Concise Notes (See bottom for Verbose Notes)

New x86 Instructions

cmpl : signed comparison

compl a, b = "Is b >= a?"

→ Let's say you perform cmpl on arbitrary registers R1 and R2.

It performs something like subl R1, R2 but only for comparison codes

Only changes the flags, does not touch anything else

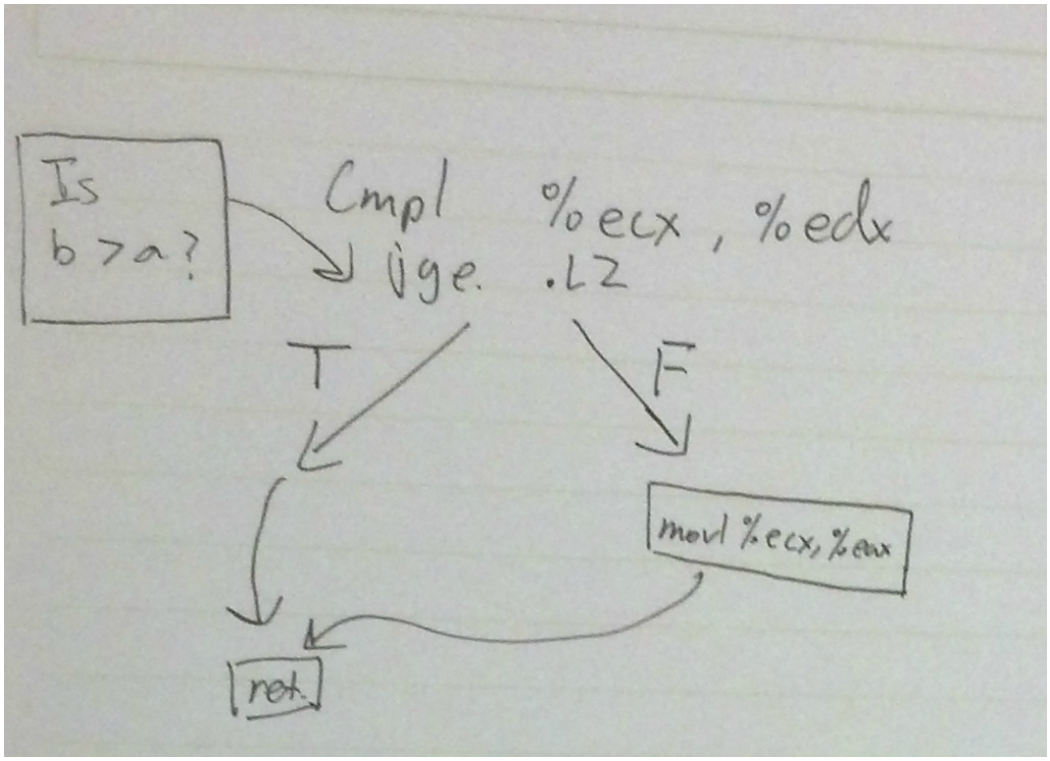
→ Source address is always on the left of the operand for non-transitive operations (IE subl a, b is b - a)

→ like subl R1 R2, but only C.C.

Example:

a stored in %ecx

b stored in %edx ::



see below for more on jge

nop : no operation

Does absolutely nothing.

Can actually make code faster when a > b

jge : Jump Greater than Or Equal To

jge .L2: Jump to label .L2 if conditions are met based on previous operation..

JGE asks !(SF ^OF) -- makes these tests on flags (see below)

Basically a Signed comparison

→ correct for signed integers.

→ compl $0, $0 → sets the ZF flag to true; all others false.

→ compl $2, $-1 → sets !ZF, SF, !CF, !OF, results in $-3 subtraction.

Therefore JGE holds.

JAE asks: !CF

An unsigned comparison; otherwise identical to JGE

Asks if

je : Jump if Equal

je .L2 == jump to label if ZF == 1.

if the last 2 items compared were equal, jump

jz: jump zero.

identical to je

asks if ZF == 1 (using flags)

jne : jump not equal

asks: !ZF

The last arithmetic function produced a non-zero result

testl: testl R1, R2

like andl R1, R1, but ONLY for C.C.

→ very common to see testl %eax, %eax

→ equivalent to saying "%eax & %eax" ;; followed by je: "%eax == 0"

rep : Always seen before ret instruction

There was a problem with AMD processors that made just using the ret function very slow. So, compilers compile "return" into rep then ret to take this into account.

Does nothing. Literally means "upcoming string operation"

popl : pops from the stack. Opposite of pushl below.

pushl: pushing arguments onto the stack. Preparing for function call.

If used at the beginning of a function, can throw off addresses of variables relative to stack pointer (ie 4(%esp) becomes 8(%esp)).

imull: multiply instruction.

Conditional branches in x86:

- Work on a HIDDEN REGISTER: %eflags
  - Condition Codes
- Arithmetic expressions change condition codes
  - Only exception is LEA
  - LEA does not change any flags. "Load effective address"

Flags

On the hidden register, there are flags (set as side-effect of every arithmetic instr.):

- ZF: Last bit was zero.
- SF: Last number was negative. (31st bit was 1)
- CF: Unsigned overflow.
  - Sets flag for unsigned integer wrap-around
- OF: Signed overflow → answer has a different sign.
  - Sets flag for signed integer wrap-around

(if a negative minus a negative yields a positive (sign wrap-around))
Flags are “way way more useful in the conditional jump than anywhere else.”

