Conditions

Young W. Lim

2023-05-06 Sat

Outline

Based on

1. "Self-service Linux: Mastering the Art of Problem Determination",

Mark Wilding

1. "Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Compling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

Carry Flag (1)

- When numbers are added and subtracted,
  carry flag CF represents
  - 9th bit, if 8-bit numbers added
  - 17th bit, if 16-bit numbers added
  - 33rd bit, if 32-bit numbers added and so on.
- With addition, the carry flag CF records
  a carry out of the high order bit. For example,

```
mov al, -1    ; AL = 0x1111111
add al, 1     ; AL = 0x0000000, ZF and CF flags are set to 1
```

http://www.c-jump.com/CIS77/ASM/Flags/F77_0030_carry_flag.htm

Carry Flag (2)

- When a *larger* number is subtracted from the *smaller* one, the carry flag CF indicates a borrow. For example,

```
mov al, 6     ; AL = 0x00000110
sub al, 9     ; AL = -3, SF and CF flags are set to 1

;                       0x00000110 (6)
;  0x00001001 (9)   0x11110111 (-9)
;                       0x11111101 (6-9)  0x00000011 (3)
```

- The result is -3, represented internally as `0FDh` (binary `11111101`).

http://www.c-jump.com/CIS77/ASM/Flags/F77_0030_carry_flag.htm

Overflow Fslag (1)

- Overflow occurs with respect to the size of the data type that must accommodate the result.
- Overflow indicates that the result was
  - too large, if positive
  - too small, if negative

  to fit in the original data type

`http://www.c-jump.com/CIS77/ASM/Flags/F77_0040_overflow.htm`

Overflow Flag (2)

- When two signed 2's complement numbers are added, the overflow flag `OF` indicates one of the following:
  - *both operands* are positive and *the result* is negative
  - *both operands* are negative and *the result* is positive
- When two unsigned numbers are added, the carry flag `CF` indicates an overflow
  - there is a carry out of the leftmost (most significant) bit.

`http://www.c-jump.com/CIS77/ASM/Flags/F77_0040_overflow.htm`

Overflow Flag (3)

- Computers don't differentiate
  between signed and unsigned binary numbers.
- This makes logic circuits fast.
- programmers must distinguish
  between signed and unsigned
- must distinguish them
  when detecting an overflow after addition or subtraction.

`http://www.c-jump.com/CIS77/ASM/Flags/F77_0040_overflow.htm`

Overflow Flag (4)

- correct approach to detect the overflow
  - Overflow when adding signed numbers is indicated by the overflow flag, `OF`
  - Overflow when adding unsigned numbers is indicated by the carry flag, `CF`

http://www.c-jump.com/CIS77/ASM/Flags/F77_0040_overflow.htm

Overflow Flag (5)

```
            .DATA
    mem8    BYTE    39        ;                                    0010 0111
                              ;
            .CODE
; Addition + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
                  ; signed  unsigned     binary   hex        2's complement
mov   al, 26      ;     26        26    0001 1010   1A
inc   al          ;     +1        +1    0000 0001   01
                  ;   ----      ----
                  ;     27        27    0001 1011   1B
add   al, 76      ;    +76       +76    0100 1100   4C
                  ;   ----      ----
                  ;    103       103    0110 0111   67
add   al, [mem8]  ;    +39       +39    0010 0111   27
                  ;   ----      ----
mov   ah, al      ;   -114       142    1000 1110   8E (OF) (SF)   0111 0010
add   al, ah      ; + -114      +142    1000 1110   8E             0111 0010
                  ;   ----      ----
                  ;     28        28    0001 1100   1C (OF) (CF)
```

http://www.c-jump.com/CIS77/ASM/Flags/F77_0040_overflow.htm

Overflow Flag (6)

```
; Subtraction- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                     ; signed  unsigned       binary  hex          2's complement
 mov   al, 95        ;     95        95    0101 1111   5F
 dec   al            ;   -  1      -  1    1111 1111   FF               0000 0001
                     ;   ----      ----
                     ;     94        94    0101 1110   5E
 sub   al, 23        ;   - 23      - 23    1110 1001   E9               0001 0111
                     ;   ----      ----
                     ;     71        71    0100 0111   47
 mov   [mem8],122 ;
 sub   al, [mem8] ;  - 122     - 122    1000 0110   7A               0111 1010
                     ;   ----      ----
                     ;    -51       205    1100 1101   CD (SF) (CF)    0011 0011
 mov   ah, 119       ;
 sub   al, ah        ;  - 119     - 119    1000 1001   77               0111 0111
                     ;   ----      ----
                     ;     86        86    0101 0110   56 (OF)
```

http://www.c-jump.com/CIS77/ASM/Flags/F77_0040_overflow.htm

Overflow Flag (7)

- assume 8-bit data registers are used
- (`OF`) overflow flag :
  the result is too large to fit in the 8-bit destination operand
  - the sum of two positive signed operands exceeds 127
    interpreted as a negative number
  - the difference of two negative operands is less than -128
    interpreted as a positive number

`http://www.c-jump.com/CIS77/ASM/Flags/F77_0040_overflow.htm`

Overflow Flag ()

- assume 8-bit data registers are used
- (CF) carry flag
  the sum of two unsigned operands exceeded 255
- (SF) sign flag
  result goes below 0

http://www.c-jump.com/CIS77/ASM/Flags/F77_0040_overflow.htm

Logical operator ! and bitwise complement operator ~

- Output values
  - logical operator (!) returns either 1 or 0
  - bitwise complement operator (~) returns 1's complement
- Input values
  - in C, any non-zero value is considered as True
  - in C, only zero value is considered as False

```
----------------------------------------------------
  b = 0x00110011 (True)     C = 0x00000001 (True)

 ~b = 0x11001100 (True)    ~C = 0x11111110 (True)
 !b = 0x00000000 (False)   !C = 0x00000000 (False)
----------------------------------------------------
  b = 0x00000000 (False)    C = 0x00000000 (False)

 ~b = 0x11111111 (True)    ~C = 0x11111111 (True)
 !b = 0x00000001 (True)    !C = 0x00000001 (True)
----------------------------------------------------
```

Assumption on a, b, and C

- two operands `a` and `b` are n-bit (8, 16, or 32-bit)
- the carry flag C is 1-bit
- to negate n-bit `b`, use `~b`
- to negate 1-bit `C`, use `!C`
- `1 - C = !C`

Transformed addition

- given 2's complement,
  a subtraction operation can be
  transformed into an addition operation:

```
z = a - b
  = a + (-b)
  = a + ~b + 1
```

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

Carry-out of the transformed addition

- the carry out Cout is set / reset according to the *transformed addition* `a + ~b +1` of `a - b` subtraction operation
  - `Cout = 0` : when borrow ($a < b$)
  - `Cout = 1` : when no borrow ($a \geq b$)

`z`	`= 0 - 1`	borrow occurs since `0 < 1`
	`= 0 + fffffffe + 1`	the transformed addition
`Cout:z`	`= 0:ffffffff`	`Cout = 0` (carry-out clear)
`z`	`= 0 - 0`	no borrow occurs since `0 >= 0`
	`= 0 + ffffffff + 1`	the transformed addtion
`Cout:z`	`= 1:00000000`	`Cout = 1` (carry-out set)

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-

Inverted carry of the transformed addition

- the carry out Cout is set / reset according to the *transformed addition* `a + ~b + 1` of `a - b` subtraction operation
- inverted carry `C = !Cout`
  - `C = 1` : when borrow ($a < b$)
  - `C = 0` : when no borrow ($a \geq b$)

`z`	`= 0 - 1`	borrow occurs since `0 < 1`
	`= 0 + fffffffe + 1`	the transformed addition
`Cout:z`	`= 0:ffffffff`	`C = 1` (inverted carry set)
`z`	`= 0 - 0`	no borrow occurs since `0 >= 0`
	`= 0 + ffffffff + 1`	the transformed addtion
`Cout:z`	`= 1:00000000`	`C = 0` (inverted carry clear)

