Assembly - You Can Do It!
アセンブリ - 頑張りましよ！

try! Swift Tokyo
２２日3月２０１９

Andrew Madsen
マドセン•アンドリュ

About Me 自己紹介

まずかんたんなじこしょうかいをします。

About Me 自己紹介

Lifelong Programmer
子どもの頃からプログラマーです

私はこどものころからプログラマーです。このしゃっしんには私は５さいでした。

About Me 自己紹介

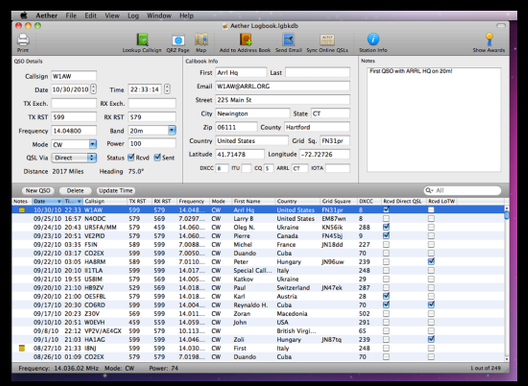
Electrical Engineer
電気技師です

でんきこうがくもすきです。だいがくで電気工学をべんきょうしました。大学のあと、しごとでロボットをデザインしました。

About Me 自己紹介

I've been doing Cocoa development since 2005.

2005年からCocoaプログラミングをしてます。



２００５年にCocoaプログラミングをはじめました。そしたら、２００８年にiOSがとうじょうしてiOSプログラミングを始めました。

いましごとはiOSプログラミングをおしえることです。LambdaSchoolというがっこではたらいています。このしごとをだいすきです。

About Me 自己紹介

I collect old computers
レトロパソコンを集めます

すきなしゅみはレトロパソコンをあつめことです。ほとんどにAppleものを集めます。

About Me 自己紹介

15 years ago, I lived in Japan.
I’ve loved it ever since.

１５年前に日本に住んでいました。
その時から日本が大好きです

１５年まえに日本に住んでいました。日本が大好きで　日本にいるのでほんとにうれしいです。

- Goal
- Intimidated
- Show you the basics, learn more
- Show you how it can help you

Let's talk about assembly. My goal here is not to teach you everything there is to know about assembly. After all, we only have 20 minutes. But I think most programmers are a little intimidated by assembly language. I want to show you the basics so that you won't be scared of it, and instead will be ready to learn more. I also want to show you how knowing assembly can help you be a better iOS developer and debugger.

