# Assembly II: Control Flow

Jo, Heeseung

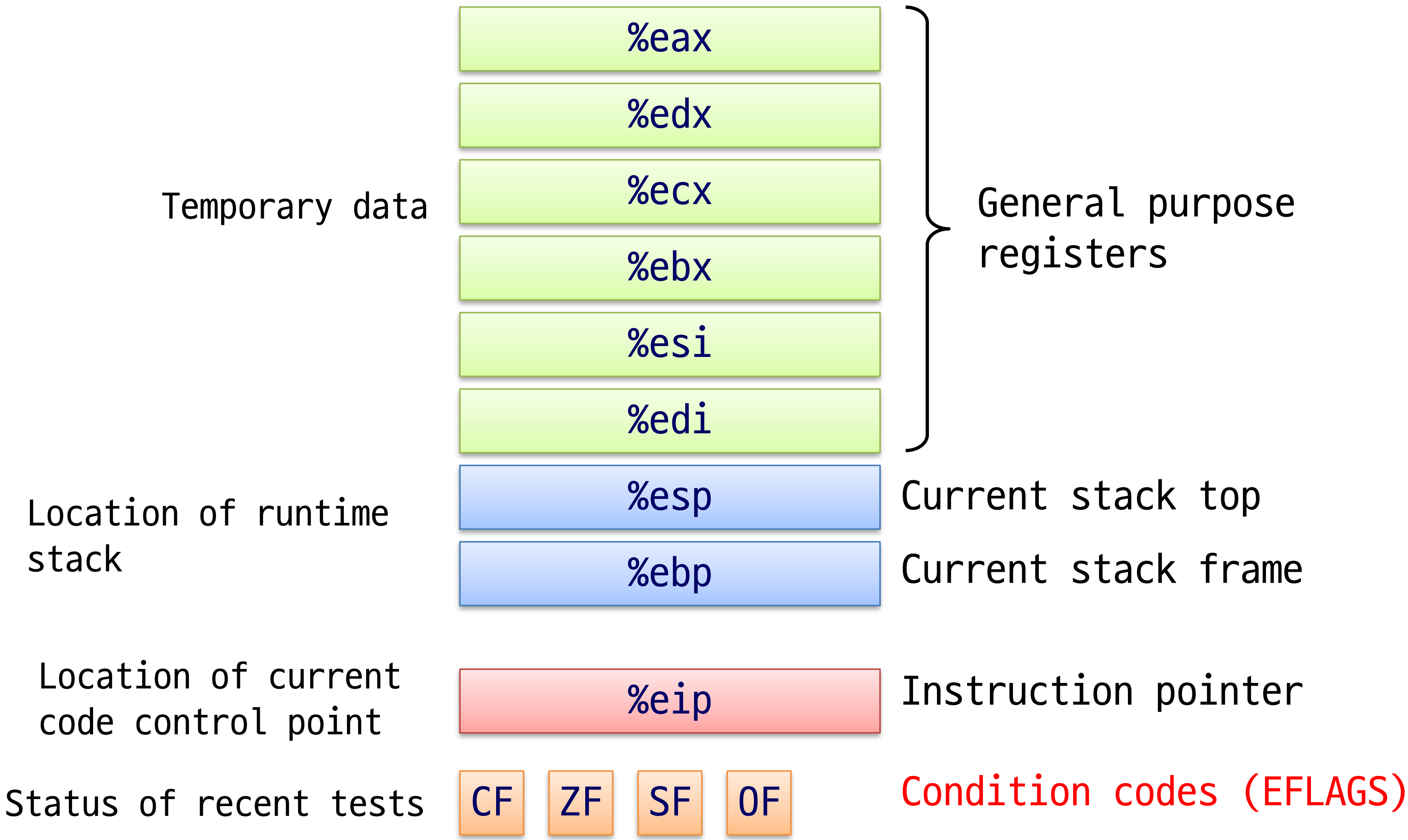
# Control Flow of Program Code

```
if (a > b) {
        do_something();
}
else {
        do_else();
}
// switch ()
```

```
while (a > b) {
        do_somthing();
}
// for () // do ... while
```

```
sub1() {
        do_something...
}
main () {
        do_a;
        sub1();
        do_b;
}
```

# IA-32 Processor State



# Setting Condition Codes (1)

Single bit registers

- CF (Carry), SF (Sign), ZF (Zero), OF (Overflow)

