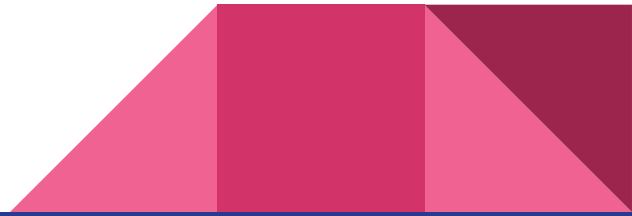
CS 33: Computer Organization

Dis 1B: Week 3 Discussion

Agenda

- Last minute questions for Data Lab!
- x86
- More x86
- More more x86

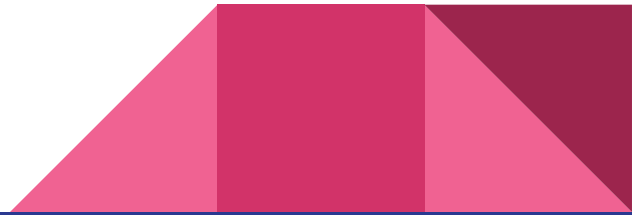


Lab 1

- Any questions?

Review

- mov instruction: moves data from the source to the destination
- D(Rb, Ri, S)
  - Mem[Data stored in Rb + data stored in Ri * S + D]
- leaq instruction: Uses addressing mode expression but doesn't do memory access
  - Used for computing simple arithmetic expression
  - Suppose %rdx has 0x10
  - leaq (%rdx, %rdx, 2), %rax
  - movq (%rdx, %rdx, 2), %rax



Review

`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

Also called shlq

Arithmetic

Logical

Review Problem

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Fill in the following table showing the values for the indicated operands:

Operand	Value
%rax	_____
0x104	_____
$0x108	_____
(%rax)	_____
4(%rax)	_____
9(%rax,%rdx)	_____
260(%rcx,%rdx)	_____
0xFC(,%rcx,4)	_____
(%rax,%rdx,4)	_____

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Review Problem

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x108	0xAB	%rcx	0x1
0x110	0x13	%rdx	0x3
0x118	0x11		

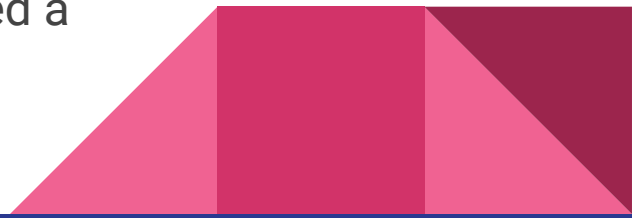
Fill in the following table showing the effects of the following instructions, in terms of both the register or memory location that will be updated and the resulting value:

Instruction	Destination	Value
`addq %rcx,(%rax)`		
`subq %rdx,8(%rax)`		
`imulq $16,(%rax,%rdx,8)`		
`incq 16(%rax)`		
`decq %rcx`		
`subq %rdx,%rax`		

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

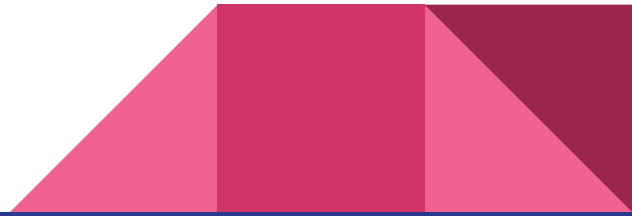
Condition Codes

- 4 single-bit code registers
- Describes attributes of the most recent arithmetic or logical operation
- **CF (carry flag)**: The most recent operation generated a carry out of the most significant bit
- **ZF (zero flag)**: The most recent operation yielded zero
- **SF (sign flag)**: The most recent operation yielded a negative value
- **OF (overflow flag)**: The most recent operation caused a two’s-complement overflow



Why do we need these Condition Codes?

- Usually, condition codes aren’t used directly
- However, combination of these condition codes are very powerful



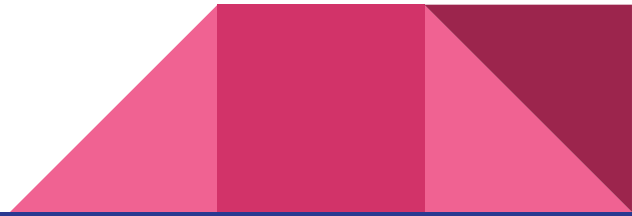
Well… What can we do with Condition Codes?

- Set a single byte to 0 or 1
- Conditionally jump to some other part of the program
- Conditionally transfer data

- Well, that doesn’t seem *that* powerful...

cmpq

- cmpq b, a
  - a - b without setting destination
  - Only used to set the condition codes



set

- Sets a single byte to 0 or 1 on some combination of the condition codes

Instruction		Synonym	Effect	Set condition
`sete`	*D*	`setz`	*D* ← ZF	Equal / zero
`setne`	*D*	`setnz`	*D* ← ~ZF	Not equal / not zero
`sets`	*D*		*D* ← SF	Negative
`setns`	*D*		*D* ← ~SF	Nonnegative
`setg`	*D*	`setnle`	*D* ← ~(SF ^ OF) & ~ZF	Greater (signed >)
`setge`	*D*	`setnl`	*D* ← ~(SF ^ OF)	Greater or equal (signed >=)
`setl`	*D*	`setnge`	*D* ← SF ^ OF	Less (signed <)
`setle`	*D*	`setng`	*D* ← (SF ^ OF) | ZF	Less or equal (signed <=)
`seta`	*D*	`setnbe`	*D* ← ~CF & ~ZF	Above (unsigned >)
`setae`	*D*	`setnb`	*D* ← ~CF	Above or equal (unsigned >=)
`setb`	*D*	`setnae`	*D* ← CF	Below (unsigned <)
`setbe`	*D*	`setna`	*D* ← CF | ZF	Below or equal (unsigned <=)

Example

```
1  int isFirstLessThanSecond(long first, long second) {
2      // first is in %rdi, second is in %rsi
3      return a < b
4  }
5
6  isFirstLessThanSecond:
7      cmpq    %rsi, %rdi
8      setl    %al
9      movzbl  %al, %eax
10     ret
```

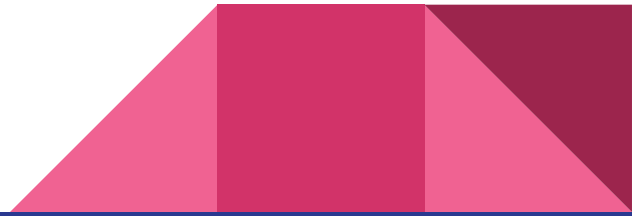
jump

- Jump to a labeled destination when the jump condition holds

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Jump is an extremely powerful instruction!

- What can we do with jump instructions?
  - If-else statement
  - While loop
  - For loop
  - Switch statement

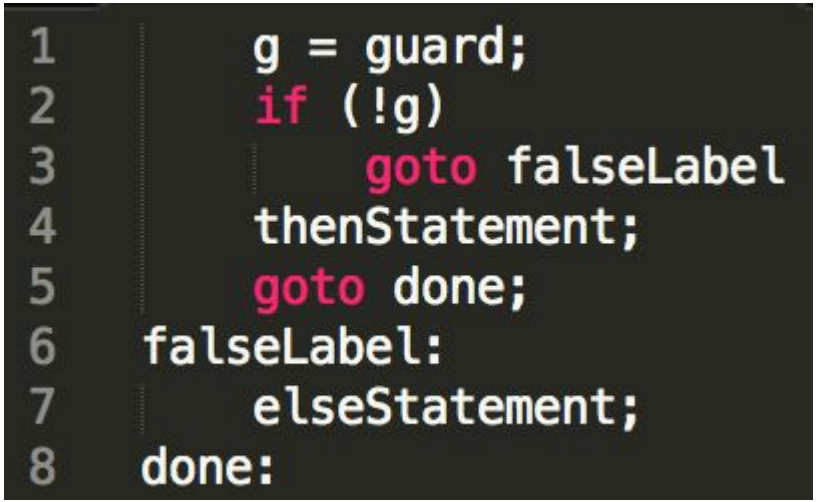


If-else Statement

- How can we change this statement using jump?
- Hint: Only one of the two branch statements is executed

General Form of If-else statement for Assembly

- Compiler creates separate blocks of code for thenStatement and elseStatement
- What is the benefit of inverting the guard?



```
long test(long x, long y, long z) {
    long val = _____;
    if (_____) {
        if (_____)
            val = _____;
        else
            val = _____;
    } else if (_____)
        val = _____;
    return val;
}
```

GCC generates the following assembly code:

```
  long test(long x, long y, long z)
  x in %rdi, y in %rsi, z in %rdx
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
```

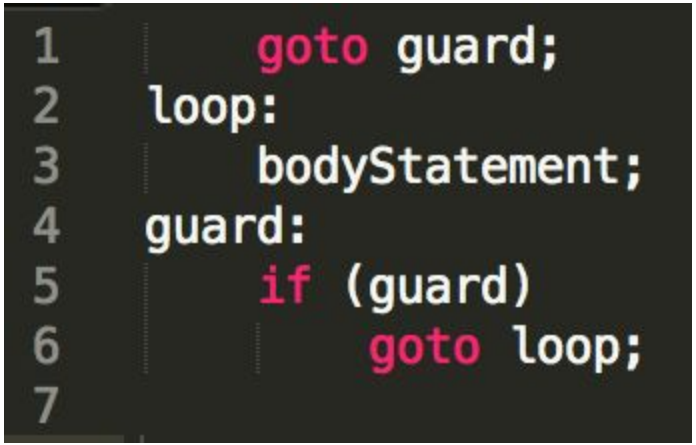
Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

While Loop

- How can we change this statement using jump?

```
1   while (guard)
2       bodyStatement;
```

General Form of While Loop in Assembly



```
long loop_while(long a, long b)
{   long result = _____;
    while (_____) {
        result = _____;
        a = _____;
    }
    return result;
}
```

GCC, run with command-line option -Og, produces the following code:

```
    long loop_while(long a, long b)
    a in %rdi, b in %rsi
1   loop_while:
2     movl    $1, %eax
3     jmp     .L2
4   .L3:
5     leaq    (%rdi,%rsi), %rdx
6     imulq   %rdx, %rax
7     addq    $1, %rdi
8   .L2:
9     cmpq    %rsi, %rdi
10    jl      .L3
11    rep; ret
```

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

```
void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
    case _____:          /* Case A */
        c = _____;
        /* Fall through */
    case _____:          /* Case B */
        val = _____;
        break;
    case _____:          /* Case C */
    case _____:          /* Case D */
        val = _____;
        break;
    case _____:          /* Case E */
        val = _____;
        break;
    default:
        val = _____;
    }
    *dest = val;
}
```

(a) Code

```
    void switcher(long a, long b, long c, long *dest)
    a in %rsi, b in %rdi, c in %rdx, d in %rcx
1   switcher:
2     cmpq    $7, %rdi
3     ja      .L2
4     jmp     *.L4(,%rdi,8)
5     .section        .rodata
6   .L7:
7     xorq    $15, %rsi
8     movq    %rsi, %rdx
9   .L3:
10    leaq    112(%rdx), %rdi
11    jmp     .L6
12  .L5:
13    leaq    (%rdx,%rsi), %rdi
14    salq    $2, %rdi
15    jmp     .L6
16  .L2:
17    movq    %rsi, %rdi
18  .L6:
19    movq    %rdi, (%rcx)
20    ret
```

(b) Jump table

```
1   .L4:
2     .quad   .L3
3     .quad   .L2
4     .quad   .L5
5     .quad   .L2
6     .quad   .L6
7     .quad   .L7
8     .quad   .L2
9     .quad   .L5
```

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition