**COMP 3350 Final Study Guide**

**EXAM 1 MATERIAL:**

**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

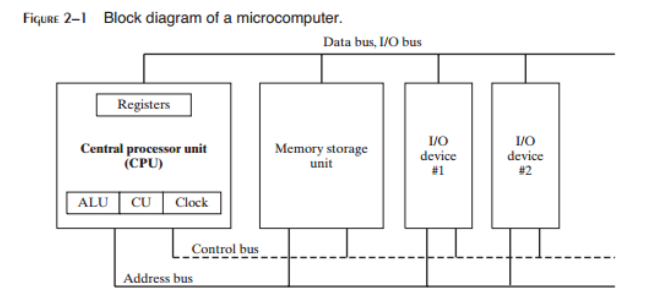
Continuously running even when computer is off.

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**

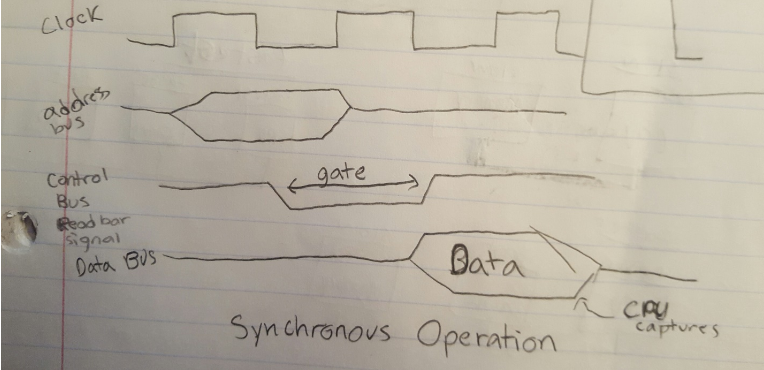
1. Place the address of the value that you want to read on the address bus.

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:

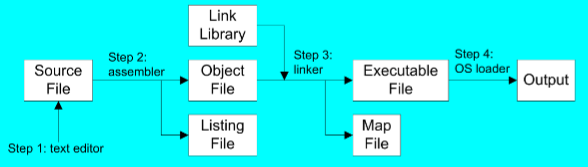
Step 1: A programmer uses a text editor to create an ASCII text file named the source file

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

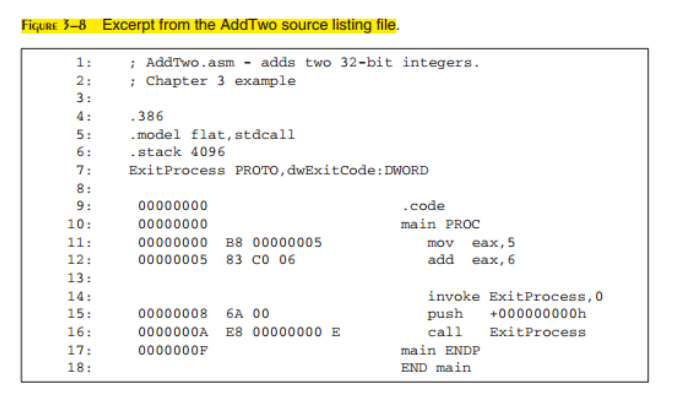


**c. .lst file**

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,

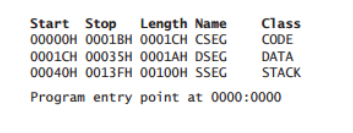
11: 00000000 B8 00000005 mov eax, 5

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

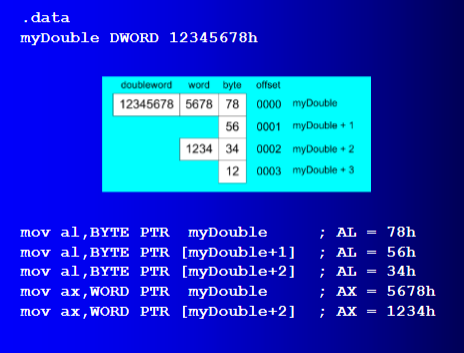
Little endian order is used when storing data in memory

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

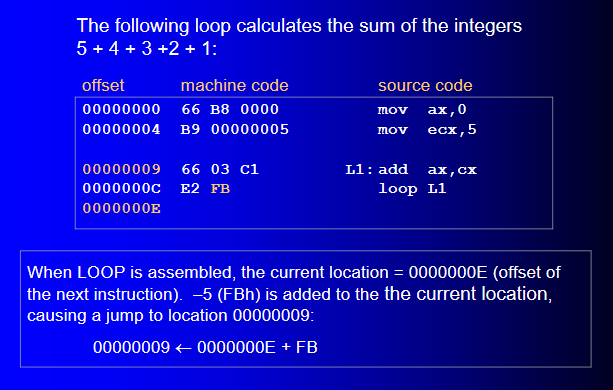
LOOP Instruction: Creates a counting loop

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:



**EXAM 1 PRACTICE QUESTIONS:**

1. The study of Comp. Org and Assembly language programming helps us:
2. The study of Comp. Org and Assembly language programming helps:
3. What is the decimal value for the signed 2’s complement byte A4h?
4. Which flag is set when the result of an unsigned arithmetic operation is too large to fit into the destination?
5. Which flag is set when the result of a signed arithmetic operation is too large to fit into the destination?
6. Which flag is set when the arithmetic operations produce a negative result?
7. CF = ? ; SF = ?
   1. ; Sally’s Code Fragment
   2. Mov ax, 7FF0h
   3. Add al, 10h
   4. Add ah, 1
   5. Add ax, 2
8. Fill in the values of the flag(s) after execution of Sally’s instruction c program fragment (assume prior instructions have executed) ZF = ? ; OF = ?
9. Fill in the values of the flag(s) after execution of Sally’s instruction d program fragment (assume prior instructions have executed) CF = ? ; SF = ?
10. Fill in the values of the flag(s) after execution of Sally’s instruction d program fragment (assume prior instructions have executed) ZF = ? ; OF = ?
11. Fill in the values of the flag(s) after execution of Sally’s instruction e program fragment (assume prior instructions have executed) CF = ? ; SF = ?
12. Fill in the values of the flag(s) after execution of Sally’s instruction e program fragment (assume prior instructions have executed) ZF = ? ; OF = ?
13. What is the value, in decimal, of ax after the following have executed?
    1. .data
    2. Array DWORD 25 dup(?), 0
    3. .code
    4. Mov ax, sizeof array
14. What is the value, in decimal, of bx after the following code?
    1. .data
    2. Array DWORD 43 dup(?), 0
    3. .code
    4. Mov bx, lengthof array
15. What is the AX register value? Use for questions 15 – 16.
    1. ; Joe’s Program
    2. .data
    3. MyBytes BYTE AAh, BBh, CCh, DDh
    4. MyWords WORD DEF0h, 9876h, ABCDh, 1234h, 5678h
    5. .code
    6. Mov esi, OFFSET MyBytes
    7. Mov ax, WORD PTR[esi]
    8. Mov ebx, DWORD PTR MyWords
16. What will be the value of ebx?
17. For use with questions 17 – 18
    1. ; Programmer: Tom
    2. ; assume offset of alpha = 404,000h
    3. .data
    4. Alpha WORD 5566h
    5. Beta WORD AAh, BBh, CCh
    6. Gamma WORD DDh, EEh, FFh
    7. .code
    8. Mov ax, 33h
    9. Mov edi, OFFSET beta
    10. Mov ebx, edi
    11. Mov [edi], ax
    12. Mov ax, [alpha]
    13. Mov esi, 4
    14. Mov ax, beta[esi]
    15. Mov ax, 2[ebx]
    16. Mov ax, 4[ebx][esi]

Write the hex addresses of memory location(s) changed after instruction n of Tom’s program.

1. In Tom’s program, what is the value in ax in line s?
2. Consider the instruction mov CS, A456. The instruction causes the following:
3. The EIP register cannot be a destination.
4. The NEG AX instruction performs:
5. What is the value (in decimal) of the maximum possible backward-jump (in bytes) in a single loop instruction?
6. A cache memory helps in which of the following ways?
7. Put, in order, the stages of an instruction execution cycle. The following abbreviations have been used (FI = Fetch Instruction, FO = Fetch Operands, DI = Decode Instructions, SO = Store Output, EI = Execution Instructions).
8. The function of the clock is to:
9. The loops in Bob’s program computes the sum 5+4+3+2+1. Below are the instructions addresses and the corresponding code generated by the assembler.
   1. ; Bob’s Program
   2. 00000000 66 B8 0000 mov ax, 0
   3. 00000004 B9 00000005 mov ecx, 5
   4. 00000009 66 03 C1 L1: add ax, cx
   5. 0000000C E2 FB loop L1
   6. 0000000E

When the loop is executed, what value is added to EIP to effect the transfer of control to L1?

1. To create a nested loop (outer loop-inner loop) using the loop instruction, the following strategy is feasible:
2. Which provides the fastest access to data for the CPU?

**Answers to Questions**

1. In precise control of timing, optimizing memory in programs, and also allows control over I/O operations
2. Understanding Computer Architecture, Operating Systems, write device drivers, and compilers.
3. -92
4. CF
5. OF
6. SF
7. CF = 1 ; SF = 0
8. ZF = 1 ; OF = 0
9. CF = 0 ; SF = 1
10. ZF = 0 ; OF = 1
11. CF = 0 ; SF = 1
12. ZF = 0 ; OF = 0
13. 104
14. 44
15. BBAAh
16. 9876DEF0
17. 404002h
18. 00EEh
19. The instruction is illegal because an immediate value cannot be placed directly into CS.
20. True
21. A 2’s complement of AX.
22. -128
23. To access data and code quicker than main memory.
24. FI, DI, FO, EI, SO
25. Synchronize CPU operations and bus operations.
26. FB
27. Save and restore ecx in a register/variable in the outer loop.
28. Registers.

**EXAM 2 MATERIAL:**

**A. Chapter 5 - Procedures**

**1. Linking to an External Library**

What is a Link Library:

A file containing procedures that have been compiled into machine code; constructed from one or more OBJ files.

To build a library…

Start with one or more ASM source files.

Assemble each into an OBJ file.

Create an empty library file (extension: .LIB).

