Advanced Computer Architecture

MCA - Project

CS-12

Q.1. Describe the architecture of Pentium Microprocessor. Following points must be discussed:

- Block diagram of the computer
- Register set
- Addressing mode
- Types of instructions
- Instruction formats
- Memory system
- Memory-processor interfacing schemes
- Input-Output techniques that can be used
- System software available
- Application software tools

Sol:

A Brief History of the Pentium Processor Family

The Pentium family of processors, which has its roots in the Intel486(TM) processor, uses the Intel486 instruction set (with a few additional instructions). The term "Pentium processor" refers to a family of microprocessors that share a common architecture and instruction set. The first Pentium processors (the P5 variety) were introduced in 1993. This 5.0-V processor was fabricated in 0.8-micron bipolar complementary metal oxide semiconductor (BiCMOS) technology. The P5 processor runs at a clock frequency of either 60 or 66 MHz and has 3.1 million transistors.

The next version of the Pentium processor family, the P54C processor, was introduced in 1994. The P54C processors are fabricated in 3.3-V, 0.6-micron BiCMOS technology. The P54C processor also has System Management Mode (SMM) for advanced power management.

The Intel Pentium processor, like its predecessor the Intel486 microprocessor, is fully software compatible with the installed base of over 100 million compatible Intel architecture systems. In addition, the Intel Pentium processor provides new levels of performance to new and existing software through a reimplementation of the Intel 32-bit instruction set architecture using the latest, most advanced, design techniques. Optimized, dual execution units provide one-clock execution for "core" instructions, while advanced technology, such as superscalar architecture, branch prediction, and execution pipelining, enables multiple instructions to execute in parallel with high efficiency. Separate code and data caches combined with wide 128-bit and 256-bit

internal data paths and a 64-bit, burstable, external bus allow these performance levels to be sustained in cost-effective systems. The application of this advanced technology in the Intel Pentium processor brings "state of the art" performance and capability to existing Intel architecture software as well as new and advanced applications.

The Pentium processor has two primary operating modes and a "system management mode". The operating mode determines which instructions and architectural features are accessible. These modes are:

Protected Mode

This is the native state of the microprocessor. In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode that all new applications and operating systems should target. Among the capabilities of protected mode is the ability to directly execute "real-address mode" 8086 software in a protected, multi-tasking environment. This feature is known as Virtual-8086 "mode" (or "V86 mode"). Virtual-8086 "mode" however, is not actually a processor "mode," it is in fact an attribute which can be enabled for any task (with appropriate software) while in protected mode.

Real-Address Mode (also called "real mode")

This mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to break out of this mode). Reset initialization places the processor in real mode where, with a single instruction, it can switch to protected mode.

System Management Mode

The Pentium microprocessor also provides support for System Management Mode (SMM). SMM is a standard architectural feature unique to all new Intel microprocessors, beginning with the Intel386 SL processor, which provides an operating-system and application independent and transparent mechanism to implement system power management and OEM differentiation features. SMM is entered through activation of an external interrupt pin (SMI#), which switches the CPU to a separate address space while saving the entire context of the CPU. SMM-specific code may then be executed transparently. The operation is reversed upon returning.

Advanced Features

The Pentium P54C processor is the product of a marriage between the Pentium processor's architecture and Intel's 0.6-micron, 3.3-V BiCMOS process The Pentium processor achieves higher performance than the fastest Intel486 processor by making use of the following advanced technologies.

- **Superscalar Execution:** The Intel486 processor can execute only one instruction at a time. With superscalar execution, the Pentium processor can sometimes execute two instructions simultaneously.

- **Pipeline Architecture:** Like the Intel486 processor, the Pentium processor executes instructions in five stages. This staging, or pipelining, allows the processor to overlap multiple instructions so that it takes less time to execute two instructions in a row. Because of its superscalar architecture, the Pentium processor has two independent processor pipelines.

- **Branch Target Buffer:** The Pentium processor fetches the branch target instruction before it executes the branch instruction.

- **Dual 8-KB on-Chip Caches**: The Pentium processor has two separate 8-kilobyte (KB) caches on chip--one for instructions and one for data--which allows the Pentium processor to fetch data and instructions from the cache simultaneously.

- **Write-Back Cache:** When data is modified; only the data in the cache is changed. Memory data is changed only when the Pentium processor replaces the modified data in the cache with a different set of data.

- **64-Bit Bus:** With its 64-bit-wide external data bus (in contrast to the Intel486 processor's 32-bit-wide external bus) the Pentium processor can handle up to twice the data load of the Intel486 processor at the same clock frequency.

- **Instruction Optimization:** The Pentium processor has been optimized to run critical instructions in fewer clock cycles than the Intel486 processor.

- **Floating-Point Optimization:** The Pentium processor executes individual instructions faster through execution pipelining, which allows multiple floating-point instructions to be executed at the same time.

- **Pentium Extensions:** The Pentium processor has fewer instruction set extensions than the Intel486 processors. The Pentium processor also has a set of extensions for multiprocessor (MP) operation. This makes a computer with multiple Pentium processors possible.

A Pentium system, with its wide, fast buses, advanced write-back cache/memory subsystem, and powerful processor, will deliver more power for today's software applications, and also optimize the performance of advanced 32-bit operating systems (such as Windows 95) and 32-bit software applications.

Block Diagram of the computer

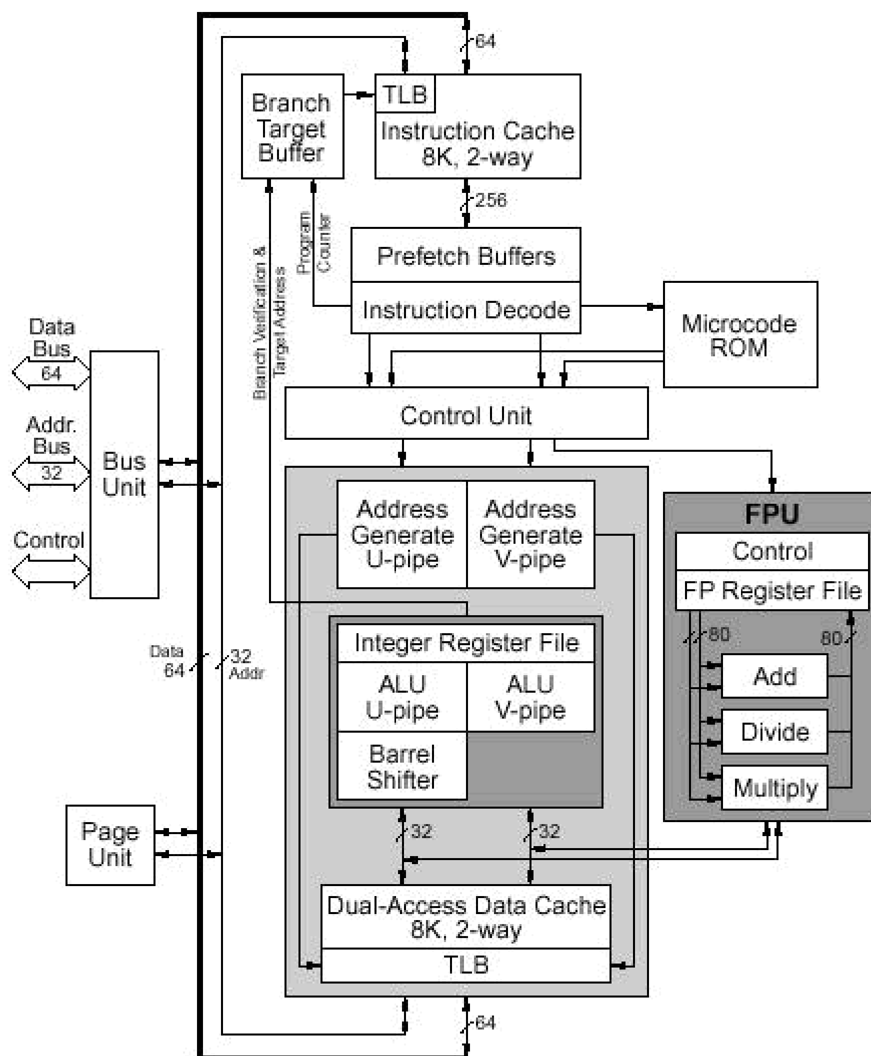

Figure 1. Pentium block diagram.

The most important enhancements over the 486 are the separate instruction and data caches, the dual integer pipelines (the U-pipeline and the V-pipeline, as Intel calls them), branch prediction using the branch target buffer (BTB), the pipelined floating-point unit, and the 64-bit external data bus. Even-parity checking is implemented for the data bus and the internal RAM arrays (caches and TLBs).

