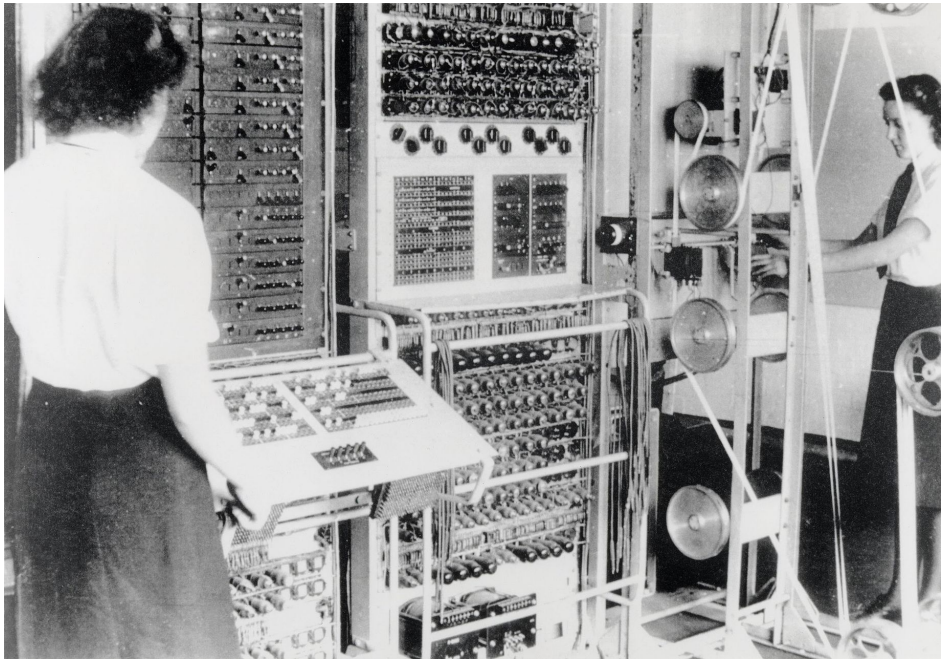
Introduction to x86-64 Assembly Programming

Aaron Bauer

First Digital Computers Programmed Manually



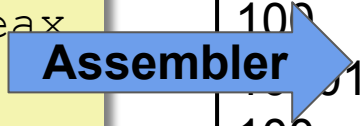
```
10000011011111000010010000011
10000000000
0111010000011000
10001011010001000010010000010
100
10001011010001100010010100010
100
10001101000001000000010
1000100111000010
11000001111110100011111
11110111011111000010010000011
```

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




```
10000011011111000010010000011
10000000000
0111010000011000
10001011010001000010010000010
100
1010110100011000100101000100
100
10001101000001000000010
1000100111000010
11000001111110100001111
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, -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:
```

Assembler

```
10000011011111000010010000011
10000000000
0111010000011000
10001011010001000010010000010
100
1010110100011000100101000100
100
10001101000001000000010
1000100111000010
11000001111110100001111
11110111011111000010010000011
```


GIANTS
ny
PRIDE
13


Giants

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 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)

