Intro to Machine Code

Objectives

- Discuss how a program is compiled from programming language source code to assembly code.

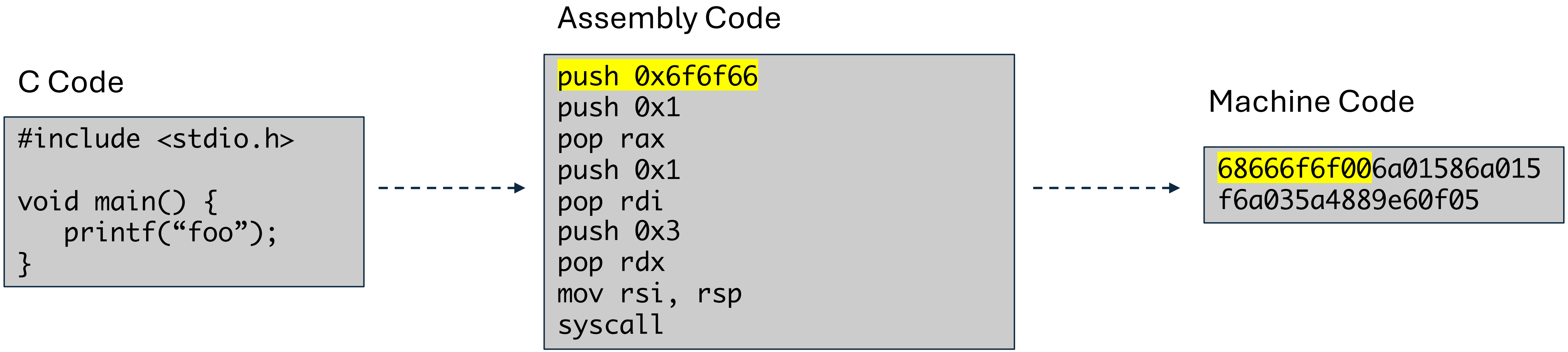
- Explore how memory registers are used to store information; discuss the usage of registers in the fastcall calling convention.

- Discuss dynamically debugging assembly code using the gdb debugger.

References

- https://dogbolt.org/
- https://godbolt.org/
- https://cloud.binary.ninja
- https://www.youtube.com/@LiveOverflow
- https://nsa-codebreaker.org/resources
- https://web.eecs.umich.edu/~sugih/pointers/gdbQS.html
- https://github.com/pwndbg/pwndbg

The life of a small program



C Code

```
#include <stdio.h>

void main() {
   printf(“foo”);
}
```

Assembly Code

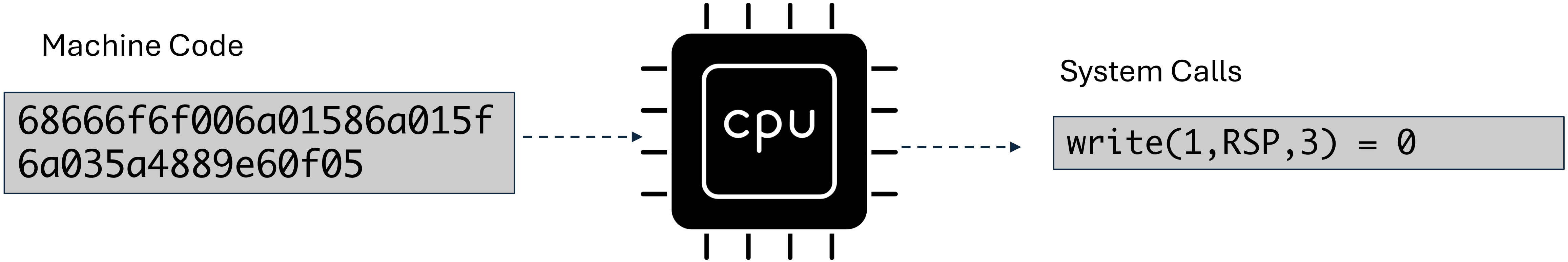
```
push 0x6f6f66
push 0x1
pop rax
push 0x1
pop rdi
push 0x3
pop rdx
mov rsi, rsp
syscall
```

Machine Code

```
68666f6f006a01586a015
f6a035a4889e60f05
```

A compiler takes a programmer’s code and converts it into assembly and ultimately machine code

