CS107 Lecture 18
Assembly: Control Flow

Reading: B&O 3.6

Control Flow in Assembly

We haven't talked about **`if`**, **`else`**, **`while`**, **`for`**, etc., even though virtually all programs require them. All of them **interrupt** the traditional, sequential execution of code.

At the **machine level**, this becomes **conditional execution**: executing some statements when a condition is **true** and others when it is **false**.

```
if (x > y) {
    // a
} else {
    // b
}
```

One execution flow: Evaluate the **test** and fall through to **a** on the condition that the test passes and jump forward to **b** on the condition that it fails. After executing all of **a**, unconditionally jump around **b**.

```
while (k < n) {
    // c
}
```

One execution flow: Evaluate the **test** and fall through to **c** on the condition that the test passes and jump beyond all of **c** on the condition that the test fails. After executing all of **c**, unconditionally jump back to reevaluate **test**.

How is this implemented in assembly?

Control Flow in Assembly

It takes **multiple instructions** to fully realize the conditional jumps that come with **`if`**, **`while`**, and **`for`** tests.

We typically use a few instructions to **evaluate a test** and one more to **conditionally jump**.

Typical Pattern:

1. **`cmp S1, S2`**
2. **`je [label]`** *or* **`jne [label]`** *or* **`jg [label]`** *or* **`jle [label]`** …

jump if equal

jump if not equal

jump if greater than

jump if less than or equal

Examples:

```
cmp $-5, %rax
jle 0x400472
```

Interpretation: jump to instruction at **`0x400472`** if the **signed value** in **`%rax`** is less than or equal to **`-5`**. Note that **`cmp`** compares the **second argument to the first**. It's important we read **`cmp`** instructions that way if we're to properly interpret how the conditional jump acts on **`cmp`**'s result.

Control Flow in Assembly: Conditional Jumps

Here's the full list of **conditional jump instructions** that branch **if and only if certain conditions** are met. The **label** is encoded into the instruction.

Instruction	**Synonym**	**Condition Tested**
`je` *Label*	`jz`	Equal / zero
`jne` *Label*	`jnz`	Not equal / not zero
`js` *Label*		Negative
`jns` *Label*		Nonnegative
`jg` *Label*	`jnle`	Greater (signed **>**)
`jge` *Label*	`jnl`	Greater or equal (signed **>=**)
`jl` *Label*	`jnge`	Less (signed **<**)
`jle` *Label*	`jng`	Less or equal (signed **<=**)
`ja` *Label*	`jnbe`	Above (unsigned **>**)
`jae` *Label*	`jnb`	Above or equal (unsigned **>=**)
`jb` *Label*	`jnae`	Below (unsigned **<**)
`jbe` *Label*	`jna`	Below or equal (unsigned **<=**)

Control Flow in Assembly: Etudes

It takes **multiple instructions** to fully realize the conditional jumps that come with **`if`**, **`while`**, and **`for`** tests.

Let's study some real conditional jump examples.

```
cmp $0, %eax
jge 0x400300
```

Interpretation: jump to instruction at **`0x400300`** if the **`signed int`** in **`%eax`** is **greater than or equal** to zero.

```
cmp %dx, %bx
je 0x400104
```

Interpretation: jump to instruction at **`0x400104`** if the 16-bit pattern in **`%bx`** **matches** the 16-bit pattern in **`%dx`**.

```
cmp $0x300, %r10
jbe 0x40148
```

Interpretation: jump to instruction at **`0x400148`** if the **`unsigned long`** in **`%r10`** is **less than or equal** to **`0x300`**.

Aside: The **b** in **`jbe`** stands for **below** and assumes the operands are **unsigned** quantities.

```
cmp $0xFFFF, %cx
ja 0x400244
```

Interpretation: jump to instruction at **`0x400244`** if the **`unsigned short`** in **`%cx`** is **greater than** **`0xFFFF`**.

Thought Question: What's odd about this final **`cmp`**?

More Important Thought Question: How can these conditional jumps **know the outcomes of the preceding comparisons**?

