Control flow

Condition codes

Conditional and unconditional jumps

Loops

Switch statements

Conditionals and Control Flow

- **A conditional branch is sufficient to implement most control flow constructs offered in higher level languages**
  - `if (condition) then {...} else {…}`
  - `while (condition) {…}`
  - `do {…} while (condition)`
  - `for (initialization; condition; iterative) {...}`
- **Unconditional branches implement related control flow constructs**
  - `break, continue`
- **x86 calls branches "jumps"** (conditional or unconditional)

Jumping

jX Instructions

Jump to different part of code depending on condition codes

Takes address as argument

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Processor State (IA32, Partial)

Info about current executing program

Temporary data
(**%eax, …**)

Location of stack
(**%ebp,%esp**)

Location of next instruction
(**%eip**)

Status of recent tests
(**CF,ZF,SF,OF**)

`%eax`
`%ecx`
`%edx`
`%ebx`
`%esi`
`%edi`

General purpose registers

`%esp` Current stack top

`%ebp` Current stack frame

`%eip` Instruction pointer

`CF` `ZF` `SF` `OF` Condition codes

Condition Codes (flags)

Single-bit registers

`CF` Carry Flag (for unsigned) `SF` Sign Flag (for signed)
`ZF` Zero Flag `OF` Overflow Flag (for signed)

Arithmetic instructions set flags implicitly

Example: **`addl/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

`sign(a) == sign(b) && sign(b) != sign(t)`

LEA does not set flags.

Full documentation (IA32): http://www.jegerlehner.ch/intel/IntelCodeTable.pdf

cmp: compare values with subtraction

Single-bit registers

`CF` Carry Flag (for unsigned) `SF` Sign Flag (for signed)
`ZF` Zero Flag `OF` Overflow Flag (for signed)

Compare instructions set flags

`cmpl/cmpq` *Src2,Src1*

`cmpl b,a` computes **`a-b`**, discards result

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

ZF set if **`a == b`**

SF set if **`(a-b) < 0`** (as signed)

OF set if two's complement (signed) overflow

`sign(a) == sign(b) && sign(b) != sign(t)`

test: compare values with &

Single-bit registers

`CF` Carry Flag (for unsigned) `SF` Sign Flag (for signed)
`ZF` Zero Flag `OF` Overflow Flag (for signed)

Test instructions set flags

`testl`/**`testq`** *Src2,Src1*

`testl b,a` computes **`a & b`**, discards result

Sets condition codes based on value of *Src1* & *Src2*

Useful to have one of the operands be a **mask**

ZF set if **`a&b == 0`**

SF set if **`a&b < 0`**

`testl %eax, %eax`

Sets SF and ZF, check if **`%eax`** is +,0,-

Inspecting Condition Codes

SetX Instructions

Set a single byte to 0 or 1 based on combinations of condition codes

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Inspecting Condition Codes

SetX Instructions:

Set single byte to 0 or 1 based on combination of condition codes

One of 8 addressable byte registers

Does not alter remaining 3 bytes

Typically use **`movzbl`** to finish job

zero-extend from byte (8 bits) to longword (32 bits)

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

```
int gt (int x, int y) {
  return x > y;
}
```

Body: y at 12(%ebp), x at 8(%ebp)

```
movl 12(%ebp),%eax
cmpl %eax,8(%ebp)
setg %al
movzbl %al,%eax
```

9

Jump!

jX Instructions

Jump to different part of code depending on condition codes

Takes address as argument

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

10

Conditional Branch Example

```
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp               } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx      } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave                      } Finish
    ret
.L7:
    subl    %edx, %eax         } Body2
    jmp     .L8
```

11

Conditional Branch Example

```
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
int goto_ad(int x, int y) {
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

- **C allows "goto" as means of transferring control**
  - Closer to machine-level programming style
- **Bad style**

12

http://xkcd.com/292/

13

Conditional Branch Example

```
int goto_ad(int x, int y) {
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
int x      %edx
int y      %eax
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

14

General Conditional Expression Translation

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

```
result = x>y ? x-y : y-x;
```

```
if (Test)
  val = Then-Expr;
else
  val = Else-Expr;
```

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then-Expr;
Done:
  . . .
Else:
  val = Else-Expr;
  goto Done;
```

Test is expression returning integer
- = 0 interpreted as false
- ≠ 0 interpreted as true

Create separate code regions for *then* and *else* expressions

Execute appropriate one

19

Typical control flow code in assembly

```
       cmpl %eax,%ebx
       je label
       …
       …
       …
label: …
```

21

PC Relative Addressing