Add the OBJ file(s) to the library file, using the Microsoft LIB utility.

How the Linker Works:

Your programs link to Irvine32.lib using the linker command inside a batch file named make32.bat.

Notice the two LIB files: Irvine32.lib, & kernel32.lib.

The latter is part of the Microsoft Win32 Software Development Kit (SDK).

**2. Calling library functions**

**i. Usage of standard library functions (HW)**

Calling Irvine32 Library Procedures: Call each procedure using the CALL instruction. Some procedures require input arguments. The INCLUDE directive copies in the procedure prototypes (declarations).

**3. DumpRegs, CRLF**

DumpRegs: Displays general-purpose registers & flags (hex).

Format: call DumpRegs

No call arguments or return arguments from procedure

Crlf: Writes end of line sequence to standard output.

Format: call crlf

No call arguments or return arguments from procedure

**4. WriteHex, WriteDec**

WriteHex: Writes an unsigned 32-bit integer in hexadecimal format.

Format: mov eax, 077Dh

Call WriteHex

Call Arguments: EAX = Value to write

Return Arguments: None

WriteDec: Writes unsigned 32-bit integer in decimal format.

Format: mov eax, 077Dh

Call WriteDec

Call Arguments: EAX = Value to write

Return Arguments: None

**5. Stack Operations (Push/pop)**

**i. What is a push/pop**

PUSH Operation: A 32-bit push operation decrements the stack pointer by 4 & copies a value into the location pointed to by the stack pointer.

Syntax: PUSH r/m16, PUSH r/m32, PUSH imm32

POP Operation: Copies value at stack[ESP] into a register or variable.

Adds n to ESP, where n is either 2 or 4. The value of n depends on the attribute of the operand receiving the data.

Syntax: POP r/m16, POP r/m32

**ii. When to push/pop a register inside procedures**

Using PUSH & POP: Save & restore registers when they contain important values. PUSH & POP instructions occur in the opposite order.

Ex: push esi ; push registers

Push ecx

Push ebx

Mov esi, OFFSET dwordVal ; display some memory

Mov ecx, LENGTHOF dwordVal

Mov ebx, TYPE dwordVal

Call DumpMem

Pop ebx ; restore registers

Pop ecx

Pop esi

Nested Loops: When creating a nested loop, push the outer loop counter before entering the inner loop (remember to pop it after the second loop completes).

Reversing a String:

Use a loop with indexed addressing.

Push each character on the stack.

Start at the beginning of the string, pop the stack in reverse order, insert each character back into the string.

**iii. When not to push a register**

Do not push/pop a register when you want to modify the value within it permanently.

**6. Call/return**

CALL & RET Instructions:

The CALL instruction calls a procedure.

Pushes offset of next instruction on the stack.

Copies the address of the called procedure into EIP.

The RET instruction returns from a procedure.

Pops top of stack into EIP.

**7. Local/Global labels**

Local & Global Labels: A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

**8. Uses Operator**

USES Operator: Lists the registers that will be preserved.

Syntax: ArraySum PROC USES esi ecx

Mov eax,0 ; set the sum to zero

Etc.

Replaces the push & pop operations.

**9. Passing parameters**

Procedure Parameters:

A good procedure might be usable in many different programs, but not if it refers to specific variable names.

Parameters help to make procedure flexible because parameter values can change at runtime.

Utilize the USES operator if you want to allow the procedure to take various inputs as parameters

**B. Chapter 6 – Conditional Processing**

**1. PF, AC**

Parity Flag: Set when an instruction generates an even number of 1 bits in the low byte of the destination operand.

Auxiliary Carry Flag: Set when an operation produces a carry out from bit 3 to bit 4.

**2. AND/OR/XOR/NOT**

AND Instruction: Performs a Boolean AND operation between each pair of matching bits in two operands.

Syntax: AND destination, source

OR Instruction: Performs a Boolean OR operation between each pair of matching bits in two operands.

Syntax: OR destination, source

XOR Instruction: Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands.

Syntax: XOR destination, source

NOT Instruction: Performs a Boolean NOT operation on a single destination operand.

Syntax: NOT destination

**i. Uses of Boolean operations**

Applications:

Task: Convert the character in AL to upper case.

Solution: Use the AND instruction to clear bit 5.

Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.

Solution: Use the OR instruction to set bits 4 & 5.

Task: Turn on the keyboard CapsLock key.

Solution: Use the OR instruction to set bit 6 in the keyboard flag byte at 0040:0017h in the BIOS data area.

Task: Jump to a label if an integer is even.

Solution: AND the lowest bit with a 1. If the result is Zero, the number was even.

Task: Jump to a label if the value in AL is not zero.

Solution: OR the byte with itself, then use the JNZ (jump if not zero) instruction.

**3. Bit-mapped set operations**

**i. Intersection**

Set Intersection:

Mov eax, setX

And eax, setY

**ii. Union**

Set Union:

Mov eax, setX

Or eax, setY

**4. TEST/CMP**

TEST Instruction:

Performs a nondestructive AND operation between each pair of matching bits in two operands.

No operands are modified, but the Zero flag is affected.

CMP Instruction:

Compares the destination operand to the source operand.

Nondestructive subtraction of source from destination (destination operand is not changed).

