SYSTEM PROGRAMMING

-Gaurav Suryawanshi

1 MICROPROCESSOR

Masm: Microsoft Macro Assembler

To execute commands do

Name Command Operand

1.1 REGISTERS

1.1.1 Data Registers

Four 32 bit data registers are EAX,EBX,ECX & EDX.

Four 16 bit data registers are there namely: AX, BX, CX, DX where

AX: 16 Bits is comprised of

AH	AL
8 bits	8 bits

Similarly, BX (16 bits) is comprised of

BH	BL
8 bits	8 bits

Hence, in total

16 Bit	8 Bit Registers	Values	Remarks
AX	AH+AL	AH=AX/256 AL=AX%256 AX=AH*256+AL	**Accumulator**; Ax is preffered for Airthmetic and data transfer. I/O
BX	BH+BL	BH=BX/256 BL=BX%256	**Base** Serves as address for table lookup
CX	CH+CL	CH=CX/256 CL=CX%256	**Count** Looping
DX	DH+DL	DH=DX/256 DL=DX%256	**Data** Used in Mul/DIV. Used in I/O

1.1.2 Segment Registers

Four Segment Registers

Register name	Bits	Function
CS	16	**Code Segments**

		Stores address of Code segment
DS	16	Data Segments
SS	16	Stack Segments
ES	16	Extra Segment

1.1.3 Pointer And Index Register

All Registers are 16 Bits

Operand	Name	Remarks
SP	Stack Point	Used for accesing Stack Segment
BP	Base Pointer	Used for accessing data from stack segment
SI	Source Index	Used to point to memory location in the data segments addressed by DS
DI	Destination Index	Same as SI
IP	Instruction Pointer	Contains the offset address of the next instruction to be exectued

1.1.4 Flag Register

16 Bit Register containing Various Flags like status and control flags

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Operand	Name	Remarks
CF	Carry Flag	CF=1, If there is any carry out from MSB on addition/sub
PF	Parity Flag	PF=1, IF the low byte of a result has a even no's of 1 bit, otherwise 0
AF	Auxiliary Flag	AF=1, If there is a carry out from bit on addition/ subtractions otherwise AF=0
ZF	Zero Flag	ZF=1 for zero result otherwise ZF=0
SF	Sign Flag	SF=1 if MSB of result it 1 Otherwise SF=0
OF	Overflow Flag	OF=1 if sign overflow occurs Otherwise OF=0
DF	Direction Flag	DF=0, SI & DI proceed in increasing memory address DF=1, decreasing

2 VARIABLES

Execution:

VAR_NAME COMMAND VALUES

Command	Meaning	Bits
DB	Define Byte	8
DW	Define Word	16
DD	Define Double Word	32
DQ	Define QuadWord	64
DT	Define 10 bytes	80

```
Eg:   ALPHA   DB   4
      BY      DB   ? ;For Value Not initialized
      WRD     DW   2;Initialized to 2
      B_arr   DB   10H,20H,30H; Stored as B_arr, B_arr+1 & B_arr+2
      B_arr   DW   10H,20H,30H; Stored as B_arr, B_arr+2 & B_arr+4
      string  DB   'ABC'; A=string, B=String+1 …
      msq     DB   'Hello',ODH,OAH,'$'; H,E,L,L,O,ODH,OAH,$
```

2.1 CONSTANTS

Exectution:

VAR_NAME EQU VALUES

3 PROGRAM STRUCTURE

The Model Structure is as follows

```
.MODEL Small
.STACK 100H
.DATA
;Define Data
.CODE
somefn PROC
      ;Define fn
somefn ENDP
MAIN PROC

      ;Instructions
MAIN ENDP
END MAIN
```

3.1 .MODEL MEMORY_MODEL

Memory_model	Remarks
SMALL	Code in 1 segment Data in 1 Segment
MEDIUM	Code in multiple segments Data in 1
COMPACT	Code in 1 Data in multiple
LARGE	Code in multiple Data in multiple Array size max = 64kb
HUGE	Code in multiple Data in multiple Array size max >64kb

