**Assembly Language Lab 4**

**1. FLAGS Register**

Flags register is a special register, its bits indicate the status of various activities from the last instruction executed. Its bits values are accessed by conditional jumps instructions (explained later). Most important flags are:

**CF (Carry flag):** contains carry out of high‐order bit of data item following anarithmetic operation and some shift and rotate operations. It is often used to indicate if the result of an unsigned arithmetic operation is too large to fit in the destination.

**OF (Overflow flag):** indicates the result of asignedarithmetic operation is toolarge to fit in the destination such that there is a carry to sign bit (i.e. it is set if there is a carry into sign bit and no carry out).

**ZF (Zero flag):** if result of arithmetic or logic operation is zero then it is set to 1,otherwise it is cleared to 0.

**SF (Sign Flag):** a positive result from an arithmetic operation clears it to 0,otherwise it is set to 1

**2. Basic Instructions**

***MOV/MOVZX/MOVSX Instructions***

The most basic instruction is the **MOV** instruction. It moves data from one location to another (like the assignment operator in the high‐level language). It takes two operands: a source (a register, a memory location or an immediate value) and a destination (a register or a memory location).

Format:

***mov dest, src***

*dest* and *src* must have the same size and both *should not* be memory operands. Also,you cannot move an immediate value to segment registers.

Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| mov | eax, 3 | ; | store 3 into EAX register | (3 is an immediate value) |
| mov | bx, ax | ; | copy the value of AX into | the BX register |

Moving data from a smaller location to a larger location is really an issue and need special instructions as CPU don’t know exactly how to fill the high order bits of the destination. Study this example:

If we have an 8‐bit value 11111110b and we want to move it to a larger register, say a 16‐bit register. If the CPU filled the higher 8 bits with 0’s then the new number will be

00000000 11111110

But this number will be totally different if the original number is treated as a signed number. Why? Because if 11111110 is a signed number, then it represents ‐2 while 00000000 11111110 represents 254. This difference happens as the number is negative and because 0’s were added in the higher bits, the sign bit is changed to 0 and so the number is converted to positive. So, in case of signed number, the higher order bits should follows the sign bit of the original number. In other words, if the sign bit is 1, then fill them by 1’s. Otherwise, fill them by 0’s. Then what about the unsigned number? Actually, they are not a problem as they are always positive, and filling the higher bits by 0’s is always correct.

MOVZX and MOVSX instructions are dedicated instructions to deal with moving from different sizes locations. MOVZX instruction is with unsigned values while MOVSX instruction is used with signed values. Remember that the programmer is who knows a value is treated as signed or unsigned according to the program logic.

Example:

|  |  |  |
| --- | --- | --- |
| **MOV AL, 11111110B** | ;AL = 254 | (unsigned), AL = -2 (signed) |
| **MOVZX BX,AL** | ;BX = | 254 |  |
| **MOVSX CX, AL** | ;CX = | -2 |  |

***INC/ DEC Instructions***

INC (increment) and DEC (decrement) instructions are used to add 1 and subtract 1 from a single operand respectively.

Format:

**INC reg/mem**

**DEC reg/mem**

Flags affected: **OF, SF, ZF**

**Note: INC and DEC instructions do not affect the carry flag.**

***ADD/SUB Instructions***

The ADD and SUB instruction are used to add and subtract integers respectively.

***ADD/SUB dest, src***

Format:

**ADD/SUB reg, reg**

**ADD/SUB mem, reg**

**ADD/SUB reg, mem**

Flags affected: **OF, CF, SF, ZF**

Examples:

add eax, 4 ; eax = eax + 4 sub ebx, 3 ; ebx = ebx – 3

***NEG Instruction***

NEG instruction reverses the sign of a number by converting it to the two’s complement form.

