**Exam 1 Study Guide – COMP 3350**

**Number system – Chapter 1**

1. **Unsigned, signed, 2’s complement**

Integer Storage Sizes – Byte (8), Word (16), Doubleword (32), Quadword (64)

Unsigned: 0 .. 2^(n) – 1

Signed: -(2^(n – 1)) .. 2^(n – 1) – 1

2’s Complement: Flip the bits, then add 1 (for binary AND hexadecimal)

Ex: 2Ah 🡪 D5 + 1 🡪 D6

Ex: 0010 🡪 1101 + 1 🡪 1110

Signed Integers: The highest bit indicates the sign (1 = negative, 0 = positive). If the highest digit of a hexadecimal integer is > 7, the value is negative (Ex: 8A).

1. **Hex-decimal-binary**

Unsigned Decimal to Binary:

Unsigned decimal 37; 37/2 = 18, r = 1

18/2 = 9, r = 0

9/2 = 4, r = 1

4/2 = 2, r = 0

2/2 = 1, r = 0

1/2 = 0, r = 1

So, unsigned decimal 37 = 100101 binary

Hexadecimal Integers – 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, …

Binary to Hexadecimal – Each hexadecimal digit corresponds to 4 binary bits (ex: 0001 0110 1010 0111 1001 0100 = 16A794)

Hexadecimal to Decimal – Multiply each digit by its corresponding power of 16 (ex: 1234h = (1 x 163) + (2 x 162) + (3 x 161) + (4 x 160) = decimal 4660)

Decimal to Hexadecimal – Almost the same as decimal to binary

Decimal 422; 422/16 = 26, rem 6

26/16 = 1, rem A

1/16 = 0, rem 1

So, decimal 422 = 1A6 hexadecimal

1. **Familiar with ASCII values too**

Character Storage

Standard ASCII (0 – 127)

ASCII Codes (Decimal):

1. – Backspace (moves one column to the left)
2. – Horizontal tab (skips forward n columns)
3. – Line feed (moves to next output line)
4. – Form feed (moves to next printer page)
5. – Carriage return (moves to leftmost output column)
6. – Escape character

Extended ASCII (0 – 255)

ANSI (0 – 255)

Unicode (0 – 65,535)

UTF-8: Used in HTML, and has the same byte values as ASCII

UTF-16: Used in environments that balance efficient access to characters with economical use of storage. Recent versions of Microsoft Windows, for example, use UTF-16 encoding. Each character is encoded in 16 bits

UTF-32: Used in environments where space is no concern and fixed-width characters are required. Each character is encoded in 32 bits

Use the ASCII table to find values

Numeric Data Representation

Pure Binary – Can be calculated directly

ASCII Binary – String of digits: “01010101”

ASCII Decimal – String of digits: “65”

ASCII Hexadecimal – String of digits: “9C”

**2. Computer organization – Chapter 2**

**a. Architecture**

Central Processor Unit (CPU) – where calculations and logical operations take place, contains the registers, a clock, a control unit, and an arithmetic logic unit.

Control Unit (CU) – coordinates the sequencing of steps involved in executing machine instructions.

Arithmetic Logic Unit (ALU) – performs arithmetic operations such as addition and subtraction and logical operation such as AND, OR, and NOT.

**b. Function of the clock**

Synchronizes all CPU and BUS operations

Machine clock cycle measures time of a single operation

Used to trigger events

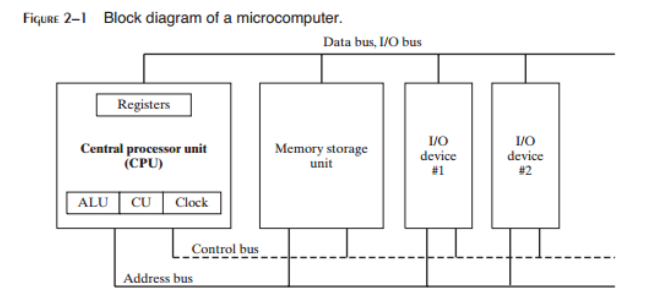
Continuously running even when computer is off.

Ex: Let us say your computer is running at 2.2 GHz. You come to know that the Add instruction takes 4 clock periods in your computer. Express the time taken by the Add instruction in nanoseconds.