3.2 .DATA

Variables are declared here, as previously discussed

3.3 .STACK

.STACK SIZE

EG .STACK 100H

3.4 .CODE

For writing Code to solve any problem

4 PROCEDURES

Similar to the function in c++. Structure is as follows

```
Name PROC type
;Body of Procedure
RET
Name ENDP

;For calling
Call Name
```

Here,

- Type
    - Near: The statement that calls the procedure is in the same segment as the procedure. *Default* value.
    - Far: Calling statement is in the different segment
- Return: Returns back to calling place.
- Call: Calls the function. It does 2 operations

- The return address of the calling proc is saved in top of the stack. This is the offset address of next instruction to be executed, after proc is completed.
- IP(instruction pointer) gets the offset address of the first instruction of the procedure.

5 ARRAYS

5.1 1-D ARRAY

Eg are

- MSG DB 'HELLO$'
- W DW 10,20,30,40,50
- GAMMA DW 100 DUP(?)
- DELTA DB 212 DUP(?)
- LINE DB 5,4,2 DUP(2,3DUP(0),1) ;{5,4,2,0,0,0,1,2,0,0,0,1}

5.1.1 Addressing Modes

5.1.1.1 Register Mode

```
[register]
```

Is used to call the reference. However, this is only restricted to few registers.

BX,SI,DI	DS
BP	SS

Eg:

- MOV AX, [SI] ;Moves the content addressed by SI to AX
- MOV BX,[BX] ;Content of address BX into BX

5.1.1.2 Based and Indexed Addressing mode

```
[reg.+displacement]
[displacement+reg.]
[reg]+displacement
Displacement+[reg]
```

5.1.2

Registered that can be used is same as previous.

Eg: Let W be a variable. All examples mean the same

- MOV AX, W[BX]
- MOV AX, [W+BX]
- MOV AX, W+[BX]
- MOV AX, [BX]+W

5.1.2.1 PTR

```
type PTR address_expression
```

Here Types are either BYTE or WORD.

It is used to override the declared type of expression-

Eg

- DOLLAR DB 1AH
  CENT DB 52H
  ...
  MOV AX,DOLLAR ;Wrong
  MOV AX, WORD PTR Dollar ;Right

5.1.2.2 Accessing elements in the stack

Eg

- MOV BP, SP
- MOV AX, [BP]
- MOV BX, [BP+2]
- MOV CX, [BP+4]

5.2 2-D Array $[A]_{MXN}$

Eg

- B DW 10,20,30,40
  DW 50,60,70,80
  DW 90,100,311,121 ;Row Major order A+[(i-1)N+(j-1)]S; S is the size (2 for W, 1 for B)
- B DW 10,20
  DW 50,60
  DW 90,100 ;Column Major order A+[(i-1)+(j-1)M]S

5.2.1 Addressing mode

5.2.1.1 Base-Index mode

```
Variable[base_reg][index_reg]
```

What it does is gets info, after shifting the origin to Variable[Base_reg] and adds offset of index_reg. So A[4][3]==A[7]

6 String

Instruction	Command	Destination	Source
Move String	MOVSB	ES:DI	DS:SI
Compare String	CMPSB	ES:DI	DS:SI
Load String	LODSB	AL or AX	DS:SI
Scan String	SCASB	ES:DI	AL or AX

Store String	STOSB	ES:DI	AL or AX

7 Macros

A procedure is called at execution time, control transfers to the procedure and returns after executing Its statements. A macro Is invoked at assembly time. The assembler copies the macro's statements into the program at the position of the invocation. When the program executes, there is no transfer of control.

```
Macro_name MACRO d1,d2 … dn
      ;Statements
ENDM
;Bunch of shit
Macro_name d1,d2 … dn
```

Where d1,d2 … is the dummy list of arguments used by the macro

7.1 Local Psuedo-op

A macro with a loop or decision structure contains one or more labels. If such a macro Is Invoked more than once in a program, a duplicate label appears, resulting in an assembly error: This problem can be avoided by using local labels in the macro. To declare them, we use the LOCAL pseudo-op, whose syntax Is

```
LOCAL list_of__labels
```

