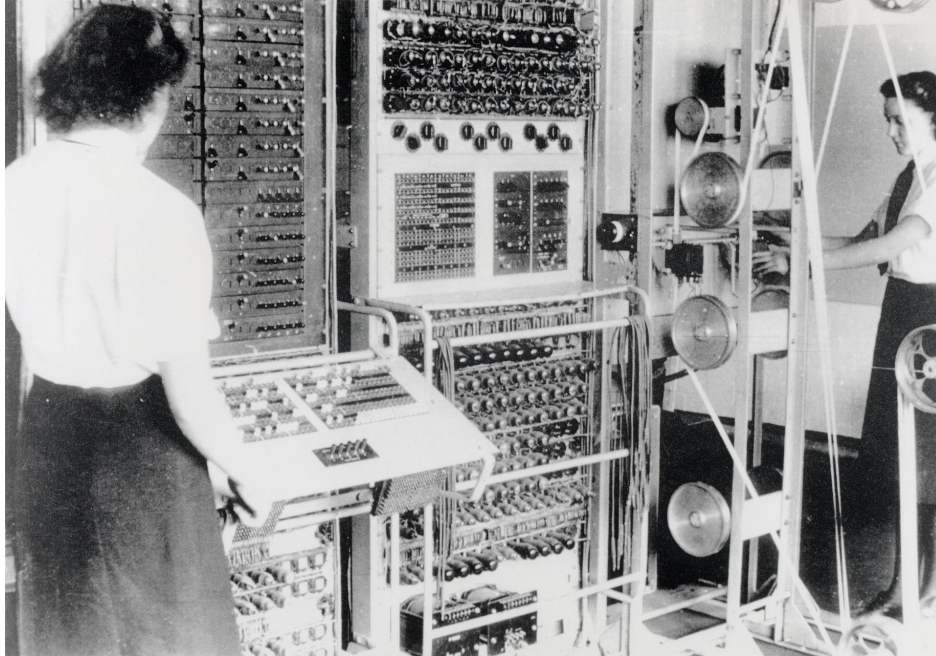


Introduction to x86-64 Assembly Programming

Aaron Bauer

First Digital Computers Programmed Manually



```
1000011011111000010010000011
10000000000
0111010000011000
10001011010001000010010000010
100
10001011010001100010010100010
100
1000110100000100000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011
```

Machine Code

Assembly Languages Provided Text Representation

```
    cmpl    $0, -4(%ebp)
    je      .L2
    movl    -12(%ebp), %eax
    movl    -8(%ebp), %edx
    leal    (%edx,%eax), %eax
    movl    %eax, %edx
    sarl    $31, %edx
    idivl  -4(%ebp)
    movl    %eax, -8(%ebp)
.L2:
```

Assembly

Assembler

```
10000011011111000010010000011
100000000000
0111010000011000
10001011010001000010010000010
100
01011010001100010010100010
100
1000110100000100000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011
```

Machine Code

High-Level Languages Are Turned Into Machine Code

```
if (x != 0) y = (y+z)/x;
```

High-level language (e.g., Java, C)

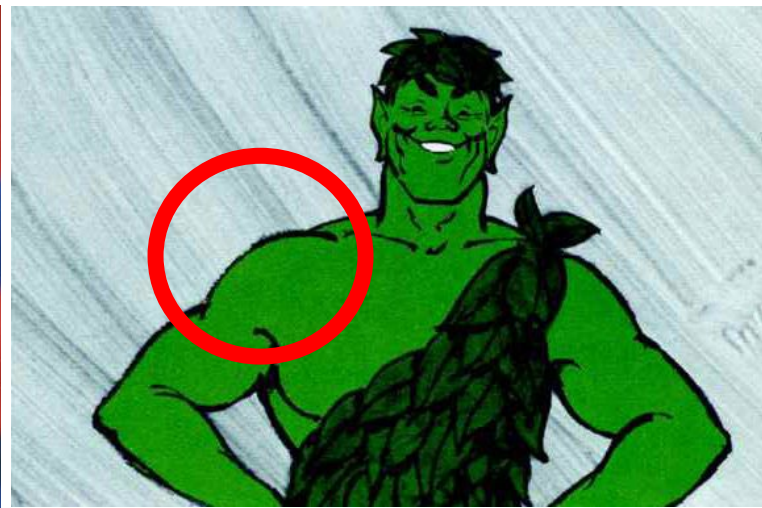
Compiler

```
    cmpl    $0, (%ebp)
    je      .L2
    movl    -12(%ebp), %eax
    movl    -8(%ebp), %edx
    leal    (%edx,%eax), %eax
    movl    %eax, %edx
    sarl    $31, %edx
    idivl   -4(%ebp)
    movl    %eax, -8(%ebp)
```

```
.L2:
```

Assembler

```
100001101111100010010000011
10000000000
0111010000011000
10001011010001000010010000010
100
01011010001100010010100010
100
1000110100000100000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011
```



Why Study Assembly Programming?

- Understand optimizations made by the compiler and how your high-level code might affect them
- High-level languages can hide details we need to know
 - Ex. investigate exactly where data is stored—can be crucial for concurrent programs
- Write more secure software
 - Many of the ways programs can be attacked involve exploiting the way programs store their run-time control information

Mainstream Instruction Set Architectures



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, M)

[x86-64 Instruction Set](#)

ARM

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility ^[1]
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)

[ARM Instruction Set](#)



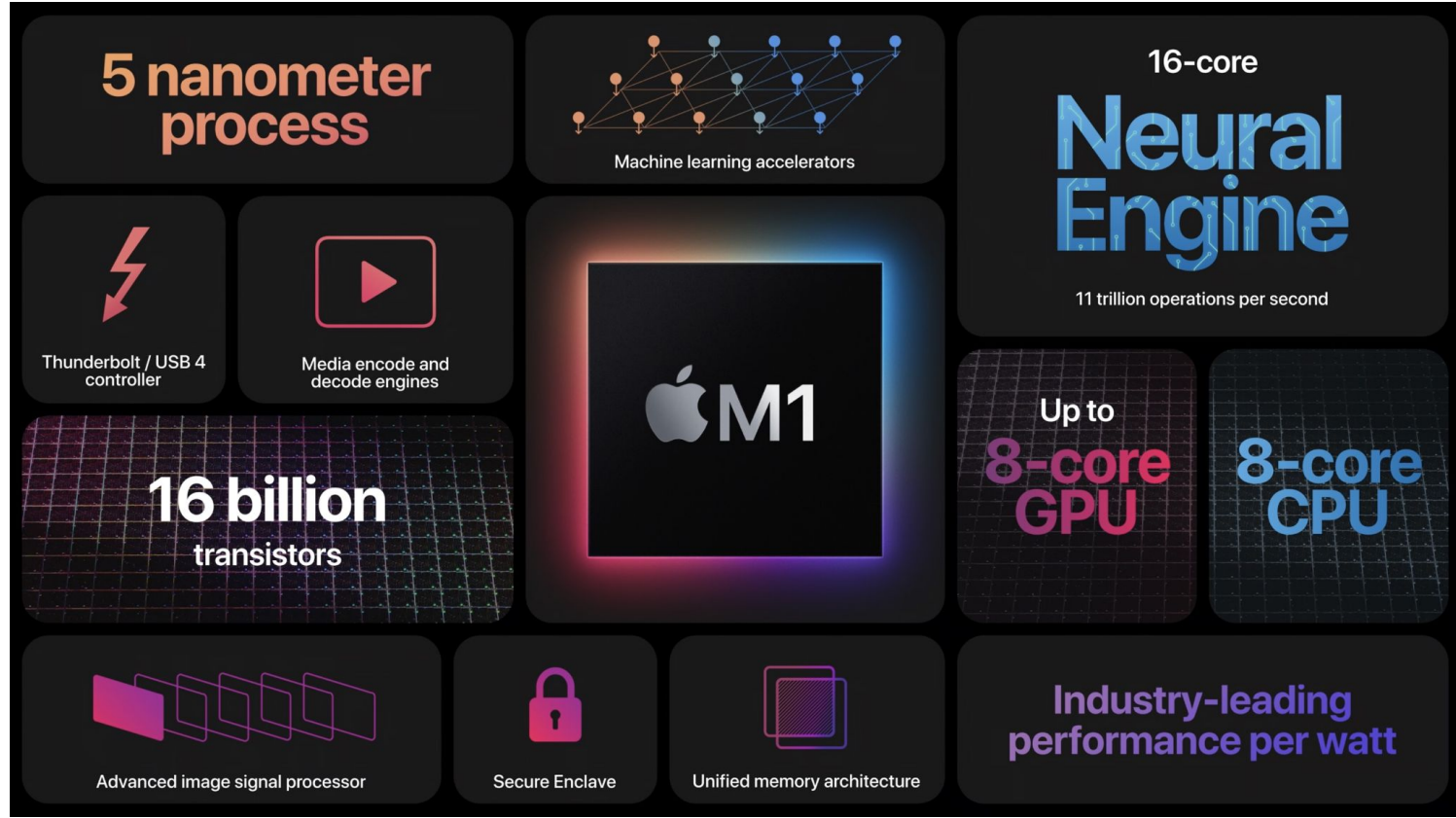
MIPS

Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981; 35 years ago
Design	RISC
Type	Register-Register
Encoding	Fixed
Endianness	Bi

Digital home & networking gear
(Blu-ray, Playstation 2)

[MIPS Instruction Set](#)

The new Apple M1 Architecture is based on ARM



The infographic features a central square with a glowing border containing the Apple logo and 'M1'. Surrounding this are various feature tiles with icons and text. The top row includes '5 nanometer process', 'Machine learning accelerators' (with a neural network diagram), and '16-core Neural Engine' (with '11 trillion operations per second'). The middle row has 'Thunderbolt / USB 4 controller' (with a lightning bolt icon), 'Media encode and decode engines' (with a play button icon), and 'Up to 8-core GPU' and '8-core CPU'. The bottom row includes '16 billion transistors' (with a grid icon), 'Advanced image signal processor' (with a camera icon), 'Secure Enclave' (with a padlock icon), 'Unified memory architecture' (with a memory stack icon), and 'Industry-leading performance per watt'.

5 nanometer process

Machine learning accelerators

16-core
Neural Engine
11 trillion operations per second

Thunderbolt / USB 4 controller

Media encode and decode engines

16 billion transistors

Up to **8-core GPU**

8-core CPU

Advanced image signal processor

Secure Enclave

Unified memory architecture

Industry-leading performance per watt

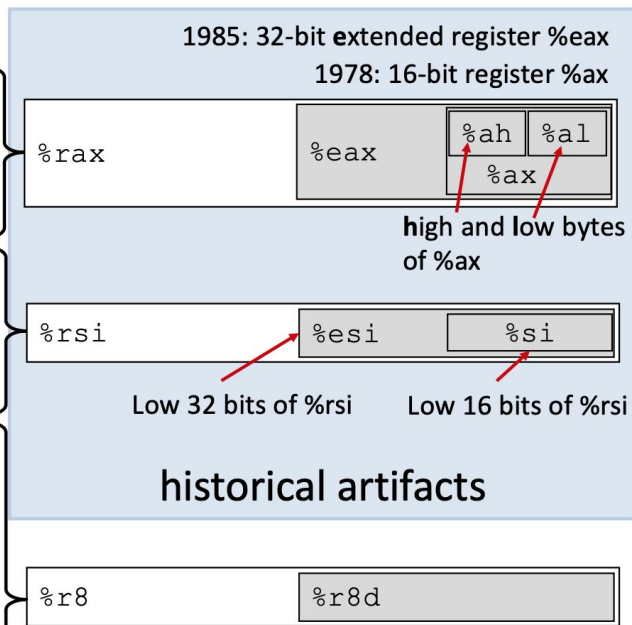
x86-64 integer registers — 64 bits wide

x86-64 registers

%rax	Return Value
%rbx	
%rcx	Argument 4
%rdx	Argument 3
%rsi	Argument 2
%rdi	Argument 1
%rsp	Special Purpose: Stack Pointer
%rbp	
%r8	Argument 5
%r9	Argument 6
%r10	
%r11	
%r12	
%r13	
%r14	
%r15	

64-bits / 8 bytes

sub-registers



Some have special uses for particular instructions

Registers

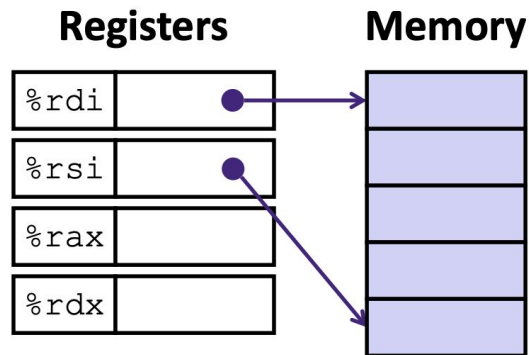
<code>%rax</code>	0x100
<code>%rcx</code>	0x1
<code>%rdx</code>	0x3

Memory Address

	0x120
0x11	0x118
0x13	0x110
0xAB	0x108
0xFF	0x100

Understanding swap ()

```
void swap(long* xp, long* yp) {  
    long t0 = *xp;  
    long t1 = *yp;  
    *xp = t1;  
    *yp = t0;  
}
```



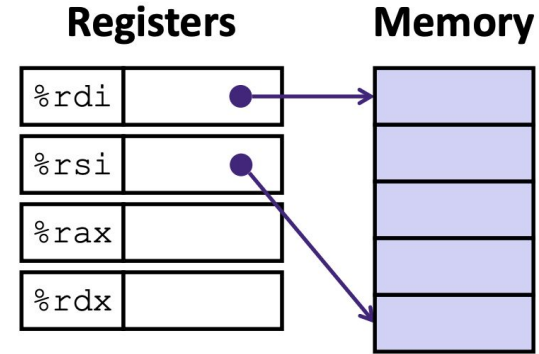
<u>Register</u>		<u>Variable</u>
%rdi	↔	xp
%rsi	↔	yp
%rax	↔	t0
%rdx	↔	t1

Understanding swap ()

```
void swap(long* xp, long* yp) {  
    long t0 = *xp;  
    long t1 = *yp;  
    *xp = t1;  
    *yp = t0;  
}
```

swap:

```
movq (%rdi), %rax  
movq (%rsi), %rdx  
movq %rdx, (%rdi)  
movq %rax, (%rsi)  
ret
```



<u>Register</u>		<u>Variable</u>
%rdi	↔	xp
%rsi	↔	yp
%rax	↔	t0
%rdx	↔	t1

Understanding `swap()`

Registers

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	
<code>%rdx</code>	

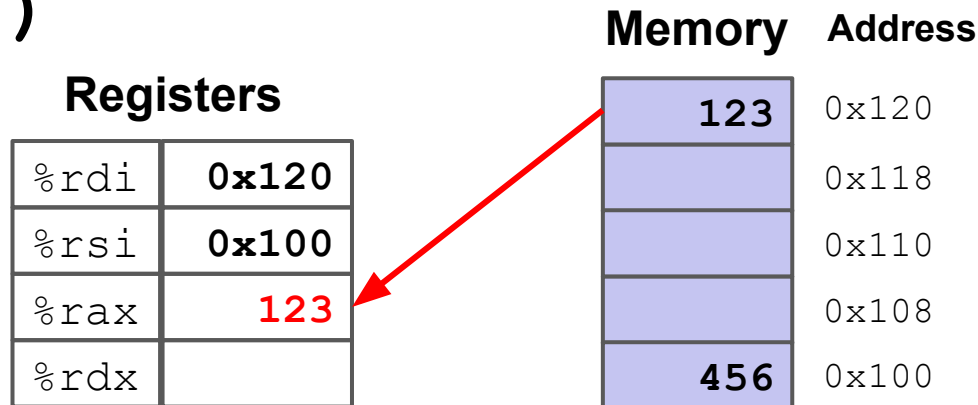
Memory Address

<code>123</code>	<code>0x120</code>
	<code>0x118</code>
	<code>0x110</code>
	<code>0x108</code>
<code>456</code>	<code>0x100</code>

```
swap:
```

```
    movq (%rdi), %rax # t0 = *xp
    movq (%rsi), %rdx # t1 = *yp
    movq %rdx, (%rdi) # *xp = t1
    movq %rax, (%rsi) # *yp = t0
    ret
```

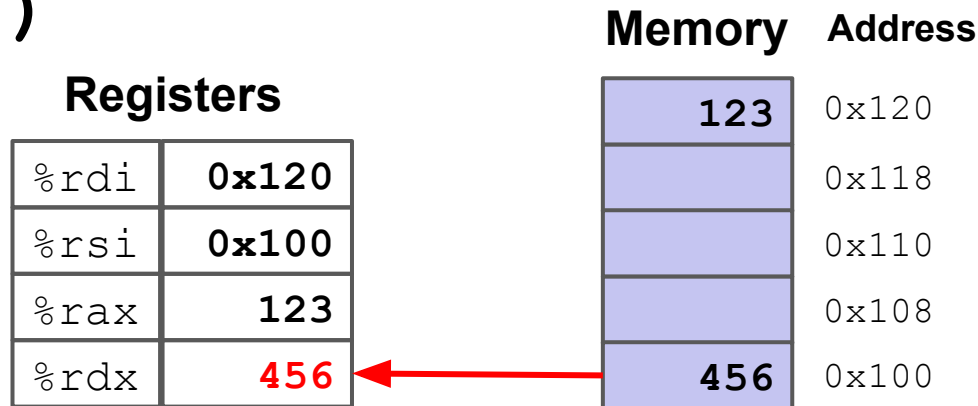
Understanding `swap()`



```
swap:
```

```
movq (%rdi), %rax # t0 = *xp
movq (%rsi), %rdx # t1 = *yp
movq %rdx, (%rdi) # *xp = t1
movq %rax, (%rsi) # *yp = t0
ret
```

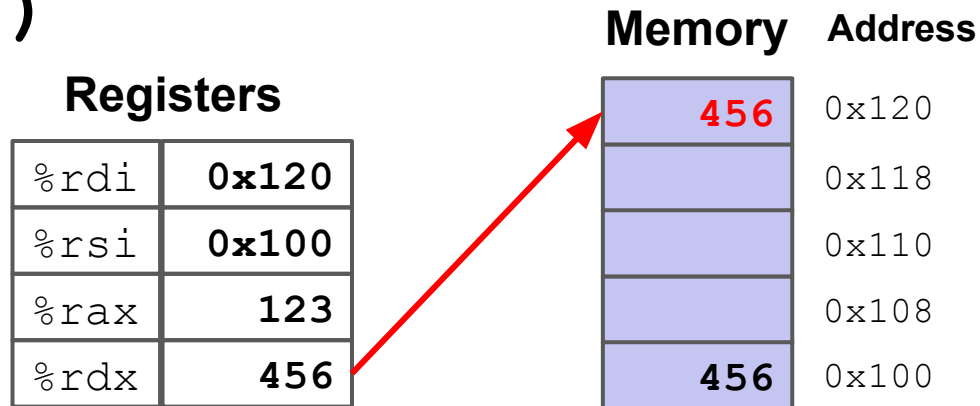
Understanding `swap()`



```
swap:
```

```
    movq (%rdi), %rax # t0 = *xp  
    movq (%rsi), %rdx # t1 = *yp  
    movq %rdx, (%rdi) # *xp = t1  
    movq %rax, (%rsi) # *yp = t0  
    ret
```

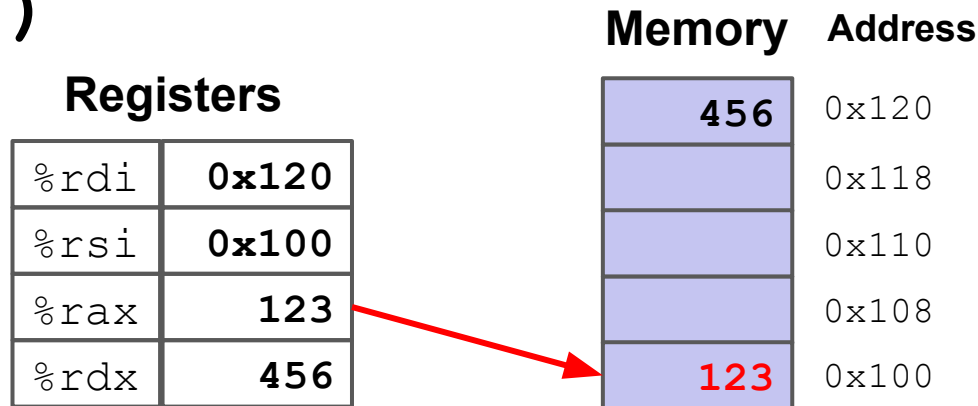
Understanding `swap()`



```
swap:
```

```
    movq (%rdi), %rax # t0 = *xp
    movq (%rsi), %rdx # t1 = *yp
    movq %rdx, (%rdi) # *xp = t1
    movq %rax, (%rsi) # *yp = t0
    ret
```


Understanding `swap()`



```
swap:
```

```
    movq (%rdi), %rax # t0 = *xp
    movq (%rsi), %rdx # t1 = *yp
    movq %rdx, (%rdi) # *xp = t1
    movq %rax, (%rsi) # *yp = t0
    ret
```

Decoding `mystery()`

```
void mystery(long* xp, long* yp, long *zp)  
// xp in %rdi, yp in %rsi, zp in %rdx
```

```
mystery:  
    movq (%rdi), %r8  
    movq (%rsi), %rcx  
    movq (%rdx), %rax  
    movq %r8, (%rsi)  
    movq %rcx, (%rdx)  
    movq %rax, (%rdi)  
    ret
```

Registers

<code>%rax</code>	0x100
<code>%rcx</code>	0x1
<code>%rdx</code>	0x3

Operand

<code>\$0x108</code>	
<code>260(%rcx, %rdx)</code>	
<code>(%rax, %rdx, 4)</code>	

Value

Memory Address

	<code>0x110</code>
0x11	<code>0x10C</code>
0x13	<code>0x108</code>
0xAB	<code>0x104</code>
0xFF	<code>0x100</code>

Registers

<code>%rax</code>	0x100
<code>%rcx</code>	0x1
<code>%rdx</code>	0x3

Memory Address

	0x110
0x11	0x10C
0x13	0x108
0xAB	0x104
0xFF	0x100

Operand Value

<code>\$0x108</code>	0x108
<code>260(%rcx, %rdx)</code>	0x13
<code>(%rax, %rdx, 4)</code>	0x11