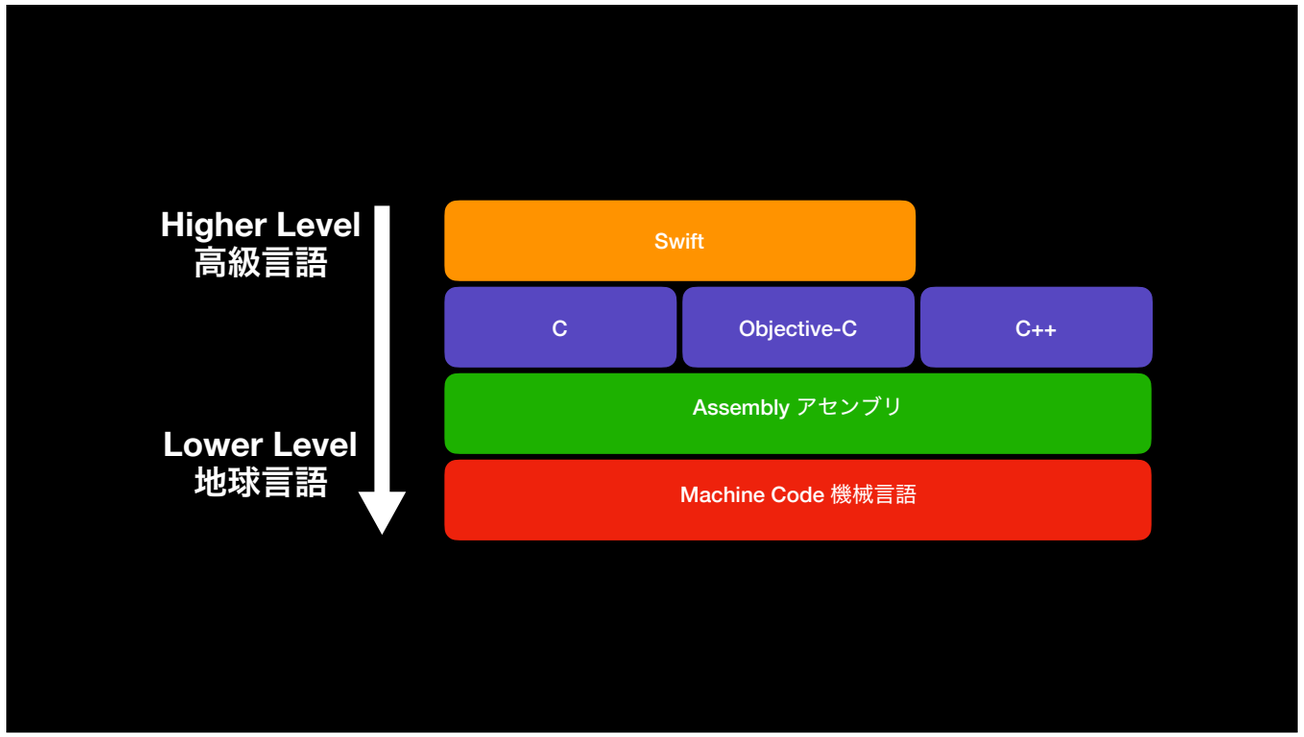
Assembly
アセンブリ 😍

Why Learn Assembly?

なぜアセンブリ言語を学んだ方がいいですか？

- Don't usually need to write
- Understanding can help you with:
    - Debugging
    - Exploring Apple frameworks

You don’t often need to actually write assembly, unless you’re doing *extremely* performance intensive work, or working on very low-power embedded processors. However, understanding assembly can help you with debugging. You can also explore Apple’s frameworks to understand exactly what they’re doing inside. Knowing a little assembly goes a long way when trying to debug tricky problems, or to understand exactly _how_ something works.



- Machine code
- Assembly is almost a direct representation of machine code
- When you compile, you get machine code
- Disassemble to see it in assembly

At the lowest level, the CPU in every device runs machine code, which is just binary numbers. Assembly is very close to being a direct representation of machine code, it's just written in a way that's easier for humans to read, write, and understand. When you compile a language like Swift, C, Objective-C, or C++, the compiler outputs a machine code executable. Various tools will "disassemble" this and show it to you as if it were assembly. This allows you to look at compiled code and understand what it's doing without the source.

Registers
レジスター

In order to understand assembly, there are some vocabulary words we need to learn. The first is "register".

Registers レジスター

- Small, individual pieces of memory
- How many there are, names, and sizes depends on the CPU
- Some are general purpose, some dedicated, some have usage specified by convention
- Directly inside the CPU, unlike RAM
- *Fast*

A register is a small, individual piece of memory. On x86_64, which we'll be talking about today, each register can hold an 8-byte (64-bit) value. The specifics of registers, their names, sizes, and what they're used for, varies depending on the CPU architecture and platform you're on. Some registers are general purpose, meaning you can use them for whatever you'd like. Some are dedicated to a specific usage by the CPU. And some have usage that is specified by convention. Registers are directly inside the CPU, unlike RAM, which means that accessing them is very fast.

Registers レジスター

You can think of them like variables.
レジスタは可変と同じぐらいです

Think of registers like temporary variables in assembly code.

Registers レジスター

rax	r8
rbx	r9
rcx	r10
rdx	r11
rbp	r12
rsp	r13
rsi	r14
rdi	r15

On x86_64, there are 16 registers you should know about. They are rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, and r8-r15. We'll talk about what a few of these are used for later on.

Mnemonics
ニーモニック

In assembly language, each line of code is a single *instruction*. The CPU defines the specific instructions you can use. We use mnemonics to refer to instructions in assembly.

Mnemonic

A mnemonic is an easy to remember name for a CPU **instruction**.

ニーモニック

ニーモニックは覚えやすい**CPU命令**の名前です。

A mnemonic is an easy to remember name for a CPU instruction. Lets look at some examples.

Mnemonics ニーモニック

mov
add
sub
cmp
jne
nop
call
pop
ret

- Here are few common instruction mnemonics from the x86_64
- mov
- add
- sub
- cmp
- jne
- nop
- pop
- ret

Here are a few common instruction mnemonics from the x86_64 instruction set. mov moves data from one place to another. add adds two things together. sub subtracts two things. cmp compares two numbers (essentially performing subtraction). jne stands for jump (if) not equal, and allows you to jump to a different part of the code based on the results of the previous cmp instruction. nop is a "nothing" instruction. It tells the CPU to take a break and not do anything this cycle, and can be used to introduce intentional delays. call is used to call a function/subroutine. pop pops the stack, and ret returns from a subroutine to the calling code.

