CSCI 2021: x86-64 Control Flow

Chris Kauffman

Last Updated:
Wed 18 Mar 2020 10:26:56 AM CDT
New Logistics

Reading Bryant/O’Hallaron

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

Goals

- Finish Assembly Basics
- Jumps and Control flow
  x86-64
- Procedure calls

Project 3

- Problem 1: Clock Assembly Functions (50%)
- Problem 2: Binary Bomb via debugger (50%)

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed 3/04</td>
<td>Asm Control Flow</td>
</tr>
<tr>
<td>Fri 3/06</td>
<td>Asm Control</td>
</tr>
<tr>
<td>Mon 3/09</td>
<td>Spring Break</td>
</tr>
<tr>
<td>Wed 3/18</td>
<td>Asm Control</td>
</tr>
<tr>
<td>Fri 3/20</td>
<td>Asm Wrap-up</td>
</tr>
<tr>
<td>Mon 3/23</td>
<td>Review</td>
</tr>
<tr>
<td>Tue 3/24</td>
<td>P3 Due</td>
</tr>
<tr>
<td>Wed 3/25</td>
<td>Exam 2</td>
</tr>
</tbody>
</table>
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  # COMPILATE 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>
<tr>
<td>Test</td>
<td>testX B, A</td>
<td>testq %rcx,%rdx</td>
<td>Like if(rdx &amp; rcx){...}</td>
</tr>
<tr>
<td></td>
<td>Like: A &amp; B</td>
<td>testl %rax,%rax</td>
<td>Like if(rax){...}</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:

```assembly
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>jmp LAB</td>
<td>Unconditional jump</td>
</tr>
<tr>
<td>je LAB</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>jz LAB</td>
<td>Not equal / non-zero</td>
</tr>
<tr>
<td>jne LAB</td>
<td>Negative (&quot;signed&quot;)</td>
</tr>
<tr>
<td>jnz LAB</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>js LAB</td>
<td>Greater-than signed</td>
</tr>
<tr>
<td>jns LAB</td>
<td>Greater-than-equal signed</td>
</tr>
<tr>
<td>jl LAB</td>
<td>Less-than signed</td>
</tr>
<tr>
<td>jle LAB</td>
<td>Less-than-equal signed</td>
</tr>
<tr>
<td>ja LAB</td>
<td>Above unsigned</td>
</tr>
<tr>
<td>jae LAB</td>
<td>Above-equal unsigned</td>
</tr>
<tr>
<td>jb LAB</td>
<td>Below unsigned</td>
</tr>
<tr>
<td>jbe LAB</td>
<td>Below-equal unsigned</td>
</tr>
<tr>
<td>jmp *OPER</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
  - \(\text{if()}\{ \ldots \}\) usually has a single jump
  - \(\text{if()}\{\}\) else \(\{\}\) may have a couple

```
## Assembly translation of
## if(rbx >= 2){
##   rdx = 10;
## }  
## else{
##   rdx = 5;
## }
## return rdx;

cmpq $2,%rbx  # compare: rbx-0
j1 .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`
cmov Family: Conditional Moves

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

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

- Note that condition flags are set on 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:

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

Have seen basics so far:

```asm
call PROCNAME  # call a function
## Pushes return address %rip onto stack adjusting %rsp

movl  $0,%eax  # set up return value
ret             # return from function
## Pops old %rip off of stack adjusting %rsp
```

Need several additional notions

- Control Transfer?
- Where are arguments to functions?
- Return value?
- Anything special about the registers?
- How is the stack used?
Function/Procedure Control Transfer

**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
Example: Control Transfer with call

Example derived from sum_range.s file in code pack

### BEFORE CALL
main: ...

0x555555554687 <+11>: mov $0x5,%esi

=> 0x55555555468c <+16>: callq 0x55555555466a <sum_range>

0x555555554691 <+21>: mov %eax,%ebx

rip = 0x55555555468c -> call -> 0x555555554691

rsp = 0x7fffffffe460

(gdb) steopi

### AFTER CALL
sum_range:

=> 0x55555555466a <+0>: mov $0x0,%eax

0x55555555466f <+5>: jmp 0x555555554676 <.TOP>

rip = 0x55555555466a

rsp = 0x7fffffffde458 # pushed return address: rsp -= 8

(gdb) x/xg $rsp
0x7fffffffde458: 0x555555554691 # return address in main
Control Transfer with \texttt{ret}

### BEFORE RET:

\texttt{sum\_range:...}

\begin{itemize}
  \item 0x555555554678 <+2>: jle 0x555555554671 <.BODY>
  \item => 0x55555555467a <+4>: repz retq
\end{itemize}

\begin{itemize}
  \item rip = 0x555555555467a \rightarrow \text{return}
  \item rsp = 0x7fffffffffe458
\end{itemize}

\begin{verbatim}
(gdb) x/xg $rsp
0x7fffffffffe458: 0x5555555554691 # return address in main
\end{verbatim}

\begin{verbatim}
(gdb) stepi
\end{verbatim}

### AFTER RET

\begin{itemize}
  \item 0x555555554687 <+11>: mov $0x5,%esi
  \item 0x55555555468c <+16>: callq 0x555555555466a <sum\_range>
  \item => 0x555555554691 <+21>: mov %eax,%ebx
\end{itemize}

\begin{itemize}
  \item rip = 0x5555555554691
  \item rsp = 0x7fffffffffe460 # popped return address: \texttt{rsp} += 8
\end{itemize}
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
  - Always enter a function with 8-byte Return Address on the stack
  - Means that it is aligned to 8-byte boundary
- \( \text{rsp} \) changes must be undone prior to return

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

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

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

   ### ASSEMBLY CODE
   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 # load address of x into rdi
       leaq 4(%rsp), %rsi # load address of y into rsi
       call swap_ptr # call swap function

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

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

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

```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);
    ...
}
```

| REG  | VALUE | Name |
|-------+-------+--------------|
| $rsp$ | #1024 | top of stack |
|       |       | during main   |

```
## EQUIVALENT 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
```
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:

```assembly
foo:
    pushl %ebp # modifying ebx, 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
```