Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

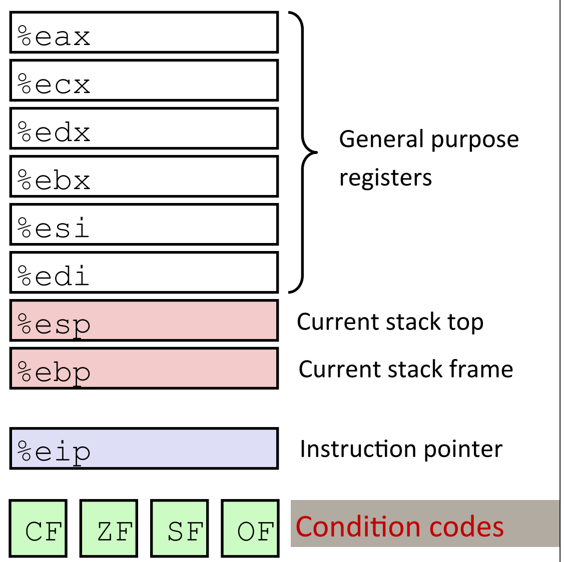
With Slides from Bryant and O'Hallaron

University of Minnesota

Control Flow

Processor State (IA32, Partial)

- Information about currently executing program
  - Temporary data (`%eax`, …)
  - Location of runtime stack (`%ebp`,`%esp`)
  - Location of current code control point (`%eip`, …)
  - Status of recent tests (CF, ZF, SF, OF)



With Slides from Bryant and O'Hallaron

Condition Codes (Implicit Setting)

- Single bit registers
  - •CF Carry Flag (for unsigned) SF Sign Flag (for signed)
  - •ZF Zero Flag OF Overflow Flag (for signed)
- Implicitly set (think of it as side effect) by arithmetic operations
  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
  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- Not set by `lea` instruction

With Slides from Bryant and O'Hallaron

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - `cmpl/cmpq` Src2, Src1
  - `cmpl b,a` like computing `a-b` without setting destination

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

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - `testl/testq` Src2, Src1
    `testl b,a` like computing `a&b` without setting destination

  - Sets condition codes based on value of Src1 & Src2
  - Useful to have one of the operands be a mask

  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`

Reading Condition Codes

- SetX Instructions
  - Set single byte 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)

Reading Condition Codes (Cont.)

- SetX Instructions
  - Set single byte based on combinations of condition codes
  - One of 8 addressable byte registers
    - Embedded within first 4 integer registers
    - Does not alter remaining 3 bytes
    - Typically use `movzbl` to finish job


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

Body

```
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!

Reading Condition Codes: x86-64

- SetX Instructions:
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

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

```
long lgt (long x, long y)
{
  return x > y;
}
```

```
cmpl %esi, %edi
setg %al
movzbl %al, %eax
```

```
cmpq %rsi, %rdi
setg %al
movzbl %al, %eax
```

32-bit instructions set high order 32 bits to 0!

Jumping

jX Instructions: Jump to different part of code

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)

If-then

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     .L6
    subl    %eax, %edx        } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:                          } Body2b
    subl %edx, %eax
.L7:                          } Finish
    popl %ebp
    ret
```

Conditional Branch Example (Cont.)

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

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

```
absdiff:
    pushl   %ebp             } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx    } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx       } Body2a
    movl    %edx, %eax
    jmp .L7                  } Body2b
.L6:
    subl %edx, %eax          } Finish
.L7:
    popl %ebp
    ret
```

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp             } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx    } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx       } Body2a
    movl    %edx, %eax
    jmp .L7                  } Body2b
.L6:
    subl %edx, %eax          } Finish
.L7:
    popl %ebp
    ret
```

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp              } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx     } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx        } Body2a
    movl    %edx, %eax
    jmp .L7                   } Body2b
.L6:
    subl %edx, %eax           } Finish
.L7:
    popl %ebp
    ret
```

With Slides from Bryant and O'Hallaron

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp              } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx     } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx        } Body2a
    movl    %edx, %eax
    jmp .L7                   } Body2b
.L6:
    subl %edx, %eax           } Finish
.L7:
    popl %ebp
    ret
```

With Slides from Bryant and O'Hallaron

General Conditional Expression Translation

C Code

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

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

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Test is expression returning integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

With Slides from Bryant and O'Hallaron

Conditional Move Example: x86-64

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

x in %edi

y in %esi

```
absdiff:
   movl    %edi, %edx
   subl    %esi, %edx    # tval = x-y
   movl    %esi, %eax
   subl    %edi, %eax    # result = y-x
   cmpl    %esi, %edi    # Compare x:y
   cmovg   %edx, %eax    # If >, result = tval
   ret
```

With Slides from Bryant and O'Hallaron

Bad Cases for Conditional Move

Expensive Computations

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

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Loops

"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

Goto Version

```
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
```

General "Do-While" Translation

C Code

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

Goto Version

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

Body can be any C statement or compound statement:

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

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

"While" Loop Example #1

C Code

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

First Goto Version

```
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

```
fact_while:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    $1, %eax
        cmpl    $1, %edx
        jle     .L4
        movl    $1, %eax
        movl    $1, %ecx
.L5:
        imull   %edx, %eax
        subl    $1, %edx
        cmpl    %ecx, %edx
        jne     .L5
.L4:
        popl    %ebp
        ret
```

p Translation

Second Goto Version

```
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;
}
```

21 25

With Slides from Bryant and O'Hallaron

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:
```

With Slides from Bryant and O'Hallaron

"For" Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Is this code equivalent to other versions?

"For" Loop Form

General Form

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

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```

"For" Loop → While Loop

For Version

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

While Version

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

With Slides from Bryant and O'Hallaron

"For" Loop → … → Goto

For Version

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

While Version

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

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

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

With Slides from Bryant and O'Hallaron

"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;            Init
  i = 0;
  if (!(i < WSIZE))          ! Test
    goto done;
 loop:                       Body
  {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }      Update
  i++;               Test
  if (i < WSIZE)
    goto loop;
 done:
  return result;
}
```

With Slides from Bryant and O'Hallaron

Switch Statement

With Slides from Bryant and O'Hallaron

Switch Statement Example

```
long switch_eg
    (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;
}
```

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

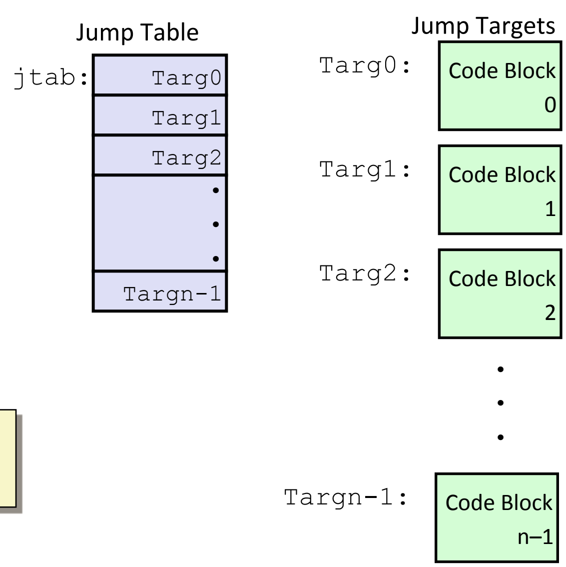
Jump Table Structure

Switch Form

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

Approximate Translation

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



With Slides from Bryant and O'Hallaron

Switch Statement Example (IA32)

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

Note that **w** not initialized here

What range of values takes default?

Setup:

```
switch_eg:
   pushl  %ebp             # Setup
   movl   %esp, %ebp       # Setup
   movl   8(%ebp), %eax    # %eax = x
   cmpl   $6, %eax         # Compare x:6
   ja     .L2              # If unsigned > goto default
   jmp    *.L7(,%eax,4)    # Goto *JTab[x]
```

With Slides from Bryant and O'Hallaron

Switch Statement Example (IA32)

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

Jump table

```
.section    .rodata
  .align 4
.L7:
  .long     .L2 # x = 0
  .long     .L3 # x = 1
  .long     .L4 # x = 2
  .long     .L5 # x = 3
  .long     .L2 # x = 4
  .long     .L6 # x = 5
  .long     .L6 # x = 6
```

