CSCI 2021: x86-64 Control Flow

Chris Kauffman

Last Updated:
Mon Oct 26 01:24:59 PM CDT 2020
Logistics

Reading Bryant/O’Hallaron

- Ch 3.6: Control Flow
- Ch 3.7: Procedure calls

Goals

- Finish Assembly Basics
- Jumps and Control flow
- Comparison / Test Instructions
- Procedure calls

Labs/HW

- Lab07: Assembly coding, must submit code to Gradescope by next Wed night, Autograded
- HW07: Data in assembly, gdb and assembly/binaries, Gradescope Quiz for Credit

Project 3:

- Problem 1: Thermometer Assembly Functions (50%)
- Problem 2: Binary Bomb via GDB (50%)
Control Flow in Assembly and the Instruction Pointer

- No high-level conditional or looping constructs in assembly
- Only `%rip`: Instruction Pointer or “Program Counter”: memory address of the next instruction to execute
- Don’t mess with `%rip` by hand: automatically increases as instructions execute so the next valid instruction is referenced
- Jump instructions modify `%rip` to go elsewhere
- Typically label assembly code with positions of instructions that will be the target of jumps
- **Unconditional Jump** Instructions always jump to a new location.
- **Comparison / Test** Instruction, sets EFLAGS bits indicating relation between registers/values
- **Conditional Jump** Instruction, jumps to a new location if certain bits of EFLAGS are set, ignored if bits not set
Examine: Loop Sum with Instruction Pointer (rip)

- Can see direct effects on rip in disassembled code
- rip increases corresponding to instruction length
- Jumps include address for next rip

// C Code equivalent
int sum=0, i=1, lim=100;
while(i<=lim){
    sum += i;
    i++;
}
return sum;

```
00000000000005fa <main>:
ADDR HEX-OPCODES ASSEMBLY EFFECT ON RIP
5fa: 48 c7 c0 00 00 00 00 mov $0x0,%rax # rip = 5fa -> 601
601: 48 c7 c1 01 00 00 00 mov $0x1,%rcx # rip = 601 -> 608
608: 48 c7 c2 64 00 00 00 mov $0x64,%rdx # rip = 608 -> 60f

000000000000060f <LOOP>:
60f: 48 39 d1 cmp %rdx,%rcx # rip = 60f -> 612
612: 7f 08 jg 61c <END> # rip = 612 -> 614 OR 61c
614: 48 01 c8 add %rcx,%rax # rip = 614 -> 617
617: 48 ff c1 inc %rcx # rip = 617 -> 61a
61a: eb f3 jmp 60f <LOOP> # rip = 61a -> 60f

000000000000061c <END>:
61c: c3 retq # rip 61c -> return address
```
Disassembling Binaries

- Binaries hard to read on their own
- Many tools exist to work with them, notably objdump on Unix
- Can **disassemble** binary: show “readable” version of contents

```bash
> gcc -Og loop.s  # COMPILER AND ASSEMBLE

> file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),

> objdump -d a.out  # DISASSEMBLE BINARY
a.out: file format elf64-x86-64
...
Disassembly of section .text:
...
0000000000001119 <main>:
  1119: 48 c7 c0 00 00 00 00  mov $0x0,%rax
  1120: 48 c7 c1 01 00 00 00  mov $0x1,%rcx
  1127: 48 c7 c2 64 00 00 00  mov $0x64,%rdx
000000000000112e <LOOP>:
  112e: 48 39 d1  cmp %rdx,%rcx
  1131: 7f 08  jg 113b <END>
  1133: 48 01 c8  add %rcx,%rax
  1136: 48 ff c1  inc %rcx
  1139: eb f3  jmp 112e <LOOP>
000000000000113b <END>:
  113b: c3  retq
```
FLAGS: Condition Codes Register

- Most CPUs have a special register with “flags” for various conditions.
- In x86-64 this register goes by the following names:

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAGS</td>
<td>16-bit</td>
<td>Most important bits in first 16</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>32-bit</td>
<td>Name shown in gdb</td>
</tr>
<tr>
<td>RFLAGS</td>
<td>64-bit</td>
<td>Not used normally</td>
</tr>
</tbody>
</table>

- Bits in FLAGS register are automatically set based on results of other operations.
- Pertinent examples with conditional execution:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Abbrev</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CF</td>
<td>Carry flag</td>
<td>Set if last op caused unsigned overflow</td>
</tr>
<tr>
<td>6</td>
<td>ZF</td>
<td>Zero flag</td>
<td>Set if last op yielded a 0 result</td>
</tr>
<tr>
<td>7</td>
<td>SF</td>
<td>Sign flag</td>
<td>Set if last op yielded a negative</td>
</tr>
<tr>
<td>8</td>
<td>TF</td>
<td>Trap flag</td>
<td>Used by gdb to stop after one ASM instruction</td>
</tr>
<tr>
<td>9</td>
<td>IF</td>
<td>Interrupt flag</td>
<td>1: handle hardware interrupts, 0: ignore them</td>
</tr>
<tr>
<td>11</td>
<td>OF</td>
<td>Overflow flag</td>
<td>Set if last op caused signed overflow/underflow</td>
</tr>
</tbody>
</table>
Comparisons and Tests

Set the EFLAGS register by using comparison instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Instruction</th>
<th>Examples</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>cmpX B, A</td>
<td>cmpl $1,%eax</td>
<td>Like if(eax &gt; 1){...}</td>
</tr>
<tr>
<td></td>
<td>Like: A - B</td>
<td>cmpq %rsi,%rdi</td>
<td>Like if(rdi &gt; rsi){...}</td>
</tr>
</tbody>
</table>

| Test   | testX B, A    | testq %rcx,%rdx      | Like if(rdx & rcx){...}        |
|        | Like: A & B   | testl %rax,%rax      | Like if(rax){...}             |

- ▶ Immediates like $2 must be the first argument B
- ▶ B,A are NOT altered with cmp/test instructions
- ▶ EFLAGS register IS changed by cmp/test to indicate less than, greater than, 0, etc.

### EXAMPLES:
```
cmpl $1, %eax     # compare eax to 1
                  # EFLAGS bits set based on result of eax - 1
                  # ZF (zero flag) now 1 if eax==1
                  # SF (sign flag) now 1 if eax<1

testq %rax,%rax   # test rax against rax
                  # EFLAGS bits set based on result of rax & rax
                  # ZF (zero flag) now 1 if rax==0 (falsey)
                  # ZF (zero flag) now 0 if rax!=0 (truthy)
```
Jump Instruction Summary

- **jmp LAB**: unconditional jump, always go to another code location.
- Control structures like `if/else/for/while` use `cmpX / testX` followed by **conditional jumps**.
- `ja` used by compiler for `if(a < 0 || a > lim)`; Consider sign/unsigned to explain why.
- **jmp *%rdx** allows **function pointers**, powerful but no time to discuss.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Jump Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jmp LAB</code></td>
<td>Unconditional jump</td>
</tr>
<tr>
<td><code>je LAB</code></td>
<td>Equal / zero</td>
</tr>
<tr>
<td><code>jz LAB</code></td>
<td></td>
</tr>
<tr>
<td><code>jne LAB</code></td>
<td>Not equal / non-zero</td>
</tr>
<tr>
<td><code>jnz LAB</code></td>
<td></td>
</tr>
<tr>
<td><code>js LAB</code></td>
<td>Negative (&quot;signed&quot;)</td>
</tr>
<tr>
<td><code>jns LAB</code></td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>jg LAB</code></td>
<td>Greater-than signed</td>
</tr>
<tr>
<td><code>jge LAB</code></td>
<td>Greater-than-equal signed</td>
</tr>
<tr>
<td><code>jl LAB</code></td>
<td>Less-than signed</td>
</tr>
<tr>
<td><code>jle LAB</code></td>
<td>Less-than-equal signed</td>
</tr>
<tr>
<td><code>ja LAB</code></td>
<td>Above unsigned</td>
</tr>
<tr>
<td><code>jae LAB</code></td>
<td>Above-equal unsigned</td>
</tr>
<tr>
<td><code>jb LAB</code></td>
<td>Below unsigned</td>
</tr>
<tr>
<td><code>jbe LAB</code></td>
<td>Below-equal unsigned</td>
</tr>
<tr>
<td><code>jmp *OPER</code></td>
<td>Unconditional jump to variable address</td>
</tr>
</tbody>
</table>
Examine: Compiler Comparison Inversion