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

Binary adder

- the transformed addition is performed by a n-bit binary adder
- inputs
    - n-bit augend `X`
    - n-bit addend `Y`
    - 1-bit carry in `Cin`
- outputs
    - 1-bit carry out `Cout`
    - n-bit sum `S`

`https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i`

Multi-word addition

- for 4n-bit addition
- using 4 n-bit binary adders : 4 hardware replications

  $C_{out0}, S_0 \leftarrow X_0 + Y_0 + C_{in0}$
  $C_{out1}, S_1 \leftarrow X_1 + Y_1 + C_{in1}$
  $C_{out2}, S_2 \leftarrow X_2 + Y_2 + C_{in2}$
  $C_{out3}, S_3 \leftarrow X_3 + Y_3 + C_{in3}$

  serial connection

  $C_{in3} \leftarrow C_{out2}, \; C_{in2} \leftarrow C_{out1} \; C_{in1} \leftarrow C_{out0},$
- using only one n-bit binary adder : 4 software iterations

  $C_{out}, S \leftarrow X + Y + C_{in}$

  feedback connection

  $C_{in} \leftarrow C_{out}$

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

Transformed addition with `Cin`

- the carry out `Cout` is set / reset according to the *transformed addition* `a + ~b + Cin` which is `a + ~b + Cout` in a multi-word addition
    - in the inverted carry sytem
        - `C = !Cout` : inverted carry
        - `Cin = !C` : double negation (`Cin` $\leftarrow$ `Cout`)
        - then `a + ~b + Cout` becomes `a + ~b + !C`
    - in the normal carry sytem
        - `C = Cout` : normal carry
        - `Cin = C` : simple feedback (`Cin` $\leftarrow$ `Cout`)
        - then `a + ~b + Cout` becomes `a + ~b + C`

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-

Transformed addition in a multi-word operation

- the carry out `Cout` is set / reset according to the *transformed addition* `a + ~b + Cin` which is `a + ~b + Cout` in a multi-word addition
    - in the inverted carry sytem
        - `a + ~b + Cout` becomes `a + ~b + !C`
        - `a + ~b + !C = a + ~b + 1 - C = a - b - C`
        - therefore, `a - b + !C` is the transformed addition of `a - b - C` subtraction operation
    - in the normal carry sytem
        - `a + ~b + Cout` becomes `a + ~b + C`
        - `a + ~b + C = a + ~b + 1 - !C = a - b - !C`
        - therefore, `a - b + C` is the transformed addition of `a - b - !C` subtraction operation

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-

Borrow operation in a multi-word operation

- the carry out `Cout` is set / reset according to the *transformed addition* `a + ~b + Cin` which is `a + ~b + Cout` in a multi-word addition
  - in the inverted carry sytem
    - `a + ~b + Cout` becomes `a + ~b + !C`
    - `a - b - C` subtraction operation
    - `C` is considered as a borrow flag
  - in the normal carry sytem
    - `a + ~b + Cout` becomes `a + ~b + C`
    - `a - b - !C` subtraction operation
    - `!C` is considered as a borrow flag

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

Inverted carry and normal carry systems

- SBB (subtract with borrow, x86 instruction)

`a + ~b + Cout`	`!Cout` as borrow
`C = !Cout`	inverted carry
`Cin = !C`	double negation (`Cin` $\leftarrow$ `Cout`)
`a + ~b + !C`	subtract with borrow (`a - b - C`)
`B = C`	borrow flag (= `C`)

- SBC (subtract with carry, ARM instruction)

`a + ~b + Cout`	`Cout` as carry
`C = Cout`	normal carry
`Cin = C`	simple feedback (`Cin` $\leftarrow$ `Cout`)
`a + ~b + C`	subtract with carry (`a - b - !C`)
`B = !C`	borrow flag (= `!C`)

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

Carry updating in subtraction only

- subtract without borrowing operation `a - b`
  - the x86 uses *inverted carry system*
    - subtraction without borrowing : `a - b - 0` = `a - b - C (C=0)`
    - the transformed addition : `a + ~b + 1` = `a + ~b + !C`
    - carry `C` is the inverted carry out of the transformed addition
    - carry `C` is set when $a < b$ (borrow occurs)
  - the ARM uses *normal carry system*
    - subtraction without borrowing : `a - b - 0` = `a - b - !C (C=1)`
    - the transformed addition : `a + ~b + 1` = `a + ~b + C`
    - carry `C` is the normal carry out of the transformed addition
    - carry `C` is clear when $a < b$ (borrow occurs)

x86	inverted carry	
new C = 1	when $a < b$	borrow
new C = 0	when $a \geq b$	
ARM	normal carry	
new C = 0	when $a < b$	borrow
new C = 1	when $a \geq b$	

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

Carry updating in subtraction with borrowing

- subtract with borrowing operation `a - b - 1`
  - the x86 uses *inverted carry system*
    - subtraction with borrowing : `a - b - 1 = a - b - C (C=1)`
    - the transformed addition : `a + ~b + 0 = a + ~b + !C`
    - carry `C` is the inverted carry out of the transformed addition
    - carry `C` is set when `a` $<$ `(b+C)` (borrow occurs)
  - the ARM uses *normal carry system*
    - subtraction with borrowing : `a + b - 1 = a - b - !C (C=0)`
    - the transformed addition : `a + ~b + 0 = a + ~b + C`
    - carry `C` is the normal carry out of the transformed addition
    - carry `C` is clear when `a` $<$ `(b+!C)` (borrow occurs)

x86	inverted carry	
new `C = 1`	when `a` $<$ `(b+C)`	borrow
new `C = 0`	when `a` $\geq$ `(b+C)`	
ARM	normal carry	
new `C = 0`	when `a` $<$ `(b+!C)`	borrow
new `C = 1`	when `a` $\geq$ `(b+!C)`	

`https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-`

Performing a borrow operation in x86 and ARM

- borrow operation `a - b - BORROW`

x86	*inverted carry system*	`C` = inverted carry = borrow
SBB	subtraction with borrow	`a - b - C` (borrow = `C`)
	the transformed addition	`= a + ~b + !C`
ARM	*normal carry system*	`C` = normal carry = not(borrow)
SBC	subtraction with carry	`a - b - !C` (borrow = `!C`)
	the transformed addition	`= a + ~b + C`

x86	inverted carry	
new `C = 1`	when `a` $<$ `(b+C)`	borrow
new `C = 0`	when `a` $\geq$ `(b+C)`	
ARM	**normal carry**	
new `C = 0`	when `a` $<$ `(b+!C)`	borrow
new `C = 1`	when `a` $\geq$ `(b+!C)`	

`https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i`

The same transformed addition in x86 and ARM

- borrow operation `a - b - BORROW`

x86 `SBB`	subtraction with borrow	*inverted carry system*
	borrow = inverted carry C_1	
	`a - b -` C_1	`= a + ~b + `$!C_1$
	substitute C_1 with $!C_2$	substitute C_1 with $!C_2$
	`a - b -` $!C_2$	`= a + ~b + `C_2
ARM `SBC`	subtract with carry	*normal carry system*
	borrow = not (carry) = $!C_2$	
	`a - b -` $!C_2$	`= a + ~b + `C_2

x86	inverted carry C_1	$(= !C_2)$
new $C_1 = 1$	when `a` $<$ `(b+C)`	borrow
new $C_1 = 0$	when `a` $\geq$ `(b+C)`	
ARM	**normal carry C_2**	$(= !C_1)$
new $C_2 = 0$	when `a` $<$ `(b+!C)`	borrow
new $C_2 = 1$	when `a` $\geq$ `(b+!C)`	

