Assembly Basics

SPRING, 2019

HYOUNG-KEE CHOI

This Powerpoint slides are modified from its original version available at http://www.cs.cmu.edu/afs/cs/academic/class/15213-s09/www/lectures/ppt-sources/

NO Cellphone in Class

- Power off
- Not in the desk
  - Keep in you bag
- If the phone rings
  - You will get “F” grade, No excuse

Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Intel x86 Processors

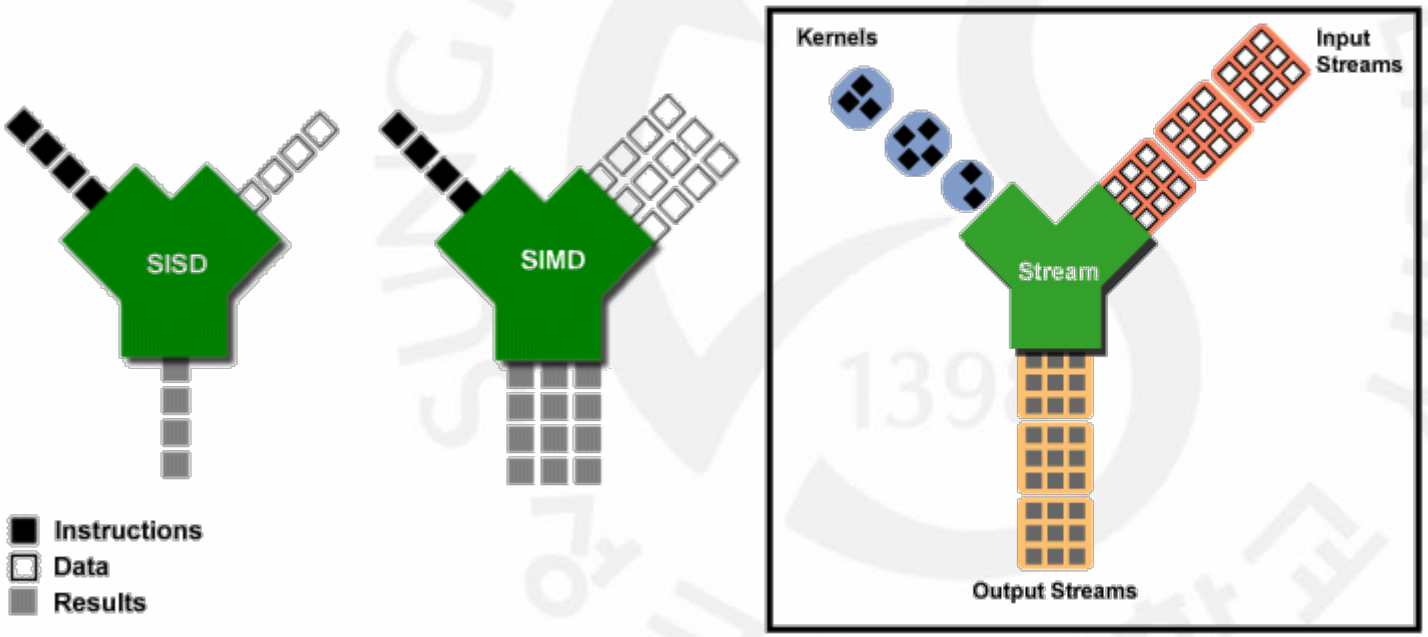
- Totally dominate computer market
- Revolutionary design
  - Backwards **compatible** up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - Only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!

Intel x86 Processors: Overview

Architectures	Processors
X86-16	8086
	286
X86-32/IA32	386
	486
	Pentium
MMX	Pentium MMX
SSR	Pentium III
SSR2	Pentium 4
SSR3	Pentium 4R
X86-64 / RM64t	Pentium 4F
SSR4	Core 2 Duo
	Core i7

time

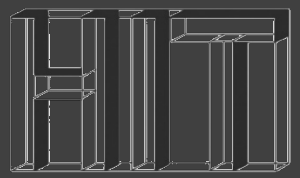
SIMD


SISD
SIMD
Instructions
Data
Results
Kernels
Input Streams
Stream
Output Streams

Intel x86 Processors

486	1989	1.9M	486 DX2/66
Pentium	1993	3.1M	P5 / 80586
Pentium Pro	1995	6.5M	P6, 150/166 MHz, 16 L1/256 L2
Pentium MMX	1997	4.5M	SIMD: MMX
Pentium II	1997	7.5M	233/266 MHz, 32 L1/512 L2, SSR, Celeron/Xeon
Pentium III	1999	9.5M	500~1133 MHz, SSR
Pentium 4	2000	42M	1.4/1.5 GHz, Hyperthreding, NetBurst, SSR2, L3, Prescott
Pentium M	2003	77M	For mobile
Core Duo	2006	151M	NetBurst, Enhanced M
Core 2 Duo	2006	291M	2 processors in one chip
Core i3, i5, i7	2008		Nehalem (2008), Sandy Bridge (2011), Haswell (2013)

