Addition and Subtraction

INC (Increment): The INC instruction increments the value of a register or memory operand by 1. Here's the syntax:

```
INC reg/mem
```

For example, if you have a data segment with a variable myWord initialized to 1000h, you can use INC like this:

```
.data
    myWord WORD 1000h

.code
    inc myWord ; myWord = 1001h
```

DEC (Decrement): The DEC instruction decrements the value of a register or memory operand by 1. Its syntax is similar to INC:

```
DEC reg/mem
```

For instance, you can use DEC to decrement the value stored in the bx register:

```
.data
    myWord WORD 1000h

.code
    inc myWord ; myWord = 1001h
    mov bx, myWord ; Load myWord into bx
    dec bx ; bx = 1000h
```

Flags Affected: The INC and DEC instructions affect various CPU status flags, including:

- Overflow
- Sign
- Zero
- Auxiliary Carry
- Parity

The specific changes to these flags depend on the value of the destination operand. Notably, the Carry flag is not affected by these instructions, which might be surprising given its role in other arithmetic operations.

In assembly language, attention to detail is crucial, as each instruction can have subtle effects on the state of the CPU and memory. Understanding how these instructions affect flags is essential for effective programming.

ADD INSTRUCTION

The ADD instruction is used to add a source operand to a destination operand of the same size. Here's the syntax:

```
ADD dest, source
```

or

```
add dest, source
```

Assembler doesn't care about capital letters!

- The source operand remains unchanged.
- The sum of the operands is stored in the destination operand.

For example, let's add two 32-bit integers:

```
.data
    var1 DWORD 10000h
    var2 DWORD 20000h

.code
    mov eax, var1    ;EAX = 10000h
    add eax, var2    ;EAX = 30000h
```

The instruction affects various CPU flags, including Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity. How these flags change depends on the result placed in the destination operand.

SUB INSTRUCTION

The SUB instruction subtracts a source operand from a destination operand. The syntax is the same as for ADD:

```
SUB dest, source
```

- Like ADD, the source operand remains unchanged.
- The result of the subtraction is stored in the destination operand.

For example, let's subtract two 32-bit integers:

```
.data
var1 DWORD 30000h
var2 DWORD 10000h

.code
mov eax, var1     ; EAX = 30000h
sub eax, var2     ; EAX = 20000h
```

Again, this instruction affects CPU flags such as Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity based on the value stored in the destination operand.

The SUB instruction subtracts var2 from var1. So, in this case:

- var1 is the source operand, and it contains the value 30000h.
- var2 is the destination operand, and it contains the value 10000h.

The SUB instruction subtracts var2 from var1, resulting in EAX being set to 20000h. Therefore, var2 is subtracted from var1.

NEG INSTRUCTION

The NEG (negate) instruction reverses the sign of a number by converting it to its two's complement. It can be applied to registers or memory. Here's the syntax:

To find the two's complement, reverse all the bits in the destination operand and add 1.

For example, to negate the value in EAX:

```
NEG reg/mem
```

```
neg eax ; Negate the value in EAX
```

As with ADD and SUB, the NEG instruction also affects CPU flags based on the result.

IMPLEMENTING THE ARITHMETIC EXPRESSIONS

With the ADD, SUB, and NEG instructions, you can implement arithmetic expressions in assembly language. You can break down an expression into individual operations and combine them.

For instance, if you want to calculate **Rval = -Xval + (Yval - Zval)**, you can do it step by step as demonstrated in the code provided in the notes.

```
.data
Rval  SDWORD ?   ; Define a signed 32-bit variable for Rval
Xval  SDWORD 26  ; Initialize Xval to 26
Yval  SDWORD 30  ; Initialize Yval to 30
Zval  SDWORD 40  ; Initialize Zval to 40

.code
; Step 1: Negate Xval and store it in EAX
mov eax, Xval    ; EAX = 26
neg eax          ; Negate EAX, now EAX = -26

; Step 2: Subtract Zval from Yval and store it in EBX
mov ebx, Yval    ; EBX = 30
sub ebx, Zval    ; Subtract Zval from EBX, now EBX = -10

; Step 3: Add the results (EAX and EBX) and store it in Rval
add eax, ebx     ; Add EAX and EBX, result in EAX
mov Rval, eax    ; Store the result in Rval

; At this point, Rval contains the desired value: Rval = -Xval + (Yval - Zval)
```