Format:

**NEG reg/mem**

Flags affected: **CF, OF, SF, ZF**

**Note:** NEG instruction produce invalid values when the result cannot fit the operand.If so, the OF is set. For example, when we try to NEG the value ‐128, the value +128 cannot fit in 8‐bit and so the overflow flag is set.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| mov | al, -128 | ;al | = | 10000000b |
| neg | al | ;al | = | 10000000b, OF = 1 |

***Example : Basic Instructions***

Debug the following program and notice the changes in registers values

**INCLUDE Irvine32.inc**

**.data Rval DD ?**

**Xval DD 26 Yval DD 30**

**Zval DD 40**

|  |  |  |  |
| --- | --- | --- | --- |
| **.code** |  |  |  |
| **Main:** | **DumpRegs** |  |  |
| **call** |  |  |
| **;MOVSX, MOVZX** |  |  |
| **mov al, 0AAh** | **;ecx = 0FFAAh** |  |
| **movsx ecx, al** |  |
| **movzx ebx, al** | **;ecx = 00AAh** |  |
| **call** | **dumpRegs** |  |  |
| **; INC and DEC** |  |  |
| **mov ax,1000h** | **; 1001h** |  |
| **inc ax** |  |
| **dec ax** | **; 1000h** |  |
| **call** | **DumpRegs** |  |  |
| **; Expression: Rval = -Xval + (Yval - Zval)** |  |
| **mov** | **eax,Xval** | **; -26** |  |
| **neg** | **eax** |  |
| **mov** | **ebx,Yval** | **; -10** |  |
| **sub** | **ebx,Zval** |  |
| **add** | **eax,ebx** | **; -36** |  |
| **mov** | **Rval,eax** |  |
| **call** | **DumpRegs** |  |  |

**; Overflow flag example:**

|  |  |  |  |
| --- | --- | --- | --- |
| **mov** | **al,+127** | **; OF = 1 as +128 cannot fit the 8-bit register (al)** |  |
| **add** | **al,1** |  |
| **call** | **dumpregs** |  |  |
| **mov** | **al,-128** | **; OF = 1 as -129 cannot fit the 8-bit register (al)** |  |
| **sub** | **al,1** |  |
| **call dumpregs** |  |  |
| **; Zero flag example:** |  |
| **mov** | **cx,1** | **; ZF = 1** |  |
| **sub** | **cx,1** |  |
| **mov** | **ax,0FFFFh** | **; ZF = 1** |  |
| **inc** | **ax** |  |
| **; Sign flag example:** |  |
| **mov** | **cx,0** | **; SF = 1** |  |
| **sub** | **cx,1** |  |
| **mov** | **ax,7FFFh** | **; SF = 1** |  |
| **add** | **ax,2** |  |
| **; Carry flag example:** |  |
| **mov** | **al,0FFh** | **; CF = 1, AL = 00** |  |
| **add** | **al,1** |  |
| **exit** |  |  |  |
| **END main** |  |  |
|  |  |  |  |

High level language provides program control statements like if‐statement and while‐loop to control the program execution flow. Assembly language does not have such complex program control as instructions. Instead, it uses *branch* (jump, or goto‐ like) instructions to control the execution flow. However, there are a set of MASM directives that allow writing control structures like if and while statements in assembly code. In the next sections, program control instructions are introduced with a set of illustrating examples.

**3. JMP ­ Unconditional Jump**

JMP instruction transfers program execution to the address specified by the destination operand. It does not affect flags register. It just updates the EIP register by the new address and so the program execution resumes from that address.

Jumps may be ***short*** (between –128 and +127 bytes), ***near*** (between –32,768 and +32,767 bytes from the instruction following the jump), or ***far*** (in a different code segment). ***When the 80386+ processors are in FLAT memory model, short jumps*** ***range is from –128 to +127 bytes and near jumps range is from –2 to +2 gigabytes***.

**Format:**

**Jmp** *label*

***Example: Using JMP instruction, infinite loop***

This program will never end. It continues printing registers values and never exit.

**INCLUDE Irvine32.inc**

**.data**

**.code main PROC**

L1:

call DumpRegs

jmp L1 ;go to L1 line of code

;the program execution will never get here.

**exit main ENDP END main**

**4. CMP ­ Compare Two Operands**

CMP instruction compares two operands as a test for a subsequent conditional‐ jump (Conditional jumps will be explained later). **CMP** make comparison by ***subtracting*** the source operand from the destination operand and setting the flagsaccording to the result. **CMP** is the same as the **SUB** instruction, except that the result is not stored anywhere. Only the flags are affected (like SF, ZF, CF, and OF).