F = 1/t = 2.2 \* 10^9

4 Clock periods takes = 4/(2.2\*10^9) = 1.818 \* (10^(-9)) secs = 1.818 nanosecs

1. **Basic organization**



1. **Instruction-execution cycle**

1. Fetch the Instruction – First the CPU has to fetch the instruction from an area of memory called the instruction queue. Right after doing this, it increments the instruction pointer.

2. Decode the Instruction – Next, the CPU decodes the instruction by looking at its binary bit pattern. This bit pattern might reveal that the instruction has operands (input values).

3. Fetch the Operands – If operands are involved, the CPU fetches the operands from registers and memory. Sometimes this involves address calculations.

4. Execute the Instruction – Next, the CPU executes the instruction, using any operand values it fetched during the earlier step. It also updates a few status flags, such as Zero, Carry, and Overflow.

5. Store the Result – Finally, if an output operand was part of the instruction, the CPU stores the result of its execution in the operand.

**e. Reading and writing to memory**

**i. Cycles**

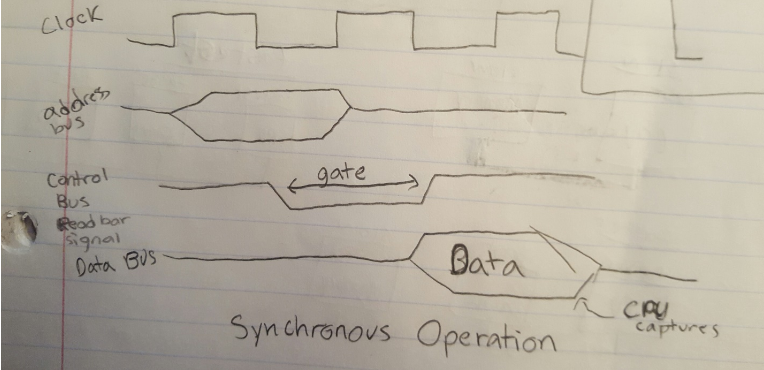
1. Place the address of the value that you want to read on the address bus.

2. Assert (change the value of) the processor’s RD (read) pin.

3. Wait one clock cycle for the memory chips to respond.

4. Copy the data from the data bus into the destination operand.

**ii. Synchronous read/write occurs**



**f. Cache: Cache hit, cache miss**

Cache – High speed expensive static RAM both inside and outside the CPU

Level 1 Cache – inside the CPU

Level 2 Cache – outside the CPU

Cache Hit – when data to be read is already in cache memory.

Cache Miss – when data to be read is not in cache memory.

**g. Protected mode, real-address mode, system management mode**

Protected Mode –Programs are given separate memory areas named segments, and the processor prevents programs from referencing memory outside their assigned segments. Can run multiple programs at the same time. It assigns each process a total of 4 GB of memory. Each program can be assigned its own reserved memory area, and programs are prevented from accidently accessing each other’s code and data.

Real Address Mode – only 1 MB of memory can be addressed. The processor can only run one program at a time, but can momentarily interrupt that program to process requests (called interrupts) from peripherals. Programs are permitted to access any memory location, including address that are linked directly to system hardware.

System Management Mode – provides an operating system with a mechanism for implementing functions such as power management and system security which are usually implemented by computer manufacturers.

**h. Registers**

General-Purpose:

EAX – extended accumulator register; used by multiplication and division instructions.

ECX – automatically used as a loop counter by the CPU

ESP – stack pointer; addresses data on the stack (a system memory structure)

ESI, EDI – index registers; used by high-speed memory transfer instructions

EBP – extended frame pointer (stack); should only be used for ordinary arithmetic or data transfer at an advanced level of programming

Segment:

CS – code segment

DS – data segment

SS – stack segment

ES, FS, GS – additional segments

EIP – instruction pointer; contains the address of the next instruction to be executed; can be manipulated by machine instructions to cause the program to branch to a new location

EFLAGS – status and control flags; each flag is a single binary bit; consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation

**i. Status flags**

Zero Flag: Set when the result of an operation produces zero in the destination operand

Sign Flag: Set when the destination operand is negative. The flag is clear when the destination is positive**;** The sign flag is a copy of the destination’s highest bit

Carry Flag: Set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand)

Overflow Flag: Set when the signed result of an operation is invalid or out of range

Parity Flag: Indicates whether or not an even number of 1 bits occurs in the least significant byte of the destination operand, immediately after an arithmetic or Boolean instruction has executed

Auxiliary Carry Flag: Set when a 1 bit carries out of position 3 in the least significant byte of the destination operand

**j. Segmented memory, linear address computation for real address mode**

A multi-tasking operating system (protected mode) allows several programs (tasks) to run in memory at the same time. Each program has its own unique area for called Segments

Segmentation – provides a way to isolate memory segments from each other.