Setup:

```
switch_eg:
   pushl  %ebp             # Setup
   movl   %esp, %ebp       # Setup
   movl   8(%ebp), %eax    # eax = x
   cmpl   $6, %eax         # Compare x:6
   ja     .L2              # If unsigned > goto default
   jmp    *.L7(,%eax,4)    # Goto *JTab[x]
```

Indirect jump → jmp

With Slides from Bryant and O'Hallaron

Assembly Setup Explanation

- Table Structure
  - Each target requires 4 bytes
  - Base address at `.L7`
- Jumping
  - Direct: **`jmp .L2`**
  - Jump target is denoted by label `.L2`
  - Indirect: **`jmp *.L7(,%eax,4)`**
  - Start of jump table: `.L7`
  - Must scale by factor of 4 (labels have 32-bits = 4 Bytes on IA32)
  - Fetch target from effective Address `.L7 + eax*4`
    - Only for $0 \le x \le 6$

Jump table

```
.section   .rodata
  .align 4
.L7:
  .long    .L2 # x = 0
  .long    .L3 # x = 1
  .long    .L4 # x = 2
  .long    .L5 # x = 3
  .long    .L2 # x = 4
  .long    .L6 # x = 5
  .long    .L6 # x = 6
```

With Slides from Bryant and O'Hallaron

Jump Table

Jump table

```
.section   .rodata
  .align 4
.L7:
  .long    .L2 # x = 0
  .long    .L3 # x = 1
  .long    .L4 # x = 2
  .long    .L5 # x = 3
  .long    .L2 # x = 4
  .long    .L6 # x = 5
  .long    .L6 # x = 6
```

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L4
    w = y/z;
    /* Fall Through */
case 3:       // .L5
    w += z;
    break;
case 5:
case 6:       // .L6
    w -= z;
    break;
default:      // .L2
    w = 2;
}
```

With Slides from Bryant and O'Hallaron

Handling Fall-Through

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
   . . .
}
```

```
case 3:
        w = 1;
        goto merge;
```

```
    case 2:
        w = y/z;
merge:
        w += z;
```


Code Blocks (Partial)

```
switch(x) {
case 1:       // .L3
      w = y*z;
      break;
 . . .
case 3:       // .L5
    w += z;
    break;
  . . .
default:      // .L2
    w = 2;
}
```

```
.L2:                # Default
  movl  $2, %eax    # w = 2
  jmp   .L8         # Goto done

.L5:                # x == 3
  movl  $1, %eax    # w = 1
  jmp   .L9         # Goto merge

.L3:                # x == 1
  movl  16(%ebp), %eax   # z
  imull 12(%ebp), %eax   # w = y*z
  jmp   .L8         # Goto done
```

Code Blocks (Rest)

```
switch(x) {
  . . .
  case 2:  // .L4
      w = y/z;
      /* Fall Through */
  merge:    // .L9
      w += z;
      break;
  case 5:
  case 6:  // .L6
      w -= z;
      break;
}
```

```
.L4:                # x == 2
  movl  12(%ebp), %edx
  movl  %edx, %eax
  sarl  $31, %edx
  idivl 16(%ebp)  # w = y/z

.L9:                # merge:
  addl  16(%ebp), %eax # w += z
  jmp   .L8         # goto done

.L6:                # x == 5, 6
  movl  $1, %eax        # w = 1
  subl  16(%ebp), %eax # w = 1-z
```


Switch Code (Finish)

```
return w;
```

```
.L8:                # done:
  popl  %ebp
  ret
```

- Noteworthy Features
  - Jump table avoids sequencing through cases
    - Constant time, rather than linear
  - Use jump table to handle holes and duplicate tags
  - Use program sequencing to handle fall-through
  - Don't initialize w = 1 unless really need it

x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
  . . .
}
```

```
.L3:
  movq    %rdx, %rax
  imulq   %rsi, %rax
  ret
```

Jump Table

```
  .section  .rodata
  .align 8
.L7:
  .quad     .L2     # x = 0
  .quad     .L3     # x = 1
  .quad     .L4     # x = 2
  .quad     .L5     # x = 3
  .quad     .L2     # x = 4
  .quad     .L6     # X = 5
  .quad     .L6     # x = 6
```

With Slides from Bryant and O'Hallaron

IA32 Object Code

- Setup
  - Label `.L2` becomes address `0x8048422`
  - Label `.L7` becomes address `0x8048660`

Assembly Code