7.2 Rept Macro

The REPT macro can be used to repeat a block of statements.

```
REPT expression
      ;Statements
ENDM
```

7.3 Indefinite Repeat

```
IRP d, <a1,a2,…,an>
      ;Statements
ENDM
```

7.4 Conditionals

```
CONDITIONAL
	;Statements
ELSE
	;Statements
ENDIF
```

7.4.1 List of Conditional

Form	True if
IF exp	Constant expression is non-zero
IFE exp	Constant expression is zero
IFB <arg>	Argument is missing
IFNB <arg>	Argument is not missing
IFDEF <sym>	Symbol sym is defined in the program (or as extern)
IFNDEF <sym>	Symbol sym is not defined in the program (or as extern)
IFIDN <string1> <string2>	String1 and string2 are identical. Brackets are require.
IFDEF <string1>, <string2>	Are different
IF1	Assembler is making the first assembly pass
IF2	Assembler is making the second assembly pass

8 Memory Management

.COM : has one segment only. ORG 100h is used for stack no need for @data lines.

8.1 Program Modules

8.1.1 Near v/s Far

A procedure Is NEAR if the statement that calls it is in the same segment as the procedure itself; a procedure is FAR if it Is called from a different segment.

FAR procedure Is In a different segment from Its calling statement, the CALL instruction causes first CS and then IP to be saved on the stack, then CS:IP gets the segment offset of the procedure. To return, RET pops the stack twice to restore the original CS:IP.

8.1.2 Extern

When assembling a module, the assembler must be Informed of names which are used In the module but are defined In other modules, otherwise these names will be flagged as undeclared.

```
EXTERN external_name_list
```

Here, external_name_list Is a list of arguments of the form name:type where name Is an external name, and type Is one of the following: NEAR, FAR, WORD, BYTE, or DWORD.

8.1.3 Public

A procedure or variable must be declared with the PUBUC pseudo-op if it is to be used in a different module

```
PUBLIC external_name_list
```

8.2 Full Segment Definations

With the full segment definitions, the programmer can control how segments arc ordered, combined with each other, and aligned relative to each other In memory

8.2.1 Segment Directive

```
name SEGMENT align combine class
      ;Statements
name ENDS
```

8.2.1.1 Align Type

The align type of a segment declaration determines how the starting address of the segment is selected when the program is loaded in memory.

Table 14.1 Align Types

PARA	Segment begins at the next available paragraph (least significant hex digit of physical address is 0).
BYTE	Segment begins at the next available byte.
WORD	Segment begins at the next available word (least significant bit of physical address is 0).
PAGE	Segment begins at the next available page (two least significant hex digits of physical address are 0).

PARA is the default align type.

8.2.1.2 Combine type

If a program contains segments of the same type, the combine type tells how they are to be combined when the program is loaded in memory.

Table 14.2 Combine Types

PUBLIC	Segments with the same name are concatenated (placed one after the other) to form a single, continuous memory block
COMMON	Segments with the same name begin at the same place in memory: that is, are overlaid.
STACK	Has the same effect as PUBLIC, except that all offset addresses of instructions and data in the segment are relative to the SS register. SP is initialized to the end of the segment.
AT paragraph	Indicates that the segment should begin at the specified paragraph.

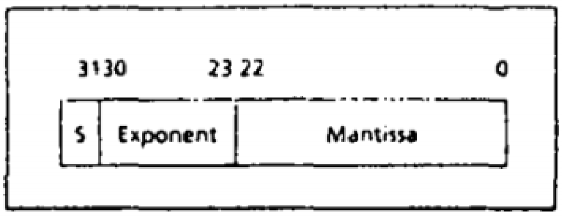
8.2.1.3 Class Type

The *class* type of a segment declaration determines the order in which segments are loaded in memory. Class type declarations must be enclosed in single quotes. If two or more segments have the same class. they are loaded in memory one after the other. If classes are not specified in segment declarations segments are loaded in the order, they appear in the source listing.