In this code:

- We first negate Xval and store the result in EAX.
- Then, we subtract Zval from Yval and store the result in EBX.
- Finally, we add the values in EAX and EBX and store the result in Rval, which gives us Rval = -Xval + (Yval - Zval).

Please note that this code assumes that Rval, Xval, Yval, and Zval are signed 32-bit integers, as indicated in the data section.

NEGATING WITHOUT REGISTERS

You can certainly negate a variable directly in assembly language without using a separate register to store the result temporarily.

The example I provided in my previous response used registers for clarity and readability to illustrate each step of the calculation.

In practice, you may choose to optimize the code for efficiency by negating the variable directly. Here's a more concise version of the code that negates Xval and calculates Rval without using an extra register:

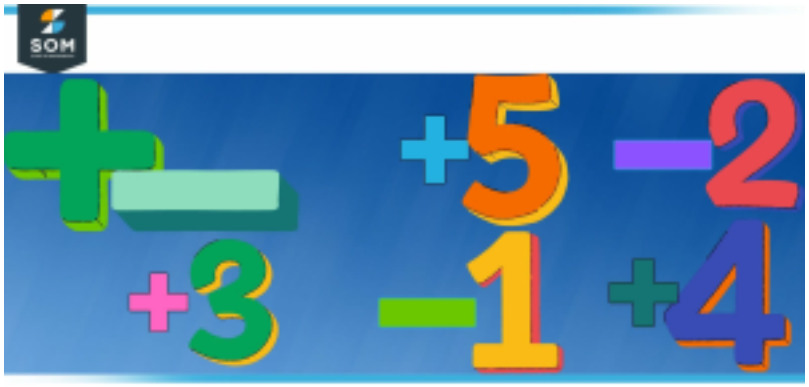
```
.data
Rval  SDWORD ?   ; Define a signed 32-bit variable for Rval
Xval  SDWORD 26  ; Initialize Xval to 26
Yval  SDWORD 30  ; Initialize Yval to 30
Zval  SDWORD 40  ; Initialize Zval to 40

.code
; Negate Xval directly
neg Xval          ; Negate Xval in place

; Calculate Rval
mov eax, Yval    ; EAX = 30
sub eax, Zval    ; Subtract Zval from EAX, now EAX = -10
add eax, Xval    ; Add Xval to EAX, now EAX = -36
mov Rval, eax    ; Store the result in Rval
```

SIGNED NUMBERS

Signed numbers represent both positive and negative values. They use a portion of the binary representation to indicate the sign of the number (positive or negative) and the remaining bits to represent the magnitude (absolute value) of the number. The most common method for representing signed numbers is using Two's Complement.



