CSCI 2021: x86-64 Assembly Extras and Wrap

Chris Kauffman

Last Updated:
Wed Mar 15 11:44:23 AM CDT 2023

Logistics

Reading Bryant/O'Hallaron

Read in Full

- ▶ Ch 3.7 Procedure Calls

Skim the following

- ▶ Ch 3.8-3.9: Arrays, Structs
- ▶ Ch 3.10: Pointers/Security
- ▶ Ch 3.11: Floating Point

Goals

- ☐ Finish Assembly Control
- ☐ Assembly vs C
- ☐ Data in Assembly
- ☐ Security Risks
- ☐ Floating Point Instr/Regs

Date	Event
Wed 15-Mar	Asm Extras
	Lab08: Asm Stack
Fri 17-Mar	Asm Extras
Mon 20-Mar	Asm Wrap-up
Tue 21-Mar	Unified OH
	Keller 3-180
Wed 22-Mar	Proc Arch
	Lab09: Timing
	P3 Due
Fri 24-Mar	Proc Arch
Mon 27-Mar	Proc Arch
Wed 29-Mar	Practice Exam 2
	Lab10: Review
Fri 31-Mar	Exam 2

Lab08 / HW08

- ▶ Stack and Function Calls
- ▶ "Stack Smashing"
- ▶ Binary Debugging

Announcements

Pi a Professor Fund Raiser

▶ $1.50 to vote on professors to pie in the face

▶ Proceeds to support K-12 STEM Education

▶ Cast Votes: https://z.umn.edu/PieAProf23

P3 Support in Lind 325

Date	Event
Tue 14-Mar 6pm	Tutorial Session
Wed 15-Mar 6pm	Tutorial Session
Thu 16-Mar 6pm	Tutorial Session
Tue 21-Mar 9-5pm	Unified Office Hours
Wed 22-Mar 11:59pm	P3 Due

Reminders of Techniques for Binary Bomb

GDB Tricks from Quick Guide to GDB

Command	Effect
break *0x1248f2	Break at specific instruction address
break *func+24	Break at instruction with decimal offset from a label
break *func+0x18	Break at instruction with hex offset from a label
x $rax	Print memory pointed to by register rax
x /gx $rax	Print as "giant" 64-bit numbers in hexadecimal format
x /5gd $rax	Print 5 64-bit numbers starting where rax points in decimal format

Disassembling Binaries: objdump -d prog > code.txt

```
>> objdump -d a.out              # DISASSEMBLE BINARY
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
...
>> objdump -d a.out > code.txt  # STORE RESULTS IN FILE
```

Exercise: All Models are Wrong...

- Rule #1: The Doctor Lies
- Below is our original model for memory layout of C programs
- Describe what is **incorrect** based on x86-64 assembly
- What is **actually** in the stack? How are registers likely used?

```
 9: int main(...){
10:    int x = 19;
11:    int y = 31;
12:    swap(&x, &y);
13:    printf("%d %d\n",x,y);
14:    return 0;
15: }

18: void swap(int *a,int *b){
19:    int tmp = *a;
20:    *a = *b;
21:    *b = tmp;
22:    return;
23: }
```

```
STACK: Caller main(), prior to swap()
| FRAME   | ADDR  | NAME | VALUE |
|---------+-------+------+-------|
| main()  | #2048 | x    |    19 |
| line:12 | #2044 | y    |    31 |
|---------+-------+------+-------|

STACK: Callee swap() takes control
| FRAME   | ADDR  | NAME | VALUE |
|---------+-------+------+-------|
| main()  | #2048 | x    |    19 |<-+
| line:12 | #2044 | y    |    31 |<-|+
|---------+-------+------+-------|  ||
| swap()  | #2036 | a    | #2048 |--+|
| line:19 | #2028 | b    | #2044 |---+
|         | #2024 | tmp  |     ? |
```

Answers: All Models are Wrong, Some are Useful

```
    9: int main(...){
   10:   int x = 19;
   11:   int y = 31;
+-<12:   swap(&x, &y);
|  13:   printf("%d %d\n",x,y);
|  14:   return 0;
V  15: }
|
|  18: void swap(int *a,int *b){
+->19:   int tmp = *a;
   20:   *a = *b;
   21:   *b = tmp;
   22:   return;
   23: }
```

```
STACK: Callee swap() takes control
| FRAME   | ADDR  | NAME | VALUE |
|---------+-------+------+-------|
| main()  | #2048 | x    |    19 |
|         | #2044 | y    |    31 |
|---------+-------+------+-------|
| swap()  | #2036 | rip  |Line 13|
|---------+-------+------+-------|
REGS as swap() starts
| REG | VALUE | NOTE          |
|-----+-------+---------------|
| rdi | #2048 | for *a        |
| rsi | #2044 | for *b        |
| rax |     ? | for tmp       |
| rip |   L19 | line in swap  |
```