- Often compiler inverts comparisons
- \( i < n \) becomes \( \text{cmpX} / \text{jge} \) (jump greater/equal)
- \( i == 0 \) becomes \( \text{cmpX} / \text{jne} \) (jump not equal)
- This allows “true” case to fall through immediately
- Depending on structure, may have additional jumps
  - \( \text{if()} \)\{ . . \} usually has a single jump
  - \( \text{if()}\{}\text{ else }\{\} \) may have a couple jumps

## Assembly translation of

```assembly
if(rbx >= 2){
  rdx = 10;
}
else{
  rdx = 5;
}
return rdx;
```

```assembly
cmpq $2,%rbx  # compare: rbx-0
jl .LESSTHAN  # goto less than
if(rbx >= 2){
movq $10,%rdx  # greater/equal
}
jmp .AFTER
```

.LESSTHAN:
```
else{
movq $5,%rdx  # less than
}
```

.AFTER:
```
  rdx is 10 if rbx >= 2
  rdx is 5 otherwise
  movq %rdx,%rax
  ret
```
Exercise: Other Kinds of Conditions

Other Things to Look For

- `testl %eax,%eax` used to check zero/nonzero
- Followed by `je / jz / jne / jnz`
- Also works for NULL checks
- Negative Values, followed by `js / jns` (jump sign / jump no sign)

See `jmp_tests_asm.s`

- Trace the execution of this code
- Determine return value in `%eax`
Exercise: Other Kinds of Conditions

main:
    movl $0,%eax
    movl $5,%edi
    movl $3,%esi
    movq $0,%rdx
    movl $-4,%ecx

    testl %edi,%edi
    jnz .NONZERO

    addl $20,%eax

.NONZERO:
    testl %esi,%esi
    jz .FALSEY

    addl $30,%eax

.FALSEY:
    testq %rdx,%rdx
    je .ISNULL

    addl $40,%eax

.ISNULL:
    testl %ecx,%ecx
    jns .NEGATIVE

    addl $50,%eax

.NEGATIVE:
    ret
### From jmp_tests_asm_commented.s

```assembly
main:
    movl $0,%eax # eax is 0

    movl $5,%edi # set initial vals
    movl $3,%esi # for registers to
    movl $0,%edx # use in tests
    movl $-4,%ecx

    ## eax=0, edi=5, esi=3, edx=NULL, ecx=-4
    testl %edi,%edi # any bits set?
    jnz .NONZERO # jump on !ZF (zero flag), same as jne
    ## if(edi == 0){
    addl $20,%eax
    ## }

    .NONZERO:
    testl %esi,%esi # any bits set?
    jz .FALSEY # jump on ZF same as je
    ## if(esi){
    addl $30,%eax
    ## }

    .FALSEY:
    testq %rdx,%rdx # any bits set
    je .ISNULL # same as jz: jump on ZF
    ## if(rdx != NULL){
    addl $40,%eax
    ## }

    .ISNULL:
    testl %ecx,%ecx # sign flag set on test to indicate negative results
    jns .NEGATIVE # jump on !SF (not sign flag/pos)
    ## if(ecx < 0){
    addl $50,%eax
    ## }

    .NEGATIVE:
    ret ## eax is return value
```

