CS 107
Lecture 13:
Assembly Part IV

Monday, February 19, 2018

Computer Systems
Winter 2018
Stanford University
Computer Science Department

Reading: Course Reader: x86-64 Assembly Language, Textbook: Chapter 3.1-3.4

Lecturer: Chris Gregg

```
0x40055d  <val_at_index>          movslq  %esi,%rsi
0x400560  <val_at_index+3>        mov    (%rdi,%rsi,4),%eax
0x400563  <val_at_index+6>        retq
```

```
0  1  2  3  4  5  6  7  8  9
42 18 12  9  0 -5 13 -8 12 23
```

Today's Topics

• Reading: Chapter 3.7.4-3.7.6, 3.8-3.9
• Programs from class: /afs/ir/class/cs107/samples/lect14
• Logistics
  • Midterm requests in by today
• Final day of x86 Assembly Language
  • Procedures
    • Local storage on the stack
    • Local storage in registers
    • Recursion
• Arrays
• Structures
• Alignment
• Function pointers
• Assembly wrap-up, and comments on binary bomb assignment.
Local Stack Storage

• We haven't seen much use of local storage on the stack, but there are times when it is necessary:
  • When there are not enough registers to hold the local data
  • When the address operator '&\;' is applied to a local variable, and we have to be able to generate an address for it.
  • Some of the local variables are arrays or structs, and must be accessed by array or structure references.
• The typical way to allocate space on the stack frame is to decrement the stack pointer.
• Remember, a function must return the stack pointer to the proper value (such that the top of the stack is the return address) before it returns.
Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}
```

```c
long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```
subq $16, %rsp        // allocate 16 bytes for stack frame
movq $534, (%rsp)    // store 534 in arg1
movq $1057, 8(%rsp)  // store 1057 in arg2
lea 8(%rsp), %rsi     // compute &arg2 as second argument
movq %rsp, %rdi       // compute &arg1 as first argument
call swap_add         // call swap_add(&arg1, &arg2)
movq (%rsp), %rdx      // get arg1
subq 8(%rsp), %rdx    // compute diff = arg1 - arg2
imulq %rdx, %rax      // compute sum * diff
addq $16, %rsp       // deallocate stack frame
ret
```

- The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

- Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx     // get arg1
subq 8(%rsp), %rdx   // compute diff = arg1 - arg2
imulq %rdx, %rax     // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

- The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

• Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:
```
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi      // compute &arg1 as first argument
call swap_add        // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx     // get arg1
subq 8(%rsp), %rdx   // compute diff = arg1 - arg2
imulq %rdx, %rax     // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

• The caller must allocate a stack frame due to the presence of address operators.
• Example:

```c
long swap_add(long *xp, long *yp) {
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller() {
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```assembly
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

• The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

• Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```assembly
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

• The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

• Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}
```

```c
caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```assembly
subq $16, %rsp     // allocate 16 bytes for stack frame
movq $534, (%rsp)  // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi  // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

• The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

- Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}
```

```c
long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:
```
subq $16, %rsp  // allocate 16 bytes for stack frame
movq $534, (%rsp)  // store 534 in arg1
movq $1057, 8(%rsp)  // store 1057 in arg2
lea 8(%rsp), %rsi  // compute &arg2 as second argument
movq %rsp, %rdi  // compute &arg1 as first argument
call swap_add  // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx  // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax  // compute sum * diff
addq $16, %rsp  // deallocate stack frame
ret
```

- The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

- Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

- The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

• Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}
```

```c
long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```assembly
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret```

• The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

- Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}
```

```c
long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

- The caller must allocate a stack frame due to the presence of address operators.
Local Stack Storage

- Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}
```

```asm
caller:
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp),%rdx   // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

- The caller must allocate a stack frame due to the presence of address operators.
You may not have noticed, but none of the examples in the book so far have used `%rbx` for anything, and gcc often uses `%rax`, `%rcx`, `%rdx`, etc., but skips right over `%rbx`.

One reason is that `%rbx` is designated, by convention, to be a "caller owned" register:

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%ax</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bl</td>
</tr>
</tbody>
</table>

What that means is that if a function uses `%rbx`, it guarantees that it will restore `%rbx` to its original value when the function returns.

The full list of caller owned registers are `%rbx`, `%rbp`, and `%r12-%r15`.

If a function uses any of those registers, it must save them on the stack to restore.
Local Storage in Registers

- The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.
- Example:

```c
long P(long x, long y) {
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

- The first time Q is called, `x` must be saved for later, and the second time Q is called, `u` must be saved for later.

```c
long P(long x, long y),
    x in %rdi, y in %rsi:
    push %rbp
    push %rbx
    mov %rdi,%rbp
    mov %rsi,%rdi
    callq 40056d <Q(long)>
    mov %rax,%rbx
    mov %rbp,%rdi
    callq 40056d <Q(long)>
    add %rbx,%rax
    pop %rbx
    pop %rbp
    retq
```
Local Storage in Registers

- The other registers are "x" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

- Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

- The first time Q is called, \( x \) must be saved for later, and the second time Q is called, \( u \) must be saved for later.
Local Storage in Registers

• The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

• Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

• The first time Q is called, \( x \) must be saved for later, and the second time Q is called, \( u \) must be saved for later.

```
long P(long x, long y),
x in %rdi, y in %rsi:
push %rbp
    push %rbx
mov %rdi,%rbp
mov %rsi,%rdi
callq 40056d <Q(long)>
mov %rax,%rbx
mov %rbp,%rdi
callq 40056d <Q(long)>
add %rbx,%rax
pop %rbx
pop %rbp
retq
```
Local Storage in Registers

- The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.
- Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

- The first time Q is called, x must be saved for later, and the second time Q is called, u must be saved for later.
Local Storage in Registers

- The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.
- Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

- The first time Q is called, x must be saved for later, and the second time Q is called, u must be saved for later.
Local Storage in Registers

• The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

• Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

• The first time Q is called, \( x \) must be saved for later, and the second time Q is called, \( u \) must be saved for later.

```assembly
long P(long x, long y),
x in %rdi, y in %rsi:
push %rbp
push %rbx
mov %rdi,%rbp
mov %rsi,%rdi
callq 40056d <Q(long)>
mov %rax,%rbx
mov %rbp,%rdi
callq 40056d <Q(long)>
add %rbx,%rax
pop %rbx
pop %rbp
retq
```
Local Storage in Registers

• The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

• Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

```asm
long P(long x, long y),
x in %rdi, y in %rsi:
push %rbp
push %rbx
mov %rdi,%rbp
mov %rsi,%rdi
callq 40056d <Q(long)>
mov %rax,%rbx
mov %rbp,%rdi
callq 40056d <Q(long)>
add %rbx,%rax
pop %rbx
pop %rbp
retq
```

• The first time Q is called, x must be saved for later, and the second time Q is called, u must be saved for later.

• The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

• Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

```asm
long P(long x, long y),
x in %rdi, y in %rsi:
push %rbp
push %rbx
mov %rdi,%rbp
mov %rsi,%rdi
callq 40056d <Q(long)>
mov %rax,%rbx
mov %rbp,%rdi
callq 40056d <Q(long)>
add %rbx,%rax
pop %rbx
pop %rbp
retq
```

• The first time Q is called, x must be saved for later, and the second time Q is called, u must be saved for later.
Local Storage in Registers

- The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

- Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

- The first time Q is called, \( x \) must be saved for later, and the second time Q is called, \( u \) must be saved for later.

```c
long P(long x, long y),
x in %rdi, y in %rsi:
push %rbp
push %rbx
mov %rdi,%rbp
mov %rsi,%rdi
callq 40056d <Q(long)>
mov %rax,%rbx
mov %rbp,%rdi
callq 40056d <Q(long)>
add %rbx,%rax
pop %rbx
pop %rbp
retq
```

put \( x \) into %rdi for call to Q
Local Storage in Registers

- The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

- Example:

```c
long P(long x, long y) {
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

- The first time Q is called, \(x\) must be saved for later, and the second time Q is called, \(u\) must be saved for later.
The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

The first time Q is called, x must be saved for later, and the second time Q is called, u must be saved for later.

```
long P(long x, long y),
x in %rdi, y in %rsi:
push %rbp
push %rbx
mov %rdi,%rbp
mov %rsi,%rdi
callq 40056d <Q(long)>
mov %rax,%rbx
mov %rbp,%rdi
callq 40056d <Q(long)>
add %rbx,%rax
pop %rbx
pop %rbp
retq
```
Local Storage in Registers

- The other registers are "callee" owned, meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

- Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

- The first time Q is called, \( x \) must be saved for later, and the second time Q is called, \( u \) must be saved for later.
Recursion!

- The conventions we have been discussing allow for functions to call themselves. Each procedure call has its own private space on the stack, and the local variables from all of the function calls do not interfere with each other.
- If written correctly, the only thing a program needs to worry about is a stack overflow, because the stack is a limited resource for a program.
- Example:

```c
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

```
0x400546 <+0>: cmp    $0x1,%rdi            comparison of 1:%rdi
0x40054a <+4>: jle    0x40055f <rfact+25>  if %rdi <= 1, jump to end
0x40054c <+6>: push   %rbx                 save %rbx
0x40054d <+7>: mov    %rdi,%rbx            store n in caller-owned reg
0x400550 <+10>: lea    -0x1(%rdi),%rdi      n--
0x400554 <+14>: callq  0x400546 <rfact>     recursive call
0x400559 <+19>: imul   %rbx,%rax            multiply n * return value (%rax)
0x40055d <+23>: jmp    0x400565 <rfact+31>  jump to end
0x40055f <+25>: mov    $0x1,%eax            store 1 in %eax
0x400564 <+30>: retq                        return
0x400565 <+31>: pop    %rbx                 restore %rbx
0x400566 <+32>: retq                        return
```
Array Allocation and Access

- Arrays in C map in a fairly straightforward way to X86 assembly code, thanks to the addressing modes available in instructions.
- When we perform pointer arithmetic, the assembly code that is produced will have address computations built into them.
- Optimizing compilers are very good at simplifying the address computations (in lab you saw another optimizing compiler benefit in the form of division — if the compiler can avoid dividing, it will!). Because of the transformations, compiler-generated assembly for arrays often doesn't look like what you are expecting.
- Consider the following form of a data type $T$ and integer constant $N$:

$$T \ A[N]$$

- The starting location is designated as $x_A$
- The declaration allocates $N \times \text{sizeof}(T)$ bytes, and gives us an identifier that we can use as a pointer (but it isn't a pointer!), with a value of $x_A$. 
Array Allocation and Access

• Example:

<table>
<thead>
<tr>
<th>Array</th>
<th>Element Size</th>
<th>Total Size</th>
<th>Start address</th>
<th>Element i</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[12]</td>
<td>1</td>
<td>12</td>
<td>$x_A$</td>
<td>$x_A + i$</td>
</tr>
<tr>
<td>B[8]</td>
<td>8</td>
<td>64</td>
<td>$x_B$</td>
<td>$x_B + 8i$</td>
</tr>
<tr>
<td>C[6]</td>
<td>4</td>
<td>24</td>
<td>$x_C$</td>
<td>$x_C + 4i$</td>
</tr>
<tr>
<td>D[5]</td>
<td>8</td>
<td>40</td>
<td>$x_D$</td>
<td>$x_D + 8i$</td>
</tr>
</tbody>
</table>

• The memory referencing operations in x86-64 are designed to simplify array access. Suppose we wanted to access $C[3]$ above. If the address of $C$ is in register $%rdx$, and 3 is in register $%rcx$

• The following copies $C[3]$ into $%eax$,

\[
\text{movl} \ (%rdx,%rcx,4), \ %eax
\]
C allows arithmetic on pointers, where the computed value is calculated according to the size of the data type referenced by the pointer.

The array reference $A[i]$ is identical to $*(A+i)$

Example: if the address of array $E$ is in `%rdx`, and the integer index, $i$, is in `%rcx`, the following are some expressions involving $E$:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>int *</td>
<td>$x_E$</td>
<td>movq %rdx, %rax</td>
</tr>
<tr>
<td>$E[0]$</td>
<td>int</td>
<td>$M[x_E]$</td>
<td>movl (%rdx), %eax</td>
</tr>
<tr>
<td>$E[i]$</td>
<td>int</td>
<td>$M[x_E+4i]$</td>
<td>movl (%rdx,%rcx,4), %eax</td>
</tr>
<tr>
<td>$&amp;E[2]$</td>
<td>int *</td>
<td>$x_E+8$</td>
<td>leaq 8(%rdx), %rax</td>
</tr>
<tr>
<td>$E+i-1$</td>
<td>int *</td>
<td>$x_E+4i-4$</td>
<td>leaq -4(%rdx,%rcx,4), %rax</td>
</tr>
<tr>
<td>$*(E+i-3)$</td>
<td>int</td>
<td>$M[x_E+4i-12]$</td>
<td>movl -12(%rdx,%rcx,4), %eax</td>
</tr>
<tr>
<td>$&amp;E[i]-E$</td>
<td>long</td>
<td>$i$</td>
<td>movq %rcx,%rax</td>
</tr>
</tbody>
</table>
Practice: x_S is the address of a `short` integer array, S, stored in `%rdx`, and a long integer index, `i`, is stored in register `%rcx`.

For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type `short`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>S+1</td>
<td>Pointer</td>
<td>+1</td>
<td></td>
</tr>
<tr>
<td>S[3]</td>
<td><code>short</code></td>
<td>S + 3 * sizeof(short)</td>
<td></td>
</tr>
<tr>
<td>&amp;S[i]</td>
<td>Pointer</td>
<td>S + i * sizeof(short)</td>
<td></td>
</tr>
<tr>
<td>S[4*i+1]</td>
<td><code>short</code></td>
<td>S + (4*i + 1) * sizeof(short)</td>
<td></td>
</tr>
<tr>
<td>S+i−5</td>
<td>Pointer</td>
<td>S + (i−5) * sizeof(long)</td>
<td></td>
</tr>
</tbody>
</table>
Pointer Arithmetic

- Practice: $x_S$ is the address of a `short` integer array, $S$, stored in `%rdx`, and a long integer index, $i$, is stored in register `%rcx`.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type `short`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S + 1$</td>
<td><code>short *</code></td>
<td>$x_S + 2$</td>
<td><code>leaq 2(%rdx),%rax</code></td>
</tr>
<tr>
<td>$S[3]$</td>
<td><code>short</code></td>
<td>$M[x_S + 6]$</td>
<td><code>movw 6(%rdx),%ax</code></td>
</tr>
<tr>
<td>$&amp;S[i]$</td>
<td><code>short *</code></td>
<td>$x_S + 2i$</td>
<td><code>leaq (%rdx,%rcx,2),%rax</code></td>
</tr>
<tr>
<td>$S[4*i+1]$</td>
<td><code>short</code></td>
<td>$M[x_S + 8i + 2]$</td>
<td><code>movw 2(%rdx,%rcx,8),%ax</code></td>
</tr>
<tr>
<td>$S+i-5$</td>
<td><code>short *</code></td>
<td>$x_S + 2i - 10$</td>
<td><code>leaq -10(%rdx,%rcx,2),%rax</code></td>
</tr>
</tbody>
</table>
Structures

• The C struct declaration is used to group objects of different types into a single unit.
• Each "field" is referenced by a name, and can be accessed using dot (.) or arrow (->) notation.
• Structures are kept in contiguous memory
• A pointer to a struct is to its first byte, and the compiler maintains the byte offset information for each field.
• In assembly, the references to the fields are via the byte offsets.
Structures

• Example:

```c
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

• This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>i</td>
<td>j</td>
<td>a[0]</td>
<td>a[1]</td>
<td>p</td>
</tr>
</tbody>
</table>

• The numbers along the top of the diagram are the byte offsets of the fields from the beginning of the structure.
• Note that the array is embedded in the structure.
• To access the fields, the compiler generates code that adds the field offset to the address of the structure.
Structures

- Example:

```c
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
</tr>
<tr>
<td>4</td>
<td>j</td>
</tr>
<tr>
<td>8</td>
<td>a[0]</td>
</tr>
<tr>
<td>16</td>
<td>a[1]</td>
</tr>
<tr>
<td>24</td>
<td>p</td>
</tr>
</tbody>
</table>

- Example: Variable `r` of type `struct rec *` is in register `%rdi`. The following copies element `r->i` to element `r->j`:

```
  movl (%rdi), %eax  // get r->i
  movl %eax, 4(%rdi)  // store in r->j
```

- The offset of `i` is 0, so `i`'s field is `%rdi`. The office of `j` is 4, so the offset of 4 is added to the address of `%rdi` to store into `j`. 
• Example:

```c
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

• This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>i</td>
<td>j</td>
<td>a[0]</td>
<td>a[1]</td>
<td>p</td>
</tr>
</tbody>
</table>
```