```
0x100     cmp   r2, r3     0x1000
0x102     je    0x70       0x1002
0x104     …                0x1004
…         …                …
0x174     add   r3, r4     0x1074
```

- **Jump instruction encodes *offset* from next instruction to destination PC.**
  - (Not the absolute address of the destination.)
  - PC relative branches are <u>relocatable</u>
  - Absolute branches are not (or they take a lot work to relocate)

Compiling Loops

C/Java code:

```
while ( sum != 0 ) {
   <loop body>
}
```

Machine code:

```
loopTop:    cmpl   $0, %eax
            je     loopDone
              <loop body code>
            jmp    loopTop
loopDone:
```

How to compile other loops should be straightforward

The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop

"Do-While" Loop Example

C Code

```
int fact_do(int x) {
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x) {
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1) goto loop;
  return result;
}
```

Use backward branch to continue looping

Only take branch when "while" condition holds

"Do-While" Loop Compilation

Register	Variable
%edx	x
%eax	result

Goto Version

```
int
fact_goto(int x) {
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

Assembly

```
fact_goto:
  pushl %ebp
  movl %esp,%ebp
  movl $1,%eax
  movl 8(%ebp),%edx

.L11:
  imull %edx,%eax
  decl %edx
  cmpl $1,%edx
  jg .L11

  movl %ebp,%esp
  popl %ebp
  ret
```

Translation?

Why put the loop condition at the end?

General "Do-While" Translation

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

Body:
```
{
    Statement1;
    Statement2;
     …
    Statementn;
}
```

Test **returns integer**

= 0 interpreted as false

≠ 0 interpreted as true

"While" Loop Translation

C Code

```
int fact_while(int x) {
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  }
  return result;
}
```

Goto Version

```
int fact_while_goto(int x) {
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```

Used by GCC for both IA32 and x86-64

Test at end, first iteration jumps over body to test.

"While" Loop Example

```
int fact_while(int x) {
  int result = 1;
  while (x > 1) {
    result *= x;
    x--;
  };
  return result;
}
```

```
# x in %edx, result in %eax
  jmp   .L34          #   goto Middle
.L35:                 # Loop:
  imull %edx, %eax    #   result *= x
  decl  %edx          #   x--
.L34:                 # Middle:
  cmpl  $1, %edx      #   x:1
  jg    .L35          #   if >, goto
                      #        Loop
```


[optional]

"For" Loop Example: Square-and-Multiply

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p) {
   int result;
   for (result = 1; p != 0; p = p>>1) {
      if (p & 0x1) {
         result *= x;
      }
      x = x*x;
   }
   return result;
}
```

$x^m * x^n = \mathbf{x^{m+n}}$

0 … 0 1 1 0 1 = 13

$1^{2^{31}} * \ldots * 1^{16} * x^8 * x^4 * 1^2 * x^1 = x^{13}$

$1 = x^0$ $x = x^1$

Algorithm

Exploit bit representation: $p = p_0 + 2p_1 + 2^2p_2 + \ldots\ 2^{n-1}p_{n-1}$

Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot \underbrace{(\ldots((z_{n-1}{}^2)^2)\ldots)^2}_{n\text{–}1 \text{ times}}$

$z_i = 1$ when $p_i = 0$

$z_i = x$ when $p_i = 1$

Complexity $O(\log p) = O(\mathrm{sizeof}(p))$

Example

$3^{10} = 3^2 * 3^8$

$= 3^2 * ((3^2)^2)^2$

[optional]

ipwr Computation

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p) {
   int result;
   for (result = 1; p != 0; p = p>>1) {
      if (p & 0x1) {
         result *= x;
      }
      x = x*x;
   }
   return result;
}
```

before iteration	result	x=3	p=10
1	1	3	10=1010_2
2	1	9	5= 101_2
3	9	81	2= 10_2
4	9	6561	1= 1_2
5	59049	43046721	0_2

30

"For" Loop Example

```
int result;
for (result = 1; p != 0; p = p>>1) {
   if (p & 0x1) {
      result *= x;
   }
   x = x*x;
}
```

General Form

```
for (Initialize; Test; Update)
    Body
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{
   if (p & 0x1) {
      result *= x;
   }
   x = x*x;
}
```

31

"For"→ "While"

For Version

```
for (Initialize; Test; Update )

     Body
```

While Version

```
Initialize;
while (Test ) {
     Body
     Update ;
}
```

Goto Version

```
  Initialize;
  goto middle;
loop:
  Body
  Update ;
middle:
  if (Test)
    goto loop;
done:
```

32

For-Loop: Compilation

For Version

```
for (Initialize; Test; Update )

     Body
```

Goto Version

```
  Initialize;
  goto middle;
loop:
  Body
  Update ;
middle:
  if (Test)
    goto loop;
done:
```

```
for (result = 1; p != 0; p = p>>1) {
  if (p & 0x1) {
    result *= x;
  }
  x = x*x;
}
```

```
  result = 1;
  goto middle;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
middle:
  if (p != 0)
    goto loop;
done:
```

33

```
long switch_eg (unsigned
   long x, long y, long z) {
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statements

Multiple case labels
- Here: 5, 6

Fall through cases
- Here: 2

Missing cases
- Here: 4

Lots to manage, we need a *jump table*

Jump Table Structure

Switch Form

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

Jump Table

```
JTab:  Targ0
       Targ1
       Targ2
         •
         •
         •
       Targn-1
```

Jump Targets

```
Targ0:   Code Block 0
Targ1:   Code Block 1
Targ2:   Code Block 2
           •
           •
           •
Targn-1: Code Block n–1
```

Approximate Translation

```
target = JTab[x];
goto target;
```

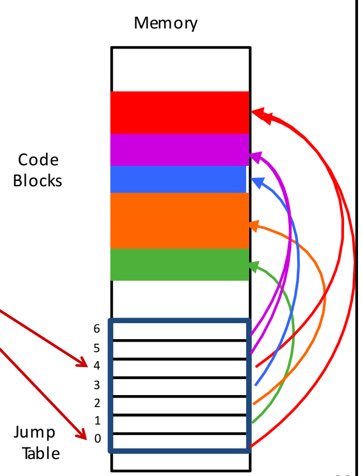

Jump Table Structure

C code:

```
switch(x) {
  case 1: <some code>
          break;
  case 2: <some code>
  case 3: <some code>
          break;
  case 5:
  case 6: <some code>
          break;
  default: <some code>
}
```

We can use the jump table when x <= 6:

```
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```



Jump Table (IA32)

declaring data, not instructions

4-byte memory alignment

Jump table

```
.section  .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

"long" as in movl: 4 bytes

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
case 5:
case 6:       // .L60
    w -= z;
    break;
default:      // .L61
    w = 2;
}
```

Switch Statement Example (IA32)

```
long switch_eg(unsigned long x, long y,
   long z) {
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section .rodata
  .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

Setup:

```
switch_eg:
    pushl %ebp              # Setup
    movl  %esp, %ebp        # Setup
    pushl %ebx              # Setup
    movl  $1, %ebx          # w = 1
    movl  8(%ebp), %edx     # edx = x
    movl  16(%ebp), %ecx    # ecx = z
    cmpl  $6, %edx
    ja    .L61
    jmp   *.L62(,%edx,4)
```

Translation?

Switch Statement Example (IA32)

```
long switch_eg(unsigned long x, long y,
   long z) {
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section .rodata
  .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

Setup:

```
switch_eg:
    pushl %ebp              # Setup
    movl  %esp, %ebp        # Setup
    pushl %ebx              # Setup
    movl  $1, %ebx          # w = 1
    movl  8(%ebp), %edx     # edx = x
    movl  16(%ebp), %ecx    # ecx = z
    cmpl  $6, %edx          # x:6
    ja    .L61              # if > 6 goto
default
    jmp   *.L62(,%edx,4)    # goto JTab[x]
```

jump above (like jg, but unsigned)

Indirect jump

Assembly Setup Explanation (IA32)

Table Structure

Each target requires 4 bytes

Base address at **.L62**

Jump table

```
.section .rodata
  .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

Jump target address modes

Direct: **jmp .L61**

Jump target is denoted by label **.L61**

Indirect: **jmp *.L62(,%edx,4)**

Start of jump table: **.L62**

Must scale by factor of 4 (labels are 32-bits = 4 bytes on IA32)

Fetch target from effective address **.L62 + edx*4**

target = JTab[x]; goto target; (only for $0 \le x \le 6$)

Code Blocks (Partial)

```
switch(x) {
  . . .
  case 2:      // .L57
    w = y/z;
    /* Fall Through */
  case 3:      // .L58
    w += z;
    break;
  . . .
  default:     // .L61
    w = 2;
}
return w;
```

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   jmp .L63
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd               # Div prep
   idivl %ecx         # y/z
   movl  %eax, %ebx # w = y/z
# Fall through – no jmp
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   jmp .L63
...
.L63
   movl  %ebx, %eax  # return w
   popl  %ebx
   leave
   ret
```

Code Blocks (Rest)

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:       // .L60
    w -= z;
    break;
  . . .
}
return w;
```

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx  # w -= z
   jmp .L63
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx     # w*= z
   jmp .L63

...
.L63
   movl  %ebx, %eax  # return w
   popl  %ebx
   leave
   ret
```


Code Blocks (Partial, return inlined)

```
switch(x) {
  . . .
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
  . . .
default:      // .L61
    w = 2;
}
```

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                  # Div prep
   idivl %ecx            # y/z
   movl  %eax, %ebx # w = y/z
# Fall through – no jmp
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```


Code Blocks (Rest, return inlined)

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:       // .L60
    w -= z;
    break;
  . . .
}
```

