UPE Tutoring:

CS 33 Midterm

Sign-in https://forms.gle/zzwiqjHCyXgnEPyX6

Slides link available upon sign-in

Integers and Binary Operations

Integers

- 4 bytes long - 00000000 00000000 00000000 00000000
- Signed vs Unsigned.
- Performing operations between signed and unsigned casts both to unsigned

Signed	Unsigned
• [-2,147,483,648, 2,147,483,647] • MSB Signed vs Two's complement ○ **MSB** - use most significant (leftmost) bit to denote negativity. 0b1000000000000001 = -1 ○ **Two's complement -** invert and add 1	• [0, 4,294,967,295]

Example of Two’s Complement

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

- Two’s complement is commonly used. What advantages does it have?

Example of Two's Complement

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Invert

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

- Two's complement is commonly used. What advantages does it have?

Example of Two's Complement

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Invert

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Add 1

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- Two's complement is commonly used. What advantages does it have?

Example of Two’s Complement

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Invert

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Add 1

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- Two’s complement is commonly used. What advantages does it have?
- Addition of numbers and their negative give us 0

AND (&)/OR (|)

`& (AND)`

A	B	
0	0	0
0	1	0
1	0	0
1	1	1

`| (OR)`

A	B	
0	0	0
0	1	1
1	0	1
1	1	1

XOR (^)

`^ (XOR)`

A	B	
0	0	0
0	1	1
1	0	1
1	1	0

```
0b1011 ^ 0b0100 = 0b1111

0b1010 ^ 0b0000 = 0b1010

0b1010 ^ 0b1010 = 0b0000

0b0011 ^ 0b1111 = 0b1100
```

Bitwise vs Logical NOT

~ (Bitwise NOT)

Flips all the bits

`~0b011100001111 = 0b100011110000`

! Logical NOT

Turns any non-zero number into 0.

Turns zero into 1.

`!0b011100001111 = 0b000000000000`

`!0b000000000000 = 0b000000000001`

Left-Shift (<<)

- Result always equivalent to multiplying by powers of 2
  - Even for negative numbers and with overflowing

```
int num = 0xFF00000F;
num << 4: 0xF00000F0      num * pow(2, 4): 0xF00000F0
```

Right-shift (>>)

- **Logical Right Shift**
  - Does not sign-extend
  - Applies on `unsigned int`
  - Like division by powers of 2

```
unsigned num = 0xCF00000F;

num >> 4:  0x0CF00000
num / 16:  0x0CF00000
```

- **Arithmetic Right Shift**
  - Sign is preserved
  - Applies on `int`
  - **Not like** division by powers of 2
    - Rounded towards -Infinity instead of zero

```
int num = 0xCF00000F;

num >> 4:  0xFCF00000
num / 16:  0xFCF00001
```

```
int num = -7;

num >> 1:  -4
num / 2:  -3
```

Common Bit Manipulation Patterns

- Negative of a number
  `-c = ~c + 1`

- Mask to 0/1 bit
  `!!mask`

- Mask from Sign-bit
  `num >> 31`

- Getting a bit

```
bool getBit(int num, int i) {
    int bit = ((1 << i) & num);
    return !!bit;
}
```

- Setting a bit

```
bool setBit(int num, int i) {
    return ((1 << i) | num);
}
```

Bit Manipulation Patterns

- Branches

```
if (test)
    output = a;
else
    output = b;
```

```
mask = ((!!test << 31) >> 31);
output = (mask & a) | (~mask & b);
```

- Online resource for solutions of harder bit-manipulation problems: https://graphics.stanford.edu/~seander/bithacks.html

Practice Question

Write a C function using only bitwise operators and “!”, to determine if the number given is non-negative. If the number is non-negative return 1 else 0.

Hint: It can be done in one line!

Answer

Write a C function using only bitwise operators and "!", to determine if the number given is non-negative. If the number is non-negative return 1 else 0.

```
int is_nonnegative(int x) {
    return !(x >> 31);
}
```

Let's try something a little harder!

Practice Question

Write a C function using only bitwise operators to return the absolute value of a number. If the number is Tmin, just return Tmin since there is no positive representation of Tmin.

```
int my_abs(int x) {
    ...
}
```

Practice Question

Write a C function using only bitwise operators to return the absolute value of a number. If the number is Tmin, just return Tmin since there is no positive representation of Tmin.

```
int my_abs(int x) {
    int sign = x >> 31;
    return (x ^ sign) + (1 & sign); // multiply by -1 if the sign is negative
}
```

Practice Questions

Assuming:

```
int x = rand();
int y = rand();
unsigned ux = unsigned(x);
unsigned uy = unsigned(y);
```

Which of the following are always true?

=> means implies

- `(x > 0) && (y >0) => x*y > 0`
- `x > uy  => x > 0`
- `(x^x)^x == y => x==y`
- `(ux-uy) == x - y`
- `(x << 32) >> 32 == x`

Practice Questions

Assuming:

int x = rand();
int y = rand();
unsigned ux = unsigned(x);
unsigned uy = unsigned(y);

Which of the following are always true?