▶ main() must have stack space for locals passed by address

▶ swap() needs no stack space for arguments: in registers

▶ Return address is next value of rip register in main()

▶ Mostly don't need to think at this level of detail but **can be useful in some situations**

Accessing Global Variables in Assembly

Global data can be set up in assembly in `.data` sections with
labels and assembler directives like `.int` and `.short`

```
.data
an_int:                 # single int
        .int 17
some_shorts:            # array of shorts
        .short 10       # some_shorts[0]
        .short 12       # some_shorts[1]
        .short 14       # some_shorts[2]
```

Modern Access to Globals

```
movl an_int(%rip), %eax
leaq some_shorts(%rip), %rdi
```

- ▶ Uses `%rip` relative addressing

- ▶ Default in gcc as it plays nice
  with OS security features

- ▶ May discuss again later during
  Linking/ELF coverage

Traditional Access to Globals

```
movl an_int, %eax          # ERROR
leaq (some_shorts), %rdi   # ERROR
```

- ▶ Not accepted by gcc by default

- ▶ Yields compile/link errors

```
/usr/bin/ld: /tmp/ccocSiw5.o:
  relocation R_X86_64_32S against `.data'
  can not be used when making a PIE object;
  recompile with -fPIE
```

Aggregate Data In Assembly (Arrays + Structs)

Arrays

Usually: base + index × size

```
arr[i] = 12;
movl $12,(%rdi,%rsi,4)

int x = arr[j];
movl (%rdi,%rcx,4),%r8d
```

- ▶ Array starting address often held in a register
- ▶ Index often in a register
- ▶ Compiler inserts appropriate size (1,2,4,8)

Structs

Usually base+offset

```
typedef struct {
  int i; short s;
  char c[2];
} foo_t;
foo_t *f = ...;

short sh = f->s;
movw 4(%rdi),%si

f->c[i] = 'X';
movb $88, 6(%rdi,%rax)
```

Packed Structures as Procedure Arguments

▶ Passing pointers to structs is 'normal': registers contain addresses to main memory

▶ Passing actual structs may result in *packed structs* where several fields are in a single register

▶ Assembly must *unpack* these through **shifts and masking**

```c
1 // packed_struct_main.c
2 typedef struct {
3   short first;
4   short second;
5 } twoshort_t;
6
7 short sub_struct(twoshort_t ti);
8
9 int main(){
10   twoshort_t ts = {.first=10,
11                    .second=-2};
12   int sum = sub_struct(ts);
13   printf("%d - %d = %d\n",
14          ts.first, ts.second, sum);
15   return 0;
16 }
```

```asm
1 ### packed_struct.s
2 .text
3 .globl  sub_struct
4 sub_struct:
5   ## first arg is twoshort_t ts
6   ## %rdi has 2 packed shorts in it
7   ## bits  0-15 are ts.first
8   ## bits 16-31 are ts.second
9   ## upper bits could be anything
10
11   movl %edi,%eax       # eax = ts;
12   andl $0xFFFF,%eax    # eax = ts.first;
13   sarl $16,%edi        # edi = edi >> 16;
14   andl $0xFFFF,%edi    # edi = ts.second;
15   subw %di,%ax         # ax = ax - di
16   ret                  # answer in ax
```

Example: `coins_t` in HW06 / Lab07

```c
// Type for collections of coins
typedef struct {     // coint_t has the following memory layout
  char quarters;     //
  char dimes;        // |              | Pointer | Packed | Packed |
  char nickels;      // |              | Memory  | Struct | Struct |
  char pennies;      // | Field        | Offset  | Arg#   | Bits   |
} coins_t;           // |--------------+---------+--------+--------|
                     // | quarters     |   +0    | #1     | 0-7    |
                     // | dimes        |   +1    | #1     | 8-15   |
                     // | nickels      |   +2    | #1     | 16-23  |
                     // | pennies      |   +3    | #1     | 24-31  |
```

```
## | #2048 | c->quarters | 2 |
## | #2049 | c->dimes    | 1 |
## | #2050 | c->nickels  | - |
## | #2051 | c->pennies  | - |
```

```
set_coins:
### int set_coins(int cents, coins_t *coins)
### %edi = int cents
### %rsi = coints_t *coins
 ...
 # rsi: #2048
 # al: 0  %dl: 3
 movb   %al,2(%rsi)     # coins->nickels = al;
 movb   %dl,3(%rsi)     # coins->pennies = dl;
```

```
## | #2048 | c->quarters | 2 |
## | #2049 | c->dimes    | 1 |
## | #2050 | c->nickels  | 0 |
## | #2051 | c->pennies  | 3 |
```

```
total_coins:
### args are
### %rdi packed coin_t struct with struct fields
### { 0- 7: quarters, 8-15: dimes,
###  16-23: nickels, 24-31: pennies}
 ...

### rdi: 0x00 00 00 00 03 00 01 02
###                      p  n  d  q
  movq   %rdi,%rdx        # extract dimes
### rdx: 0x00 00 00 00 03 00 01 02
###                      p  n  d  q
  sarq   $8,%rdx          # shift dimes to low bits
### rdx: 0x00 00 00 00 00 03 00 01
###                         p  n  d
  andq   $0xFF,%rdx       # rdx = dimes
### rdx: 0x00 00 00 00 00 00 00 01
###                            p  n  d
```