9 FLOATING POINT REPRESENTATION

In the floating-point representation, each number is represented in two parts: a mantissa, which contains the leading significant bits in a number, and an exponent, which is used to adjust the position of the binary point. For example, the number 2.5 in binary is 10.1b, and its floating-point representation has a mantissa of 1.1 and an exponent of 1. This is because 10.1b can he written as 1.01 x 21. Negative exponents are not represented as signed numbers. Instead, a number called the bais is added to the exponent to create a positive number. For example, if we use eight bits for the exponent, then the number (2^7 - 1) or 127 is chosen as the bias. To represent the number 0.0001b, we have a mantissa of 1.0 and an exponent of -4. After adding the bias of 127, we get 123 or 0111110 I b. It starts with a sign bit, followed by an 8-bit exponent, and a 23-bit mantissa. For short real figure 18.1 is represented

Figure 18.1 Floating-Point Representation



9.1 8087 STACK

The 8087 has eight 80-bit data registers, and they function as a stack. Data may be pushed or popped from the stack. The top of the stack is addressed as ST or ST(0). The register directly beneath the top is addressed as ST(1). In general, the i[th] register in the stack is addressed as ST(i), where I must be a constant. Data types in 8087 are

Data Formats	Range	Precision	Most Significant Byte (7 0 7 0 7 0 7 0 7 0 7 0 7 0 7 0 7 0 7 0)
Word integer	10^4	16 Bits	I_{15} … I_0 Two's Complement
Short integer	10^9	32 Bits	I_{31} … I_0 Two's Complement
Long integer	10^{18}	64 Bits	I_{63} … I_0 Two's Complement
Packed BCD	10^{18}	18 Digits	S $D_{17}D_{16}$ … D_1D_0
Short Real	$10^{\pm 38}$	24 Bits	S E_7 … E_0 F_1 … F_{23} F_0 Implicit
Long Real	$10^{\pm 308}$	53 Bits	S E_{10} … E_0 F_1 … F_{62} F_0 Implicit
Temporary Real	$10^{\pm 4932}$	64 Bits	S E_{14} … E_0 F_0 … F_{63}

Integer: I
Packed BCD: $(-1)^S(D_{17}...D_0)$
Real: $(-1)^S(2^{E-Bias})(F_0 \cdot F_1...)$
bias = 127 for Short Real
1023 for Long Real
16383 for Temp Real

Figure 18.2 8087 Data Types

10 IMPORTANT COMMANDS

Command Name	Syntax	Eg	Remarks

ASCII Adjusted Addition	AAA	MOV AL,6
		ADD AL,7
		AAA
			Used to check for carry in BCD values. Values is stored in AL. It does
			AH=AL/10
			AL=AL%10

ASCII Adjusted Multiplication	AAD	AAD	AL=AL+10*AH AH=0
ASCII Adjusted Multiplication	AAM	AAM	Same as above

ASCII Adjusted Subtraction	AAS	MOV AL,6
		SUB AL,7
		AAS
			Used to check for carry in BCD values. Values is stored in AL. It does
			AH=AL/10
			AL=AL%10

Add	ADD dest, src/val	ADD AH,2
		ADD AH,BX
		ADD Word1,AX
			Dest=dest+src/val

Add with Carry	ADC dest, src/val	ADC AX,1 ADC AX,BX	Adds the carry bit along with normal ops
And	AND dest, src/val	AND AL,7FH {Clears sign bit of AL}	Can be used to clear specific Bits while reserving others. 7FH stands for DEL. Changes the destination.
Convert Byte to word	CBW	CBW	Converts byte to word. Used in DIV/IDIV
Clear Direction Flag	CLD	CLD	Clears the direction flag. If 0 then pointer to the data is incremented

Compare	CMP dest, src	CMP AX,BX
		CMP AX, ‘$’
		CMP AX, 5
			Sets flag CF=0 if not equal, CF=1 if equal