=> means implies

- (x > 0) && (y >0) => x*y > 0 False (undefined)
- x > uy => x > 0 False
- (x^x)^x == y => x==y True
- (ux-uy) == x - y True
- (x << 32) >> 32 == x Undefined

ISAs/x86-64

ISA - x86-64

- ISA - Instruction Set Architecture. Interface between software and hardware
- “CISC” architecture - relatively complex instructions. Opposite would be RISC
- Little-Endian
- Used by Intel Chips
- 64-bit

Byte Ordering

Big Endian

- Least significant byte has highest address

Little Endian

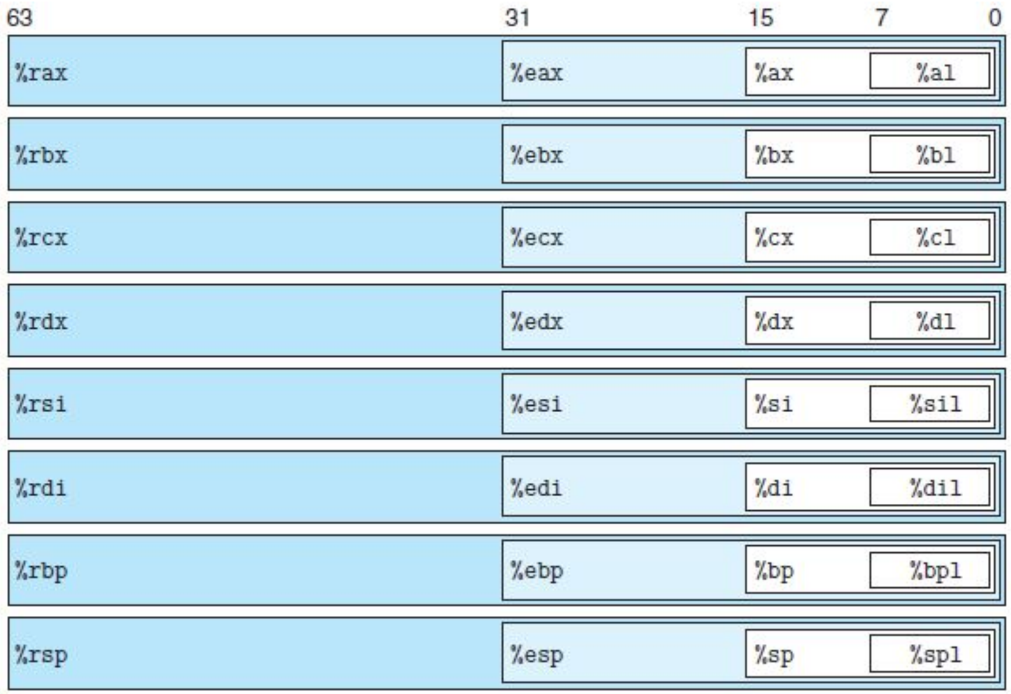
- Least significant byte has lowest address

Example - 0x01234567 at 0x100

0x100	0x108	0x110	0x118	
01	23	45	67	**Big Endian**
67	45	23	01	**Little Endian**

Registers

- Registers are small hardware based storage devices closest to the CPU that generally store addresses into physical memory.
- Assembly code suffix to size in bytes
  - **b:** 1
  - `w:` 2
  - `l:` 4
  - `q:` 8
- (Floating-points)
  - `s:` 4
  - `l:` 8



Registers

- `%rax`: Return value
- `%rsp`: Stack Pointer
- `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
  - Store the 1st-6th argument passed to a function call
- `%r10, %r11`
  - Caller-saved registers
  - After a callq instruction, their values might be different
  - So, the caller needs to save their values if these registers are holding data
- `%rbx, %rbp, %r12, %r13, %r14, %r15`
  - Callee-saved registers
  - Before existing a function call, ensure that the values in these registers did not change

Memory Addressing (x86-64)

Memory Addressing Modes

The general form is as follows:

```
D(Rb, Ri, S)
```

- `D` is a constant displacement that can be as wide as 32-bits.
- `Rb` is the base register, which is any of the 16 integer registers.
- `Ri` is the index register, which can be any register except for `%rsp`.
- `S` is the scale. This can only be **1, 2, 4, or 8.**

- This translates to `MEM[D + REG[Rb] + REG[Ri]*S]`.

Memory Addressing Modes

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$R[\mathtt{r}_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$M[R[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$M[Imm + R[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$M[R[\mathtt{r}_b] + R[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$M[Imm + R[\mathtt{r}_b] + R[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$M[R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$M[Imm + R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$M[R[\mathtt{r}_b] + R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$M[Imm + R[\mathtt{r}_b] + R[\mathtt{r}_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Memory Addressing Modes

Consider the following instruction:

```
movq $10, -8(%rax, %rdx, 4)
```

Memory Addressing Modes

Consider the following instruction:

```
movq $10, -8(%rax, %rdx, 4)
```

1. We're applying the `movl` instruction, which is responsible for moving (or *copying*) a value from one memory location to another.

Memory Addressing Modes

Consider the following instruction:

```
movq $10, -8(%rax, %rdx, 4)
```

1. We're applying the `movl` instruction, which is responsible for moving (or *copying*) a value from one memory location to another.
2. We're moving the immediate value `$10`. We know that it's an *immediate* because it is prepended by a $ sign. An immediate is like a constant.

Memory Addressing Modes

Consider the following instruction:

```
movq $10, -8(%rax, %rdx, 4)
```

1. We're applying the `movl` instruction, which is responsible for moving (or *copying*) a value from one memory location to another.
2. We're moving the immediate value `$10`. We know that it's an *immediate* because it is prepended by a $ sign. An immediate is like a constant.
3. We're moving the immediate into a memory location, `MEM[REG[%rax]+4*REG[%rdx]+(-8)]`. We know that it's a memory location because of the parentheses!

The `leaq` Instruction

Let's consider the following instruction:

`leaq` *`source, destination`*

`leaq`, or load effective address, does exactly as its name suggests. The instruction takes a memory address expressed through the parameters `(D, Rb, Ri, S)` and, rather than returning the value at the address, returns the address itself.

This in contrast with `movq`, which moves the *contents* of a specific address.

Practice Question

Now, suppose we have:

```
leaq D(Rb, Ri, S), %rbx
```

And that our variable, `N`, is in the register `%rax`.

