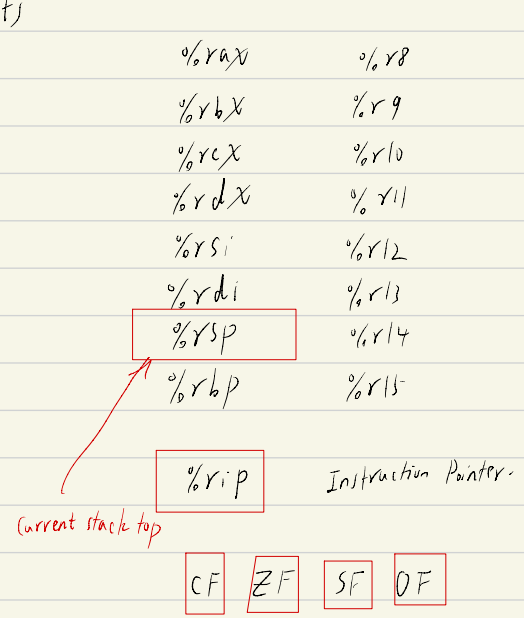
Processor State (x86-64, Partial)

(I) Information about currently executing program
- (i) Temporary data (%rax, ...)
- (ii) Location runtime stack (%rsp)
- (iii) Location of current code control point (%rip, ...)
- (iv) Status of recent tests (CF, ZF, SF, OF)



Condition codes (Implicit setting)

(I) single bit registers

(i) CF: Carry Flag (for unsigned) SF: Sign Flag (for signed)

(ii) ZF: zero Flag OF: Overflow Flag (for signed)

↑ the value you have computed is zero.

(II) Implicit set (think of it as side effect) by arithmetic operations.

Eg.: addq Src, Dest ⟷ t = a+b.

CF set if carry out from most significant bit (unsigned overflow)

ZF set if t == 0

SF set if t < 0 (as signed)

OF set if two's-complement (signed) overflow

(a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t >= 0)

(II) Explicit codes (Explicit setting: compare)

(i) Explicit setting by compare instruction.

(ii) cmpq src2, src1.

(iii) cmpq b, a like computing a-b without setting destination.

quadw. 64位

① CF set: if carry out from most significant bit (used for unsigned comparisons)

② ZF set: if a == b

③ SF set: if (a-b) < 0 (as signed)

④ OF set: if two's-complement (signed) overflow

(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

(III) Explicit Setting by Test instruction.

(I) testq Src2, Src1

testq b, a like computing a & b without setting destination.

(ii) Sets condition codes based on value of Src1 & Src2

(iii) useful to have one of the operands be a mask.

ZF set when a & b == 0

SF set when a & b < 0.

(2) setX Instructions

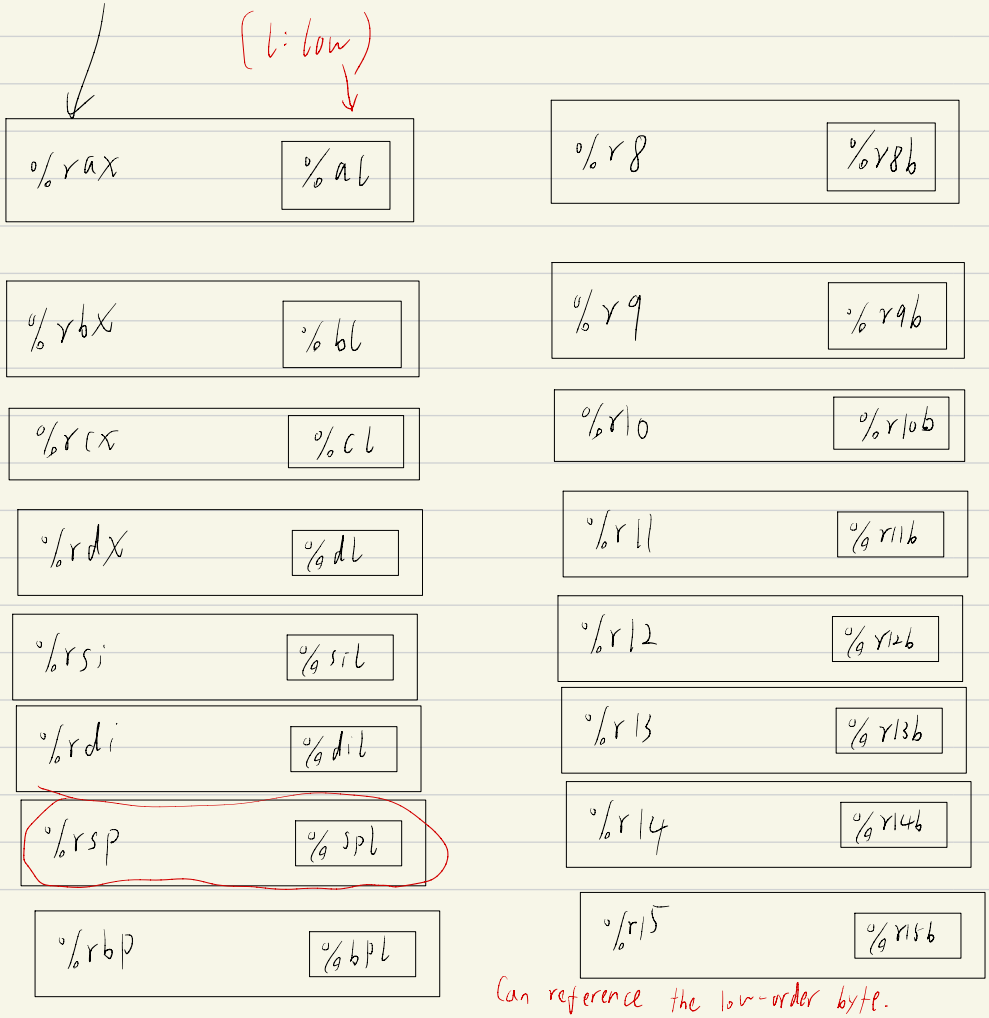
(i) set low-order byte of destination to 0 or 1 based on combinations of condition codes.