**Format:**

CMP **reg/mem**, **reg/mem/immed**

The following restrictions on CMP instruction should be considered:

1. All combinations of operands are allowed except **CMP mem**, **mem.** It is not accepted.
2. Can't compare two segment registers.
3. Can't compare operands with different sizes.

***Example: Using CMP instruction***

Debug the following program and notice changes in the flags after each CMP instruction.

**INCLUDE Irvine32.inc**

**.data**

**x dword ?**

**.code main PROC**

**mov ax, 10 mov bx, 50 cmp ax, bx**

**mov eax, 10 mov ecx, 10 cmp eax, ecx**

**mov eax, 50 mov x, 10 cmp eax, x**

**exit**

**main ENDP**

**END main**

;subtract: 10 - 50. Result is Negative. ;CF = 1, ZF = 0, OF = 0, SF = 1

;subtract: 10 - 10. Result is Zero. ;CF = 0, ZF = 1, OF = 0, SF = 0

;subtract: 50 - 10. Result is Positive. ;CF = 0, ZF = 0, OF = 0, SF = 0

**5 . Conditional Jumps**

Conditional jump instructions transfer the program execution to a specified label if certain flags conditions are true. They are often used after a CMP instruction to give the effect of *if‐statement* (or even *while‐statement*) in high level languages. Thus, CMP is used to compare between two operands and set flags accordingly while conditional jumps are used to test these flags to determine the relation between those two operands and jump if the relation is occurred.

**Format:** conditional jump instructions have the formatJ*xxx label*where xxxexpress the relation tested. The following table shows the conditional jump instructions:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Relation |  | For unsigned data |  | For signed data |  |
| **Equal/Zero** |  | JE/JZ |  |  |
| **Not Equal/Not Zero** |  | JNE/JNZ |  |
| **Above/Greater** |  | JA/JNBE |  | JG/JNLE |  |
| **Above or equal/** |  | JAE/JNB |  | JGE/JNL |  |
| **Greater or equal** |  |  |  |
|  |  |  |  |  |  |
| **Below/Less** |  | JB/JNAE |  | JL/JNGE |  |
| **Below or equal/** |  | JBE/JNA |  | JLE/JNG |  |
| **Less or equal** |  |  |  |

**Notes:**

* The difference between the two columns of instructions in the above table is that the first column of instructions considers that the data compared are ***unsigned*** data while the second column considers that the data are ***signed*** data.
* For example, JA jumps to the given label if the flags specify that the first operand is larger than the second one as *unsigned data* but JG jumps to the given label if the flags specify that the first operand is larger than the second one as *signed data*. We should note that if 11111111B is compared with 0 then it is larger as an unsigned value (255) but smaller as a signed value (‐1).

***Example: Using Conditional Jumps***

This example accepts two integers from the user and prints the relation between those integers: greater, less, or equal.

**INCLUDE Irvine32.inc**

**.data**

**strgreater byte "X is above than Y", 0**

**strless byte "X is below than Y", 0 strequal byte "X is equal to Y", 0 x dword ?**

**y dword ?**

|  |  |  |
| --- | --- | --- |
| **.code** |  |  |
| **main PROC** | **;**An Irvine function that reads an 32-bit **unsigned** |  |
| **call ReadDec** |  |
|  | ;decimal integer from user and stores it in EAX |  |
| **mov x, eax** | ;register. |  |
|  |  |
| **call ReadDec** |  |  |
| **mov y, eax** |  |  |
| **cmp x, eax** | ;eax still has the y value |  |
| **ja above** | **;**Assume unsigned data. use JA and JB (Not JG and JL) |  |
| **jb below** |  |  |
| **je equal** |  |  |

**above:**

**mov edx, offset strgreater** ;handle **above** case **call writestring**

**jmp next**

**below:**

**mov edx, offset strless** ;handle **below** case **call writestring**

**jmp next**

**equal:**

**mov edx, offset strequal** ;handle **equal** case **call writestring**

**jmp next**

|  |  |  |
| --- | --- | --- |
| **next:** | **;**An Irvine function that prints new line |  |
| **call CrLf** |  |
| **exit** |  |  |
| **main ENDP** |  |  |
| **END main** |  |  |