Syntax: CMP destination, source

**i. Way they affect flags**

Ex: Jump to a label if either bit 0 or bit 1 in AL is set.

test al, 00000011b

Jnz ValueFound

Ex: Jump to a label of neither bit 0 nor bit 1 in AL is set.

Test al, 00000011b

Jz ValueNotFound

Ex: Destination == source

Mov al, 5

Cmp al, 5 ; Zero flag set

Ex: destination < source

Mov al, 4

Cmp al, 5 ; Carry flag set

Ex: Destination > source

Mov al, 6

Cmp al, 5 ; ZF = 0, CF = 0

The next two comparisons shown are performed with signed integers.

Ex: Destination > source

Mov al, 5

Cmp al, -2 ; Sign flag == Overflow flag

Ex: Destination < source

Mov al, -1

Cmp al, 5 ; Sign flag != Overflow flag

**5. Conditional Jumps**

**i. JCC**

Jcond Instruction: A conditional jump instruction branches to a label when specific register or flag conditions are met.

Specific Jumps:

JB, JC: Jump to a label if the Carry flag is set.

JE, JZ: Jump to a label if the Zero flag is set.

JS: Jump to a label if the Sign flag is set.

JNE, JNZ: Jump to a label if the Zero flag is clear.

JECXZ: Jump to a label if ECX = 0.

Jcond Ranges:

Prior to the 386: Jump must be within -128 to +127 bytes from current location counter.

X86 Processors: 32-bit offset permits jump anywhere in memory.

Jumps Based on Specific Flags:

JZ: Jump if zero; ZF = 1

JNZ: Jump if not zero; ZF = 0

JC: Jump if carry; CF = 1

JNC: Jump if not carry; CF = 0

JO: Jump if overflow; OF = 1

JNO: Jump if not overflow; OF = 0

JS: Jump if signed; SF = 1

JNS: Jump if not signed; SF = 0

JP: Jump if parity (even); PF = 1

JNP: Jump if not parity (odd); PF = 0

Jumps Based on Equality:

JE: Jump if equal (leftOp = rightOp)

JNE: Jump if not equal (leftOp != rightOp)

JCXZ: Jump if CX = 0

JECXZ: Jump if ECX = 0

**ii. Unsigned and signed differences**

Jumps Based on Unsigned Comparisons:

JA: Jump if above (if leftOp > rightOp)

JNBE: Jump if not below or equal (same as JA)

JAE: Jump if above or equal (if leftOp >= rightOp)

JNB: Jump if not below (same as JAE)

JB: Jump if below (if leftOp < rightOp)

JNAE: Jump if not above or equal (same as JB)

JBE: Jump if below or equal (if leftOp <= rightOp)

JNA: Jump if not above (same as JBE)

Jumps Based on Signed Comparisons:

JG: Jump if greater (if leftOp > rightOp)

JNLE: Jump if not less than or equal (same as JG)

JGE: Jump if greater than or equal (if leftOp >= rightOp)

JNL: Jump if not less (same as JGE)

JL: Jump if less (if leftOp < rightOp)

JNGE: Jump if not greater than or equal (same as JL)

JLE: Jump if less than or equal (if leftOp <= rightOp)

JNG: Jump if not greater (same as JLE)

**6. Bit Test**

BT (Bit Test) Instruction:

Copies bit n from an operand into the Carry flag.

Syntax: BT bitBase, n

bitBase may be r/m16 or r/m32

n may be r16, r32, or imm8

**7. Conditional Loops**

**i. Loopz/Loopnz**

LOOPZ & LOOPE:

Syntax: LOOPE destination & LOOPZ destination

Logic:

ECX 🡨 ECX – 1

If ECX > 0 & ZF = 1, jump to destination

Useful when scanning an array for the first element that does not match a given value.

LOOPNZ & LOOPNE:

LOOPNZ (LOOPNE) is a conditional loop instruction.

Syntax: LOOPNZ destination & LOOPNE destination

Logic:

ECX 🡨 ECX – 1

If ECX > 0 & ZF = 0, jump to destination

Useful when scanning an array for the first element that matches a given value.

**ii. Expressions -> Programs**

LOOPNZ Example

.data

array SWORD -3,-6,-1,-10,10,30,40,4

sentinel SWORD 0

.code

mov esi,OFFSET array

mov ecx,LENGTHOF array

next:

test WORD PTR [esi],8000h ; test sign bit

pushfd ; push flags on stack

add esi,TYPE array

popfd ; pop flags from stack

loopnz next ; continue loop

jnz quit ; none found

sub esi,TYPE array ; ESI points to value

quit:

**8. Table Driven Selection**

Table-driven selection uses a table lookup to replace a multiway selection structure.

Create a table containing lookup values & the offsets of labels or procedures.

Use a loop to search the table.

Suited to a large number of comparisons.

Step 1: Create a table containing lookup values & procedure offsets.

Step 2: Use a loop to search the table. When a match is found, call the procedure offset stored in the current table entry.