10

Large Packed Structs

- ▶ Large structs that don't fit into single registers may be packed across several argument registers
- ▶ This is the case in P3

```c
typedef struct{
  int   day_secs;  // 4
  short time_secs; // 2
  short time_mins; // 2
  short time_hours;// 2
  char  ampm;      // 1+1 pad
} tod_t;           // 12 bytes
```

```c
int set_display_from_tod(tod_t tod, ...)
//                       ^^^ Large packed struct
```

C Field Access	Register	Bits in reg	Shift Required	Size
tod.day_secs	%rdi	0-31	None	4 bytes
tod.time_secs	%rdi	32-47	Right by 32	2 bytes
tod.time_mins	%rdi	48-63	Right by 48	2 bytes
tod.time_hours	%rsi	0-15	None	2 bytes
tod.ampm	%rsi	16-23	Right by 16	1 bytes

General Cautions on Structs

Struct Layout by Compilers

- ▶ Compiler honors order of source code fields in struct
- ▶ BUT compiler may add padding between/after fields for alignment
- ▶ Compiler determines total struct size

Struct Layout Algorimths

- ▶ Baked into compiler
- ▶ **May change from compiler to compiler**
- ▶ May change through history of compiler

Structs in Mem/Regs

- ▶ Local var structs spread across several registers
- ▶ Don't need a struct on the stack at all in some cases (just like don't need local variables on stack)
- ▶ Struct arguments packed into 1+ registers

Stay Insulated

- ▶ Programming in C insulates you from all of this
- ▶ Feel the **warmth** of gcc's abstraction blanket

Security Risks in C

Buffer Overflow Attacks

- ▶ No default bounds checking in C: Performance favored over safety

- ▶ Allows classic security flaws:

```c
char buf[1024];
printf("Enter you name:");
fscanf(file,"%s",buf); // BAD
// or
gets(buf);             // BAD
// my name is 1500 chars
// long, what happens?
```

- ▶ For data larger than buf, begin overwriting other parts of the stack

  - ▶ Clobber return addresses
  - ▶ Insert executable code and run it

Counter-measures

- ▶ **Stack protection** is default in gcc in the modern era

- ▶ Inserts "canary" values on the stack near return address

- ▶ Prior to function return, checks that canaries are unchanged

- ▶ **Stack / Text Section Start randomized** by kernel, return address and function addresses difficult to predict ahead of time

- ▶ Kernel may also vary virtual memory address as well

- ▶ Disabling protections is risky

13

Stack Smashing

- ▶ Explored in a recent homework
- ▶ See stack_smash.c for a similar example
- ▶ Demonstrates detection of changes to stack that could be harmful / security threat

```
void demo(){
  int arr[4];    // fill array off the end
  for(int i=0; i<8; i++){
    arr[i] = (i+1)*2;
  }

  for(int i=0; i<4; i++){
    printf("[%d]: %d\n",i,arr[i]);
  }
}

int main(){
  printf("About to do the demo\n");
  demo();
  printf("Demo Complete\n");
  return 0;
}
```

```
> cd 08-assembly-extras-code/
> gcc stack_smash1.c
> ./a.out
About to do the demo
[0]: 2
[1]: 4
[2]: 6
[3]: 8
*** stack smashing detected ***: terminat
Aborted (core dumped)
```

14

Demonstration of Buffer Overflow Attack

▶ See the code `buffer_overflow.c`

▶ Presents an easier case to demo stack manipulations

▶ Prints addresses of functions `main()` and `never()`

▶ Reads `long` values which are 64-bits, easier to line up data in stack than with strings; still overflowing the buffer by reading too much data as in:

```c
void always(){
  long buf[1] = {0xABCD};        // room for 1
  ...
  printf("Enter 4 hex values: ");
  fscanf(stdin,"%lx %lx %lx %lx", // reads 4
         &buf[0], &buf[1], &buf[2], &buf[3]);
```

▶ When compiled via

>> gcc -fno-stack-protector buffer_overflow.c

can get `never()` to run by entering its address as input which will overwrite the return address