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores



Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. (DEC) and other downward trending companies
  - Built **Opteron**: tough competitor to Pentium 4
  - Developed **x86-64**, their own extension to 64 bits

- Recent Years
  - Intel got its act together
    - Leads the world in semiconductor technology
  - AMD has fallen behind
    - Relies on external semiconductor manufacturer

Intel's 64-Bit History

- 2001: Intel attempts radical shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- 2003: AMD steps in with evolutionary solution
  - x86-64 (now called "AMD64")
- Intel felt obligated to focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel announces RM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

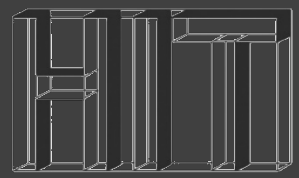
Our Coverage

- IA32
  - The traditional x86
- x86-64
  - The standard
- Presentation
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

- Why can’t you run your favorite PC applications on your fancy Phone?
- How is it that some of the applications that you were running as a kid on your big desktop computer still works on your latest laptop?

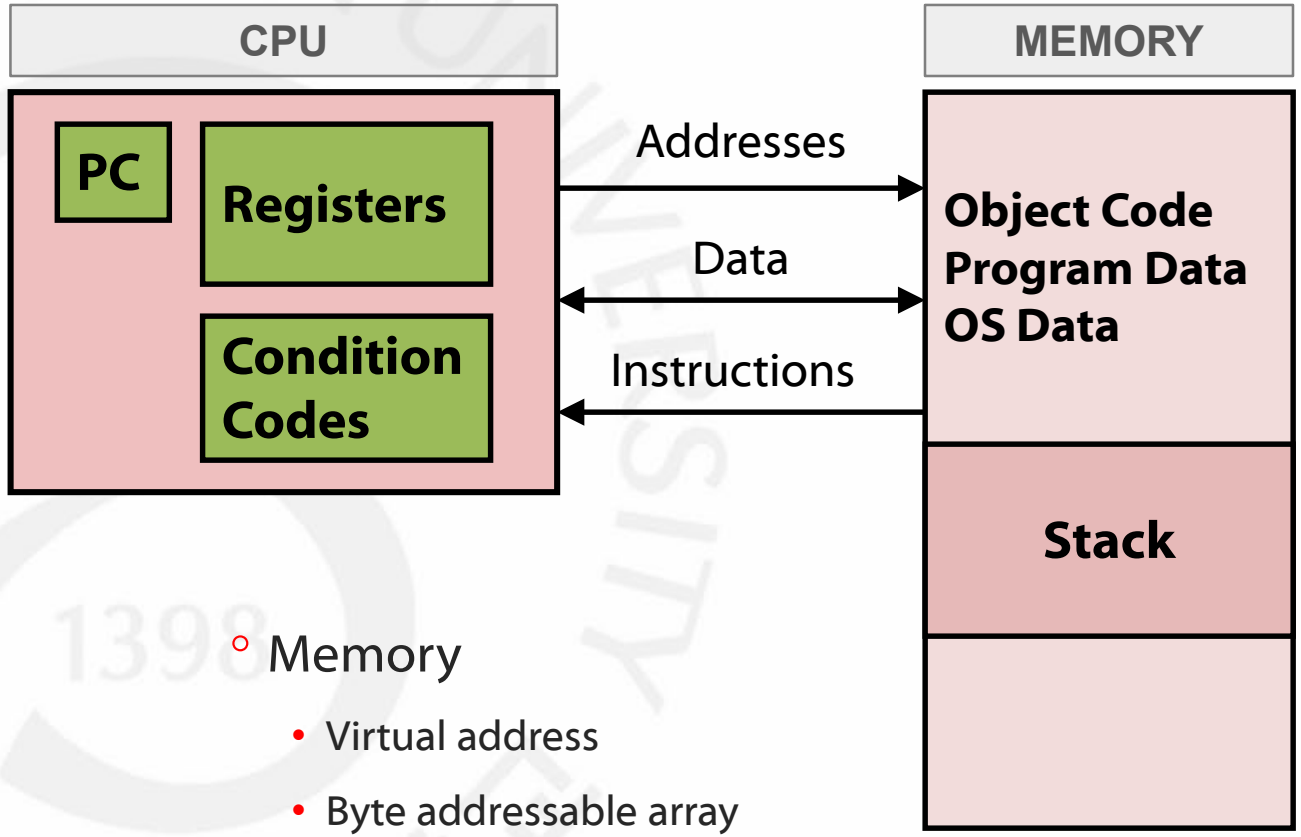


Definitions

- Every processor specifies a set of instructions that can be utilized by the software that intends to run on top of it
  - Every processor will comply to a certain instruction set
    - AMD and Intel comply to the x86 ISA
- **Architecture:** (also ISA: instruction set architecture) parts of a processor design that one needs to understand or write assembly/machine code
  - Examples: instruction set specification, registers
- **MicroArchitecture** specifies details regarding the implementation of features specified in its ISA
  - MicroArchitecture gives the details about how this particular instruction is designed within the processor

Assembly Programmer's View

- Programmer-Visible State
  - PC: Program counter
    - Address of next instruction
