



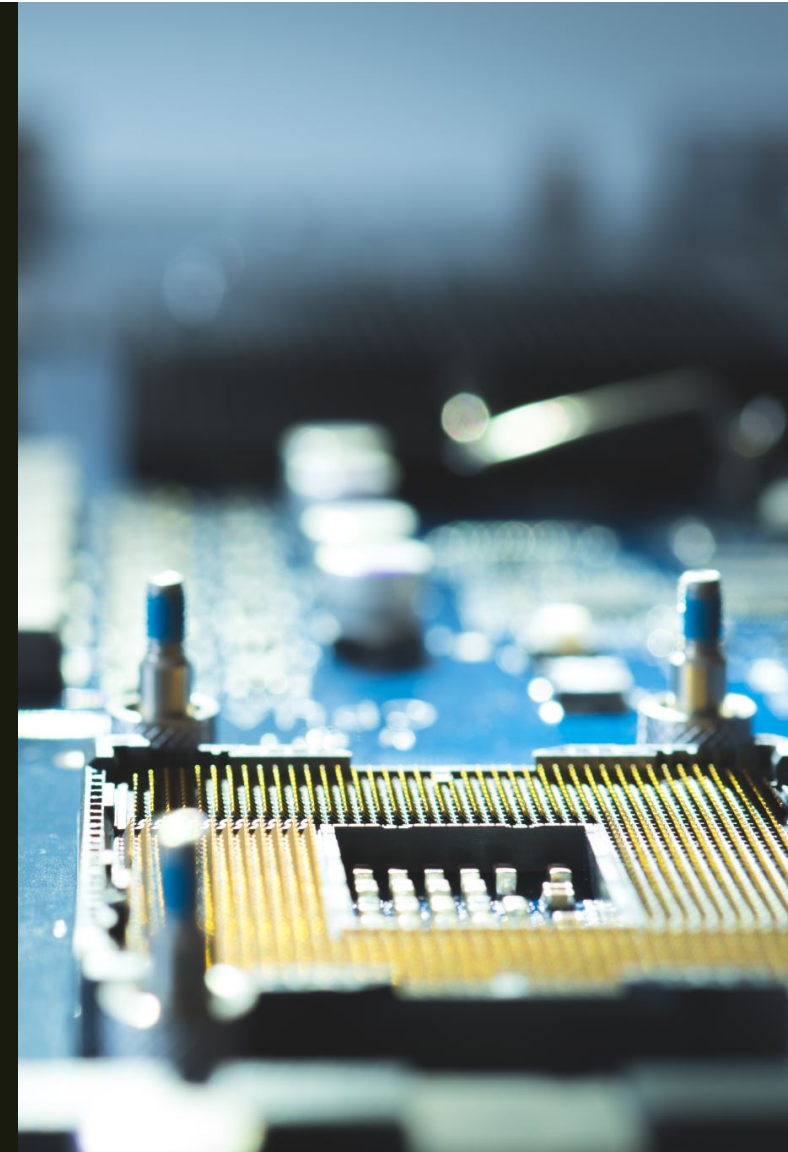
CS 211 RECITATIONS

WEEK 9

Wenjie Qiu
Teaching Assistant
Office Hour: Thu. noon – 1pm
wenjie.qiu@rutgers.edu

Content

- Condition codes & branches
- Loops
- Switch statements
- Functions
- Recursions



Processor State

■ Info about current program

- 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	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

Current stack top

%rip Instruction pointer

CF	ZF	SF	OF	Condition codes
----	----	----	----	-----------------

■ Explicit Setting by Compare Instruction

- `cmpq Src2, Src1`
- `cmpq b, a` like computing `a-b` 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)`

■ Implicitly set (think of it as side effect) by arithmetic operations

Example: `addq Src, Dest` \leftrightarrow `t = a+b`

CF set if carry 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)`

■ Single bit registers

- | | | |
|-----|---------------------------|-------------------------------|
| ▪CF | Carry Flag (for unsigned) | SF Sign Flag (for signed) |
| ▪ZF | Zero Flag | OF Overflow Flag (for signed) |

■ Explicit Setting by Test instruction

- `testq Src2, Src1`
 - `testq b,a` like computing `a&b` without setting destination
- Sets condition codes based on value of `Src1` & `Src2`
- Useful to have one of the operands be a mask
- **ZF set** when `a&b == 0`
- **SF set** when `a&b < 0`

■ SetX Instructions

- 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
<code>sete</code>	ZF	Equal / Zero
<code>setne</code>	\sim ZF	Not Equal / Not Zero
<code>sets</code>	SF	Negative
<code>setns</code>	\sim SF	Nonnegative
<code>setg</code>	\sim (SF \wedge OF)& \sim ZF	Greater (Signed)
<code>setge</code>	\sim (SF \wedge OF)	Greater or Equal (Signed)
<code>setl</code>	(SF \wedge OF)	Less (Signed)
<code>setle</code>	(SF \wedge OF) ZF	Less or Equal (Signed)
<code>seta</code>	\sim CF& \sim ZF	Above (unsigned)
<code>setb</code>	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

```
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
```