Sample Buffer Overflow Code

```c
#include <stdio.h>
void print_all_passwords(){
  ...
}
int main(){
  printf("file to open: ");
  char buf[128];
  fscanf(stdin,"%s",buf);
  printf("You entered: %s\n",buf);

  ...;
  return 0;
  // By entering the correct length of string followed by the ASCII
  // representation of the address of print_all_passwords(), one might
  // be able to get that function when "return" is reached if there
  // are no stack protection mechanisms at work ...
  // (which was the case in 1999 on Windows :-)
}
```

Details of GCC / Linux Stack Security

▶ Programs compiled with GCC + Glibc on Linux for x86-64 will default to having stack protection

▶ This is can be seen in compiled code as short blocks near the beginning and end of functions which

  1. At the beginning of the function uses an instruction like `movq %fs:40, %rax` and places a value in the stack beneath the return address
  2. At the end of the function again accesses `%fs:40` and the value earlier placed in the stack.

▶ The `%fs` register is a special segment register originally introduced in the 16-bit era to surmount memory addressing limitations; now used only for limited purposes

▶ The complete details are beyond the scope of our course BUT

▶ **A somewhat detailed explanation has been added to** `08-assembly-extras-code/stack_protect.org`

Floating Point Operations

- ▶ Original Intel 8086 Processor **didn't do floating point ops**
- ▶ Had to buy a co-processor (Intel 8087) to enable FP ops
- ▶ Most modern CPUs support FP ops but they feel separate from the integer ops: FPU versus ALU

x86-64 "Media" Registers

512	256	128-bits	Use
%zmm0	%ymm0	%xmm0	FP Arg1/Ret
%zmm1	%ymm1	%xmm1	FP Arg2
...	...	...	...
%zmm7	%ymm7	%xmm7	FP Arg 8
%zmm8	%ymm8	%xmm8	Caller Save
...	...	...	...
%zmm15	%ymm15	%xmm15	Caller Save

- ▶ Can be used as "scalars" - single values but…
- ▶ xmmI is 128 bits big holding
  - ▶ $2 \times$ 64-bit double's OR
  - ▶ $4 \times$ 32-bit float's
- ▶ ymmI / zmmI extend further

Instructions

```
vaddss %xmm2,%xmm4,%xmm0
# xmm0[0] = xmm2[0] + xmm4[0]
# Add Scalar Single-Precision

vaddps %xmm2,%xmm4,%xmm0
# xmm0[:] = xmm2[:] + xmm4[:]
# Add Packed Single-Precision
# "Vector" Instruction
```

- ▶ Operates on single values or "vectors" of packed values
- ▶ 3-operands common in more "modern" assembly languages

18

Example: `float_ops.c` to Assembly

```c
// float_ops.c: original C Code
void array_add(float *arr1, float *arr2, int len){
  for(int i=0; i<len; i++){
    arr1[i] += arr2[i];
  }
}
```

```
# >> gcc -S -Og float_ops.c
# Minimal optimizations
array_add:    ## 16 lines asm
.LFB0:
  .cfi_startproc
  movl  $0, %eax
  jmp   .L2
.L3:
  movslq %eax, %r8
  leaq  (%rdi,%r8,4), %rcx
  movss (%rsi,%r8,4), %xmm0
  addss (%rcx), %xmm0  ## add single
  movss %xmm0, (%rcx)  ## single prec
  addl  $1, %eax
.L2:
  cmpl  %edx, %eax
  jl    .L3
  ret
```

```
# >> gcc -S -O3 -mavx float_ops.c
# Max optimizations, Use AVX hardware
array_add:    ## 100 lines asm
...
.L5:          ## vector move/adds
  vmovups (%rcx,%rdx), %ymm1
  vaddps  (%rsi,%rdx), %ymm1, %ymm0
  vmovups %ymm0, (%rcx,%rdx)
  addq  $32, %rdx
  cmpq  %rdi, %rdx
  jne   .L5
...
.L9:          ## single move/adds
  vmovss (%rcx,%rax), %xmm0
  vaddss (%rsi,%rax), %xmm0, %xmm0
  vmovss %xmm0, (%rcx,%rax)
  addq  $4, %rax
  cmpq  %rax, %rdx
  jne   .L9
  ret
```

Floating Point and ALU Conversions

- ▶ Recall that bit layout of Integers and Floating Point numbers are quite different (**how?**)
- ▶ Leads to a series of assembly instructions to interconvert between types

```
# int eax = ...;
# double xmm0 = (double) eax;
vcvtsi2sd       %eax,%xmm0,%xmm0

# double xmm1 = ...
# long rcx = (int) xmm1;
vcvttsd2siq     %xmm1,%rcx
```

- ▶ These are non-trivial conversions: 5-cycle latency (delay) before completion, can have a performance impact on code which does conversions