Chapter 3.6: Control

- execution order of a set of machine-code instructions can be altered with a *jump* instruction

3.6.1: Condition Codes

- CPU maintains a set of single-bit *condition code* registeers describing attributes of the most recent arithmetic or logical operation

CF: Carry Flag

- the most recent operation generated a carry out of the most significant bit
- used to detect overflow for unsigned operations

ZF: Zero Flag

- the most recent operation yielded zero

SF: Sign Flag

- the most recent operation yielded a negative value

OF: Overflow Flag

- the most rcent operation caused a two's-complement overflow — either negative or positive

- the `leaq` instruction does not alter any condition codes, since it is for address computations
- all unary, binary operators cause the condition codes to be set
- for logical operations, such as XOR
  - CF and OF are set to zero
- for shift operations
  - CF is set to last bit shifted out, OF is set to 0
- **there are also instruction classes that set condition codes with altering other registers**
  - CMP instructions set condition codes according to differences of their two operands
  - behave in the same way as the SUB instructions, except they don't update destination, they only set condition codes
  - TEST instructions behave in the same way as the AND instructions, except they don't update destination, only set condition codes

CMP S1, S2

S2 — S1 (compare)

- set zero flag if two operands are equal

Instruction	Description
cmpb	Compare byte
cmpw	Compare word
cmpl	Compare double word
cmpq	Compare quad word

TEST S1, S2

S1 & S2 (test)

Instruction	Description
testb	Test byte
testw	Test word
testl	Test double word
testq	Test quad word

Figure 3.13

3.6.2: Accessing the Condition Codes

There are three common ways of using condition codes:

1. Set a single byte to 0 or 1 depending on some combination of the condition codes.

Instruction		Synonym	Effect	Set condition
`sete`	D	`setz`	$D \leftarrow$ ZF	Equal / zero
`setne`	D	`setnz`	$D \leftarrow$ ~ZF	Not equal / not zero
`sets`	D		$D \leftarrow$ SF	Negative
`setns`	D		$D \leftarrow$ ~SF	Nonnegative
`setg`	D	`setnle`	$D \leftarrow$ ~(SF ^ OF) & ~ZF	Greater (signed >)
`setge`	D	`setnl`	$D \leftarrow$ ~(SF ^ OF)	Greater or equal (signed >=)
`setl`	D	`setnge`	$D \leftarrow$ SF ^ OF	Less (signed <)
`setle`	D	`setng`	$D \leftarrow$ (SF ^ OF) | ZF	Less or equal (signed <=)
`seta`	D	`setnbe`	$D \leftarrow$ ~CF & ~ZF	Above (unsigned >)
`setae`	D	`setnb`	$D \leftarrow$ ~CF	Above or equal (unsigned >=)
`setb`	D	`setnae`	$D \leftarrow$ CF	Below (unsigned <)
`setbe`	D	`setna`	$D \leftarrow$ CF | ZF	Below or equal (unsigned <=)

3.6.3: Jump Instructions

Instruction	Synonym	Jump condition	Description
jmp *Label*		1	Direct jump
jmp **Operand*		1	Indirect jump
je *Label*	jz	ZF	Equal / zero
jne *Label*	jnz	~ZF	Not equal / not zero
js *Label*		SF	Negative
jns *Label*		~SF	Nonnegative
jg *Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge *Label*	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl *Label*	jnge	SF ^ OF	Less (signed <)
jle *Label*	jng	(SF ^ OF) | ZF	Less or equal (signed <=)
ja *Label*	jnbe	~CF & ~ZF	Above (unsigned >)
jae *Label*	jnb	~CF	Above or equal (unsigned >=)
jb *Label*	jnae	CF	Below (unsigned <)
jbe *Label*	jna	CF | ZF	Below or equal (unsigned <=)

- jump targets can be direct or indirect
  - **direct**: jump target is encoded as part of the instruction
  - **indirect**: jump target is read from a register or memory location
    - use value in register: `jmp *%rax`
    - reads from memory in address held by register: `jmp *(%rax)`

3.6.4: Jump Instruction Encodings

- most commonly used jump instruction encodings are PC **relative**
  - encodes difference between address of target instruction and address of instruction immediately following the jump

```
; Assembly code
    movq    %rdi, %rax
    jmp     .L2
.L3:
    sarq    %rax
.L2:
    testq   %rax, %rax
    jg      .L3
    rep; ret
```

```
1 ; Disassembled version
2 0:   48 89 f8    mov     %rdi, %rax
3 3:   eb 03       jmp     8 <loop+0x8>
4 5:   48 d1 f8    sar     %rax
5 8:   48 45 c0    test    %rax, %rax
6 b:   7f f8       jg      5 <loop+0x5>
7 d:   f3 c3       repz retq
```

- Notice that in the disassembled version, line 3 is encoded `eb 03`
    - Which says, take the next line's address (5) and add (3) to it
    - AKA, go to line 8