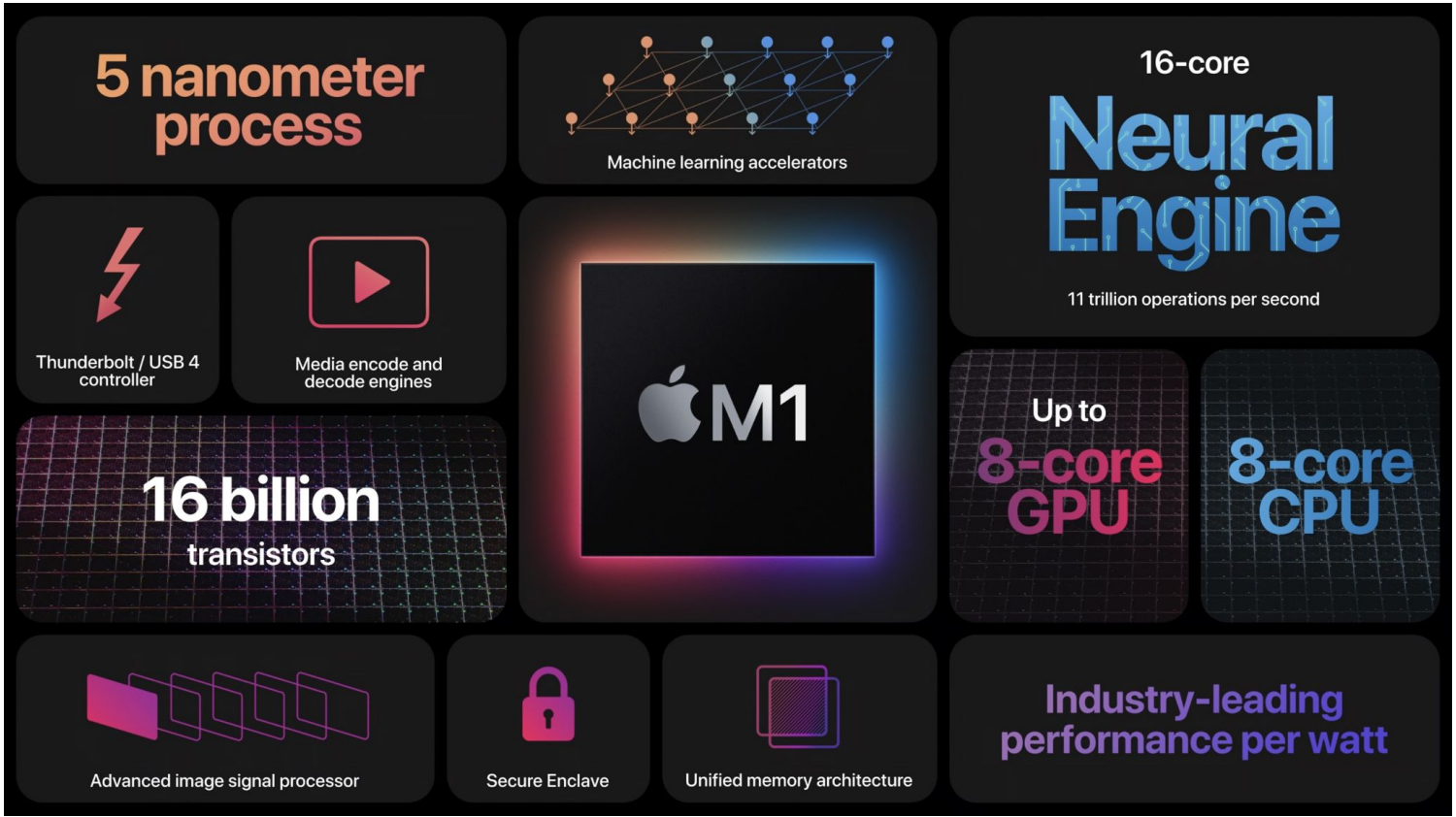
Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

MIPS

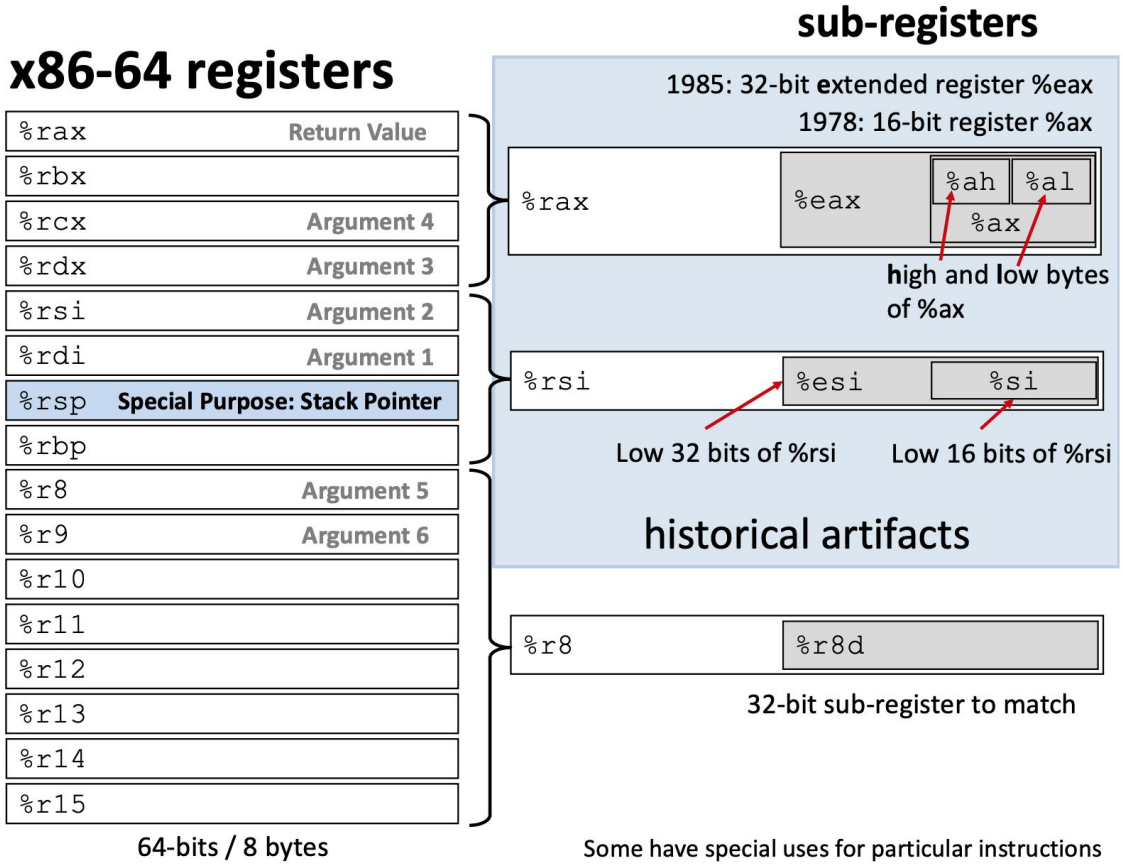
Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981; 35 years ago
Design	RISC
Type	Register-Register
Encoding	Fixed
Endianness	Bi