What values `K*N` can we produce with the `leaq` instruction? (Where `K` is some integer.)

Answer

Now, suppose we have:

```
leaq D(Rb, Ri, S), %rbx
```

And that our variable, `N`, is in the register `%rax`.

What values `K*N` can we produce with the `leaq` instruction? (Where `K` is some integer.)

1. If we use the form `(, %rax, S)` we can produce `N`, `2N`, `4N`, and `8N`.

Answer

Now, suppose we have:

```
leaq D(Rb, Ri, S), %rbx
```

And that our variable, `N`, is in the register `%rax`.

What values `K*N` can we produce with the `leaq` instruction? (Where `K` is some integer.)

1. If we use the form `(, %rax, S)` we can produce `N`, `2N`, `4N`, and `8N`.
2. If we use the form `(%rax, %rax, S)`, we can take advantage of the adding of the first parameter and produce `3N`, `5N`, and `9N`.

Practice: `movq` vs. `leaq`

```
leaq (%rdx), %rdi

movq (%rdx), %rsi
```

Registers	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Address	Memory
0x120	0x400
0x118	0xF
0x110	0x8
0x108	0x10
0x100	0x1

Practice: `movq` vs. `leaq`

```
leaq (%rdx), %rdi

movq (%rdx), %rsi
```

Registers	
%rcx	0x4
%rdx	0x100
%rdi	*0x100*
%rsi	*0x1*

Address	Memory
0x120	0x400
0x118	0xF
0x110	0x8
0x108	0x10
0x100	0x1

Practice: `movq` vs. `leaq`

```
leaq (%rdx, %rcx, 4), %rdi

movq (%rdx, %rcx, 4), %rsi
```

Registers	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Address	Memory
0x120	0x400
0x118	0xF
0x110	0x8
0x108	0x10
0x100	0x1

Practice: `movq` vs. `leaq`

```
leaq (%rdx, %rcx, 4), %rdi

movq (%rdx, %rcx, 4), %rsi
```

Registers	
%rcx	0x4
%rdx	0x100
%rdi	*0x110*
%rsi	*0x8*

Address	Memory
0x120	0x400
0x118	0xF
0x110	0x8
0x108	0x10
0x100	0x1

Condition Flags and Jumps

- CF - carry flag
- ZF - zero flag
- SF - sign flag
- OF - overflow flag

Implicitly set by arithmetic operations

Explicitly set by cmp and test

cmp b, a => a - b

test b, a => a & b

je - jump equal

jne - jump not equal

js - jump if signed

jns - jump not signed

jg - jump greater (signed)

jge - jump greater than or equal (signed)

jl - jump less (signed)

jle - jump less than equal (signed)

ja - jump above (unsigned)

jb - jump below (unsigned)

Conditional move

`cmove S, D`

Example:

```
testq %rdi, %rdi
cmove %rdx, %rax
```

Equivalent C code:

```
v  = then-expr;
ve = else-expr;
t  = test-expr;
if (!t) v = ve;
```

Why this code cannot be assembled into a conditional move?

```
long cread(long *xp) {
    return (xp ? *xp : 0);
}
```

Control Statements/Code Tracing

If Statements

What would the following code look like in C/pseudo-code?

```
0x400572 <+0>:     cmp  $0x6,%edi
0x400575 <+3>:     jg  0x40057d <func0+11>
0x400577 <+5>:     mov$0x0,%eax
0x40057c <+10>:    ret
0x40057d <+11>:    mov$0x6,%eax
0x400582 <+16>:    ret
```

If Statements

```
if (a > 6)
        return 6;
 return 0;
```

```
0x400572 <+0>:    cmp  $0x6,%edi
0x400575 <+3>:    jg  0x40057d <func0+11>
0x400577 <+5>:    mov$0x0,%eax
0x40057c <+10>:   ret
0x40057d <+11>:   mov$0x6,%eax
0x400582 <+16>:   ret
```

Loops

What would the following code look like in C/pseudo-code?

```
...
0x400589 <+6>:    mov   %edi,%ebp
0x40058b <+8>:    mov   $0x0,%ebx
0x400590 <+13>:   jmp   0x4005a6 <func1+35>
0x400592 <+15>:   mov   %ebx,%esi
0x400594 <+17>:   mov   $0x4006b4,%edi
0x400599 <+22>:   mov   $0x0,%eax
0x40059e <+27>:   call   0x400460 <printf@plt>
0x4005a3 <+32>:   add   $0x1,%ebx
0x4005a6 <+35>:   cmp   %ebp,%ebx
0x4005a8 <+37>:   jl    0x400592 <func1+15>
0x4005aa <+39>:   mov   $0x0,%eax
0x4005af <+44>:   add   $0x8,%rsp
...
0x4005b5 <+50>:   ret
```

Loops

```
int i = 0;
while(i < a) {
        printf("%d", i);
        i++;
 }
 return 0;
```

```
...
0x400589 <+6>:    mov   %edi,%ebp
0x40058b <+8>:    mov   $0x0,%ebx
0x400590 <+13>:   jmp   0x4005a6 <func1+35>
0x400592 <+15>:   mov   %ebx,%esi
0x400594 <+17>:   mov   $0x4006b4,%edi
0x400599 <+22>:   mov   $0x0,%eax
0x40059e <+27>:   call  0x400460 <printf@plt>
0x4005a3 <+32>:   add   $0x1,%ebx
0x4005a6 <+35>:   cmp   %ebp,%ebx
0x4005a8 <+37>:   jl    0x400592 <func1+15>
0x4005aa <+39>:   mov   $0x0,%eax
0x4005af <+44>:   add   $0x8,%rsp
...
0x4005b5 <+50>:   ret
```