**Notes:**

* ReadDec function is a function defined in Irvine library. It reads a 32‐bit **unsigned** decimal integer from the user and stores it in EAX Register, stoppingwhen the Enter key is pressed, leading spaces are ignored. ReadDec will set the Carry flag, and reset EAX to zero if the value entered cannot be represented as a 32‐bit unsigned integer (blank or larger than 2^32‐1). If you need to store the entered value in a variable, you need to move it from EAX to this variable.
* OFFSET reserved word is an operator that retrieves the offset address of the given variable.
* Writestring function is a function defined in Irvine library. It prints a string constant to which the EDX register points. The string must be null‐terminated (ends with a byte contains 0). This explains why string variables are appended by 0 at its initializer.
* CrLf function is a function defined in Irvine library. It prints a new line. It actually prints carriage return/linefeed sequence (0Dh, 0Ah) to standard output which makes cursor to go to a new line.

***Example: Using Conditional Jumps (Revisited)***

This example is the same as Example 3a but with signed numbers and fewer labels.

INCLUDE Irvine32.inc

.data

strgreater byte "X is greater than Y", 0 strless byte "X is less than Y", 0 strequal byte "X is equal to Y", 0

**x SDWORD ? y SDWORD ?**

.code main PROC

call readint **;**An Irvine function that reads an 32- bit **signed** ;decimal integer from user and stores it in EAX register.

mov x, eax

call readint

mov y, eax

cmp x, eax

**jge greaterOrequal ;jump if greater or equal**

**mov edx, offset strless**  ;handle **less** case

**call writestring** ;(not greater and not equal)

**jmp next ;**go to end to avoid executing

**greaterOrequal: ;greater or equal case**

**je equal ;flags still preserve their values.**

**;No need to compare again.**

**mov edx, offset strgreater** ;handle **greater c**ase

**call writestring**

**jmp next**

**equal: ;handle equal case**

**mov edx, offset strequal**

**call writestring**

next:

call CrLf exit

main ENDP END main

**Notes:**

* 1. - ReadInt function is a function defined in Irvine library. It reads a 32‐bit **signed** decimal integer from the user and stores it in EAX Register, stopping when the Enter key is pressed, leading spaces are ignored, and an optional leading + or ‐ sign is permitted. ReadInt will display an error message, set the Overflow flag, and reset EAX to zero if the value entered cannot be represented as a 32‐bit signed integer. If you need to store the entered value in a variable, you need to move it from EAX to this variable.

**6 . LOOP instruction ­ *like for loop statement***

Assembly language provides several instructions designed to implement *for‐like*

loops. Each of these instructions takes a code label as its single operand. These instructions are LOOP, LOOPE/LOOPZ, and LOOPNE/LOOPNZ. We concern now LOOP instruction. LOOP instruction considers ECX register as its loop counter. It keeps decrementing ECX and jumping to its label till ECX becomes zero. Therefore, the operation of LOOP instruction is:

**LOOP**: Decrements ECX. If ECX != 0, branch to label.

**Format**

**LOOP** *label*

***Example : Sum of an Integer Array***

This example calculates the summation of an array. It first gets the offset of the array and stores it in ESI register. Then, initialize ECX by the length of the array. Within the loop, it accumulates the current element of the array then slides ESI to point to the next element of the array and so on till the ECX becomes zero.