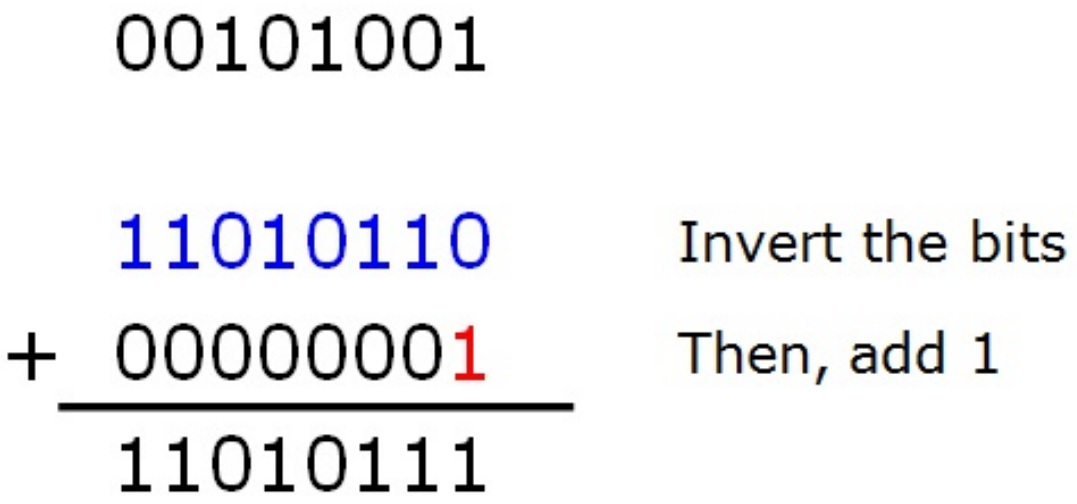
Here's how it works:

• In Two's Complement, the leftmost bit (the most significant bit) is used as the sign bit. If it's 0, the number is positive; if it's 1, the number is negative.
• The remaining bits represent the absolute value of the number. To negate a signed number, you invert all the bits (change 0s to 1s and vice versa) and then add 1 to the result.

For example:

• 0010 (binary) is +2 in Two's Complement.
• 1010 (binary) is -2 in Two's Complement. (Invert all bits, add 1: 0101 + 1 = 0110, which is -2 in decimal)

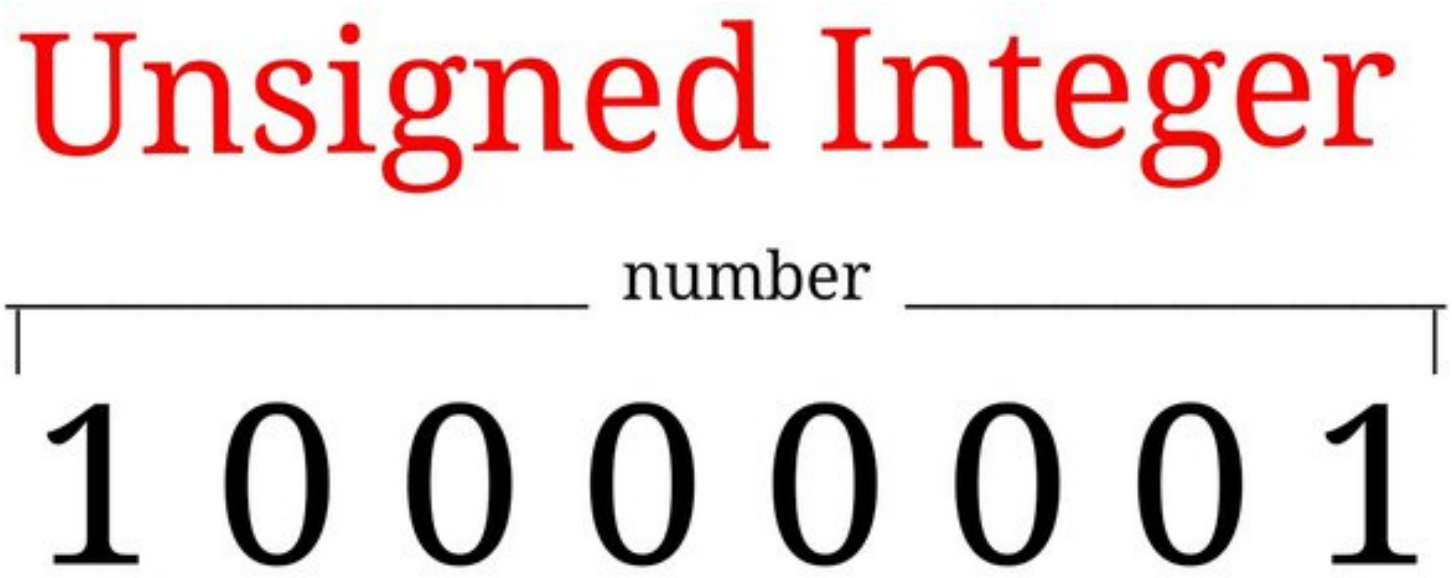
For example: Let's find out the 2's complement of given 8-bit number



The 2's complement of 00101001 is **11010111**

UNSIGNED NUMBERS

Unsigned numbers, on the other hand, only represent positive values. They use all their bits to represent the magnitude of the number, with no sign bit. This means unsigned numbers can only take on non-negative values.



