Carry and Borrow

Young W. Lim

2023-07-01 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 and Overflow

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

```
            .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

Borrow and Subtraction

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-i

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

- 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` ← `Cout`)
    - then `a + ~b + Cout` becomes `a + ~b + !C`
  - in the normal carry sytem
    - `C = Cout` : normal carry
    - `Cin = C` : simple feedback (`Cin` ← `Cout`)
    - then `a + ~b + Cout` becomes `a + ~b + C`

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

- 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-i

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 ← 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 ← 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 ≥ (b+C)	
ARM	**normal carry**	
new C = 0	when a < (b+!C)	borrow
new C = 1	when a ≥ (b+!C)	

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

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 ≥ (b+C)	
ARM	**normal carry**	
new C = 0	when a < (b+!C)	borrow
new C = 1	when a ≥ (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 $\leftarrow$ Rn + Op2
Subtract	`SUB` Rd, Rn, Op2	Rd $\leftarrow$ Rn – Op2
Add with Carry	`ADC` Rd, Rn, Op2	Rd $\leftarrow$ Rn + Op2 + C
Subtract with Carry	`SBC` Rd, Rn, Op2	Rd $\leftarrow$ Rn – Op2 – !C
Reverse Subtract	`RSB` Rd, Rn, Op2	Rd $\leftarrow$ Op2 – Rn
Reverse Subtract wiht Carry	`RSC` Rd, Rn, 0	Rd $\leftarrow$ 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

- 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 ≥ (b + !C)
new C = 0	a ≥ (b + old C)	a < (b + old !C)
new C = 1	a < (b + old C)	a ≥ (b + 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

- 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 and SBB instructions

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 and DEC instructions

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