Implicitly set by arithmetic operations

- Example: **addl Src, Dest** (t = a + b)
- CF set if carry out from most significant bit
  - Used to detect unsigned overflow
- ZF set if **t == 0**
- SF set if **t < 0**
- OF set if two's complement overflow
  - **(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)**

Not set by **leal**, **incl**, or **decl** instruction

# Setting Condition Codes (2)

## Explicitly setting by compare instruction

- Example: **`cmpl b, a`**
- Computes **(a – b)** without saving the result (arithmetic)
- CF set if carry out from most significant bit
  - Used for unsigned comparisons
- ZF set if **a == b**
- SF set if **(a – b) < 0**
- OF set if two's complement overflow

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

# Setting Condition Codes (3)

Explicitly setting by test instruction

- Example: **testl b, a**
- Sets condition codes based on value of a and b
  - Useful to have one of the operands be a mask
- Computes **a & b** without setting destination (logical)
- ZF set when **a & b == 0**
- SF set when **a & b < 0**
- CF and OF are cleared to 0

# Jumping

```
cmpl    %eax, %edx
je      L9
```

## jX instructions

- Jump to different part of code depending on condition codes
- Jump instruction changes the value of PC(EIP)

| 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 >) |
| jae | ~CF | Above or Equal (Unsigned >=) |
| jb | CF | Below (Unsigned <) |
| jbe | CF ¦ ZF | Below or Equal (Unsigned <=) |

# Reading Condition Codes (1)

setX instructions

- Set single byte based on combinations of condition codes

| setX | Condition | Description |
|---|---|---|
| sete $R_8$ | $R_8$ <- ZF | Equal / Zero |
| setne $R_8$ | $R_8$ <- ~ZF | Not Equal / Not Zero |
| sets $R_8$ | $R_8$ <- SF | Negative |
| setns $R_8$ | $R_8$ <- ~SF | Nonnegative |
| setg $R_8$ | $R_8$ <- ~(SF ^ OF) & ~ZF | Greater (Signed >) |
| setge $R_8$ | $R_8$ <- ~(SF ^ OF) | Greater or Equal (Signed >=) |
| setl $R_8$ | $R_8$ <- (SF ^ OF) | Less (Signed <) |
| setle $R_8$ | $R_8$ <- (SF ^ OF) ¦ ZF | Less or Equal (Signed <=) |
| seta $R_8$ | $R_8$ <- ~CF & ~ZF | Above (Unsigned >) |
| setae $R_8$ | $R_8$ <- ~CF | Above or Equal (Unsigned >=) |
| setb $R_8$ | $R_8$ <- CF | Below (Unsigned <) |
| setbe $R_8$ | $R_8$ <- CF ¦ ZF | Below or Equal (Unsigned <=) |

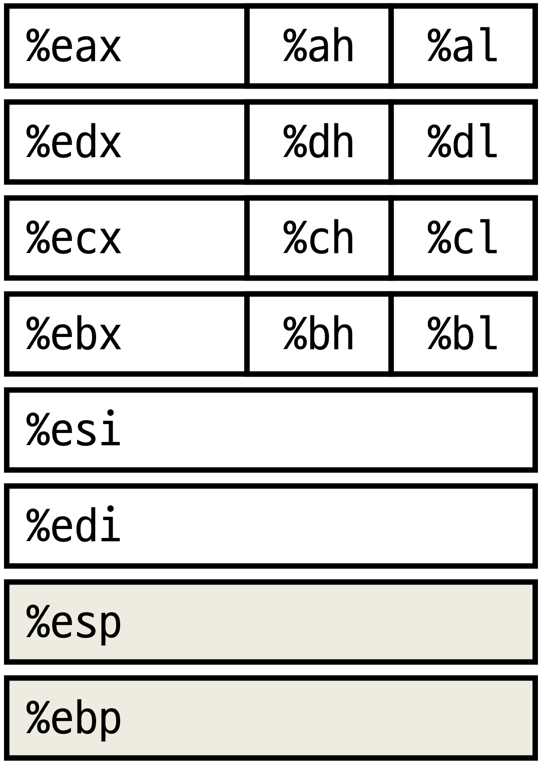
# Reading Condition Codes (2)

**setX** instructions

- One of 8 addressable byte registers

  **%ah, %al, %bh, %bl,**

  **%ch, %cl, %dh, %dl**
- Does not alter remaining 3 bytes
- Typically use **movzbl** to finish job




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

```
movl 12(%ebp),%eax        # %eax = y
cmpl %eax,8(%ebp)         # Compare x : y
setg %al                  # al = x > y
movzbl %al,%eax           # Zero rest of %eax
```

Note inverted ordering!