**INCLUDE Irvine32.inc**

**.data**

**Arr1 DWORD 10, 20, 30, 40, 50**

**sum\_val DWORD ?**

**.code**

**main PROC**

 **mov esi, offset Arr1 ;put array address in esi**

 **mov eax, 0 ;initialize eax by zero for temp sum**

 **mov ecx, 5 ;initialize ecx (loop counter)**

 **;by array size**

**sum\_loop:**

 **add eax, DWORD PTR [esi]**

 **add esi, 4 ;increment esi pointer by 4**

 **;(size of array element)**

 **loop sum\_loop** ;ECX decremented implicitly by LOOP instruction

 **call writeint ;output the sum** **(which already stored in EAX)**

 **mov sum\_val, eax**

 **call CrLf**

 **exit**

**main ENDP**

**END main**

**Notes:**

* PTR is an operator that overrides the size of an operand. It is always preceded by a Type (BYTE, WORD, DWORD…etc). In the instruction add eax, DWORD PTR [esi], you can remove DWORD PTR as the assembler will assume a default size equals to the size of the second operand which in this case DWORD. If ax is used instead of eax, WORD size will be assumed and so on.
* Writeint function is a function defined in Irvine library. It prints a signed integer stored in EAX on the screen.
* We specified the length of the array and the size of each element explicitly within the code. There are operators that retrieve the length of an array (i.e. number of items) and get the type of items.
* LENGTHOF operator retrieves the length of an array. For example, the instruction mov ECX, LENGTHOF Arr1 gets the length of Arr1 array and stores it in ECX.
* TYPE operator retrieves the number of bytes allocated for each item in the

given array. For example, the instruction add esi, TYPE Arr1 adds 4 to esi if Arr1 is DWORD array, adds 2 if Arr1 is WORD array and adds 1 if Arr1 is BYTE array.

**7. Translating Standard Control Structures**

The following table shows examples of how to translate high level language program control statements into their equivalent in assembly language using CMP and conditional jump instructions (assuming unsigned values).

|  |  |  |
| --- | --- | --- |
| Statement | In C++ | In Assembly |
| IF‐ELSE | **If(i==j)****{****X = 10;****}****else****{****Y = 10;****}** | **Mov eax, i****Cmp eax, j****Jne else****Mov X, 10****Jmp endif****else:****Mov Y, 10****endif:** |
| IF | **If(i>j)****{****X = 5;****}****Y = 5;** | **Mov eax, i****Cmp eax, j****Jna endif****Mov X, 5****endif:****Mov Y, 5** |
| While‐loop | **while(i > j)****{****x += i;****i--;****}** | **Mov eax, i****while:****cmp eax, j****jna endwhile****add x, eax****dec eax****jmp while****endwhile:****mov i, eax** |

***Example: Find the Minimum Value***

This example finds the minimum value in an integer array. The array is first entered by the user then the minimum value is printed on the screen

**INCLUDE Irvine32.inc**

**ARR\_SIZE equ 5 ;size of the array**

**.DATA**

**Arr1 SDWORD ARR\_SIZE dup(0)** ;allocate SDWORD array of size ARR\_SIZE

 ;initialized by zeros

**minVal SDWORD 7fffffffh** ;initialize the min variable by the

 ;largest value SDWORD can contain

**strPrompt BYTE 'Enter array items: ', 0**

**strMsg BYTE 'The min value = ', 0**

**.CODE**

**main PROC**

**mov edx, offset strPrompt** ;put the address of strPrompt in edx

;as a parameter to WriteString

**call WriteString**

**mov ecx, LENGTHOF Arr1** ;initialize loop counter ecx

**mov edi, OFFSET Arr1**

**L1:**

**call readint ;take a signed integer from user**

**mov [edi], eax ;store this integer in the array**

**add edi, TYPE Arr1 ;slide EDI to point to the next item**

**cmp eax, minVal ;compare current integer with the min**

**jge loop\_stat ;if eax >= min, jump to LOOP statement**

**mov minVal, eax ;otherwise, update the min value**

**loop\_stat:**

**loop L1 ;loop till ecx == 0**

**mov edx, offset strMsg ;put the address of strMsg in edx**

 **;as a parameter for WriteString function**

**call WriteString**

**mov eax, minVal ;put the min value in eax as a**

 **;parameter for WriteInt function**

**call WriteInt**

**call CrLf**

**exit ;terminate the program execution**

**main ENDP**

**END main**

**Notes**

* Largest value of any signed variable is the variable that its bits are all 1’s except the sign bit (the most significant bit). Therefore, largest signed BYTE value is 7FH (+127); largest signed WORD value is 7FFFH (+32767) and so on. (Can you predict the smallest value how it should be?)
* We don’t need to store integer values entered by the user in an array to find the minimum value. We store them for demonstration purposes.