CS61 Fall 2011 Section 2
Additional Notes on Condition Codes

1 Interpreting condition codes (status flags)

The `cmp` instruction sets the four condition codes (`CF`, `ZF`, `SF`, `OF`) according to the difference between its two arguments. Subsequent operations can then either set a value or conditionally jump based on the flags set.

Remember that in ATT syntax, `cmpl src, dst` compares `dst` to `src`. Thus the following code jumps to `.L0` if `%ebx < %eax` rather than the other way around.

```
cmpl     %eax, %ebx
jl       .L0
```

So, how should the status flags be interpreted to implement `jl` (jump if *signed* less than)? Let's derive it from the behavior of `cmp`.

First, remember that `cmp S,D` yields the same status flags as computing `sub S,D` ($\mathtt{D} - \mathtt{S}$). We consider two cases.

$\mathtt{D} - \mathtt{S}$ **does not overflow:** Then $\neg\mathtt{OF}$ is true. Furthermore, then if $D < S$, the result will be negative. If $D \geq S$ the result will be zero if they are equal, or positive, if $D > S$.

Thus, if $\neg\mathtt{OF} \wedge \mathtt{SF}$ is true, $D < S$. (Remember that `SF` is true when the result is negative.)

$\mathtt{D} - \mathtt{S}$ **overflows:** If $\mathtt{D} - \mathtt{S}$ overflows, there are two possibilities: either $\mathtt{D} - \mathtt{S}$ is too large to represent (a positive overflow), or $\mathtt{D} - \mathtt{S}$ is too small to represent (a negative overflow).

In the first case, we know that `D` must be positive and `S` must be negative. We know this must be the case because if both were positive, there would be no pair of values that give a result less than the minimum representable signed negative ($\mathtt{0} - \mathtt{MAX\_INT} > \mathtt{MIN\_INT}$ due to the asymmetry of two's complement). Likewise, if `D` were negative, there is no large enough negative `S` to cause a postive overflow. If this is the case, we know then that $\mathtt{D} \not< \mathtt{S}$.

In the second case, there is a negative overflow and it must be the case that `D` is negative and `S` is positive, by a similar argument (left to the reader). Thus $\mathtt{D} < \mathtt{S}$.

But how do we tell if the overflow is positive or negative? OF doesn't distinguish! We can tell by looking at the sign of the result. If there was an overflow and the result is negative, then the correct result would have been too large (so the overflow was positive). If the result is positive, then the correct result was too small to represent (so the overflow was negative). Thus, if we had an overflow, then $\mathtt{D} < \mathtt{S}$ when $\neg\mathtt{SF}$ is true. That is $\mathtt{OF} \wedge \neg\mathtt{SF}$.

To reach the final answer, we combine these two cases:

$$\mathtt{D} < \mathtt{S} \Leftrightarrow (\neg\mathtt{OF} \wedge \mathtt{SF}) \vee (\mathtt{OF} \wedge \neg\mathtt{SF})$$
$$\mathtt{D} < \mathtt{S} \Leftrightarrow (\mathtt{OF} \oplus \mathtt{SF})$$

By a similar process, we can work out the correct logic for other comparison operations. A listing of the results is given in the book and in the lecture slides. You needn't worry too much about exactly which flags are used for each operation, but its good to know how it really works in the processor!

2 OF versus CF

OF is the *signed* overflow flag. CF is the *unsigned* overflow flag. Because most arithmetic operations work for both signed and unsigned numbers, we must always set both OF and CF appropriately. That way a later instruction can apply its own interpretation and look at the appropriate flag.

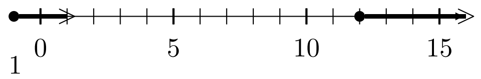
A simple way to think about when each flag should be set is to imagine the number line of representable values for each interpretation.

2.1 Example: $\mathtt{CF} \wedge \neg\mathtt{OF}$

Suppose we are computing $1100 + 0101 = 0001$. Lets consider the computation in each interpretation.

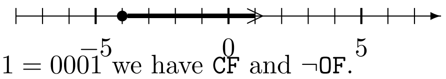
Unsigned: $12 + 5 = 17$

If we consider this with an unsigned interpretation, we start at 12 and move right (since we're adding a positive number). Once we exhaust the representable values, we "wrap around" to the bottom, arriving at positive 1. Thus, we've *overflowed* the representable values of our unsigned integers and don't get the expected value. CF should be set.



Signed: $-4 + 5 = 1$

In a signed interpretation, we start at -4 and move right (since we're still adding a positive number). No overflow occurs and we get 1.



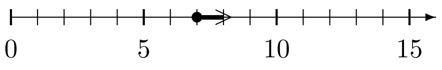
So for $1100 + 0101 = 0001$ we have CF and $\neg\mathtt{OF}$.

2.2 Example: $\neg$`OF` $\wedge$ `CF`

Let's consider another example: $0111 + 0001 = 1000$.

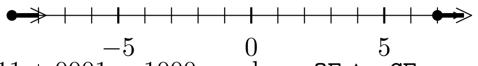
Unsigned: $7 + 1 = 8$

If we consider this with an unsigned representation, we start at 7 and move right 1. No overflow occurs and we get the expected answer, 8. `CF` is not set.



Signed: $7 + 1 = 8$

In a signed interpretation, we start at 7 and move 1 right. Unfortunately, 7 is the largest representable signed integer, so the computation wraps around to -8—an overflow. The `OF` is set.



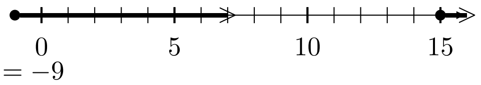
In this case, $0111 + 0001 = 1000$, we have `OF` $\wedge \neg$`CF`.

2.3 Example: `OF` $\wedge$ `CF`

Finally, let's look at $1111 + 1000 = 0111$.

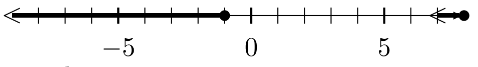
Unsigned: $15 + 8 = 23$

If we consider this unsigned, we start at 15 and move right 8. But 15 is already the maximum representable integer, so we wrap around to 7. Since this is an overflow, `CF` is set.



Signed: $-1 + -8 = -9$

In a signed representation, we start at -1 and move 8 to the *left* since we're adding a negative number. The result we expect, -9 is too small a negative number and wraps around to 7. This is a negative overflow so `OF` is set.



In this case, both `OF` and `CF` are set.