As for new functions, there are only a few; nearly all the enhancements in Pentium are included to improve performance, and there are only a handful of new instructions. Pentium is the first high-performance micro-processor to include a system management mode like those found on power-miserly processors for notebooks and other battery-based applications; Intel is holding to its promise to include SMM on all new CPUs. Pentium uses about 3 million transistors on a huge 294 mm 2 (456k mils 2). The caches plus TLBs use only about 30% of the die. At about 17 mm on a side, Pentium is one of the largest microprocessors ever fabricated and probably pushes Intel's production equipment to its limits. The integer data path is in the middle, while the floating-point data path is on the side opposite the data cache. In contrast to other superscalar designs, such as SuperSPARC, Pentium's integer data path is actually bigger than its FP data path. This is an indication of the extra logic associated with complex instruction support. Intel estimates about 30% of the transistors were devoted to compatibility with the x86 architecture. Much of this overhead is probably in the microcode ROM, instruction decode and control unit, and the adders in the two address generators, but there are other effects of the complex instruction set. For example, the higher frequency of memory references in x86 programs compared to RISC code led to the implementation of the dual-ac.

Register set

The purpose of the Register is to hold temporary results, and control the execution of the program. General-purpose registers in Pentium are EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

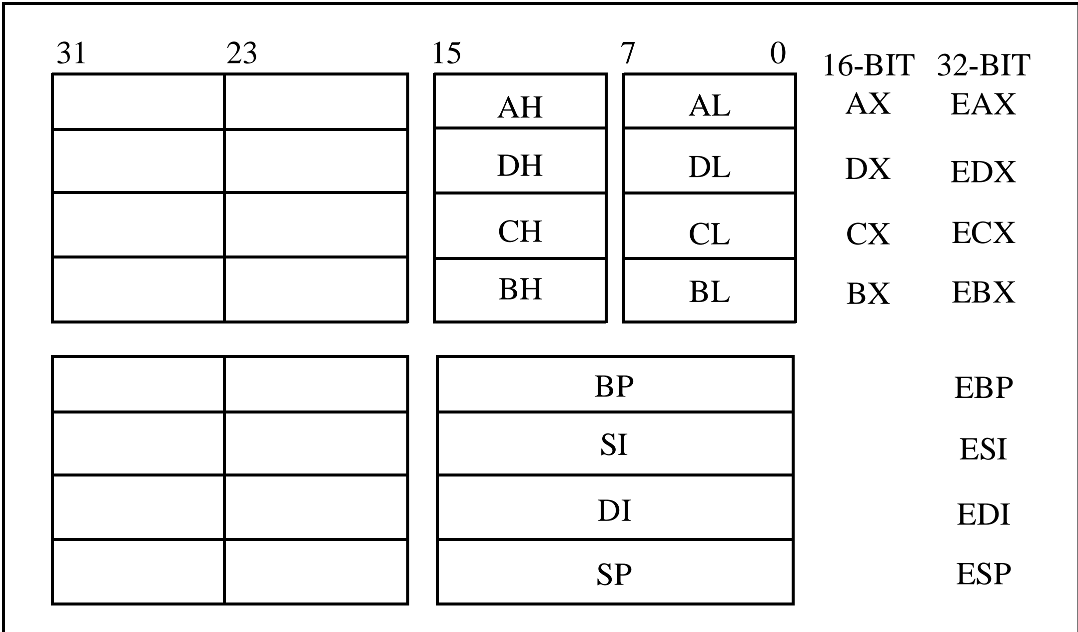

Figure 2. Pentium Register Set

The 32-bit registers are named with prefix E, EAX, etc, and the least 16 bits 0-15 of these registers can be accessed with names such as AX, SI. Similarly the lower eight bits (0-7) can be accessed with names such as AL & BL. The higher eight bits (8-15) can be accessed with names such as AH & BH. The instruction pointer EAP known as program counter (PC) in 8-bit microprocessor, is a 32-bit register to handle 32-bit memory addresses, and the lower 16 bit segment IP is used for 16-bit memory address.

The flag register is a 32-bit register, however 14-bits are being used at present for 13 different tasks; these flags are upward compatible with those of the 8086 and 80286. The comparison of the available flags in 16-bit and 32-bit microprocessor may provide some clues related to capabilities of these processors. The 8086 has 9 flags, the 80286 has 11 flags, and the 80286 has 13 flags. All of these flag registers include 6 flags related to data conditions (sign, zero, carry, auxiliary, carry, overflow, and parity) and three flags related to machine operations (interrupts, Single-step and Strings). The 80286 has two additional: I/O Privilege and Nested Task. The I/O Privilege uses two bits in protected mode to determine which I/O instructions can be used, and the nested task is used to show a link between two tasks.

The processor also includes control registers and system address registers, debug and test registers for system and debugging operations.

Addressing mode & Types of instructions

Instruction set is divided into 9 categories of operations and has 11 addressing modes. In addition to commonly available instructions in a 8 bit microprocessor and this set includes operations such as bit manipulation and string operations, high level language support and operating system support. An instruction may have 0-3 operands and the operand can be 8, 16, or 32- bits long. The 80386 handles various types of data such as Single bit , string of bits , signed and unsigned 8-, 16-, 32- and 64- bit data, ASCII character and BCD numbers.

High level language support group includes instructions such as ENTER and LEAVE. The ENTER instruction is used to ENTER from a high level language and it assigns memory location on the stack for the routine being entered and manages the stack. On the other hand the LEAVE generates a return procedure for a high level language. The operating system support group includes several instructions, such as APRL. (Adjust Requested Privilege Level) and the VERR/W (Verify Segment for Reading or Writing). The APRL is designed to prevent the operating system from gaining access to routines with a higher priority level and the instructions VERR/W verify whether the specified memory address can be reached from the current privilege level.

Instruction formats

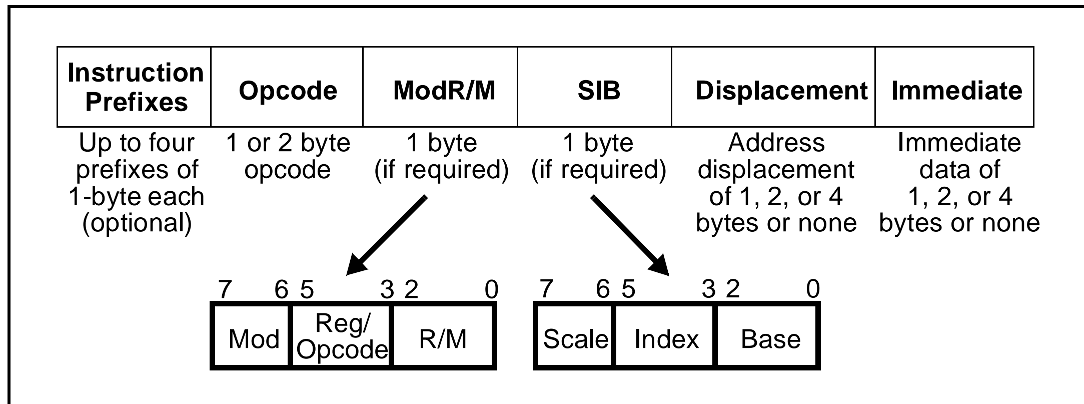

Figure 3. Intel Architecture Instruction Format

General instruction format

All Intel Architecture instruction encoding are subsets of the general instruction format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), one or two primary opcode bytes, an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

Generalities:

- Many (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. I left out the 16-bit ones to reduce presentation of the instruction set. Note that on a 32-bit machine, with newly written code, the 16-bit form will never be used.

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

```
mov   reg, r/m          ; copy data
r/m, reg
reg, immed
r/m, immed
movsx reg, r/m8         ; sign extend and copy data
movzx reg, r/m8         ; zero extend and copy data
lea   reg, m            ; get effective address
```

(A newer instruction, so its format is much restricted over the other ones.)

Examples:

```
mov EAX, 23             ; places 32-bit 2's complement immediate 23
                        ; into register EAX
movsx ECX, AL           ; sign extends the 8-bit quantity in register
                        ; AL to 32 bits, and places it in ECX
mov [esp], -1           ; places value -1 into memory, address given
                        ; by contents of esp
lea EBX, loop_top       ; put the address assigned (by the assembler)
                        ; to label loop_top into register EBX
```

Integer Arithmetic

```
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 double word 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

```
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
```

Floating Point Arithmetic

Since the newer architectures have room for floating point hardware on chip, Intel defined a simple-to-implement extension to the architecture to do floating point arithmetic. In their usual zeal, they have included MANY instructions to do floating point operations.

The mechanism is simple. A set of 8 registers are organized and maintained (by hardware) as a stack of floating point values. ST refers to the stack top. ST(1) refers to the register within the stack that is next to ST. ST and ST(0) are synonyms.

There are separate instructions to test and compare the values of floating point variables.

```
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
ETC.
```

I/O

The only instructions which actually allow the reading and writing of I/O devices are privileged. 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 Instructions

These are the same control instructions that all started with the character 'b' in SASM.

```
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
```

Instruction prefixes

The instruction prefixes are divided into four groups, each with a set of allowable prefix codes:

1. Lock and repeat prefixes.

   - F0H—LOCK prefix.
   - F2H—REPNE/REPNZ prefix (used only with string instructions).
   - F3H—REP prefix (used only with string instructions).
   - F3H—REPE/REPZ prefix (used only with string instructions).
   - F3H—Streaming SIMD Extensions prefix.

2. Segment override.

   - 2 EH—CS segment override prefix.
   - 36H—SS segment override prefix.
   - 3EH—DS segment override prefix.
   - 26H—ES segment override prefix.
   - 64H—FS segment override prefix.
   - 65H—GS segment override prefix.

3. Operand-size override, 66H

4. Address-size override, 67H
   For each instruction, one prefix may be used from each of these groups and be placed in any order. The effect of redundant prefixes (more than one prefix from a group) is undefined and may vary from processor to processor.

5. Streaming SIMD Extensions prefix, 0FH
   The nature of Streaming SIMD Extensions allows the use of existing instruction formats. Instructions use the ModR/M format and are preceded by the 0F prefix byte. In general, operations are not duplicated to provide two directions (i.e. separate load and store variants).

OPCODE

The primary OPCODE is either 1 or 2 bytes. An additional 3-bit OPCODE field is sometimes encoded in the ModR/M byte. Smaller encoding fields can be defined within the primary OPCODE. These fields define the direction of the operation, the size of displacements, the register encoding, condition codes, or sign extension. The encoding of fields in the OPCODE varies, depending on the class of operation.

ModR/M and SIB bytes

Most instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary OPCODE. The ModR/M byte contains three fields of information:

- The *mod* field combines with the r/m field to form 32 possible values; eight registers and 24 addressing modes.

- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the reg/opcode field is specified in the primary opcode.

- The *r/m* field can specify a register as an operand or can be combined with the mod field to encode an addressing mode.

Certain encoding of the ModR/M byte requires a second addressing byte, the SIB byte, to fully specify the addressing form. The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

Displacement and immediate bytes

Some addressing forms include a displacement immediately following either the ModR/M or SIB byte. If a displacement is required, it can be 1, 2, or 4 bytes. If the instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2, or 4 bytes.

CMC—Complement Carry Flag

OPCODE Column

The “OPCODE” column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **/digit**—A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

- **/r**—Indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.

- **cb, cw, cd, cp**—A 1-byte (cb), 2-byte (cw), 4-byte (cd), or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

- **ib, iw, id**—A 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and double words are given with the low-order byte first.

- **+rb, +rw, +rd**—A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

- **+i**—A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

OPCODE Instruction Description
F5 CMC Complement carry flag

Instruction Column

The “Instruction” column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- **rel8**—A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

- **rel16 and rel32**—A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.

- **ptr16:16 and ptr16:32**—A far pointer, typically in a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.

- **r8**—One of the byte general-purpose registers AL, CL, DL, BL, AH, CH, DH, or BH.

- **r16**—One of the word general-purpose registers AX, CX, DX, BX, SP, BP, SI, or DI.

- **r32**—One of the double word general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

- **imm8**—An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or double word operand, the immediate value is sign-extended to form a word or double word. The upper byte of the word is filled with the topmost bit of the immediate value.

- **imm16**—An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.

- **imm32**—An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.

- **r/m8**—A byte operand that is either the contents of a byte general-purpose register (AL,BL, CL, DL, AH, BH, CH, and DH), or a byte from memory.

- **r/m16**—A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, BX, CX,DX, SP, BP, SI, and DI. The contents of memory are found at the address provided by the effective address computation.

- **r/m32**—A double word general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The double word general-purpose registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. The contents of memory are found at the address provided by the effective address computation.

- **m**—A 16- or 32-bit operand in memory.

- **m8**—A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions and the XLAT instruction.

- **m16**—A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.

- **m32**—A double word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.

- **m64**—A memory quadword operand in memory. This nomenclature is used only with the CMPXCHG8B instruction.

- **m128**—A memory double quadword operand in memory. This nomenclature is used only with the Streaming SIMD Extensions.

- **m16:16, m16:32**—A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

- **m16&32, m16&16, m32&32**—A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a double word with which to load the base field of the corresponding GDTR and IDTR registers.