### Answers: Other Kinds of Conditions
cmov Family: Conditional Moves

- A family of instructions allows conditional movement of data into registers
- Can limit jumping in simple assignments

\[
\text{cmovq} \quad \%r8,\%r9 \\
\text{cmovge} \quad \%r11,\%r10 \quad # \text{if}(r9 \geq r8) \{ \ r10 = r11 \ \} \\
\text{cmovg} \quad \%r13,\%r12 \quad # \text{if}(r9 > r8) \{ \ r12 = r13 \ \}
\]

- Note that condition flags are set on arithmetic operations
- \text{cmovX} is like \text{subQ}: both set FLAG bits the same
- Greater than is based on the SIGN flag indicating subtraction would be negative allowing the following:

\[
\text{subq} \quad \%r8,\%r9 \quad # \ r9 = r9 - r8 \\
\text{cmovge} \quad \%r11,\%r10 \quad # \text{if}(r9 \geq 0) \{ \ r10 = r11 \ \} \\
\text{cmovg} \quad \%r13,\%r12 \quad # \text{if}(r9 > 0) \{ \ r12 = r13 \ \}
\]
Procedure Calls

Have seen basics so far:

main:

...  
call my_func  # call a function  
## arguments in %rdi, %rsi, %rdx, etc.  
## control jumps to my_func, returns here when done  
...

my_func:

## arguments in %rdi, %rsi, %rdx, etc.

...  
movl $0,%eax  # set up return value  
ret  # return from function  
## return value in %rax  
## returns control to wherever it came from

Need several additional notions

▶ Control Transfer to called function?
▶ Return back to calling function?
▶ Stack alignment and conventions
▶ Register conventions
Procedure Calls Return to Arbitrary Locations

- call instructions always transfer control to start of `return_seven` at line 4/5, like `jmp` instruction which modifies `%rip`

- ret instruction at line 6 must transfer control to different locations
  1. call-ed at line 11 ret to line 12
  2. call-ed at line 17 ret to line 18

ret cannot be a normal `jmp`

- To enable return to multiple places, record a Return Address when call-ing, use it when ret-turning

```
1 #### return_seven_asm.s
2 .text
3 .global return_seven
4 return_seven:
5     movl $7, %eax
6     ret  ## jump to line 12 OR 18??
7 .global main
8 main:
9     subq $8, %rsp
10
11    call return_seven  ## to line 5
12    leaq .FORMAT_1(%rip), %rdi
13    movl %eax, %esi
14    movl $0, %eax
15    call printf@PLT
16
17    call return_seven  ## to line 5
18    leaq .FORMAT_2(%rip), %rdi
19    movl %eax, %esi
20    movl $0, %eax
21    call printf@PLT
22
23    addq $8, %rsp
24    movl $0, %eax
25    ret
26 .data
27 .FORMAT_1: .asciz "first: %d\n"
28 .FORMAT_2: .asciz "second: %d\n"
```
call / ret with Return Address in Stack

**call Instruction**

1. Push the “caller” **Return Address** onto the stack
   Return address is for instruction after call
2. Change rip to first instruction of the “callee” function

**ret Instruction**

1. Set rip to Return Address at top of stack
2. Pop the Return Address off the stack shrinking stack

![Figure: Bryant/O’Hallaron Fig 3.26 demonstrates call/return in assembly](image)
### BEFORE CALL

return_seven:

```asm
0x555555555139 <return_seven>  mov $0x7,%eax
0x55555555513e <return_seven+5>  retq
```

main: ...

```asm
0x55555555513f <main>  sub $0x8,%rsp
=> 0x555555555143 <main+4>  callq 0x555555555139 <return_seven>
0x555555555148 <main+9>  lea 0x2ee1(%rip),%rdi
0x55555555514f <main+16>  mov %eax,%esi
```

(gdb) stepi

`rsp = 0x7fffffffde450 -> call -> 0x7fffffffde448 # push on return address`

`rip = 0x55555555555143 -> call -> 0x55555555555139 # jump control to procedure`

### AFTER CALL

return_seven:

```asm
=> 0x555555555139 <return_seven>  mov $0x7,%eax
0x55555555513e <return_seven+5>  retq
```

main: ...

```asm
0x55555555513f <main>  sub $0x8,%rsp
0x555555555143 <main+4>  callq 0x555555555139 <return_seven>
0x555555555148 <main+9>  lea 0x2ee1(%rip),%rdi
0x55555555514f <main+16>  mov %eax,%esi
```

