**Chapter 12: PROGRAMMING RULES**

**Topic – 1: Variables**

**Define Directives**

* **Define directive:** Syntaxes used for **reserving space**, for constants or variables.
* For example, **DW**, **DQ**, **RESB** etc.

**Note!**

**🡪 Each character of a string is stored as its ASCII hex value in memory.**

**🡪 Characters are of 1-byte each & ASCII codes are 8-bit long.**

**🡪 Processor uses the little-endian byte ordering which we discussed in "Chapter 7".**

**🡪 Negative numbers are stored as their 2’s complement.**

**🡪 Short floating-point numbers are represented using 4-bytes.**

**🡪 Long floating-point numbers are represented using 8-bytes.**

**Multiple Initializations**

* We use the **TIMES** directive for it.
* But it can only be used with ***.data*** sections.

***msg times 9 dw 0 ; 000000000***

**Topic – 2: Constants**

**Introduction**

* **Constants** are written **outside** & even **within** segments.
* There are various **directives** to describe **constants** like **EQU**, **%assign**, **%define**.

**EQU Directive**

***length equ 50***

***width equ 20***

***area equ length\*width***

**%assign Directive**

* The **%assign** directive allows us to **reassign** the value later in the program.

***%assign TOTAL 10***

***; Some lines later…***

***%assign TOTAL 20***

**Note!**

**🡪 %assign directive is case-sensitive.**

**🡪 We can reassign the constant within main program too.**

**%define Directive**

* We can **redefine** the **constant** using it too.
* The difference is that we can define a **string** too, using it.
* This directive is also **case-sensitive**.

***%define PTR [EBP+4]***

***%define STR "World"***

**Topic – 3: Arithmetic Instructions**

**INC Instruction**

* Increments the value of operand by **1**.
* Only one operand is passed.
* This operand can be a **variable**, **memory address** or even a **register**.

**DEC Instruction**

***\*Now you know\****

**ADD & SUB Instructions**

* Down beneath, the calculation is performed among **each bit** of the **two operands**.
* This can even happen among **memory to registers** (vice-versa), **register to data** (vice-versa), **memory to data** (vice-versa) etc.
* The only exception is **memory-to-memory** addition/subtraction.
* **ADD** and **SUB** instructions can affect the **overflow** & **carry flag**.

**Note!**

**🡪 It’s a good practice to use [ ] around variables for directly referring to its memory address & manipulating its value from there.**

**Conversions**

***add eax, '0' ; Decimal to ASCII***

***sub ebx, '0' ; ASCII to decimal***

**MUL & IMUL Instructions**

* **IMUL** stands for **integer multiply**.
* **MUL** is used to treat **strings** as **integers**.
* They can affect **carry** & **overflow** flags.
* But **overflow** doesn’t occur as product is kept in **double** length.
* It can be done only on one operand i.e. the **accumulator**.
* But the storage is done in separate ways depending on **operand sizes**, as shown in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operands Size** | **Multiplicand** | **Multiplier** | **Upper Bits Storage** | **Lower Bits Storage** |
| **BYTE** | **AL** | **8-bit source** | **AH** | **AL** |
| **WORD** | **AX** | **16-bit source** | **DX** | **AX** |
| **DWORD** | **EAX** | **32-bit source** | **EDX** | **EAX** |
| **QWORD** | **RAX** | **64-bit source** | **RDX** | **RAX** |

**DIV & IDIV Instructions**

* It generates a **quotient** & a **remainder**.
* **Overflow** can occur in it.
* **IDIV** is used for **signed data** & **DIV** for **unsigned data**.
* Just like **multiply** instructions, we use **accumulator** as **dividend**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operands Size** | **Dividend** | **Divisor** | **Upper Bits Storage** | **Lower Bits Storage** |
| **BYTE** | **AL** | **8-bit source** | **AH** | **AL** |
| **WORD** | **DX, AX** | **16-bit source** | **DX** | **AX** |
| **DWORD** | **EDX, EAX** | **32-bit source** | **EDX** | **EAX** |
| **QWORD** | **RDX, RAX** | **64-bit source** | **RDX** | **RAX** |