Strategy for Assessing Assembly Instructions:

- Use breakpoints.
- Working backwards and forwards in code, from start and beginning of each function
- Write in terms of C code as much as possible.
- Use what you already know about x86 instructions

Comparisons:

- Signed versus unsigned matters.
    - 0x 8000 0000 < 1 → true if signed; false if unsigned.
- In x86, there are often variations, like jge vs jae

Drawbacks of Logical Branches and Jumps:

- Modern processors are so fast by performing operations before they are actually reached by the program
    - For logical branches, the processor has to try to guess which way the program will go and work along that branch.
    - Likely cases will often be lumped together in memory.
    - Unlikely cases (like errors) will be put somewhere else.
    - This can actually enhance performance hugely.
- Caching
    - If a branch is very far away from in code, then processor might have to go to the cache to bring in those instructions, resulting in performance hits
    - A good compiler will place likely branches next to each other to avoid going to the cache, which can increase performance by sometimes 50%.
    - You can use macros like ‘likely’ and ‘unlikely’ to optimize branch placement
    - You can also use logging to log statistically the most and least likely branches, and compile more efficiently based off of that information.

Loops!

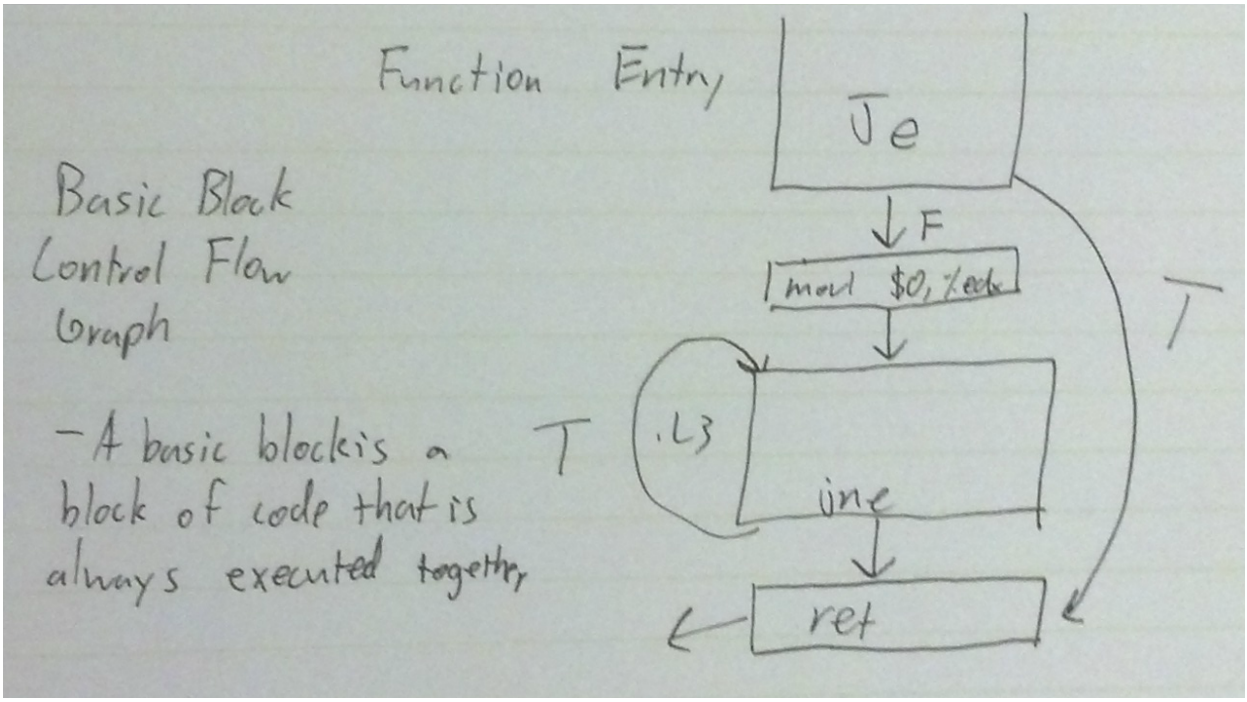
Loops always have some properties:

- One essential item: backwards jump.
- Entry point like .L3 at beginning of loop block.
- Comparison and jump to that entry point.
- Complex Conditions
    - A loop that exits when counter = some constant OR when another condition is met.

Picking out a loop in assembly instructions.

- Look for the counter (something modified each loop, might be initialized to a constant like $0)
- Comparison right before the jump at the end of the loop is likely to be the loop conditional.

A Basic Block Control Flow Graph of a loop, following True and False results of JE and JNE



Calling Conventions:

- Some registers are saved by the caller; some by the callee.
- In other words, there are some registers that any function can use ("scratch" registers = "caller saved"), and there are "precious" registers that a function must restore ("restore on exit," callee-saved).

- Caller-saved:
  - %eax
  - %ecx
  - %edx
  - These have been used in most of the programs so far, because then the compiler can make the functions faster to run because it does not need to save or restore any value
- Callee-saved:
  - %esp
  - %ebp
  - %ebx
  - %esi
  - %edi

Calling convention influences push and pop.

- Return to hello kitty - How do problems like this happen?
  - Here we have a program, smash01.c
  - What does hti program do?
- Gets just reads lines, and for every line jst do ++lines and then prints the numbers of lines in standard input