    - Called "RIP" (IA32) or "RIP" (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching



- Memory
  - Virtual address
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

Text Box Legend

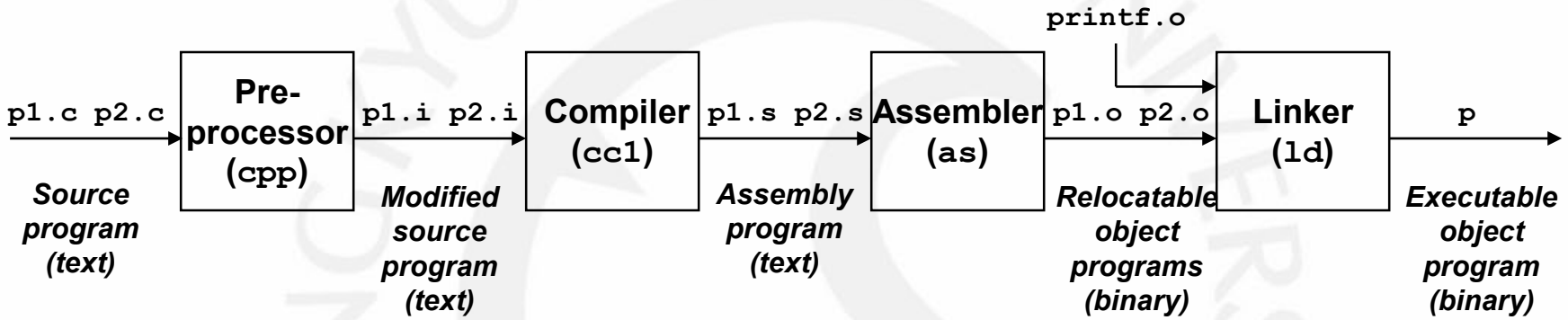
```
int C_code_looks_like_this
{
  int t = x+y;
  return t;
}
```

```
assembly:
        pushl %rbp
        movl %rsp,%rbp
        popl %rbp
        ret
```

Comments will look like this

```
0x40040 <object>:   0x55
                    0x89
                    0xe5
                    0x8b
                    0x45
                    0x0c
                    0x03
                    0x45
                    0x08
                    0x89
                    0xec
                    0x5d
                    0xc3
```


Turning C into Object Code



- Code in files **`p1.c p2.c`**
- Compile with command **`gcc –Og p1.c p2.c -o p`**
  - Use basic optimizations (**`–Og`**)
  - Put resulting binary in file **`p`**

Compiling Into Assembly

- C Code (sum.c)

```
long plus(long x, long y);

void sumstore (long x, long y, long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

- Obtain with command **`gcc -Og -S sum.c`** produces file **`sum.s`**

Data Types in Assembly

- "Integer" data of 1, 2, 4 or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- Code
  - Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Operations in Assembly

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for **`malloc`**, **`printf`**
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution

- Code for **`sumstore`**
  - Total of 14 bytes
  - Each instruction 1, 3, or 5 bytes
  - Starts at address 0x0400595

```
0x0400595:
        0x53
        0x48
        0x89
        0xd3
        0xe8
        0xf2
        0xff
        0xff
        0xff
        0x48
        0x89
        0x03
        0x5b
        0xc3
```


Machine Instruction Example

```
*dest = t;
```

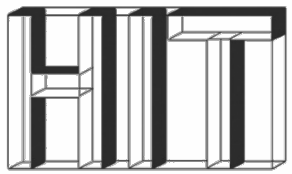
```
movq %rax, (%rbx)
```