```
switch_eg:
  . . .
  ja     .L2            # If unsigned > goto default
  jmp    *.L7(,%eax,4)  # Goto *JTab[x]
```

Disassembled Object Code

```
08048410 <switch_eg>:
 . . .
 8048419:77 07                  ja     8048422 <switch_eg+0x12>
 804841b:ff 24 85 60 86 04 08   jmp    *0x8048660(,%eax,4)
```

With Slides from Bryant and O'Hallaron

IA32 Object Code (cont.)

- Jump Table
  - Doesn't show up in disassembled code
  - Can inspect using GDB
  - `gdb switch`
  - `(gdb) x/7xw 0x8048660`
    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "`help x`" to get format documentation

```
0x8048660:    0x08048422    0x08048432    0x0804843b    0x08048429
0x8048670:    0x08048422    0x0804844b    0x0804844b
```

With Slides from Bryant and O'Hallaron

IA32 Object Code (cont.)

- Deciphering Jump Table

```
0x8048660:    0x08048422    0x08048432    0x0804843b    0x08048429
0x8048670:    0x08048422    0x0804844b    0x0804844b
```

Address	Value	x
0x8048660	0x8048422	0
0x8048664	0x8048432	1
0x8048668	0x804843b	2
0x804866c	0x8048429	3
0x8048670	0x8048422	4
0x8048674	0x804844b	5
0x8048678	0x804844b	6

With Slides from Bryant and O'Hallaron

Disassembled Targets

```
8048422:   b8 02 00 00 00     mov    $0x2,%eax
8048427:   eb 2a              jmp    8048453 <switch_eg+0x43>
8048429:   b8 01 00 00 00     mov    $0x1,%eax
804842e:   66 90              xchg   %ax,%ax # noop
8048430:   eb 14              jmp    8048446 <switch_eg+0x36>
8048432:   8b 45 10           mov    0x10(%ebp),%eax
8048435:   0f af 45 0c        imul   0xc(%ebp),%eax
8048439:   eb 18              jmp    8048453 <switch_eg+0x43>
804843b:   8b 55 0c           mov    0xc(%ebp),%edx
804843e:   89 d0              mov    %edx,%eax
8048440:   c1 fa 1f           sar    $0x1f,%edx
8048443:   f7 7d 10           idivl  0x10(%ebp)
8048446:   03 45 10           add    0x10(%ebp),%eax
8048449:   eb 08              jmp    8048453 <switch_eg+0x43>
804844b:   b8 01 00 00 00     mov    $0x1,%eax
8048450:   2b 45 10           sub    0x10(%ebp),%eax
8048453:   5d                 pop    %ebp
8048454:   c3                 ret
```


Matching Disassembled Targets

Value
0x8048422
0x8048432
0x804843b
0x8048429
0x8048422
0x804844b
0x804844b

```
8048422:   mov    $0x2,%eax
8048427:   jmp    8048453 <switch_eg+0x43>
8048429:   mov    $0x1,%eax
804842e:   xchg   %ax,%ax
8048430:   jmp    8048446 <switch_eg+0x36>
8048432:   mov    0x10(%ebp),%eax
8048435:   imul   0xc(%ebp),%eax
8048439:   jmp    8048453 <switch_eg+0x43>
804843b:   mov    0xc(%ebp),%edx
804843e:   mov    %edx,%eax
8048440:   sar    $0x1f,%edx
8048443:   idivl  0x10(%ebp)
8048446:   add    0x10(%ebp),%eax
8048449:   jmp    8048453 <switch_eg+0x43>
804844b:   mov    $0x1,%eax
8048450:   sub    0x10(%ebp),%eax
8048453:   pop    %ebp
8048454:   ret
```

Summarizing

- C Control
  - if-then-else
  - do-while
  - while
  - for
  - switch
- Assembler Control
  - jump
  - Conditional jump
- Compiler
  - Must generate assembly code to implement more complex control