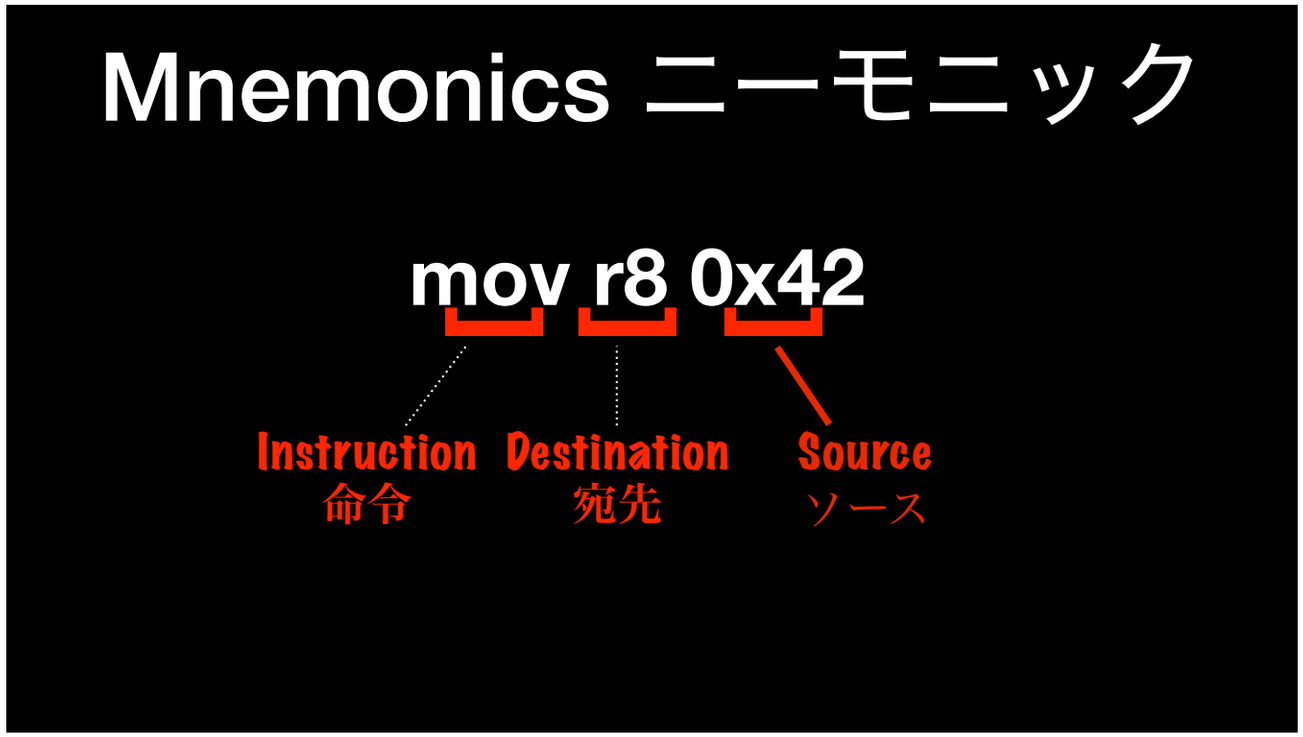
This list is *far* from exhaustive, but all of these instructions are used frequently.

Mnemonics ニーモニック

mov r8 0x42
add r8 0x01
sub r8 0x01
cmp rdi 0x42
jne <label_or_addr>
nop
call <label_or_addr>
pop
ret

In assembly language, each line of code will have an instruction followed by between 0 and 2 operands. For example, here mov r8 0x42 means mov (or copy) the hex value 0x42 into register r8. add r8 0x01 means add one to the value in r8 (storing the result in r8).

Note that there are two common syntax formats used for assembly, Intel, and AT&T. While AT&T is the default in Xcode, I find Intel syntax easier to read, and will be using that in this talk.



In Intel syntax, for instructions that take two operands, the destination of the operation (a register, memory location, etc.) is always the first operand, and the source of the operation is the second operand.

Calling Convention
呼出規約

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return a + b;
}
```

When some code calls a function, it has to pass in its arguments, and retrieve the value the function returns. This is done by having the calling code put the argument values in registers, while the function puts its return value in a register. Which registers to use for the arguments and return value have to be agreed upon by the calling code and the function being called. “Calling convention” refers to the agreed upon registers to use for function arguments and return values. The specifics of the calling convention depend on the CPU and platform. On UNIX systems running on x86_64 like iOS and OS X, the standard calling convention is the System V calling convention.

Calling Convention 呼出規約

argument 1	rdi
argument 2	rsi
argument 3	rdx
argument 4	rcx
argument 5	r8
argument 6	r9
return value	rax

Here’s a list of arguments and the register used to pass them. If a function takes more than 6 arguments, additional arguments are passed on the stack. It’s helpful to memorize this list, because you can use them in debugging.

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return a + b;
}
```

arg 1	rdi	
arg 2	rsi	
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	

Let’s take a look at how this works.

