Module 0218: Subtracting in assembly language

Tak Auyeung, Ph.D.
February 18, 2010

1 About this module

- Prerequisites: [0217]
- Objectives: This module discusses how subtraction is performed in an ALU, and how subtraction and comparison affect various flags.

2 Binary subtraction

2.1 The rules

Binary subtraction has simple rules, just as binary addition as discussed in module [0217].

- $0 - 0 = 0$ with a borrow of 0.
- $1 - 0 = 1$ with a borrow of 0.
- $0 - 1 = 1$ with a borrow of 1.
- $1 - 1 = 0$ with a borrow of 0.

2.2 Logic gates of a bit subtractor

Let us define $R(x, y)$ as the result (difference) of subtracting bit $y$ from bit $x$. Likewise, $B(x, y)$ is the borrow from the next digit as the result of subtracting bit $y$ from bit $x$.

The rules states in the previous section can be captured by logical operations. For $R(x, y)$, the result is a 1 iff $x \neq y$. As a result, $R(x, y) = \overline{x}y + \overline{xy}$ (logical operators), which is the same as $R(x, y)$ for addition.

$B(x, y)$ is a 1 iff $x = 0$, $y = 1$. This can be coded by $B(x, y) = \overline{xy}$.

2.3 Combinatorial multi-bit subtractor

Let us now consider a multi-bit subtraction. Let $x$ and $y$ both have $n$ bits, and $x_i$ denote bit $i$ of $x$. We will use $d$ to represent the difference $d = x - y$. $d$ also has $n$ bits.

First, $d_0 = R(x_0, y_0)$ as there is no borrow from a less significant digit.

Second, $d_1 = R(R(x_1, y_1), B(x_0, y_0))$ because if the subtraction of bit 0 yields a borrow of 1, then $d_1$ is affected by it.

Third, $d_2 = R(R(x_2, y_2), B(x_1, y_1) + B(R(x_1, y_1), B(x_0, y_0)))$ because there are two possible sources of the borrow that affects $d_2$. The first source is $B(x_1, y_1)$, and the second source is $B(R(x_1, y_1), B(x_0, y_0))$. Although only up to one of these sources can be a 1, they still need to be combined to become the borrow that affects the computation of $d_2$.

You can see how these equations are similar to the computation of $R(x_i, y_i)$ in an addition. Just like with the case of a combinatorial multi-bit adder, the complexity of computing $d_i$ increases as the square of $i$.

2.4 Half subtractors and full subtractors

When resources (especially routing resources) is limited, it helps to use the concept of half subtractors and full subtractors. This is equivalent to the concept of half adders and full adders in module [0217]. The only difference is that instead of using “C” for $C(X, Y)$, we use “B” for $B(X, Y)$.

Each column of half subtractors is a full-subtractor.
2.5 The borrow from the most significant bit

The borrow as a result of computing $d_{n-1}$ cannot be stored in the result. However, it is still latched in a processor so that it can be used by subsequent instructions.

Note that the i386 processor (like almost all processors) combine the carry flag and the borrow flag into one. The name of the flag is usually “carry”, but it also has “borrow” as an alias.

3 Subtraction and comparison

The `sub` (subtraction) instruction of the i386 is much like the `add` instruction. However, it is important to note that in the AT&T notation, the difference is computed by subtracting the first operand from the second operand, and it is stored at the second operand.

This means that the following instruction subtracts 32 from `eax`, then the result is stored in `eax`. This essentially is decrementing `eax` by 32.

```
subl $32,%eax
```

`sub` has to follow all the same rules as `add` and `mov`, namely it can only contain up to one memory operand.

The compare (`cmp`) instruction is like a `sub` instruction, except it does not store the difference. `cmp` affects all the flags the same way as `sub`, however.

4 Flags affected by subtraction and comparison

The same four flags affected by `add` in module 0217 are also affected by `sub` and `cmp`.

4.1 The Carry flag

As already discussed, the `C` flag is the resulting borrow from computing $d_{n-1}$.

4.2 The Zero flag

This is the same as the zero flag after an addition. If the difference from a subtraction or comparison is zero, then the zero flag is set.

$$Z = d_0 + d_1 + \cdots + d_{n-1}$$

4.3 The Sign flag

This is the same as the sign flag after an addition.

$$S = d_{n-1}$$

4.4 The Overflow flag

In a subtraction, the sign of the result makes no sense when:

- non-negative subtracted from negative, but the result is non-negative.
- negative is subtracted from non-negative, but the result is negative.

Since the MSb is the sign bit, we can define

$$O = x_{n-1}y_{n-1}d_{n-1} + x_{n-1}y_{n-1}d_{n-1}$$

In this equation, $y$ is subtracted from $x$, and $d$ is the result of the subtraction.

5 Results of comparison

Because a `cmp` instruction does not store the result, its sole purpose is to affect the flags so that other instructions can use the flags.

This section explains the meanings of the flags after a `cmp` instruction.
5.1 the Z-flag

This is the easiest one. What does it mean when \( Z = 1 \) after a `cmp` instruction?

The only time a difference is zero is when the two operands are the same! As a result, the Z-flag is also sometimes known as the E-flag for “equal”.

5.2 the C-flag

The C-flag, which means borrow after a subtraction, is also somewhat simple to understand. What does it mean when a subtraction results in a non-zero borrow from the most significant digit?

Let us consider the following `pseudo` instruction:

```pseudo
cmp x, y
```

The flags are set according to the result of \( y - x \). What does it mean when \( C = 1 \)?

This is a borrow from bit \( n \) when each operand has \( n \) bits. Just based on what we understand about the role of borrowing in subtraction, we should conclude that \( x > y \). Technically, this means that \( 2^n + y - x = d \). However, since \( d \) only has \( n \) bits, we can deduce that \( 2^n + y - x < 2^n \). We can then subtract \( 2^n \) from both sides, resulting in \( +y - x < 0 \). One more step and we conclude that \( y < x \).

However, we still have an important issue to address. Under which interpretation (signed or unsigned) that we can conclude \( C = 1 \iff x < y \)? For example, in a 4-bit system, \( 1111_2 \) can be interpreted as 15 using the unsigned interpretation, or it can be interpreted as -1 using the signed interpretation.

The answer is unsigned. This is because the bit-wise subtraction that affects the borrow/carry flag is not sign-aware. In other words, \( 0001_2 - 1111_2 \) results in a borrow of 1 when \( 1111_2 \) is treated as just that, an unsigned (non-negative) number.

One can come up with counter-examples of why \( C = 1 \) does not imply \( y < x \) in the signed interpretation. In a four-bit system, try \( x = 1, y = -1 \). The 4-bit binary subtraction \( y - x \) does not result in \( C = 1 \)!

5.3 the S-flag

Let us continue to use the `pseudo` instruction:

```pseudo
cmp x, y
```

The sign flag indicates whether the difference of a subtraction is negative or not, but it cannot be used to indicate whether \( x > y \) when both numbers are interpreted signed.

As an exercise, try the following combinations (use the 4-bit system to make things simpler). First, figure out the bit patterns for each value. Then carry out the binary subtraction and determine the MSb of the result.

- \( x = 0, y = -1, S = 1, x > y \).
- \( x = 7, y = -8, S = 0, x > y \).

Because the sign flag has different values in two comparisons that should have the same conclusion, the sign flag, at least by itself, cannot be used to indicate the result of a comparison of signed numbers.

5.4 the O-flag

Even the overflow flag is not sufficient to indicate ordering in a comparison of signed numbers. Prove to yourself the following (using subtraction of 4-bit representations):

- \( x = 1, y = -8, O = 1, x > y \)
- \( x = -1, y = 7, O = 1, x < y \)

In this case, although the overflow flag is set in both cases, the orderings are actually opposite to each other. This means that the overflow flag, at least by itself, is not sufficient to indicate the ordering of two signed numbers after a comparison.
5.5 the L-flag

Although the i386 processor does not really have an L-flag, we can conceptually think there is one. The L-flag is important because it indicates ordering when signed numbers are ordered.

The previous two sections already illustrated how the S-flag and O-flag are not sufficient to indicate order when signed numbers are compared. However, when these two flags are combined, they can be used to indicate ordering. Let us define \( L = OS + \bar{OS} \). In other words, \( L = O \oplus S \) (the exclusive-or of \( O \) and \( S \)).

Let us think about this in intuitive terms.

\( OS = 1 \) means that although the sign flag is cleared, it is wrong. Since the sign flag can either be cleared or set, it means the sign flag should have been set, or that the result of the subtraction/comparison is negative (and we just don’t have enough bits to store it).

\( \bar{OS} = 1 \) means that the sign flag is correct, and it is set. This directly means the result of a subtraction or comparison is negative.

In both cases, the L-flag is set iff the result of a subtraction is negative. In other words, if the pseudo instruction is \texttt{cmp x, y}, then \( L = 1 \iff y - x < 0 \). This makes sense because \( y - x < 0 \) is equivalent to \( y < x \).