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

x86 addition / subtraction instructions

add	`add` src, dest	dest $+$ src $\rightarrow$ dest
subtract	`sub` src, dest	dest $-$ src $\rightarrow$ dest
add with carry	`adc` src, dest	dest $+$ src $+$ CF $\rightarrow$ dest
subtract with borrow	`sbb` src, dest	dest $-$ src $-$ CF $\rightarrow$ dest

`https://en.wikibooks.org/wiki/X86_Assembly/Arithmetic`

ARM addition / subtraction instructions

Add	`ADD` Rd, Rn, Op2	Rd ← Rn + Op2
Subtract	`SUB` Rd, Rn, Op2	Rd ← Rn – Op2
Add with Carry	`ADC` Rd, Rn, Op2	Rd ← Rn + Op2 + C
Subtract with Carry	`SBC` Rd, Rn, Op2	Rd ← Rn – Op2 – !C
Reverse Subtract	`RSB` Rd, Rn, Op2	Rd ← Op2 – Rn
Reverse Subtract wiht Carry	`RSC` Rd, Rn, 0	Rd ← Op2 – Rn – !C

`https://www.davespace.co.uk/arm/introduction-to-arm/arithmetic.html`

(1) Subtraction and transformed addition

- `SBB` (subtract with borrow, x86 instruction)
  `a - b - C = a + ~b + 1 - C = a + ~b + !C`
  - `a - b - C` (subtraction)
    `C` is used as the borrow of a previous subtraction
  - `a + ~b + !C` (transformed addition)
    `!C` is the carry-in of the transformed addition
- `SBC` (subtract with carry, ARM instruction)
  `a - b - !C = a + ~b + 1 - !C = a + ~b + C`
  - `a - b - !C` (subtraction)
    `!C` is used as the borrow of a previous subtraction
  - `a + ~b + C` (transformed addition)
    `C` is the carry-in of the transformed addition

`https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i`

(2) Carry in and carry out of an adder

- SBB (subtract with borrow, x86 instruction)
  `a - b - C = a + ~b + 1 - C`
  `= a + ~b + !C` : the transformed addition
  - C is the inverted carry-out of the transformed addition
  - !C is the carry-in of the transformed addition
  - C is *updated* as a result of the transformed addition
  - C is used as a borrow flag
- SBC (subtract with carry, ARM instruction)
  `a - b - !C = a + ~b + 1 - !C`
  `= a + ~b + C` : the transformed addition
  - C is the normal carry-out of the transformed addition
  - C is the carry-in of the transformed addition
  - C is *updated* as a result of the transformed addition
  - !C is used as a borrow flag

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

(3) Borrow operation

- SBB (subtract with borrow, x86 instruction)
  - `a - b - C = a + ~b + !C`
    - `C` = borrow
    - `!C` = Cin of the transformed addition

if read old `C = 0`	no borrow	perform `a - b - 0 = a + ~b + 1`
if read old `C = 1`	borrow	perform `a - b - 1 = a + ~b + 0`

- SBC (subtract with carry, ARM instruction)
  - `a - b - !C = a + ~b + C`
    - `!C` = borrow
    - `C` = Cin of the transformed addition

if read old `C = 0`	borrow	perform `a - b - 1 = a + ~b + 0`
if read old `C = 1`	no borrow	perform `a - b - 0 = a + ~b + 1`

`https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i`

(4) Carry updating U

- SBB (subtract with borrow, x86 instruction)
  - `a - b - C = a + ~b + !C`
    - new `C` = inverted Cout of the transformed addition
    - new `C` = borrow for the next stage

write new `C = 0`	no borrow	if a $\geq$ (b + old `C`)
write new `C = 1`	borrow	if a $<$ (b + old `C`)

- SBC (subtract with carry, ARM instruction)
  - `a - b - !C = a + ~b + C`
    - new `C` = normal Cout of the transformed addition
    - new `!C` = borrow for next stage

write new `C = 0`	borrow	if a $<$ (b + old `!C`)
write new `C = 1`	no borrow	if a $\geq$ (b + old `!C`)

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

(5) SBB and SBC instructions ~

- SBB (subtract with borrow, x86 instruction)
  - borrow is carry (CF)

`sbb src, dest`	(`dest - src - CF` $\rightarrow$ `dest`)

  - new carry is set to the inverted carry of the transformed addition

write new CF = 0	no borrow	if dest $\geq$ (src + old CF)
write new CF = 1	borrow	if dest $<$ (src + old CF)

- SBC (subtract with carry, ARM instruction)
  - borrow is not carry (!C)

`SBC Rd, Rn, Op2`	(`Rd` $\leftarrow$ `Rn - Op2 - !C`)

  - new carry is set to the normal carry of theIP transformed addition

write new CF = 0	borrow	if Rn $<$ (Op2 + old !C)
write new CF = 1	no borrow	if Rn $\geq$ (Op2 + old !C)

https://stackoverflow.com/questions/41253124/i-cant-understand-some-instructions-i

SBB, SBC, and SUB instructions

1. Subtract with borrow (`SBB`, x86, inverted carry, `borrow=C`)

   `a - b - C = a + ~b + 1 - C = a + ~b + !C`

$C = 0$	no borrow	`a + ~b + 1`	
$C = 1$	borrow	`a + ~b + 0`	`(B = C)`

1. Subtract with carry (`SBC`, ARM, normal carry, `borrow=!C`)

   `a - b - !C = a + ~b + 1 - !C = a + ~b + C`

$C = 0$	borrow	`a + ~b + 0`	`(B = !C)`
$C = 1$	no borrow	`a + ~b + 1`	

1. Subtract without carry and borrow

   `a - b = a + ~b + 1`

`https://en.wikipedia.org/wiki/Carry_flag`

Subtraction with borrowing

	SBB (x86) inverted carry C Borrow when old C=1	SBC (ARM) normal carry C Borrow when old C=0
subtraction	a - b - C	a - b - !C
old C = 0	a - b - 0	a - b - 1 (B)
old C = 1	a - b - 1 (B)	a - b - 0
implementation	a + ~b + !C	a + ~b + C
old C = 0	a + ~b + 1	a + ~b + 0 (B)
old C = 1	a + ~b + 0 (B)	a + ~b + 1
carry updating	$a < (b + C)$	$a \geq (b + !C)$
new C = 0	$a \geq (b + \text{old } C)$	$a < (b + \text{old } !C)$
new C = 1	$a < (b + \text{old } C)$	$a \geq (b + \text{old } !C)$

- old C is to be read for a subtraction with borrowing operation
- new C is to be written as a result of a subtraction operation

https://en.wikipedia.org/wiki/Carry_flag

Subtraction only

	SUB (x86) inverted carry C no Borrow, old C=0	SUB (ARM) normal carry C no Borrow, old C=1
subtraction	a - b - C	a - b - !C
old C = 0	a - b - 0 (nB)	
old C = 1		a - b - 0 (nB)
implementation	a + ~b + !C	a + ~b + C
old C = 0	a + ~b + 1 (nB)	
old C = 1		a + ~b + 1 (nB)
carry updating	$a < b$	$a \geq b$
new C = 0	$a \geq b$	$a < b$
new C = 1	$a < b$	$a \geq b$

