CSCI 2021: x86-64 Control Flow

Chris Kauffman

Last Updated:
Fri 18 Oct 2019 11:40:36 AM CDT

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

Assignment 3

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

Date	Event
Mon 10/14	Asm Control Flow
	A3 Released
Wed 10/16	Discuss A3
Fri 10/18	Asm Control Wrap
Mon 10/21	Asm Wrap-up
Wed 10/23	Review
Fri 10/25	**Exam 2**
Mon 10/28	CPU Architecture
Tue 10/29	A3 Due

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

```
> 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
- In x86-64 this register goes by the following names

Name	Width	Notes
`FLAGS`	16-bit	Most important bits in first 16
`EFLAGS`	32-bit	Name shown in `gdb`
`RFLAGS`	64-bit	Not used normally

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

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

Comparisons and Tests

Set the `EFLAGS` register by using comparison instructions

Name	Instruction	Examples	Notes
Compare	`cmpX B, A`	`cmpl $1,%eax`	Like `if(eax > 1){...}`
	Like: `A - B`	`cmpq %rsi,%rdi`	Like `if(rdi > rsi){...}`
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

Instruction	*Jump Condition*
`jmp LAB`	Unconditional jump
`je LAB`	Equal / zero
`jz LAB`	
`jne LAB`	Not equal / non-zero
`jnz LAB`	
`js LAB`	Negative ("signed")
`jns LAB`	Nonnegative
`jg LAB`	Greater-than signed
`jge LAB`	Greater-than-equal signed
`jl LAB`	Less-than signed
`jle LAB`	Less-than-equal signed
`ja LAB`	Above unsigned
`jae LAB`	Above-equal unsigned
`jb LAB`	Below unsigned
`jbe LAB`	Below-equal unsigned
`jmp *OPER`	Unconditional jump to variable address

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

`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:

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

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

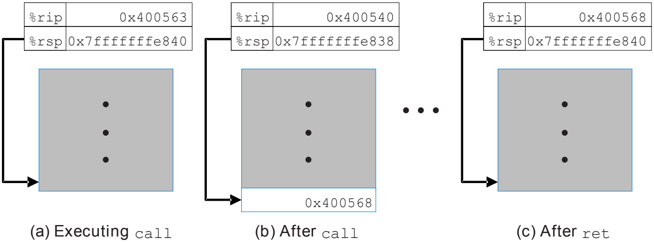

Figure: Bryant/O'Hallaron Fig 3.26 demonstrates call/return in assembly

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) stepi

### AFTER CALL
sum_range:
=> 0x55555555466a <+0>: mov    $0x0,%eax
   0x55555555466f <+5>: jmp    0x555555554676 <.TOP>

rip = 0x55555555466a
rsp = 0x7fffffffe458     # pushed return address: rsp -= 8
(gdb) x/xg $rsp
0x7fffffffe458: 0x555555554691 # return address in main
```

Control Transfer with `ret`

```
### BEFORE RET:
sum_range:...
   0x555555554678 <+2>: jle    0x555555554671 <.BODY>
=> 0x55555555467a <+4>: repz retq

rip = 0x55555555467a -> return
rsp = 0x7fffffffe458
(gdb) x/xg $rsp
0x7fffffffe458: 0x555555554691 # return address in main

(gdb) stepi

### AFTER RET
   0x555555554687 <+11>:       mov    $0x5,%esi
   0x55555555468c <+16>:       callq  0x55555555466a <sum_range>
=> 0x555555554691 <+21>:       mov    %eax,%ebx

rip = 0x555555554691
rsp = 0x7fffffffe460    # popped return address: rsp += 8
```

Stack Alignment

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

```
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

```
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` + offsets

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

REG	VALUE	Name
rsp	#1024	top of stack during main
MEM		
...	...	...
#1031	h	buf[3]
#1030	s	buf[2]
#1029	u	buf[1]
#1028	p	buf[0]
#1024	-1	size

```
1  ## EQUIVALENT ASSEMBLY
2  main:
3      subq    $24, %rsp        # space for buf/size and stack alignment
4      movl    $-1,(%rsp)       # old rip already in stack so: 20+4+8 = 32
5      ....                     # initialize buf and size: main line 6
6      leaq    0(%rsp), %rdi    # address of size arg1
7      leaq    4(%rsp), %rsi    # address of buf  arg2
8      call    set_buf          # call function, aligned to 16-byte boundary
9      movl    %eax,%r8         # get return value
10     ...
11     addq    $24, %rsp        # shrink stack size
```

Historical Aside: Base Pointer `rbp` was Important

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

- 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

```
# 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
```