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return a + b;
}
```

arg 1	rdi	
arg 2	rsi	
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	

First, the arguments to the add call must be placed in the correct registers. According to the calling convention, the first argument, 2, goes in rdi.

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return a + b;
}
```

arg 1	rdi	2
arg 2	rsi	
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	

The second argument, 3, goes in rsi.

Calling Convention 呼出規約

```
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return a + b;
}
```

arg 1	rdi	2
arg 2	rsi	3
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	

Then, the add() function is called, and starts executing.

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return a + b;
}
```

arg 1	rdi	2
arg 2	rsi	3
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	

It needs to retrieve its arguments. So, it gets a (2) from rdi.

Calling Convention 呼出規約

```
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return 2 + b;
}
```

arg 1	rdi	2
arg 2	rsi	3
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	

And it gets the second argument, b, from rsi.

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return 2 + 3;
}
```

arg 1	rdi	2
arg 2	rsi	3
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	

It performs its calculation, then puts the return value (5) in rax.

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return 2 + 3;
}
```

arg 1	rdi	2
arg 2	rsi	3
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	5

Control is returned to the call site in main().

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return 2 + 3;
}
```

arg 1	rdi	2
arg 2	rsi	3
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	5

Now, to get the return value to assign to c, the value in the rax register is retrieved.

Calling Convention 呼出規約

```c
int main(int argc, char *argv[]) {
 long c = 5 add(2, 3);
 printf("%li", c);
}


long add(long a, long b) {
 return 2 + 3;
}
```

arg 1	rdi	2
arg 2	rsi	3
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	5

Calling Convention 呼出規約

```
int main(int argc, char *argv[]) {
 long c = add(2, 3);
 printf("%li", 5);
}


long add(long a, long b) {
 return 2 + 3;
}
```

arg 1	rdi	2
arg 2	rsi	3
arg 3	rdx	
arg 4	rcx	
arg 5	r8	
arg 6	r9	
return value	rax	5

And the execution proceeds to the next line.

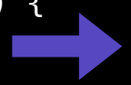
Addition 加算

```
long add(long a, long b) {
 return a + b;
}
```

Let's look at a very simple function that takes two arguments, adds them together and returns the result. We're writing this in C, but it would look very similar written in Swift.

Addition 加算

```
long add(long a, long b) {
 return a + b;
}
```



```
_add:
0 push      rbp
1 mov       rbp, rsp
2 mov       qword [rbp-8], rdi
3 mov       qword [rbp-16], rsi
4 mov       rsi, qword [rbp-8]
5 add       rsi, qword [rbp-16]
6 mov       rax, rsi
7 pop       rbp
8 ret
```

After compiling this function, if we look at the assembly, it will look like this. Note that despite this function only being one line long, it's 9 lines long in (unoptimized) assembly! However, we can break this down line by line and understand what's going on. Let's dive in.

Addition 加算

```
_add:
0 push      rbp
1 mov       rbp, rsp
2 mov       qword [rbp-8], rdi
3 mov       qword [rbp-16], rsi
4 mov       rsi, qword [rbp-8]
5 add       rsi, qword [rbp-16]
6 mov       rax, rsi
7 pop       rbp
8 ret
```

Addition 加算

```
_add:
0 push        rbp                                  Set up stack
1 mov         rbp, rsp                             スタックを設定する
2 mov         qword [rbp-8], rdi
3 mov         qword [rbp-16], rsi
4 mov         rsi, qword [rbp-8]
5 add         rsi, qword [rbp-16]
6 mov         rax, rsi
7 pop         rbp
8 ret
```

At the beginning of the function is what is called the *function prologue*. The function prolog sets some things up for the body of the function to execute. In particular, first we set up the stack by pushing the existing stack pointer onto the stack, then saving a new base pointer.

Addition 加算

```
_add:
0 push      rbp
1 mov       rbp, rsp
2 mov       qword [rbp-8], rdi
3 mov       qword [rbp-16], rsi
4 mov       rsi, qword [rbp-8]
5 add       rsi, qword [rbp-16]
6 mov       rax, rsi
7 pop       rbp
8 ret
```

Get Arguments
引き数をフェッチする

Next, we get the arguments. Remember, that according to calling convention, rdi contains the first argument to a function, and rsi contains the second argument. We copy these arguments on to the stack.