■ 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 Branch Example

- Generation

```
$ gcc -Og -S -fno-if-conversion control.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 # 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

“Do-While” Loop Example

C Code

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

“Do-While” Loop Compilation

Goto Version

```
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
    ret
```


General “While” Translation #1

- “Jump-to-middle” translation
- Used with **-Og**

While version

```
while (Test)
    Body
```



Goto Version

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

While Loop Example #1

C Code

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

Jump to

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

- Compare to do-while version of function
- Initial goto starts loop at test

“For” Loop Do-While Conversion

```
#define WSIZE 8*sizeof(int)
```

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

- Initial test can be optimized away

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (!(i < WSIZE)) Init
    goto done; !Test
loop:
{
    unsigned bit =
        (x >> i) & 0x1; Body
    result += bit;
}
i++; Update
if (i < WSIZE) Test
    goto loop;
done:
    return result;
}
```

Jump Table Structure

Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
    ...  
  case val_n-1:  
    Block n-1  
}
```

Jump Table

jtab:	Targ0
	Targ1
	Targ2
	•
	•
	•
	Targn-1

Jump Targets

Targ0:	Code Block 0
Targ1:	Code Block 1
Targ2:	Code Block 2
	•
	•
	•
Targn-1:	Code Block n-1

Translation (Extended C)

```
goto *jTab[x];
```

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Jump table

.section	.rodata
.align 8	
.L4:	
.quad	.L8 # x = 0
.quad	.L3 # x = 1
.quad	.L5 # x = 2
.quad	.L9 # x = 3
.quad	.L8 # x = 4
.quad	.L7 # x = 5
.quad	.L7 # x = 6

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8          # Use default
    Indirect jump → jmp    *.L4(,%rdi,8) # goto *JTab[x]
```

Assembly Setup Explanation

■ Table Structure

- Each target requires 8 bytes
- Base address at .L4

Jump table

.section	.rodata
.align 8	
.L4:	
.quad	.L8 # x = 0
.quad	.L3 # x = 1
.quad	.L5 # x = 2
.quad	.L9 # x = 3
.quad	.L8 # x = 4
.quad	.L7 # x = 5
.quad	.L7 # x = 6

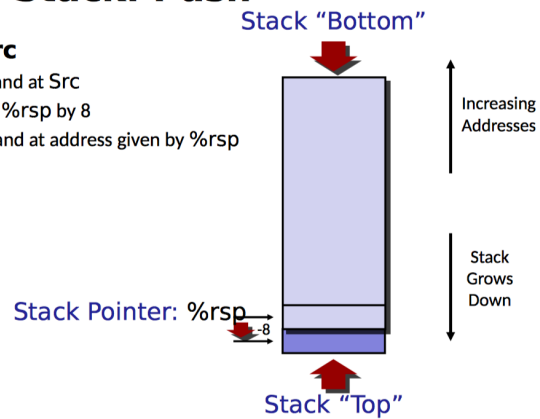
■ Jumping

- Direct:** `jmp .L8`
- Jump target is denoted by label .L8
- Indirect:** `jmp *.L4(,%rdi,8)`
- Start of jump table: .L4
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective Address `.L4 + x*8`
 - Only for $0 \leq x \leq 6$

x86-64 Stack: Push

■ **pushq Src**

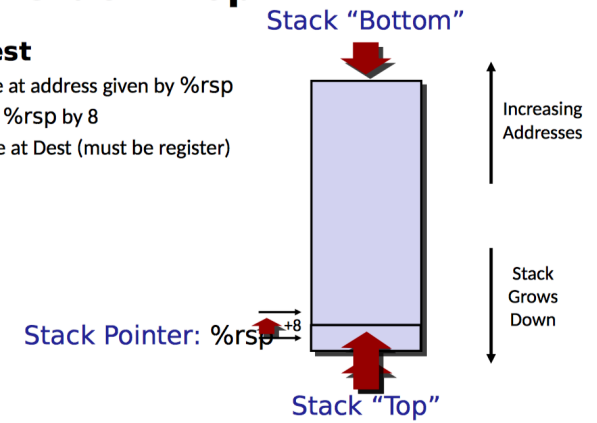
- Fetch operand at Src
- Decrement %rsp by 8
- Write operand at address given by %rsp



x86-64 Stack: Pop

■ **popq Dest**

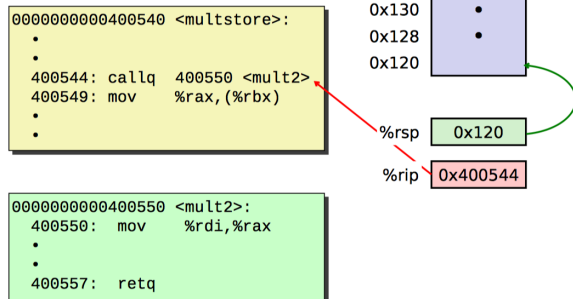
- Read value at address given by %rsp
- Increment %rsp by 8
- Store value at Dest (must be register)