**9. String Encryption**

Encrypting a String

KEY = 239 ; can be any byte value

BUFMAX = 128

.data

buffer BYTE BUFMAX+1 DUP(0)

bufSize DWORD BUFMAX

.code

mov ecx,bufSize ; loop counter

mov esi,0 ; index 0 in buffer

L1:

xor buffer[esi],KEY ; translate a byte

inc esi ; point to next byte

loop L1

**C. Chapter 7 – Integer Arithmetic**

**1. Shift and Rotate**

**i. Difference between arithmetic and logical shift**

Logical Shift: Shifts every bit right and fills the newly created position with zero.

Arithmetic Shift: Shifts every bit right and fills the newly created bit position with a copy of the number’s sign bit.

**ii. SHL/SHR – with or without carry**

SHL Instruction: Performs a logical left shift on the destination operand, filling the lowest bit with 0.

Operand Types for SHL: SHL reg, imm8, SHL mem, imm8, SHL reg, CL, SHL mem, CL

SHR Instruction: Performs a logical right shift on the destination operand. The highest bit position is filled with a zero.

Operand Types for SHR: SHR reg, imm8, SHR mem, imm8, SHR reg, CL, SHR mem, CL

**1. Logical or arithmetic**

Both SHL and SHR instructions are logical instructions, because they logically shift the bits to the left or right and fill the empty spaces.

An arithmetic shift preserves the number’s sign.

**iii. SAL/SAR**

SAL & SAR Instructions:

SAL (Shift Arithmetic Left): Identical to SHL.

SAR (Shift Arithmetic Right): Performs a right arithmetic shift on the destination operand.

**iv. ROL,ROR, RCR, RCL**

ROL (Rotate) Instruction: Shifts each bit to the left.

The highest bit is copied into both the Carry flag & into the lowest bit.

No bits are lost.

ROR (Rotate Right) Instruction: Shifts each bit to the right.

The lowest bit is copied into both the Carry flag & into the highest bit.

No bits are lost.

RCL (Rotate Carry Left): Shifts each bit to the left.

Copies the Carry flag to the least significant bit.

Copies the most significant bit to the Carry flag.

RCR (Rotate Carry Right): Shifts each bit to the right.

Copies the Carry flag to the most significant bit.

Copies the least significant bit to the Carry flag.

**v. SHLD/SHRD**

SHLD Instruction: Shifts a destination operand a given number of bits to the left.

The bit positions opened up by the shift are filled by the most significant bits of the source operand.

The source operand is not affected.

Syntax: SHLD destination, source, count

Operand Types: SHLD reg16/32, reg16/32, imm8/CL, SHLD mem16/32, reg16/32, imm8/CL

SHRD Instruction: Shifts a destination operand a given number of bits to the right.

The bit positions opened up by the shift are filled by the least significant bits of the source operand.

The source operand is not affected.

Syntax: SHRD destination, source, count

Operand Types: SHD reg16/32, reg16/32, imm8/CL, SHLD mem16/32, reg16/32, imm8/CL

**vi. STC, CLC**

STC: Sets the carry flag (Carry Flag = 1)

CLC: Clears the carry flag (Carry Flag = 0)

**2. APP**

**i. Fast multiply, fast divide (binary)**

Fast Multiplication: Shifting left 1 bit multiplies a number by 2.

Shifting left n bits multiplies the operand by 2n.

For example, 5 \* 22 = 20

Shifting right n bits divides the operand by 2n.

**ii. Fast moving graphic images (shifting multiple double words)**

Programs sometimes need to shift all bits within an array, as one might when moving a bitmapped graphic image from one screen location to another.

The following shifts an array of 3 doublewords 1 bit to the right:

.data

ArraySize = 3

array DWORD ArraySize DUP(99999999h) ; 1001 1001...

.code

mov esi,0

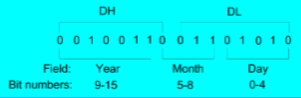
shr array[esi + 8],1 ; high dword

rcr array[esi + 4],1 ; middle dword, include Carry

rcr array[esi],1 ; low dword, include Carry

**iii. Masking – Isolating bit string**

The MS-DOS file date field packs the year, month, and day into 16 bits:



Isolate the Month field:

Mov ax, dx ; make a copy of DX

Shr ax, 5 ; shift right 5 bits

And al, 00001111b ; clear bits 4 – 7

Mov month, al ; save in month variable

**3. Multiplication and Division**

**i. MUL/IMUL (signed/unsigned)**

MUL (Unsigned Integer Multiplication) Instruction: In 32-bit mode, MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.

Instruction Formats: MUL r/m8, MUL r/m16, MUL r/m32.

IMUL (Signed Integer Multiplication) Instruction: Multiplies an 8-, 16-, or 32-bit signed operand by either AL, AX, or EAX.

Preserves the sign of the product by sign-extending it into the upper half of the destination register.

**ii. DIV/IDIV**

DIV (Unsigned Integer Division) Instruction: Performs 8-, 16-, or 32-bit division on unsigned integers.

A single operand is supplied (register or memory operand), which is assumed to be the divisor.

Instruction Formats: DIV reg/mem8, DIV reg/mem16, DIV reg/mem32.

IDIV (Signed Integer Division) Instruction: Performs signed integer division; same syntax & operands as DIV instruction.

Signed integers must be sign-extended before division takes place.

Fill high byte/word/doubleword with a copy of the low byte/word/doubleword’s sign bit.

For example, the high byte contains a copy of the sign bit from the low byte.

**iii. CBW/CWD/DWQ – extensions for instructions**

CBW, CWD, CDQ Instructions: Provide important sign-extension operations.

CBW (Convert Byte to Word): Extends AL into AH.

CWD (Convert Word to Doubleword): Extends AX into DX.

CDQ (Convert Doubleword to Quadword): Extends EAX into EDX.

**iv. Interpretation of the Carry/Overflow Flag**

MUL/IMUL:

The Carry Flag is set if the upper half of the product contains significant digits.

The Overflow Flag is set if the upper half of the product contains a sign extension of the lower half (only matters in the IMUL instruction).

DIV/IDIV:

The Carry Flag is irrelevant for division instructions.

The Overflow Flag is set if a division instruction caused the sign to flip.

**v. When an overflow occurs**

A divide overflow occurs when the result of a DIV/IDIV instruction is too large for a register and cannot be stored properly.

**vi. How to handle divide by zero**

**1. Interrupt and overflow – INTO**

Use jz to check if the divisor is 0; if it is, jump past the DIV/IDIV instruction so that the divide by zero error does not occur.

Use the cmp instruction to check if the divisor is too big and will create an overflow error (meaning the result is too large for the destination register).

**4. Arithmetic Expressions**

**i. Carry/Overflow flag**

The Carry Flag is set if the result of an addition/multiplication instruction cannot be properly represented in the destination register as an unsigned value.

The Overflow Flag is set if the result of an addition/multiplication instruction cannot be properly represented in the destination register as a signed value.

**5. Extended precision**

**i. ADC**

ADC (Add with Carry) Instruction: Adds both a source operand and the contents of the Carry flag to a destination operand.

Operands are binary values; same syntax as ADD, SUB, etc.

**ii. SBB**

SBB (Subtract with Borrow) Instruction: Subtracts both a source operand & the value of the Carry flag from a destination operand.

Syntax: Same as for the ADC instruction.

**6. ASCII**

**i. What is ASCII decimal**

ASCII Decimal: A number using ASCII Decimal representation stores a single ASCII digit in each byte.

**ii. AAA/AAS**

AAA (ASCII Adjust after Addition) Instruction: Adjusts the binary result of an ADD or ADC instruction. It makes the result in AL consistent with ASCII decimal representation.

The Carry value, if any, ends up in AH.

AAS (ASCII Adjust after Subtraction) Instruction: Adjusts the binary result of any SUB or SBB instruction. It makes the result in AL consistent with ASCII decimal representation.

It places the Carry value, if any, in AH.

**7. BCD**

**i. What are packed and unpacked BCD**

Binary-Coded Decimal:

Binary-coded decimal (BCD) integers use 4 binary bits to represent each decimal digit.

A number using unpacked BCD representation stores a decimal digit in the lower four bits of each byte.

Packed Decimal integers store two decimal digits per byte.

**ii. Unpacked BCD ops—AAM, AAD**

AAM (ASCII Adjust after Multiplication) Instruction: Adjusts the binary result of a MUL instruction. The multiplication must have been performed on unpacked BCD numbers.

AAD (ASCII Adjust before Division) Instruction: Adjusts the unpacked BCD dividend in AX before a division operation.

**iii. Packed BCD ops--DAA/DAS**

DAA (Decimal Adjust after Addition) Instruction: Converts the binary result of an ADD or ADC operation to packed decimal format.

The value to be adjusted must be in AL.

If the lower digit is adjusted, the Auxiliary Carry flag is set.

If the upper digit is adjusted, the Carry flag is set.

DAS (Decimal Adjust after Subtraction) Instruction: Converts the binary result of a SUB or SBB operation to packed decimal format.

The value must be in AL.

**8. Loops and Procedures**

**i. JA/JG**

JA: Jump if above (if leftOp > rightOp) for unsigned comparisons.

JG: Jump if greater (if leftOp > rightOp) for signed comparisons.

**ii. JB/JL**

JB: Jump if below (if leftOp < rightOp) for unsigned comparisons.

JL: Jump if less (if leftOp < rightOp) for signed comparisons.

**Excluded: Repeat, .while,.If, and .else directives**

**EXAM 2 PRACTICE QUESTIONS**

1. Let ESP = 8 and AX = 4 and the word at the top of the stack has the value 000A. After the execution of POP AX, which of the following is true?
   1. ESP = 6, AX = 6
   2. ESP = 7, AX = A
   3. ESP = A, AX = 4
   4. ESP = A, AX = A
   5. ESP = 6, AX = A
2. The RET instruction transfers control to the calling program at an address:
   1. Of the corresponding Call instruction
   2. Of the instruction immediately preceding the Call instruction
   3. Of the instruction immediately succeeding the Call instruction
   4. Which is EIP – 2
   5. Which is EIP – 4
3. Registers, which are altered within a procedure, are usually pushed at the beginning and popped at the end of the procedure, except those registers holding what kind of value?
   1. Inputs to the procedure
   2. Inputs but not outputs from the procedure
   3. Outputs from the procedure
   4. Outputs but not inputs to the procedure
   5. None of the above
4. The order of pushing registers at the beginning and popping registers at the end of a procedure should be:
   1. The same
   2. The opposite
   3. The order does not matter as the result is the same
5. An assembly language library can comprise of:
   1. Source code of only one subroutine
   2. Source code of one or more subroutines
   3. Object code of only one subroutine
   4. Object code of one or more subroutines
   5. None of the above
6. A simple nested loop using loop instructions for inner and outer loops can be created by:
   1. Pushing/Popping ecx within the outer loop
   2. Pushing/Popping ecx within the inner loop
   3. Pushing ecx within the outer loop and popping ecx within the inner loop
   4. Either A or B
   5. Both A and B
7. In the code below, the following jump takes place.