# Conditional Branch (1)

```c
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp                 } Set Up
        movl %esp,%ebp

        movl 8(%ebp),%edx          
        movl 12(%ebp),%eax
        cmpl %eax,%edx             } Body
        jle L9
        movl %edx,%eax
L9:

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

# Conditional Branch (2)

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

- C allows "goto" as means of transferring control
  - Closer to machine-level programming style
- Generally considered bad coding style

```
    movl 8(%ebp),%edx      # edx = x
    movl 12(%ebp),%eax     # eax = y
    cmpl %eax,%edx         # x : y
    jle L9                 # if <= goto L9
    movl %edx,%eax         # eax = x
  L9:                      # Done:
```

Skipped when x <= y

%edx is less than or equal with %eax

When a function ends, the value of %eax is the return value

# Conditional Branch (3)

```
int positive(int x)
{
  if (x > 0)
    rval = 1;
  else
    rval = 0;
  return rval;
}
```

```
    movl 8(%ebp), %edx       # edx = x
    cmpl $0, %edx            # x : 0
    jle L9                   # if <= goto L9
    movl $1, %eax            # rval = 1
L9:
    movl $0, %eax            # rval = 0
```

%edx is less than or equal with $0

What's wrong?

# Conditional Branch (4)

```
int positive(int x)
{
  int rval;
  if (x > 0)
    rval = 1;
  else
    rval = 0;
  return rval;
}
```

```
int positive(int x)
{
  int rval = 0;
  if (x <= 0)
    goto L9;
  rval = 1;
L9:
  return rval;
}
```

```
    movl 8(%ebp), %edx      # edx = x
    movl $0, %eax           # rval = 0
    cmpl $0, %edx           # x : 0
    jle L9                  # if <= goto L9
    movl $1, %eax           # rval =1
L9:
```

%edx is less than or equal with $0

# Ex.

What does this code mean?

```
_func:
   pushl %ebp              # setup
   movl %esp,%ebp          # setup

   movl 8(%ebp),%edx
   movl 12(%ebp),%eax
   cmpl %eax, %edx
   jg L9
   je L10
   movl %edx,%eax
L10:
   movl $0,%eax
L9:

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

```
int func (int x, int y)
{
```

# Looping statements

```
int total () {
  int sum = 0, x = 1;
  do {
    sum = sum + x;
    x = x + 1;
  } while (x <= 10);
  return sum;
}
```

```
int total () {
  int sum = 0, x = 1;
  while (x <= 10) {
    sum = sum + x;
    x = x + 1;
  }
  return sum;
}
```

```
int total () {
  int sum = 0, int x;
  for ( x = 1; x <= 10; x++) {
    sum = sum + x;
  }
  return sum;
}
```

```
int total () {
  int sum = 0, x = 1;
loop:
  sum = sum + x;
  x = x + 1;
  if (x <= 10)
    goto loop;
  return sum;
}
```

# "Do-While" Loop (1)

## C Code

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

## Goto

```
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 (2)

## Goto

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

Registers

| | |
|---|---|
| %edx | x |
| %eax | result |

## Assembly

```
_fact_goto:
   pushl %ebp           # Setup
   movl %esp,%ebp       # Setup

   movl $1,%eax         # eax = 1
   movl 8(%ebp),%edx    # edx = x
L11:
   imull %edx,%eax      # result *= x
   decl %edx            # x--
   cmpl $1,%edx         # Compare x : 1
   jg L11               # if > goto loop

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

# "Do-While" Loop (3)

General "Do-While" translation

**C Code**

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

**Goto**

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

- Body can be any C statement
  - Typically compound statement:

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

- Test is expression returning integer

  = 0 interpreted as false
  ≠ 0 interpreted as true

# "While" Loop (1)

## C Code

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

## First Goto

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

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

# "While" Loop (2)

## First Goto

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

## Second Goto

```
int fact_while_goto2 (int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x - 1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

# "While" Loop (3)

General "While" translation

## C Code

```
while (Test)
  Body
```

## Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
  while(Test);
done:
```