Compare String byte **Compare String Word**	CMPSB CMPSW		subtracts the byte with address ES:DI from the byte with address DS:SI, and sets the flags. The result is not stored.
Convert Word to double word	CWD	CWD	Converts word to double word. Used for IDIV
Decrement	DEC dest	DEC BX	Dest -=1

		DEC AX	
Division	DIV divisor IDIV divisor	DIV BX DIV BL	For **DIV** DX=0 (AH=0) For **IDIV**, DX(AH) should be signed extension. Use CWD. **Byte:** Divisor is 8-bit register. AL = AX/divisor AH = AX%divisor **Word Form:** Divisor is 16 bit. Dividend is 32 bit DX:AX AX = DX:AX/div DX = DX:AX%Div
DUP	DUP(values)	DUP(?) DUP(42)	Fills random value Fills 42
Load onto 8087 stack	FLD (load real) FILD (int load) FBLD (packed BCD load)	FLD source FILD source FBLD source	Where Source is a memory location
Store 8087 into destination	FST (store real) FIST (store int) FSTP (Store real pop) FISTP (store int pop) FBSTP (store BCDpop)	FST destination FIST destination FSTP destination FISTP destination FBSTP destination	Where destination is a memory location
Arithmetic on 8087 stack	FADD [[dest,] src] FSUB [dest, src] FMUL [dest, src] FDIY [dest, src] FIADD source FISUB source FIMUL source FIDIV source	Add Subtract Mul Divide Integer add Integer subtract Integer mul Int div	Each opcode can take zero, one, or two operands. An instruction with no operands assumes ST(0) as the source and ST(1) as the destination; the instruction also pops the stack. In an instruction with one operand, the operand specifies a memory location as the source; the destination. is assumed to be ST(0).
Increment	INC dest	INC BX INC AX	Dest +=1
Interrupt	INT 21h	Int 21h	Does when: **AH=1**: gets ASCII Code of key pressed **Input:** *AL=ASCII of Key pressed* **AH=2**: prints single char output **Input:** *DL=ascii value of char to be printed* **AH=9**: String output till it encounters ‘$’

			Input: *DX=offset address of the string*
Jump if greater *(signed)*	JG/JNLE	JG K3	Condition for jump: ZF=0 & SF=OF
Jump if greater than equal	JGE/JNL	JGE K4	SF=OF
Jump if less	JL/JNGE	JL I5	SF<>OF
Jump if less than & equal	JLE/JNG	JLE L4	ZF=1 & SF <> OF
Jump if greater *(unsigned)* **i.e Jump if Above**	JA/JNBE	JA K4	Condition for jump: CF=0 & ZF=0
Jump if above or equal	JAE/JNB	JAE	Condition for jump: CF=0
Jump if below	JB/JNAE	JB	Condition for jump: CF=1
Jump if below of equal/not above	JBE/JNA	JBE K3	Condition for jump: CF=1 or ZF=1
Jump if equal *(Single flag Jump)*	JE/JZ	JE	Condition for Jump: ZF=1
Jump if not equal	JNE/JNZ		ZF=0
Jump if carry **Jump if not carry**	JC JNC		CF=1 CF=0
Jump if overflow **Jump if not overflow**	JO JNO		OF=1 OF=0
Jump if signed **Jump if not signed**	JS JNS		SF=1 SF=0
Jump if even parity **Jump if odd parity**	JP/JPE JNP/JPO		PF=1 PF=0
Unconditional Jump	JMP	JMP k4	
Load Effective Address	LEA dest,src	LEA DX, MSG LEA SI, AX	Loads address of source to Destination
Load String	LODSB LODSW		moves the byte addressed by DS:SI into AL. SI is then incremented if DF = 0 or decremented if DF.= 1.

Move	MOV dest, src	MOV AH, WORD1
		MOV AX,BX
		MOV AH, 'A'
			Both destination and source cannot be memory variable. MOV Word1,Word2 Won't work.

