

Lecture #1 (LAB)

Wednesday, 23 August 2023 9:19 am

- we use HLL such as C++
- switch is made from transistors that's why only two options on and off 1 and 0
- machine language uses binary
- HLL-> assembly language ->machine language -> given to hardware
- HLL to assembly code is done by compiler
- assembly code to object code is done by assembler
- object code to machine code is done by linker and then goes to hardware
- Registers are used few are general purpose and few have special purpose.
- Each register is 16 bit that is 2 bytes [general purpose registers]
- Segment register: memory is divided in different segments eg some memory is save for only variables
- Control registers: such as flags which are 1 bit.
- Data registers: there are total four of such registers names as AX, BX, CX, DX. Each is 2 bytes. These are also known as general purpose registers.
- MOV == move syntax: MOV destination register, value
- Each register for eg AX can be further divided into AL and AH. Each has 8 bit separately.
- It is comparatively easier to access data from register than memory.
- MOV= move, ADD= add, CMP= compare

▪

- Advantages of HLL
- Program development is faster
- Maintenance is easier
- Programs are portable

- Assembly language has one to one correspondence to machine language

- Advantages of Assembly Language
- Accessibility to system hardware
 - Useful for implementing system software
 - Useful for small embedded system application
- Space and time efficiency
 - Tuning program performance
 - Writing compact code

▪ MASM

LECTURE #1(Theory)

Friday, 25 August 2023 2:53 pm

- ▶ Computer architecture refer to those objects that are visible to the programmer and these attributes have a direct impact on the execution of the program.
- ▶ Computer organization refers to the operational units and their interconnections that realize the architectural specifications
- ▶ Convert from base₂ to base₁₀
 - $(10011)_2 = 1*2^0 + 1*2^1 + 0*2^2 + 0*2^3 + 1*2^4$
 - $= 1 + 2 + 16 = 19$
- ▶ Convert from base₁₆ to base₁₀
 - $8CE_{16} = 14*16^0 + 12*16^1 + 8*16^2$
 - $= 2048 + 192 + 14 = 2254$
- ▶ Convert from base₂ to base₁₆
 - Make groups of 4 starting from the left side
 - Write down values of each group of 4
- ▶ Convert base₁₀ to base₂

2	19	
2	9	1
2	4	1
2	2	0
2	1	0
	0	1

$(10011)_2$

- ▶ High-Level Language
 - Used by the programmers to write programs
 - Looks more like natural language
 - Example C++, C#
 - Compiler translates HLL into machine code
- ▶ Machine Language
 - Understood by the computer
 - Represented in for of 0s and 1s
 - Each ML instruction contains an op code and operands
 - Numeric instructions and operands that can be stored in memory and are directly executed by the computer system
- ▶ Assembly Language
 - Uses instruction mnemonics that have one to one correspondence with machine language

- An instruction is a symbolic representation of a single machine instruction. It consists of

Label	Always optional
Mnemonic	Always required
operands	Required by some instructions
Comment	Always optional

- ▶ Assembler converts the source code programs into machine language i.e. object file
- ▶ Linker joins two or more object files and produces a single executable file
- ▶ Byte= 8 bits; word= 16 bits ; doubleword= 32 bits ; quadword= 64
- ▶ Unsigned byte= 0-255
- ▶ Unsigned word= 0-65535
- ▶ Unsigned doubleword= 0-4294967295 (0 to $2^{32}-1$)
- ▶ Unsigned quadword= 0 to ($2^{64}-1$)
- ▶ Representation of negative numbers
 - Signed magnitude notation : the left most bit represents the sign integer.
 - Excess notation:
 - Two's complement
- ▶ Carry and overflow
 - Carry is important when adding/subtracting unsigned integers
 - Indicates that unsigned sum is out of range
 - Either <0 or >maximum unsigned n-bit value
- ▶ Overflow is important
 - Adding or subtracting signed integers
 - Indicates that the signed sum is out of range
- ▶ Overflow occurs when
 - Adding two +ve numbers and the sum is negative
 - Adding two -ve numbers and the sum is positive

Processor consists of

- ▶ Datapath
 - ALU
 - Performs arithmetic and logical instructions
 - Registers
- ▶ Control unit
 - Generates control signals required to execute instructions
- ▶ Clock
 - Synchronizes process and bus operations
 - Clock cycle= clock period= $1/\text{clock rate}$
 - Clock rate= clock frequency/cycles per second
 - 1 Hz=1 cycle/sec