(gdb) x/gx $rsp  # stack grew 8 bytes with call
`0x7fffffffde448: 0x00005555555555148 # return address in main on stack`
### BEFORE RET

```assembly
return_seven:
    0x555555555139 <return_seven>  mov $0x7,%eax
    => 0x55555555513e <return_seven+5> retq

main: ...
    0x55555555513f <main>        sub $0x8,%rsp
    0x555555555143 <main+4>      callq 0x555555555139 <return_seven>
    0x555555555148 <main+9>      lea 0x2ee1(%rip),%rdi
    0x55555555514f <main+16>     mov %eax,%esi
```

(gdb) x/gx $rsp
0x7fffffffe448: 0x0000555555555148  # return address pointed to by %rsp

(gdb) stepi  # EXECUTE RET INSTRUCTION
rsp = 0x7fffffffe448 -> ret -> 0x7fffffffe450  # pops return address off
rip = 0x55555555513e -> ret -> 0x555555555148  # sets %rip to return address

### AFTER RET

```assembly
return_seven:
    0x555555555139 <return_seven>  mov $0x7,%eax
    0x55555555513e <return_seven+5> retq

main: ...
    0x55555555513f <main>        sub $0x8,%rsp
    0x555555555143 <main+4>      callq 0x555555555139 <return_seven>
    => 0x555555555148 <main+9>    lea 0x2ee1(%rip),%rdi
    0x55555555514f <main+16>     mov %eax,%esi
```

(gdb) print $rsp  -->  $3 = 0x7fffffffe450
Warning: %rsp is important for returns

- When a function is about to return %rsp MUST refer to the memory location of the return address
- ret uses value pointed to %rsp as the return address
- Major problems arise if this is not so
- Using pushX / subq instructions to extend stack during a function MUST be coupled with popX / addq instructions
- There are computer security issues associated stack-based return value we will discuss later
Stack Alignment

- According to the strict x86-64 ABI, must align \texttt{rsp} (stack pointer) to 16-byte boundaries when calling functions.
- Will often see arbitrary pushes or subtractions to align.
  - Always enter a function with 8-byte Return Address on the stack.
  - Means that it is aligned to 8-byte boundary.
- \texttt{rsp} changes must be undone prior to return.

```assembly
main: # enter with at 8-byte boundary
    subq $8, %rsp # align stack for func calls
    ... 
    call sum_range # call function
    ... 
    addq $8, %rsp # remove rsp change
    ret
```

- Failing to align the stack may work but may break.
- Failing to “undo” stack pointer changes will likely result in return to the wrong spot: major problems.
x86-64 Register/Procedure Convention

► Used by Linux/Mac/BSD/General Unix
►Params and return in registers if possible

Parameters and Return
► First 6 arguments are put into
  1. rdi / edi / di (arg 1)
  2. rsi / esi / si (arg 2)
  3. rdx / edx / dx (arg 3)
  4. rcx / ecx / cx (arg 4)
  5. r8 / r8d / r8w (arg 5)
  6. r9 / r9d / r9w (arg 6)
► Additional arguments are pushed onto the stack
► Return Value in rax / eax / ...