- SUB is compatible with SBB when old C=0 (x86)
- SUB is compatible with SBC when old C=1 (ARM)

https://en.wikipedia.org/wiki/Carry_flag

x86 SBB - Subtraction with borrowing

- a SBB (SuBtract with Borrow) x86 instruction
  - the inverted carry `C` is used as a borrow flag
    `a - b - C`
  - replace `a - b` with `a + ~b + 1`, then
    `(a + ~b + 1) - C = a + ~b + (1 - C)`
  - in an ALU adder implentation,
    `a + ~b + !C` is computed
  - the carry out of the ALU adder is inverted (inverted carry `C`)
  - inverted carry `C` is negated to be used as a carry input (`!C`)
- the carry bit is updated
  - `C = 0` if `a >= (b+C)` (no borrow)
  - `C = 1` if `a < (b+C)` (borrow)

https://en.wikipedia.org/wiki/Carry_flag

x86 SUB - Subtraction only

- a SUB x86 instruction
  - performs `a - b = a - b - 0 = a - b - C`
    as if the borrow bit were *clear* (`C = 0`)
  - computes `a - b` as
    `a + ~b + 1 = a + ~b + !0 = a + ~b + !C`
- the carry bit is updated
  - `C = 0` if `a >= b` (no borrow)
  - `C = 1` if `a < b` (borrow)

https://en.wikipedia.org/wiki/Carry_flag

ARM SBC - Subtraction with borrowing

- a SBC (SuBtract with Carry) ARM instruction
  - the normal carry `C` is negated to be used as a borrow flag (`!C`)
    `a - b - !C`
  - replace `a - b` with `a + ~b + 1`, then
    `(a + ~b + 1) - !C = a + ~b + (1 - !C)`
  - in an ALU adder implentation,
    `a + ~b + C` is computed
  - the carry out of the ALU adder is used directly (normal carry `C`)
  - normal carry `C` is used directly as a carry input (`C`)
- the carry bit is updated
  - `C = 0` if `a < (b+!C)` (borrow)
  - `C = 1` if `a >= (b+!C)` (no borrow)

https://en.wikipedia.org/wiki/Carry_flag

ARM SUB - Subtraction only

- a SUB ARM instruction
  - performs `a - b = a - b - 0 = a - b - !C` as if the borrow bit were *clear* (`!C = 0`)
  - computes `a - b` as `a + ~b + 1 = a + ~b + C`
- the carry bit is updated
  - `C = 0` if `a < b` (borrow)
  - `C = 1` if `a >= b` (`!B = C`, no borrow)

https://en.wikipedia.org/wiki/Carry_flag

Subtraction methods of various processors (1)

- the first approach : subtract with borrow
  - The 8080, 6800, Z80, 8051, x86 and 68k families (among others) use a borrow bit.
- the second approach : subtract with carry
  - The System/360, 6502, MSP430, COP8, ARM and PowerPC processors use this convention.
  - The 6502 is a particularly well-known example because it does not have a subtract without carry operation, so programmers must ensure that the carry flag is set before every subtract operation where a borrow is not required.

`https://en.wikipedia.org/wiki/Carry_flag`

Subtraction methods of various processors (2)

- However, there are exceptions in both directions; the VAX, NS320xx, and Atmel AVR architectures
  - use the borrow bit convention (inverted carry),
  - `a - b - C = a + ~b + !C` operation is called subtract with carry (`SBWC`, `SUBC` and `SBC`).
- The PA-RISC and PICmicro architectures
  - use the carry bit convention (normal carry),
  - `a - b - !C = a + ~b + C` operation is called subtract with borrow (`SUBB` and `SUBWFB`).

https://en.wikipedia.org/wiki/Carry_flag

ADC instruction (1)

- The ADC (add with carry) instruction adds both a source operand and the contents of the Carry flag to a destination operand:

  ```
  ADC op1, op2    ; op1 += op2, op1 += CF
  ```

- The instruction formats are the same as for the ADD instruction:

  ```
  ADC reg, reg
  ADC mem, reg
  ADC reg, mem
  ADC mem, imm
  ADC reg, imm
  ```

http://www.c-jump.com/CIS77/MLabs/M11arithmetic/M11_0180_sbb_instruction.htm

ADC instruction (2)

- The ADC instruction does not distinguish between signed or unsigned operands.
- Instead, the processor evaluates the result for both data types and sets
  - OF flag to indicate a carry out from the signed result.
  - CF flag to indicate a carry out from the unsigned result.
- The sign flag SF indicates the sign of the signed result.
- The ADC instruction is usually executed as part of a chained multibyte or multiword addition, in which an ADD or ADC instruction is followed by another ADC instruction.

http://www.c-jump.com/CIS77/MLabs/M11arithmetic/M11_0180_sbb_instruction.htm

ADC instruction (3)

- The following fragment adds two 8-bit integers (`FFh + FFh`), producing a 16-bit sum in `DL:AL`, which is `01h:FEh`.

```
mov dl, 0
mov al, 0FFh
add al, 0FFh  ; AL = FEh, CF = 1
adc dl, 0     ; DL += CF, add "leftover" carry
```

- Similarly, the following instructions add two 32-bit integers (`FFFFFFFFh + FFFFFFFFh`).
- The result is a 64-bit sum in `EDX:EAX`, `00000001h:FFFFFFFEh`,

```
mov edx, 0
mov eax, 0FFFFFFFFh
add eax, 0FFFFFFFFh
adc edx, 0    ; EDX += CF, add "leftover" carry
```

http://www.c-jump.com/CIS77/MLabs/M11arithmetic/M11_0180_sbb_instruction.htm

ADC instruction (4)

- The following instructions add two 64-bit numbers received in EBX:EAX and EDX:ECX:
  - The result is returned in EBX:EAX.
  - Overflow/underflow conditions are indicated by the Carry flag.

    ```
    add eax, ecx ; add low parts EAX += ECX, set CF
    adc ebx, edx ; add high parts EBX += EDX, EBX += CF
    ; The result is in EBX:EAX
    ; NOTE: check CF or OF for overflow (*)
    ```

- The 64-bit subtraction is also simple and similar to the 64-bit addition:

  ```
  sub eax, ecx ; subtract low parts EAX -= ECX, set CF (borrow)
  sbb ebx, edx ; subtract high parts EBX -= EDX, EBX -= CF
  ; The result is in EBX:EAX
  ; NOTE: check CF or OF for overflow (*)
  ```

- The Carry flag CF is normally used for unsigned arithmetic.
- The Overflow flag OF is normally used for signed arithmetic.

http://www.c-jump.com/CIS77/MLabs/M11arithmetic/M11_0180_sbb_instruction.htm

SBB instruction (1)

- After subtraction, the carry flag `CF = 1` indicates a need for a borrow.
- The `SBB` (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag `CF` from a destination operand:

```
SBB op1, op2   ; op1 -= op2, op1 -= CF
```

- The possible operands are the same as for the `ADC` instruction.
- The following fragment of code performs 64-bit subtraction:

```
mov edx, 1 ; upper half
mov eax, 0 ; lower half
sub eax, 1 ; subtract 1 from the lower half, set CF.
sbb edx, 0 ; subtract carry CF from the upper half.
```

http://www.c-jump.com/CIS77/MLabs/M11arithmetic/M11_0180_sbb_instruction.htm

SBB instruction (2)

- The example logic:
    - Sets `EDX:EAX` to `00000001h:00000000h`
    - Subtracts 1 from the value in `EDX:EAX`
        1. The lower 32 bits are subtracted first, setting the Carry flag `CF`
        2. The upper 32 bits are subtracted next, including the Carry flag.

`http://www.c-jump.com/CIS77/MLabs/M11arithmetic/M11_0180_sbb_instruction.htm`

SBB instruction (3)

- When an immediate value is used in SBB as an operand, it is sign-extended to the length of the destination operand.
- The SBB instruction does not distinguish between signed or unsigned operands.
- Instead, the processor evaluates the result for both data types and sets the
  - OF flag to indicate a borrow in the signed result.
  - CF flag to indicate a borrow in the unsigned result.
- The SF flag indicates the sign of the signed result.
- The SBB instruction is usually executed as part of a chained multibyte or multiword subtraction, in which a SUB or SBB instruction is followed by another SBB instruction.

http://www.c-jump.com/CIS77/MLabs/M11arithmetic/M11_0180_sbb_instruction.htm

INC / DEC (1)

- The INC instruction adds one to the destination operand, while preserving the state of the carry flag CF:
  - The destination operand can be a register or a memory location.
  - This instruction allows a loop counter to be updated without disturbing the CF flag.
    (Use ADD instruction with an immediate operand of 1 to perform an increment operation that does update the CF flag.)
- The DEC instruction subtracts one from the destination operand, while preserving the state of the CF flag.
  (To perform a decrement operation that does update the CF flag, use a SUB instruction with an immediate operand of 1.)