- ▶ 32 bit general purpose
 - Eax
 - EBX
 - ECX
 - EDX
 - EBP-base pointer==>used to access data of stack and other segments
 - ESP-stack pointer==> used for accessing stack segment
 - ESI-source index==>point to memory location in the data segment
 - EDI-destination index==>point to destination address
- ▶ 16 bit segment registers
 - CS- code segment==>holds base address of code segment
 - ES-extra segment
 - SS-stack segment==>holds base address of stack segment
 - DS-data segment==>holds base address of data segment
 - FS
 - GS
- ▶ Extended instruction pointer
 - Used to access instructions
 - Works along with EIP which stores the base address of the next instruction and EIP contains the offset
- ▶ Memory is divided into segments
- ▶ Stack, code and data
- ▶ Registers= 14; divided in groups
- ▶ General purpose: AX,BX,CX,DX (accumulator, base, counter, data)
- ▶ MOV is equivalent to assign in C++
 - MOV destination address, value to be stored
 - MOV AL,CX this will give error as the size of the registers don't matter
 - The size of the registers must be same if you want to move data b/w them
- ▶ MOV AL,5
- ▶ MOV BL,4
- ▶ ADD AL,BL

First 32 characters of ASCII table are used for control form 00-1F

- ▶ Defining string
 - G BYTE "good mornng",0 //takes 13 bytes
- ▶ DUP operator
 - X byte 2 DUP(0) //stores 0 0
 - Z byte 3 dup(2 dup(12,13)); //stores 12 13 12 13 12 13 12 13 12 13
 - Z byte 10 ,20, 2 DUP(0),10 //stores 10 20 0 0 10
- ▶ Symbolic constant
 - Don't reserve storage variables do
 - Value of symbolic constant doesn't change during run time
 - Name EQU expression or name EQU <text>
 - Unlike EQU directive the = directive can redefine a symbol any

number of times. It can associate a symbolic name with an integer expression but not with text

- Name= expression
- Current Location counter
 - The symbol \$ is the current location counter that returns the offset of the current location
 - List byte 1,20,23,11
 - Listsize=(\$-list) //returns the size of the list

STRING MANIPULATION

```
Str1 word "lab 03", '$'
Mov al, type str1 ;al=2 bytes
Mov bl, lengthof str ;bl=7 bytes
Mov cl, sizeof str1 ;cl=14 (2*7)
Mov eax, offset str1 ; gives starting address of str1
String cant be word as every character is a byte. Therefore string
will always be a byte
Length tells number of elements
Size tells number of bytes ==> number of elements(LENGTHOF) * TYPE
(datatype of elements)
Type tells size in bytes
```

LAB 4

\$ is equivalent to null character
Value of null character can't be changed during the course of program

```
.data
Str1 db "lab", '$'
Strlen db ?
Strlen= $-str1           //must be done soon after declaration of str1
.data                    //what is the use of textequ
```

Definition of array
Arr1 db 1,2,3,4,5

Direct addressing arr1[index_of elements]
Str1 dd 12345678h

Mov a;, byte ptr var1		//78564321	
-----------------------	--	------------	--

- Structure: the way in which the components are interrelated
- Function: the operation of each individual components as part of the structure
- Control Unit: controls the operation of the CPU; synchronizes operations
- Arithmetic and logic unit(ALU): performs data processing instructions
- Registers: Provides storage internal to the CPU
- CPU interconnections: provides communication among CU, ALU and registers
- 8086: 16 bit machine; it is the first appearance of x86 architecture

•

► Arithmetic instructions

◦ MUL operandl

- Operand can be 8 bit and is multiplies wit the contents placed in al, and the result is stored in AX
- Operand can be 16 bit and is multiplied with contents of ax, and is placed in AX or DX
- Operand can be of 32 bits and is multiplied with the contents of DX:AX results will be stored in EDX:EAX

◦ DIV operandl

- AL stores quotient and AH stores remainder
- The number is stored in AX
- If operand is 16b bits quotient is in AX and remainder is in DX
- When operand in 32 bits quotient is stored in EAX and remainder is stored in EDX

► LABELS

- Identifier followed by a colon
- Names suffixed with : are symbolic labels
Way to tell the assembler that those locations have symbolic names

► Operand types

- Immediate: these are literal expression
- Register: name of the register
- Memory: references a location in the memory