```
0x40059e: 48 89 03
```

- C Code
  - Store value `t` where designated by `dest`
- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands

Variable	Register	Memory
t	%rax	
dest	%rbx	
*dest		M[%rbx]

- Object Code
  - 3-byte instruction
  - Stored at address `0x40059e`



Disassembling Object Code

- **`objdump -d sum`**
- Useful tool for examining object code
- Analyzes bit pattern of instructions
- Produces approximate rendition of assembly code
- Can be run on either **`a.out`** (complete executable) or **`.o`** file

- X86-64 instructions can range in length from 1 to 15 bytes
  - Commonly used instructions and those with fewer operands require a smaller number of bytes
- Unique decoding of bytes into machine instructions
  - Only **`pushq %rbx`** can start with value 0x53

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```


Alternate Disassembly

- Within **gdb** Debugger
  - **gdb sum**
  - disassemble **sumstore**
    - Disassemble procedure
  - **x/14xb sumstore**
    - Examine the 14 bytes starting at **sumstore**

```
Dump of assembler code for function sumstore:

0x0000000000400595 <+0>: push   %rbx
0x0000000000400596 <+1>: mov    %rdx,%rbx
0x0000000000400599 <+4>: callq  0x400590 <plus>
0x000000000040059e <+9>: mov    %rax,(%rbx)
0x00000000004005a1 <+12>:pop    %rbx
0x00000000004005a2 <+13>:retq
```

```
0x0400595:
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0xf2
  0xff
  0xff
  0xff
  0x48
  0x89
  0x03
  0x5b
  0xc3
```


Assembly-Code Format

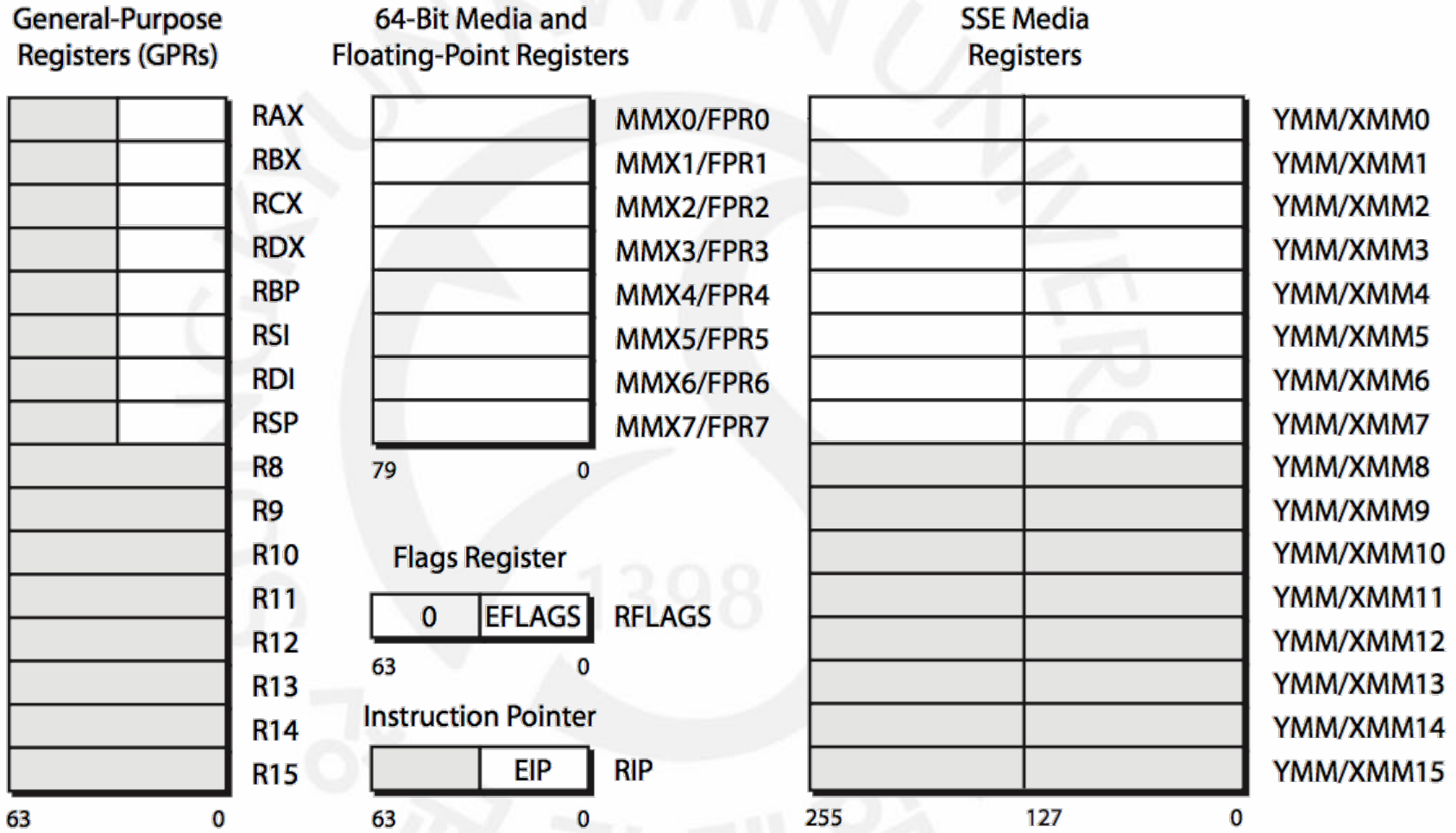
- Aside at p 213
- **ATT** versus Intel
- `gcc -Og -S -masm=intel mstore.c`
- Intel code
  - Omits the size destination suffixes
    - `push` against `pushq`
  - Omits the "%" character in front of register name
    - `rbx` against `%rbx`
  - Has a different way of describing location in memory
    - `QWORD PTR [rbx]`
- Instructions with multiple operands list them in reverse order