`http://www.c-jump.com/CIS77/ASM/Flags/F77_0070_inc_dec.htm`

INC / DEC (2)

- Especially useful for incrementing and decrementing counters.
- A register is the best place to keep a counter.
- The INC and DEC instructions
  - always treat integers as unsigned values
  - never update the carry flag CF, which would otherwise (i.e. ADD and SUB) be updated for carries and borrows.
- The instructions affect the OF, SF, ZF, AF, and PF flags just like addition and subtraction of one.

http://www.c-jump.com/CIS77/ASM/Flags/F77_0070_inc_dec.htm

INC / DEC (3)

```
 xor al, al   ; Sets AL = 0. XOR instruction always clears OF and CF flags.
    mov bl, 0FEh
    inc bl       ; 0FFh SF = 1, CF flag not affected.
    inc bl       ; 000h SF = 0, ZF = 1, CF flag not affected.
```

```
    BL 1111 1110 (0xFE)   Carry Flag 0
INC BL 1111 1111 (0xFF)   Carry Flag 0
INC BL 0000 0000 (0x00)   Carry Flag 0
```

http://www.c-jump.com/CIS77/ASM/Flags/F77_0070_inc_dec.htm

TOC: Conditional codes

Condition codes (1)

- When the x86 Arithmetic Logic Unit (ALU) performs operations like `NOT` and `ADD`, it flags the results of these operations ("became zero", "overflowed", "became negative") in a special 16-bit `FLAGS` register
- 32-bit processors upgraded this to 32 bits (`EFLAGS`)
- 64-bit processors upgraded this to 64 bits (`RFLAGS`)

https://riptutorial.com/x86/example/6976/flags-register

Condition codes (2)

Condition Code	Name	Definition
`E, Z`	Equal, Zero	`ZF == 1`
`S`	Overflow	`OF == 1`
`P`	Signed	`SF == 1`
`O`	Parity	`PF == 1`
`NE, NZ`	Not Equal, Not Zero	`ZF == 0`
`NO`	No Overflow	`OF == 0`
`NP`	Not Signed	`SF == 0`
`NS`	No Parity	`PF == 0`

`https://riptutorial.com/x86/example/6976/flags-register`

Condition codes (3)

Condition Code	Name	Definition
NC,	No Carry,	CF==0
AE, NB	Above or Equal, Not Below	CF==0
BE, NA	Above, Not Below or Equal	CF==0 and ZF==0
A, NBE	Below or Equal, Not Above	CF==1 or ZF==1
GE, NL	Greater or Equal, Not Less	SF==OF
L, NGE	Less, Not Greater or Equal	SF!=OF
G, NLE	Greater, Not Less or Equal	ZF==0 and SF==OF
LE, NG	Less or Equal, Not Greater	ZF==1 or SF!=OF

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_

Condition codes (4) `ZF` (zero flag)

- Set whenever the previous arithmetic result was zero.
- Can be used by

`jz`	jump if last result was zero
`jnz`	jump if last result was <u>not</u> zero
`je`	jump if equal, alias of `jz`
`jne`	jump if not equal, alias of `jnz`

  - because if the difference is <u>zero</u>,
    then the two values are <u>equal</u>

`https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html`

Condition codes (5) CF (carry flag)

- Contains the bit that carries out of an addition or subtraction.
- Can be used by the `jc` (jump if carry flagis set) instruction.
- Set by all the arithmetic instructions.
- Can be added into another arithmetic operation with `adc` (add with carry).
  - For example, you can preserve the bit overflowing out of an `add` using a subsequent `adc`
  - For example, here we do a tiny 16-bit add between `cx` and `si`, that overflows. We can catch the overflow bit and fold it into the next higher add:

https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html

Condition codes (6) CF (carry flag)

- adc is used in the compiler's implementation of
  the 64-bit long long datatype,
  and in general in "multiple precision arithmetic" software,
  like the GNU Multiple Precision Arithmetic Library.

https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html

Condition codes (7) CF (carry flag)

- The carry flag (or overflow flag below) could also be used to implement overflow checking in a careful compiler, like Java!
- The carry and zero flags are also used by the unsigned comparison instructions:

jb	jump if unsigned below
jbe	jump if unsigned below or equal
ja	jump if unsigned above
jae	jump if unsigned above or equal

  in a fairly obvious way.
  For example, a carry means a negative result, so a<b.
  The zero flag means a==b

https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html

Condition codes (8) SF (sign flag)

- indicates a negative signed result.
- Used together with OF to implement signed comparison.

`https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html`

Condition codes (9) OF (overflow flag)

- Set by subtract, add, and compare, and used in the signed comparison instructions

`jl`	jump if less than
`jle`	jump if less than or equal to
`jg`	jump if greater than
`jge`	jump if greater than or equal to

  instructions.

https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html

Condition codes (10) `OF` (overflow flag)

- `jae`: jump if above or equal
    - unsigned >=
    - jump if `CF==0`
    - compute `a - b`
    - if `a - b` is positive or zero (`a >= b`)
      then `CF==0` and jump is taken
    - if `a - b` is negative (`a < b`)
      then `CF==1`, and jump is <u>not</u> taken

`https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html`

Condition codes (11) OF (overflow flag)

- jge: jump if greater or equal
  - signed >=
  - jump if SF==OF
  - if no overflow occurs in the signed a - b,
    then OF==0 and SF is correct
    SF==0 (positive result a >= 0)
    SF==1 (negative result a < 0)
    (jge is the same as jae)
  - if an overflow occurs in the signed a - b,
    then OF==1 and SF is not correct
    SF==1 (corrected positive a >= 0)
    SF==0 (corrected negative a < 0)
    (jge is not the same as jae)

https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html

Condition codes (12) OF (overflow flag)

- jge: jump if greater or equal
  - signed >=
  - jump if `SF==OF`
  - in a signed compare, a carry happens
    if we're comparing negative numbers,
    so `CF` must not be used
  - if an overflow occurs, then the sign bit is wrong,
    so if `OF==1`, we compare `SF==1`,
    which flips the comparison back the right way again.

`https://www.cs.uaf.edu/2009/fall/cs301/lecture/12_07_flags.html`

Essential flags

Z	Zero flag	destination equals zero
S	Sign flag	destination is negative
C	Carry flag	unsigned value out of range
O	Overflow flag	signed value out of range

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_

Zero Flag ZF

- Whenever the destination operand equals Zero, the Zero flag is set

ZF examples

```
movw $1, %cx
subw $1, %cx              ; %cx = 0, ZF = 1
movw $0xFFFF, %ax
incw %ax                  ; AX = 0, ZF = 1
incw %ax                  ; AX = 1, ZF = 0
```

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_

Sign Flag SF

- the Sign flag is set when the destination operand is negative
- the Sign flag is clear when the destination operand is positive

SF examples

```
movw $0, %cx
subw $1, %cx              ; %cx = -1, SF = 1
addw $2, %cx              ; %cx =  1, SF = 0
```

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_

Carry Flag CF

- Addition : copy carry out of MSB to CF
- Subtraction : copy inverted carry out of MSB to CF
- INC / DEC : not affect CF
- Applying NEG to a nonzero operand sets CF

CF examples

```
movw $0x00ff, %cx
addw $1,      %ax        ; %ax = 0x0100, SF = 0, ZF = 0, CF = 0
subw $1,      %ax        ; %cx = 0x00ff, SF = 0, ZF = 0, CF = 0
addb %1,      %al        ; %al =   0x00, SF = 0, ZF = 1, CF = 1
movb $0x6c,   %bh
addb %0x95,   %bh        ; %bh =   0x01, SF = 0, ZF = 0, CF = 1

movb $2,      %al
subb $3,      %al        ; %al =   0xff, SF = 1, ZF = 0, CF = 1
```