• We can generate a pointer to a field by adding the field's offset to the struct address. To generate `&(r->a[1])` we add offset `8 + 4 = 12`. For a pointer `r` in register `%rdi` and long `int` variable `i` in `%rsi`, we can generate the pointer value `&(r->a[i])` with one instruction:

```
leaq 8(%rdi,%rsi,4), %rax // set %rax to &r->a[i]
```
Example:

```c
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
</tr>
<tr>
<td>4</td>
<td>j</td>
</tr>
<tr>
<td>8</td>
<td>a[0]</td>
</tr>
<tr>
<td>16</td>
<td>a[1]</td>
</tr>
<tr>
<td>24</td>
<td>p</td>
</tr>
</tbody>
</table>

The following code implements `r->p = &r->a[r->i + r->j];`

```assembly
// r in %rdi
movl 4(%rdi),%eax      // get r->j
addl (%rdi),%eax       // add r->i
cltq                   // extend %rax to 8 bytes
leaq 8(%rdi,%rax,4), %rax // compute &r->a[r->i + r->j]
movq %rax, 16(%rdi)     // store in r->p
```
Structures

- Example:

```c
c struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

- This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte `int` pointer, for a total of 24 bytes:

```
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>j</td>
</tr>
<tr>
<td>a[0]</td>
</tr>
<tr>
<td>a[1]</td>
</tr>
<tr>
<td>p</td>
</tr>
</tbody>
</table>
```

- Notice that all struct manipulation is handled at compile time, and the machine code doesn't contain any information about the field declarations or the names of the fields.
- The compiler does all the work, keeping track as it produces the assembly code.
- BTW, if you're curious about how the compiler actually does the transformation from C to assembly, take a compilers class, e.g., CS143.
Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value $K$ (normally 2, 4, or 8).

These alignment restrictions simplify the design of the hardware.

For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any double will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.

For x86-64, Intel recommends the following alignments for best performance:

<table>
<thead>
<tr>
<th>$K$</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
</tr>
<tr>
<td>8</td>
<td>long, double, char *</td>
</tr>
</tbody>
</table>
Data Alignment

• The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.
• For example, let's look at the following struct:

```c
struct S1 {
    int i;
    char c;
    int j;
};
```
• If the compiler used a minimal allocation:
  • This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:

![Alignment Diagram]

• So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!
Let's look at the following code:

```c
void *gfind_max(void *arr, int n, size_t elemsz,
                 int (*compar)(const void *, const void *))
{
    void *pmax = arr;
    for (int i = 1; i < n; i++) {
        void *ith = (char *)arr + i*elemsz;
        if (compar(ith, pmax) > 0)
            pmax = ith;
    }
    return pmax;
}

int cmp_alpha(const void *p, const void *q)
{
    const char *first = *(const char **)p;
    const char *second = *(const char **)q;
    return strcmp(first, second);
}

int main(int argc, char *argv[])
{
    char **pmax = gfind_max(argv+1, argc-1, sizeof(argv[0]), cmp_alpha);
    printf("max = %s\n", *pmax);
    return 0;
}
```
Function Pointers in Assembly

• Let's look at the following code:

```c
void *gfind_max(void *arr, int n, size_t elemsz,
                 int (*compar)(const void *, const void *))
{
    void *pmax = arr;
    for (int i = 1; i < n; i++) {
        void *ith = (char *)arr + i*elemsz;
        if (compar(ith, pmax) > 0) {
            pmax = ith;
        }
    }
    return pmax;
}

int cmp_alpha(const void *p, const void *q)
{
    const char *first = *(const char **)p;
    const char *second = *(const char **)q;
    return strcmp(first, second);
}

int main(int argc, char *argv[])
{
    char **pmax = gfind_max(argv+1, argc-1, sizeof(argv[0]), cmp_alpha);
    printf("max = %s\n", *pmax);
    return 0;
}
```

• Because `compar` is a function pointer, the compiler calls the function via the address that is in the compar variable.
• Let's take a look at this in gdb.
References and Advanced Reading

References:
• Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
• CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
• gdbtui: https://beej.us/guide/bgdb/
• More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
• Compiler explorer: https://gcc.godbolt.org

Advanced Reading:
• Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
• history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
• x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64