Addition 加算

```
_add:
0 push      rbp
1 mov       rbp, rsp
2 mov       qword [rbp-8], rdi
3 mov       qword [rbp-16], rsi
4 mov       rsi, qword [rbp-8]
5 add       rsi, qword [rbp-16]
6 mov       rax, rsi
7 pop       rbp
8 ret
```

Addition 加算

```
_add:
0 push        rbp
1 mov         rbp, rsp
2 mov         qword [rbp-8], rdi
3 mov         qword [rbp-16], rsi
4 mov         rsi, qword [rbp-8]
5 add         rsi, qword [rbp-16]
6 mov         rax, rsi
7 pop         rbp
8 ret
```

Load a into rsi
rsiにaを収納する

Now, we're ready to actually to the addition. First, we copy `a` into the rsi register. Remember that rbp-8 here contains the value of the a argument due to line 2.

Addition 加算

```
_add:
0 push      rbp
1 mov       rbp, rsp
2 mov       qword [rbp-8], rdi
3 mov       qword [rbp-16], rsi
4 mov       rsi, qword [rbp-8]
5 add       rsi, qword [rbp-16]
6 mov       rax, rsi
7 pop       rbp
8 ret
```

Add b to rsi
rsiにbを加算する

Next, we'll use the add instruction to add b to the value in rsi (a). rbp-16 contains the value of b, because of line 3.

Addition 加算

```
_add:
0 push        rbp
1 mov         rbp, rsp
2 mov         qword [rbp-8], rdi
3 mov         qword [rbp-16], rsi
4 mov         rsi, qword [rbp-8]
5 add         rsi, qword [rbp-16]
6 mov         rax, rsi
7 pop         rbp
8 ret
```

Load rax (return register)
rax (戻りレジスタ)を収納する

Finally, we'll mov the value in rsi into rax. Calling convention specifies that rax should hold the return value of the function, and the code that called our `add()` function is going to look there.

Addition 加算

```
_add:
0 push        rbp
1 mov         rbp, rsp
2 mov         qword [rbp-8], rdi
3 mov         qword [rbp-16], rsi
4 mov         rsi, qword [rbp-8]
5 add         rsi, qword [rbp-16]
6 mov         rax, rsi
7 pop         rbp
8 ret
```

Restore the stack
スタックを戻す

Finally, restore the stack base pointer to what it was when the function started, cleaning up after ourselves.

Addition 加算

```
_add:
0 push      rbp
1 mov       rbp, rsp
2 mov       qword [rbp-8], rdi
3 mov       qword [rbp-16], rsi
4 mov       rsi, qword [rbp-8]
5 add       rsi, qword [rbp-16]
6 mov       rax, rsi
7 pop       rbp
8 ret
```

Return
戻る

And we can return from the function!

Debugging with Assembly
アセンブリでデバッギング

Let’s talk about how you can use assembly to actually help you as an iOS developer. It comes up most often in debugging. Have you ever been debugging only to experience a crash or error in system code, where you see a bunch of assembly? With just a little knowledge of assembly, you can often find useful information.

Debugging with Assembly
アセンブリでデバッギング

In Objective-C Mode:
Objective-Cのモード:

```
po $rdi
```

In lldb, in Objective-C mode, you can print a register by prefixing its name with a dollar sign ($). For example, here’s how you’d print the value in the rdi register.

Debugging with Assembly
アセンブリでデバッギング

```
(lldb)
```

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
```

Let’s say we want to break whenever the touchesBegan: method is called on any object. We can create a breakpoint with the b command in lldb, passing the Objective-C method we want to break on.

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
Breakpoint 3: where = UIKitCore`-[UIResponder touchesBegan:withEvent:], address = 0x000000010e7bc745
(lldb)
```

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
Breakpoint 3: where = UIKitCore`-[UIResponder touchesBegan:withEvent:], address = 0x000000010e7bc745
(lldb) po $rdi
```

Now, when our program stops because the breakpoint is hit, we want to know which object it was called on. Because it’s Objective-C (or dynamically dispatched Swift), the function being called is actually objc_msgSend(), and the first argument is the receiver of the message. So, we can print out the rdi register to find out what object the method was called on.

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
Breakpoint 3: where = UIKitCore`-[UIResponder touchesBegan:withEvent:], address = 0x000000010e7bc745
(lldb) po $rdi
<UITableViewCellContentView: 0x7fbf76f24160; frame = (0 0; 414 55.5); opaque = NO; gestureRecognizers
= <NSArray: 0x600002607180>; layer = <CALayer: 0x6000028359c0>>
(lldb)
```

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
Breakpoint 3: where = UIKitCore`-[UIResponder touchesBegan:withEvent:], address = 0x000000010e7bc745
(lldb) po $rdi
<UITableViewCellContentView: 0x7fbf76f24160; frame = (0 0; 414 55.5); opaque = NO; gestureRecognizers
= <NSArray: 0x600002607180>; layer = <CALayer: 0x6000028359c0>>
(lldb) po (SEL)$rsi
```

If we print out the value of rsi, the second argument, we can see the selector. This is because the second argument to objc_msgSend() is the selector for the message being sent.

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
Breakpoint 3: where = UIKitCore`-[UIResponder touchesBegan:withEvent:], address = 0x000000010e7bc745
(lldb) po $rdi
<UITableViewCellContentView: 0x7fbf76f24160; frame = (0 0; 414 55.5); opaque = NO; gestureRecognizers
= <NSArray: 0x600002607180>; layer = <CALayer: 0x6000028359c0>>
(lldb) po (SEL)$rsi
"touchesBegan:withEvent:"
(lldb)
```

If we print out the value of rsi, the second argument, we can see the selector. This is because the second argument to objc_msgSend() is the selector for the message being sent.

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
Breakpoint 3: where = UIKitCore`-[UIResponder touchesBegan:withEvent:], address = 0x000000010e7bc745
(lldb) po $rdi
<UITableViewCellContentView: 0x7fbf76f24160; frame = (0 0; 414 55.5); opaque = NO; gestureRecognizers
= <NSArray: 0x600002607180>; layer = <CALayer: 0x6000028359c0>>
(lldb) po (SEL)$rsi
"touchesBegan:withEvent:"
(lldb) po (id)$rdx
```

Finally, we might want to inspect the touch object(s) passed into the method. Even though the touches are the *first* argument of the -touchesBegan:withEvent: method, they’re the *third* argument to objc_msgSend, so we need to look at the register for argument 3. Remembering back to our calling convention, that’s rdx.

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
Breakpoint 3: where = UIKitCore`-[UIResponder touchesBegan:withEvent:], address = 0x000000010e7bc745
(lldb) po $rdi
<UITableViewCellContentView: 0x7fbf76f24160; frame = (0 0; 414 55.5); opaque = NO; gestureRecognizers
= <NSArray: 0x600002607180>; layer = <CALayer: 0x6000028359c0>>
(lldb) po (SEL)$rsi
"touchesBegan:withEvent:"
(lldb) po (id)$rdx
{(
    <UITouch: 0x7fbf76f4dc50> phase: Began tap count: 1 force: 0.000 window: <UIWindow:
0x7fbf76f21ad0; frame = (0 0; 414 896); gestureRecognizers = <NSArray: 0x6000026048a0>; layer =
<UIWindowLayer: 0x600002834140>> view: <UITableViewCellContentView: 0x7fbf76f24160; frame = (0 0; 414
55.5); opaque = NO; gestureRecognizers = <NSArray: 0x600002607180>; layer = <CALayer:
0x6000028359c0>> location in window: {138, 525.5} previous location in window: {138, 525.5} location
in view: {138, 45.5} previous location in view: {138, 45.5}
)}
(lldb)
```

And we can see the touch that triggered the touchesBegan method!

Debugging with Assembly
アセンブリでデバッギング

```
(lldb) b -[UIResponder touchesBegan:withEvent:]
Breakpoint 3: where = UIKitCore`-[UIResponder touchesBegan:withEvent:], address = 0x000000010e7bc745
(lldb) po $rdi
<UITableViewCellContentView: 0x7fbf76f24160; frame = (0 0; 414 55.5); opaque = NO; gestureRecognizers
= <NSArray: 0x600002607180>; layer = <CALayer: 0x6000028359c0>>
(lldb) po (SEL)$rsi
"touchesBegan:withEvent:"
(lldb) po (id)$rdx
{(
    <UITouch: 0x7fbf76f4dc50> phase: Began tap count: 1 force: 0.000 window: <UIWindow:
0x7fbf76f21ad0; frame = (0 0; 414 896); gestureRecognizers = <NSArray: 0x6000026048a0>; layer =
<UIWindowLayer: 0x600002834140>> view: <UITableViewCellContentView: 0x7fbf76f24160; frame = (0 0; 414
55.5); opaque = NO; gestureRecognizers = <NSArray: 0x600002607180>; layer = <CALayer:
0x6000028359c0>> location in window: {138, 525.5} previous location in window: {138, 525.5} location
in view: {138, 45.5} previous location in view: {138, 45.5}
)}
(lldb)
```

Debugging with Assembly
アセンブリでデバッギング

Swift Hint / ヒント:

```
(lldb) command alias -H "Print value in
ObjC context as object" -h "Print ObjC
object" -- cpo expression -O -l objc --
```

When lldb is in Swift mode, you can’t directly print registers like you can in Objective-C mode. However, you can use the expression command with the -l objc option to print in the context of ObjC. I like to create a command alias called cpo so I can just do cpo $rsi and have it work.