**Note!**

**🡪 We must flush registers by moving 0 to them after use to avoid ambiguity.**

**Topic – 4: Logical Instructions**

**Introduction**

* Available **logical operations** in instruction set are **AND**, **OR**, **XOR**, **TEST** & **NOT**.
* Other than **NOT**, all requires **two operands**.
* First operand can be in **register** or **memory**.
* Second operand can be **register**, **memory** or an **immediate constant**.
* Again, **memory-to-memory** operations **aren’t** possible.
* These can affect **CF**, **OF**, **PF**, **SF** & **ZF** flags.

**AND Instruction**

* The result of a **logical operation** is stored in **first operand**.
* We can set **higher order bits** to **0** using **AND**.

***; BL contains 0011 1010***

***and bl, 0fh ; Sets BL to 0000 1010***

**OR Instruction**

* We can set **lower bits** to **1** using **OR**.

***; BL contains 0011 1010***

***or bl, 0fh ; Sets BL to 1111 1010***

**TEST Instruction**

* Does same as **AND** but **doesn’t** affect any operand unlike **AND**.

***test eax, 0fh***

***jz my\_label ; Jumps to label if previous operation result was 0***

* **JZ** can sense operations involving any **register** or **memory location**.

**Topic – 5: Conditions**

**Introduction**

* **Conditions** are used in both **looping & branching** instructions.

**CMP Instruction**

* **Compares** two operands.
* Follows **conditional execution**.
* It checks the **difference** between **two operands** to see if they are **equal**.
* It **doesn’t** affect any of the operand.
* Often used along **conditional jump instruction**.
* First operand can be **register/memory** & second can be **register/memory/constant**.

***cmp rdx, 10***

***je label1 ; If equal, then jump to label1***

**Unconditional Jump**

* **Jumps** to a label & transfers control to it.
* Can be programmed anywhere to jump **without** any condition.

***jmp label1***

**Conditional Jumps**

* Jumps only when certain criterion is met.

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Description** | **Flags Tested** | **Use Case** |
| **JE** | **-** | **ZF** | **Both** |
| **JZ** | **-** | **ZF** | **Both** |
| **JNE** | **-** | **ZF** | **Both** |
| **JNZ** | **-** | **ZF** | **Both** |
| **JG/JNLE** | **-** | **OF, SF, ZF** | **Signed integers** |
| **JGE/JNL** | **-** | **OF, SF** | **Signed integers** |
| **JL/JNGE** | **-** | **OF, SF** | **Signed integers** |
| **JLE/JNG** | **-** | **OF, SF, ZF** | **Signed integers** |
| **JA/JNBE** | **Jump above/ Jump not below or equal** | **CF, ZF** | **Unsigned integers** |
| **JAE/JNB** | **Jump above or equal/ Jump not below** | **CF** | **Unsigned integers** |
| **JB/JNAE** | **Jump below/ Jump not above or equal** | **CF** | **Unsigned integers** |
| **JBE/JNA** | **Jump below or equal/ Jump not above** | **AF, CF** | **Unsigned integers** |
| **JXCZ** | **Jump if CX is zero** | **-** | **Special use** |
| **JC** | **Jump if carry** | **CF** | **Special use** |
| **JNC** | **-** | **CF** | **Special use** |
| **JO** | **Jump if overflow** | **OF** | **Special use** |
| **JNO** | **-** | **OF** | **Special use** |
| **JP** | **Jump parity** | **PF** | **Special use** |
| **JPE** | **Jump parity even** | **PF** | **Special use** |
| **JNP** | **Jump no parity** | **PF** | **Special use** |
| **JPO** | **Jump parity odd** | **PF** | **Special use** |
| **JS** | **Jump sign (-ve value)** | **SF** | **Special use** |
| **JNS** | **Jump no sign (+ve value)** | **SF** | **Special use** |

**Topic – 7: Loops**

**Using Jump**

* We can use **jump** statements to make a **loop** environment.