```
push   rbx
mov    rbx, rbx
call   mult2
mov    QWORD PTR [rbx], rax
pop    rbx
ret
```

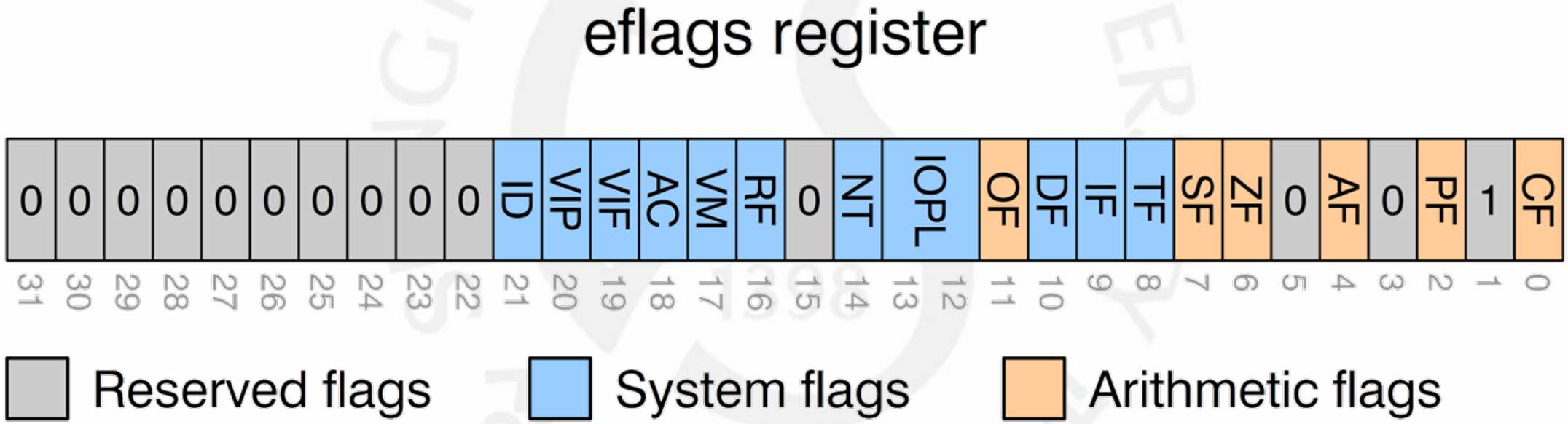

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations

Registers in x86-64



Application-programming registers not shown include Media eXension Control and Status Register (MXCSR) and x87 tag-word, control-word, and status-word registers


eflags register
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Reserved flags
System flags
Arithmetic flags

Data Type (Figure 3.1)

C declaration	Intel data type	Assembly suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

x86-64 Integer Registers

64	32	16	8	0	
%rax	%eax	%ax	%ah	%al	return value
%rbx	%ebx	%bx	%bh	%bl	callee saved
%rcx	%ecx	%cx	%ch	%cl	4th argument
%rdx	%edx	%dx	%dh	%dl	3rd argument
%rsi	%esi	%si		%sil	2nd argument
%rdi	%edi	%di		%dil	1st argument
%rbp	%ebp	%bp		%bpl	callee saved
%rsp	%esp	%sp		%spl	stack pointer
%r8	%r8d	%r8w		%r8b	5th argument
%r9	%r9d	%r9w		%r9b	6th argument
%r10	%r10d	%r10w		%r10b	caller saved
%r11	%r11d	%r11w		%r11b	caller saved
%r12	%r12d	%r12w		%r12b	callee saved
%r13	%r13d	%r13w		%r13b	callee saved
%r14	%r14d	%r14w		%r14b	callee saved
%r15	%r15d	%r15w		%r15b	callee saved

- 16 integer registers
- Can reference low-order 4 bytes
  - Also low-order 1 & 2 bytes
- Kinds of registers
  - Stack pointer
  - Return value
  - Argument up to six
  - Six callee saved
  - Two caller saved

Data Movement Instruction

Category	Instruction	Effect	Description
MOVE	`movb S,D`	`D ← S`	Move byte
	`movw S,D`		Move word
	`movl S,D`		Move double word
	`movq S,D`		Move quad word
	`movabsq I,R`	`R ← I`	Move absolute quad word

- Most heavily used instruction
- Size of register value must match size designated by last character in instruction
- **`mov`** instructions only update specific register bytes or memory location indicated by destination operand

- If source is *immediate*, regular **`movq`** can only have a 32-bit signed integer
  - Value is then *sign* extended to produce 64-bit value for destination
- Moving immediate data to a 64-bit register
  - The **`movabsq`** instruction, when a full 64-bit immediate is required
    - **`movabsq`** can have arbitrary 64-bit immediate
- Instructions that move or generate 32-bit register values also set the upper 32 bits of the register to zero
  - When **`movl`** has a register as destination it will fill upper 4 bytes with zeros

Exercise (p. 220)

movabsq	$0x0011223344556677,%rax	%rax = 00112233 44556677
movb	$-1, %al	%rax = 00112233 445566FF
movw	$-1, %ax	%rax = 00112233 4455FFFF
movl	$-1, %eax	%rax = 00000000 FFFFFFFF
movq	$-1, %rax	%rax = FFFFFFFF FFFFFFFF

Zero-extending Data Movement Instruction

Category	Instruction	Effect	Description
Zero-extending MOVE	`movzbw S,R`	`R ← ZeroExtend(S)`	byte to word
	`movzbl S,R`		byte to double word
	`movzwl S,R`		word to double word
	`movzbq S,R`		byte to quad word
	`movzwq S,R`		word to quad word

- Data movement instruction for copying smaller source to larger destination
- Absence of instruction to zero-extended move for double word to quad word
  - `movzlq`
- Instead, implemented in `movl`

Sign-extending Data Movement Instruction

Category	Instruction	Effect	Description
Sign-extending MOVE	`movsbw S,R`	`R ← SignExtend(S)`	byte to word
	`movsbl S,R`		byte to 2 word
	`movswl S,R`		word to 2 word
	`movsbq S,R`		byte to 4 word
	`movswq S,R`		2 word to 4 word
	`movslq S,R`		
	`cltq`	`%rax ← SignExtend(%eax)`	

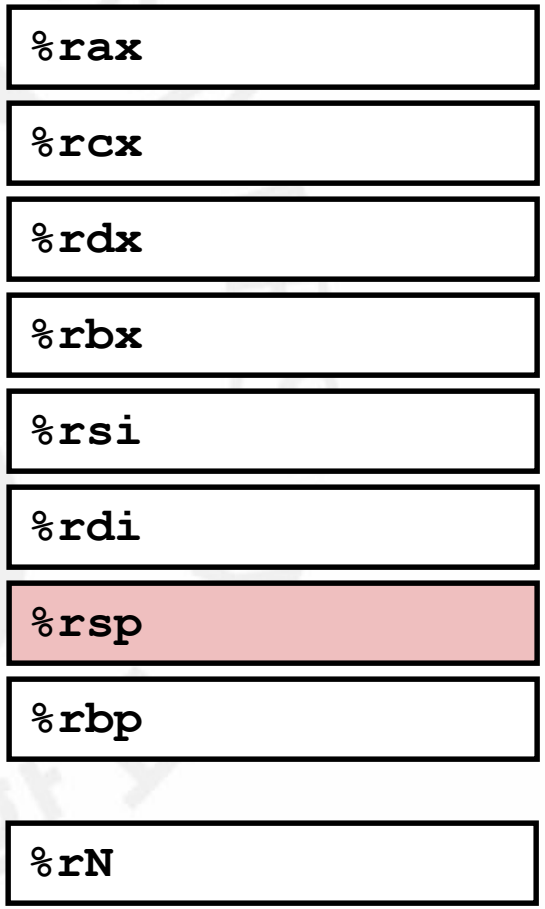
- `cltq` does not have operands
  - Same effect as `movslq %eax, %rax`

Exercise

movabsq	$0x0011223344556677,%rax	%rax = 00112233 44556677
movb	$0xAA, %dl	%dl = AA
movb	%dl, %al	%rax = 00112233 445566AA
movsbq	%dl, %rax	%rax = FFFFFFFF FFFFFFAA
movzbq	%dl, %rax	%rax = 00000000 00000AA

Operand Types

- **Immediate**: Constant integer data
  - Example: `$0x400`, `$-533`
  - Like C constant, but prefixed with '`$`'
  - Encoded with 1, 2, or 4 byte Signed
    - What about 64 bits?
- Register: One of 16 integer registers
  - Example: `%rax`, `%r13`