Caller/Callee Save

**Caller save** registers: alter freely
rax rcx rdx rdi rsi
r8  r9  r10  r11

**Callee save** registers: must restore these on return
rbx rbp r12 r13 r14 r15

Careful messing with stack pointer
rsp # stack pointer
Pushing and Popping the Stack

- If local variables are needed on the stack, can use `push / pop` for these
- `pushX %reg`: grow `rsp` (lower value), move value to top of main memory stack,
  - `pushq %rax`: grows `rsp` by 8, puts contents of `rax` at top
  - `pushl $25`: grows `rsp` by 4, puts constant 5 at top of stack
- `popX %reg`: move value from top of main memory stack to `reg`, shrink `rsp` (higher value)
  - `popl %eax`: move (%rsp) to eax, shrink `rsp` by 4

```assembly
main:
  pushq %rbp # save register, aligns stack
  # like subq $8,%rsp; movq %rbp,(%rsp)
  call sum_range # call function
  movl %eax, %ebp # save answer
  ...
  call sum_range # call function, ebp not affected
  ...
  popq %rbp # restore rbp, shrinks stack
  # like movq (%rsp),%rbp; addq $8,%rsp
ret
```
Exercise: Local Variables which need an Address

Compare code in files

- swap_pointers.c: familiar C code for swap via pointers
- swap_pointers_asm.s: hand-coded assembly version

Determine the following

1. Where are local C variables $x,y$ stored in assembly version?
2. Where does the assembly version “grow” the stack?
3. How are the values in main() passed as arguments to swap_ptr()?
4. Where does the assembly version “shrink” the stack?
Exercise: Local Variables which need an Address

1 #include <stdio.h>
2
3 void swap_ptr(int *a, int *b){
4     int tmp = *a;
5     *a = *b;
6     *b = tmp;
7     return;
8 }
9
10 int main(int argc, char *argv[]){
11     int x = 19;
12     int y = 31;
13     swap_ptr(&x, &y);
14     printf("%d %d\n",x,y);
15     return 0;
16 }

1 .text
2 .global swap_ptr
3 swap_ptr:
4     movl (%rdi), %eax
5     movl (%rsi), %edx
6     movl %edx, (%rdi)
7     movl %eax, (%rsi)
8     ret
9 .global main
10 main:
11     subq $8, %rsp
12     movl $19, (%rsp)
13     movl $31, 4(%rsp)
14     movq %rsp, %rdi
15     leaq 4(%rsp), %rsi
16     call swap_ptr
17
18     leaq .FORMAT(%rip), %rdi
19     movl (%rsp), %esi
20     movl 4(%rsp), %edx
21     movl $0, %eax
22     call printf@PLT
23
24     addq $8, %rsp
25     movl $0, %eax
26     ret
27 .data
28 .FORMAT:
29 .asciz "%d %d\n"
Answers: Local Variables which need an Address

1. Where are local C variables x, y stored in assembly version?

2. Where does the assembly version “grow” the stack?

3. How are the values in main() passed as arguments to swap_ptr()? 

   \[
   \text{// C CODE} \\
   \text{int } x = 19, y = 31; \\
   \text{swap_ptr}(&x, &y) \ // \text{need main mem addresses for } x, y
   \]

   \[
   \text{### ASSEMBLY CODE} \\
   \text{main:} \ # \text{main()} \text{ function} \\
   \text{subq } $8, \%rsp \ # \text{grow stack by 8 bytes} \\
   \text{movl } $19, (\%rsp) \ # \text{move 19 to local variable x} \\
   \text{movl } $31, 4(\%rsp) \ # \text{move 31 to local variable y} \\
   \text{movq } \%rsp, \%rdi \ # \text{address of x into rdi, 1st arg to swap_ptr()} \\
   \text{leaq } 4(\%rsp), \%rsi \ # \text{address of y into rsi, 2nd arg to swap_ptr()} \\
   \text{call swap_ptr} \ # \text{call swap function}
   \]

4. Where does the assembly version “shrink” the stack?

   \[
   \text{addq } $8, \%rsp \ # \text{shrink stack by 8 bytes} \\
   \text{movl } $0, \%eax \ # \text{set return value} \\
   \text{ret}
   \]
Diagram of Stack Variables

- Compiler determines if local variables go on stack
- If so, calculates location as \( \text{rsp} + \text{offsets} \)

```c
// C Code: locals.c
int set_buf(char *b, int *s);
int main(){
  // locals re-ordered on
  // stack by compiler
  int size = -1;
  char buf[16];
  int x = set_buf(buf, &size);
}
```

