CS107, Lecture 18
Assembly: Control Flow

Reading: B&O 3.6

Ed Discussion

Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. to write programs that are more expressive than just one instruction following another.
- This is **conditional execution of statements**: executing statements if some condition is true, executing other statements if that condition is false, etc.
- How is this represented in assembly?

Control

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

In Assembly:

1. **Calculate the condition result**
2. **Based on the result, go to a or b**

Control

- In assembly, it takes more than one instruction to do these two steps.
- Most often: 1 instruction to calculate the condition, 1 to conditionally jump

Common Pattern:

1. **`cmp S1, S2`** // compare two values
2. **je [target]** *or* **jne [target]** *or* **jl [target]** *or* ... // conditionally jump

"jump if equal"

"jump if not equal"

"jump if less than"

Conditional Jumps

There are variants of **jmp** that branch if and only if certain conditions are met. The jump location for these must be hardcoded into the instruction.

Instruction	Synonym	Set Condition
`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

Read **cmp S1, S2** as "compare S2 to S1":

```
// Jump if %edi > 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
cmp $1, %edi
jle [target]
```

Wait a minute – how do jump instructions know anything about the comparisons of earlier instructions?

Control

- The CPU has special registers called *condition codes* that act as "global variables". They automatically track information about the most recent arithmetic or logical operation.
  - **cmp** compares via calculation (subtraction) and info is stored in the condition codes
  - conditional jump instructions look at these condition codes to know whether to jump
- What exactly are the condition codes? How do they store this information?

Condition Codes

Alongside normal registers, the CPU also has single-bit **condition code** registers. They store information about the most recent arithmetic or logical operation.

Most common condition codes:

- **CF:** Carry flag. The most recent operation generated a carry beyond the most significant bit. Used to detect overflow for unsigned operations.
- **ZF:** Zero flag. The most recent operation yielded a zero.
- **SF:** Sign flag. The most recent operation produced a negative value.
- **OF:** Overflow flag. The most recent operation prompted a two's-complement overflow or underflow.

Setting Condition Codes

The **cmp** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (**Note** the operand order!)

```
CMP S1, S2          S2 – S1
```

Instruction	Description
`cmpb`	Compare byte
`cmpw`	Compare word
`cmpl`	Compare double word
`cmpq`	Compare quad word

Control

Read **cmp S1,S2** as "compare S2 to S1". It calculates S2 – S1 and updates the condition codes with the result.

```
// Jump if %edi > 2
// calculates %edi - 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi == 4
// calculates %edi - 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi != 3
// calculates %edi - 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi <= 1
// calculates %edi - 1
cmp $1, %edi
jle [target]
```

How to remember cmp/jmp

- `CMP S1, S2` is `S2 – S1` (just sets condition codes). **But generally**:

```
cmp S1, S2
jg  …
```

Conditional Jumps

Conditional jumps look at a relevant subset of the condition codes to determine whether to branch or fall through without jumping.

Instruction	Synonym	Set Condition
`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 <=)

Setting Condition Codes

The **test** instruction is like **cmp**, but for AND. It does not store the & result anywhere. It just sets condition codes.

```
TEST S1, S2          S2 & S1
```

Instruction	Description
`testb`	Test byte
`testw`	Test word
`testl`	Test double word
`testq`	Test quad word

Cool trick: if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!

The `test` Instruction

- `TEST S1, S2` is `S2 & S1`

```
test %edi, %edi
jns  …
```


`%edi & %edi is nonnegative`

`%edi is nonnegative`

Condition Codes

- Previously discussed arithmetic and logical instructions update these flags. **lea** does not (it's intended only for address computation and nothing else).
- Logical operations (**xor**, etc.) set carry and overflow flags to zero.
- Shift operations set the carry flag to the last bit shifted out and set the overflow flag to zero.
- For more complicated reasons, **inc** and **dec** set the overflow and zero flags, but leave the carry flag unchanged.

Exercise 1: Conditional jump

```
je target          jump if ZF is 1
```

Let %edi store 0x10. Will we jump in the following cases?

%edi | 0x10

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

Exercise 1: Conditional jump

```
je target          jump if ZF is 1
```

Let %edi store 0x10. Will we jump in the following cases?

%edi | 0x10

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```
S2 - S1 == 0, so jump

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```
S2 & S1 != 0, so don't jump

If Statements

How can we use instructions like **cmp** and conditional jumps to implement **`if`** statements in assembly?

Practice: Fill In The Blank

```
int if_then(int param1) {
  if ( ___________ ) {
      ___________;
  }

  return ___________;
}
```

```
0000000000401126 <if_then>:
  401126:   cmp    $0x6,%edi
  401129:   je     40112f
  40112b:   lea    (%rdi,%rdi,1), %eax
  40112e:   retq
  40112f:   add    $0x1,%edi
  401132:   jmp    40112b
```

Practice: Fill In The Blank

```
int if_then(int param1) {
  if ( param1 == 6 ) {
       param1++;
  }

  return param1 * 2;
}
```

```
0000000000401126 <if_then>:
  401126:   cmp    $0x6,%edi
  401129:   je     40112f
  40112b:   lea    (%rdi,%rdi,1), %eax
  40112e:   retq
  40112f:   add    $0x1,%edi
  401132:   jmp    40112b
```

Practice: Fill in the Blank

If-Else In C

```
long absdiff(long x, long y) {
    long result;
    if (________) {

        ________________ ;
    } else {

        ________________ ;
    }

    return result;
}
```

```
401134 <+0>:  mov    %rsi,%rax
401137 <+3>:  cmp    %rsi,%rdi
40113a <+6>:  jge    0x401140 <absdiff+12>
40113c <+8>:  sub    %rdi,%rax
40113f <+11>: retq
401140 <+12>: sub    %rsi,%rdi
401143 <+15>: mov    %rdi,%rax
401146 <+18>: retq
```

If-Else In Assembly pseudocode

Check ***opposite of code condition***
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body

Practice: Fill in the Blank

If-Else In C

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

    return result;
}
```

```
401134 <+0>:  mov    %rsi,%rax
401137 <+3>:  cmp    %rsi,%rdi
40113a <+6>:  jge    0x401140 <absdiff+12>
40113c <+8>:  sub    %rdi,%rax
40113f <+11>: retq
401140 <+12>: sub    %rsi,%rdi
401143 <+15>: mov    %rdi,%rax
401146 <+18>: retq
```

If-Else In Assembly pseudocode

Check ***opposite of code condition***
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body

If-Else Construction Variations

C Code

```
int test(int arg) {
    int ret;
    if (arg > 3) {
        ret = 10;
    } else {
        ret = 0;
    }

    ret++;
    return ret;
}
```

Assembly

```
401134 <+0>:   cmp    $0x3,%edi
401137 <+3>:   jle    0x401142 <test+14>
401139 <+5>:   mov    $0xa,%eax
40113e <+10>:  add    $0x1,%eax
401141 <+13>:  retq
401142 <+14>:  mov    $0x0,%eax
401147 <+19>:  jmp    0x40113e <test+10>
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Set %eax (i) to 0.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 0 – 99 = -99, so it sets the Sign Flag to 1.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

jg means "jump if greater than". This jumps if %eax > 0x63. The flags indicate this is false, so we do not jump.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Add 1 to %eax (i).

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Jump to another instruction.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 1 – 99 = -98, so it sets the Sign Flag to 1.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

We continue in this pattern until we make this conditional jump. When will that be?

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

We will stop looping when this comparison says that %eax > 0x63!

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

Then, we return from the function.

GCC Common While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

```
Check opposite of code condition
Skip loop if test passes
Body
Jump back to test
```

From Previous Slide:

```
0x000000000040115c <+0>:    mov    $0x0,%eax
0x0000000000401161 <+5>:    cmp    $0x63,%eax
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>
0x0000000000401166 <+10>:   add    $0x1,%eax
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>
0x000000000040116b <+15>:   retq
```

GCC Other While Loop Construction

C

```
while (test) {
      body
}
```

Assembly

```
Jump to check
Body
Check code condition
Jump to body if test passes
```

From Previous Slide:

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if passes
Body
Update
Jump to test
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

Which instructions are better when n = 0? n = 1000?

for (int i = 0; i < n; i++)

Optimizing Instruction Counts

- Both versions have the same **static** **instruction count** (# of written instructions).
- But they have different **dynamic** **instruction counts** (# of executed instructions when program is run).
  - If n = 0, left (GCC common output) is best b/c fewer instructions executed
  - If n is large, right (alternative) is best b/c fewer instructions executed
- The compiler may emit a static instruction count that is longer than some alternative, but it may be more efficient if loop executes many times.
- Does the compiler *know* that a loop will execute many times? Of course not.
- What if our code has loops that always execute a small number of times? How do we know when gcc makes a bad decision?
  - (take EE108 and EE180!)

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

- **jmp** instructions conditionally jump to a different next instruction
- **set** instructions conditionally set a byte to 0 or 1
- new versions of **mov** instructions conditionally move data

`set`: Read condition codes

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

- Reads current state of flags
- Destination is a single-byte register (e.g., %al) or single-byte memory location
- Leaves other bytes of register (e.g., everything else in %rax) alone
- Typically followed by movzbl to zero those other bytes

```
int small(int x) {
    return x < 16;
}
```

```
cmp $0xf,%edi
setle %al
movzbl %al, %eax
retq
```

`set`: Read condition codes

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
`setns D`		Nonnegative
`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 <=)

`cmov`: Conditional move

`cmovx src, dst` conditionally moves data in `src` to data in `dst`.

- Mov `src` to `dst` if condition holds; no change otherwise
- `src` is memory address/register, `dst` is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: `result = test ? then: else;`

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

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

cmov: Conditional move

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)