CS429: Computer Organization and Architecture

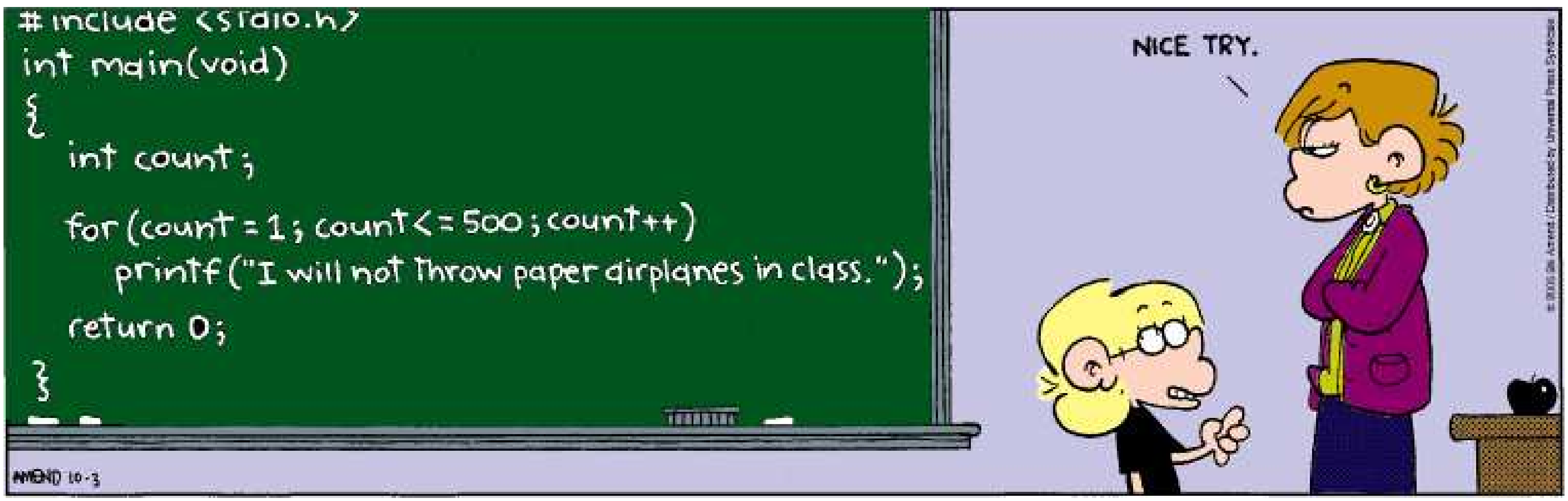
Instruction Set Architecture III

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: October 9, 2019 at 18:12

Controlling Program Execution

- We can now generate programs that execute linear sequences of instructions: access registers and memory, perform computations
- But what about loops, conditions, etc.?
- Need ISA support for:
  - comparing and testing data values
  - directing program control
    - jump to some instruction that isn't just the next one in sequence
    - Do so based on some condition that has been tested.



Processor State (x86-64, Partial)

Information about currently executing program.

- Temporary data (`%rax`, ...)
- Location of runtime stack (`%rsp`)
- Location of current code control point (`%rip`)
- Status of recent tests (CF, ZF, SF, OF)*

* contained in a 64-bit `rflags` register

Registers

`%rax`	`%r8`
`%rbx`	`%r9`
`%rcx`	`%r10`
`%rdx`	`%r11`
`%rsi`	`%r12`
`%rdi`	`%r13`
`%rsp`	`%r14`
`%rbp`	`%r15`

`%rip`	Instruction pointer

CF	ZF	SF	OF	Condition codes

PC-relative Addressing

Don't use `%rip` as a general purpose register.

However, the compiler may generate *PC-relative addressing*.

```
        jmp    0x10(%rip)
```

The effective address for a PC-relative instruction address is the offset parameter added to the address of the *next instruction*. This offset is signed to allow reference to code both before and after the instruction.

Can you guess why the compiler might generate such code?

Condition Codes (Implicit Setting)

Single bit registers

- CF: carry flag (for unsigned)
- ZF: zero flag
- SF: sign flag (for signed)
- OF: overflow flag (for signed)

Implicitly set by arithmetic operations

E.g., `addq` Src, Dest

C analog: `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 overflow:

  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t >=0)`
- Condition codes not set by `lea` instruction.

Setting Condition Codes (Explicit Setting)

Explicitly set by Compare instruction

`cmpq` Src2, Src1

- `cmpq b, a` is like computing `(a - b)` without setting destination.
- CF set if carry out from most significant bit; used for unsigned computations.
- ZF set if `a == b`
- SF set if `(a-b) < 0`
- OF set if two's complement (signed) overflow:

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

Setting Condition Codes: Test Instruction

Explicitly set by Test instruction

`testq` Src2, Src1

- Sets condition codes based on value of (Src1 & Src2).
- Often useful to have one of the operands be a mask.
- `testq b, a` is like computing a&b, without setting a destination.
- ZF set iff `(a & b) == 0`
- SF set iff `(a & b) < 0`
- CF and OF are set to 0.