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_

Overflow Flag OF

- the overflow flag is set when the signed result of an operation is invalid or out of range
  - case 1: adding two positive operands produces a negative number
  - case 2: adding two negative operands produces a positive number

OF examples

```
movb $+127, %al
addb $1,    %al          ; %al = -128,  OF = 1

movb $0x7F, %al
addb $1,    %al          ; %al = 0x80,  OF = 1

movb $0x80, %al          ; 0x80 + 0x92 = 0x112
addb $0x92, %al          ; %al = 0x12,  OF = 1

movb $-2,   %al          ; 0xfe + 0x7f = 0x17d
addb $+127  %al          ; %al = 0x7d,  OF = 0
```

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_

Signed / Unsigned Integers

- all CPU instructions operate exactly the same on signed and unsigned integers
- the CPU canot distinguish between signed and unsigned integers
- the programmer are soley responsible for using the correct data type with each instruciton

`https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_x`

Overflow / Carry Flags (1)

- `ADD` instruction
  - `CF` : (Carry out of the MSB)
  - `OF` : (Carry out of the MSB) $\bigoplus$ (Carry into the MSB)
- `SUB` instruction
  - `CF` : ~(Carry out of the MSB)
  - `OF` : (Carry out of the MSB) $\bigoplus$ (Carry into the MSB)

`https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_`

Overflow / Carry Flags (2)

	ADD	SUB
CF	C_n	$\overline{C_n}$
OF	$C_n \bigoplus C_{n-1}$	$C_n \bigoplus C_{n-1}$

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_x

Condition Codes (1)

- condition code registers describe attributes
  of the most recent arithmetic or logical operation
- these registers can be tested to perform conditional branches
- the most useful condition codes are as belows

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Condition Codes (2)

- as a result of the most recent operation

CF	a carry was generated out of the msb used to detect overflow for unsigned operations
ZF	a zero was yielded
SF	a negative value was yielded
OF	a 2's complement overflow was happened either neagtive or positive

Condition Codes and `c = a+b` (1)

- assume `addl` is used to perform `t = a + b` and `a, b, t` are of type `int`

CF	unsigned overflow	`(unsigned t) < (unsigned a)`
ZF	zero	`(t == 0)`
SF	negative	`(t < 0)`
OF	signed overflow	`(a < 0 == b < 0) && (t < 0 != a < 0)`

Condition Codes and `c = a+b` (2)

CF	(unsigned t) < (unsigned a)	mag(t) < mag(a) if C=1
ZF	(t == 0)	zero t
SF	(t < 0)	negative t
OF	(a<0 `= b<0) && (t<0 !` a<0)	sign(a) `= sign(b) !` sign(t)

Setting condition codes without altering registers (1)

- Compare and test

`cmpb S2, S1`	`S1 - S2`	Compare bytes
`cmpw S2, S1`	`S1 - S2`	Compare words
`cmpl S2, S1`	`S1 - S2`	Compare double words
`testb S2, S1`	`S1 & S2`	Test bytes
`testw S2, S1`	`S1 & S2`	Test words
`testl S2, S1`	`S1 & S2`	Test double words

Setting condition codes without altering registers (2)

- Compare and test

`cmpb S2, S1`	`-S2 + S1`	Compare bytes
`cmpw S2, S1`	`-S2 + S1`	Compare words
`cmpl S2, S1`	`-S2 + S1`	Compare double words
`testb S2, S1`	`S2 & S1`	Test bytes
`testw S2, S1`	`S2 & S1`	Test words
`testl S2, S1`	`S2 & S1`	Test double words

CMP instruction (1)

- `cmpb op1, op2`
- `cmpw op1, op2`
- `cmpl op1, op2`
- `NULL $\leftarrow$ op2 - op1`
    - subtracts the contents of the *src* operand *op1*
      from the *dest* operand *op2*
    - discard the results, only the flag register is affected

CMP instruction (2)

- `cmpb op1, op2`
- `cmpw op1, op2`
- `cmpl op1, op2`

Condition	Signed Compare	Unsigned Compare
op1 < op2	ZF == 0 && SF == OF	CF == 0 && ZF == 0
op1 < op2=	SF == OF	CF == 0
op1 = op2=	ZF == 1	ZF == 1
op1 > op2=	ZF == 1 or SF != OF	CF == 1 or ZF ==1
op1 > op2	SF != OF	CF ==1

TEST instruction

- `testb src, dest`
- `testw src, dest`
- `testl src, dest`
- NULL $\leftarrow$ dest & src
    - ands the contents of the src operand with the dest operand
    - discard the results, only the flag register is affected

CF and OF in binary arithmetic (1)

- do not confuse the carry flag
  with the overflow flag
  in integer arithmetic.
- the ALU always sets these flags appropriately
  when doing any integer math.
- these flags can occur on its own, or both together.

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

CF and OF in binary arithmetic (2)

- the CPU's ALU doesn't care or know
  whether signed or unsigned computations are performed;
- the ALU just performs integer arithmetic and
  sets the flags appropriately.
- It's up to the programmer to know
  which flag to check after the arithmetic is done.

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

CF and OF in binary arithmetic (3)

- if word is treated as an unsigned number,
  - the carry flag must be checked to see
    if the result is wrong or not
  - the overflow flag is irrelevant to an unsigned number

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

CF and OF in binary arithmetic (4)

- if word is treated as an signed number,
    - the overflow flag must be checked to see if the result is wrong or not
    - the carry flag is irrelevant to an signed number

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

CF and OF in binary arithmetic (5)

- In unsigned arithmetic, watch the carry flag to detect errors.
- In unsigned arithmetic, the overflow flag tells you nothing interesting.
- In signed arithmetic, watch the overflow flag to detect errors.
- In signed arithmetic, the carry flag tells you nothing interesting.

	carry flag	overflow flag
unsigned arithmetic	check	x
signed arithmetic	x	check

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

CF and OF in binary arithmetic (6)

- Do not confuse the English verb "to overflow" with the "overflow flag" in the ALU.
- The verb "to overflow" is used casually to indicate that some math result doesn't fit in the number of bits available;
- it could be integer math, or floating-point math, or whatever.
- The "overflow flag" is set specifically by the ALU as described below, and it isn't the same as the casual English verb "to overflow".

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

CF and OF in binary arithmetic (7)

- In English, we may say "the binary/integer math overflowed the number of bits available for the result, causing the carry flag to come on".
- Note how this English usage of the verb "to overflow" is not the same as saying "the overflow flag is on".
- A math result can overflow (the verb) the number of bits available without turning on the ALU "overflow" flag.

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Carry flag (1)

The rules for setting the carry flag are two:

1. The carry flag is set
   if the addition of two numbers causes a carry
   out of the most significant (leftmost) bits added.
   `1111 + 0001 = 0000` (carry flag is turned on)

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Carry flag (2)

The rules for setting the carry flag are two:

2. The carry (borrow) flag is also set
   if the subtraction of two numbers requires a borrow
   into the most significant (leftmost) bits subtracted.
   0000 - 0001 = 1111 (carry flag is turned on)
   `unsigned arithmetic`

   0000 + 1111 = 1111 (2's complement addition – no carry)
   `signed arithmetic`

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Carry flag (3)

Otherwise, the carry flag is turned off (zero).

- `0111 + 0001 = 1000` (carry flag is turned off [zero])
- `1000 - 0001 = 0111` (carry flag is turned off [zero]

  `unsigned arithmetic`

  `1000 + 1111 = 0111` (2's complement addition – carry set)

  `signed arith`

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Carry flag (4)

- In unsigned arithmetic,
  watch the carry flag to detect errors.
- In signed arithmetic,
  the carry flag is useless

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Overflow flag (1-1)

The rules for setting the overflow flag are two:

1. If the sum of two numbers with the sign bits off yields a result number with the sign bit on, the "overflow" flag is turned on.