Control Flow in Assembly: Condition Codes

The CPU maintains a small collection of **flags**—also called **condition codes**—that automatically **record information about the most recent arithmetic or logical operation**.

- **`cmp`** instructions compare operands via subtraction.
  - Restated, **`cmp S1, S2`** computes **`S2 – S1`**.
  - Rather than storing the result of the subtraction anywhere, it updates the condition codes to summarize key properties of the result (e.g., result was zero, or negative, or overflowed, and so forth).
- Conditional jumps typically follow comparisons and consult the condition codes to decide whether to branch or not.

What are these condition codes? How do they store this information?

Control Flow in Assembly: Condition Codes

The CPU maintains a small collection of **flags**—also called **condition codes**—that automatically **record information about the most recent arithmetic or logical operation**.

Here are the most frequently mentioned condition codes:

• **ZF:** Zero flag.	Indicates the most recent operation yielded a zero.
• **SF:** Sign flag.	Indicates the most recent operation produced a negative value.
• **CF:** Carry flag.	Indicates a carry out of or borrow into the most significant bit and is used to detect overflow in unsigned arithmetic.
• **OF:** Overflow flag.	Indicates signed overflow in two's-complement arithmetic.

Condition Code Etudes

The CPU maintains a small collection of **flags**—also called **condition codes**—that automatically **record information about the most recent arithmetic or logical operation**.

For each of the following 8-bit **`cmp`**s, let's decide which flags are set and understand why.

`cmp $0xFF, %bl`
where `%bl` stores `0x01`

Computed (8-bit): `0x01` – `0xFF` = `0x02`

ZF	SF	CF	OF
0	0	1	0

The result (0x02) is **nonzero** and **positive**, so ZF = 0 and SF = 0. As an unsigned subtraction, 0x01 − 0xFF **requires a borrow**, so CF = 1. As a signed subtraction, this is 1 − (−1) = 2, which **fits in 8 bits**, so OF = 0.

`cmp $0xFF, %bl`
where `%bl` stores `0x80`

Computed (8-bit): `0x80` – `0xFF` = `0x81`

ZF	SF	CF	OF
0	1	1	0

The result (0x81) is **nonzero** and **negative**, so ZF = 0 and SF = 1. As an unsigned subtraction, 0x80 − 0xFF **requires a borrow**, so CF = 1. As a signed subtraction, −128 − (−1) = −127, which **fits in 8 bits**, so OF = 0.

`cmp $0x80, %bl`
where `%bl` stores `0x00`

Computed (8-bit): `0x00` – `0x80` = `0x80`

ZF	SF	CF	OF
0	1	1	1

The result (0x80) is **nonzero** and **negative**, so ZF = 0 and SF = 1. As an unsigned subtraction, 0x00 − 0x80 **requires a borrow**, so CF = 1. As a signed subtraction, 0 − (−128) = 128 but **can't be represented as a signed 8-bit value**, so OF = 1.

Conditional Jump Etudes

`cmp` performs a subtraction **only to set the condition-code flags**, without storing the result anywhere. The conditional jump that follows it **examines those bits** and **decides whether control should transfer elsewhere**.

```
and $0xF0, %rax
cmp $0xC0, %al
je  0x400257
```

The code zeroes out everything sitting in **`%rax`** apart from the high nibble of the lowest byte. It compares that nibble to **`0xC`** and branches if they match. If they match, **`%al - 0xC0`** is zero and **`cmp`** sets ZF to 1.

```
add $1, %ecx
cmp $100, %ecx
jle 0x400410
```

The **add** increments some variable, and **`cmp`** sets flags for **`%ecx`** − **`100`**, and **`jle`** branches if the signed result is less than or equal to zero—that is, when ZF is 1 or SF and OF differ (for example, **`%ecx`** might be 99, or it might be so negative that subtracting 100 causes signed overflow).

```
add %r10, %rdi
cmp %rsi, %rdi
ja  0x4003a0
```

Here **`%rdi`** and **`%rsi`** behave like addresses, where **`%rsi`** serves as an end pointer. After **`%rdi`** is advanced by the offset in **`%r10`**, the **`cmp`** checks whether **`%rdi`** exceeds that end pointer. Since addresses are inherently unsigned, their comparisons are unsigned as well. That's why **`ja`** is used instead of **`jg`**.

