Procedures II, Executables
CSE 351 Summer 2019

Instructor: Sam Wolfson
Teaching Assistants: Rehaan Bhimani Corbin Modica Daniel Hsu

http://xkcd.com/1790/
Adminstrivia

- Lab 2 (x86-64) due Monday (7/22)
- Homework 3, due Monday (7/29)
  - On midterm material, but due after the midterm

- **Midterm** (Fri 7/26, 10:50-11:50am)
  - One double-sized handwritten page of notes allowed
  - Reference sheet will also be provided
Example: `increment`

```c
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; arg (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; arg (val), y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, return value</td>
</tr>
</tbody>
</table>
Procedure Call Example  (initial state)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

Initial Stack Structure

- Return address on stack is the address of instruction immediately following the call to “call_incr”
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by `call call_incr`

Return address on stack is the address of instruction immediately following the call to “call_incr”

- Shown here as `main`, but could be anything
- Pushed onto stack by `call call_incr`
Procedure Call Example (step 1)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Stack Structure**

- Allocate space for local vars
- Setup space for local variables
  - Only `v1` needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment

```
call_incr:
  subq $16, %rsp
  movq $351, 8(%rsp)
  movl $100, %esi
  leaq 8(%rsp), %rdi
  call increment
  addq 8(%rsp), %rax
  addq $16, %rsp
  ret
```
### Procedure Call Example (step 2)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Stack Structure**

- Return addr `<main+8>`
- `351` ← `%rsp+8`
- `Unused` ← `%rsp`

#### Set up parameters for call to `increment`

**Register** | **Use(s)**
---|---
%rdi | &v1
%rsi | 100

*Aside:* `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `%rsi` get set to zero automatically. It takes *one less byte* to encode a `movl` than a `movq`.

---

**call_incr:**

- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`
Procedure Call Example (step 3)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

Stack Structure

- Return addr `<main+8>`
- 351
- Unused
- Return addr `<call_incr+?>` ← %rsp

- State while inside `increment`
  - Return address on top of stack is address of the `addq` instruction immediately following call to `increment`

**Register** | **Use(s)**
--- | ---
%rdi | &v1
%rsi | 100
%rax |
Procedure Call Example (step 4)

long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}

call_incr:
  subq $16, %rsp
  movq $351, 8(%rsp)
  movl $100, %esi
  leaq 8(%rsp), %rdi
  call increment
  addq 8(%rsp), %rax
  addq $16, %rsp
  ret

Stack Structure

increment:
  movq (%rdi), %rax # x = *p
  addq %rax, %rsi # y = x+100
  movq %rsi, (%rdi) # *p = y
  ret

State while inside increment
- After code in body has been executed

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- After returning from call to `increment`:
  - Registers and memory have been modified and return address has been popped off stack.

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 6)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

Stack Structure

<table>
<thead>
<tr>
<th>Return addr &lt;main+8&gt;</th>
<th>←%rsp+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>451</td>
<td>←%rsp</td>
</tr>
</tbody>
</table>

Unused

Update %rax to contain v1+v2

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>451+351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

### Stack Structure

- Return addr `<main+8>`
- `451` unused

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td><code>&amp;v1</code></td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td><code>451</code></td>
</tr>
<tr>
<td><code>%rax</code></td>
<td><code>802</code></td>
</tr>
</tbody>
</table>

De-allocate space for local vars
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- State *just before* returning from call to `call_incr`

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>

Stack Structure

Return addr `<main+8>` ← `%rsp`
Procedure Call Example (step 9)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

State immediately after returning from call to call_incr
- Return addr has been popped off stack
- Control has returned to the instruction immediately following the call to call_incr (not shown here)

**Final Stack Structure**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the caller
  - `who` is the callee

- Can registers be used for temporary storage?