(ii) Does not alter remaining 7 bytes

set_instructions.txt.

%eax: lower 32 bits of %rax

(L: low)



Can reference the low-order byte.

(iii) One of addressable byte registers:

- Does not alter remaining bytes.
- Typically use movzbl to finish job
  - 32 bit instructions also set set upper 32 bits to 0.

(see file gt.c for more details)

Jumping

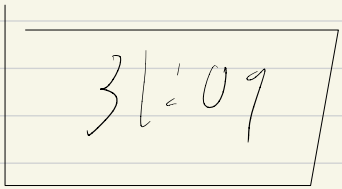
(i) jX Instructions

(ii) Jump to different part of code depending on condition codes.

Expressing with Goto code:

(i) C allows goto statement

(iii) Jump to position designated by label.



Using Conditional Moves

(I) Conditional Move Instructions:

(i) Instruction supports:
"if [Test] Dest ← Src"

(ii) Supported in post-1995 x86 processors.

(iii) GCC tries to use them
- But only when known to be safe

(II) Why?

(i) Branches are very disruptive to instruction flow through pipelines.

(ii) Conditional moves do not require control transfer

C code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto version:

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Bad Cases for conditional Move:

Expensive computations:

val = Test(x) ? Hard1(x) : Hard2(x);

(i) Both values get computed

(ii) Only makes sense when computations are very simple.

Risky computations.

val = p ? *p : 0;

Both values get computed.

may have undesirable effects

Computation with side effects:

val = x > 0 ? x *= 7; x += 5;

Both values get computed.

Must be side-effect free.

See

- pcount_do.c
- pcount_goto.c
- pcount_do.s
- pcount_goto.s

General "while" Translation #

(1) "Jump" to "Middle" Translation

(1) used with -Og

(-Og [optimize debug] basically translates our do_while loop into a goto statement.)

```
while (Test)
  Body
```

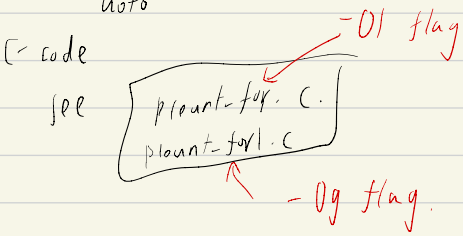
⇒

```
    goto test;
loop:
    Body
test:
    if (Test)
        goto loop;
done:
```

"For" Loop Do-while conversion

Goto Version

C code



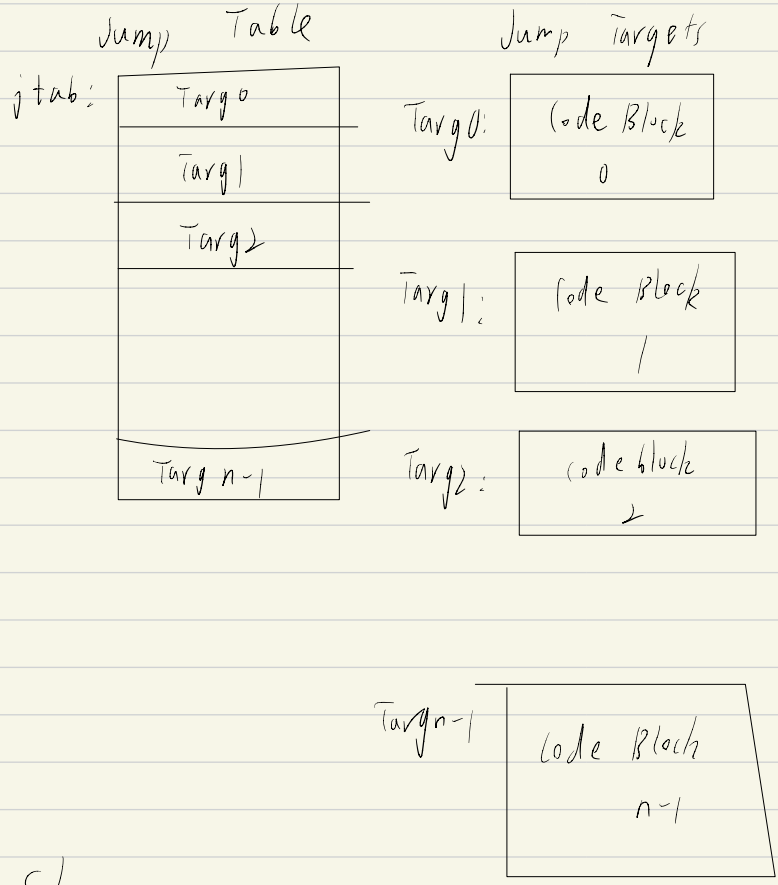
see

We could see that in the code pcount_for1.c, the code is more optimized.

Jump Table structure:

Switch Form

```
switch (x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ...
    case val_n-1:
        Block n-1
}
```



Translation (Extended C)

```
goto *jTab[x];
```