   0100 + 0100 = 1000 (overflow flag is turned on)
   0100 - 1100 = 1000 (2's complement subtraction)

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Overflow flag (1-2)

The rules for setting the overflow flag are two:

2. If the sum of two numbers with the sign bits on yields a result number with the sign bit off, the "overflow" flag is turned on.

```
1001 + 1001 = 0010 (overflow flag is turned on)
1001 - 0111 = 0010 (2's complement subtraction)
```

0111 1000 1001

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Overflow flag (2)

- Otherwise, the overflow flag is turned off.
  0100 + 0001 = 0101 (overflow flag is turned off)
  0110 + 1001 = 1111 (overflow flag is turned off)
  1000 + 0001 = 1001 (overflow flag is turned off)
  1100 + 1100 = 1000 (overflow flag is turned off)

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Overflow flag (3)

- Note that you only need to look at the sign bits (leftmost) of the three numbers to decide if the overflow flag is turned on or off.
- If you are doing two's complement (signed) arithmetic, overflow flag on means the answer is wrong - you added two positive numbers and got a negative, or you added two negative numbers and got a positive.
- If you are doing unsigned arithmetic, the overflow flag means nothing and should be ignored.

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Overflow flag (4)

- The rules for two's complement detect errors by examining the sign of the result. A negative and positive added together cannot be wrong, because the sum is between the addends. Since both of the addends fit within the allowable range of numbers, and their sum is between them, it must fit as well. Mixed-sign addition never turns on the overflow flag.
- In signed arithmetic, watch the overflow flag to detect errors. In unsigned arithmetic, the overflow flag tells you nothing interesting.

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Calculating Overflow flag - Method 1 (1)

- Overflow can only happen when adding two numbers of the same sign results in a different sign.
- to detect overflow
  - only the sign bits are considered
  - the other bits are ignored

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Calculating Overflow flag - Method 1 (2-1)

- with two operands and one result,
  three sign bits are considered
  $2^3 = 8$ possible combinations
- only two of 8 cases are considered as overflow

```
0 0 1  (+, +, -)
1 1 0  (-, -, +)
```

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Calculating Overflow flag - Method 1 (2-2)

- ADDITION SIGN BITS ($num1 + num2$)

```
   num1sign num2sign sumsign (num1 + num2)
  ----------------------------------------------------
        0 0 0
*OVER*  0 0 1 (adding two positives should be positive)
        0 1 0
        0 1 1
        1 0 0
        1 0 1
*OVER*  1 1 0 (adding two negatives should be negative)
        1 1 1
```

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Calculating Overflow flag - Method 1 (3-1)

- SUBTRACTION SIGN BITS ($num1 - num2$)

```
   num1sign num2sign subsign (num1 - num2)
  -----------------------------------------------------------------
          0 0 0
          0 0 1
          0 1 0
   *OVER* 0 1 1 (subtracting a negative is the same as adding a positive)
   *OVER* 1 0 0 (subtracting a positive is the same as adding a negative)
          1 0 1
          1 1 0
          1 1 1
```

- subtracting a positive / negative number is the same as adding a negative / positive

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Calculating Overflow flag - Method 1 (3-2)

- addition $(num1 + num2)$
```
0 0 1  (+, +, -)
1 1 0  (-, -, +)
```
- subtraction $(num1 - num2)$
```
0 1 1  (+, -, -)
1 0 0  (-, +, +)
```

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Calculating Overflow flag - Method 1 (4)

- A computer might contain a small logic gate array
  that sets the overflow flag to "1"
  iff any one of the above four OV conditions is met.
- in signed computations,
  adding two numbers of the same sign
  must produce a result of the same sign,
  otherwise overflow happened.

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Calculating Overflow flag - Method 2 (1)

- When adding two binary values,
  consider the binary carry coming
  into the leftmost place (into the sign bit)
  and the binary carry going out of that leftmost place.
  (Carry going out of the leftmost [sign] bit
  becomes the CARRY flag in the ALU.)
- Overflow in two's complement may occur,
  not when a bit is carried out of the left column,
  but when one is carried into it
  and no matching carry out occurs.
  That is, overflow happens when there is a carry
  into the sign bit but no carry out of the sign bit.

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Calculating Overflow flag - Method 2 (2)

The OVERFLOW flag is the XOR of the carry coming into the sign bit (if any) with the carry going out of the sign bit (if any). Overflow happens if the carry in does not equal the carry out.
Examples (2-bit signed 2's complement binary numbers):

```
 11
+01
===
 00

- carry in is 1
- carry out is 1
- 1 XOR 1 = NO OVERFLOW


 01
+01
===
 10

- carry in is 1
- carry out is 0
- 1 XOR 0 = OVERFLOW!
```

Calculating Overflow flag - Method 2 (3)

```
 11
+10
===
 01

- carry in is 0
- carry out is 1
- 0 XOR 1 = OVERFLOW!


 10
+01
===
 11

- carry in is 0
- carry out is 0
- 0 XOR 0 = NO OVERFLOW
```

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Calculating Overflow flag - Method 2 (4)

- Note that this XOR method only works with the binary carry that goes into the sign bit. If you are working with hexadecimal numbers, or decimal numbers, or octal numbers, you also have carry; but, the carry doesn't go into the sign bit and you can't XOR that non-binary carry with the outgoing carry.

`http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt`

Calculating Overflow flag - Method 2 (5)

- Hexadecimal addition example (showing that XOR doesn't work for hex carry):

```
 8Ah
+8Ah
====
 14h
```

- The hexadecimal carry of 1 resulting from A+A does not affect the

sign bit. If you do the math in binary, you'll see that there is no carry into the sign bit; but, there is carry out of the sign bit. Therefore, the above example sets OVERFLOW on. (The example adds two negative numbers and gets a positive number.)

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

TOC: accessing the condition codes

Set (1)

`set(e, z)`	D	(equal / zero)	D ← `ZF`
`set(ne, nz)`	D	(not equal/ not zero)	D ← `~ZF`
`set(s)`	D	(negative)	D ← `SF`
`set(ns)`	D	(non-negative)	D ← `~SF`
`set(g, le)`	D	(greater, signed >)	D ← `~(SF^OF)&~ZF`
`set(ge, nl)`	D	(greater or equal, signed >=)	D ← `~(SF^OF)`
`set(l, nge)`	D	(less, signed <)	D ← `SF^OF`
`set(le, ng)`	D	(less or equal, signed <=)	D ← `(SF^OF)` | `ZF`
`set(a, nbe)`	D	(above, usnigned >)	D ← `~CF&~ZF`
`set(ae, nb)`	D	(above or euqal, unsinged >=)	D ← `~CF`
`set(b, nae)`	D	(below, unsigned <)	D ← `CF`
`set(be, na)`	D	(below or equal, unsigned <=)	D ← `CF&~ZF`

Set (2)

`set(e, z)`	D	(equal / zero)	D ← ZF
`set(s)`	D	(negative)	D ← SF
`set(g, le)`	D	(greater, signed >)	D ← ~(SF^OF)&~ZF
`set(l, ge)`	D	(less, signed <)	D ← SF^OF
`set(a, nbe)`	D	(above, usnigned >)	D ← ~CF&~ZF
`set(b, nae)`	D	(below, unsigned <)	D ← CF

`set(ne, nz)`	D	(not equal/ not zero)	D ← ~ZF
`set(ns)`	D	(non-negative)	D ← ~SF
`set(ge, nl)`	D	(greater or equal, signed >=)	D ← ~(SF^OF)
`set(le, ng)`	D	(less or equal, signed <=)	D ← (SF^OF) | ZF
`set(ae, nb)`	D	(above or euqal, unsinged >=)	D ← ~CF
`set(be, na)`	D	(below or equal, unsigned <=)	D ← CF&~ZF