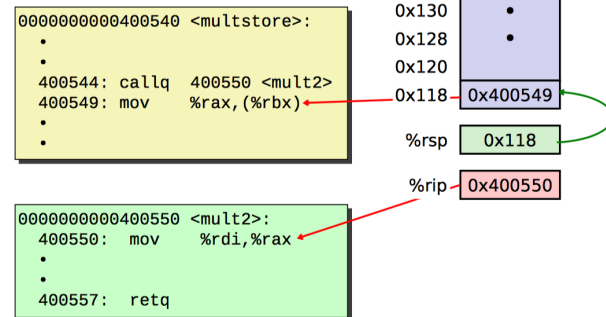
Procedure Control Flow

- **Use stack to support function call and return**
- **Procedure call:** `call label`
 - Push return address on stack
 - Jump to label
- **Return address:**
 - Address of the next instruction right after call
 - Example from disassembly
- **Procedure return:** `ret`
 - Pop address from stack
 - Jump to address

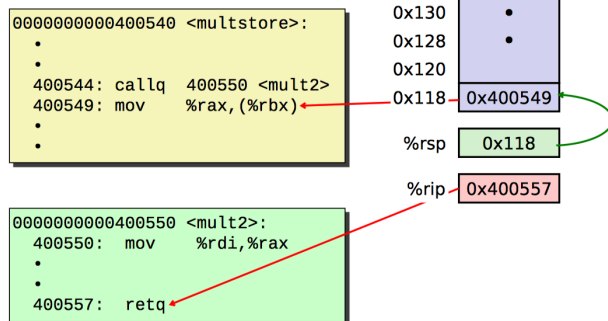
Control Flow Example



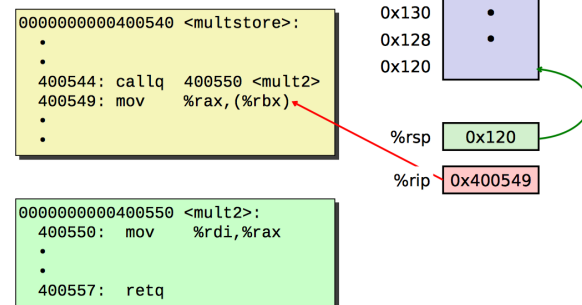
Control Flow Example



Control Flow Example



Control Flow Example



Procedure Data Flow

Registers

■ First 6 arguments

%rdi
%rsi
%rdx
%rcx
%r8
%r9

■ Return value

%rax

Stack

...
Arg n
...
Arg 8
Arg 7

■ Only allocate stack space when needed

Data Flow Examples

```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
# x in %rdi, y in %rsi, dest in %rdx
...
400541: mov    %rdx,%rbx      # Save dest
400544: callq 400550 <mult2>  # mult2(x,y)
# t in %rax
400549: mov    %rax,(%rbx)    # Save at dest
...
```

```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
# a in %rdi, b in %rsi
400550: mov    %rdi,%rax      # a
400553: imul   %rsi,%rax      # a * b
# s in %rax
400557: retq                    # Return
```

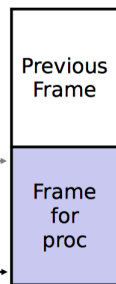

Stack Frames

■ Contents

- Return information
- Local storage (if needed)
- Temporary space (if needed)

Frame Pointer: %rbp
(Optional)

Stack Pointer: %rsp



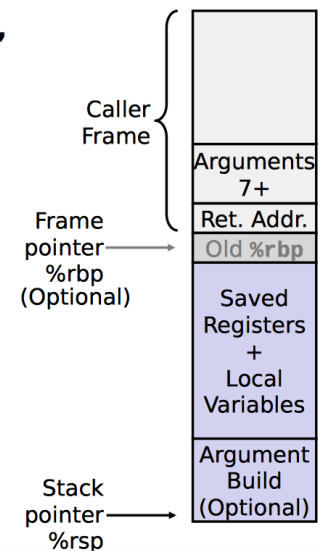
Stack "Top"

■ Management

- Space allocated when enter procedure
 - "Set-up" code
 - Includes push by **call** instruction
- Deallocated when return
 - "Finish" code
 - Includes pop by **ret** instruction

■ Current Stack Frame ("Top" to Bottom)

- "Argument build:"
Parameters for function about to call
- Local variables
If can't keep in registers
- Saved register context
- Old frame pointer (optional)



■ Caller Stack Frame

- Return address
 - Pushed by call instruction
- Arguments for this call

Observations About Recursion

■ Handled Without Special Consideration

- Stack frames mean that each function call has private storage
 - Saved registers & local variables
 - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
 - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
- Stack discipline follows call / return pattern
 - If P calls Q, then Q returns before P
 - Last-In, First-Out

■ Also works for mutual recursion

- P calls Q; Q calls P

x86-64 Procedure Summary

■ Important Points

- Stack is the right data structure for procedure call / return
 - If P calls Q, then Q returns before P

■ Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result return in %rax
- Pointers are addresses of values
 - On stack or global

