CS 24

Introduction to Computing Systems

x86-64 Basics

```
mov %edi, %eax
xor %ecx, %ecx
test %edi, %edi
setne %cl
mov $0xffffffff, %eax
cmovns %ecx, %eax
retq
```

Assembly? Why Bother?

Assembly? Why Bother?

- In 2020, almost all assembly is written by **compilers** or in the **operating system**, but who writes those?

Assembly? Why Bother?

- In 2020, almost all assembly is written by **compilers** or in the **operating system**, but who writes those?
- In 2020, the high-level language model occasionally breaks down and you have to read the assembly to understand the machine's behavior!

Assembly? Why Bother?

- In 2020, almost all assembly is written by **compilers** or in the **operating system**, but who writes those?
- In 2020, the high-level language model occasionally breaks down and you have to read the assembly to understand the machine's behavior!
- In 2020, it's important to understand the types of optimizations a compiler is capable of making–and those it isn't!

Assembly? Why Bother?

- In 2020, almost all assembly is written by **compilers** or in the **operating system**, but who writes those?
- In 2020, the high-level language model occasionally breaks down and you have to read the assembly to understand the machine's behavior!
- In 2020, it's important to understand the types of optimizations a compiler is capable of making–and those it isn't!
- In 2020, software is generally distributed in binary form; if you want to **reverse engineer** or **security audit** software, it's going to be assembly!

```
blank@compute-cpu2:~$ cat identity.c
int identity(int x) {
    return x;
}

blank@compute-cpu2:~$ clang -S identity.c
blank@compute-cpu2:~$ cat identity.s
identity:
    movl %edi, %eax
    retq
```

```
blank@compute-cpu2:~$ cat identity.c
int identity(int x) {
    return x;
}

blank@compute-cpu2:~$ clang -c identity.c
blank@compute-cpu2:~$ objdump -d identity.o

simple.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <identity>:
    0: 89 f8                    mov    %edi,%eax
    2: c3                       retq
```

What kinds of “data“ are there?

- **Immediates:** Constant integer data
  - Examples: `$0x400`, `$-533`
  - Like C literal, but prefixed with ‘$’
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction
- **Registers:** behave like “global variables”, but hardwired in the processor
  - Examples: `%eax`, `%edi`
  - Some of them are reserved for special uses or have special meanings
- **Memory:** Consecutive bytes of memory

What are these `%eax` thingies?

Registers are locations in the CPU that store a small amount of data, which can be accessed very quickly (once every clock cycle). They have names (e.g., `%rsi`)–not addresses. They are a precious commodity in all architectures, but especially x86-64.

(return)	**`%eax`**
	`%ebx`
(arg 4)	**`%ecx`**
(arg 3)	**`%edx`**
(arg 1)	**`%edi`**
(arg 2)	**`%esi`**
	`%esp`
	`%ebp`

There are three major **classes** of things assembly instructions do:

1. Transfer data between memory and registers
   - Load data from memory into register: `%reg = mem[addr]`
   - Store register data into memory: `mem[addr] = %reg`
2. Perform arithmetic operation on register or memory data
   - e.g., `%eax += %ebx`
   - e.g., `%eax += mem[addr]`
3. Re-direct control flow (jumps and gotos)

Understanding Identity (mov)

C Code

```
1 int identity(int x) {
2    return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8             mov   %edi,%eax
  400482: c3                retq
```

Understanding Identity (mov)

C Code

```
1 int identity(int x) {
2     return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8                mov   %edi,%eax
  400482: c3                   retq
```

x86-64 Instruction: mov

x86-64		C Pseudocode
`mov %src, %dst`	↔	`%dst = %src`

Understanding Identity (`mov`)

C Code

```
1 int identity(int x) {
2     return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8                mov   %edi,%eax
  400482: c3                   retq
```

x86-64 Instruction: `mov`

x86-64		**C Pseudocode**
`mov %src, %dst`	↔	`%dst = %src`

Pseudocode Translation (so far)

```
1 identity:
2     %eax = %edi
3     retq
```

Understanding Identity (`retq` and return value)

C Code

```
1 int identity(int x) {
2     return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8                mov   %edi,%eax
  400482: c3                   retq
```

x86-64 Instruction: `mov`

x86-64		C Pseudocode
`mov %src, %dst`	↔	`%dst = %src`

x86-64 Instruction: `retq`

x86-64		C Pseudocode
`retq`	↔	`return %eax`

Understanding Identity (`retq` and return value)

C Code

```
1 int identity(int x) {
2     return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8                mov   %edi,%eax
  400482: c3                   retq
```

x86-64 Instruction: `mov`

x86-64		C Pseudocode
`mov %src, %dst`	↔	`%dst = %src`

x86-64 Instruction: `retq`

x86-64		C Pseudocode
`retq`	↔	`return %eax`

Pseudocode Translation (so far)

```
1 identity:
2     %eax = %edi
3     return %eax
```

Understanding Identity (arguments)

C Code

```
1 int identity(int x) {
2     return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8                mov   %edi,%eax
  400482: c3                   retq
```

x86-64 Instruction: `mov`

x86-64		C Pseudocode
`mov %src, %dst`	↔	`%dst = %src`

x86-64 Instruction: `retq`

x86-64		C Pseudocode
`retq`	↔	`return %eax`

Understanding Identity (arguments)

C Code

```
1 int identity(int x) {
2     return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8                mov   %edi,%eax
  400482: c3                   retq
```

x86-64 Instruction: `mov`

x86-64		C Pseudocode
`mov %src, %dst`	↔	`%dst = %src`

x86-64 Instruction: `retq`

x86-64		C Pseudocode
`retq`	↔	`return %eax`

Pseudocode Translation (so far)

```
1 identity(%edi):
2     %eax = %edi
3     return %eax
```

Understanding Identity (simplifying)

C Code

```
1 int identity(int x) {
2     return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8                mov   %edi,%eax
  400482: c3                   retq
```

x86-64 Instruction: `mov`

x86-64		C Pseudocode
`mov %src, %dst`	↔	`%dst = %src`

x86-64 Instruction: `retq`

x86-64		C Pseudocode
`retq`	↔	`return %eax`

Understanding Identity (simplifying)

C Code

```
1 int identity(int x) {
2     return x;
3 }
```

x86-64 Disassembly

```
0000000000400480 <identity>:
  400480: 89 f8                mov   %edi,%eax
  400482: c3                   retq
```

x86-64 Instruction: `mov`

x86-64		C Pseudocode
`mov %src, %dst`	↔	`%dst = %src`

x86-64 Instruction: `retq`

x86-64		C Pseudocode
`retq`	↔	`return %eax`

Pseudocode Translation (so far)

```
1 identity(%edi):
2     return %edi
```

System V AMD64 ABI

- The value in `%eax` is automatically returned by **`retq`**.
- The first argument to a function is stored in `%edi`.

Things to Notice About x86-64

- There are no types!!!
- The **conventions** define what the compiler is allowed to do.

Understanding Arithmetic (arithmetic instructions)


```
0000000000000000 <arith>:
   0:  83 c7 01          add  $0x1,%edi
   3:  0f af ff          imul %edi,%edi
   6:  89 f8             mov  %edi,%eax
   8:  c3                retq
```

```
0000000000000000 <arith>:
   0:  83 c7 01          add  $0x1,%edi
   3:  0f af ff          imul %edi,%edi
   6:  89 f8             mov  %edi,%eax
   8:  c3                retq
```

Pseudocode Translation (so far)

```
1 arith(%edi):
2    add $0x1,%edi
3    imul %edi,%edi
4    %eax = %edi
5    return %eax
```

```
0000000000000000 <arith>:
   0:  83 c7 01          add  $0x1,%edi
   3:  0f af ff          imul %edi,%edi
   6:  89 f8             mov  %edi,%eax
   8:  c3                retq
```

Pseudocode Translation (so far)

```
1 arith(%edi):
2    add $0x1,%edi
3    imul %edi,%edi
4    %eax = %edi
5    return %eax
```

x86-64 Instruction: (Some) Arithmetic Operations

x86-64		C Pseudocode
`add %src, %dst`	↔	`%dst += %src`
`sub %src, %dst`	↔	`%dst -= %src`
`imul %src, %dst`	↔	`%dst *= %src`

```
0000000000000000 <arith>:
   0:  83 c7 01          add  $0x1,%edi
   3:  0f af ff          imul %edi,%edi
   6:  89 f8             mov  %edi,%eax
   8:  c3                retq
```

x86-64 Instruction: (Some) Arithmetic Operations

x86-64		C Pseudocode
`add %src, %dst`	↔	`%dst += %src`
`sub %src, %dst`	↔	`%dst -= %src`
`imul %src, %dst`	↔	`%dst *= %src`

Pseudocode Translation (so far)

```
1 arith(%edi):
2    %edi += $0x1
3    %edi *= %edi
4    %eax = %edi
5    return %eax
```

```
0000000000000000 <arith>:
   0:  83 c7 01          add   $0x1,%edi
   3:  0f af ff          imul  %edi,%edi
   6:  89 f8             mov   %edi,%eax
   8:  c3                retq
```

x86-64 Instruction: (Some) Arithmetic Operations

x86-64		C Pseudocode
`add %src, %dst`	↔	`%dst += %src`
`sub %src, %dst`	↔	`%dst -= %src`
`imul %src, %dst`	↔	`%dst *= %src`

Pseudocode Translation (so far)

```
1 arith(%edi):
2    return (%edi + $0x1) * (%edi + $0x1)
```