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

The new Apple M1 Architecture is based on ARM



x86-64 integer registers — 64 bits wide



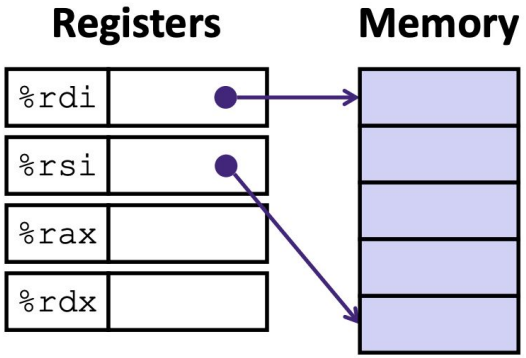
Registers

%rax	**0x100**
%rcx	**0x1**
%rdx	**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;
}
```

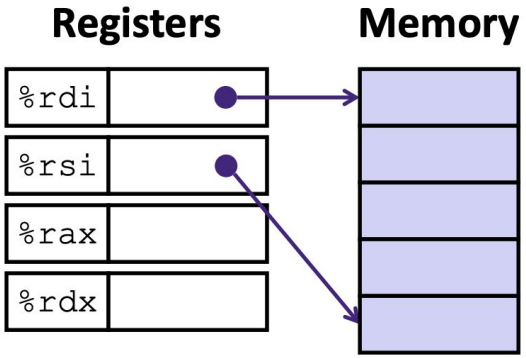


Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

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
```



Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

Understanding **swap()**

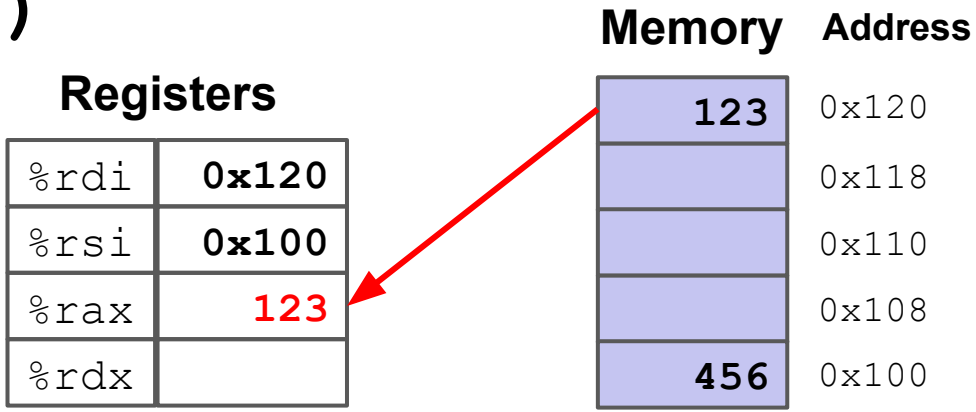
Registers

%rdi	**0x120**
%rsi	**0x100**
%rax	
%rdx	

Memory	**Address**
123	0x120
	0x118
	0x110
	0x108
456	0x100

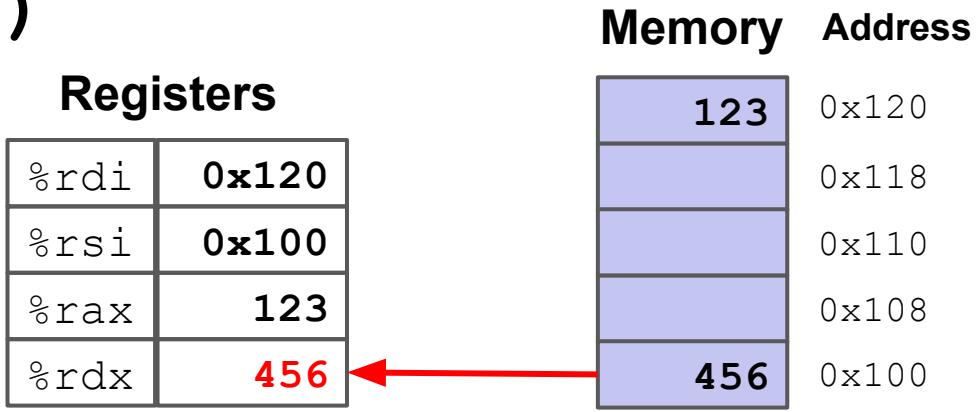
```
swap:
    movq (%rdi), %rax #  t0 = *xp
    movq (%rsi), %rdx #  t1 = *yp
    movq %rdx, (%rdi) # *xp =  t1
    movq %rax, (%rsi) # *yp =  t0
    ret
```

Understanding **swap()**



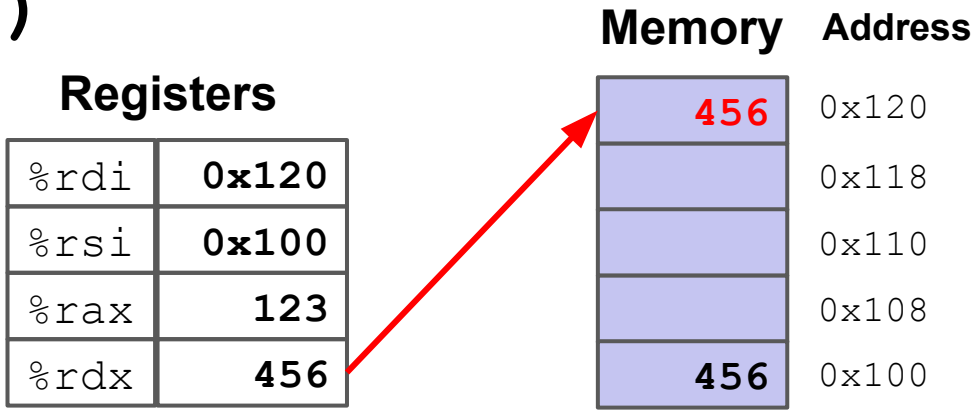

```
swap:
    movq (%rdi), %rax #  t0 = *xp
    movq (%rsi), %rdx #  t1 = *yp
    movq %rdx, (%rdi) # *xp =  t1
    movq %rax, (%rsi) # *yp =  t0
    ret
```

Understanding **swap()**



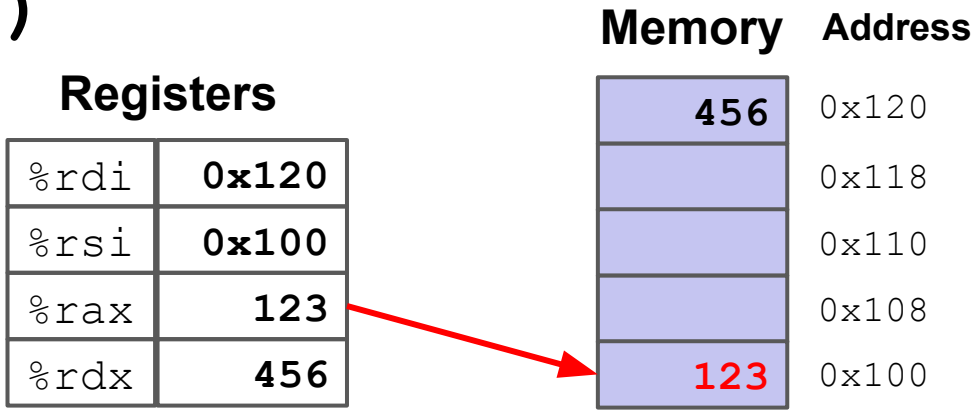

```
swap:
    movq (%rdi), %rax #  t0 = *xp
    movq (%rsi), %rdx #  t1 = *yp
    movq %rdx, (%rdi) # *xp =  t1
    movq %rax, (%rsi) # *yp =  t0
    ret
```

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

%rax	**0x100**
%rcx	**0x1**
%rdx	**0x3**

Memory

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

Operand	Value
$0x108	
260(%rcx, %rdx)	
(%rax, %rdx, 4)	

Registers

%rax	**0x100**
%rcx	**0x1**
%rdx	**0x3**

Operand	Value
$0x108	**0x108**
260(%rcx, %rdx)	**0x13**
(%rax, %rdx, 4)	**0x11**

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