Switch Statements

With a switch statement like the one on the right and the assembly in the next slide, fill in the missing jump table values.

```
switch(a) {
    case 100:
    case 102:
        return 1;
    case 103:
        return 2;
    case 104:
        return 3;
    case 106:
        return 4;
    case 107:
        return 5;
    default:
        return 0;
}
```

Switch Statements

```
0x4005b6 <+0>:   sub  $0x64,%edi
0x4005b9 <+3>:   cmp  $0x7,%edi
0x4005bc <+6>:   ja   0x4005df <func2+41>
0x4005be <+8>:   mov  %edi,%edi
0x4005c0 <+10>:  jmp  *0x4006b8(,%rdi,8)
0x4005c7 <+17>:  mov  $0x1,%eax
0x4005cc <+22>:  ret
0x4005cd <+23>:  mov  $0x3,%eax
0x4005d2 <+28>:  ret
0x4005d3 <+29>:  mov  $0x4,%eax
0x4005d8 <+34>:  ret
0x4005d9 <+35>:  mov  $0x5,%eax
0x4005de <+40>:  ret
0x4005df <+41>:  mov  $0x0,%eax
0x4005e4 <+46>:  ret
0x4005e5 <+47>:  mov  $0x2,%eax
0x4005ea <+52>:  ret
```

Switch Statements

0x4006b8:	___	___	___	0x00	0x00	0x00	0x00	0x00
0x4006c0:	___	___	___	0x00	0x00	0x00	0x00	0x00
0x4006c8:	___	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006d0:	___	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006d8:	___	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006e0:	___	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006e8:	___	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006f0:	___	0x05	0x40	0x00	0x00	0x00	0x00	0x00

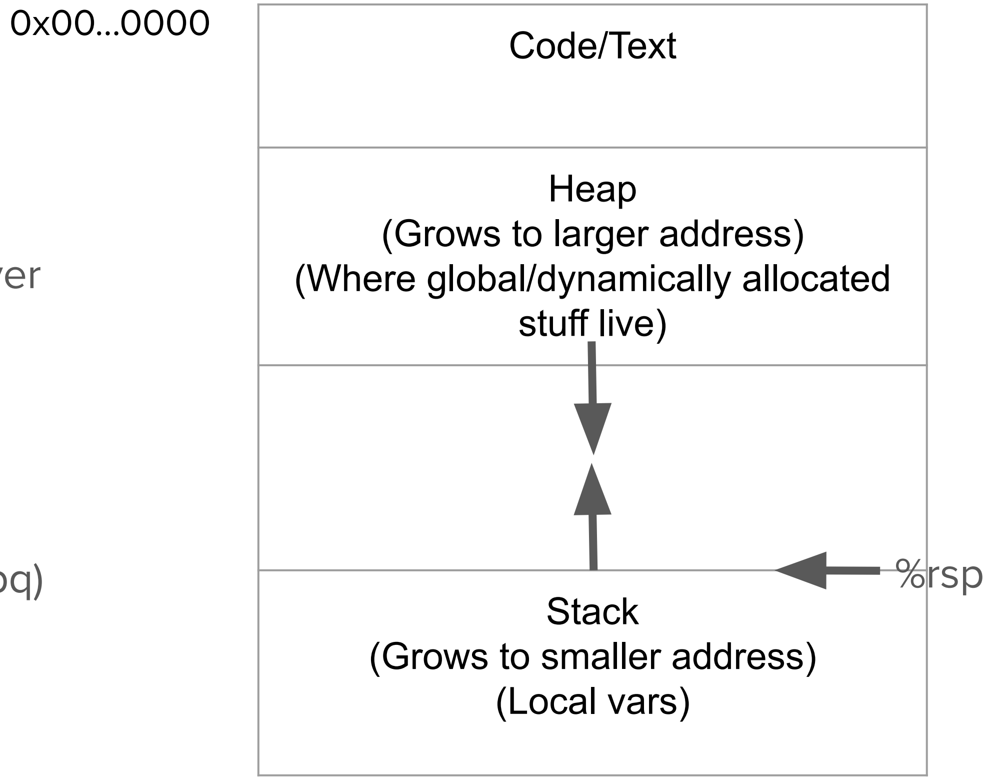
Switch Statements (Answers)

0x4006b8:	0xc7	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006c0:	0xdf	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006c8:	0xc7	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006d0:	0xe5	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006d8:	0xcd	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006e0:	0xdf	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006e8:	0xd3	0x05	0x40	0x00	0x00	0x00	0x00	0x00
0x4006f0:	0xd9	0x05	0x40	0x00	0x00	0x00	0x00	0x00

The Stack

The Stack

- Stores local variables
- Enter a new “stack frame” whenever you enter a new function
- Grows by decrementing %rsp (pushq)
- Shrinks by incrementing %rsp (popq)



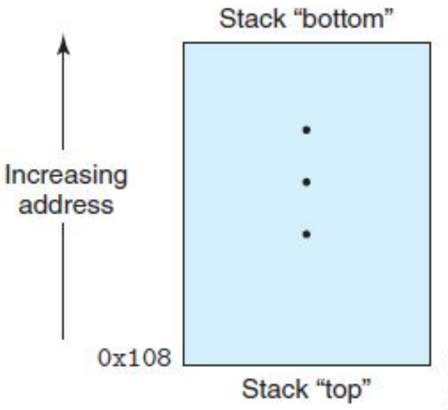
pushq

- First decrement stack pointer
- Then place the value

Instruction	Effect	Description
pushq S	R[%rsp] $\leftarrow$ R[%rsp] $-$ 8; M[R[%rsp]] $\leftarrow$ S	Push quad word