Debugging with Assembly
アセンブリでデバッギング

Swift Hint / ヒント:

```
(lldb) command alias -H "Print value in
ObjC context as object" -h "Print ObjC
object" -- cpo expression -O -l objc --
```

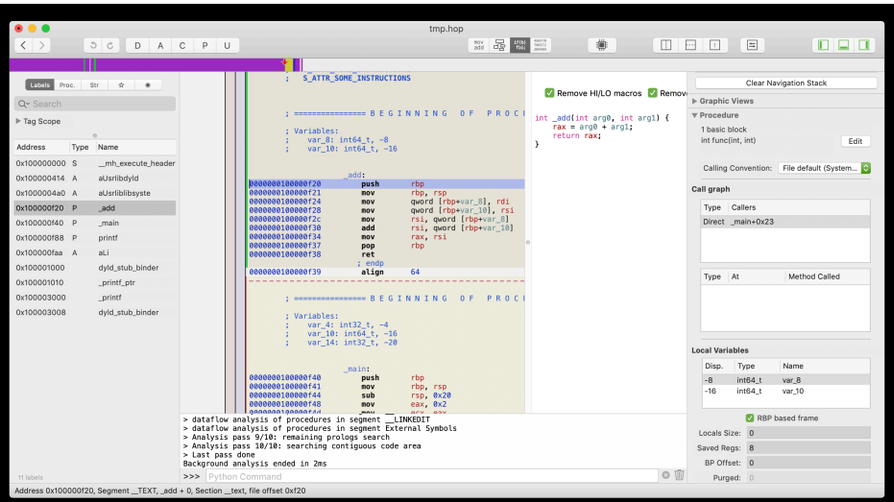
Put this in your ~/.lldbinit
これを~/.lldbinitに入れて