OPERAND	DESCRIPTION
Reg8	8 bit register
Reg16	16 bit register
Reg32	32 bit register
reg	General register
Sreg	16 bit segment register
Imm	8,16,32 bit immediate value
Imm8	8 bit immediate value
Imm16	16 bit immediate value
Imm32	32 bit immediate value
Reg/mem8	8 bit operand which can be 8 bit register or memory byte
Reg/mem16	16 bit operand which can be 16 bit register or memory word
Reg/mem32	32 bit operand which can be 32 bit register or memory double word
Mem	8,16,32 bit memory

► Direct memory operands

- Mov al,[var1]

► Zero Extension

- MOV cannot directly copy data from a smaller operand to larger
 - Mov bl,10001111b
 - Mov ax,bl //gives error as bl is 8 bits and ax is 16 bits
- MOVZX instruction fills the upper half of the destination with zeros

- Mov bl,10001111b
 - Movzx ax,bl
- MOVZX instruction move with sign extend. Fills upper half of the destination with the sign bit of the operand
- XCHG exchanges values of two operands. At least one operand must be a register and immediate values are not allowed
- Mov al,[array+1]
- ▶ INC AND DEC
 - INC increments
 - DEC decrements
- ▶ ADD
 - Add destination, source
- ▶ SUB
 - SUB destination, source
- ▶ NEG
 - Reverses the sign of the number by converting to its two's complement
 - NEG reg
 - NEG mem
- ▶ JMP and LOOP
 - Unconditional: no condition is involved and the control is transferred to new location
 - Conditional: transfers control only if a certain condition is true
 - JMP
 - Causes unconditional transfer
 - JMP destination
 - Top:
 - inc ax
 - jmp top
 - Conditional jumps
 - Jumps based on unsigned data
 - JE==> jump if equal
 - JE [LABEL]
 - JNE==> jump if not equal
 - JNE [LABEL]
 - JCXZ jump if CX=0
 - JECXZ jump if ECX=0
 - CMP instruction
 - CMP operand1, operand2

JZ	Jump if 0	ZF=1
JNZ	Jump if not 0	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
Ja	If op1>op2	CF=0 and ZF=0
Jae	If op1>=op2	CF=0
Jnae	If op1 not>=ope2	CF=1
Jb	If op1<op2	CF=1
Jbe	If op1<=op2	CF=1 ZF=1

Jnb	If op1 not<op2	CF=0
Jnbe	If oop1 not<=op2	CF=0 and ZF=0

- ▶ LOOP instructions
 - Works according to the ECX counter
 - LOOP destination
 - The destination must be within -128 to 127

Flags

Tuesday, October 3, 2023 11:33 AM

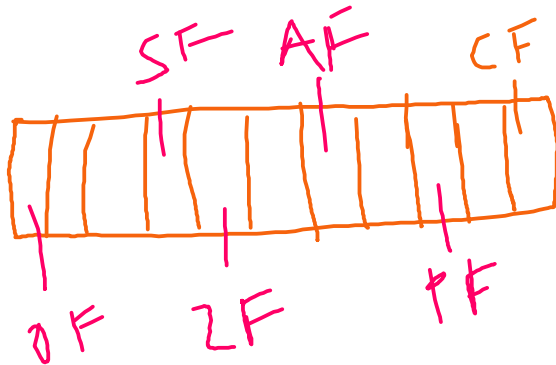
Carry flag

It is =1 if there is a Carry out from the MSB

It is =1 if there is a borrow into the MSB on subtraction

Parity flag

It is =1 if the lower byte has even number of 1s



Auxiliary flag

It is =1 when there is a Carry out from bit 3

Or when there is a borrow into bit 3 on subtraction

Zero flag

Is 0 when all terms 0

Is non-zero when non zero number

Overflow flag

Set to 1 for signed numbers

CF is set to 1 for unsigned numbers

If you add two +ve and the answer is -ve then overflow has occurred

If you add two -ve and answer is +ve then overflow has occurred

A -ve added by +ve will not generate overflow

If the carry into and out of the msb don't match then overflow has occurred

How the Processor Determines that Overflow Occurred

Many instructions can cause overflow; for simplicity, we'll limit the discussion to addition and subtraction.

Unsigned Overflow

On addition, unsigned overflow occurs when there is a carry out of the msb. This means that the correct answer is larger than the biggest unsigned number; that is, FFFFh for a word and FFh for a byte. On subtraction, unsigned overflow occurs when there is a borrow into the msb. This means that the correct answer is smaller than 0.