Initially

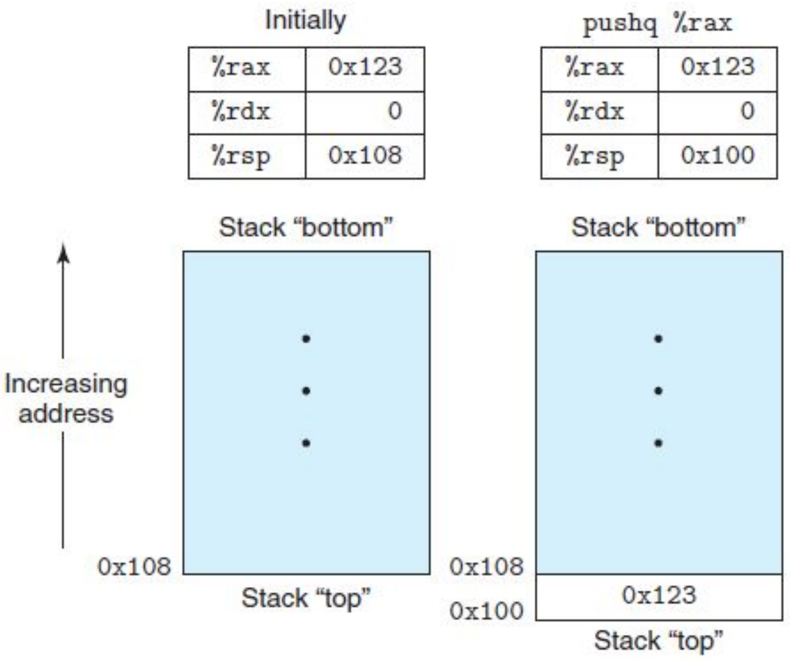
%rax	0x123
%rdx	0
%rsp	0x108



pushq

Instruction	Effect	Description
pushq S	R[%rsp] ← R[%rsp] – 8; M[R[%rsp]] ← S	Push quad word

- First decrement stack pointer
- Then place the value



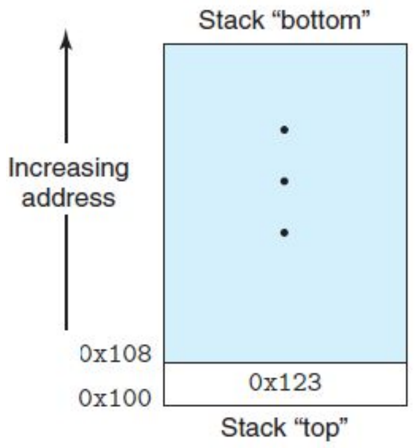
popq

- Extract the value
- Then increment the stack pointer

Instruction	Effect	Description
popq *D*	*D* ← M[R[%rsp]]; R[%rsp] ← R[%rsp] + 8	Pop quad word

Initially

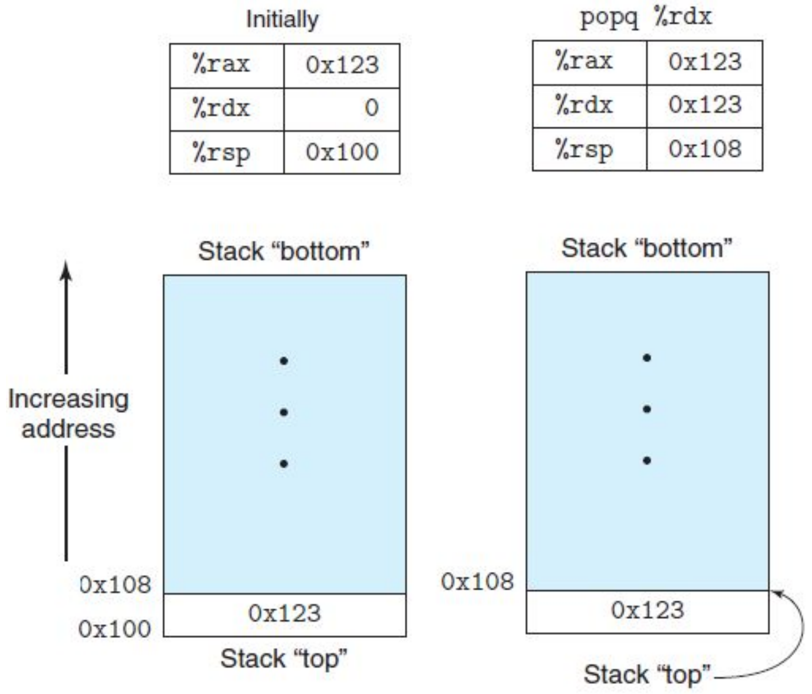
%rax	0x123
%rdx	0
%rsp	0x100



popq

- Extract the value
- Then increment the stack pointer

Instruction	Effect	Description
popq D	$D \leftarrow$ M[R[%rsp]]; R[%rsp] $\leftarrow$ R[%rsp] + 8	Pop quad word



Call and Return Procedures

Recall the usage of `call` and `ret`:

- These instructions implement a subroutine `call` and `return`.
- The `call` instruction saves a return address, then jumps to the location of the callee function.
- The `ret` instruction uses the previously saved return address to jump back to the caller function.

The `call` Instruction

- pushes the current code location onto the hardware supported stack in memory
- performs an unconditional jump to the code location indicated by the label operand.

Note: Unlike the simple jump instructions, the `call` instruction saves the location to return to when the subroutine completes.