## Goto Version

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

### C Code

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

### Goto

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

# "While" Loop (4)

## Second Goto

```
int fact_while_goto2
(int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x - 1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

## Assembly

```
_fact_while_goto2:
   pushl %ebp              # Setup
   movl %esp,%ebp          # Setup

   movl $1,%eax            # eax = 1
   movl 8(%ebp),%edx       # edx = x
   cmpl $1,%edx            # x : 1
   jle L9                  # (!(x > 1))
L11:
   imull %edx,%eax         # result *= x
   decl %edx               # x--
   cmpl $1,%edx            # Compare x : 1
   jg L11                  # if > goto loop
L9:

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

# "For" Loop (1)

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

Algorithm

- Exploit property that $p = p_0 + 2p_1 + 4p_2 + 8p_3 + \dots 2^{n-1}p_{n-1}$
- Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot ((z_3{}^2)^2)^2 \cdot \dots \cdot (\dots((z_{n-1}{}^2)^2)\dots)^2$
  - $z_i = 1$ when $p_i = 0$
  - $z_i = x$ when $p_i = 1$
- Complexity $O(\log p)$

Example

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

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

# "For" Loop (2)

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

## General Form

```
for (Init; Test; Update )

     Body
```

*Init*

```
result = 1
```

*Test*

```
p != 0
```

*Body*

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

*Update*

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

# "For" Loop (3)

## For Version

```
for (Init; Test; Update )

    Body
```

## While Version

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

## Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

## Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

# "For" Loop (4)

## Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

→

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

*Init*

```
result = 1
```

*! Test*

```
p == 0
```

*Body*

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

*Update*

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

*Test*

```
p != 0
```

# "For" Loop (5)

```
int ipwr_for(int x,
unsigned p) {

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

}
```

```
_ipwr_for:
   pushl %ebp                 # Setup
   movl %esp,%ebp             # Setup

   movl 8(%ebp), %ecx         # %ecx = x
   movl 12(%ebp), %edx        # %edx = p
   movl $1, %eax              # result = 1
   cmpl $0, %edx              # (p == 0)
   je L2
L5:
   testb $1, %dl              # (p & 0x1)
   je L4                      # (p & 0x1) == 0
   imull %ecx, %eax           # result = result * x
L4:
   imull %ecx, %ecx           # x = x * x
   shrl %edx                  # p = p >> 1; p != 0
   jne L5
L2:

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

# switch vs. if-then-else

```
char change (int x) {
  if (x==0)
    return '+';
  else if (x==1)
    return '-';
  else if (x==2)
    return '*';
  else if (x==3)
    return '/';
}
```