- **moffs8, moffs16, moffs32**—A simple memory variable (memory offset) of type byte, word, or double word used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

- **Sreg**—A segment register. The segment register bit assignments are ES=0, CS=1, SS=2,DS=3, FS=4, and GS=5.

- **m32real, m64real, m80real**—A single-, double-, and extended-real (respectively) floating-point operand in memory.

- **m16int, m32int, m64int**—A word-, short-, and long-integer (respectively) floating-point operand in memory.

- **ST or ST(0)**—The top element of the FPU register stack.

- **ST(i)**—The ith element from the top of the FPU register stack. (i = 0 through 7).

- **mm**—An MMX™ technology register. The 64-bit MMX™ technology registers are:
  - MM0 through MM7.

- **xmm**—A SIMD floating-point register. The 128-bit SIMD floating-point registers are:
  - XMM0 through XMM7.

- **mm/m32**—The low order 32 bits of an MMX™ technology register or a 32-bit memory operand. The 64-bit MMX™ technology registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

- **mm/m64**—An MMX™ technology register or a 64-bit memory operand. The 64-bit MMX™ technology registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

- **xmm/m32**—A SIMD floating-points register or a 32-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7. The contents of memory are found at the address provided by the effective address computation.

- **xmm/m64**—A SIMD floating-point register or a 64-bit memory operand. The 64-bit SIMD floating-point registers are XMM0 through XMM7. The contents of memory are found at the address provided by the effective address computation.

- **xmm/m128**—A SIMD floating-point register or a 128-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7. The contents of memory are found at the address provided by the effective address computation.

Description Column

The “Description” column following the “Instruction” column briefly explains the various forms of the instruction. The following “Description” and “Operation” sections contain more details of the instruction's operation.

Description

The “Description” section describes the purpose of the instructions and the required operands. It also discusses the effect of the instruction on flags.

Operation

The “Operation” section contains an algorithmic description (written in pseudo-code) of the instruction. The pseudo-code uses a notation similar to the Algol or Pascal language. The algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs “(*” and “*)”.

- Compound statements are enclosed in keywords, such as IF, THEN, ELSE, and FI for an *if* statement, DO and OD for a do statement, or CASE ... OF and ESAC for a *case* statement.

  A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to SI’s default segment (DS) or overridden segment.

- Parentheses around the “E” in a general-purpose register name, such as (E)SI, indicates that an offset is read from the SI register if the current address-size attribute is 16 or is read from the ESI register if the address-size attribute is 32.

- Brackets are also used for memory operands, where they mean that the content of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.

- A ¬ B; indicates that the value of B is assigned to A.

- The symbols =, ¹, ³, and £ are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.

- The expression “<< COUNT” and “>> COUNT” indicates that the destination operand should be shifted left or right, respectively, by the number of bits indicated by the count operand. The following identifiers are used in the algorithmic descriptions:

  - **OperandSize and AddressSize**—The OperandSize identifier represents the operand-size attribute of the instruction, which is either 16 or 32 bits. The AddressSize identifier represents the address-size attribute, which is either 16 or 32 bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the CMPS instruction used.

    ```
    IF instruction = CMPSW
    THEN
    OperandSize ¬ 16;
    ELSE
    IF instruction = CMPSD
    THEN OperandSize ¬ 32;
    FI;
    FI;
    ```

  - **StackAddrSize**—Represents the stack address-size attribute associated with the instruction, which has a value of 16 or 32 bits.

- **SRC**—Represents the source operand.

- **DEST**—Represents the destination operand.

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)**—Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of –10 converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.

- **SignExtend(value)**—Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value –10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.

- **SaturateSignedWordToSignedByte**—Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).

- **SaturateSignedDwordToSignedWord—**Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

- **SaturateSignedWordToUnsignedByte—**Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).

- **SaturateToSignedByte—**Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).

- **SaturateToSignedWord—**Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

- **SaturateToUnsignedByte—**Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).

- **SaturateToUnsignedWord—**Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).

- **LowOrderWord(DEST * SRC)—**Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.

- **HighOrderWord(DEST * SRC)—**Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.

- **Push(value)—**Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. Refer to the “Operation” section in “PUSH—Push Word or Doubleword Onto the Stack” in this chapter for more information on the push operation.

- **Pop()** removes the value from the top of the stack and returns it. The statement EAX ¬ Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute. Refer to the “Operation” section in “POP—Pop a Value from the Stack” in this chapter for more information on the pop operation.

- **PopRegisterStack—**Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.

- **Switch-Tasks—**Performs a task switch.