Mov ax, 2452h

Cmp ax, F46Eh

JA Target

* 1. TRUE
  2. FALSE

1. In the following code, the jump to the label Target takes place.

Mov ax, 2452h

Cmp ax, A46Eh

JG Target

* 1. TRUE
  2. FALSE

1. After the execution of the following code AX is:

Mov ax, 5

Mov bx, 4

Cmp ax, bx

* 1. -1
  2. 1
  3. 9
  4. 5
  5. 4

1. After the execution of the following code, AX is:

Mov ax, 2

Mov bx, 0

Test ax, bx

* 1. 0
  2. 1
  3. 2

1. Choose an instruction to be placed at label HERE, that would convert AL into an ASCII character:

Mov al, 4h

HERE: \_\_\_\_\_

* 1. Xor al, 30h
  2. And al, 30h
  3. Or al, 30h
  4. Sub al, 30h

1. Choose the best answer below.
   1. DIV is for unsigned and IDIV is for signed
   2. DIV is for general division whereas IDIV is used for fast integer division
   3. DIV is for signed and IDIV is for unsigned
   4. All of the above
   5. None of the above
2. What purpose does CBW serve in division?
   1. To sign extend quotient after division
   2. To sign extend divisor before division
   3. To sign extend remainder after division
   4. To sign extend dividend before division
   5. All of the above
3. What is the value of DX:AX after the following instructions execute?

Mov ax, 0

Mov dx, 1

Sub ax, 1

Sbb dx, 0

* 1. Fffe:ffff
  2. 0000:ffff
  3. Ffff:ffff
  4. Ffff:0000
  5. Fffe:0000

1. What might be the use of the above sequence of instructions (in the previous question) in a 16-bit computer?
   1. Extended precision arithmetic
   2. Financial calculations
   3. Astronomical calculations
   4. All of the above
   5. None of the above
2. What is the value of dl after the execution of the following program fragment?

Mov dl, -84

Sar dl, 2

* 1. -42
  2. -21
  3. 21
  4. 10
  5. -20

1. Which of the following is correct after the execution of the following instructions?

Mov al, 48d

Mov bl, 4

Imul bl

* 1. AX = 00C0, OF = 1
  2. AX = 00C0, OF = 0
  3. AX = 0192, OF = 1
  4. AX = 00C0, CF = 1

1. What will be the value of the register AL (in hex) after the following instructions have executed?

Mov al, 23h

Add al, 49h

DAA

* 1. 6Ch
  2. 72h
  3. 72d
  4. 62h
  5. 62d

1. Select the correct answer after the execution of the following:

Mov ax, 1000h

Mov bl, 10h

Div bl

* 1. AL = 10h, AH = 0
  2. AL = 100h, AH = 0
  3. AL = 0h, AH = 10h
  4. Divide overflow
  5. None of the above

1. Compared to regular 2’s complement, in BCD:
   1. Decimal addition is easier
   2. Scaling by a factor of 10d is easier
   3. Displaying characters is easier
   4. A and C
   5. A, B and C

**ANSWERS TO EXAM 2 PRACTICE QUESTIONS**

1. D.
2. C.
3. C.
4. B.
5. D.
6. A.
7. FALSE
8. A.
9. D.
10. C.
11. C.
12. A.
13. D.
14. B.
15. D.
16. B.
17. A.
18. B.
19. D.
20. E.

**MATERIAL AFTER EXAM 2**

**Chapter 8 – Advanced Procedures**

**1) Stack Frame**

Stack Frame: Also known as an Activation Record.

Area of the stack set aside for a procedure’s return address, passed parameters, saved registers, & local variables.

Created by the following steps:

Calling program pushes arguments on the stack & calls the procedure.

The called procedure pushes EBP on the stack, & sets EBP to ESP.

If local variables are needed, a constant is subtracted from ESP to make room on the stack.

**2) Pass by value/pass by reference**

Passing Arguments by Value:

Push argument values on stack.

Use only 32-bit values in protected mode to keep the stack aligned.

Call the called-procedure.

Accept a return value in EAX, if any.

Remove arguments from the stack if the called-procedure did not remove them.

Passing by Reference:

Push the offsets of arguments on the stack.

Call the procedure.

Accept a return value in EAX, if any.

Remove arguments from the stack if the called procedure did not remove them.

Passing an Array by Reference:

The ArrayFill procedure fills an array with 16-bit random integers.

ArrayFill can reference an array without knowing the array’s name.

The calling program passes the address of the array, along with a count of the number of array elements.