```
char change (int x) {
  switch (x) {
    case 0 :  return '+';
    case 1 :  return '*';
    case 2 :  return '-';
    case 3 :  return '/';
  }
}
```

# "Switch" Statement (1)

Implementation options

1. Series of conditionals
   - Good if few cases
   - Slow if many
2. Jump table
   - Lookup branch target
   - Avoids conditionals
   - Possible when cases are small integer constants

- GCC
  - Picks one based on case structure
- Bug in example code ?

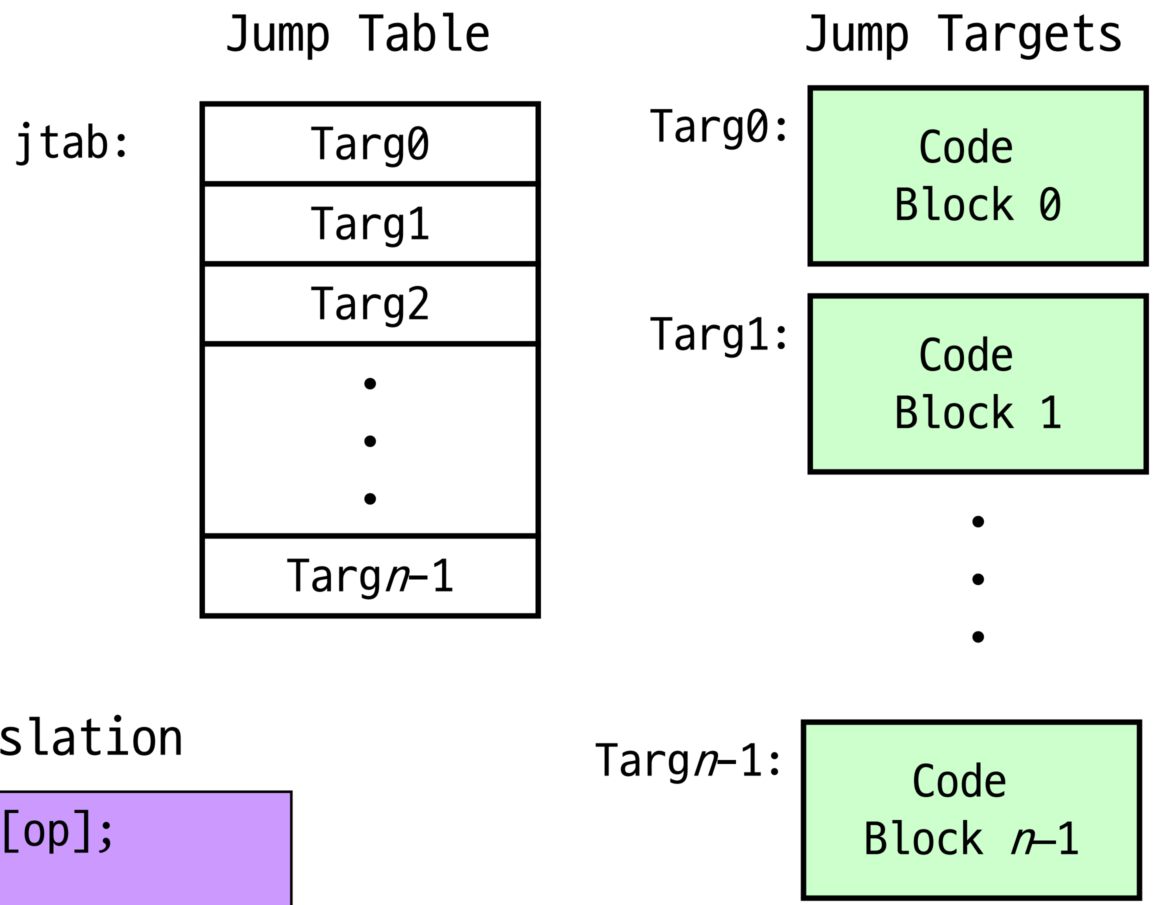
```
typedef enum {
   ADD, MULT, MINUS, DIV,
   MOD, BAD
}  op_type;

char unparse_symbol
(op_type op) {
  switch (op) {
  case ADD :  return '+';
  case MULT:  return '*';
  case MINUS: return '-';
  case DIV:   return '/';
  case MOD:   return '%';
  case BAD:   return '?';
  }
}
```

# "Switch" Statement (2)

Jump table structure

Switch Form

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




Approx. Translation

```
target = JTab[op];
goto *target;
```

# "Switch" Statement (3)

## Branching Possibilities

```
typedef enum {
   ADD, MULT, MINUS, DIV, MOD, BAD
}  op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

## Enumerated Values

| | |
|---|---|
| ADD | 0 |
| MULT | 1 |
| MINUS | 2 |
| DIV | 3 |
| MOD | 4 |
| BAD | 5 |

```
unparse_symbol:
   pushl %ebp              # Setup
   movl %esp,%ebp          # Setup

   movl 8(%ebp),%eax       # eax = op
   cmpl $5,%eax            # Compare op : 5
   ja .L49                 # If > goto done
   jmp *.L57(,%eax,4)      # goto Table[op]
```

# "Switch" Statement (4)

```
unparse_symbol:
    pushl %ebp              # Setup
    movl %esp,%ebp          # Setup

    movl 8(%ebp),%eax       # eax = op
    cmpl $5,%eax            # Compare op : 5
    ja .L49                 # If > goto done
    jmp *.L57(,%eax,4)      # goto Table[op]
```

Symbolic labels

- Labels of form .LXX translated into addresses by assembler

Table structure

- Each target requires 4 bytes, Base address at **.L57**

Jumping

- **jmp .L49**
  - Jump target is denoted by label **.L49**
- **jmp *.L57(,%eax,4)**
  - Start of jump table denoted by label **.L57**
  - Register **%eax** holds **op**
  - Must scale by factor of 4 to get offset into table
  - Fetch target from effective address **.L57 + op * 4**

