Machine-Level Representation of Programs: Control

Yipeng Huang

Rutgers University

March 26, 2024

Table of contents

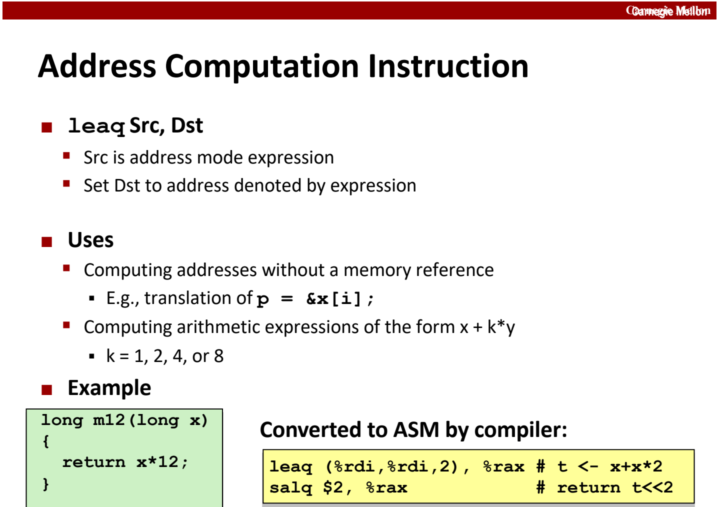
Announcements

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.
- Thursday, 3/28: Function calls in assembly. (Book chapter 3.7). Bomblab phase_4.
- Tuesday, 4/2: Arrays and data structures in assembly. (Book chapter 3.8). Bomblab phase_5, phase_6.

Table of contents

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



Carnegie Mellon

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**

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

Converted to ASM by compiler:

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

Example: 3_leaq.c

Load effective address

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

```c
long mulAdd (
    long base, long index
) {
    return base+index*8+8;
}
```

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 $y + ax + b$.

Table of contents

What is control flow?

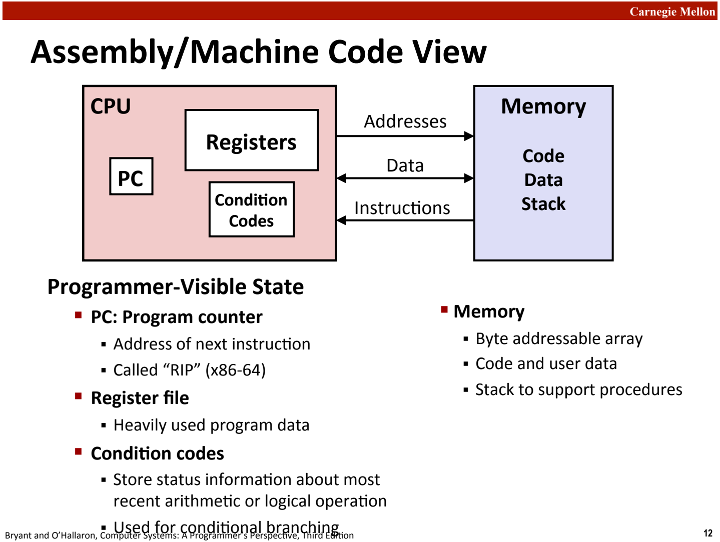
Control flow is:

- 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



Carnegie Mellon

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

Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition

12

Condition codes

Automatically set by most arithmetic instructions.

Applicable types	Condition code	Name	Use
Signed and unsigned	`ZF`	Zero flag	The most recent operation yielded zero.
Unsigned types	`CF`	Carry flag	The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations
Signed types	`SF`	Sign flag	The most recent operation yielded a negative value.
Signed types	`OF`	Overflow flag	The most recent operation yielded a two's complement positive or negative overflow.

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

```
short equal_sl (
    long x,
    long y
) {
    return x==y;
}
```

C code function above translates to the assembly on the right.

```
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.

Test if unsigned x is below unsigned y

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

```c
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

- $\neg A \land \neg B \iff \neg(A \lor B)$
- $(\sim A)\&(\sim B) \iff \sim (A|B)$

Set instructions

`cmp source1, source2` performs source2 – source1, sets condition codes.

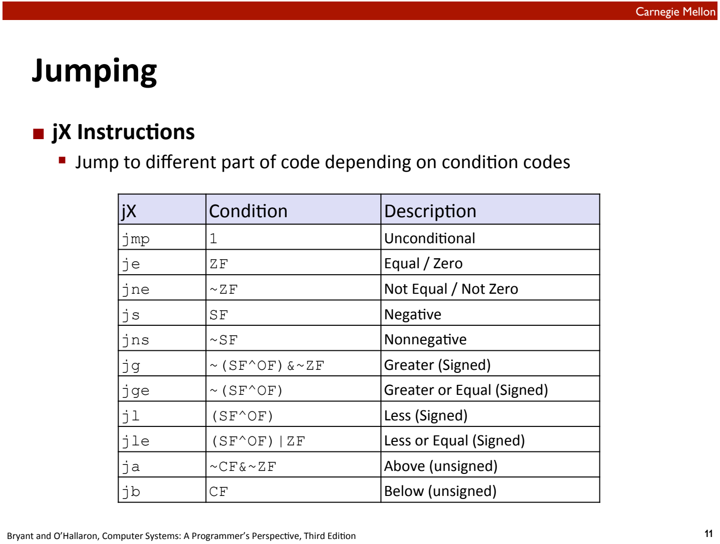
Applicable types	Set instruction	Logical condition	Intutive condition
Signed and unsigned	`sete` / `setz`	$\mathtt{ZF}$	Equal / zero
Signed and unsigned	`setne` / `setnz`	$\sim \mathtt{ZF}$	Not equal / not zero
Unsigned	`setb` / `setnae`	$\mathtt{CF}$	Below
Unsigned	`setbe` / `setna`	$\mathtt{CF} \vert \mathtt{ZF}$	Below or equal
Unsigned	`seta` / `setnbe`	$\sim \mathtt{CF} \& \sim \mathtt{ZF}$	Above
Unsigned	`setnb` / `setae`	$\sim \mathtt{CF}$	Above or equal
Signed	`sets`	$\mathtt{SF}$	Negative
Signed	`setns`	$\sim \mathtt{SF}$	Nonegative
Signed	`setl` / `setnge`	$\mathtt{SF}\ \hat{}\ \mathtt{OF}$	Less than
Signed	`setle` / `setng`	$(\mathtt{SF}\ \hat{}\ \mathtt{OF}) \vert \mathtt{ZF}$	Less than or equal
Signed	`setg` / `setnle`	$\sim (\mathtt{SF}\ \hat{}\ \mathtt{OF}) \& \sim \mathtt{ZF}$	Greater than
Signed	`setge` / `setnl`	$\sim (\mathtt{SF}\ \hat{}\ \mathtt{OF})$	Greater than or equal

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

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)

Branch 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 (-fno-if-conversion) to assembly at right.

```
absdiff_if_else:
absdiff_goto:
    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

Conditional move statements

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

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

```
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

```
absdiff_ternary:
absdiff_if_else:
absdiff_goto:
    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

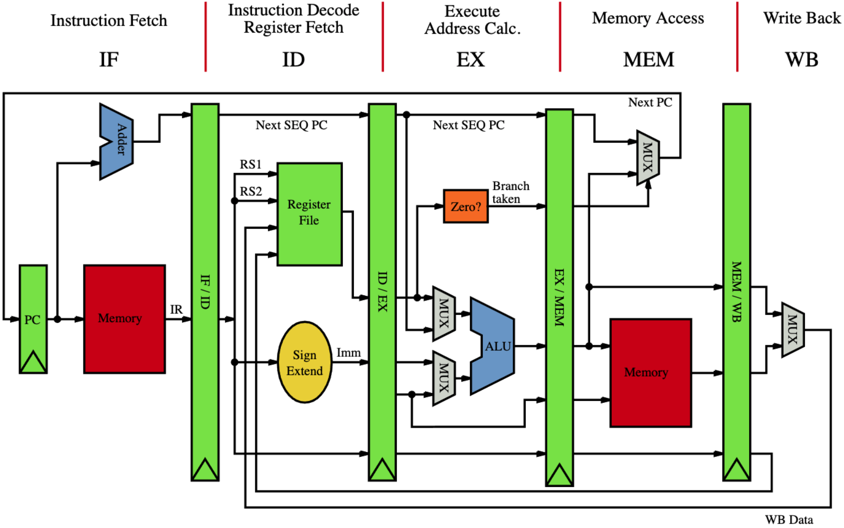


Figure: Pipelined CPU stages. Image credit wikimedia

Table of contents

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;
}
```

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

Compiling while loops to do-while loops

```
unsigned long count_bits_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  while (
    shift<8*sizeof(unsigned long) // ↩
        test
  ) {
    // body
    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
  return tally;
}
```

If initial iteration is guaranteed to run, then do one fewer test.

Compiling do-while loops to goto statements

```
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
  return tally;
}
```

```
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)) { ↩
        // test
    goto LOOP;
  }
  return tally;
}
```

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)) { // test
    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>>shi
  cmpl $64, %ecx  # shift<8*sizeof(unsigned lo
  jne .LOOP       # goto LOOP;
  ret             # return tally;
```

Table of contents