Move string Byte	MOVSB REP MOVSB	MOVSB REP MOVSB	Moves a single byte from DS:SI to ES:DI. Repeats the command CX number of bytes
Move String Word	MOVSW REP MOVSW	MOVSW REP MOVSW	SI, DI increase by 2. CX number of words.

Multiplication	MUL source IMUL source	MUL BX MUL BL	Unsigned Signed In **byte form**: AX=source*AL **Word Form**: DX:AX=source*AX
Negation	NEG destination	NEG AX NEG AH	To negate destination
NOT	NOT Destination	NOT K	Changes bit from 0 to 1 and 0 from 1. {If K =1101b NOT K=0010b}
OR	OR	OR AL, 81H {Sets the msb and lsb of AL}	OR is used to set specific bit while reserving the others. Changes the source
Pop	POP destination	POP AX POP BX	Destination must be 16 bit. Does 2 operations 1) Pop ->destination 2) SP+=2
PopF	POPF	POPF	Content of top of the stack will be moved to flag register
Push	PUSH source	PUSH AX PUSH BX	Source must be 16 bit. It does 2 operations 1) SP-=2 2) SS:SP
PushF	PUSHF	PUSHF	Pushes the content of flag register on top of the stack
Rotate and Carry Left	RCL		Shifts CF into higher order word
Repeat until not equal to **Repeat until equal to**	REPNE SCASB REPE SCASB REPE CMPSB REPNE CMPSB	REPNE SCASB	will repeatedly subtract each string byte from AL, update DI, and decrement CX until there is a zero result (the target is found) or CX = 0 (the string ends).
Roll Left	ROL Dest, bits_shifted	Same as SHL	The ROL instruction shifts each bit to the left, with the highest bit copied in the Carry flag and also into the lowest bit
Roll Right	ROR Dest, bits_shifted	Same as SHL	The ROL instruction shifts each bit to the Right, with the lowest

			bit copied in the Carry flag and also into the highest bit
Shift Arithmetic Left	SAL Dest, bits_shifted	SAL reg,CL	Identical to SHL.
Subtract with borrow	SBB Dest, Src/val	SBB AX, BX SBB AX, 13	Subtracts the carry bit too along with normal subtraction
Scan Byte **Scan Word**	SCASB SCASW	SCASB	SCASB subtracts the string byte pointed to by ES:DI from the contents of AL and uses the result to set the flags. ZF=0 ;If diff ZF=1; if same char
Shift Arithmetic Right	SHR Dest,Bits_shifted	SAR reg,CL Same as SHl	The lowest bit is copied to carry flag. And signed bit is copied to right
Shift Left	SHL Dest, bits_shifted	SHL reg, CL SHL mem, CL SHL reg, imm8 SHL mem, imm8	The highest bit is moved to carry flag. Fills lowest bit with 0. MSB->CF.
Shift Right	SHR Dest, bits_shifted	SHR reg, CL SHR mem, CL SHR reg, imm8 SHR mem, imm8	The lowest bit is moved to carry flag. Fills highest bit with 0. MSB->CF.
Set Direction Flag	STD	STD	Sets the direction flag. Pointer is decremented
Store String	STOSB STOSW		Moves the content of AL to ES:DI, and inc/dec the pointer according to byte/word.

Subtract	SUB dest, src/val	SUB AH,2
		SUB AH,BX
		SUB Word1,AX
			Dest=dest-src/val

Test	TEST Dest, src	TEST AH, 01H	Same as and but without changing the source.
Exchange	XCHG dest, src	XCHG AH,BL XCHG AX,WORD1 XCHG AX,BX	Both destination and source cannot be memory variable. XCHG Word1,Word2 Won't work.
XLAT	XLAT	MOV AL, 5 LEA BX, T1 XLAT ;This will do AL=[T1+5]	Low operand instruction which is used to convert byte value into another value that comes from

			byte table. It adds the content of AL to the address in BX and retrieves the value of the address to AL
XOR	XOR		XOR is used to Compliment specific bits while reserving the others