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx  # w -= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx     # w*= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
```


Switch machine code

Setup

Label **.L61** will mean address **0x08048630**

Label **.L62** will mean address **0x080488dc**

Assembly Code

```
switch_eg:
  . . .
  ja    .L61             # if > goto default
  jmp   *.L62(,%edx,4)   # goto JTab[x]
```

Disassembled Object Code

```
08048610 <switch_eg>:
 . . .
08048622:  77 0c                      ja     8048630
08048624:  ff 24 95 dc 88 04 08       jmp    *0x80488dc(,%edx,4)
```

Switch machine code

Jump Table

- Doesn't show up in disassembled code
- Can inspect using GDB if we know its address.
- `(gdb) x/7xw 0x080488dc`
  - Examine 7 hexadecimal format "words" (4 bytes each)
  - Use command "**help x**" to get format documentation

```
0x080488dc:
    0x08048630
    0x08048650
    0x0804863a
    0x08048642
    0x08048630
    0x08048649
    0x08048649
```

46

Matching Disassembled Targets

```
0x080488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```

```
8048630:    bb 02 00 00 00    mov
8048635:    89 d8             mov
8048637:    5b                pop
8048638:    c9                leave
8048639:    c3                ret
804863a:    8b 45 0c          mov
804863d:    99                cltd
804863e:    f7 f9             idiv
8048640:    89 c3             mov
8048642:    01 cb             add
8048644:    89 d8             mov
8048646:    5b                pop
8048647:    c9                leave
8048648:    c3                ret
8048649:    29 cb             sub
804864b:    89 d8             mov
804864d:    5b                pop
804864e:    c9                leave
804864f:    c3                ret
8048650:    8b 5d 0c          mov
8048653:    0f af d9          imul
8048656:    89 d8             mov
8048658:    5b                pop
8048659:    c9                leave
804865a:    c3                ret
```

47

Question

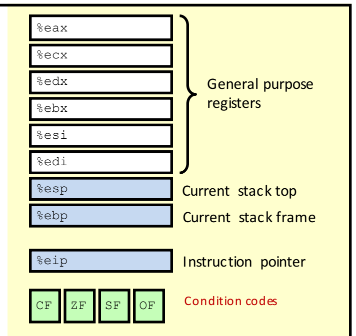
- **Would you implement this with a jump table?**

```
switch(x) {
  case 0:      <some code>
               break;
  case 10:     <some code>
               break;
  case 52000:  <some code>
               break;
  default:     <some code>
               break;
}
```

48

Quick Review

Processor State



Memory addressing modes

```
(%eax)
17(%eax)
12(%edx, %eax)
2(%ebx, %ecx, 8)
```

Immediate (constant), Register, and Memory Operands

```
subl %eax, %ecx          # ecx = ecx + eax
sall $4,%edx             # edx = edx << 4
addl 16(%ebp),%ecx       # ecx = ecx + Mem[16+ebp]
imull %ecx,%eax          # eax = eax * ecx
```

49

Quick Review

Control

carry zero sign overflow

- 1-bit condition code/flag registers `CF` `ZF` `SF` `OF`
- Set by arithmetic instructions (`addl`, `shll`, etc.), `cmp`, `test`
- Access flags with `setg`, `setle`, … instructions

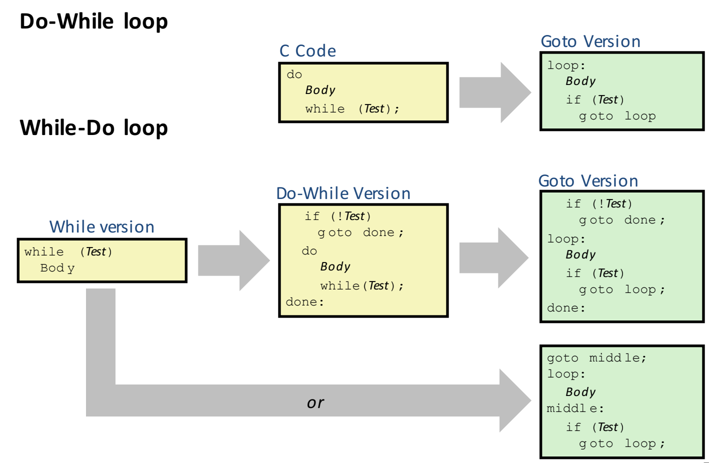
- **Conditional jumps** use flags for decisions (`jle .L4`, `je .L10`, …)
- **Unconditional jumps** always jump: `jmp`
- Direct or indirect jumps

Standard Techniques

- Loops converted to do-while form
- Large switch statements use jump tables

50

Quick Review



51