- Memory: 8 consecutive bytes of memory at address given by register
  - Simplest example: (`%rax`)
  - Various other "address modes"



Operand Combinations

Source	Destination	Example	C analogy
immediate (Imm)	register	`movq $0x4,%rax`	`temp = 0x4;`
	memory	`movq $-147,(%rax)`	`*p = -147;`
register	register	`movq %rax,%rdx`	`temp2 = temp1;`
	memory	`movq %rax,(%rdx)`	`*p = temp;`
memory	register	`movq (%rax),%rdx`	`temp = *p;`

- Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- Normal
  - `(R)`
    - `Mem[Reg[R]]`
  - Register `R` specifies memory address
  - `movq (%rcx),%rax`
- Displacement
  - `D(R)`
    - `Mem[Reg[R]+ D]`
  - Register `R` specifies start of memory region
  - Constant displacement `D` specifies offset
  - `movq 8(%rbp),%rdx`

`Reg[K]`: Value in register `K`
`Mem[L]`: Value in memory `L`

Indexed Addressing Modes

- Most General Form
  - **D(Rb,Ri,S)**
    - **Mem[Reg[R_b]+S × Reg[R_i]+D]**
  - **D** (constant) : displacement 1, 2, or 4 bytes
  - **R_b** (base register) : Any of 16 integer registers
  - **R_i** (index register) : Any, except for **%rsp**
    - Both register must be **64**-bit registers
  - **s** (scale) : 1, 2, 4, or 8

- Special Cases
  - **(R_b,R_i) ⇒ Mem[Reg[R_b]+Reg[R_i]]**
  - **D(R_b,R_i) ⇒ Mem[Reg[R_b]+Reg[R_i]+D]**
  - **(R_b,R_i,S) ⇒ Mem[Reg[R_b]+S*Reg[R_i]]**

Operand Forms

Type	Form	Operand Value	Name
Immediate	$Imm	Imm	Immediate
Register	R_a	R[R_a]	Register
Memory	Imm	M[Imm]	Absolute
Memory	(R_a)	M[R[R_a]]	Indirect
Memory	Imm(R_b)	M[Imm + R[R_b]]	Base + Displacement
Memory	(R_b,R_i)	M[R[R_b]+ R[R_i]]	Indexed
Memory	Imm(R_b,R_i)	M[Imm + R[R_b]+ R[R_i]]	Indexed
Memory	(,R_i,s)	M[R[R_i]× s]	Scaled indexed
Memory	Imm(,R_i,s)	M[Imm+R[R_i× s]	Scaled indexed
Memory	(R_b,R_i,s)	M[R[R_b]+R[R_i]× s]	Scaled indexed
Memory	Imm(R_b,R_i,s)	M[Imm+R[R_b]+R[R_i]× s]	Scaled indexed

Example of Simple Addressing Modes

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

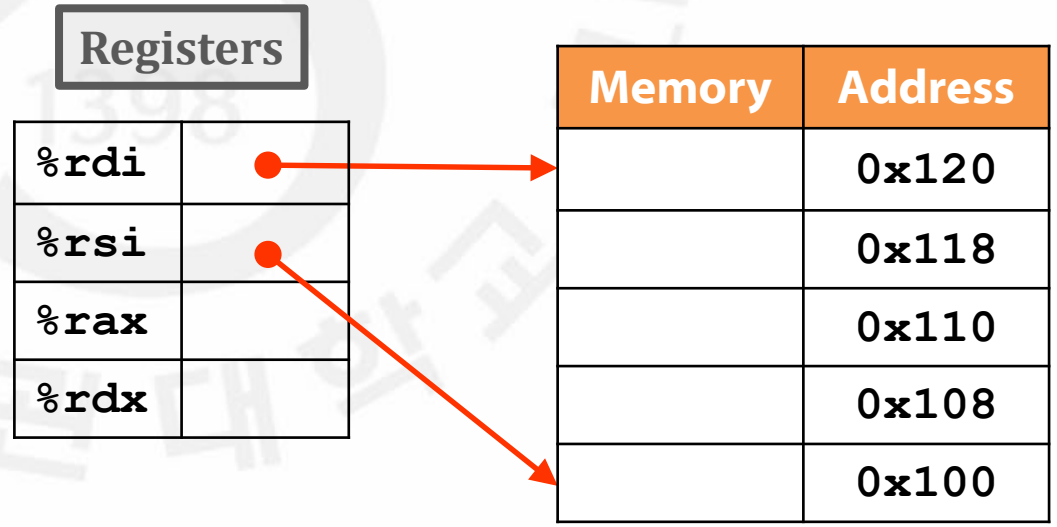
```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```


