IDIV Instruction

The **IDIV (signed divide) instruction** performs signed integer division, using the same operands as DIV.

However, before executing 8-bit division, the dividend (AX) must be completely sign-extended. The remainder always has the same sign as the dividend.

Syntax:

```
840 IDIV reg/mem8
841 IDIV reg/mem16
842 IDIV reg/mem32
```

Operands:

reg/mem8: An 8-bit register or memory location containing the divisor.

reg/mem16: A 16-bit register or memory location containing the divisor.

reg/mem32: A 32-bit register or memory location containing the divisor.

Operation:

The IDIV instruction divides the signed integer dividend in the AX register by the signed integer divisor in the operand.

The quotient is stored in the AL register and the remainder is stored in the AH register.

Example 1:

The following instructions divide -48 by 5. After IDIV executes, the quotient in AL is -9 and the remainder in AH is -3:

```
847 .data
848     byteVal SBYTE -48
849     ;D0 hexadecimal
850 .code
851     mov al,byteVal
852     ;lower half of dividend
853     cbw
854     ;extend AL into AH
855     mov bl,+5
856     ;divisor
857     idiv bl
858     ;AL = -9, AH = -3
```

Explanation:

The CBW instruction sign-extends the AL register into the AX register.

This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the AX register by the BL register and stores the quotient in the AL register and the remainder in the AH register.

Example 2:

The following instructions divide -5000 by 256:

```
.data
    wordVal SWORD -5000
.code
    mov ax,wordVal
    ;dividend, low
    cwd
    ;extend AX into DX
    mov bx,+256
    ;divisor
    idiv bx
    ;quotient AX = -19, rem DX = -136
```

Explanation:

The CWD instruction sign-extends the AX register into the DX register. This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the DX:AX registers by the BX register and stores the quotient in the AX register and the remainder in the DX register.

Example 3:

The following instructions divide 50,000 by -256:

```
.data
    dwordVal SDWORD + 50000
.code
    mov eax,dwordVal
    ;dividend, low
    cdq
    ;extend EAX into EDX
    mov ebx,-256
    ;divisor
    idiv ebx
    ;quotient EAX = -195, rem EDX = +80
```

Explanation:

The CDQ instruction sign-extends the EAX register into the EDX register.

This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the EDX:EAX registers by the EBX register and stores the quotient in the EAX register and the remainder in the EDX register.

Important:

The IDIV instruction undefines all arithmetic status flag values.

The IDIV instruction can also be used to perform unsigned integer division.

However, in this case, the dividend does not need to be sign-extended.

==========

Divide Overflow

==========

A divide overflow condition occurs when the result of a division operation is too large to fit into the destination operand. This causes a processor exception and halts the current program.

The following instructions generate a divide overflow because the quotient (100h) is too large for the 8-bit AL destination register:

```
892 mov ax,1000h
893 mov bl,10h
894 div bl
895 ; AL cannot hold 100h
```

Avoiding Divide Overflow:

Use a larger destination operand. For example, instead of using the AL register, you could use the AX register or the EAX register.

Use a smaller divisor. For example, instead of dividing by 10h, you could divide by 2h. Use a combination of the above two approaches. For example, you could use the AX register as the destination operand and divide by 2h. Test the divisor before dividing to avoid division by zero.

The following code uses a 32-bit divisor and 64-bit dividend to reduce the probability of a divide overflow condition:

```
901 mov eax,1000h
902 cdq
903 mov ebx,10h
904 div ebx
905 ; EAX = 00000100h
```

Explanation:

The CDQ instruction sign-extends the EAX register into the EDX register. This creates a 64-bit dividend in the EDX:EAX registers. The DIV instruction then divides the EDX:EAX registers by the EBX register and stores the quotient in the EAX register.

The following code uses a 32-bit divisor and 64-bit dividend to reduce the probability of a divide overflow condition and tests the divisor before dividing to avoid division by zero:

```
mov eax, dividend
mov bl, divisor
cmp bl, 0
je NoDivideZero

; Not zero: continue
div bl

; ...

NoDivideZero:
; Display "Attempt to divide by zero"
```

Explanation:

The **MOV instructions load the dividend and divisor** into the EAX and BL registers, respectively. The CMP instruction compares the BL register to zero. If the BL register is equal to zero, the JE instruction jumps to the NoDivideZero label.

If the **BL register is not equal to zero**, the DIV instruction divides the EAX register by the BL register and stores the quotient in the EAX register.