The life of a small program



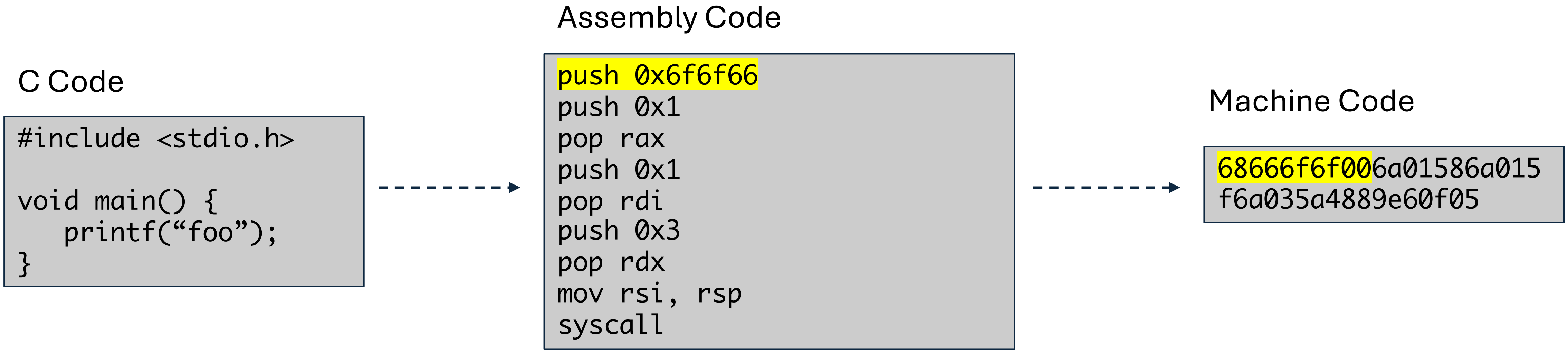
Machine Code

```
68666f6f006a01586a015f
6a035a4889e60f05
```

System Calls

```
write(1,RSP,3) = 0
```

Disassembling a small program



C Code

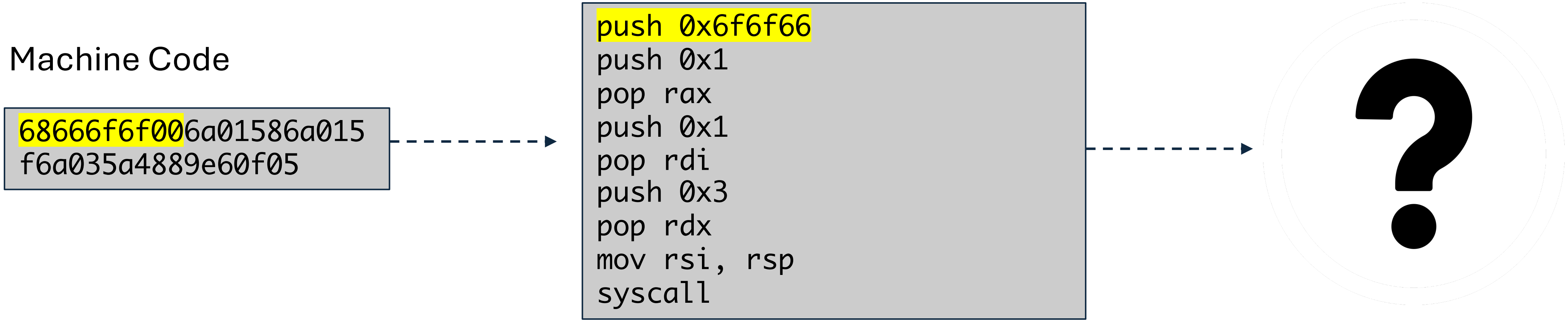
```
#include <stdio.h>

void main() {
   printf(“foo”);
}
```

Assembly Code

```
push 0x6f6f66
push 0x1
pop rax
push 0x1
pop rdi
push 0x3
pop rdx
mov rsi, rsp
syscall
```

Machine Code

```
68666f6f006a01586a015
f6a035a4889e60f05
```



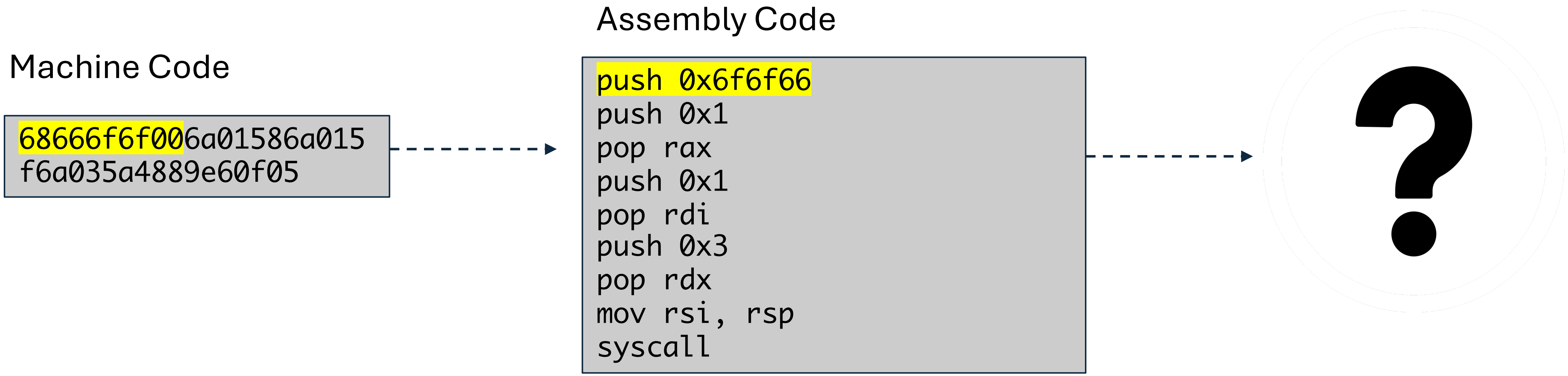
Machine Code

```
68666f6f006a01586a015
f6a035a4889e60f05
```

```
push 0x6f6f66
push 0x1
pop rax
push 0x1
pop rdi
push 0x3
pop rdx
mov rsi, rsp
syscall
```

Reverse Engineering

- Our goal as reverse engineers is to get the best representation of the original source code from either machine or assembly code
- Using debugging, tracing memory, and symbolically executing the program we can answer questions we have about how it works.



Machine Code

```
68666f6f006a01586a015
f6a035a4889e60f05
```

Assembly Code

```
push 0x6f6f66
push 0x1
pop rax
push 0x1
pop rdi
push 0x3
pop rdx
mov rsi, rsp
syscall
```

Memory Registers

- Processor Memory
  - Act as variables used by the processor
  - Are addressed directly by name in assembly code
  - Very efficient
  - Good alternative to RAM

- Many flavors
  - Data registers
  - Address registers
  - Conditional registers
  - General purpose registers
  - Special purpose registers

Text copied from https://nsa-codebreaker.org/resources (into to x86 assembly)

General Purpose Registers

- Sixteen general purpose 64-bit registers
- First eight, historically labeled
  - RAX,RBX,RCX,RDX,RBP,RSI,RDI,RSP
- Second eight, named R8-R15
- Replacing R w/E accesses the lower 32 bits

RAX	EAX
RBX	EBX
RCX	ECX
RDX	EDX
RBP	EBP
RSI	ESI
RDI	EDI
RSP	ESP
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

Calling Conventions

fastcall calling convention – arguments are passed as registers

Register	Purpose
RAX	Stores return value from function
RBX	Optionally used as base pointer
RCX	4th Argument to a function
RDX	3rd Argument to a function
RSP	Stack pointer
RBP	Frame pointer
RSI	2nd Argument to a function
RDI	1st Argument to a function
R8	5th Argument to a function
R9	6th argument to a function

Debugging With GDB

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char password[50];

  printf("Enter the password: ");
  scanf("%49s", password);
  if (strcmp(password, "supersecret") == 0) {
    system("cat flag.txt");
  } else {
    printf("Incorrect password!\n");
  }
  return 0;
}
```

```
*RAX  0x7fffffffe290 ◂— 'AAAAAAAAAAAAAAAAAA'
*RBX  0x7fffffffe3e8 —▸ 0x7fffffffe667 ◂— '/root/workspace/test'
*RCX  0x0
*RDX  0x40201e ◂— 'supersecret'
*RDI  0x7fffffffe290 ◂— 'AAAAAAAAAAAAAAAAAA'
*RSI  0x40201e ◂— 'supersecret'
*R8   0x12
*R9   0x7ffff7f94aa0 (_IO_2_1_stdin_) ◂— 0xfbad2288
*R10  0x0


 ► 0x4011ae <main+72>    call   strcmp@plt                <strcmp@plt>
     s1: 0x7fffffffe290 ◂— 'AAAAAAAAAAAAAAAAAA'
     s2: 0x40201e ◂— 'supersecret'
```

In the next block, we’ll introduce debugging a program. Understanding the calling convention can be helpful to debugging a program. Here we have a small program that asks for a password and then compares that password guess against the string “supersecret”. Notice that if we pass at the instruction for the comparison, RDI = our guess and RSI = the password.

Setting Breakpoints

- To set a breakpoint: break *<address>
- To run a program: run
- To continue debugging: continue

- To list current breakpoints: info break
- To delete a breakpoint: del [breakpointnumber]
- To temporarily disable a breakpoint: dis [breakpointnumber]
- To enable a breakpoint: en [breakpointnumber]
- To ignore a breakpoint until it has been crossed x times: ignore [breakpointnumber] [x]

Text partially copied from: https://web.eecs.umich.edu/~sugih/pointers/gdbQS.html

Continue or Stepping or Next

- step will execute a line of code
- next will execute the entire function
- continue will resume normal execution

```
pwndbg> next

 0x401385 <main+23>    call   printf@plt                          <printf@plt>
 ► 0x40138a <main+28>    lea    rax, [rbp - 0x20]
   0x40138e <main+32>    mov    rsi, rax
   0x401391 <main+35>    lea    rax, [rip + 0xcf5]
```

Execute printf()

```
pwndbg> step

 ► 0x7ffff7e16b30 <printf>        sub    rsp, 0xd8
   0x7ffff7e16b37 <printf+7>      mov    qword ptr [rsp + 0x28], rsi
   0x7ffff7e16b3c <printf+12>     mov    qword ptr [rsp + 0x30], rdx
   0x7ffff7e16b41 <printf+17>     mov    qword ptr [rsp + 0x38], rcx
```

Step into printf()

PwnDbg

- We’ve installed a GDB plug-in that provides enhanced commands and visualization.

- Type: context to see the current context of the program, showing the current instructions
- Type: regs to see the state of the general purpose register
- Type: stack to see the state of the program stack

Text partially copied from: https://web.eecs.umich.edu/~sugih/pointers/gdbQS.html