CS 261
Fall 2016

Mike Lam, Professor

x86-64 Control Flow

Topics

- Condition codes
- Jumps
- Conditional moves
- Jump tables

Motivation

- Can we translate the following C function to assembly, using only data movement and arithmetic operations?
  - Fundamental requirement: ability to control the flow of program execution based on the result of evaluating expressions

```
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

Control flow

- The program counter (`%rip`) tracks the address of the next instruction to be executed
  - To implement structured code in assembly, we can "jump" to another location (which changes the PC)
  - In assembly, the target of a jump is usually a label, which is converted to an address by the assembler
  - Unconditional jumps, however, aren't terribly useful by themselves
- Conditional jumps only change the PC if certain condition codes are set
  - Fundamental computation primitive (more about this in CS 430)
  - In C code: "**`if`** `<cond>` **`goto`** `<label>`"
  - C "goto code": code that uses only **`if`**/**`goto`** and **`goto`**

Condition codes

- Special `%flags` register stores individual bits for the following condition codes
  - **CF** (carry): last operation resulted in a carry out
  - **ZF** (zero): last operation yielded zero
  - **SF** (sign): last operation yielded a negative value
  - **OF** (overflow): last operation caused a two's complement overflow (negative or positive)
- Special `cmp` and `test` instructions
  - `cmp` used to compare two values (computes arg_2 - arg_1)
    - NOTE REVERSED ORDERING – also, the result is not saved
  - `test` used to check bits (computes arg_1 & arg_2)
    - Often, the arguments are the same (or one is a bit mask)

Compiler transformations

- Block-structured code to linear code:

C code:

```
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

C goto code:

```
int min (int x, int y)
{
    if (x >= y)
        goto L3;
    return x;
L3:
    return y;
}
```

Compiler transformations

- Block-structured code to linear code:

C code:

```
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:
(x in %edi, y in %esi)

```
min:
    cmpl  %esi, %edi
    jge   .L3
    movl  %edi, %eax
    ret
.L3:
    movl  %esi, %eax
    ret
```

Compiler transformations

- Block-structured code to linear code:

C code:

```
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:
(x in %edi, y in %esi)

```
min:          y       x
    cmpl  %esi, %edi
    jge   .L3
    movl  %edi, %eax
    ret
.L3:
    movl  %esi, %eax
    ret
```

Compiler transformations

- Block-structured code to linear code:

C code:

```
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:
(x in %edi, y in %esi)

```
min:
    cmpl  %esi, %edi
    jge   .L3
    movl  %edi, %eax
    ret
.L3:
    movl  %esi, %eax
    ret
```

Compiler transformations

- Block-structured code to linear code:

C code:

```
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:
(x in %edi, y in %esi)

```
min:
    cmpl  %esi, %edi
    jge   .L3
    movl  %edi, %eax
    ret
.L3:
    movl  %esi, %eax
    ret
```

Conditionals (in goto code)

```
if (<test-expr>)
    <true-branch>
else
    <false-branch>
```

```
  if (!<test-expr>)
    goto false;
  <true-branch>
  goto done;
false:
  <false-branch>
done:
```

If/else

```
if (<top-expr>)
    if (<test-expr>)
        <true-branch>
    else
        <false-branch>
else
    <else-branch>
```

```
  if (!<top-expr>)
    goto else;
  if (!<test-expr>)
    goto false;
  <true-branch>
  goto done;
false:
  <false-branch>
done:
  goto end;
else:
  <else-branch>
end:
```

Nested if/else

Conditionals (in goto code)

```
if (<test-expr>)
    <true-branch>
else
    <false-branch>
```

```
  if (!<test-expr>)
    goto false;
  <true-branch>
  goto done;
false:
  <false-branch>
done:
```

If/else

```
if (<top-expr>)
    if (<test-expr>)
        <true-branch>
    else
        <false-branch>
else
    <else-branch>
```

```
  if (!<top-expr>)
    goto else;
  if (!<test-expr>)
    goto false;
  <true-branch>
  goto done;
false:
  <false-branch>
done:
  goto end;
else:
  <else-branch>
end:
```

Nested if/else

Jump instructions

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Type difference

Figure 3.15 **The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Accessing condition codes

Instruction	Synonym	Effect	Set condition
`sete D`	`setz`	`D ← ZF`	Equal / zero
`setne D`	`setnz`	`D ← ~ZF`	Not equal / not zero
`sets D`		`D ← SF`	Negative
`setns D`		`D ← ~SF`	Nonnegative
`setg D`	`setnle`	`D ← ~(SF ^ OF) & ~ZF`	Greater (signed >)
`setge D`	`setnl`	`D ← ~(SF ^ OF)`	Greater or equal (signed >=)
`setl D`	`setnge`	`D ← SF ^ OF`	Less (signed <)
`setle D`	`setng`	`D ← (SF ^ OF) | ZF`	Less or equal (signed <=)
`seta D`	`setnbe`	`D ← ~CF & ~ZF`	Above (unsigned >)
`setae D`	`setnb`	`D ← ~CF`	Above or equal (unsigned >=)
`setb D`	`setnae`	`D ← CF`	Below (unsigned <)
`setbe D`	`setna`	`D ← CF | ZF`	Below or equal (unsigned <=)

Figure 3.14 The SET instructions. Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate names for the same machine instruction.