Conditional Jumps: Don't Overcomplicate

While it's true that all conditional jumps consult the condition-code flags to decide whether to branch, it's **often easier to eyeball** the preceding **`cmp`** and **articulate what the conditional jump expects from it** if it's to branch.

```
sub $0x10, %rdi
cmp $0x4032, %rdi
jge 0x401930
```

BSEE Explanation: Jump to the instruction as address `0x401930` if the **`cmp`** sets OF and SF to be the same value (ZF doesn't matter).

BSCS Explanation: Jump to the instruction as address `0x401930` if `%rdi` – `0x4032` (as a signed value) is greater than or equal to 0.

Conditional Jumps: Not Always cmp

Very occasionally, the instruction preceding a conditional jump is **an instruction other than `cmp`**. **All** arithmetic and logical operations update condition codes, not just **`cmp`**.

```
and  $0x7, %ax
je   0x400772
```

This jumps if the value in **`%ax`** was a **multiple of 8**. If the low three bits were 000, the result becomes 0, so the jump is taken.

```
sub %esi, %edx
jl   0x500040
```

This jumps if **`%edx`** − **`%esi`** is **negative** (as a signed value). The subtraction stores its result in **`%edx`**, and if that result is negative, the branch is taken.

```
test %rdi, %rdi
jl   0x500040
```

This is clever but needs a bit of explanation: **`test`** performs a **bitwise and** of its two operands and **updates condition codes without storing the result** (just like **`cmp`**).

With **`test %rdi, %rdi`**, **`%rdi`** is **and**'ed with itself and determines whether the value is zero or negative **without modifying** **`%rdi`**. Here, **`jl`** branches if **`%rdi`** holds a negative value. Compilers prefer **`test %rdi, %rdi`** over **`cmp $0, %rdi`** because **it accomplishes the same check but requires fewer bytes to encode**.

Conditional Jumps and `if` Statements

How can we use instructions like **cmp** and conditional jumps to **implement if statements in assembly**? It's all about **conditionally skipping** blocks of code.

Assembly:

```
daisy:
   cmpl $10, %edi
   je .L4
   movl %edi, %eax
   negl %eax
   ret
.L4:
   leal 1(%rdi), %eax
   ret
```

C:

```
int daisy(int x) {
   if (_x_==_10_) {
      ___x++___;
   } else {
      _x_=_-x_;
   }
   return ___x____;
}
```

Key Insights:

- The first **cmp** instruction and the subsequent **`je`** instruction **jointly imply** the **`if`** test is **`x == 10`**. Here the branch is taken when **`x`** is **`10`**, so the fall-through path handles all other values. (**Pro tip**: compilers work to avoid branching, since the hardware is optimized for sequential execution and branching away from that is expensive.)
- The code that executes when **`x`** is 10 loads **`%eax`** with **`x + 1`**, and the code that executes when **`x`** is anything else loads **`%eax`** with **`x`** and then negates it. Since **the same C return statement executes in both scenarios**, it's simplest to just **`return x`** after updating **`x`** itself in the conditionally executed blocks to be either **`x + 1`** or **`-x`** accordingly.

Conditional Jumps and `if` Statements

How can we use instructions like **cmp** and conditional jumps to **implement if statements in assembly**? It's all about **conditionally skipping** blocks of code.

