Conditions

Young W. Lim

2022-06-21 Tue

Outline

1. Based on
2. Condition Codes
3. Accessing the Conditon Codes

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

TOC: Conditional codes

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_x

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_x

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_x

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_x

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_x

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_x

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_

Borrow and subtraction (1)

- While the carry flag is
  well-defined for addition,

- there are *two ways* in common use
  to use the carry flag
  for subtraction operations.
  - subtract with borrow
    uses the carry bit as a borrow flag
  - subtract with carry
    uses the identity directly
    `-x = (not x)+1`
    (i.e. without storing the carry bit inverted)

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

Borrow and subtraction (2)

- subtract with borrow
  uses the carry bit as a borrow flag

- when computing `a - b`
  - if `a < b`, the carry bit is *set*
    and a borrow must be performed.
  - If `a >= b`, the bit is *cleared*.

- a SBB (subtract with borrow) instruction
  will compute `a-b-C = a-(b+C)`

- a SUB (subtract without borrow)
  acts `a-b-0 = a - b`
  as if the borrow bit were *clear*.

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

Borrow and subtraction (3)

- subtract with carry uses the identity directly
  `-x = (not x)+1`
  (i.e. without storing the carry bit *inverted*)
- computes `a - b` as `a+(not b)+1`
  the carry bit is set according to this addition
- subtract with carry computes `a+not(b)+C`
- subtract without carry acts as if the carry bit were *set*
- The result is that the carry bit is
  set if `a >= b`,
  clear if `a < b`.

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

Borrow and subtraction (4)

- 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`

Borrow and subtraction (5)

- However, there are exceptions in both directions;
  the VAX, NS320xx, and Atmel AVR architectures
  use the borrow bit convention,
  but *call* their `a-b-C` operation subtract with carry
  (`SBWC`, `SUBC` and `SBC`).
- The PA-RISC and PICmicro architectures
  use the carry bit convention, but *call* their `a+not(b)+C` operation
  subtract with borrow (`SUBB` and `SUBWFB`).

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

Borrow and subtraction (6)

- The ST6 8-bit microcontrollers are perhaps
  the most confusing of all.
  Although they do not have any sort of subtract with carry
  instruction, they do have a carry bit which is set
  by a subtract instruction, and the convention
  depends on the processor model.
- The ST60 processor uses the "carry" convention,
  while the ST62 and ST63 processors use the "borrow" convention.

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

Borrow and subtraction (7)

Summary of different uses of carry flag in subtraction

Carry or borrow bit	Subtract without carry/borrow	Subtract with borrow	Subtract with carry
C = 0	a - b = a+not(b)+1	a - b - 0 = a+not(b)+1	a - b - 1 = a+not(b)+ 0
C = 1	a - b = a+not(b)+1	a - b - 1 = a+not(b)+0	a - b - 0 = a+not(b)+1

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, 0000000lh: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.
The lower 32 bits are subtracted first, setting the Carry flag CF.
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`

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

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