Flag registers (1) - Z, O, S, P

`E, Z`	Equal, Zero	`ZF == 1`
`NE, NZ`	Not Equal, Not Zero	`ZF == 0`
`O`	Overflow	`OF == 1`
`NO`	No Overflow	`OF == 0`
`S`	Signed	`SF == 1`
`NS`	Not Signed	`SF == 0`
`P`	Parity	`PF == 1`
`NP`	No Parity	`PF == 0`

https://riptutorial.com/x86/example/6976/flags-register

Flag registers (2) - unsigned arithmetic

`C, B` `NAE`	Carry, Below, Not Above or Equal	`CF == 1`
`NC, NB` `AE`	No Carry, Not Below, Above or Equal	`CF == 0`
`A, NBE`	Above, Not Below or Equal	`CF==0 and ZF==0`
`NA, BE`	Not Above, Below or Equal	`CF==1 or ZF==1`

`https://riptutorial.com/x86/example/6976/flags-register`

Flag registers (3) - signed arithmetic

GE, NL	Greater or Equal, Not Less	SF==OF
NGE, L	Not Greater or Equal, Less	SF!=OF
G, NLE	Greater, Not Less or Equal	ZF==0 and SF==OF
NG, LE	Not Greater, Less or Equal	ZF==1 or SF!=OF

https://riptutorial.com/x86/example/6976/flags-register

Flag registers (4)

- The condition codes are grouped into three blocks :

`Z`, `O`, `S`, `P`	Zero Overflow Sign Parity
unsigned arithmetic	Above Below
signed arithmetic	Greater Less

- JB would be "Jump if Below" (unsigned)
- JL would be "Jump if Less" (signed)

`https://riptutorial.com/x86/example/6976/flags-register`

Flag registers (3)

- In 16 bits, subtracting 1 from 0

from	to	
0	65,535	unsigned arithmetic
0	-1	signed arithmetic
0x0000	0xFFFF	bit representation

- It's only by interpreting the condition codes that the meaning is clear.
- 1 is subtracted from 0x8000:

from	to	
32,768	32,767	unsigned arithmetic
-32,768	32,767	signed arithmetic
0x8000	0x7FFF	bit representation

(0111 1111 1111 1111 + 1 = 1000 0000 0000 0000)

https://riptutorial.com/x86/example/6976/flags-register

Set (3)

- accessing the condition codes
  - to read the condition codes directly
  - to set an integer register
  - to perform a conditional branch

  based on some combination of condition codes

Set (4)

- the `set` instructions set a single *byte* to 0 or 1
  depending on some combination of the condition codes
- the destination operand `D` is
  - either one of the eight single *byte* register elements
  - or a memory location where the single *byte* is to be stored
- to generate a 32-bit result,
  the high-order 24-bits must be *cleared*

Set (5)

a typical assembly for a c predicate

```
; a is in %edx
; b is in %eax

cmpl   %eax, %edx       ; compare a and b  ; (a - b)
setl   %al              ; set low order byte of %eax to 0 or 1
movzbl %al, %eax        ; set remaining bytes of %eax to 0
```

- `movzbl` instruction is used to clear the high-order three bytes
- | `set(l, ge)` | `D` | (less, signed <) | `D` ← `SF^OF` |

movz instruciton (1)

- Purpose: To convert an unsigned integer to a wider unsigned integer
- `opcode src.rx, dst.wy`
- `dst <- zero extended src;`

- `MOVZBW` (Move Zero-extended Byte to Word) 8-bit zero `BW`
- `MOVZBL` (Move Zero-extended Byte to Long) 24-bit zero `BL`
- `MOVZWL` (Move Zero-extended Word to Long) 16-bit zero `WL`

movz instruciton (2)

- `MOVZ BW` (Move Zero-extended Byte to Word) 8-bit zero
  - the low 8 bits of the destination are replaced by the source operand
  - the top 8 bits are set to 0.
- `MOVZ BL` (Move Zero-extended Byte to Long) 24-bit zero
  - the low 8 bits of the destination are replaced by the source operand.
  - the top 24 bits are set to 0.
- `MOVZ WL` (Move Zero-extended Word to Long) 16-bit zero
  - the low 16 bits of the destination are replaced by the source operand.
  - the top 16 bits are set to 0.
- The source operand is unaffected.

register operand types (1)

byte 3	byte 2	byte 1	byte 0
		%ah	%al
		%ax_1	%ax_0
%eax_3	%eax_2	%eax_1	%eax_0
		%ch	%cl
		%cx_1	%cx_0
%ecx_3	%ecx_2	%ecx_1	%ecx_0
		%dh	%dl
		%dx_1	%dx_0
%edx_3	%edx_2	%edx_1	%edx_0
		%bh	%bl
		%bx_1	%bx_0
%ebx_3	%ebx_2	%ebx_1	%ebx_0

register operand types (2)

byte 3	byte 2	byte 1	byte 0
		%si_1	%si_0
%esi_3	%esi_2	%esi_1	%esi_0
		%di_1	%di_0
%edi_3	%edi_2	%edi_1	%edi_0
		%sp_1	%sp_0
%esp_3	%esp_2	%esp_1	%esp_0
		%bp_1	%bp_0
%ebp_3	%ebp_2	%ebp_1	%ebp_0

register operand types (3)

byte 3	byte 2	byte 1	byte 0
		%ah	%al
		%ch	%cl
		%dh	%dl
		%bh	%bl
		%ax_1	%ax_0
		%cx_1	%cx_0
		%dx_1	%dx_0
		%bx_1	%bx_0
		%si_1	%si_0
		%di_1	%di_0
		%sp_1	%sp_0
		%bp_1	%bp_0

register operand types (4)

byte 3	byte 2	byte 1	byte 0
%eax_3	%eax_2	%eax_1	%eax_0
%ecx_3	%ecx_2	%ecx_1	%ecx_0
%edx_3	%edx_2	%edx_1	%edx_0
%ebx_3	%ebx_2	%ebx_1	%ebx_0
%esi_3	%esi_2	%esi_1	%esi_0
%edi_3	%edi_2	%edi_1	%edi_0
%esp_3	%esp_2	%esp_1	%esp_0
%ebp_3	%ebp_2	%ebp_1	%ebp_0

Set (6)

- for some of the underlying machine instructions, there are multiple possible names (synonyms),
  - `setg` (set greater)
  - `setnle` (set not less or equal)
- compilers and disassemblers make arbitrary choices of which names to use

Set (7)

- although all arithmetic operations set the condition codes, the descriptions of the different set commands apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation `t = a - b`
- for example, consider the `sete`, or "Set when equal" instruction
- when `a = b`, we will have `t = 0`, and hence the zero flag indicates equality

Set (8)

- Similarly, consider testing a signed comparison with the `setl` or "Set when less"
- when `a` and `b` are in two's complement form, then for `a < b` we will have `a - b < 0` if the true difference were computed
- when there is no overflow, this would be indicated by having the sign flag set

Set (9)

- when there is positive overflow, because `a - b` is a large positive number, however, we will have `t < 0`
- when there is negative overflow, because `a - b` is a small negative number, we will have `t > 0`
- in either case, the sign flag will indicate the opposite of the sign of the true difference

Set (10)

- in either case, the sign flag will indicate the opposite of the sign of the true difference
- hence, the Exclusive-Or of the overflow and sign bits provides a test for whether `a < b`
- the other signed comparison tests are based on other combinations of `SF ^ OF` and `ZF`

Set (11)

- for the testing of unsigned comparisons, the carry flag will be set by the `cmpl` instruction when the integer difference `a - b` of the unsigned arguments `a` and `b` would be negative, that is when `(unsinged) a < (unsigned) b`
- thus, these tests use combinations of the carry and zero flags