For example:

- 0010 (binary) is 2 in unsigned representation.
- There is no negative representation for unsigned numbers, so concepts like negation do not apply.

In summary, signed numbers have a sign bit that allows them to represent both positive and negative values, while unsigned numbers only represent non-negative values. The choice between signed and unsigned representation depends on the specific requirements of a given problem or application.

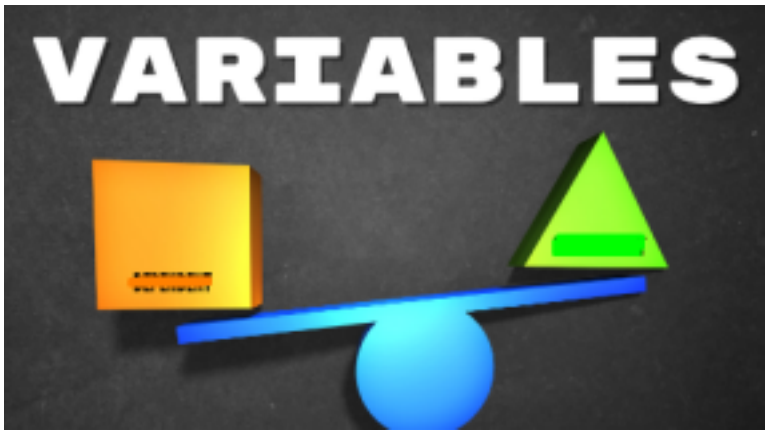
Someone might confuse in signed and unsigned eg if they meet a number binary with 1 at the front they assume its signed and its negative, because it has a 1 at the front. How to counter this confusion?

You're absolutely right that the presence of a leading 1 in a binary number might lead people to assume it's a signed (negative) number, especially if they are not familiar with the representation used. To counter this confusion, it's important to follow some best practices in both programming and documentation:

Clear Documentation: When working with binary numbers in your code or data structures, always provide clear and explicit documentation to indicate whether the numbers are signed or unsigned. Include comments or annotations that clarify the representation being used.

Use Standard Conventions: Adhere to standard conventions for representing signed and unsigned numbers. In many programming languages and assembly languages, signed numbers are represented using Two's Complement notation, while unsigned numbers use straightforward binary representation.

Variable Naming: Choose variable names that reflect the type of data they store. For example, use names like unsignedValue or signedValue to make it explicit whether a variable is intended to hold an unsigned or signed number.



Consistent Use of Data Types: Ensure that your code consistently uses the appropriate data types for signed and unsigned values. For instance, if your programming language supports specific data types like int (signed) and uint (unsigned), use them consistently.

Testing and Validation: During code development, test your calculations and operations with known inputs and expected results to verify that your assumptions about the data representation are correct.

Educate Team Members: If you're working on a team, take the time to educate your colleagues about data representation conventions and the significance of the leading bit. Encourage open communication about data representations to avoid misunderstandings.

Avoid Mixing Representations: In situations where both signed and unsigned data might be present, avoid mixing them within the same context or calculation. Keep signed and unsigned operations separate to minimize confusion.

Error Handling: Implement appropriate error handling and checks in your code to catch any unexpected or invalid data representations. This can help identify and address issues early.

By following these practices and promoting awareness of data representation conventions among your team members, you can mitigate the potential confusion associated with the leading bit of binary

numbers and ensure that your code is clear and unambiguous.