```
<table>
<thead>
<tr>
<th>yoo:</th>
<th>who:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>movq</strong> $15213, %rdx</td>
<td><strong>subq</strong> $18213, %rdx</td>
</tr>
<tr>
<td><strong>call</strong> who</td>
<td></td>
</tr>
<tr>
<td><strong>addq</strong> %rdx, %rax</td>
<td><strong>addq</strong> %rdx, %rax</td>
</tr>
<tr>
<td></td>
<td><strong>ret</strong></td>
</tr>
</tbody>
</table>
```

- No! Contents of register `%rdx` overwritten by `who`!
- This could be trouble – something should be done. Either:
  - Caller should save `%rdx` before the call (and restore it after the call)
  - Callee should save `%rdx` before using it (and restore it before returning)
Register Saving Conventions

❖ **“Caller-saved” registers**
  - It is the caller’s responsibility to save any important data in these registers before calling another procedure (i.e. the callee can freely change data in these registers)
  - Caller saves values in its stack frame before calling Callee, then restores values after the call

❖ **“Callee-saved” registers**
  - It is the callee’s responsibility to save any data in these registers before using the registers (i.e. the caller assumes the data will be the same across the callee procedure call)
  - Callee saves values in its stack frame before using, then restores them before returning to caller
Silly Register Convention Analogy

1) Parents *(caller)* leave for the weekend and give the keys to the house to their child *(callee)*
   - Being suspicious, they put away/hid the valuables *(caller-saved)* before leaving
   - Warn child to leave the bedrooms untouched: “These rooms better look the same when we return!”

2) Child decides to throw a wild party, spanning the entire house
   - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed *(callee-saved)* before the guests trash the house
   - Child cleans up house after the party and moves stuff back to bedrooms

3) Parents return home and are satisfied with the state of the house
   - Move valuables back and continue with their lives
x86-64 Linux Register Usage, part 1

- %rax
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure

- %rdi, ..., %r9
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure

- %r10, %r11
  - **Caller**-saved & restored
  - Can be modified by procedure
x86-64 Linux Register Usage, part 2

- **%rbx, %r12, %r13, %r14, %r15**
  - **Callee**-saved
  - **Callee** must save & restore

- **%rbp**
  - **Callee**-saved
  - **Callee** must save & restore
  - May be used as frame pointer
  - Can mix & match

- **%rsp**
  - Special form of **callee** save
  - Restored to original value upon exit from procedure
# x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Saved By</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Saved By</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r10</td>
<td></td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
Callee-Saved Example (step 1)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

**Initial Stack Structure**

```
%rsp
... ret addr
```

**Resulting Stack Structure**

```
%rsp+8
Saved %rbx

351
Unused
```

call_incr2:
```
pushq %rbx
subq $16, %rsp
movq %rdi, %rbx
movq $351, 8(%rsp)
movl $100, %esi
leaq 8(%rsp), %rdi
call increment
addq %rbx, %rax
addq $16, %rsp
popq %rbx
ret
```
Callee-Saved Example (step 2)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

**call_incr2:**
- `pushq %rbx`
- `subq $16, %rsp`
- `movq %rdi, %rbx`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq %rbx, %rax`
- `addq $16, %rsp`
- `popq %rbx`
- `ret`

**Stack Structure**
- ...  
- Rtn address
- Saved `%rbx`
- 351  
- Unused
- `%rsp+8`

**Pre-return Stack Structure**
- ...  
- Rtn address
- `%rsp`
Why Caller and Callee Saved?

- We want one calling convention to simply separate implementation details between caller and callee.

- In general, neither caller-save nor callee-save is “best”:
  - If caller isn’t using a register, caller-save is better.
  - If callee doesn’t need a register, callee-save is better.
  - If “do need to save”, callee-save generally makes smaller programs.
    - Functions are called from multiple places.

- So... “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring.
Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values

- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don’t forget to restore/pop the values later!
Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Recursive Function

/*@ Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

Compiler Explorer:
https://godbolt.org/z/xFCrsw
• Compiled with −O1 for brevity instead of −Og
• Try −O2 instead!

Popcount: number of “1”s in the binary representation of x.
Recursive Function: Base Case

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>

Trick because some AMD hardware doesn’t like jumping to ret
Recursive Function: **Callee Register Save**

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Need original value of `x` after recursive call to `pcount_r`.

“Save” by putting in `%rbx` (**callee** saved), but need to save old value of `%rbx` before you change it.

**Register Use(s) Type**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
</tbody>
</table>

**The Stack**