Linear address = (Segment \* 10h) + Offset

Ex: What is the linear address corresponding to the following segment-offset: 04C2:1032?

Segment \* 10h = 04C20

04C20 + 1032 = 05C52

So, the linear address is 05C52.

**k. Paging**

Paging - a feature that permits segments to be divided into 4096-byte blocks of memory called pages. Paging permits the total memory used by all programs running at the same time to be much larger than the computers physical memory. Operating systems have utility programs named virtual memory managers. When a task is running, parts of it can be stored on disk if they are not currently in use. Parts of the task are paged to disk. Other actively executing pages remain in memory. When the processor beings to execute code that has been paged out of memory it issues a page fault, causing the page or pages containing the required code or data to be loaded back into memory.

Ex: Let us say your computer has only 256MB available for your process but your program needs 512 MB of memory. How does the computer make it possible to execute your program, as well as other processes?

Answer: Virtual memory is used to execute a process with more memory than available for the process. Using paging the process is loaded into main memory from the virtual memory.

**3. Fundamentals – Chapter 3**

**a. Adding and subtracting integers**

Hexadecimal Addition – Divide the sum of two digits by the number base (16). The quotient becomes the carry value, and the remainder is the sum digit (Ex: 6A + 4B = B5 (21/16 = 1, rem 5))

Hexadecimal Subtraction – When a borrow is required from the digit to the left, add 16 (decimal) to the current digit’s value (Ex: 75 – 47 = 2E (7 – 1 = 6, 16 + 5 = 21, 21 – 7 = E))

Binary Addition – Starting with the LSB, add each pair of digits, include the carry if present (ex: 00000100 + 00000111 = 00001011)

Binary Subtraction – When subtracting A – B, convert B to its two’s complement (i.e., add A to (-B)) (Ex: 00001100 – 00000011 -> 00001100 + 11111101 = 00001001)

Constant Integer Expressions: A mathematical expression involving integer literals and arithmetic operators. Each expression must evaluate to an integer, which can be stored in 32 bits (0 through FFFFFFFFh)

Operators and Precedence Levels:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Precedence Level** |
| ( ) | Parenthesis | 1 |
| +, - | Unary plus, minus | 2 |
| \*, / | Multiply, divide | 3 |
| MOD | Modulus | 3 |
| +, - | Add, subtract | 4 |

Examples:

|  |  |
| --- | --- |
| **Expression** | **Value** |
| 16 / 5 | 3 |
| - (3 + 4) \* (6 – 1) | -35 |
| -3 + 4 \* 6 – 1 | 20 |
| 25 mod 3 | 1 |

Real Number (Floating-point) Literals: Represented as either decimal reals or encoded (hexadecimal) reals

Decimal Real: Contains an optional sign followed by an integer, a decimal point, an optional integer that expresses a fraction, and an optional exponent: [sign]integer.[integer][exponent]

Encoded Real: Represents a real number in hexadecimal, using the IEEE floating-point format for short reals

Ex: 0011 1111 1000 0000 0000 0000 0000 0000 🡪 3F800000r

Sign {+, -}; exponent E[{+, -}]integer

**b. Assemble-link-execute cycle**

The following diagram describes the steps from creating a source program through executing the compiled program:

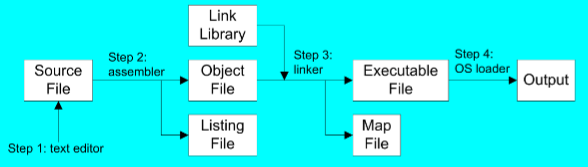
Step 1: A programmer uses a text editor to create an ASCII text file named the source file

Step 2: The assembler reads the source file and produces an object file, a machine-language translation of the program. Optionally, it produces a listing file. If any errors occur, the programmer must return to Step 1 and fix the problem.