How could you use testq to jump if the value in `%rbx` is even?

Setting Condition Codes: Test Instruction

Explicitly set by Test instruction

`testq` Src2, Src1

- Sets condition codes based on value of (Src1 & Src2).
- Often useful to have one of the operands be a mask.
- `testq b, a` is like computing a&b, without setting a destination.
- ZF set iff `(a & b) == 0`
- SF set iff `(a & b) < 0`
- CF and OF are set to 0.

How could you use testq to jump if the value in `%rbx` is even?

```
        testq  $1, %rbx
        je     even
odd:
```

Reading Condition Codes

SetX Instructions: Set low order bytes of destination to 0 or 1, based on combinations of condition codes.

Does not alter remaining 7 bytes.

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)

x86-64 Registers: Least Significant Byte

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

Can reference the low-order byte.

Reading Condition Codes

SetX instructions

- Set single byte based on combinations of conditions codes.

Argument is one of addressable byte registers.

- does not alter remaining bytes;
- typically use `movzbl` to finish the job (will also zero 4 high order bytes).

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

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	return value

```
    cmpq    %rsi, %rdi        # compare x:y
    setg    %al               # Set if >
    movzbl  %al, %eax         # Zero rest of %rax
    retq
```

Jumping

jX Instructions: Jump to different parts of the code depending on condition codes.

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)

Conditional Branch Example (Old Style)

Generation: `gcc -Og -fno-if-conversion control.c`

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

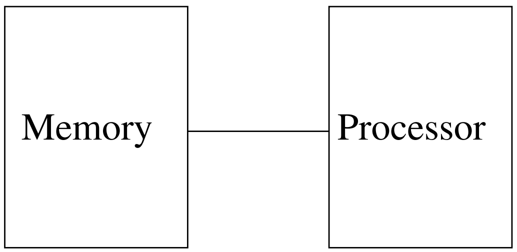
```
absdiff:
   cmpq    %rsi, %rdi    # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   retq
.L4:                    # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   retq
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	return value

Converting C into Assembly

Machine Models

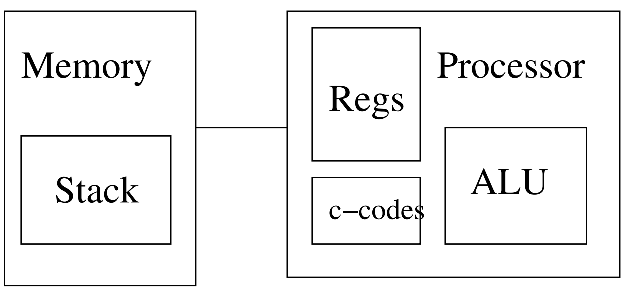
C



Data
1) char
2) int, float
3) double
4) struct, array
5) pointer

Control
1) loops
2) conditionals
3) switch
4) proc. call
5) proc. return

Assembly



1) byte
2) 2-byte word
3) 4-byte long word
4) 8-byte quad word
5) contiguous byte allocation
6) address of initial byte

1) branch/jump
2) call
3) ret

A common compilation strategy is to take a C construct and rewrite it into an equivalent C version that is closer to assembly, as an intermediate step toward assembly.

Expressing with Goto Code

- C allows "goto" as a means of transferring control.
- Jump to position designated by label.
- Generally considered bad coding style in high level language.

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

```c
long absdiff_j (long x,
                long y)
{
  long result;
  int ntest = x <= y;
  if (ntest) goto Else;
  result = x-y;
  goto Done;
Else:
  result = y-x;
Done:
  return result;
}
```

General Conditional Expression Translation

C Code:

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

Example:

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

Goto Version:

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

- Create separate code regions for then and else expressions.
- Execute the appropriate one.

Conditional Move Instructions

- Refer to generically as “cmovXX”
- Based on values of condition codes
- Conditionally copy value from source to destination.
- Can be used to eliminate conditional jump.

Conditional Move Instructions

Inst.	Synonym	Description
cmove	cmovz	Equal / zero
cmovne	cmovnz	Not equal / not zero
cmovs		Negative
cmovns		Not negative
cmovg	cmovnle	Greater (signed)
cmovge	cmovnl	Greater or equal (signed)
cmovl	cmovnge	Less (signed)
cmovle	cmovng	Less or equal (signed)
cmova	cmovnbe	Above (unsigned)
cmovae	cmovnb	Above or equal (unsigned)
cmovb	cmovnae	Below (unsigned)
cmovbe	cmovna	Below or equal (unsigned)

Using Conditional Moves

Conditional Move Instructions

- Instruction supports:
  if (Test) Dest ← Src
- Supported in post-1995 x86 processors
- GCC tries to use them, but only when safe

Why?

- Branches are very disruptive to instruction flow through pipelines.
- 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;
```

Conditional Move Example

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

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	return value

```
absdiff:
   movq    %rdi, %rax     # x
   subq    %rsi, %rax     # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx     # eval = y-x
   cmpq    %rsi, %rdi     # x:y
   cmovle  %rdx, %rax     # if <=, result = eval
   retq
```

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 side effects.