Given the protocol above, the `call` instruction can be interpreted as follows:

```
call <label>:
  pushq %rip  // Note: %rip points to the next
              // instruction to be executed.
  jmp <label>
```

The Ret Instruction

- Transfers program control to a return address located on the top of the stack
- The address is usually placed on the stack by a CALL instruction

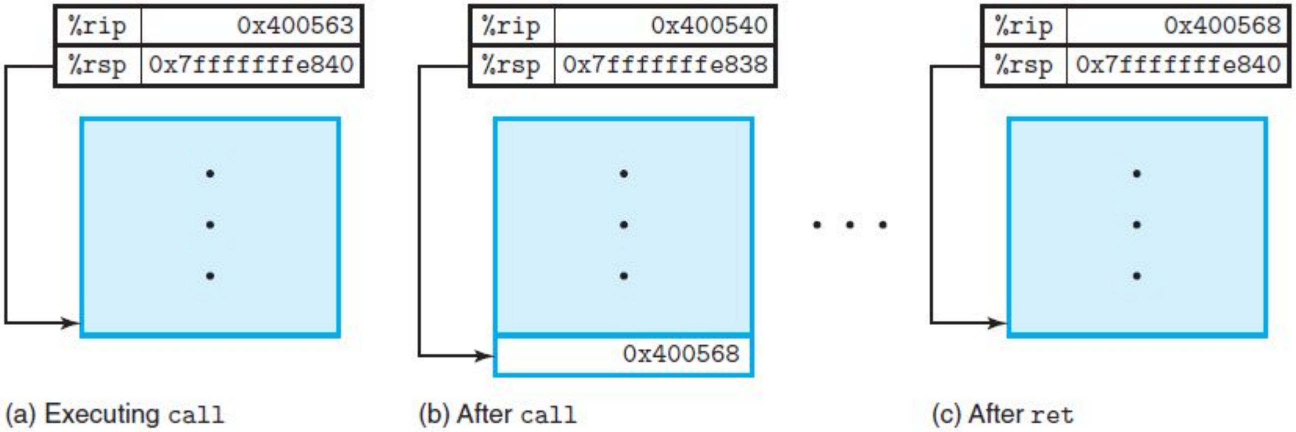
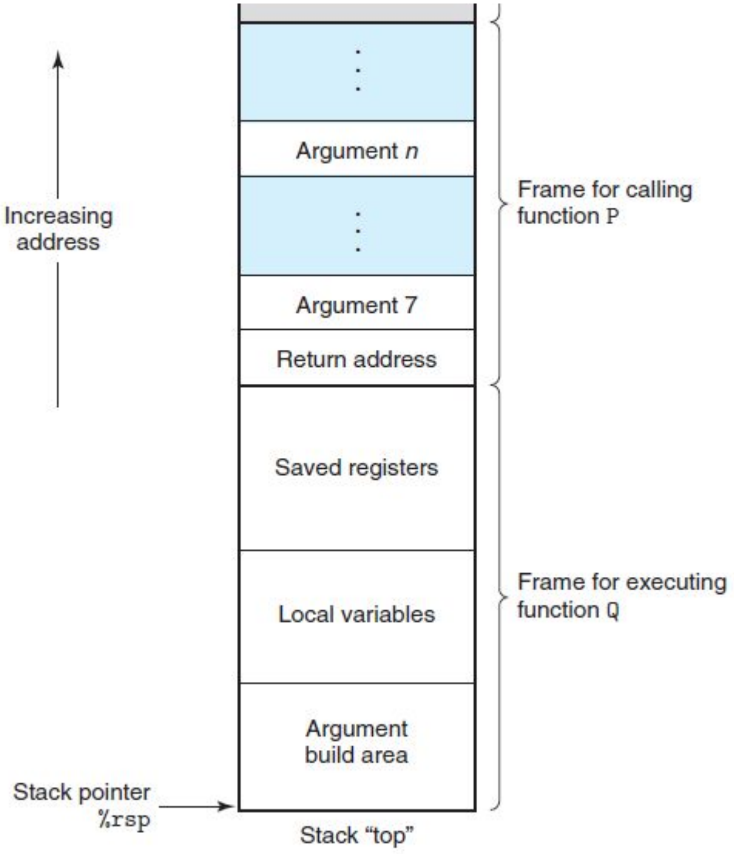


Figure 3.26 Illustration of `call` and `ret` functions. The `call` instruction transfers control to the start of a function, while the `ret` instruction returns back to the instruction following the call.

Call Stack

- Stack grows when more items are added to its “top”
- Pushing on the stack grows it.
- Stack grows in the direction of **decreasing** address
- Pushing on the stack = decrementing the stack pointer



Data Representations

Alignment and `struct`

- Structs are always aligned by the largest data member they contain
- If they contain multiple data members of different sizes, the smaller sized-values are "padded" with extra bytes until they also uphold the alignment property
- The order in memory of the data members in the struct is the same order as the defined.
- If x is the starting address of a struct, then a data member of size 4 would start at either x, x+4, x+8, x+12...

Practice Question

How big is this struct in bytes? (Assume 64-bit architecture)

```
struct westeros {
    char lannister;
    double* stark;
    short frey;
};
```

Answer

How big is this struct in bytes? (Assume 64-bit architecture)

```
struct westeros {
    char lannister;
    double* stark;
    short frey;
};
```

Recall that in a 64-bit architecture, pointers are 8 bytes.

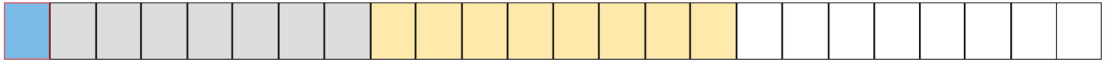
Answer

How big is this struct in bytes? (Assume 64-bit architecture)

```
struct westeros {
    char lannister; // 1 byte,  plus 7 bytes of padding
    double* stark;
    short frey;
};
```

Recall that in a 64-bit architecture, pointers are 8 bytes.

