Machine-Level Representation of Programs: Control

Yipeng Huang

Rutgers University

March 26, 2024
Table of contents

Announcements

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

Comparisons and program control flow

  What is control flow?
  Condition codes
  Comparison and set instructions

Modifying control flow via conditional branch statements

  Jump instructions
  Conditional branch statements

Modifying data flow via conditional move statements

Loop statements

  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
Class session plan

▶ Tuesday, 3/26: Control flow (conditionals, if, for, while, do loops, switch statements) in assembly. (Book chapter 3.6). Bomblab phase_2, phase_3.
▶ Tuesday, 4/2: Arrays and data structures in assembly. (Book chapter 3.8). Bomblab phase_5, phase_6.
Table of contents
Announcements
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic
Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements
Modifying data flow via conditional move statements
Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions
Switch statements
Borrowing memory address calculation to efficiently implement arithmetic

Address Computation Instruction

- **leaq Src, Dst**
  - Src is address mode expression
  - Set Dst to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k = 1, 2, 4, or 8`

- **Example**

```c
long m12(long x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```asm
leaq (%rdi,%rdi,2), %rax  # t <- x+x*2
salq $2, %rax             # return t<<2
```
Load effective address

Both C code functions above translate to the assembly on the right.

leaq:
mulAdd:

```
leaq 8(%rdi,%rsi,8), %rax
ret
```

Explanation

- `leaq src, dest` takes the effective address of the memory (index, displacement) expression of src and puts it in dest.
- `leaq` has shorter latency (takes fewer CPU cycles) than `imulq`, so GCC will use `leaq` whenever it can to calculate expressions like $y + ax + b$. 

```
long * leaq (  
    long * ptr, long index  
) {  
    return &ptr[index+1];  
}
```

```
long mulAdd (  
    long base, long index  
) {  
    return base+index*8+8;  
}
```
Table of contents

Announcements

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Modifying data flow via conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
What is control flow?

Control flow is:
- Change in the sequential execution of instructions.
- Change in the steady incrementation of the program counter / instruction pointer (%rip register).

Control primitives in assembly build up to enable C and Java control statements:
- if-else statements
- do-while loops
- while loops
- for loops
- switch statements
Condition codes

Assembly/Machine Code View

Programmer-Visible State
- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

**Memory**
- Byte addressable array
- Code and user data
- Stack to support procedures

Carnegie Mellon
# Condition codes

Automatically set by most arithmetic instructions.

<table>
<thead>
<tr>
<th>Applicable types</th>
<th>Condition code</th>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed and unsigned</td>
<td>ZF</td>
<td>Zero flag</td>
<td>The most recent operation yielded zero.</td>
</tr>
<tr>
<td>Unsigned types</td>
<td>CF</td>
<td>Carry flag</td>
<td>The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations</td>
</tr>
<tr>
<td>Signed types</td>
<td>SF</td>
<td>Sign flag</td>
<td>The most recent operation yielded a negative value.</td>
</tr>
<tr>
<td>Signed types</td>
<td>OF</td>
<td>Overflow flag</td>
<td>The most recent operation yielded a two’s complement positive or negative overflow.</td>
</tr>
</tbody>
</table>

**Table:** Condition codes important for control flow
Comparison instructions

cmpq source1, source2
Performs source2 – source1, and sets the condition codes without setting any destination register.
Test for equality

equal_sl:
   xorl %eax, %eax
   cmpq %rsi, %rdi
   sete %al
   ret

Explanation

➤ xorl %eax, %eax: Zeros the 32-bit register %eax.
➤ cmpq %rsi, %rdi: Calculates %rdi - %rsi (x - y), sets condition codes without updating any destination register.
➤ sete %al: Sets the 8-bit %al subset of %eax if op yielded zero.

C code function above translates to the assembly on the right.
Test if unsigned x is below unsigned y

```
short below_ul (  
    unsigned long x,  
    unsigned long y  
) {  
    return x<y;  
}
```

```
short nae_ul (  
    unsigned long x,  
    unsigned long y  
) {  
    return !(x>=y);  
}
```

Both C code functions above translate to the assembly on the right.

```
below_ul:        
nae_ul:         
    xorl %eax, %eax  
    cmpq %rsi, %rdi  
    setb %al         
    ret
```

Explanation

▶ xorl %eax, %eax: Zeros %eax.
▶ cmpq %rsi, %rdi: Calculates %rdi − %rsi (x − y), sets condition codes without updating any destination register.
▶ setb %al: Sets %al if CF flag set indicating unsigned overflow.
Side review: De Morgan’s laws

- $\neg A \land \neg B \iff \neg (A \lor B)$
- $(\sim A) \& (\sim B) \iff \sim (A \mid B)$
### Set instructions

**cmp source1, source2** performs source2 − source1, sets condition codes.

<table>
<thead>
<tr>
<th>Applicable types</th>
<th>Set instruction</th>
<th>Logical condition</th>
<th>Intuitive condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed and unsigned</td>
<td>sete / setz</td>
<td>ZF</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>Signed and unsigned</td>
<td>setne / setnz</td>
<td>∼ ZF</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>Unsigned</td>
<td>setb / setnae</td>
<td>CF</td>
<td>Below</td>
</tr>
<tr>
<td>Unsigned</td>
<td>setbe / setna</td>
<td>CF</td>
<td>ZF</td>
</tr>
<tr>
<td>Unsigned</td>
<td>seta / setnbe</td>
<td>∼ CF &amp; ∼ ZF</td>
<td>Above</td>
</tr>
<tr>
<td>Unsigned</td>
<td>setnb / setae</td>
<td>∼ CF</td>
<td>Above or equal</td>
</tr>
<tr>
<td>Signed</td>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>Signed</td>
<td>setns</td>
<td>∼ SF</td>
<td>Nongative</td>
</tr>
<tr>
<td>Signed</td>
<td>setl / setnge</td>
<td>SF ^ OF</td>
<td>Less than</td>
</tr>
<tr>
<td>Signed</td>
<td>setle / setng</td>
<td>(SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>Signed</td>
<td>setg / setnle</td>
<td>∼ (SF ^ OF) &amp; ∼ ZF</td>
<td>Greater than</td>
</tr>
<tr>
<td>Signed</td>
<td>setge / setnl</td>
<td>∼ (SF ^ OF)</td>
<td>Greater than or equal</td>
</tr>
</tbody>
</table>

**Table:** Set instructions
Table of contents

Announcements

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Modifying data flow via conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
Jump instructions

Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Branch statements

```c
1 unsigned long absdiff_ternary (  
2 unsigned long x, unsigned long y ){
3     return x<y ? y-x : x-y;
4 }

1 unsigned long absdiff_if_else (  
2 unsigned long x, unsigned long y ){
3     if (x<y) return y-x;
4     else return x-y;
5 }

1 unsigned long absdiff_goto (  
2 unsigned long x, unsigned long y ){
3     if (!(x<y)) goto Else;
4     return y-x;
5     Else:       
6     return x-y;
7 }
```

All C functions above translate (-fno-if-conversion) to assembly at right.

```asm
absdiff_if_else:
    cmpq %rsi, %rdi
    jnb .ELSE
    movq %rsi, %rax
    subq %rdi, %rax
    ret

.ELSE:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

**Explanation**

- `cmpq %rsi, %rdi`: Calculates `%rdi - %rsi` (\(x - y\)), sets condition codes.
- `jnb .ELSE`: Sets program counter / instruction pointer in `%rip` (ELSE) if CF flag not set indicating no unsigned overflow.
Table of contents

Announcements
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Modifying data flow via conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
Conditional move statements

```c
unsigned long absdiff_ternary (unsigned long x, unsigned long y ){
    return x<y ? y-x : x-y;
}
```

```c
unsigned long absdiff_if_else (unsigned long x, unsigned long y ){  
    if (x<y) return y-x;
    else return x-y;
}
```

```c
unsigned long absdiff_goto (unsigned long x, unsigned long y ){  
    if (!(x<y)) goto Else;
    return y-x;
Else:  
    return x-y;
}
```

All C functions above translate (-fif-conversion or -O1) to assembly at right.

absdiff_ternary:
```assembly
movq %rsi, %rdx // y
subq %rdi, %rdx // y-x
movq %rdi, %rax // x
subq %rsi, %rax // x-y
cmpq %rsi, %rdi
cmovb %rdx, %rax
ret
```

absdiff_if_else:
```assembly
movq %rsi, %rdx // y
subq %rdi, %rdx // y-x
movq %rdi, %rax // x
subq %rsi, %rax // x-y
cmpq %rsi, %rdi
cmovb %rdx, %rax
ret
```

absdiff_goto:
```assembly
movq %rsi, %rdx // y
subq %rdi, %rdx // y-x
movq %rdi, %rax // x
subq %rsi, %rax // x-y
cmpq %rsi, %rdi
cmovb %rdx, %rax
ret
```

Explanation

- `cmpq %rsi, %rdi`: Calculates `%rdi - %rsi (x - y), sets condition codes.
- `jnb .ELSE:` Sets program counter / instruction pointer in `%rip (.ELSE) if CF flag not set indicating no unsigned overflow.
Modifying control flow vs. data flow in deep CPU pipelines

Figure: Pipelined CPU stages. Image credit wikimedia
Table of contents

Announcements
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Modifying data flow via conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
Compiling for loops to while loops

C loop statements such as for loops, while loops, and do-while loops do not exist in assembly. They are instead constructed from conditional jump statements.

```c
unsigned long count_bits_for (
    unsigned long number
) {
    unsigned long tally = 0;
    for (int shift=0; // init
         shift<8*sizeof(unsigned long); // ← test
         shift++ // update
    ) {
        // body
        tally += 0b1 & number>>shift;
    }
    return tally;
}

unsigned long count_bits_while (
    unsigned long number
) {
    unsigned long tally = 0;
    int shift=0; // init
    while (shift<8*sizeof(unsigned long) // ← test
           shift++ // update
    ) {
        // body
        tally += 0b1 & number>>shift;
    }
    return tally;
}
```
Compiling while loops to do-while loops

```c
unsigned long count_bits_while (unsigned long number)
    { unsigned long tally = 0;
      int shift=0; // init
      while (shift<8*sizeof(unsigned long) // ← test
                tally += 0b1 & number>>shift;
            shift++; // update
      } return tally;
    }

unsigned long count_bits_do_while (unsigned long number)
    { unsigned long tally = 0;
      int shift=0; // init
      do {
        // body
        tally += 0b1 & number>>shift;
        shift++; // update
      } while (shift<8*sizeof(unsigned long← test
                )); // test
      return tally;
    }
```

If initial iteration is guaranteed to run, then do one fewer test.
Compiling do-while loops to goto statements

Loops get compiled into goto statements which are readily translated to assembly.
Compiling goto statements to assembly conditional jump instructions

```c
unsigned long count_bits_goto (unsigned long number) {
    unsigned long tally = 0;
    int shift=0; // init
    LOOP:
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
    if (shift<8*sizeof(unsigned long)) {
        goto LOOP;
    }
    return tally;
}
```

All C loop statements so far translate to assembly at right.

count_bits_for:
count_bits_while:
count_bits_do_while:
count_bits_goto:

```
xorl %ecx, %ecx # int shift=0; // init
xorl %eax, %eax # unsigned long tally = 0;

.Loop:
movq %rdi, %rdx # number
shrq %cl, %rdx # number>>shift
incl %ecx # shift++; // update
andl $1, %edx. # 0b1 & number>>shift
addq %rdx, %rax # tally += 0b1 & number>>shift
```

cmpl $64, %ecx # shift<8*sizeof(unsigned long)
jne .LOOP # goto LOOP;
ret # return tally;
```
Table of contents
Announcements
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic
Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements
Modifying data flow via conditional move statements
Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions
Switch statements