Computations with Side Effects:

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

- Both values get computed
- Must be side effect free

Do-While Loop Example

Following our strategy of rewriting a C construct into a semantically equivalent C version that is closer to assembly.

C Code:

```
long pcount_do
   (unsigned long x) {
   long result = 0;
   do {
      result += x & 0x1;
      x >>= 1;
   } while (x);
   return result;
}
```

Goto Version:

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
loop:
   result += x & 0x1;
   x >>= 1;
   if (x) goto loop;
   return result;
}
```

- Count number of 1's in argument x ("popcount")
- Use conditional branch to either continue looping or to exit loop

Do-While Loop Compilation

Goto Version:

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
loop:
   result += x & 0x1;
   x >>= 1;
   if (x) goto loop;
   return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	return value

```
   movl    $0, %eax              # result = 0
.L2:                             # loop:
   movq    %rdi, %rdx
   andl    $1, %edx              # t = x & 0x1
   addq    %rdx, %rax            # result += t
   shrq    $1, %rdi              # x >>= 1
   jne     .L2                   # if (x) goto loop
   retq
```

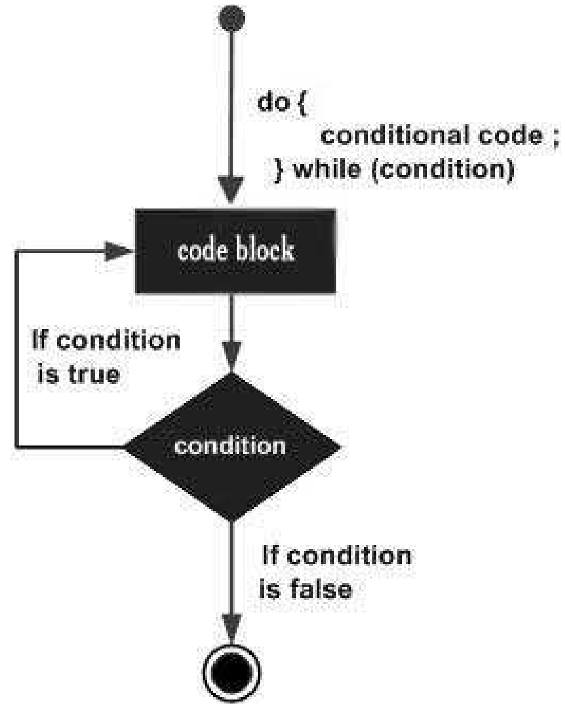
General Do-While Translation

C Code:

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

Goto Version:

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



- Body can be any C statement, typically is a compound statement.
- Test is an expression returning an integer.
  - If it evaluates to 0, that's interpreted as false.
  - If it evaluates to anything but 0, that's interpreted as true.

General While Translation #1

- "Jump-to-middle" translation
- Used with -Og

While version

```
while (Test)
  Body
```

Goto version

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

While Loop Example #1

C Code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
     result += x & 0x1;
     x >>= 1;
  }
  return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
  (unsigned long x) {
  long result = 0;
  goto test;
loop:
  result += x & 0x1;
  x >>= 1;
test:
  if (x) goto loop;
  return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General While Translation

C Code

```
while (Test)
   Body
```

which is equivalent to:

Do-While Version

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

which gets compiled as if it were:

Goto Version

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

Are all three versions semantically equivalent?

While Loop Example #2

C Code

```
long pcount_while
    (unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Goto version

```
long pcount_goto_dw
    (unsigned long x) {
    long result = 0;
    if (!x) goto done;
loop:
    result += x & 0x1;
    x >>= 1;
    if (x) goto loop;
done:
    return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

For Loop Form

General Form

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

```
#define WSIZE 8*sizeof(long)
long pcount_for
    (unsigned long x)
{
    size_t i;
    long result = 0;
    for (i=0; i<WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
    unsigned bit =
        (x >> i) & 0x1;
    result += bit;
}
```

For Loop to While Loop

For version

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

translates to:

While version

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

For-While Conversion Example

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
    unsigned bit =
          (x >> i) & 0x1;
    result += bit;
}
```

```
long pcount_for_while
    (unsigned long x)
{
    size_t i;
    long result = 0;
    i = 0;
    while (i < WSIZE)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
        i++;
    }
    return result;
}
```

For Loop Do-While Conversion

C Code:

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for(i=0; i<WSIZE; i++)
  {
     unsigned bit =
        (x >> i) & 0x1;
     result += bit;
  }
  return result;
}
```

Note that the initial test is not needed. Why?