Signed Overflow

On addition of numbers with the same sign, signed overflow occurs when the sum has a different sign. This happened in the preceding example when we were adding 7FFFh and 7FFFh (two positive numbers), but got FFFEh (a negative result).

Subtraction of numbers with different signs is like adding numbers of the same sign. For example, $A - (-B) = A + B$ and $-A - (+B) = -A + -B$. Signed overflow occurs if the result has a different sign than expected. See example 5.3, in the next section.

In addition of numbers with different signs, overflow is impossible, because a sum like $A + (-B)$ is really $A - B$, and because A and B are small enough to fit in the destination, so is $A - B$. For exactly the same reason, subtraction of numbers with the same sign cannot give overflow.

Actually, the processor uses the following method to set the OF: If the carries into and out of the msb don't match—that is, there is a carry into the msb but no carry out, or if there is a carry out but no carry in—then signed overflow has occurred, and OF is set to 1. See example 5.2, in the next section.

MOV AL, DOLLARS
MOV AH, CENTS

Example 10.9 Suppose the following data are declared:

```
.DATA  
A DW 1234h  
B LABEL BYTE  
  DW 5678h  
C LABEL WORD  
C1 DB 9Ah  
C2 DB 0BCh
```

Tell whether the following instructions are legal, and if so, give the value moved.

Instruction

- a. MOV AX, B
- b. MOV AH, B
- c. MOV CX, C
- d. MOV BX, WORD PTR B
- e. MOV DL, WORD PTR C
- f. MOV AX, WORD PTR C1

Solution:

- a. illegal—type conflict
- b. legal, 78h
- c. legal, 0BC9Ah
- d. legal, 5678h
- e. legal, 9Ah
- f. legal, 0BC9Ah

override

In register

JMPS+AND+OR+XOR

Monday, October 9, 2023 8:06 PM

- ▶ AND Instructions
 - AND instructions can happen between memory and register, memory and immediate value, register and immediate value and register to register
 - Size of operands must be same
 - AND will always clear overflow and carry flag may manipulate the other flags too.
- ▶ OR Instruction
 - Clears carry and overflow flag
 - Modifies other flags consistent with the final answer
- ▶ XOR Instruction
 - Performs Boolean XOR operation
 - Returns 1 if the values are different eg 01 and 10
 - Returns 0 if the values are same eg 00 and 11
 - Clears overflow and carry flag
 - Changes other flags consistent with the final answer
- ▶ NOT Instruction
 - Inverts the bits
 - Inverting memory and registers is possible
 - No flags are affected
- ▶ TEST Instruction
 - Performs AND operation however the value of destination is not changed
 - Always clears overflow and carry
- ▶ CMP Instruction

CMP RESULTS(unsigned)	ZF/CF
Destination < source	CF=1
Destination > source	
Destination==source	ZF=1

CMP RESULTS(SIGNED)	ZF/CF
Destination<source	SF≠OF
Destination> source	SF=OF
Destination=source	ZF=1

- ▶ STC and CLC
 - Setting carry flag
 - Use STC
 - Clear
 - Use CLC
 - To set overflow flag
 - Add two positive number which produce negative number
 - To clear it OR with 0

Table 6-2 Jumps Based on Specific Flag Values.

Mnemonic	Description	Flags / Registers
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

Table 6-3 Jumps Based on Equality.

Mnemonic	Description
JE	Jump if equal (<i>leftOp</i> = <i>rightOp</i>)
JNE	Jump if not equal (<i>leftOp</i> ≠ <i>rightOp</i>)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0
JRCXZ	Jump if RCX = 0 (64-bit mode)

Unsigned Comparisons: The jumps in following table are only meaningful when comparing unsigned values. Signed operands use a different set of jumps.

Table 6-4 Jumps Based on Unsigned Comparisons.

Mnemonic	Description
JA	Jump if above (if $leftOp > rightOp$)
JNBE	Jump if not below or equal (same as JA)
JAЕ	Jump if above or equal (if $leftOp \geq rightOp$)
JNB	Jump if not below (same as JAЕ)
JB	Jump if below (if $leftOp < rightOp$)
JNAЕ	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if $leftOp \leq rightOp$)
JNA	Jump if not above (same as JBE)

Table 6-5 Jumps Based on Signed Comparisons.