Conditional moves

- Similar to conditional jumps, but they move data if certain condition codes are set
  - Benefit: no branch prediction penalty
  - In C code: "`x = ( <cond> ? <tvalue> : <fvalue>)`"

```
  cmp  %rax, %rbx
  jg L01
  movq %rax, %rcx
  jmp L02
L01:
  movq %rbx, %rcx
L02:
```

```
movq   %rax, %rcx
cmp    %rax, %rbx
cmovg  %rbx, %rcx
```

Loops

- Basic idea: jump back to an earlier label
- Three basic forms:
  - Do-while loops
  - Jump-to-middle loops
  - Guarded-do loops
- Note: we'll use goto code in C instead of assembly
  - Just to avoid unnecessary complication

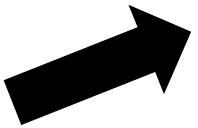
Loops

```
do
    <body-statement>
while (<test-expr>);
```

```
loop:
  <body-statement>
  if (<test-expr>)
    goto loop;
```

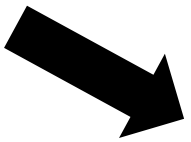
Do-while loop

```
while (<test-expr>)
    <body-statement>
```



```
  goto test;
loop:
  <body-statement>
test:
  if (<test-expr>)
    goto loop
```

Jump-to-middle loop



```
  if (!<test-expr>)
    goto done
loop:
  <body-statement>
  if (<test-expr>)
    goto loop
done:
```

Guarded-do loop

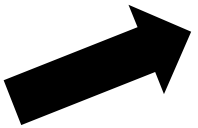
Loops

```
do
    <body-statement>
while (<test-expr>);
```

```
loop:
  <body-statement>
  if (<test-expr>)
    goto loop;
```

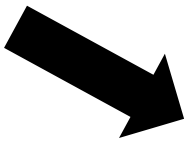
Do-while loop

```
while (<test-expr>)
    <body-statement>
```



```
  goto test;
loop:
  <body-statement>
test:
  if (<test-expr>)
    goto loop
```

Jump-to-middle loop



```
  if (!<test-expr>)
    goto done
loop:
  <body-statement>
  if (<test-expr>)
    goto loop
done:
```

Guarded-do loop

Loops

```
for (<init-expr>; <test-expr>; <update-expr>)
    <body-statement>
```

```
  goto test;
loop:
  <body-statement>
test:
  if (<test-expr>)
    goto loop
```

Jump-to-middle loop

```
  if (!<test-expr>)
    goto done
loop:
  <body-statement>
  if (<test-expr>)
    goto loop
done:
```

Guarded-do loop

Loops

```
for (<init-expr>; <test-expr>; <update-expr>)
    <body-statement>
```

```
  <init-expr>
  goto test;
loop:
  <body-statement>
  <update-expr>
test:
  if (<test-expr>)
    goto loop
```

Jump-to-middle loop

```
  <init-expr>
  if (!<test-expr>)
    goto done;
loop:
  <body-statement>
  <update-expr>
  if (<test-expr>)
    goto loop
done:
```

Guarded-do loop

Loops

```
for (<init-expr>; <test-expr>; <update-expr>)
    <body-statement>
```

```
  <init-expr>
  goto test;
loop:
  <body-statement>
  <update-expr>
test:
  if (<test-expr>)
    goto loop
```

Jump-to-middle loop

```
  <init-expr>
  if (!<test-expr>)
    goto done;
loop:
  <body-statement>
  <update-expr>
  if (<test-expr>)
    goto loop
done:
```

Guarded-do loop

Switch statements

- One approach: convert to if/elseif code
  - Problem: performance varies based on ordering and actual runtime values!

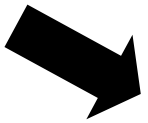
```
switch (x) {
  case 10: do_blah();
           break;
  case 11: do_foo();
           break;
  case 13: do_bar();
           break;
  case 15: do_baz();
           break;
  default: error();
}
```

```
if (x == 10) {
    do_blah();
} else if (x == 11) {
    do_foo();
} else if (x == 13) {
    do_bar();
} else if (x == 15) {
    do_baz();
} else {
    error();
}
```

Switch statements

- Indexed indirect jump ("computed goto")
  - Implemented using a data structure called a jump table
  - Very efficient when # of options is high and the value range is small

```
switch (x) {
  case 10: do_blah();
           break;
  case 11: do_foo();
           break;
  case 13: do_bar();
           break;
  case 15: do_baz();
           break;
  default: error();
}
```



Jump Table

Index	Destination
0	0x400e51
1	0x400e88
2	0x401900
3	0x400f12
4	0x401900
5	0x400f34

x is in %rdx, address of jump table is in %rbx

```
subq $0xA, %rdx
movq (%rbx,%rdx,0x8), %rcx
jmp *%rcx
```

Related coursework

- Intriguing notion: can we always automatically translate from structured code to linear/goto code?
  - Yes, this is what a compiler does!
  - If you're interested in learning more about how this works, plan to take CS 432 as your systems elective