- **Bit(BitBase, BitOffset)—**Returns the value of a bit within a bit string, which is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register.

Q. 2. Discuss the architectural trends in today's microprocessor.

Sol.

Today's Microprocessor trends – an Introduction

From their humble beginning 25 years ago, microprocessors have proliferated into an astounding range of chips, powering devices ranging from telephones to supercomputers. Today, microprocessors for personal computers get widespread attention--and have enabled Intel to become the world's largest semiconductor maker. In addition, embedded microprocessors are at the heart of a diverse range of devices that have become staples of affluent consumers worldwide.

The past decade has seen the evolution of microprocessor packaging from a simple protective scheme to a complex combination of different elements that enable microprocessor performance while still providing the basic function of protection. Packaging today's microprocessor on the one hand entails tailoring the package to enable microprocessor performance, a complex task considering the rapid rate of microprocessor performance growth. This challenge is in terms of schedule and technical complexity. On the other hand, the package forms the interface between the microprocessor and the external world of the motherboard and the computing system. In this capacity, package design must allow for an easy interface and must meet a diverse set of form factor requirements.

The package provides a conduit for the microprocessor through a space transformation allowing small-scale features on the silicon to be electrically connected to the external environment. This is a challenging geometrical problem and requires that packaging interconnection densities must closely track the evolution of microprocessor interconnection densities. In connecting the die to the motherboard, the package must also ensure that the connections do not unduly inhibit the microprocessor performance by introducing unnecessary electrical impediments usually referred to as package "parasitics." As microprocessors have evolved, they have increased in speed, which in turn needs increasingly sophisticated power delivery schemes. Another consequence of microprocessor evolution has been increasing power dissipation. Package design must now provide a path for thermal dissipation, requiring a better understanding of the thermal characteristics of packaging materials and design. Package design also requires a good understanding of the structural characteristics of the package to ensure it is designed for reliability and robustness. Attention is increasingly focussed today on understanding the electrical, thermal and mechanical characteristics of packaging to optimize all these aspects.

The package is also the interface that connects the microprocessor to the motherboard. In this capacity it must have a compatible interface to allow for easy acceptance on the motherboard as well as the system design. The form factor of the package is a critical element for easy interface to the motherboard. The requirements are usually different in different market segments and often drive the need for form factors that are tailored to these different segments. For instance, the height of the package is critical to enable a microprocessor in a mobile market where a slim and low weight package is critical to success. On the other hand, the ability to dissipate high power, and hence features that enables this, are critical in a server or desktop market segment. Cost, compatibility and fit within the computer system are key parameters that must be designed for in making a microprocessor successful. This challenges us into concurrently developing multiple solutions and technologies geared towards specific market segments.

Aside from the challenges of package design, there is a need to develop efficient and cost-effective manufacturing processes that allow us to meet the schedule and volume demands of today's market place. These have presented us with interesting challenges in understanding the manufacturability, testability and reliability of packaging. Some of these issues are discussed in greater detail in this issue.

MAJC is an example of the design architecture of today's microprocessor. We can easily relate to the current trend by analysing the architecture of MAJC.

MAJC (pronounced "magic") is an acronym for "Microprocessor Architecture for Java Computing." MAJC is a microprocessor architecture designed to meet the broadband demands of the 21st century. Addressing the challenge of high bandwidth and the need for state-of-the-art computational performance, MAJC architecture is characterized by:

- Scalability to take full advantage of advances in semiconductor technology.
- Broad scalability to systems with large numbers of processors.
- A new standard of performance for applications with DSP or New Media computational needs.
- Focus on bandwidth throughput.

Processor Needs into the 21st Century

Several microprocessor trends were identified and accommodated in the design of the MAJC Architecture:

- Convergence of communication media and computers (audio, video, and data) require processors to compute information at wire speed.
- Advancements in semiconductor technology will provide rapidly-increasing resources on each microprocessor chip.
- As microprocessors are used in increasingly disparate applications from smart cards to supercomputers there is great value in the ability to create a wide span of implementations from a given processor architecture.
- Software, over time, will become independent of specific instruction sets; Just-In-Time (JIT) compilation techniques are expected to predominate for general-purpose processors and eliminate binary compatibility issues.
- Bandwidth between processors, memory, and I/O devices needs to be available to move information in real-time.
- The content processed by computers is becoming increasingly media-rich; DSP-like functions are required to process this media content.