Step 3: The linker reads the object file and checks to see if the program contains any calls to procedures in a link library. The linker copies any required procedures from the link library, combines them with the object file, and produces the executable file. Optionally, it produces a map file

Step 4: The operating system loader utility reads the executable file into memory and branches the CPU to the program’s starting address, and the program begins to execute

If the source code is modified, Steps 2 through 4 must be repeated

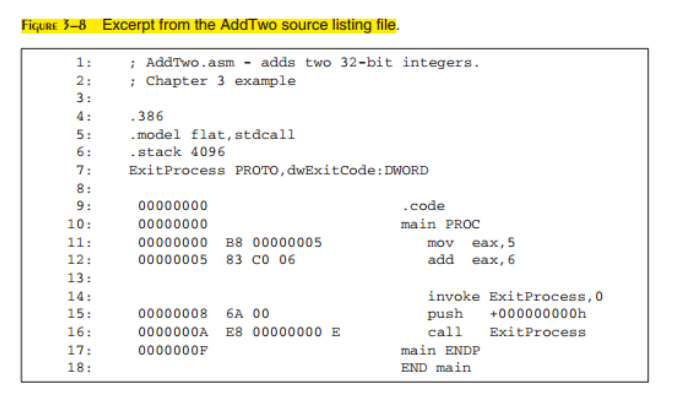


**c. .lst file**

A listing file contains a copy of the program’s source code, with line numbers, the numeric address of each instruction, the machine code bytes of each instruction (in hexadecimal), and a symbol table.

The symbol table contains the names of all program identifiers, segments, and related information.

Advanced programmers use the listing file to get detailed info about a program.



Above, line 11 says,

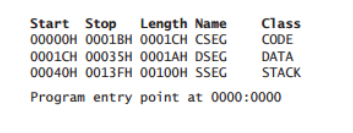
11: 00000000 B8 00000005 mov eax, 5

The first number, 00000000, is the starting address of the instruction and the one after B8, 00000005 is the location of the next offset. B8 is the operation code (opcode) because it represents a specific instruction.

**d. Map file**

Map File – Lists all segments in the program.

Ex: Map File showing one code segment, one data segment, and one stack segment:



**e. Data definitions**

**i. Little endian order**

Little Endian Order:

All data types larger than a byte store their individual bytes in reverse order. The least significant byte occurs at the first (lowest) memory address.

Ex: val1 DWORD 12345678h

|  |  |
| --- | --- |
| 0000: | 78 |
| 0001: | 56 |
| 0002: | 34 |
| 0003: | 12 |

Big Endian Order:

The opposite of little endian order; bytes are stored high to low

Ex: val1 DWORD 12345678h

|  |  |
| --- | --- |
| 0000: | 12 |
| 0001: | 34 |
| 0002: | 56 |
| 0003: | 78 |

**ii. =, EQU, $**

Equal-Sign Directive:

Name = expression

Expression is a 32-bit integer (expression or constant)

May be redefined

Name is called a symbolic constant

When a program is assembled, all instances of name are replaced by expression during the assembler’s preprocessor step

Good programming style to use symbols (i.e., COUNT = 500)

EQU Directive:

Define a symbol as either an integer or text expression

Three Formats: name EQU expression, name EQU symbol, name EQU <text>

Cannot be redefined

Ex: PI EQU <3.1416>

pressKey EQU <”Press any key to continue…”,0>

.data

Prompt BYTE pressKey

Current location counter: $

Subtract address of list

Difference is the number of bytes

List BYTE 10,20,30,40

ListSize = ($ - List)

Must be done IMMEDIATELY after variable is declared; listsize will be incorrect if declared later in the program

**4. Data transfers, addressing, arithmetic – Chapter 4**

**a. Immediate, Register-Register**

Immediate: A constant integer (8, 16, or 32 bits); value is encoded within the instruction

Register: The name of a register; register name is converted to a number and encoded within the instruction

MOV Instruction:

Move from source to destination

Syntax: MOV destination, source

Standard MOV Instruction Formats: MOV reg,reg

MOV mem,reg

MOV reg,mem

MOV mem,imm

MOV reg,imm

No more than one memory operand permitted

CS, EIP, and IP cannot be the destination

No immediate to segment moves

.data

bVal BYTE 100

bVal2 BYTE ?