**LOOP Statement**

* There is a **keyword** called **LOOP**.
* It assumes **RCX** register as **loop counter**.
* When loop is executed, **RCX** **decrements** each iteration until it reaches **0**.
* And for this, the **label** described by it is used as **loop**.

***mov rcx, 10 ; Loop will iterate 10 times***

***label1:***

***; Loop body***

***loop label1***

**Topic – 8: Printing Decimal To Screen**

**Introduction**

* **Decimals** can be shown on screen **only** after being converted to **ASCII form**.
* This part deals closely with **memory** & is very important to discuss.

**Program**

***segment .data***

***num dd 7***

***segment .bss***

***buffer resb 1 ; Remember this declaration***

***segment .text***

***global \_start***

***\_start:***

***mov eax, [num]***

***add eax, '0' ; Decimal to ASCII conversion***

***mov [buffer], eax ; An intermediate container is hell important***

***mov edx, 1***

***mov ecx, buffer ; Direct transfer from EAX won’t worked***

***mov ebx, 1***

***mov eax, 4***

***int 0x80***

***xor ebx, ebx***

***mov eax, 1***

***int 0x80***

**Topic – 9: Numbers**

**Introduction**

* Arithmetic instructions operate on **binary data**.
* When numbers are taken **entered from keyboard** (**input**) or **displayed on screen** (**output**), they are in **ASCII** form.
* Converting an **ASCII to decimal** & vice-versa has a big **overhead**.
* **More efficient** option is to use it in **BCD** form.
* **BCD** stands for **Binary Coded Decimal**.

**ASCII Representation**

|  |  |
| --- | --- |
| **Symbol** | **ASCII Code** |
| **0** | **0x30** |
| **1** | **0x31** |
| **2** | **0x32** |
| **3** | **0x33** |
| **4** | **0x34** |
| **5** | **0x35** |
| **6** | **0x36** |
| **7** | **0x37** |
| **8** | **0x38** |
| **9** | **0x39** |

**Numbers Processing Instructions**

* **AAA –** ASCII adjust after **addition**
* **AAS –** ASCII adjust after **subtraction**
* **AAM –** ASCII adjust after **multiplication**
* **AAD –** ASCII adjust **before** **division**

**Note!**

**🡪 These instructions assume the target operand to be AL.**

**AAS Example**

***sub ah, ah ; Emptying AH***

***mov al, '9' ; Loads 39H to AL***

***sub al, '3' ; 39H – 33H = 6H***

***aas ; Tells assembler that we subtracted result in AL***

***or al, 30h ; 6H OR 30H = 36H = 6***

***mov [res], ax ; res will contain 36H now***

**BCD Representation**

* **Packed BCD representation:** Each **4-bits** group stores a digit from our number.

**For example:**

**73 = 73H**

* **Unpacked BCD representation:** Each **byte** stores a digit from our number.

**For example:**

**73 = 7 3 = 07H 03H**

* **AAA**, **AAS**, **AAM** & **AAD** are used for processing **unpacked BCDs**.
* **DAA** & **DAS** are used for processing **packed BCDs**.
* There is however no **multiplication** **& division** support for them.

**Topic – 10: Strings**

**String Length**

***len equ $ - msg***

* ***$*** points to **byte** after **last character** of our **string**.
* So, (***$ - msg***) gives us the **length** of our string.
* Otherwise, we can also **explicitly mention** its size as following.

***len equ 13***

* We can also **alternatively** add a **sentinel character** (special character) at the **end** using its **Unicode**.
* Doing so, we **won’t** be requiring to store its length **explicitly** in another variable.

***msg db "We love NASM!", 0xa ; 0xa is a sentinel character***

**String Instructions**

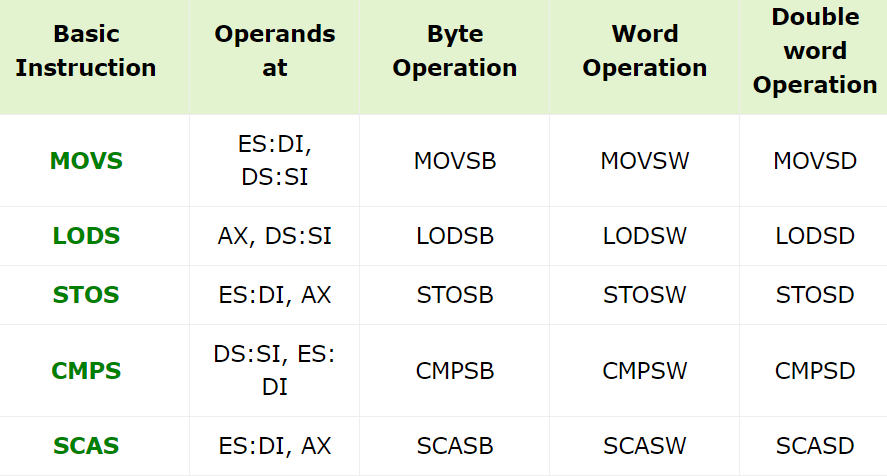
* These instructions require **destination operand** and/or **source operand**.
* We generally use **SI** & **DI** series registers for this work, like **RSI** & **RDI**.
* **MOVS –** Moves **1-byte** of **WORD** or **DWORD** from **memory location** to **destination**.
* **LODS –** Loads **source operand** at a **memory location** to **accumulator**. But whether it stores it at **AL**, **AX**, **EAX** or **RAX** depends solely on **size** of operand.
* **STOS –** It stores data in **accumulator** (size may **vary**) to a **memory location**.
* **CMPS –** Compares **two data items** in memory of size **BYTE**, **WORD** or **DWORD**.
* **SCAS –** Compares the item at **accumulator** with an item in **memory**.

**Note!**

**🡪 64-bit architectures even allow QWORD data to be used by instructions above.**

* These string instructions use **ES:DI** (**RSI**) & **DS:SI** (**RDI**) pair.
* **ES –** Extra segment
* **DS –** Data segment
* **RSI** is assumed to be at **source operand** & **RDI** to be **destination**.

**String Specific Instructions**



**Repeat Instructions**

* When a **repeat instruction** is added before a string instruction, the string instruction is **repeatedly executed** until certain condition is met.

***rep movsb ; Repeated until CX becomes 0 (CX decrements)***

***repe movesb ; Repeats until CX becomes 0 or ZF is indicated as 1***

***repz movesb ; Same as previous***

***repne movsb ; Repeats until CX becomes 0 or ZF is indicated as 0***

***repnz movsb ; Same as previous***

* **CX** decrements on **each iteration** of the repeat instructions.

**Direction Flag Instructions**

***cld ; Clear direction (DF = 0), DF points to left***

***std ; Set direction (DF = 1), DF points to right***

**Topic – 11: Arrays**

**Introduction**

* We use ***data definition directives*** for allocating space to **variables**.
* For example, **DB**, **DW**, **DQ** etc.
* It can be defined in form of **decimal**, **binary** or even **hexadecimal** form.

***months dw 12***

***months dw 01010b ; Leading 0 is added for clarity & convention***

***months dw 0ch ; Same reason as above***

**Array Definition**

***numbers dw 10, 20, 30, 40, 50***

* Note that we are using **DW** data definition.
* This means that **elements** in array are of **2 bytes** each.
* Size of number tells the **possibility** or **range of values** that it can take.
* So, these numbers must **not** exceed value of **(216 - 1)**.
* **-1** there is the **MSB** reserved for **sign** of number.
* ***numbers*** mean **1st element** & ***numbers + 1*** mean **2nd** etc.

**Alternative Array Definition**

***numbers dw 10***

***dw 20***

***dw 30***

***dw 40***

***dw 50***

* **Same** **array** as previous with **same elements**, but defined in different manner.

**Array With Same Elements**

***binaries times 8 dw 1 ; Array 'binaries' containing 8 elements of value 1***

**Note!**

**🡪 Sections always run whether called or not, when they appear.**