Mnemonic	Description
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 $leftOp \geq rightOp$)
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 $leftOp \leq rightOp$)
JNG	Jump if not greater (same as JLE)

display-Lab_09

Wednesday, 1 November 2023

8:49 am

PROCEDURES

Add_1 proc

//

ret ; return

endp ;signifies the end of procedure

To call in the main procedure use call add_1

Procedures must be made between main PROC and endmain {in .code}

- Procedures with arguments

Add_1 uses ax,bx ;values of these registers are exactly copied

;could use variables as well and can't use immediate values

If you don't want to use variables then use general purpose registers such as ax,bx,cx,dx

MACROS

Doesn't take up memory

Add_1 macro ; name of the macro is add_1

// code

//code

endm

Macro has no return statement

Executes certain lines executes and returns to the previous instruction

Called in main proc

Doesn't require the use of word call just write the name of the macro

- Arguments with macros

Add_1 macro a1,a2,a3 ; macros with arguments

Mov ax,a1

///code

endmain

No need to initialize and no need to mention the data type

Main proc

Add_1 2,5,x here 2 and 5 are numbers stored in a1, a2 and the result is stored in a variable x created in data part

Endmain

Interrupt

- Getstdhandle proto, a1:dword ;used for read/write instructions
- writeconsoleA proto ;writes a character string to the console screen buffer beginning at the current cursor location
- ReadconsoleA proto
- For writing to console send -11
- For reading from console -10
- Such as invoke getstdhandle,-10 ;reads from console
- invoke getstdhandle,-11 ;writes

Integer Arithmetic

Saturday, 4 November 2023

9:19 pm

⊙ Logical and Arithmetic Shifts

- Logical shifts fills newly created bit with a 0 and moves the bit shifted to CF
- Arithmetic Shift: the sign bit is copied and then the shifting is performed

1	0	1	0	0	1	1	1
1	1	0	1	0	0	1	1

- The shifted bit is moved to the CF, above I have performed Right shift

SHL	SHR	SAL	SAR	SHLD	SHRD
-----	-----	-----	-----	------	------

- Can happen b/w reg,imm8 and mem,imm8 and mem,cl
- Left shift is bitwise multiplication and right shift is division (unsigned)
- SAL works in the same way as SHL
- SAR copies sign and then shifts
- SAR can be used to extend sign

⊙ Rotation

- Bits moved around in circular form

ROL	ROR	RCL	RCR
-----	-----	-----	-----

- ROL MSB is copied to CF and the value of MSB is moved to the LSB
- ROR the LSB is copied to the MSB and the value of LSB is moved to CF
- RCL copies value of CF to LSB and then moves the value of MSB to CF
- RCR copies the value of to MSB and then moves the value of LSB to CF

⊙ SHLD AND SHRD

- Shift left/right double SHLD dest,source,count
- This actually shifts count time in dest and source
- Can happen between reg16,reg16,cl/imm8 and mem16,reg16,cl/imm8 and same for 32 bits as well

⊙ Multiplication and division instructions

- The process is different for signed and unsigned.
- The MUL and IMUL instructions perform unsigned and signed integer multiplication
- The DIV is used for unsigned integers and IDIV performs signed integer division
- MUL is used for unsigned multiply:
 - MUL
 - Multiply and 8 bit operand by the AL register =>product stored in AX
 - Multiply a 16 bit operand by the AX register =>product stored in DX:AX
 - Multiply a 32 bit operand by EAX register==>EDX:EAX
 - The CF/OF is set if DX or EDX is not equal to zero which lets us know that the product is larger than size of AX or EAX
 - IMUL
 - Used for signed multiplication; preserves sign of the product
 - Extends the MSG of the lower half of the product to the upper half
 - Can have one, two, three operands

IMUL reg/mem8	AX=AL * reg/mem8
IMUL reg/mem16	DX:AX=AX*reg/mem16
IMUL reg,mem32	EDX:EAX=EAX* reg/mem32

- Two operand formats(32 bit mode): stores the product in the first operand, which must be a register. The second operand(multiplier)

can be a register, memory or immediate value

- IMUL reg16,reg/mem16
- IMUL reg16,imm8
- IMUL reg16,imm16
- IMUL reg32,imm8
- IMUL reg32,imm32
- IMUL reg16,reg/mem32
- After CF/OF= 0 then the upper half of the result is the sign extension of the lower half; otherwise CF/OF is 1