```
...                
rtn <main+?>       
Saved %rbx        
%rbx → %rsp
```

**pcount_r:**

```asm
movl $0, %eax
testq %rdi, %rdi
jne .L8
rep ret
.L8
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
```
Recursive Function: Call Setup

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x (new)</td>
<td>Argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

```
Rtn <main+?>
saved %rbx

%rsp \rightarrow
```

The Stack

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    rep ret
    .L8
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```
Recursive Function: Call

/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

The Stack

Register | Use(s) | Type
---|---|---
%rax | Recursive call return value | Return value
%rbx | x (old) | Callee saved

pcount_r:
- movl $0, %eax
- testq %rdi, %rdi
- jne .L8
- rep ret .L8
- pushq %rbx
- movq %rdi, %rbx
- shrq %rdi
- call pcount_r
- andl $1, %ebx
- addq %rbx, %rax
- popq %rbx
- ret
Recursive Function: Result

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>x&amp;1</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

**The Stack**

```
pcount_r:  
  movl $0, %eax  
  testq %rdi, %rdi  
  jne .L8  
  rep ret  
  .L8  
  pushq %rbx  
  movq %rdi, %rbx  
  shrq %rdi  
  call pcount_r  
  andl $1, %ebx  
  addq %rbx, %rax  
  popq  
  ret
```

%rsp →

saved %rbx
Recursive Function: Completion

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Previous %rbx value</td>
<td>Callee restored</td>
</tr>
</tbody>
</table>

The Stack

```assembly
movl $0, %eax
testq %rdi, %rdi
jne .L8
rep ret
.L8
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
```
Observations About Recursion

- Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return address
  - Register saving conventions prevent one function call from corrupting another’s data
    - Unless the code explicitly does so (e.g. buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)
- Also works for mutual recursion (P calls Q; Q calls P)
Many x86-64 procedures have a minimal stack frame
- Only return address is pushed onto the stack when procedure is called

A procedure *needs* to grow its stack frame when it:
- Has too many local variables to hold in `caller`-saved registers
- Has local variables that are arrays or structs
- Uses `&` to compute the address of a local variable
- Calls another function that takes more than six arguments
- Is using `caller`-saved registers and then calls a procedure
- Modifies/uses `callee`-saved registers
x86-64 Procedure Summary

- **Important Points**
  - Procedures are a combination of instructions and conventions
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions
- **Heavy use of registers**
  - Faster than using memory
  - Use limited by data size and conventions
- **Minimize use of the Stack**

### Diagram
- **Caller Frame**
  - Arguments
    - 7+
  - Return Addr
    - Old %rbp
  - %rbp (Optional)
  - Saved Registers
    - +
    - Local Variables
  - Argument Build
    - %rsp
Roadmap

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:

```
get_mpg:
  pushq %rbp
  movq %rsp, %rbp
  ...
  popq %rbp
  ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000000111111001000011111
```

OS:

- Windows 10
- OS X Yosemite

Computer system:

Memory & data
Integers & floats
x86 assembly
Executables
Arrays & structs
Processes
Virtual memory
Memory allocation
Java vs. C
Building an Executable from a C File

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Put resulting machine code in file `p`
- Run with command: `./p`

```
C program (p1.c p2.c) -> Compiler (gcc -Og -S)

Asm program (p1.s p2.s) -> Assembler (gcc -c or as)

Object program (p1.o p2.o) -> Linker (gcc or ld)