Assembly:

```
rose:
   movq %rdi, %rax
   subq %rsi, %rax
   cmpq %rsi, %rdi
   jge .L5
   movq %rsi, %rax
   subq %rdi, %rax
.L5:
   ret
```

C:

```
long rose(long x, long y) {
   long result = __x - y__;
   if (__x < y__) {
      __result = y - x__;
   }
   return result;
}
```

Key Insights:

- **`result`** is the **only local variable** and what's returned, so it's backed by **`%rax`**, since **that's where return values go**.
- The first two lines copy **x** into **`%rax`** before subtracting **y** from it. By copying into **`%rax`**, the code copies into **`result`**.
- The **cmp** and the subsequent **`jge`** **skip over several lines** when **x** is greater than or equal to **y**. When **x** is less than **y**, **execution falls through** the **`jge`** instruction as if it weren't there. This strongly implies the **`if`** test itself is **x < y**.
- The code under the jurisdiction of the **`if`** test overwrites the previously computed **x – y** with **y – x**.
- The function returns one of two expressions, depending on the test outcome. It basically returns **| x – y |**.

Conditional Jumps and `if` Statements

How can we use instructions like **cmp** and conditional jumps to **implement if statements in assembly**? It's all about **conditionally skipping** blocks of code.

Assembly:

```
rose:
   movq %rdi, %rax
   subq %rsi, %rax
   cmpq %rsi, %rdi
   jge .L5
   movq %rsi, %rax
   subq %rdi, %rax
.L5:
   ret
```

C:

```
long rose(long x, long y) {
   long result = __x - y__;
   if (__x < y__) {
      _result = y - x_;
   }
   return result;
}
```

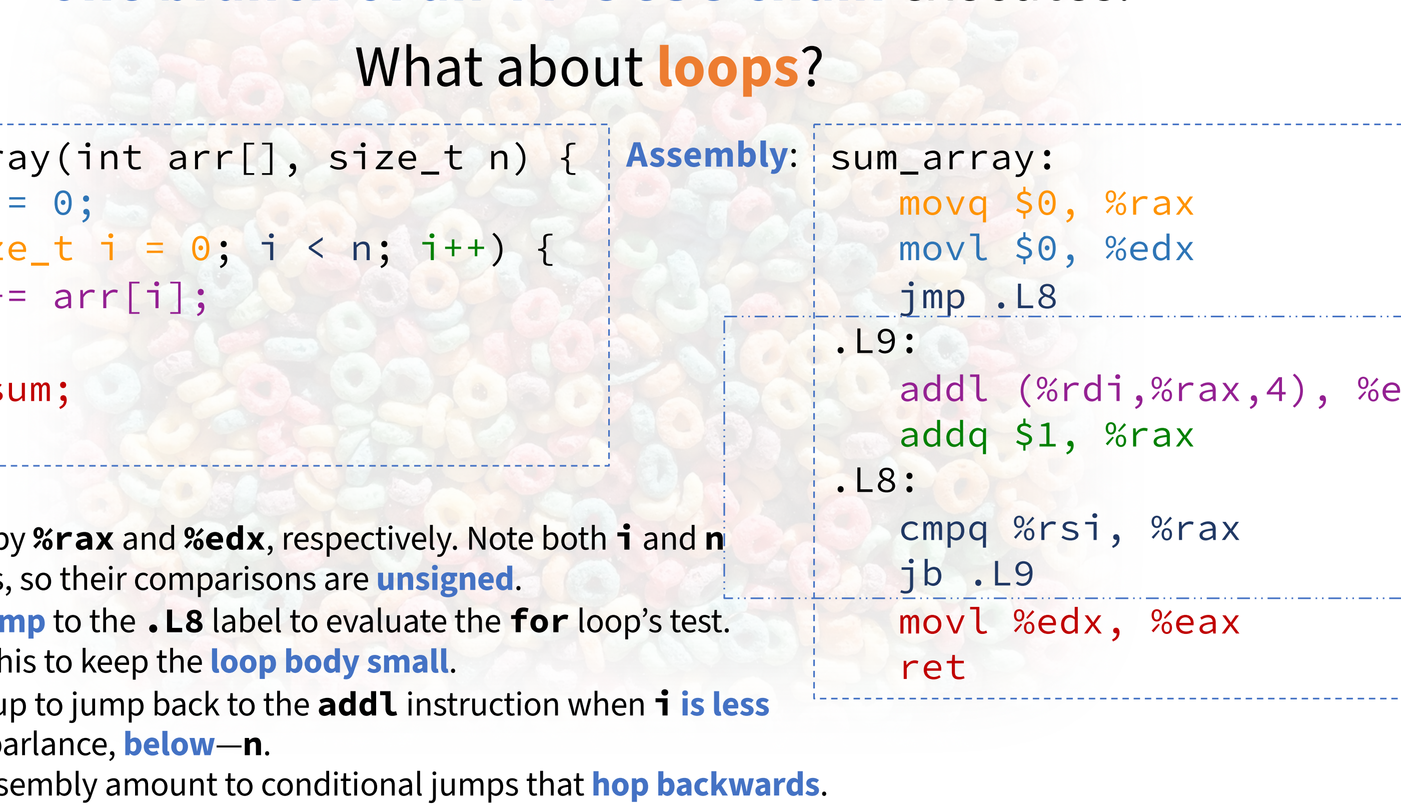
Key Insights:

- **`result`** is the **only local variable** and what's returned, so it's backed by **`%rax`**, since **that's where return values go**.
- The first two lines copy **x** into **`%rax`** before subtracting **y** from it. By copying into **`%rax`**, the code copies into **`result`**.
- The **cmp** and the subsequent **`jge`** **skip over several lines** when **x** is greater than or equal to **y**. When **x** is less than **y**, **execution falls through** the **`jge`** instruction as if it weren't there. This strongly implies the **`if`** test itself is **x < y**.
- The code under the jurisdiction of the **`if`** test **overwrites** the previously computed **x – y** with **y – x**.
- The function returns one of two expressions, depending on the outcome of a test. It basically returns **| x – y |**.

Conditional Jumps and Loops

We've seen **`cmp`**s and conditional jumps collaborate to ensure that **at most one branch of an `if`-`else` chain** executes.

What about **loops**?



C:

```
int sum_array(int arr[], size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Assembly:

```
sum_array:
    movq $0, %rax
    movl $0, %edx
    jmp .L8
.L9:
    addl (%rdi,%rax,4), %edx
    addq $1, %rax
.L8:
    cmpq %rsi, %rax
    jb .L9
    movl %edx, %eax
    ret
```

loop

Key Insights:

- **`i`** and **`sum`** are backed by **`%rax`** and **`%edx`**, respectively. Note both **`i`** and **`n`** (in **`%rsi`**) are **`size_t`**s, so their comparisons are **unsigned**.
- Note the **immediate jump** to the **`.L8`** label to evaluate the **`for`** loop's test. **Pro tip**: Compilers do this to keep the **loop body small**.
- The **`cmp`** and **`jb`** team up to jump back to the **`addl`** instruction when **`i`** **is less than**—or, in unsigned parlance, **below**—**`n`**.
- **Takeaway**: Loops in assembly amount to conditional jumps that **hop backwards**.

Dynamic Instruction Counts

Let's expand on that "Compilers do this to keep the **loop body small**." **pro tip** from the prior slide. Why **emit code that lunges forward** to evaluate a **`for`** loop's test for the very first time?

Here's the assembly taken from the last page. Here we're calling out the **unconditional jump forward** to the code at L8.

The code **conditionally jumps** back to execute the body when **`i`** is less than **`n`**.

```
sum_array:
    movq $0, %rax
    movl $0, %edx
    jmp .L8
.L9:
    addl (%rdi,%rax,4), %edx
    addq $1, %rax
.L8:
    cmpq %rsi, %rax
    jb .L9
    movl %edx, %eax
    ret
```