# "Switch" Statement (5)

```
jmp *.L57(,%eax,4)
```

## Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51   #Op = 0
  .long .L52   #Op = 1
  .long .L53   #Op = 2
  .long .L54   #Op = 3
  .long .L55   #Op = 4
  .long .L56   #Op = 5
```

## Enumerated Values

```
ADD     0
MULT    1
MINUS   2
DIV     3
MOD     4
BAD     5
```

## Targets & Completion

```
.L51:
     movl $43, %eax   # '+'
     jmp .L49
.L52:
     movl $42, %eax   # '*'
     jmp .L49
.L53:
     movl $45, %eax   # '-'
     jmp .L49
.L54:
     movl $47, %eax   # '/'
     jmp .L49
.L55:
     movl $37, %eax   # '%'
     jmp .L49
.L56:
     movl $63, %eax   # '?'
     # Fall Through to .L49
```

# "Switch" Statement (6)

Switch statement completion

```
.L49:                       # Done:
     movl %ebp,%esp         # Finish
     popl %ebp              # Finish
     ret                    # Finish
```

- What value returned when op is invalid?
  - Register %eax set to op at beginning of procedure
  - This becomes the return value

Advantage of jump table

- Can do k-way branch in O(1) operations

# "Switch" Statement (7)

```
unparse_symbol:
    pushl %ebp            # Setup
    movl %esp,%ebp        # Setup

    movl 8(%ebp),%eax     # eax = op
    cmpl $5,%eax          # Compare op : 5
    ja .L49               # If > goto done
    jmp *.L57(,%eax,4)    # goto Table[op]

.section .rodata
    .align 4
.L57:
   .long .L51     #Op = 0
   .long .L52     #Op = 1
   .long .L53     #Op = 2
   .long .L54     #Op = 3
   .long .L55     #Op = 4
   .long .L56     #Op = 5
```

```
.L51:
     movl $43, %eax      # '+'
     jmp .L49
.L52:
     movl $42, %eax      # '*'
     jmp .L49
.L53:
     movl $45, %eax      # '-'
     jmp .L49
.L54:
     movl $47, %eax      # '/'
     jmp .L49
.L55:
     movl $37, %eax      # '%'
     jmp .L49
.L56:
     movl $63, %eax      # '?'
     # Fall Through to .L49

