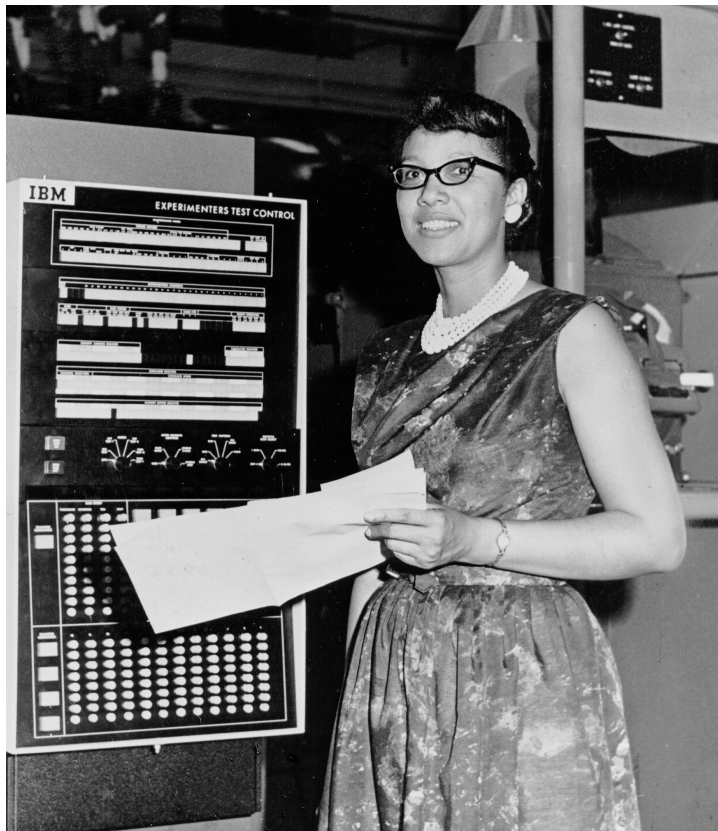
Lecture 5: Assembly Arithmetic and Conditionals

CS 105

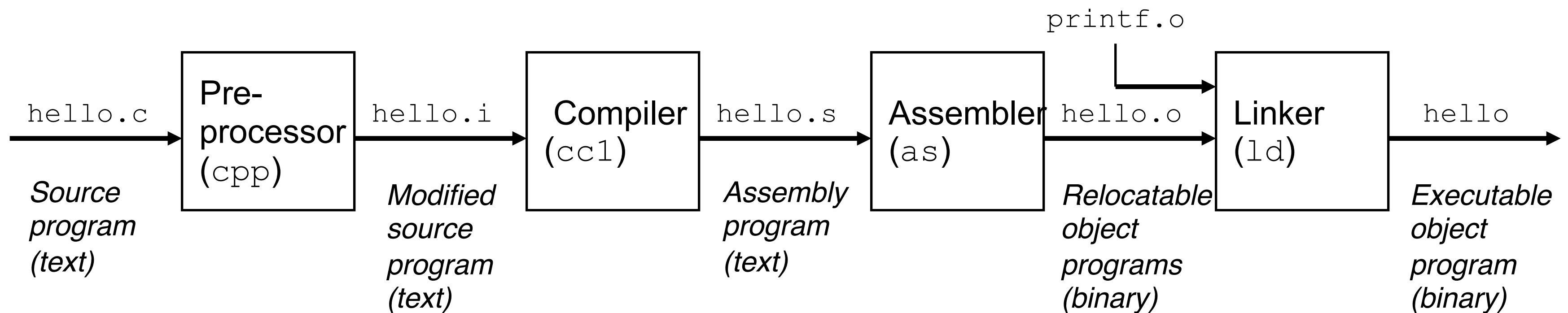
September 17, 2019

Programs




```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

Compilation




```
#include<stdio.h>

int main(int argc,
         char ** argv){

  printf("Hello
          world!\n");
  return 0;
}
```

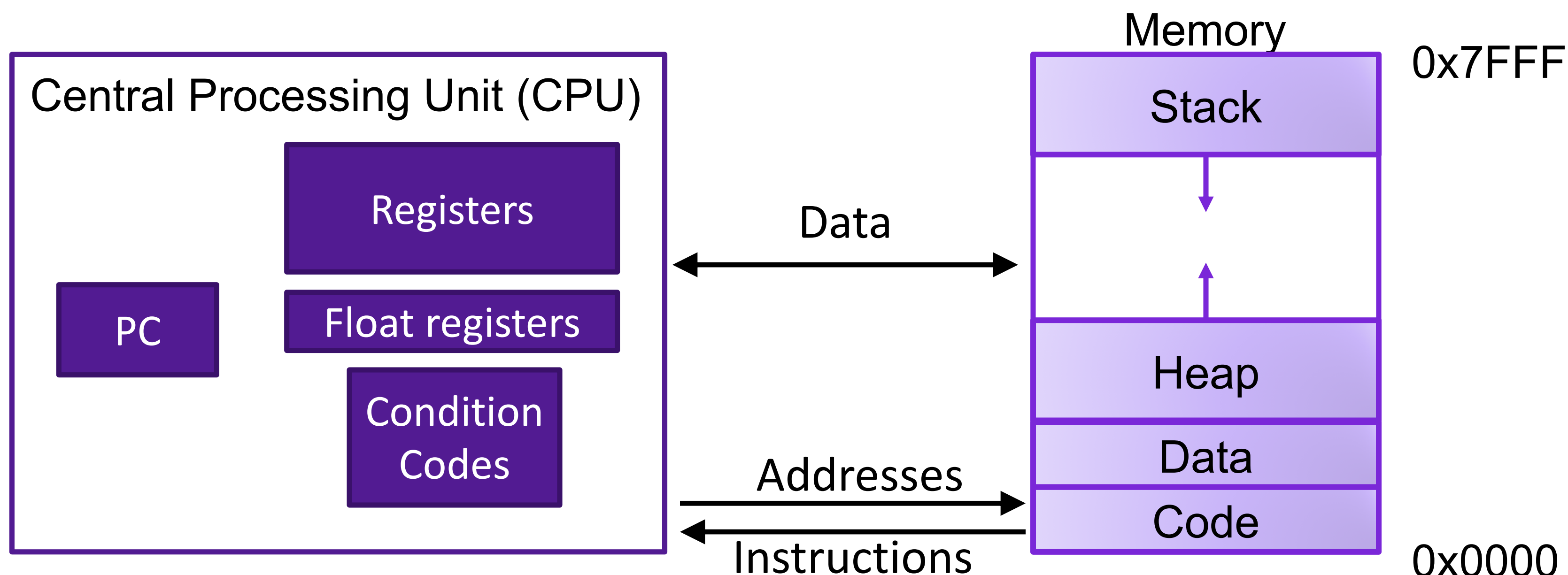
```
…
int printf(const char *
           restrict,
           ...)
__attribute__((__format_
_ (__printf__, 1, 2)));
…
int main(int argc,
         char ** argv){

  printf("Hello
          world!\n");
  return 0;
}
```

```
pushq   %rbp
movq    %rsp, %rbp
subq  $32, %rsp
leaq  L_.str(%rip), %rax
movl  $0, -4(%rbp)
movl  %edi, -8(%rbp)
movq  %rsi, -16(%rbp)
movq  %rax, %rdi
movb  $0, %al
callq _printf
xorl  %ecx, %ecx
movl  %eax, -20(%rbp)
movl  %ecx, %eax
addq  $32, %rsp
popq  %rbp
retq
```

```
55
48 89 e5
48 83 ec 20
48 8d 05 25 00 00 00
c7 45 fc 00 00 00 00
89 7d f8
48 89 75 f0
48 89 c7
b0 00
e8 00 00 00 00
31 c9
89 45 ec
89 c8
48 83 c4 20
5d
c3
```

Assembly/Machine Code View



Programmer-Visible State

- PC: Program counter (%rip)
- Register file: 16 Registers
- Float registers
- Condition codes

Memory

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

DATA TRANSFER IN ASSEMBLY

Data Movement Instructions

- MOV source, dest   Moves data source->dest

Sizes of C Data Types in x86-64

C declaration	Intel data type	Assembly suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

Data Movement Instructions

- MOV source, dest — Move data source->dest
  - `movb` — Move byte
  - `movw` — Move word
  - `movl` — Move double word
  - `movq` — Move quad word

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;`

Cannot do memory-memory transfer with a single instruction

X86-64 Integer Registers

64-bit	32-bit	16-bit	8-bit
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%bsl
%rbp	%ebp	%bp	%bpl

64-bit	32-bit
%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

Exercise

Register	Value
%rax	0x100
%rcx	0x1
%rdx	0x3

Memory Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13

- What are the values of the following operands (assuming register and memory state shown above)?
  1. %rax
  2. 0x104
  3. $0x108
  4. (%rax)
  5. 4(%rax)

X86-64 Register Usage Conventions

`%rax`, function result	`%r8`
`%rbx`	`%r9`
`%rcx`, fourth argument	`%r10`
`%rdx`, third argument	`%r11`
`%rsi`, second argument	`%r12`
`%rdi`, first argument	`%r13`
`%rsp`, stack pointer	`%r14`
`%rbp`, base pointer	`%r15`

Callee-saved registers are in yellow

Register	Use(s)
%rdi	Argument **xp**
%rsi	Argument **yp**
%rdx	Argument **zp**

Exercise

- Write a C function void decode1(long *xp, long *yp, long *zp) that will do the same thing as the following assembly code:

```
decode1:
    movq (%rdi), %r8
    movq (%rsi), %rcx
    movq (%rdx), %rax
    movq %r8, (%rsi)
    movq %rcx, (%rdx)
    movq %rax, (%rdi)
    ret
```

```
void decode1(long *xp,
             long *yp,
             long *zp){



}
```

ARITHMETIC IN ASSEMBLY

Some Arithmetic Operations

char	b	1
short	w	2
int	l	4
long	q	8
pointer	q	8

Suffixes

- Two Operand Instructions:

Format		Computation	
andq	Src,Dest	Dest = Dest & Src	
orq	Src,Dest	Dest = Dest | Src	
xorq	Src,Dest	Dest = Dest ^ Src	
salq	Src,Dest	Dest = Dest << Src	Also called shlq
sarq	Src,Dest	Dest = Dest >> Src	Arithmetic
shrq	Src,Dest	Dest = Dest >> Src	Logical
addq	Src,Dest	Dest = Dest + Src	
subq	Src,Dest	Dest = Dest – Src	
imulq	Src,Dest	Dest = Dest * Src	Signed multiply

- Note: different instructions for signed/unsigned multiply and divide
- Otherwise, no distinction between signed and unsigned int (why?)

Some Arithmetic Operations

char	**b**	**1**
short	**w**	**2**
int	**l**	**4**
long	**q**	**8**
pointer	**q**	**8**

Suffixes

- One Operand Instructions

notq	DestDest = ~Dest
incq	DestDest = Dest + 1
decq	DestDest = Dest – 1
negq	DestDest = – Dest

- See text for more instructions

Assembly Operations

char	b	1
short	w	2
int	l	4
long	q	8
pointer	q	8

Suffixes

- addq $47, %rax
- addq %rbx, %rax
- addq (%rbx), %rax
- addq %rbx, (%rax)
- addq 12(%rbx,%rdi,2), %rax

- Also movq, subq, andq, …

- leaq 12(%rbx,%rdi,2), %rax

Address Computation Instruction

- **`leaq`** Source, Dest
  - Source is address mode expression
  - Set Dest 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
```

Arithmetic Example

Register	Use(s)
`%rdi`	Argument **xp**
`%rsi`	Argument **yp**
`%rdx`	Argument **zp**
`%rax`	return value

```
long arith(long x, long y,
            long z){
  long rval = x+y;
  rval = rval+z;

  z = y * 48;
  long temp = y + z + 4;
  rval = rval * temp;
  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

Arithmetic Exercise

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	return value

```
arith2:
  orq     %rsi, %rdi
  sarq    $3, %rdi
  notq    %rdi
  movq    %rdx, %rax
  subq    %rdi, %rax
  ret
```

```
long arith2(long x, long y,
            long z){



}
```

Interesting Instructions

- **`sarq`**: arithmetic right shift

CONDITIONAL BRANCHES

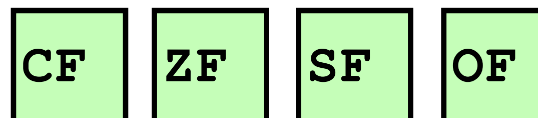
Branches and Jumps

- Processor state (partial)
  - Temporary data
    (`%rax`, …)
  - Location of runtime stack
    (`%rsp`)
  - Location of current code control point
    (`%rip`, …)
  - Status of recent tests
    (CF, ZF, SF, OF)

Registers

`%rax` (return val)	`%r8`
`%rbx`	`%r9`
`%rcx` (4th arg)	`%r10`
`%rdx` (3rd arg)	`%r11`
`%rsi` (2nd arg)	`%r12`
`%rdi` (1st arg)	`%r13`
`%rsp` (stack ptr)	`%r14`
`%rbp` (base ptr)	`%r15`

`%rip` Instruction pointer



Condition codes

Condition Codes

- Single bit registers
  - SF Sign Flag (for signed)
  - ZF Zero Flag
  - CF Carry Flag (for unsigned)
  - OF Overflow Flag (for signed)
- Implicitly set (as a side effect) by arithmetic operations and comparison operations
- Not set by **`leaq`** instruction

Condition Codes: `compare`

- Explicit setting by compare instruction
  - **`cmpq a,b`** like computing **`b-a`** without setting destination

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

Condition Codes: `test`

- Explicit setting by test instruction
  - `testq b,a` like computing `a&b` without setting destination
  - Useful to have one of the operands be a mask

    ```
    testq $(1<<63), %rax
    ```

  - Test for zero: `testq %rax, %rax`
  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`

Conditional Set Byte

- Set low-order byte of destination to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes

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)

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

Conditional Set Byte

- setX instruction: set a single byte based on condition codes
- Does not alter remaining bytes of destination
- Typically use movzbl to finish the job
  - 32 bit instruction, also sets upper 32 bits to zero

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

```
gt:
     cmpq   %rsi, %rdi   # set flags for x-y
     setg   %al          # ~(SF^OF)&~ZF, True when x > y
     movzbl %al, %eax    # Zero rest of %rax
     ret                 # return
```

Jumps

- A jump instruction can cause the execution to switch to a completely new position in the program (updates the program counter)
  - jmp Label
  - jmp *Operand

```
.L0:
  movq   $0, %rax
  jmp    .L1
  movq   (%rax), %rdx
.L1:
  movq   %rcx, %rax
```

```
jmp *%rax
```

Conditional Jumps

- jX instructions
  - Jump to different part of code if condition is true

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)

Conditional Branching

Register	Use
%rdi	x
%rsi	y
%rax	return value

```c
long absdiff(long x, long y){
  long result;

  if (x > y){
    result = x-y;
  } else {
    result = y-x;
  }

  return result;
}
```

```
absdiff:
   cmpq     %rsi, %rdi
   jle      .L4
   movq     %rdi, %rax
   subq     %rsi, %rax
   ret
.L4:        # x-y <= 0, x <= y
   movq     %rsi, %rax
   subq     %rdi, %rax
   ret
```

Exercise

```
test:
  leaq (%rdi, %rsi), %rax
  addq %rdx, %rax
  cmpq $-3, %rdi
  jge .L2
  cmpq %rdx, %rsi
  jge .L3
  movq %rdi, %rax
  imulq %rsi, %rax
  ret
.L3:
  movq %rsi, %rax
  imulq %rdx, %rax
  ret
.L2
  cmpq $2, %rdi
  jle .L4
  movq %rdi, %rax
  imulq %rdx, %rax
.L4:
  rep; ret
```

```
long test(long x, long y, long z){
  long val = ______;

  if(_______){

    if(_______){

      val = ______;


    } else {
      val = ______;

    }

 } else if (______){

    val = _______;

  }
  return val
}
```

Loops

- All use conditions and jumps
  - do-while
  - while
  - for

Do-while Loops

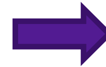
```
long bitcount(unsigned long x){
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

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

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

While Loops

```
while (Condition) {
    Body
}
```



```
if (Condition){
    do {
      Body
    } while (Condition)
}
```

```
long bitcount(unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

```
        movq    $0, %rax
        jmp     .L2
.L3:
        movq    %rdi, %rdx
        andq    $1, %rdx
        addq    %rdx, %rax
        shrq    %rdi
.L2:
        testq   %rdi, %rdi
        jne     .L3
        rep ret
```

For loops

```
for (Init; Cond; Incr){
    Body
}
```

```
Init;
while (Cond) {
    Body;
    Incr;
}
```

```
long bitcount(unsigned long x) {
  long result;
  for (result = 0; x; x >>= 1)
    result += x & 0x1;
  return result;
}
```

```
        movq    $0, %rax
        jmp     .L2
.L3:
        movq    %rdi, %rdx
        andq    $1, %rdx
        addq    %rdx, %rax
        shrq    %rdi
.L2:
        testq   %rdi, %rdi
        jne     .L3
        rep ret
```

Initial test can often be optimized away:

```
for (j = 0; j < 99; j++)
```