission , but also on encoding. Many different formats have been developed for high or low bandwith to encode photos and then stream it over the internet or e.t.c.

Machine/Robot vision

Apart form the many challenges that a robot face today , one of the biggest challenge still is to increase the vision of the robot. Make robot able to see things , identify them , identify the hurdles e.t.c. Much work has been contributed by this field and a complete other field of computer vision has been introduced to work on it.

Hurdle detection

Hurdle detection is one of the common task that has been done through image processing, by identifying different type of objects in the image and then calculating the distance between robot and hurdles.

Fig 5.40 Hurdle detection

Line follower robot

Most of the robots today work by following the line and thus are called line follower robots. This help a robot to move on its path and perform some tasks. This has also been achieved through image processing.

Fig 5.41 Line follower robot

Color processing

Color processing includes processing of colored images and different color spaces that are used. For example RGB color model , YCbCr, HSV. It also involves studying transmission , storage , and encoding of these color images.

Pattern recognition

Pattern recognition involves study from image processing and from various other fields that includes machine learning (a branch of artificial intelligence). In pattern recognition , image processing is used for identifying the objects in an images and then machine learning is used to train the system for the change in pattern. Pattern recognition is used in computer aided diagnosis , recognition of handwriting , recognition of images e.t.c

Video processing

A video is nothing but just the very fast movement of pictures. The quality of the video depends on the number of frames/pictures per minute and the quality of each frame being used. Video processing involves noise reduction , detail enhancement , motion detection , frame rate conversion , aspect ratio conversion , color space conversion e.t.c.

TEXT / REFERENCE BOOKS

1. **Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing", 2nd Edition,Pearson Education, Inc., 2004**
2. **Barry B.Brey , "The Intel Microprocessors 8086/8088, 8086, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, Architecture, Programming and interfacing", Prentice Hall of India Private Limited, New Delhi, 2003**
3. **Alan Clements, "The Principles of computer Hardware", Oxford University Press, 3rd Edition, 2003**
4. **John Paul Shen, Mikko H.Lipasti, "Modern Processor Design", Tata McGraw Hill, 2006**