Answer



How big is this struct in bytes? (Assume 64-bit architecture)

```
struct westeros {
    char lannister; // 1 byte,  plus 7 bytes of padding
    double* stark;  // 8 bytes
    short frey;
};
```

Recall that in a 64-bit architecture, pointers are 8 bytes.

Answer

How big is this struct in bytes? (Assume 64-bit architecture)

```
struct westeros {
    char lannister; // 1 byte,  plus 7 bytes of padding
    double* stark;  // 8 bytes
    short frey;     // 2 bytes, plus 6 bytes of padding
};
```

Recall that in a 64-bit architecture, pointers are 8 bytes.

Answer

How big is this struct in bytes? (Assume 64-bit architecture)

```
struct westeros {
    char lannister; // 1 byte,  plus 7 bytes of padding
    double* stark;  // 8 bytes
    short frey;     // 2 bytes, plus 6 bytes of padding
};

// Total size: 24 bytes
```

Recall that in a 64-bit architecture, pointers are 8 bytes.

Practice Question

What if we swap the places of short and the pointer? (Assume 64-bit architecture)

```
struct westeros {
    char lannister;
    short frey;
    double* stark;
};
```

Answer

How big is this struct in bytes? (Assume 64-bit architecture)

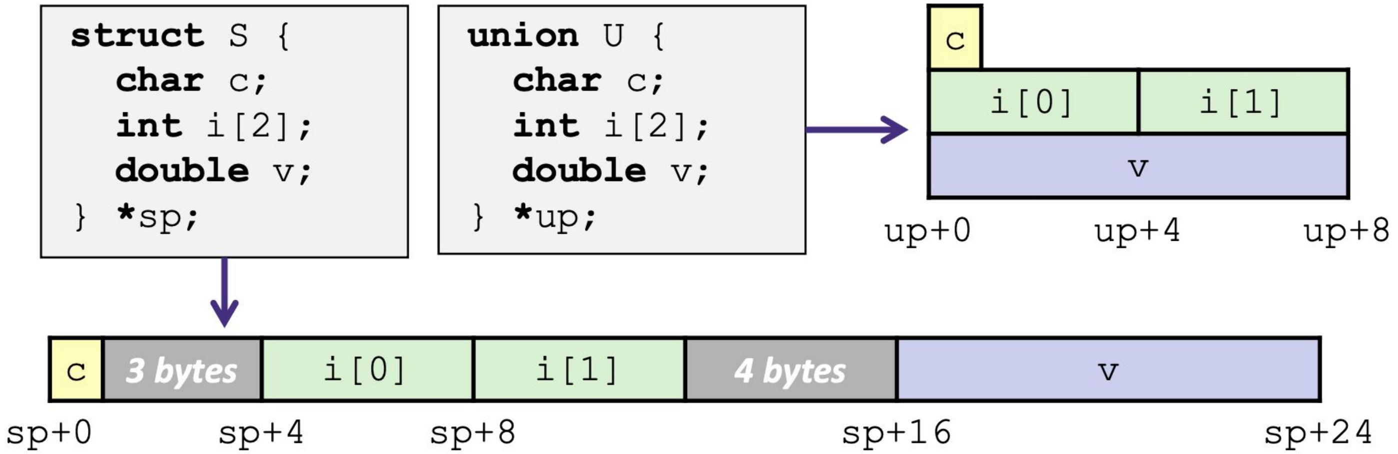
```
struct westeros {
    char lannister; // 1 byte,  plus 1 bytes of padding
    short frey;     // 2 bytes, plus 4 bytes of padding
    double* stark;  // 8 bytes
};

// Total size: 16 bytes
```

Recall that in a 64-bit architecture, pointers are 8 bytes.

Union

- Size of union depends on the largest element in the union
- Only one member can have a value at a given time
- Example below has size 8 bytes



Multidimensional Arrays vs Multi-level Array

Multidimensional

- Continuous Memory
- Ex. int myArray[3][3];

Multi-Level

- Array of pointers to arrays
- Ex: int myArray2[3]*;

Multidimensional Arrays vs Multi-level Array

Multidimensional

- Continuous Memory

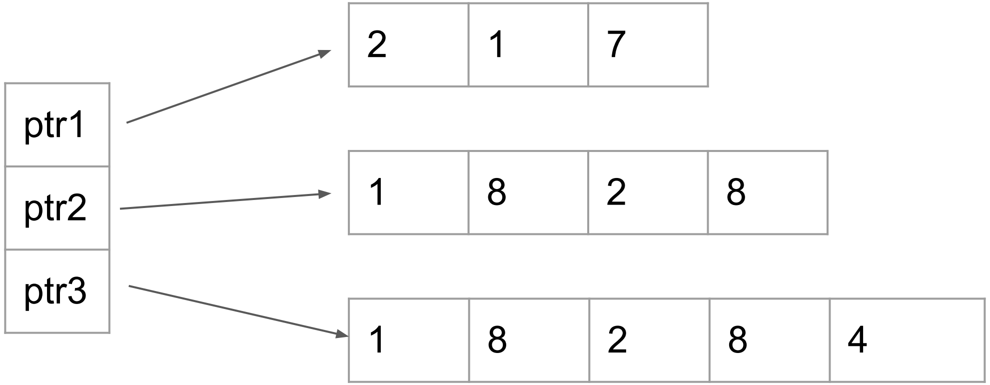
- Ex. int myArray[3][3];

3	1	4
1	5	9
2	6	5

Multi-Level

- Array of pointers to arrays

- Ex: int myArray2[3]*;



Floating Point Numbers

Floating Point Numbers

IEEE Floating Point

[s][e][f]

- IEEE-754 32-bit ‘Single’
  - 1 sign bit, 8 exponent bits, 23 fraction bits
- IEEE-754 64-bit ‘Double’
  - 1 sign bit, 11 exponent bits, 52 fraction bits

Floating Point Numbers

IEEE Floating Point

[s][e][f]

- First separate the sign, exponent and the fraction bits **according to the standard**
  - Maybe the exam uses 1 sign, 4 exponent, 10 fraction bits. Use that then!