3.6.5: Implementing Conditional Branches with Conditional Control

- **the most general way to translate conditional expressions and statements from C into machine code**
    - use combinations of conditional and unconditional jumps

```
GENERAL FORM

if (test-expr)
    then-statement
else
    else-statement

------------------

    t = test-expr;
    if (!t)
        goto false;
    then-statement
    goto done;
false:
    else-statement
done:
```

```
; Example if-else assembly code
; Takes absolute alue of two numbers
; x in %rdi, y in %rsi

absdiff_se:
    cmp     %rsi, %rdi          ; Compare x:y
    jge     .L2                 ; If >= goto x_ge_y
    mov     %rsi, %rax
    sub     %rdi, %rax          ; result = y - x
    ret                         ; Return
.L2:
    mov     %rdi, %rax
    sub     %rsi, %rax          ; result = x - y
    ret                         ; Return
```

3.6.6: Implementing Conditional Branches with Conditional Moves

- using conditional transfer of *control* is conventional but can be very inefficient on modern processors
- alternate strategy is to use conditional transfers of *data*
    - compute both outcomes of a conditional operation
    - select one based on whether or not condition holds

Pros

- this method is more efficient because it allows the processor to constantly keep the *pipeline* full
    - pipelining allows the processor to predict the code flow, so that it knows what operation is next
    - conditional moves avoids excessive branching ➤ more accurate pipeline prediction

Cons

- can lead to a lot of unnecessary computation if branches are very long
- may lead to invalid operations, such as null pointer deferencing

```
; Example if-else assembly code
; Takes absolute alue of two numbers
; x in %rdi, y in %rsi

absdiff:
    mov     %rsi, %rax
    sub     %rdi, %rax      ; rval = y - x
    mov     %rdi, %rdx
    sub     %rsi, %rdx      ; eval = x - y
    cmp     %rsi, %rdi      ; Compare x:y
    cmovge  %rdx, %rax      ; If >=, rval = eval
    ret                     ; Return tval
```

Conditional Move Instructions

Instruction		Synonym	Move condition	Description
cmove	S, R	cmovz	ZF	Equal / zero
cmovne	S, R	cmovnz	~ZF	Not equal / not zero
cmovs	S, R		SF	Negative
cmovns	S, R		~SF	Nonnegative
cmovg	S, R	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	S, R	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl	S, R	cmovnge	SF ^ OF	Less (signed <)
cmovle	S, R	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
cmova	S, R	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae	S, R	cmovnb	~CF	Above or equal (Unsigned >=)
cmovb	S, R	cmovnae	CF	Below (unsigned <)
cmovbe	S, R	cmovna	CF | ZF	Below or equal (unsigned <=)

- source and destination values can be 16, 32, or 64 bits long
    - **single-byte conditional moves are not supported**
- assembler can infer operand length of a conditional move instruction based on destination register

3.6.7: Loops

Do-While Loops

```c
long fact_do(long n) {
    long result = 1;
    do {
        result *= n;
        n = n - 1;
    } while (n > 1);
    return result;
}
```

```
; n in %rdi
fact_do:
    mov     $1, %eax            ; Set result = 1
.L2:                            ; loop
    imul    %rdi, %rax          ; Compute result *= n
    sub     $1, %rdi            ; Decrement n
    cmp     $1, %rdi            ; Compare n:1
```

```
    jg      .L2             ; If >, goto loop
```

While Loops

```c
long fact_while(long n) {
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n - 1;
    }
    return result;
}
```

```
; n in %rdi
fact_while:
    mov     $1, %eax        ; Set result = 1
    jmp     .L5             ; Goto test
.L6:
    imul    %rdi, %rax      ; Compute result *= n
    sub     $1, %rdi        ; Decrement n
.L5:
    cmp     $1, %rdi        ; Compare n:1
    jg      .L6             ; If >, goto loop
    rep; ret                ; Return
```

For Loops

- pretty much the same as a while loop, just looks prettier in C

3.6.8: Switch Statements

```c
void switch_eg(long x, long n, long *dest) {
    long val = x;

    switch(n) {
    case 100:
        val *= 13;
        break;

    case 102:
        val += 10;
        /* Fall through */

    case 103:
        val += 11;
        break;

    case 104:
    case 106:
        val *= val;
```

```c
        break;

    default:
        val = 0;
    }
    *dest = val;
}
```

```asm
switch_eg:
    sub     $100, %rsi              ; Compute index = n - 100
    cmp     $6, %rsi                ; Compare index:6
    ja      .L8                     ; If >, goto loc_def
    jmp     *.L4(,%rsi,8)           ; Goto *jg[index]
.L3:                            ; case 100
    lea     (%rdi,%rdi,2), %rax     ; 3*x
    lea     (%rdi,%rax,4)           ; val = 13*x
    jmp     .L2                     ; Goto done
.L5:                            ; case 102
    add     $10, %rdi               ; x = x + 10
.L6:                            ; case 103
    addq    $11, %rdi               ; val = x + 11
    jmp     .L2                     ; Goto done
.L7:                            ; case 104
    imul    %rdi, %rdi              ; val = x * x
    jmp     .L2                     ; Goto done
.L8:                            ; case default
    mov     $0, %edi                ; val = 0
.L2:                            ; done:
    mov     %rdi, (%rdx)            ; *dest = val
    ret                             ; Return
```

```asm
; jump table
.L4:
    .quad   .L3     ; case 100
    .quad   .L8     ; case default
    .quad   .L5     ; case 102
    .quad   .L6     ; case 103
    .quad   .L7     ; case 104
    .quad   .L8     ; case 105
    .quad   .L7     ; case 106
```