wVal WORD 2

dVal DWORD 5

.code

Mov bl, bVal

Mov ax, wVal

Mov bVal, al

Mov al, wVal ; error

Mov ax, bVal ; error

Mov eax, bVal ; error

Mov ds, 45 ; immediate move to DS not permitted

Mov esi, wVal ; size mismatch

Mov eip, dVal ; EIP cannot be the destination

Mov 25, bVal ; immediate value cannot be destination

Mov bVal2, bVal ; memory-to-memory move not permitted

**b. Memory: Direct, indirect, offset, base index, base index w. display**

Memory: Reference to a location in memory; memory address is encoded within the instruction, or a register holds the address of a memory location

A direct memory operand is a named reference to storage in memory

The named reference (label) is automatically dereferenced by the assembler

Direct-Offset Operands:

A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

Ex: .data

arrayB BYTE 10h, 20h, 30h, 40h

arrayW WORD 1000h, 2000h, 3000h

arrayD DWORD 1, 2, 3, 4

.code

Mov al, arrayB+1 ; AL = 20h

Mov al, [arrayB+1] ; alternative notation

Mov ax, [arrayW+2] ; AX = 2000h

Mov ax, [arrayW+4] ; AX = 3000h

Mov eax, [arrayD+4] ; EAX = 00000002h

Indirect Operands: Holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer)

Can be any 32-bit general-purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP)

Ex: .data

Val1 BYTE 10h, 20h, 30h

.code

Mov esi, OFFSET val1

Mov al, [esi] ; dereference ESI (AL = 10h)

Inc esi

Mov al, [esi] ; AL = 20h

Inc esi

Mov al, [esi] ; AL = 30h

Use PTR to clarify the size attribute of a memory operand

Ex: .data

myCount WORD 0

.code

Mov esi, OFFSET myCount

Inc [esi] ; error: ambiguous

Inc WORD PTR [esi] ; ok

OFFSET Operator: Returns the distance in bytes, of a label from the beginning of its enclosing segment

BYTE, SBYTE: 8-bit unsigned integer; 8-bit signed integer

Ex: Value1 BYTE ‘A’ ; character constant

Value2 BYTE 0 ; smallest unsigned byte

Value3 BYTE 255 ; largest unsigned byte

Value4 SBYTE -128 ; smallest signed byte

Value5 SBYTE +127 ; largest signed byte

Value6 BYTE ? ; uninitialized byte

Value1 is located at offset 0000, Value2 is at offset 0001, Value3 is at offset 0002, and so on

If there were two bytes in Value1, then they would be offset 0000 and offset 0001, respectively

WORD, SWORD: 16-bit unsigned & signed integer

Ex: Word1 WORD 65535 ; largest unsigned value

Word2 SWORD -32768 ; smallest signed value

Word3 WORD ? ; uninitialized, unsigned

Word4 WORD “AB” ; double characters

myList WORD 1,2,3,4,5 ; array of words

array WORD 5 DUP(?) ; uninitialized array

Word1 is located at offset 0000, Word2 is at offset 0002, and so on

DWORD, SDWORD: 32-bit unsigned & signed integer

Ex: Val1 DWORD 12345678h ; unsigned

Val2 SDWORD -2147483648 ; signed

Val3 DWORD 20 DUP(?) ; unsigned array

Val4 SDWORD -3,-2,-1,0,1 ; signed array

Val1 is located at offset 0000, Val2 is at offset 0004, and so on

Indexed Operands: Adds a constant to a register to generate an effective address. There are two notational forms: [constant + reg] and constant[reg]

Ex: .data

arrayW WORD 1000h, 2000h, 3000h

.code

Mov esi, 0

Mov ax, [arrayW + esi] ; AX = 1000h

Mov ax, arrayW[esi] ; alternate format

Add esi, 2

Add ax, [arrayW + esi]

Index Scaling: You can scale an indirect or indexed operand to the offset of an array element. This is done by multiplying the index by the array’s TYPE:

Ex: .data

arrayB BYTE 0, 1, 2, 3, 4, 5

arrayW WORD 0, 1, 2, 3, 4, 5

arrayD DWORD 0, 1, 2, 3, 4, 5

.code

Mov esi, 4

Mov al, arrayB[esi\*TYPE arrayB] ; 04

Mov bx, arrayW[esi\*TYPE arrayW] ; 0004

Mov edx, arrayD[esi\*TYPE arrayD] ; 00000004

**c. Movsx, movzx**

Zero Extension:

When you copy a smaller value into a larger destination, the MOVZX instruction fills (extends) the upper half of the destination with zeros

Ex: mov bl, 10001111b

Movzx ax, bl ; AX = 0000000010001111b

The destination must be a register

Sign Extension:

The MOVSX instruction fills the upper half of the destination with a copy of the source operand’s sign bit

Ex: mov bl, 10001111b

Movsx ax, bl ; AX = 1111111110001111b

The destination must be a register

**d. Inc, dec, add, sub, neg**

INC and DEC Instructions:

Syntax: INC reg/mem and DEC reg/mem

Add 1, subtract 1 from destination operand; operand may be register or memory

INC destination (Logic: destination 🡨 destination + 1)

DEC destination (Logic: destination 🡨 destination – 1)

Ex: mov ax, 00FFh

Inc ax ; AX = 0100h

Mov ax, 00FFh

Inc al ; AX = 0000h

ADD and SUB Instructions:

ADD destination, source (Logic: destination 🡨 destination + source)

SUB destination, source (Logic: destination 🡨 destination – source)

Same operand rules as for the MOV instruction

Ex: .data

Var1 DWORD 10000h

Var2 DWORD 20000h

.code ; -----EAX-----

Mov eax, var1 ; 00010000h

Add eax, var2 ; 00030000h

Add ax, 0FFFFh ; 0003FFFFh

Add eax, 1 ; 00040000h

Sub ax, 1 ; 0004FFFFh

NEG (Negate) Instruction:

Syntax: NEG reg/mem

Reverses the sign of an operand. Operand can be a register or memory operand.

Ex: .data

valB BYTE -1

valW WORD +32767

.code

Mov al, valB ; AL = -1

Neg al ; AL = +1

Neg valW ; valW = -32767

**e. Flags**

**i. Impact of arithmetic on flags**

Flags Affected by Arithmetic:

The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations

Based on the contents of the destination operand

The MOV instruction never affects the flags

A flag is set when it equals 1, and clear when it equals 0

NEG Instruction and the Flags:

The processor implements NEG using the following internal operation SUB 0, operand

Any nonzero operand causes the Carry flag to be set

Ex: .data

valB BYTE 1, 0

valC SBYTE -128

.code

Neg valB ; CF = 1, OF = 0

Neg [valB + 1] ; CF = 0, OF = 0

Neg valC ; CF = 1, OF = 1

Zero Flag:

mov cx, 1

sub cx, 1 ; CX = 0, ZF = 1

mov ax, 0FFFFh

inc ax ; AX = 0, ZF = 1

inc ax ; AX = 1, ZF = 0

Sign Flag:

mov cx, 0

sub cx, 1 ; CX = -1, SF = 1

add cx, 2 ; CX = 1, SF = 0

mov al, 0

sub al, 1 ; AL = 11111111b, SF = 1

add al, 2 ; AL = 00000001b, SF = 0

Carry Flag:

ADD causes the CF = (carry out of the MSB)

SUB causes the CF = INVERT (carry out of the MSB)

mov al, 0FFh

add al, 1 ; CF = 1, AL = 00

mov al, 0

sub al, 1 ; CF = 1, AL = FF

Overflow Flag:

ADD causes the OF = CF XOR MSB

SUB causes the OF = CF XOR MSB

A Rule of Thumb:

When adding two integers, remember that the Overflow flag is only set when…

Two positive operands are added and their sum is negative

Two negative operands are added and their sum is positive

mov al, +127

add al, 1 ; OF = 1, AL = ??

mov al, 7Fh

add al, 1 ; OF = 1, AL = 80h

Parity Flag:

Ex: mov al, 10001100b

Add al, 00000010b ; AL = 10001110, PF = 1

Sub al, 10000000b ; AL = 00001110, PF = 0

Auxiliary Carry Flag:

Ex:mov al, 0Fh

Add al, 1 ; AC = 1

**f. Offset operator, type, ptr, lengthof, sizeof, label directive**

OFFSET Operator: Returns the distance in bytes, of a label from the beginning of its enclosing segment

Protected Mode: 32 bits

The programs we write in this mode will use only a single segment (flat memory model)

Real Mode: 16 bits

Ex: .data

bVal BYTE ?

wVal WORD ?

dVal DWORD ?

dVal2 DWORD ?