Executable program (p) -> Static libraries (.a)
```

Loader (the OS)
Compiler

- **Input:** Higher-level language code (*e.g.* C, Java)
  - `foo.c`

- **Output:** Assembly language code (*e.g.* x86, ARM, MIPS)
  - `foo.s`

- First there’s a preprocessor step to handle `#`directives
  - Macro substitution, plus other specialty directives

- Super complex, whole courses (*e.g.*, CSE 401) devoted to these!

- Compiler optimizations
  - “Level” of optimization specified by capital ‘O’ flag (*e.g.* `-Og`, `-O3`)
Compiling Into Assembly

- **C Code (sum.c)**
  ```c
  void sumstore(long x, long y, long *dest) {
    long t = x + y;
    *dest = t;
  }
  ```

- **x86-64 assembly (gcc -Og -S sum.c)**
  - Generates file sum.s (see [https://godbolt.org/g/o34FHp](https://godbolt.org/g/o34FHp))
    ```assembly
    sumstore(long, long, long*):
    addq  %rdi, %rsi
    movq  %rsi, (%rdx)
    ret
    ```

**Warning**: You may get different results with other versions of gcc and different compiler settings
Assembler

- **Input:** Assembly language code (e.g. x86, ARM, MIPS)
  - foo.s
- **Output:** Object files (e.g. ELF, COFF)
  - foo.o
  - Contains *object code* and *information tables*
- Reads and uses *assembly directives*
  - *e.g.* .text, .data, .quad
  - x86: [https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html](https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html)
- Produces “machine language”
  - Does its best, but object file is *not* a completed binary
- **Example:** gcc -c foo.s
Producing Machine Language

- **Simple cases:** arithmetic and logical operations, shifts, etc.
  - All necessary information is contained in the instruction itself

- What about the following?
  - Conditional jump
  - Accessing static data (e.g. global var or jump table)
  - call

- Addresses and labels are problematic because final executable hasn’t been constructed yet!
  - So how do we deal with these in the meantime?
Object File Information Tables

- **Symbol Table** holds list of “items” that may be used by other files
  - *Non-local labels* – function names for call
  - *Static Data* – variables & literals that might be accessed across files

- **Relocation Table** holds list of “items” that this file needs the address of later (currently undetermined)
  - Any label or piece of *static data* referenced in an instruction in this file
    - Both internal and external

- Each file has its own symbol and relocation tables
Object File Format

1) **object file header**: size and position of the other pieces of the object file
2) **text segment**: the machine code
3) **data segment**: data in the source file (binary)
4) **relocation table**: identifies lines of code that need to be “handled”
5) **symbol table**: list of this file’s labels and data that can be referenced
6) **debugging information**

*More info: ELF format*
- [http://www.skyfree.org/linux/references/ELF_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)
Linker

- **Input:** Object files (e.g. ELF, COFF)
  - foo.o
- **Output:** executable binary program
  - a.out

- Combines several object files into a single executable (*linking*)
- Enables separate compilation/assembling of files
  - Changes to one file do not require recompiling of whole program
Linking

1) Take text segment from each .o file and put them together
2) Take data segment from each .o file, put them together, and concatenate this onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry
Disassembling Object Code

- **Disassembled:**

  
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000400536 &lt;sumstore&gt;:</td>
<td>400536:</td>
<td>48 01 fe</td>
</tr>
<tr>
<td></td>
<td>400539:</td>
<td>48 89 32</td>
</tr>
<tr>
<td></td>
<td>40053c:</td>
<td>c3</td>
</tr>
</tbody>
</table>

- **Disassembler** (objdump -d sum)
  - Useful tool for examining object code (man 1 objdump)
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can run on either a.out (complete executable) or .o file
What Can be Disassembled?

- Anything that can be interpreted as executable code
- Disassembler examines bytes and attempts to reconstruct assembly source

```bash
% objdump -d WINWORD.EXE

WINWORD.EXE:  file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55             push   %ebp
30001001:  8b  ec          mov   %esp,%ebp
30001003:  6a ff          push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91

ReVeRsE eNgIneErInG FoRbIdDeN bY tHe MiCrOsOfT uSeR llcEnSiNg AgReEmEnT
```
Loader

- **Input:** executable binary program, command-line arguments
  - ./a.out arg1 arg2
- **Output:** <program is run>

- Loader duties primarily handled by OS/kernel
  - More about this when we learn about processes
  - For even more, take operating systems 😊
- Memory sections (Instructions, Static Data, Stack) are set up
- Registers are initialized