**3) Ret and ret n**

Returns from subroutine.

Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.

Syntax: RET, RET n

Optional operand n causes n bytes to be added to the stack pointer after EIP (or IP) is assigned a value.

**4) LOCAL**

Only statements within subroutine can view or modify local variables.

Storage used by local variables is released when subroutine ends.

Local variable name can have the same name as a local variable in another function without creating a name clash.

Essential when writing recursive procedures, as well as procedures executed by multiple execution threads.

LOCAL Directive: Declares a list of local variables.

Immediately follows the PROC directive.

Each variable is assigned a type.

Syntax: LOCAL varlist

**5) LEA vs Offset**

LEA Instruction: Returns offsets of direct & indirect operands.

OFFSET operator only returns constant offsets.

LEA required when obtaining offsets of stack parameters & local variables.

**6) Enter and Leave implementation**

ENTER Instruction: Creates stack frame for a called procedure.

Pushes EBP on the stack.

Sets EBP to the base of the stack frame.

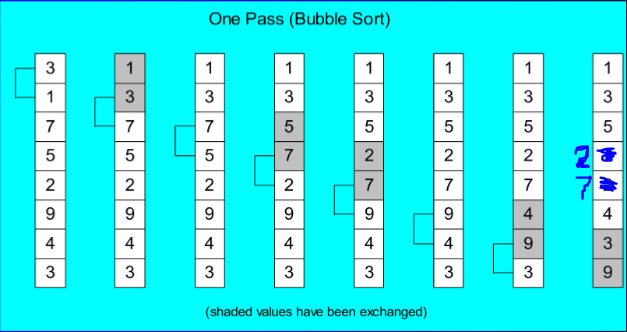
Reserves space for local variables.

LEAVE Instruction: Terminates the stack frame for a procedure.

**7) Bubble Sort**

Bubble Sort: A simple sorting algorithm that works well for small arrays

Each pair of adjacent values is compared, and exchanged if the values are not ordered correctly:



Implementation:

BubbleSort PROC USES eax ecx esi,

pArray:PTR DWORD,Count:DWORD

mov ecx,Count

dec ecx ; decrement count by 1

L1: push ecx ; save outer loop count

mov esi,pArray ; point to first value

L2: mov eax,[esi] ; get array value

cmp [esi+4],eax ; compare a pair of values

jge L3 ; if [esi] <= [edi], skip

xchg eax,[esi+4] ; else exchange the pair

mov [esi],eax

L3: add esi,4 ; move both pointers forward

loop L2 ; inner loop

pop ecx ; retrieve outer loop count

loop L1 ; else repeat outer loop

L4: ret

BubbleSort ENDP

**Chapter 9 – Strings**

**1) MOVSB/W/D**

The MOVSB, MOVSW, and MOVSD instructions copy data from the memory location pointed to by ESI to the memory location pointed to by EDI.

ESI and EDI are automatically incremented or decremented:

MOVSB increments/decrements by 1

MOVSW increments/decrements by 2

MOVSD increments/decrements by 4

**2) CMPSB/W/D**

The CMPSB, CMPSW, and CMPSD instructions each compare a memory operand pointed to by ESI to a memory operand pointed to by EDI.

CMPSB compares bytes

CMPSW compares words

CMPSD compares doublewords

**3) SCASB/W/D**

The SCASB, SCASW, and SCASD instructions compare a value in AL/AX/EAX to a byte, word, or doubleword, respectively, addressed by EDI.

Useful types of searches:

Search for a specific element in a long string or array.

Search for the first element that does not match a given value.

**4) STOSB/W/D**

The STOSB, STOSW, and STOSD instructions store the contents of AL/AX/EAX, respectively, in memory at the offset pointed to by EDI.

**5) LODSB/W/D**

LODSB, LODSW, and LODSD load a byte or word from memory at ESI into AL/AX/EAX, respectively.

**6) Base IDX**

A base-index operand adds the values of two registers (called base and index), producing an effective address. Any two 32-bit general-purpose registers may be used. (Note: esp is not a general-purpose register).

Base-index operands are great for accessing array or structures. (A structure groups together data under a single name)

**7) Base IDX, DISPL**

A base-index-displacement operand adds base and index registers to a constant, producing an effective address. Any two 32-bit general-purpose registers can be used.

Common Formats: [base + index + displacement] OR displacement[base + index]

**Interrupts**

**1) Reset trap**

Involves using the AND instruction to reset the Trap Flag (TF) to 1

PUSHF (push flags on stack) then change bit in 8th position

AND WORD PTR[BP], 0FEFFh

POPF (restore flags)

The Trap Flag is used to slow the system to a single-step execution (debugging).

**2) INT21**

Reads a block of bytes

Can be interrupted by CTRL-Break (^C)

Ex: Read string from keyboard

**3) NMI**

A (non-maskable) INT request pin cannot be ignored by the processor.

Associated with high priority tasks.

**4) INTR** – INT request pin

**5) INTA** – INT acknowledge pin

**6) Vector-table** – Holds all interrupt instructions

**7) Software interrupts vs hardware interrupts**

Software Interrupts: Generated internally by the currently running process.

Hardware Interrupts: Driven by hardware devices (external interrupt requests).