The **NoDivideZero label** is where the code will jump if the divisor is zero. At this point, the code could display an error message or take other appropriate action.

==========

Implementing Arithmetic Expressions(ASM)

==========

To implement arithmetic expressions in assembly language, we need to break them down into their constituent operations. For example, the following C++ statement:

```
var4 = (var1 + var2) * var3;
```

can be broken down into the following assembly language instructions:

```
928 mov eax, var1
929 add eax, var2
930 mul var3
931 mov var4, eax
```

The first instruction loads the value of var1 into the EAX register. The second instruction adds the value of var2 to the EAX register.

The third instruction multiplies the value of var3 by the value in the EAX register and stores the result in the EAX register.

The fourth instruction stores the value in the EAX register into the var4 variable.

Handling Overflow

When performing arithmetic operations in assembly language, it is important to be aware of the possibility of overflow.

Overflow occurs when the result of an operation is too large to fit into the destination operand.

For example, the following assembly language instruction:

```
mul var3
```

will multiply the value in the EAX register by the value of var3 and store the result in the EAX register. If the product of the multiplication is too large to fit into the EAX register, overflow will occur.

To handle overflow, we can use the JC (jump on carry) instruction. The JC instruction will jump to a specified label if the carry flag is set.

The carry flag is set if there was an overflow when performing the previous arithmetic operation.

previous arithmetic operation.

The following assembly language code shows how to handle overflow when multiplying two unsigned 32-bit integers:

```
945 mov eax, var1
946 add eax, var2
947 mul var3
948 jc tooBig        ;jump if overflow
949 mov var4, eax
950 jmp next
951 tooBig:
952     ;display error message
```

If the MUL instruction generates a product larger than 32 bits, the JC instruction will jump to the tooBig label. The tooBig label can then display an error message or take other appropriate action.

Handling Signed Integers

When performing arithmetic operations on signed integers, it is important to be aware of the possibility of sign extension.

Sign extension is the process of copying the sign bit of an integer to all of the higher bits of the integer.

For example, the following assembly language instruction:

```
idiv var3
```

will divide the value in the EDX:EAX registers by the value of var3 and store the quotient in the EAX register and the remainder in the EDX register.

If the dividend is a signed integer, it is important to sign-extend the dividend into EDX before performing the division.

The following assembly language code shows how to divide two signed 32-bit integers:

32-bit integers:

```
960 mov eax, var2
961 neg eax
962 cdq              ;sign-extend dividend
963 idiv var3
964 mov ebx, edx     ;EBX = right side
```

The CDQ instruction sign-extends the EAX register into the EDX register. This ensures that the EDX:EAX registers contain the correct signed value of the dividend before the division operation is performed.

Questions:

Explain why overflow cannot occur when the MUL and one-operand IMUL instructions execute.

Overflow cannot occur because these instructions ensure that the destination operand is twice the size of the multiplicand and multiplier. This means there is always enough space to hold the result without overflowing.

How is the one-operand IMUL instruction different from MUL in the way it generates a multiplication product?

The one-operand IMUL instruction, unlike MUL, can perform signed integer multiplication. It generates a product that can be positive or negative, depending on the signs of the multiplicand and multiplier.

What has to happen for the one-operand IMUL to set the Carry and Overflow flags?

The Carry and Overflow flags are set when the product of one-operand IMUL is too large to fit into the destination operand size, signifying an overflow condition. This occurs when the result is outside the representable range for the given operand size.

When EBX is the operand in a DIV instruction, which register holds the quotient?

When EBX is the operand in a DIV instruction, the EAX register holds

the quotient.

When BX is the operand in a DIV instruction, which register holds the quotient?

When BX is the operand in a DIV instruction, the AX register holds the quotient.

When BL is the operand in a MUL instruction, which registers hold the product?

When BL is the operand in a MUL instruction, the AX and DX registers hold the product. AX contains the low 16 bits, and DX contains the high 16 bits of the 32-bit product.

Show an example of sign extension before calling the IDIV instruction with a 16-bit operand.

Sign extension is necessary when working with signed integers. Here's an example of sign extension before calling IDIV with a 16-bit operand:

```
970 movsx  eax, word ptr [your_16_bit_variable] ; Sign extend 16-bit value to 32 bits
971 idiv   ebx ; Perform signed division with the extended value
```

This code first sign-extends the 16-bit value to a 32-bit value in the EAX register before performing a signed division with the IDIV instruction.