```
long pcount_for_goto_dw
   (unsigned long x) {
   size_t i;
   long result = 0;
   i = 0;

   if (!(i < WSIZE))   # drop
      goto done;       # drop
loop:
   {
      unsigned bit =
          (x >> i) & 0x1;
      result += bit;
   }
   i++;
   if (i < WSIZE)
      goto loop;
done:
   return result;
}
```

Switch Statement Example

```c
long switch_eq
  (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 (e.g., 5, 6)
- Fall through cases (e.g., 2)
- Missing cases (e.g., 4)

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];
```

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

Switch Example

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

Setup:

```
switch_eq:
    movq     %rdx, %rcx
    cmpq     $6, %rdi           # x:6
    ja       .L8
    jmp      *.L4(, %rdi, 8)
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	return value

Note that `w` is not initialized here.

Switch Statement Example

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

Jump table

```
.section   .rodata
   .align 8
.L4:
   .quad      .L8   # x = 0
   .quad      .L3   # x = 1
   .quad      .L5   # x = 2
   .quad      .L9   # x = 3
   .quad      .L8   # x = 4
   .quad      .L7   # x = 5
   .quad      .L7   # x = 6
```

Setup:

```
switch_eq:
   movq    %rdx, %rcx
   cmpq    $6, %rdi             # x:6
   ja      .L8                  # use default
   jmp     *.L4(, %rdi, 8)      # goto *JTAB[x],
                                # indirect jump
```

Assembly Setup Explanation

Table Structure

- Each target requires 8 bytes
- Base address at .L4

Jumping

- Direct: `jmp .L8`
- Jump target is denoted by label .L8
- Indirect: `jmp *.L4(, %rdi, 8)`
- Start of jump table: .L4
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective address `(.L4 + x*8)`, but only for $0 \leq x \leq 6$

```
.section  .rodata
    .align 8
.L4:
    .quad    .L8   # x = 0
    .quad    .L3   # x = 1
    .quad    .L5   # x = 2
    .quad    .L9   # x = 3
    .quad    .L8   # x = 4
    .quad    .L7   # x = 5
    .quad    .L7   # x = 6
```

Jump Table

Jump Table:

```
.section   .rodata
    .align 8
.L4:
    .quad     .L8   # x = 0
    .quad     .L3   # x = 1
    .quad     .L5   # x = 2
    .quad     .L9   # x = 3
    .quad     .L8   # x = 4
    .quad     .L7   # x = 5
    .quad     .L7   # x = 6
```

```c
long switch_eq
  (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;
}
```

Code Blocks (x == 1)

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

```
.L3:
   movq    %rsi, %rax    # y
   imulq   %rdx, %rax    # y*z
   retq
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	return value

Handling Fall-Through

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

```
case 2:
   w = y/z;
   goto merge;
 . . .
case 3:
   w = 1;
merge:
   w += z;
   retq
```

Code Blocks (x == 2, x == 3)

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

```
.L5:                          # Case 2
    movq   %rsi, %rax
    cqto
    idivq  %rcx               #  y/z
    jmp    .L6                #  goto merge
.L9:                          # Case 3
    movl   $1, %eax           #   w = 1
.L6:                          # merge:
    addq   %rcx, %rax         #   w += z
    retq
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	return value

Code Blocks (x == 5, x == 6, default)

```
switch (x) {
   ...
case 5:          // .L7
case 6:          // .L7
     w -= z;
     break;
default:         // .L8
    w = 2;
}
```

```
.L7:                        # Case 5, 6
    movl   $1, %eax         #   w = 1
    subq   %rdx, %rax       #   w -= z
    retq
.L8:                        # default
    movl   $2, %eax         #   2
    retq
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	return value

Jump Table Structure

Suppose you have a set of switch labels that are “sparse” (widely separated).

In this case, it doesn't make sense to use a jump table.

- If there are only a few labels, simply use a nested if structure.
- If there are many, build a balanced binary search tree.

```
switch(x) {
  case 0:
     Block 0
  case 620:
     Block 620
  ...
  case 1040:
    Block 1040
}
```

The compiler decides the appropriate thresholds for what's “sparse,” what are “a few,” etc.

Summarizing

C Control

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

Assembler Control

- Conditional jump
- Conditional move
- Indirect jump (via jump tables)
- Compiler generates code sequence to implement more complex control

Standard Techniques

- Loops converted to do-while or jump-to-middle form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees