Discussion – Week 2

TA : Mathanky
Email : mathanky04@ucla.edu
Office Hours : 12.30PM - 2.30PM, Boelter Hall 3256F

DATA LAB PRACTICE PROBLEM

Write a function that, given a number n, returns another number where the k^{th} bit from the right is set to to 0.

Examples:

killKthBit(37, 3) = 33 because $37_{10} = 100\textbf{1}01_2 \rightsquigarrow 100\textbf{0}01_2 = 33_{10}$

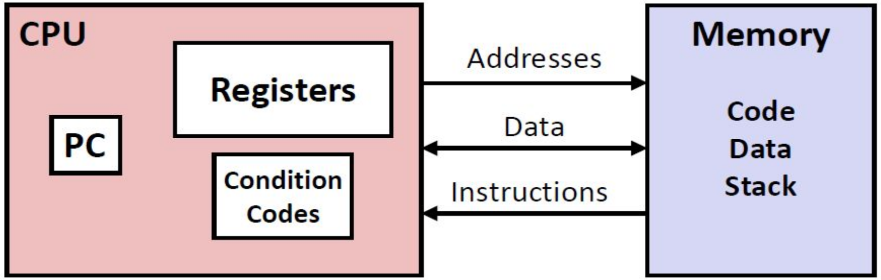
killKthBit(37, 4) = 37 because the 4^{th} bit from the right is already 0.

REVIEW OF WEEK 2

- **Machine Level Programming - Basics**
  - Registers and Memory
  - Arithmetic and Logical Operations
  - Assembly Code
- **Machine Level Programming - Control**
  - Conditional Codes
  - Branches
  - Loops
  - Switch Statements

Machine Level Programming – Basics

Assembly/Machine Code View



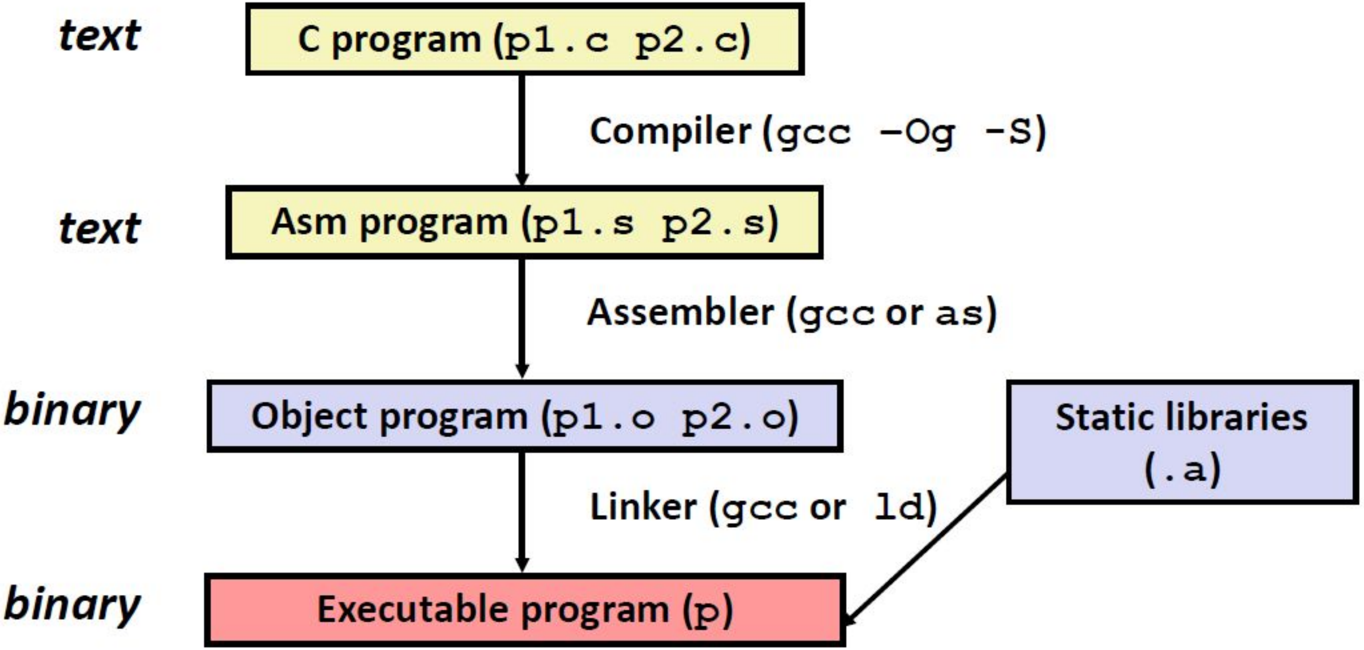
Programmer-Visible State

- **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

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc –Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`



Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- (SIMD vector data types of 8, 16, 32 or 64 bytes)

- Code: Byte sequences encoding series of instructions

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Simple Memory Addressing Modes

- **Normal** **(R)** **Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement** **D(R)** **Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Complete Memory Addressing Modes

- **Most General Form**

  D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]]**
D(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]+D]**
(Rb,Ri,S)	**Mem[Reg[Rb]+S*Reg[Ri]]**

Example of Simple Addressing Modes

```c
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Address Computation Instruction

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

- **Example**

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

Converted to ASM by compiler:

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

mov vs lea

movl (%rdx), %rax
leal (%rdx), %rax

movl takes the **contents** of what’s stored in register %rdx and moves it to %rax.

leal computes the load effective **address** and stores it in %rax. leal analogous to returning a pointer, whereas movl is analogous to returning a dereferenced pointer.

`addq`	*Src,Dest*	Dest = Dest + Src
`subq`	*Src,Dest*	Dest = Dest – Src
`imulq`	*Src,Dest*	Dest = Dest * Src
`salq`	*Src,Dest*	Dest = Dest << Src
`sarq`	*Src,Dest*	Dest = Dest >> Src
`shrq`	*Src,Dest*	Dest = Dest >> Src
`xorq`	*Src,Dest*	Dest = Dest ^ Src
`andq`	*Src,Dest*	Dest = Dest & Src
`orq`	*Src,Dest*	Dest = Dest | Src

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Interesting Instructions

- `leaq`: address computation
- `salq`: shift
- `imulq`: multiplication
  - But, only used once

Machine Level Programming – Control

Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  - **ZF** Zero Flag **OF** Overflow Flag (for signed)

- **Implicitly set (as side effect) of arithmetic operations**

  Example: `addq` *Src,Dest* ↔ `t = a+b`

  CF set if carry/borrow out from most significant bit (unsigned overflow)

  ZF set if `t == 0`

  SF set if `t < 0` (as signed)

  OF set if two's-complement (signed) overflow

  ```
  (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
  ```

- **Not set by `leaq` instruction**

ZF set when

000000000000…00000000000

SF set when

```
  yxxxxxxxxxxxx...
+ yxxxxxxxxxxxx...
------------------
  1xxxxxxxxxxxx...
```

CF set when

```
   1xxxxxxxxxxxxx...
+  1xxxxxxxxxxxxx...
--------------------
1  xxxxxxxxxxxxxx...
```

Carry

```
1  0xxxxxxxxxxxxx...
-  1xxxxxxxxxxxxx...
--------------------
   1xxxxxxxxxxxxx...
```

Borrow

For unsigned arithmetic, this reports overflow

OF set when

```
  yxxxxxxxxxxxx...    a
+ yxxxxxxxxxxxx...    b
--------------------
  zxxxxxxxxxxxx...    t
```

$$z = \sim y$$

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

For signed arithmetic, this reports overflow

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpq` *Src2*, *Src1*
  - `cmpq b,a` like computing `a-b` without setting destination

  - **CF set** if carry/borrow out from most significant bit (used for unsigned comparisons)
  - **ZF set** if `a == b`
  - **SF set** if `(a-b) < 0` (as signed)
  - **OF set** if two's-complement (signed) overflow
    `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Condition Codes (Explicit Reading: Set)

- **Explicit Reading by Set Instructions**
  - **`setX`** *Dest*: Set low-order byte of destination *Dest* to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes of *Dest*

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (signed)
`setge`	`~(SF^OF)`	Greater or Equal (signed)
`setl`	`SF^OF`	Less (signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use **`movzbl`** to finish job
    - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
    cmpq   %rsi, %rdi   # Compare x:y
    setg   %al          # Set when >
    movzbl %al, %eax    # Zero rest of %rax
    ret
```

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes
  - Implicit reading of 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)

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

```
absdiff:
   cmpq    %rsi, %rdi  # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:       # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rax`	Return value

Conditional Move Example

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

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
absdiff:
   movq     %rdi, %rax   # x
   subq     %rsi, %rax   # result = x-y
   movq     %rsi, %rdx
   subq     %rdi, %rdx   # eval = y-x
   cmpq     %rsi, %rdi   # x:y
   cmovle   %rdx, %rax   # if <=, result = eval
   ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Bad Performance

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Unsafe

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Illegal

"Do-While" Loop Compilation

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
`%rdi`	Argument **x**
`%rax`	`result`

```
    movl    $0, %eax     #   result = 0
.L2:                     # loop:
    movq    %rdi, %rdx
    andl    $1, %edx     #   t = x & 0x1
    addq    %rdx, %rax   #   result += t
    shrq    %rdi         #   x >>= 1
    jne     .L2          #   if(x) goto loop
    rep; ret
```

General "While" Translation #2

While version

```
while (Test)
  Body
```

- "Do-while" conversion
- Used with –O1

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
   (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

```c
int cool1(int a, int b) {
      if ( b < a )
            return b;
      else
            return a;
}

int cool2(int a, int b) {
      if ( a < b )
            return a;
      else
            return b;
}

int cool3(int a, int b) {
      unsigned ub = (unsigned) b;
      if ( ub < a )
            return a;
      else
            return ub;
}
```

```
        pushl %ebp
        movl %esp, %ebp
        movl 8(%ebp), %edx
        movl 12(%ebp), %eax
        cmpl %eax, %edx
        jge .L4
        movl %edx, %eax
.L4:    movl %ebp, %esp
        popl %ebp
        ret
```

Which of these C-functions converts to the above assembly code?

Address	Value	Register	Value
`0x104`	`0x34`	`%rax`	`0x104`
`0x108`	`0xCC`	`%rcx`	`0x5`
`0x10C`	`0x19`	`%rdx`	`0x3`
`0x110`	`0x42`	`%rbx`	`0x4`

Assume the following values are stored in the indicated registers/memory addresses. Fill in the table for the indicated operands:

Operand	Value	Operand	Value
`$0x110`	______	`3(%rax, %rcx)`	______
`%rax`	______	`256(, %rbx, 2)`	______
`0x110`	______	`(%rax, %rbx, 3)`	______
`(%rax)`	______		
`8(%rax)`	______		
`(%rax, %rbx)`	______		

Operand	Value	Operand	Value
`$0x110`	0x110	`3(%rax, %rcx)`	0x19
`%rax`	0x104	`256(, %rbx, 2)`	0xCC
`0x110`	0x42	`(%rax, %rbx, 3)`	0x42
`(%rax)`	0x34		
`8(%rax)`	0x19		
`(%rax, %rbx)`	0xCC		