Features of Today's Microprocessors

Modular Architecture

To support the creation of a wide range of implementations the architecture supports modular implementations. A basic implementation might comprise a single processor unit with four functional units. By replicating those design elements, an implementation can be built that includes a few or even hundreds of processors, each with four functional units, each of which can operate on many data items simultaneously with parallel-operation (SIMD) instructions. Conversely, a tiny application-specific implementation can be derived from the basic one by trimming the complement of functional units down to one or two and/or removing hardware support for any instructions not needed in its target application.

Software Portability

The architecture was designed to efficiently execute code generated by installation-time or just-in-time (JIT) compilation techniques. It may be the first commercial architecture designed without a requirement for binary compatibility between generations. This allows implementations to evolve over time without accumulating the baggage required to support old binaries, as traditional architectures have always done. Instead, software portability across implementations is obtained through use of architecture-neutral means of software distribution.

Multiple Levels of Parallelism

The architecture provides the ability to exploit parallelism at many levels - at the data word level through SIMD instructions, at the instruction level through multiple functional units per processor, at the thread-of-execution level through support for multithreaded software, and at the system level through its intrinsic support for "MPs-on-a-chip" (multiple processor units per implementation). A implementation with more than one functional unit per processor unit provides MSIMD: multiple single-instruction multiple-data parallelism.

Multiple Processor Units per Cluster

Although a MAJC implementation can be a single processor unit, the architecture explicitly incorporates the concept of multiple processors per implementation. Given 21st century semiconductor density, each such array of processor units or "processor cluster" can be implemented on a single chip. As semiconductor technology advances, clusters with more processors per chip can be implemented.

Multiple Functional Units per Processor Unit

Every MAJC processor unit can issue multiple instructions simultaneously, one to each of its functional units. Most implementations are expected to provide two to four functional units per processor unit.

Multithreaded Software

Execution of multithreaded software comes naturally given the architecture's ability to execute multiple threads simultaneously on multiple processor units. MAJC implementations with hardware support for vertical micro-threading can efficiently execute multiple threads on each processor unit.

SIMD Instructions

At the lowest level of parallelism, MAJC architecture provides SIMD (Single Instruction/ Multiple Data) or "vector" instructions. A SIMD instruction executing in a single functional unit could perform the same operation on multiple data items simultaneously.

Integral Support for Media-Rich Data

The MAJC architecture is particularly well-suited for processing media-rich content because it directly supports common media data types and can process multiple simultaneous operations on that data. Processing power is multiplied on three levels: Single Instruction/Multiple Data (SIMD) DSP-like instructions in each functional unit, multiple functional units per processor unit, and multiple processor units per processor cluster.

Balanced Performance: Processor versus Memory and I/O

A MAJC implementation is designed to utilize several techniques to balance processor speed with access to external memory and I/O devices:

- 100's of general-purpose registers per processor unit, which reduce the frequency of memory accesses.
- Load-Group instructions, which increase bandwidth into the processor by simultaneously loading multiple registers from memory or an I/O device.
- Store buffering, which increases bandwidth out of the processor by optimizing Store operations initiated by software.

Data Type-Independent Registers

The general-purpose register file in a MAJC implementation is datatype-agnostic: any register can hold information of any data type and be accessed by any instruction. In particular, there is no distinction between integer and floating-point registers. This allows registers to be allocated as needed by each application, without restrictions imposed by hardware partitioning of the register set.

Instruction Grouping

Grouping instructions across multiple functional units can be performed dynamically in hardware (as in a superscalar processor), statically by a compiler, or by some combination of the two. Rather than devoting valuable chip area to hardware grouping logic, MAJC relies primarily on software compilers to group instructions across functional units.

Data and Address Size

A MAJC implementation may implement either 32 or 64-bit addressing and data operations, as dictated by the needs of its target applications.

Context Switch Optimization

Process (task) context switch time can be reduced by using the architecture's "register dirty bits", which allow an operating system to minimize the number of registers saved and restored during a context switch.

Memory Byte Order

The MAJC architecture's native byte order is "big-endian"; that is, multi-byte values are stored in memory with the most significant byte at the lowest address and the least significant byte at the highest address. However, a MAJC implementation can manipulate data stored in any memory byte-order (notably "little-endian"). The BYTESHUFFLE instruction can reorder bytes efficiently in an arbitrary manner. Also, an implementation may define an Alternate Space Identifier (ASI) dedicated to performing automatic byte reordering whenever corresponding Load and Store from Alternate Address Space instructions are executed.