Debugging with Assembly
アセンブリでデバッギング

Swift Hint / ヒント:

```
(lldb) cpo $rsi
```

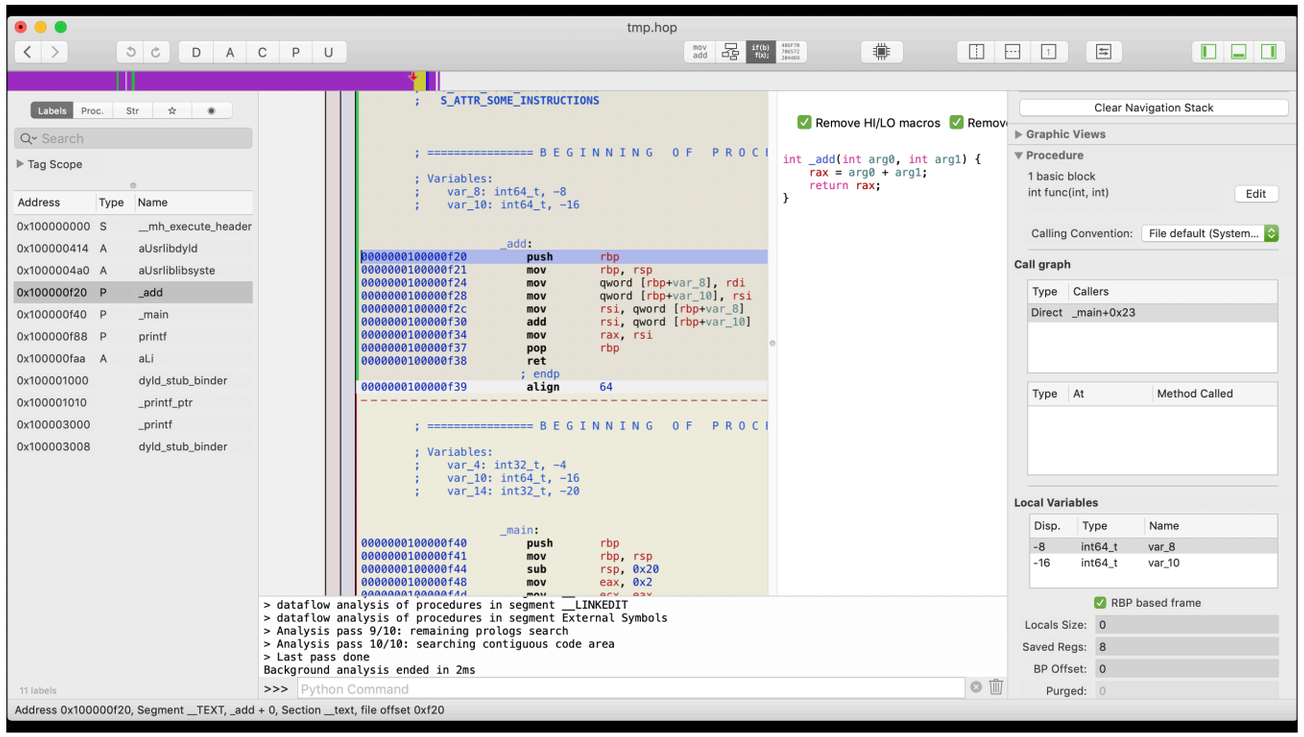
Works in Swift!
Swiftで動作する！



Hopper Disassembler

https://www.hopperapp.com

Hopper is a very useful tool for disassembling Mac and iOS binaries, including apps, libraries, and system frameworks. It knows about Swift and Objective-C, and can even create C-like pseudo code to help you understand what the assembly is doing. If you’re doing reverse engineering or disassembly frequently, it’s well worth buying.


tmp.hop
D A C P U
Labels Proc. Str
Search
Tag Scope
Address Type Name
0x100000000 S __mh_execute_header
0x100000414 A aUsrlibdyld
0x1000004a0 A aUsrliblibsyste
0x100000f20 P _add
0x100000f40 P _main
0x100000f88 P printf
0x100000faa A aLi
0x100001000 dyld_stub_binder
0x100001010 _printf_ptr
0x100003000 _printf
0x100003008 dyld_stub_binder
11 labels
; S_ATTR_SOME_INSTRUCTIONS
; ========== B E G I N N I N G O F P R O C
; Variables:
; var_8: int64_t, -8
; var_10: int64_t, -16
_add:
0000000100000f20 push rbp
0000000100000f21 mov rbp, rsp
0000000100000f24 mov qword [rbp+var_8], rdi
0000000100000f28 mov qword [rbp+var_10], rsi
0000000100000f2c mov rsi, qword [rbp+var_8]
0000000100000f30 add rsi, qword [rbp+var_10]
0000000100000f34 mov rax, rsi
0000000100000f37 pop rbp
0000000100000f38 ret
; endp
0000000100000f39 align 64
; ========== B E G I N N I N G O F P R O C
; Variables:
; var_4: int32_t, -4
; var_10: int64_t, -16
; var_14: int32_t, -20
_main:
0000000100000f40 push rbp
0000000100000f41 mov rbp, rsp
0000000100000f44 sub rsp, 0x20
0000000100000f48 mov eax, 0x2
Remove HI/LO macros
Remov
int _add(int arg0, int arg1) {
rax = arg0 + arg1;
return rax;
}
> dataflow analysis of procedures in segment __LINKEDIT
> dataflow analysis of procedures in segment External Symbols
> Analysis pass 9/10: remaining prologs search
> Analysis pass 10/10: searching contiguous code area
> Last pass done
Background analysis ended in 2ms
>>> Python Command
Address 0x100000f20, Segment __TEXT, _add + 0, Section __text, file offset 0xf20
Clear Navigation Stack
Graphic Views
Procedure
1 basic block
int func(int, int)
Edit
Calling Convention: File default (System...
Call graph
Type Callers
Direct _main+0x23
Type At Method Called
Local Variables
Disp. Type Name
-8 int64_t var_8
-16 int64_t var_10
RBP based frame
Locals Size: 0
Saved Regs: 8
BP Offset: 0
Purged: 0

More Info 他の情報

- https://www.raywenderlich.com/615-assembly-register-calling-convention-tutorial
- http://cs.lmu.edu/~ray/notes/nasmtutorial/
- https://mikeash.com/pyblog/friday-qa-2011-12-16-disassembling-the-assembly-part-1.html
- https://www.hopperapp.com

Here are some links with good information if you'd like to learn more.

Personal Info 自分の情報

 : @armadsen

 : andrew@openreel.co

 : http://blog.andrewmadsen.com

If you want to contact me, you can find me on Twitter at @armadsen, email me, or visit my blog.