.code

mov esi, OFFSET bVal ; ESI = 00404000

mov esi, OFFSET wVal ; ESI = 00404001

mov esi, OFFSET dVal ; ESI = 00404003

mov esi, OFFSET dVal2 ; ESI = 00404007

The value returned by OFFSET is a pointer, just like how we use pointers in C++

TYPE Operator: Returns the size, in bytes, of a single element of a data declaration

Ex: .data

Var1 BYTE ?

Var2 WORD ?

Var3 DWORD ?

Var4 QWORD ?

.code

Mov eax, TYPE var1 ; 1

Mov eax, TYPE var2 ; 2

Mov eax, TYPE var3 ; 4

Mov eax, TYPE var4 ; 8

PTR Operator: Overrides the default type of a label (variable). Provides the flexibility to access part of a variable

Ex: .data

myDouble DWORD 12345678h

.code

mov ax, myDouble ; error

mov ax, WORD PTR myDouble ; loads 5678h

mov WORD PTR myDouble, 4321h ; saves 4321h

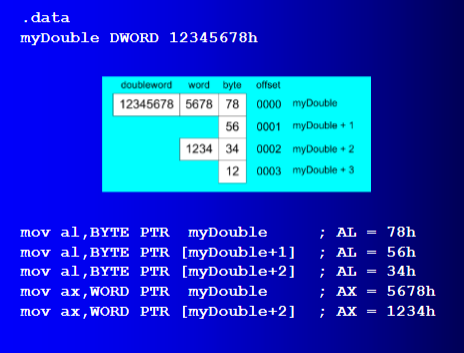
Little endian order is used when storing data in memory

Little endian order refers to the way Intel stores integers in memory

Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions

PTR Operator Examples:



PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The CPU will automatically reverse the bytes

Ex: .data

myBytes BYTE 12h, 34h, 56h, 78h

.code

Mov ax, WORD PTR [myBytes] ; AX = 3412h

Mov ax, WORD PTR [myBytes+2] ; AX = 7856h

Mov eax, DWORD PTR myBytes ; EAX = 78563412h

LENGTHOF Operator: Counts the number of elements in a single data declaration

Ex: .data

Byte1 BYTE 10, 20, 30 ; 3

Array1 WORD 30 DUP(?), 0, 0 ; 32

Array2 WORD 5 DUP(3 DUP(?)) ; 15

Array3 DWORD 1, 2, 3, 4 ; 4

DigitStr BYTE “12345678”, 0 ; 9

.code

Mov ecx, LENGTHOF array1 ; 32

SIZEOF Operator: Returns a value that is equivalent to multiplying LENGTHOF by TYPE

Ex: .data

Byte1 BYTE 10, 20, 30 ; 3

Array1 WORD 30 DUP(?), 0, 0 ; 64

Array2 WORD 5 DUP(3 DUP(?)) ; 30

Array3 DWORD 1, 2, 3, 4 ; 16

DigitStr BYTE “12345678”, 0 ; 9

.code

Mov ecx, SIZEOF array1 ; 64

LABEL Directive: Assigns an alternate label name and type to an existing storage location; does not allocate any storage of its own

Removes the need for the PTR operator

Ex: .data

dwList LABEL DWORD

wordList LABEL WORD

intList BYTE 00h, 10h, 00h, 20h

.code

mov eax, dwList ; 20001000h

mov cx, wordList ; 1000h

mov dl, intList ; 00h

**g. Jmp, loop**

Unconditional Transfer: Control is transferred to a new location in all cases; a new address is loaded into the instruction pointer, causing execution to continue at the new address. The JMP instruction does this

Conditional Transfer: The program branches if a certain condition is true. A wide variety of conditional transfer instructions can be combined to create conditional logic structures. The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

JMP Instruction: An unconditional jump to a label that is usually within the same procedure

Syntax: JMP target; Logic: EIP 🡨 target

Ex: top:

.

.

jmp top

A jump outside the current procedure must be to a special type of label called a global label

LOOP Instruction: Creates a counting loop

Syntax: LOOP target; Logic: ECX 🡨 ECX – 1, if ECX != 0, jump to target

Implementation: The assembler calculates the distance, in bytes, between the offset of the following instruction and the offset of the target label. It is called the relative offset

The relative offset is added to EIP

Ex:

