CSCI 2021: x86-64 Control Flow

Chris Kauffman

Last Updated:
Sun Nov 6 02:06:04 PM CST 2022
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
▶ Stack Manipulation

P2 Grades Posted

Lab07 / HW07
▶ Assembly Coding and debugging
▶ Chance to configure assembly environment
▶ All techniques used in Project 3

Project 3
1. Battery ASM Functions
2. Binary Bomb via GDB

Overview Video Posted
Testing bugs fixed via make sanity-restore
Announcements

Unified Office Hours Tue 25-Oct @ Toaster, Walter Library

- 8am-6pm
- All in-person TA office hours there
- Basement of Walter Library, behind Wise Owl Cafe

Unified Office Hours Tue 01-Nov @ Keller 3-180

- 8am-6pm
- All in-person TA office hours there
- Just outside Atrium on ground level
Control Flow in Assembly and the Instruction Pointer

**Instruction Pointer Register**
- **%rip:** *special register* (not general purpose) referred to as the *Instruction Pointer* or Program Counter
- %rip contains main memory address of next assembly instruction to execute
- After executing an instruction, %rip automatically updates to the subsequent instruction
- OR in a Jump instruction, %rip changes non-sequentially
- **Do not** add/subtract with %rip via addq/subq: %rip automatically updates after each instruction

**Jump Instructions**
- **Labels** in assembly indicate jump targets like `.LOOP:`
- **Unconditional Jump:** always jump to a new location by changing %rip non-sequentially
- **Comparison / Test:** Instruction, sets EFLAGS bits indicating relation between registers/values (greater, less than, equal)
- **Conditional Jump:** Jumps to a new location if certain bits of EFLAGS are set by changing %rip non-sequentially; otherwise continues sequential execution
Exercise: 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 # COMPILE 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: each bit is True/False for a specific condition.
- 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>Like: A - B</td>
<td>cmpq %rsi,%rdi</td>
<td>Like if(rdi &gt; rsi){...}</td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td>testX B, A</td>
<td>testq %rcx,%rdx</td>
<td>Like if(rdx &amp; rcx){...}</td>
</tr>
<tr>
<td>Like: A &amp; B</td>
<td>testl %rax,%rax</td>
<td>Like if(rax){...}</td>
<td></td>
</tr>
</tbody>
</table>

- 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:

```c
movl $5, %eax  # 5 = 0b0101
cmpl $1, %eax  # [ ] 5-1=4 : No flags
cmpl $5, %eax  # [ZF ] 5-5=0 : Zero flag
cmpl $8, %eax  # [ SF] 5-8=-3 : Sign flag
testl $0b0110, %eax # [ ] 0101 & 0110 = 0100
testl $0b1010, %eax # [ZF ] 0101 & 1010 = 0000
```
Jump Instruction Summary

All control structures implemented using combination of Compare/Test + Jump instructions.

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

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

### Assembly translation of
```assembly
## if(rbx >= 2){
##   rdx = 10;
## }
## else{
##   rdx = 5;
## }
## return rdx;
```
```assembly
cmpq   $2,%rbx        # compare: rbx-2
jl    .LESSTHAN        # goto less than
## if(rbx >= 2){
movq   $10,%rdx       # greater/equal
## }
jmp    .AFTER          
```
`.LESSTHAN:
```assembly
## else{
movq   $5,%rdx        # less than
## }
```
`.AFTER:
```assembly
## rdx is 10 if rbx >= 2
## rdx is 5 otherwise
movq   %rdx,%rax
ret
```
Exercise: The test Instruction

```assembly
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 .NONNEGATIVE
    addl $50,%eax
.NONNEGATIVE:
    ret
```

- `testl %eax,%eax` uses bitwise AND to examine a register
- Selected by compiler to check for zero, NULL, negativity, etc.
- Followed by `je / jz / jne / jnz / js / jns`
- Demoed in `jmp_tests_asm.s`
- Trace the execution
- Determine final value in `%eax`
Answers: The test Instruction

```assembly
### From jmp_tests_asm_commented.s

```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  .NONNEGATIVE  # jump on !SF (not signed; e.g. positive)
```## if(ecx < 0){
```addl  $50,%eax  
```## }
```.

.NONNEGATIVE:
```ret  ## eax is return value```
Logical And / Or in Assembly

Logical boolean operators like a && b and x || y translate sequences of compare/test instructions followed by conditional jumps

```c
// andcond.c
int andcond(int edi){
    int ecx;
    if(edi >= 2 && edi <= 10){
        ecx = 10;
    }
    else{
        ecx = 5;
    }
    return ecx;
}
```

C Boolean expressions may “short circuit”: never execute code associated with later parts of the condition if early part resolves conditional

```asm
### andcond_asm.s
.text
.global andcond
andcond:
    cmpl $2,%edi        # compare: edi-2
    jl .ELSE            #
    cmpl $10,%edi       # compare: edi-10
    jg .ELSE            #
    ## if(edi >= 2 && edi <= 10){
    movl $10,%ecx       # greater/equal
    ## }
    jmp .AFTER
.ELSE:
    ## else{
    movl $5,%ecx        # less than
    ## }
.AFTER:
    movl %ecx,%eax
    ret
```
cmov Family: Conditional Moves

▶ Instruction family which copies data conditioned on FLAGS$^1$
▶ Can limit jumping in simple assignments

```plaintext
cmpq    %r8,%r9
cmovge  %r11,%r10  # if(r9 >= r8) { r10 = r11 }
cmovg   %r13,%r12  # if(r9 >  r8) { r12 = r13 }
```

▶ Note flags set on all Arithmetic Operations
▶ cmpX is like subQ: both set FLAG bits the same
▶ Greater than is based on the SIGN flag indicating subtraction would be negative allowing the following:

```plaintext
subq    %r8,%r9    # r9 = r9 - r8
cmovge  %r11,%r10  # if(r9 >=  0) { r10 = r11 }
cmovg   %r13,%r12  # if(r9 >   0) { r12 = r13 }
```

$^1$Other architectures like ARM have conditional versions of many instructions like addlt r1, r2, r3; RISC V ditches the FLAGS register in favor of jumps based on comparisons like BLT x0, x1, LOOP
Procedure Calls

Have seen basics so far:

main:

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

my_func:

```plaintext
## 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-urning

```assembly
## return_seven_asm.s
.text
.global return_seven
return_seven:
  movl $7, %eax
  ret  ## jump to line 12 OR 18??
.global main
main:
  subq $8, %rsp
  call return_seven  ## to line 5
  leaq .FORMAT_1(%rip), %rdi
  movl %eax, %esi
  movl $0, %eax
  call printf@PLT
  call return_seven  ## to line 5
  leaq .FORMAT_2(%rip), %rdi
  movl %eax, %esi
  movl $0, %eax
  call printf@PLT
  addq $8, %rsp
  movl $0, %eax
  ret
.data
.FORMAT_1: .asciz "first: %d\n"
.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 to shrink stack

---

**Figure:** Bryant/O'Hallaron Fig 3.26 demonstrates call/return in assembly
### BEFORE CALL

```
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) stePi
rSp = 0x7fffffffffe450  ->  call  ->  0x7fffffffffe448  # push on return address
rIp = 0x5555555555143  ->  call  ->  0x555555555139  # jump control to procedure
```

### AFTER CALL

```
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  # stack grew 8 bytes with call
0x7fffffffffe448: 0x0000555555555148  # return address in main on stack
### BEFORE RET

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
0x7fffffffffe448: 0x0000555555555148  # return address pointed to by %rsp

(gdb) stepi
# EXECUTE RET INSTRUCTION
rsp = 0x7fffffffffe448 -> ret -> 0x7fffffffffe450  # pops return address off
rip = 0x55555555513e -> ret -> 0x555555555148  # sets %rip to return address

### AFTER RET

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 = 0x7fffffffffe450
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
- Segmentation Faults often occur if %rsp is NOT the return address: attempt to fetch/execute instructions out of bounds
- Stack is often used to store local variables, stack pointer %rsp is manipulated via pushX / subq instructions to grow the stack.
- Before returning MUST shrink stack and restore %rsp to its original value via popX / addq instructions
- There are computer security issues associated stack-based return value we will discuss later
### return_seven_buggy_asm.s

```
.text
.global return_seven
return_seven:
    pushq $0x42
    # push but no pop before returning
    movl $7, %eax
    # %rsp points to a 0x42 return address - BAD!
    ret
```

<table>
<thead>
<tr>
<th>REG</th>
<th>VALUE</th>
<th>ADDRESS</th>
<th>VALUE</th>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>---------</td>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>rax</td>
<td>7</td>
<td>0x77128</td>
<td>0x554210</td>
<td>Ret Address</td>
</tr>
<tr>
<td>rsp</td>
<td>0x77120</td>
<td>---&gt;</td>
<td>0x77120</td>
<td>0x42</td>
</tr>
</tbody>
</table>

> gcc -g return_seven_buggy_asm.s

> ./a.out
Segmentation fault (core dumped)  ## definitely a memory problem

> valgrind ./a.out  ## get help from Valgrind
...  
==2664132== Jump to the invalid address stated on the next line
==2664132== at 0x42: ???
==2664132== by 0x109149: ??? (return_seven_buggy_asm.s:18)
==2664132== Address 0x42 is not stack'd, malloc'd or (recently) free'd

Valgrind reports like this often indicate clobbering a return address as happened here through unbalanced push/pop instructions.
Stack Alignment

- According to the strict x86-64 ABI, must align \( \text{rsp} \) (stack pointer) to 16-byte boundaries when calling functions
- Will often see arbitrary pushes or subtractions to align
  - Functions called with 16-byte alignment
  - call pushes 8-byte Return Address on the stack
  - At minimum, must grow stack by 8 bytes to call again
- \( \text{rsp} \) 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

<table>
<thead>
<tr>
<th>RetVal</th>
<th>rax / eax / ax / al</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arg 1</td>
<td>rdi / edi / di / dil</td>
</tr>
<tr>
<td>Arg 2</td>
<td>rsi / esi / si / sil</td>
</tr>
<tr>
<td>Arg 3</td>
<td>rdx / edx / dx / dl</td>
</tr>
<tr>
<td>Arg 4</td>
<td>rcx / ecx / cx / cl</td>
</tr>
<tr>
<td>Arg 5</td>
<td>r8 / r8d / r8w / r8b</td>
</tr>
<tr>
<td>Arg 6</td>
<td>r9 / r9d / r9w / r9b</td>
</tr>
<tr>
<td>Arg 7</td>
<td>Push into the stack</td>
</tr>
<tr>
<td>Arg 8</td>
<td>Push into the stack</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Caller/Callee Save

**Caller save** registers: alter freely

rax rcx rdx rdi rsi
r8  r9  r10 r11 # 9 regs

**Callee save** registers: must restore these before returning

rbx rbp r12 r13 r14 r15 # 6 regs

**Stack Pointer**: special considerations discussed in detail

rsp # 1 reg

C function prototype indicates number, order, type of args so it is known which registers args will be in

```c
int myfunc(char *cp,
           int a, long b);
```
Caller and Callee Save Register Mechanics

main:       # main: the calleR
...
        movq $21, %rdi       # calleR save arg 1
        movq $31, %rsi       # calleR save arg 2
        movq $41, %r10       # calleR save
        movq $7, %rbx        # calleE save
        movq $11, %r12       # calleE save

call foo  # foo: the calleE

## | ? | %rdi | calleR save arg 1 |
## | ? | %rsi | calleR save arg 2 |
## | ? | %r10 | calleR save |
## | 7 | %rbx | calleE save |
## | 11 | %r12 | calleE save |

        cmpq $21, %rdi       # unpredictable
        cmpq $7, %rbx        # predictably equal

# main MUST restore %rbx and %r12 to
# original values as function above
# main() expects them to be unchanged

CalleR Save Regs
May all change across function call boundaries.
Not a problem for Leaf Functions which do not call any other funcs

CalleE Save Regs
Have the same values in them after a function call
Using them requires saving their original values in the stack and restoring them

sumrange_asm.s
Full example of callee save regs like sumrange_c.c
Pushing and Popping the Stack

- If local variables or callee save regs are needed on the stack, can use push / pop for these
- Push and Pop Instructions are compound: manipulate %rsp and move data in single instruction

| pushX data | Grow Stack, store data at top |
| pushq %rax | Like: subq $8,%rsp; movq %rax, (%rsp) |
| pushl $24 | Like: subq $4,%rsp; movq $25, (%rsp) |

| popX data | Shrink Stack, restore data from it |
| popl %edi | Like: movl (%rsp),%edi; addq $4,%rsp; |
| popq %rax | Like: movq (%rsp),%rax; addq $8,%rsp; |

**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 // swap_pointers.c
2 #include <stdio.h>
3
4 void swap_ptr(int *a, int *b){
5     int tmp = *a;
6     *a = *b;
7     *b = tmp;
8     return;
9 }
10
11 int main(int argc, char *argv[]){
12     int x = 19;
13     int y = 31;
14     swap_ptr(&x, &y);
15     printf("%d %d\n",x,y);
16     return 0;
17 }

1 # swap_pointers_asm.s
2 .text
3 .global swap_ptr
4 swap_ptr:
5     movl (%rdi), %eax
6     movl (%rsi), %edx
7     movl %edx, (%rdi)
8     movl %eax, (%rsi)
9     ret
10
11 .global main
12 main:
13     subq $8, %rsp
14     movl $19, (%rsp)
15     movl $31, 4(%rsp)
16     movq %rsp, %rdi
17     leaq 4(%rsp), %rsi
18     call swap_ptr
19     leaq .FORMAT(%rip), %rdi
20     movl (%rsp), %esi
21     movl 4(%rsp), %edx
22     movl $0, %eax
23     call printf@PLT
24
25     addq $8, %rsp
26     movl $0, %eax
27     ret
28
29 .data
30 .FORMAT:
31     .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()`?

```c
// C CODE
int x = 19, y = 31;
swap_ptr(&x, &y) // need main mem addresses for x, y
```

### ASSEMBLY CODE

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

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

```assembly
addq $8, %rsp      # shrink stack by 8 bytes
movl $0, %eax      # set return value
ret
```
Diagram of Stack Variables

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

C Code: locals.c

```c
1 // locals.c
2 int set_buf(char *b, int *s);
3 int main(){
4   // locals re-ordered on stack by compiler
5   // stack by compiler
6   int size = -1;
7   char buf[16];
8   ...
9   int x = set_buf(buf, &size);
10  ...
11 }
```

---

#### EQUIVALENT ASSEMBLY

```
main:
1  subq $24, %rsp       # space for buf/size and stack alignment
2  movl $-1,(%rsp)     # retAddr:8, locals: 20, padding: 4, tot: 32
3  ....                # initialize buf and size: main line 6
4  leaq 4(%rsp), %rdi   # address of buf arg1
5  leaq 0(%rsp), %rsi   # address of size arg2
6  call set_buf         # call function, aligned to 16-byte boundary
7  movl %eax, %r8       # get return value
8  ....                # ...
9  addq $24, %rsp       # shrink stack size
```
Summary of Procedure Calls: ABC() calls XYZ()

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Caller</th>
<th>Instruction</th>
<th>Notes</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>Callee</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.
History: Base Pointer \texttt{rbp} was Special Use

- 32-bit x86 / IA32 assembly used \texttt{rbp} and \texttt{rsp} to describe stack frames
- All function args pushed onto the stack when calling, changes both \texttt{rsp} and \texttt{rbp}
- x86–64: default \texttt{rbp} to general purpose register, not used for stack purposes

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

```assembly
# Old x86 / IA32 calling sequence: set both \%esp and \%ebp for function call
# Push all arguments into the stack
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
```