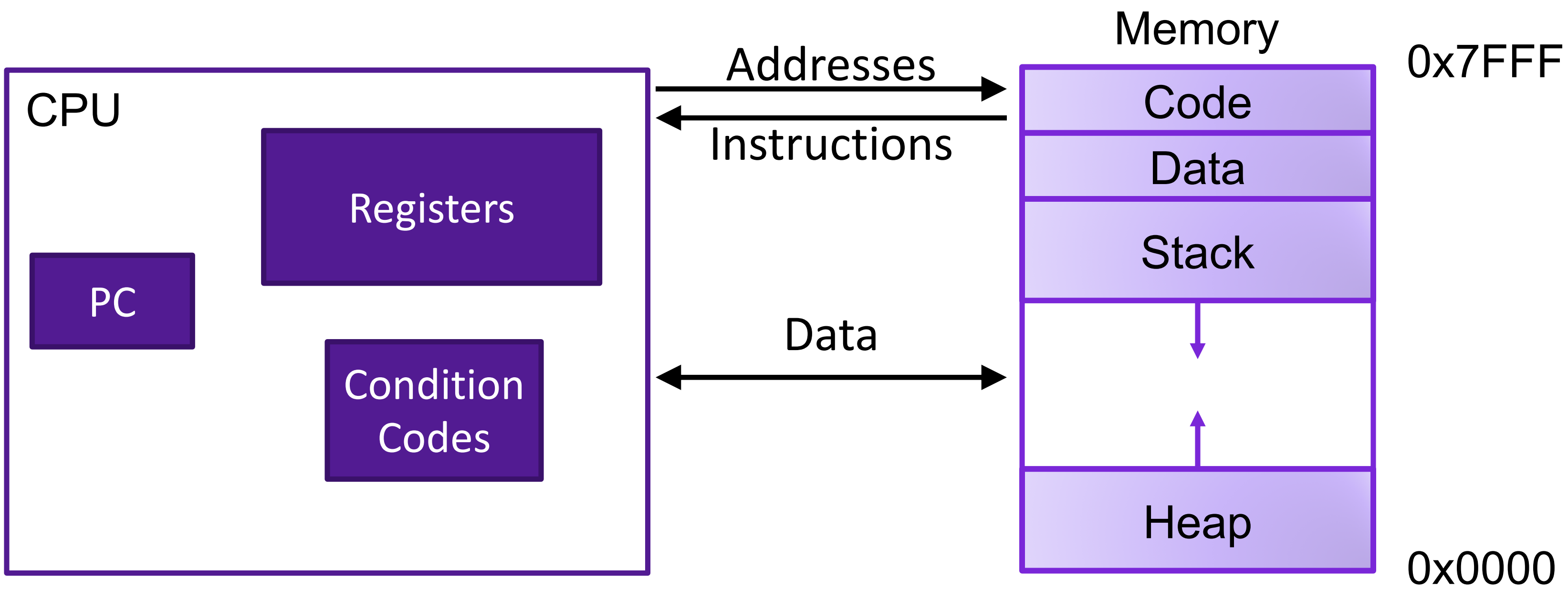
Lecture 5: Assembly Arithmetic and Control

CS 105 February 6, 2019

Assembly/Machine Code View



Programmer-Visible State

- PC: Program counter
- 16 Registers
- Condition codes

Memory

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

Assembly Characteristics: Operations

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

ARITHMETIC IN ASSEMBLY

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

Suffixes

Some Arithmetic Operations

- 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 Expression Example

```
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
```

```
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;
}
```

Interesting Instructions

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

Arithmetic Exercise

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

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





}
```

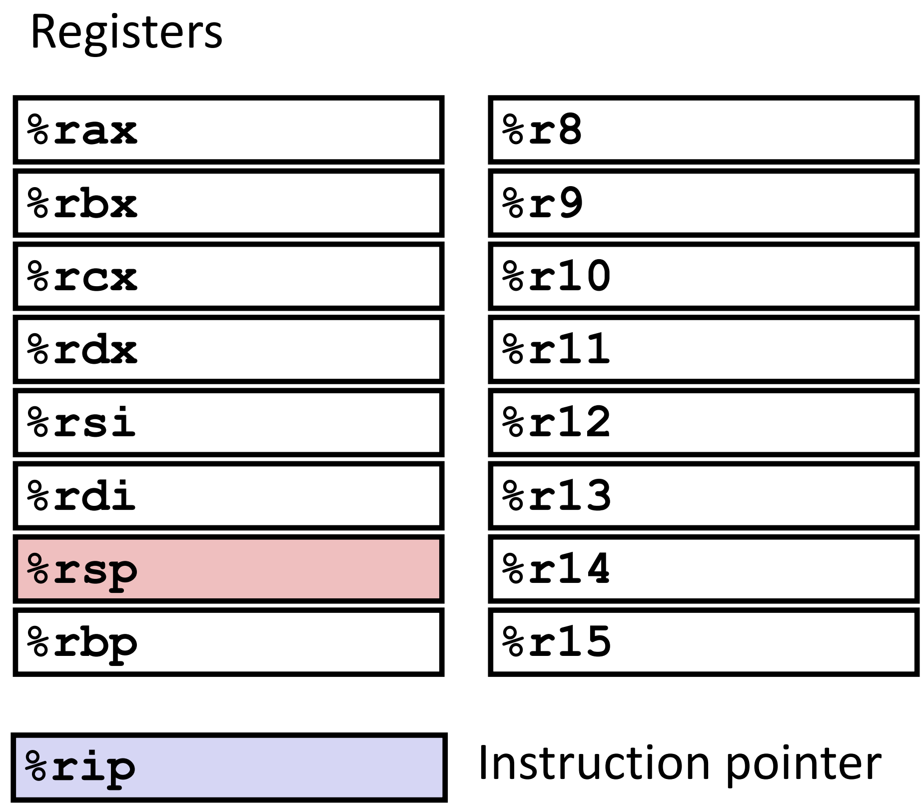
Interesting Instructions

- **sarq**: arithmetic shift

CONTROL FLOW

New Topic: Branches and Jumps

- Processor state (partial)
  - Temporary data (`%rax`, …)
  - Location of runtime stack (`%rsp`)
  - Location of current code control point (`%rip`, …)



Unconditional Jumps

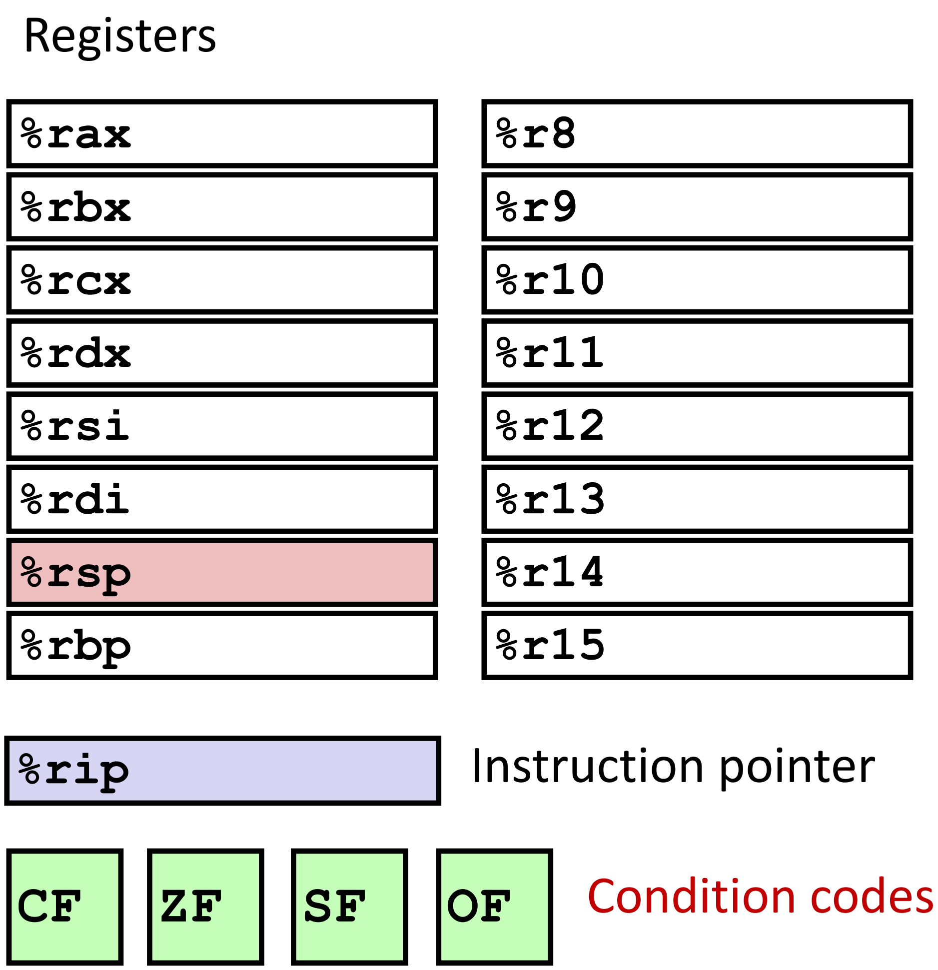
- A jump instruction can cause the execution to switch to a completely new position in the program (updates the program counter)
  - jmp Label
  - jmp *Operand

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

```
jmp *%rax
```

New Topic: 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)



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

  - CF set if carry 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: `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: **`testl %rax, %rax`**
  - ZF set when **`a&b == 0`**
  - SF set when **`a&b < 0`**

Reading Condition Codes

- 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)

Reading Condition Codes, continued

- 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

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

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

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

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)

Conditional Branching

```
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
%rdi	x
%rsi	y
%rax	return value

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
- Example: count number of 1's in x

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

Do-while, translated to goto

```
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;
}
```

Do-while translation, continued

```
long bitcount(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
```

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;
}
```

```
        movl   $0, %eax
        jmp    .L2
.L3:
        movq   %rdi, %rdx
        andl   $1, %edx
        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;
}
```

Initial test can often be optimized away:

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