```
sum_array:
    movq $0, %rax
    movl $0, %edx
.L11:
    cmpq %rsi, %rax
    jae .L10
    addl (%rdi,%rax,4), %edx
    addq $1, %rax
    jmp .L11
.L10
    movl %edx, %eax
    ret
```

Here's some hand-written assembly that could have been emitted instead. This version **unconditionally jumps backward** after the body of the **`for`** loop executes.

Each time we reach the top of the loop, **`cmpq`** checks whether **`i`** **>=** **`n`**. If so, execution jumps around the body of the loop.

Key Takeaways:

- Both versions have a **static instruction count of 9**, though static instruction count **doesn't tell the whole story**.
- The assembly on the left runs the loop in **just four instructions per iteration**—**`addl`**, **`addq`**, **`cmpq`**, and **`jb`**. The assembly on the right requires **five instructions** per iteration, because each one includes **an unconditional backward jump** right at the end.
- For larger arrays, that extra instruction per iteration means the version on the right **executes ~25% more instructions**.
    - Restated, the **dynamic instruction counts** are very different, and the left version is therefore **more efficient**.

Conditional `sets`: Answers in a Single Byte

`set` instructions **conditionally set a byte to 0 or 1**.

- **`setg`**, for instance, **sets the destination byte to 1** when the preceding signed comparison indicates the second operand is greater than the first, and **0 otherwise**.
  - The **same set of suffixes you saw with conditional jumps** are all available with **`set`** instructions as well: **`setle`**, **`setn`**, **`seta`**, **`setbe`**, and so forth.
  - The destination can either be a **single-byte subregister** (e.g., **`%bl`**) or a **memory address**. When the destination is a subregister, **only that byte changes** — **the rest of the register is left alone**.

C:

```
bool is_small(unsigned long x) {
   return x < 256;
}
```

Assembly:

```
is_small:
   cmp $255, %rdi
   setbe %al
   movzbl %al, %eax
   ret
```

  - **Pro tip**: Only **`%al`** matters for the one-byte **`bool`** return value, but compilers often follow **`set`** with **`movzbl`** so the whole register **cleanly holds 0 or 1 for whatever runs next**.

Conditional `sets`: Answers in a Single Byte

Instruction	Synonym	Set Condition (1 if true, 0 if false)
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative (regardless of overflow)
`setns D`		Nonnegative (regardless of overflow)
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

Conditional Moves: Changing Your Mind

`cmovcc` instructions **conditionally move** data, from a source to a destination register.

- **`cmovge`** moves data **in precisely the same scenario** that **`jge`** jumps.
- Compilers sometimes use conditional moves to translate **ternaries** into assembly.

C:

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

Assembly:

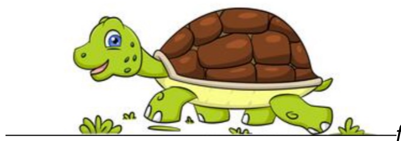
```
max:
    cmp    %edi, %esi
    mov    %edi, %eax
    cmovge %esi, %eax
    retq
```

only four instructions with no branching: often fast

Note the **unconditional move** stages **x** to be returned, as if by default. The **conditional move** decides if the default **wasn't right**.
Thought question: The **cmp** and **cmovge** are separated by an unconditional **mov** instruction. Why does **cmovge** still work?

Aside: This is what the code for **max** might look like if conditional moves didn't exist.

```
max:
    cmp    %edi, %esi
    jge    .L1
    mov    %edi, %eax
    jmp   .L2
.L1:
    mov    %esi, %eax
.L2:
    retq
```



six instruction total, either four or five run, one or two branches: slower

Conditional Moves: Changing Your Mind

Instruction	Synonym	Move Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF = 1)
`cmovne S,R`	`cmovnz`	Not equal / not zero (ZF = 0)
`cmovs S,R`		Negative (SF = 1)
`cmovns S,R`		Nonnegative (SF = 0)
`cmovg S,R`	`cmovnle`	Greater (signed >) (SF = 0 and SF = OF)
`cmovge S,R`	`cmovnl`	Greater or equal (signed >=) (SF = OF)
`cmovl S,R`	`cmovnge`	Less (signed <) (SF != OF)
`cmovle S,R`	`cmovng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`cmova S,R`	`cmovnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`cmovae S,R`	`cmovnb`	Above or equal (unsigned >=) (CF = 0)
`cmovb S,R`	`cmovnae`	Below (unsigned <) (CF = 1)
`cmovbe S,R`	`cmovna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)