---

<table>
<thead>
<tr>
<th>REG</th>
<th>VALUE</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsp</td>
<td>#1024</td>
<td>top of stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>during main</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>#1031</td>
<td>h</td>
<td>buf[3]</td>
</tr>
<tr>
<td>#1030</td>
<td>s</td>
<td>buf[2]</td>
</tr>
<tr>
<td>#1029</td>
<td>u</td>
<td>buf[1]</td>
</tr>
<tr>
<td>#1028</td>
<td>p</td>
<td>buf[0]</td>
</tr>
<tr>
<td>#1024</td>
<td>-1</td>
<td>size</td>
</tr>
</tbody>
</table>

---

```assembly
main:
  subq $24, %rsp          # space for buf/size and stack alignment
  movl $-1,(%rsp)         # old rip already in stack so: 20+4+8 = 32
  ....                    # initialize buf and size: main line 6
  leaq 0(%rsp), %rdi      # address of size arg1
  leaq 4(%rsp), %rsi      # address of buf arg2
  call set_buf            # call function, aligned to 16-byte boundary
  movl %eax,%r8           # get return value
  ....
  addq $24, %rsp          # shrink stack size
```
Summary of Procedure Calls: ABC() calls XYZ()

<table>
<thead>
<tr>
<th>Function</th>
<th>Caller</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC()</td>
<td></td>
<td>callq XYZ</td>
<td># ABC to XYZ</td>
</tr>
<tr>
<td>XYZ()</td>
<td></td>
<td>retq</td>
<td># XYZ to ABC</td>
</tr>
</tbody>
</table>

1. **ABC()** “saves” any Caller Save registers it needs by either copying them into Callee Save registers or pushing them into the stack.
2. **ABC()** places up to 6 arguments in %rsi, %rdi, %rdx, ..., remaining arguments in stack.
3. **ABC()** ensures that stack is “aligned”: %rsp contains an address that is evenly divisible by 16.
4. **ABC()** issues the callq ABC instruction which (1) grows the stack by subtracting 8 from %rsp and copies a return address to that location and (2) changes %rip to the starting address of func.
5. **XYZ()** now has control: %rip points to first instruction of **XYZ()**.
6. **XYZ()** may issue pushX val instructions or subq N,%rsp instructions to grow the stack for local variables.
7. **XYZ()** may freely change Caller Save registers BUT Callee Save registers it changes must be restored prior to returning.
8. **XYZ()** must shrink the stack to its original position via popX %reg or addq N,%rsp instructions before returning.
9. **XYZ()** sets %rax / %eax / %ax to its return value if any.
10. **XYZ()** finishes, issues the retq instruction which (1) sets the %rip to the 8-byte return address at the top of the stack (pointed to by %rsp) and (2) shrinks the stack by doing addq $8,%rsp.
11. **ABC()** function now has control back with %rip pointing to instruction after call **XYZ**; may have a return value in %rax register.
12. **ABC()** must assume all Caller Save registers have changed.
Historical Aside: Base Pointer `rbp` was Important

- 32-bit x86 / IA32 assembly used `rbp` as bottom of stack frame, `rsp` as top.
- Push all arguments onto the stack when calling changing both `rsp` and `rbp`.
- x86-64: default `rbp` to general purpose register, not used for stack tracking.

```c
int bar(int, int, int);
int foo(void) {
    int x = callee(1, 2, 3);
    return x + 5;
}
```

# Old x86 / IA32 calling sequence: set both `%esp` and `%ebp` for function call
```
foo:
pushl %ebp       # modifying ebp, save it
### Set up for function call to bar()
movl %esp,%ebp   # new frame for next function
pushl 3          # push all arguments to
pushl 2          # function onto stack
pushl 1          # no regs used
call bar         # call function, return val in %eax
### Tear down for function call bar()
movl %ebp,%esp   # restore stack top: args popped
### Continue with function foo()
addl 5,%eax      # add onto answer
popl %ebp        # restore previous base pointer
ret
```