▪ DIV

- Used for unsigned division
- DIV reg,mem8 ; reg/mem16 ; reg/mem32
- Quotient and remainder have the same size as the divisor

<u>Dividend</u>	<u>Divisor</u>	<u>Quotient</u>	<u>Remainder</u>
AX	Reg/mem8	AL	AH
DX:AX	Reg/mem16	AX	DX
EDX:EAX	Reg/mem32	EAX	EDX

▪ Signed Integer Division

- CBW= used to convert byte to word extends the sign of AL into AH
- CWD used to convert word to double word extends the sign of AX into DX (MOVSX DX,AX)
- CDQ used to convert double word into quadword extends sign of EAX to EDX

▪ IDIV

- Performs signed division
- Before executing 8 bit division the dividend AX must be completely sign extended


ASCII and unpacked decimal

Monday, 6 November 2023 12:25 pm

- When taking input from user the user enters a string so supposed to perform operations like addition, subtraction, multiplication and division we have two options:
 - Convert both operands to binary and perform the operations
 - Perform the operation directly on the ASCII digits


AAA	Ascii adjust after addition
AAS	ASCII adjust after subtraction
AAM	ASCII adjust after multiplication
AAD	ASCII adjust after division

- ❖ AAA (this instruction is performed if AF==1 or lower nibble>9)
 - If the lower nibble of AL >9 then the lower nibble is incremented by 6 and the higher nibble is incremented by 1
 - AF is set to 1, CF =1

Instruction	Operands	Description
AAA	No operands	<p>ASCII Adjust after Addition. Corrects result in AH and AL after addition when working with BCD values.</p> <p>It works according to the following Algorithm:</p> <p>if low nibble of AL > 9 or AF = 1 then:</p> <ul style="list-style-type: none"> AL = AL + 6 AH = AH + 1 AF = 1 CF = 1 <p>else</p> <ul style="list-style-type: none"> AF = 0 CF = 0 <p> in both cases: clear the high nibble of AL.</p> <p>Example: MOV AX, 15 ; AH = 00, AL = 0Fh AAA ; AH = 01, AL = 05</p>

- ❖ AAS ()
 - Adjustment required when the subtraction generates negative result

AAS	No operands	<p>ASCII Adjust after Subtraction. Corrects result in AH and AL after subtraction when working with BCD values.</p> <p>Algorithm:</p> <p>if low nibble of AL > 9 or AF = 1 then:</p>
-----	-------------	---


		<ul style="list-style-type: none"> • $AL = AL - 6$ • $AH = AH - 1$ • $AF = 1$ <ul style="list-style-type: none"> • $CF = 1$
		
		<p>else</p> <ul style="list-style-type: none"> • $AF = 0$ • $CF = 0$ <p>in both cases: clear the high nibble of AL.</p> <p>Example: <code>MOV AX, 02FFh ; AH = 02, AL = 0FFh</code> <code>AAS ; AH = 01, AL = 09</code> <code>RET</code></p>

❖ ADC

- Add with carry adds source operand and the contents of carry flag to the destination operand
- ADC reg,reg
- ADC mem,reg
- ADC reg,mem
- ADC mem,imm
- ADC reg,imm


❖ SBB

- Subtract with borrow
- Subtracts both source operand and the value of carry flag from the destination operand

		<p>Decrease CX, jump to label if CX not zero and ZF = 0.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• CX = CX - 1• if (CX > 0) and (ZF = 0) then<ul style="list-style-type: none">○ jump <p>else</p> <div><ul style="list-style-type: none">○ no jump, continue</div> <p>Example:</p> <pre>; Loop until '7' is found, ; or 5 times.</pre> <pre>include 'emu8086.inc' ORG 100h MOV SI, 0 MOV CX, 5 label1: PUTC '*' MOV AL, v1[SI] INC SI ; next byte (SI=SI+1). CMP AL, 7 LOOPNZ label1 RET v1 db 9, 8, 7, 6, 5</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														

LOOPZ	label	<p>Decrease CX, jump to label if CX not zero and ZF = 1.</p> <p>Algorithm:</p> <ul style="list-style-type: none"> • CX = CX - 1 • if (CX \neq 0) and (ZF = 1) then <ul style="list-style-type: none"> ◦ jump else <ul style="list-style-type: none"> ◦ no jump, continue <p>Example:</p> <pre>; Loop until result fits into AL alone, ; or 5 times. The result will be over 255 ; on third loop (100+100+100),</pre>
-------	-------	--



		<p>; so loop will exit.</p> <pre>include 'emu8086.inc' ORG 100h MOV AX, 0 MOV CX, 5 label1: PUTC '*' ADD AX, 100 CMP AH, 0 LOOPZ label1 RET</pre> <div style="border: 1px solid black; padding: 2px; display: inline-block;">C Z S O P A</div>
--	--	---

Procedures-LAB 10

Wednesday, 15 November 2023 8:45 am

Input output make it single digit

When taken input from console(reading from console) it comes as a character so subtract from 48

and to write on console add 48

Task 3 ALWAYS Take odd numbers; number of lines taken from user

MACROS

Name_macro macro

//code

Endm

WITH PARAMETERS

Name_macro macro a1,a2

//code

Endm

Name_proc Proc

//code

Ret

Name_proc endp

WITH PARAMETERS

Name_proc Proc uses ax

//code

Ret

Name_proc endp

MAIN CODE

Call proc_name

; leaves the current line and goes to the procedure and executes its instructions

Macro_name

;just copies that line to main and executes then for example if L1 is defined in main and in macro then it will give redefinition error

;used only when a set of lines is repeated a number of times

WHERE TO DEFINE MACROS AND PROCEDURES

Procedures are defined below main while macros are defined above main --general practice

Procedures takes memory while macros don't

Can make nested macros and procedures

CACHE

Wednesday, 6 December 2023

10:40 pm

Cache hit: when the data is found in cache

Cache Miss: when the data is not found in L1, L2

Locality

- Temporal locality: locality in time, if data is used recently and likely to use again
- Copies nearby accessed data into cache
- | | |
|-----------------|--|
| How to exploit: | Keep recently accessed data in higher levels of memory hierarchy |
|-----------------|--|
- Spatial Locality: locality in space, if data is used recently keep data nearby
- How to exploit: when access data bring nearby data into higher levels of memory hierarchy too
- Copies neighbouring data into cache too
- Eg would be supposed u accessed array's 0th index it then copies the next few array elements to cache
- Miss Rate= number of misses/ number of total memory accesses
- Miss rate= 1- hit rate
- Hit Rate= number of hits/ number of total memory accesses
- Hit Rate= 1- Miss rate

Cache Terminologies

- Cache Capacity [C]: total number of bytes that can be stored in cache
- The capacity decides how much data each block can store
- Cache lines/blocks: data transferred between memory and cache in blocks of fixed sizes
- EXAMPLE: b=32 bits; C 64 bits
 - Block capacity is 4 bytes and number of total blocks is 8
- | | | | | |
|--------------|----|----|----|----|
| Block 0 = 4B | -- | -- | -- | -- |
| Block 1 = 4B | | | | |
- Offset bits= $\log_2(b \text{ [bytes]})$
 - CAPACITY: number of bytes in cache
 - BLOCK SIZE(b): bytes of data brought into cache at once
 - NUMBER OF BLOCKS (B=C/b)
 - DEGREE of ASSOCIATIVITY: number of blocks in a set
 - NUMBER OF SETS (S=B/N): each memory address maps to exactly one cache set
- One Way associative cache: 1 block per set
- Two way associative cache: 2 blocks per set
- Four way associative cache: 4 blocks per set
- Fully Associative: all cache in blocks in one set

EXAMPLE

b 4B

C 64B

N 1 way (direct mapped)

$$B=C/b=64/4=16$$

$$S=B/N=16/1=16$$

$$\text{Offset bits is } \log_2(b)=\log_2(4)=2$$

$$\text{Set/index bits}=\log_2(S)=\log_2(16)=4$$

0000				
0001				
...				
1111				

EXAMPLE

b 32B

C 128B

N 1(direct)

$B = C/b = 128/32 = 4$

S=B/N=4

Offset bits= $\log_2(32)=5$

Set/index bits= $\log_2(S)=2$

00								
----	--	--	--	--	--	--	--	--

Tag bits:

000000

111111(32)

11

Cache is the fastest memories available and at the same time they provide large memory size

There are three cache levels L1, L2 and L3

L1 is the fastest and L3 is the slowest

- **Labels in Procedures** are visible only within the procedure in which they are declared.
- In the following example, the label named *Destination* must be located in the same procedure as the JMP instruction:

```
jmp Destination
```

- It is possible to work around this limitation by declaring a *global label*, identified by a double colon (::) after its name:

```
Destination::
```