- Then, interpret the bits in [e] part as a positive integer.
  - That’s the value of **e** for now.

Floating Point Numbers

IEEE-754 32-bit 'Single'
1 sign, 8 exponent, 23 fraction bits

Normalized Numbers

When exponent in range 1 <= e <= 254   (bits are of the form 0...01 to 1...10)

`result = (sign) * 2^(e - 127) * 1.f`

$$\text{result} = (\text{sign}) * 2^{(e\ -\ 127)} * 1.f$$

Denormalized "Tiny" Numbers

When e == 0

$$\text{result} = (\text{sign}) * 2^{(e\ -\ 127)} * 0.f$$

Floating Point Numbers

IEEE-754 32-bit 'Single'
1 sign, 8 exponent, 23 fraction bits

Why do we subtract exponent bias from the value of e?

- Linearly displace so that roughly half of the **e** represent negative exponent

Why exactly 127?

- About the mid of the possible values of e, when e is 8 bits

$$\text{Bias} = \frac{2^{|e|}}{2} - 1$$

$$127 = \frac{2^8}{2} - 1 = 128 - 1$$

Floating Point Numbers

IEEE-754
Applies to any bit distribution

What's `1.f` or `0.f`?

```
If f is             1 1 1 0 0 0…

Then 1.f is       1.1 1 1 0 0 0…
                = 1+½+¼+⅛+0+0+0…
                = 1 + 0.5 + 0.25 + 0.125
                = 1.875
```

Floating Point Numbers

IEEE-754 32-bit ‘Single’
1 sign, 8 exponent, 23 fraction bits

- Infinity
  - `e = 255, f = 0`
- Zero
  - `s = 0 or 1, e = 0, f = 0`
- NaN
  - `e = 255, f != 0`
  - Most operations involving NaN will return a NaN (e.g. addition)
  - Comparing NaN to any value will return false/0
    - `(NaN == 4)         // equals 0`
    - `(NaN == NaN)  // also equals 0`

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$-1^{S} * 2^{(e - 127)} * 1.F$

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$-1^{S} * 2^{(e - 127)} * 1.F$

Signed bit: 0, S = 0

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000 $-1^{S} * 2^{(e - 127)} * 1.F$

Signed bit: 0, S = 0

Exponent: $10001101_2 = 141_{10} = E$

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$-1^{S} * 2^{(e\ -\ 127)} * 1.F$

Signed bit: 0, S = 0

Exponent: $10001101_2 = 141_{10} = E$

$F = 10000000000000000000000_2 = 2^{-1}$

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$-1^{S} * 2^{(e - 127)} * 1.F$

Signed bit: 0, S = 0

Exponent: $10001101_2 = 141_{10}$ = E

F = $\mathbf{1}0000000000000000000000_2 = \mathbf{2^{-1}} = 0.5$ = F

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$-1^S * 2^{(e - 127)} * 1.F$

Signed bit: 0, $S = 0$

Exponent: $10001101_2 = 141_{10} = E$

$F = 10000000000000000000000_2 = 2^{-1} = 0.5 = F$ or $1.F = 1.5$

$-1^0 * 2^{141 - 127} * 1.5$

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000 $-1^{S} * 2^{(e - 127)} * 1.F$

Signed bit: 0, $S = 0$

Exponent: $10001101_2 = 141_{10} = E$

$F = 10000000000000000000000_2 = 2^{-1} = 0.5 = F$ or $1.F = 1.5$

$-1^{0} * 2^{141 - 127} * 1.5 = 16384 * 1.5$

Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$-1^{S} * 2^{(e - 127)} * 1.F$

Signed bit: 0, S = 0

Exponent: $10001101_2 = 141_{10} = E$

$F = 10000000000000000000000_2 = 2^{-1} = 0.5 = F$

$-1^{0} * 2^{141 - 127} * 1.5 = 16384 * 1.5 =$ **24576 :)**

Floating Point Numbers

- Problems
  - Overflow and underflow
  - Invalid operations (e.g., divide by zero, square root of negative)
  - Precision issues
    - $(1 + 23) \times \log_{10}(2) \approx 7.22$ digits of precision for `float`
    - $(1 + 52) \times \log_{10}(2) \approx 15.95$ digits of precision for `double`
- Tips
  - About half of all numbers between -1 and 1 in floating point
  - Write calculations to return results in that range
  - `printf(“%.17f\n”, value)`
    - Usually the best (lossless) printing of `double` value

Floating Point Numbers

What is the first positive integer that can't be represented exactly by a `float`?

Floating Point Numbers

What is the first positive integer that can't be represented exactly by a `float`?

```
2^(1 + 23) + 1 = 16,777,217
```

What happens if you try to add 1 to 16,777,216?

```
0   10010111     00000000000000000000000
```

$1 \times 2^{24} \times 1.00000000000000000000000_2 = 1000000000000000000000000.0_2$

After moving the decimal point to the right 24 times, we see that the one's places is not represented by `f`

Good luck!

Sign-in https://forms.gle/zzwiqjHCyXgnEPyX6
Slides https://tinyurl.com/cs33midtermsp22
Feedback https://tinyurl.com/uclaupetutoringfeedback
Practice https://github.com/uclaupe-tutoring/practice-problems/wiki

Questions? Need more help?

- Come up and ask us! We'll try our best.
- Other resources include your LAs and ACM Cyber
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.