.L49:                   # Done:
     movl %ebp,%esp     # Finish
     popl %ebp          # Finish
     ret                # Finish
```

# "Switch" Statement (8)

Sparse switch example

- Not practical to use jump table
  - Would require 1000 jump table entries
- Obvious translation into if-then-else would have max. of 10 tests
- Here, suppose x is multiple of 111

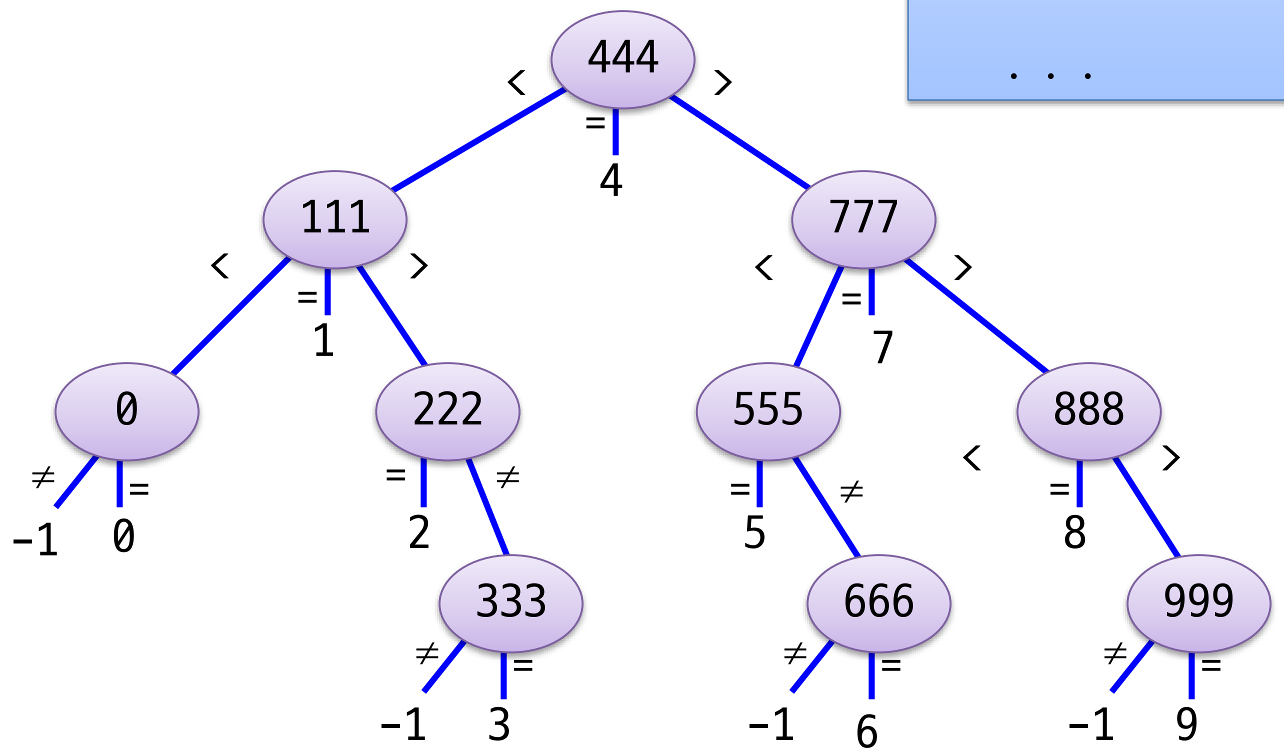
```
/* Return x/111 if x is
   multiple of 111 && <= 999.
   Return -1 otherwise */
int div111(int x) {
  switch(x) {
  case   0: return 0;
  case 111: return 1;
  case 222: return 2;
  case 333: return 3;
  case 444: return 4;
  case 555: return 5;
  case 666: return 6;
  case 777: return 7;
  case 888: return 8;
  case 999: return 9;
  default: return -1;
  }
}
```

# "Switch" Statement (9)

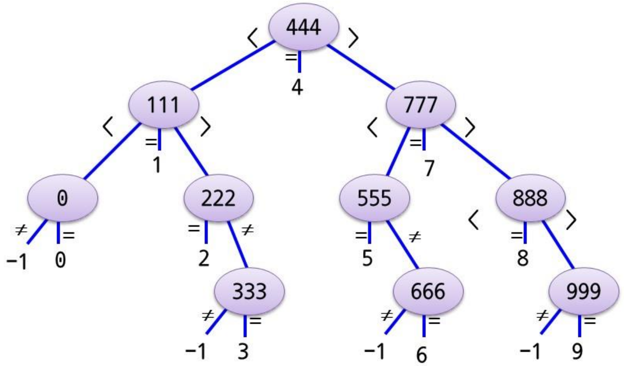
Sparse switch code structure

- Organizes cases as binary tree
- Logarithmic performance

```
movl 8(%ebp),%eax      # get x
cmpl $444,%eax         # x:444
je L8
jg L16
cmpl $111,%eax         # x:111
je L5
jg L17
testl %eax,%eax        # x:0
je L4
jmp L14

. . .
```



# "Switch" Statement (10)




- Compares x to possible case values
- Jumps different places depending on outcomes

```
    movl 8(%ebp),%eax      # get x
    cmpl $444,%eax         # x:444
    je L8
    jg L16
    cmpl $111,%eax         # x:111
    je L5
    jg L17
    testl %eax,%eax        # x:0
    je L4
    jmp L14

    . . .
```

```
    . . .
L5:
    movl $1,%eax
    jmp L19
L6:
    movl $2,%eax
    jmp L19
L7:
    movl $3,%eax
    jmp L19
L8:
    movl $4,%eax
    jmp L19
    . . .
```

- We can finish it in max. 4 comparisons

# Summary

C Control

- if-then-else
- do-while
- while, for
- switch

Assembler control

- Jump
- Conditional jump
- Indirect jump

Compiler

- Must generate assembly code to implement more complex control

# Summary

Standard techniques

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

Conditions in CISC

- CISC machines generally have condition code registers

Conditions in RISC

- Use general registers to store condition information
- Special comparison instructions
- E.g., on Alpha: cmple $16, 1, $1
  - Sets register $1 to 1 when $16 <= 1

# Ex.

What does this code mean?

```
_func:
   pushl %ebp              # Setup
   movl %esp,%ebp          # Setup

   movl $0,%eax
   movl 8(%ebp),%edx

L11:
   addl %edx,%eax
   incl %edx
   cmpl $10,%edx
   jg L11

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

```
int func (int x)
{
```