Understanding Swap

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

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

Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

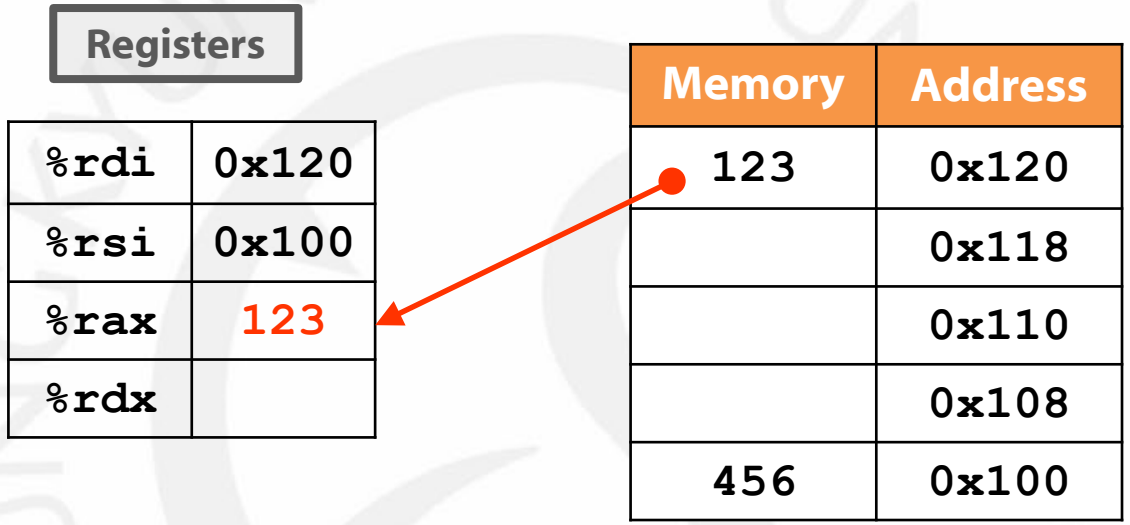


Registers

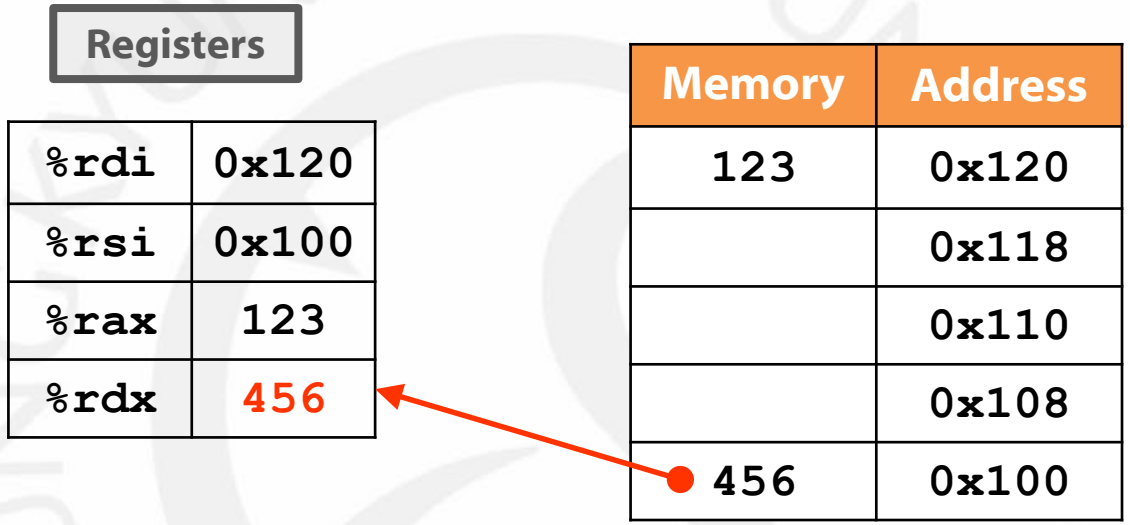
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

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

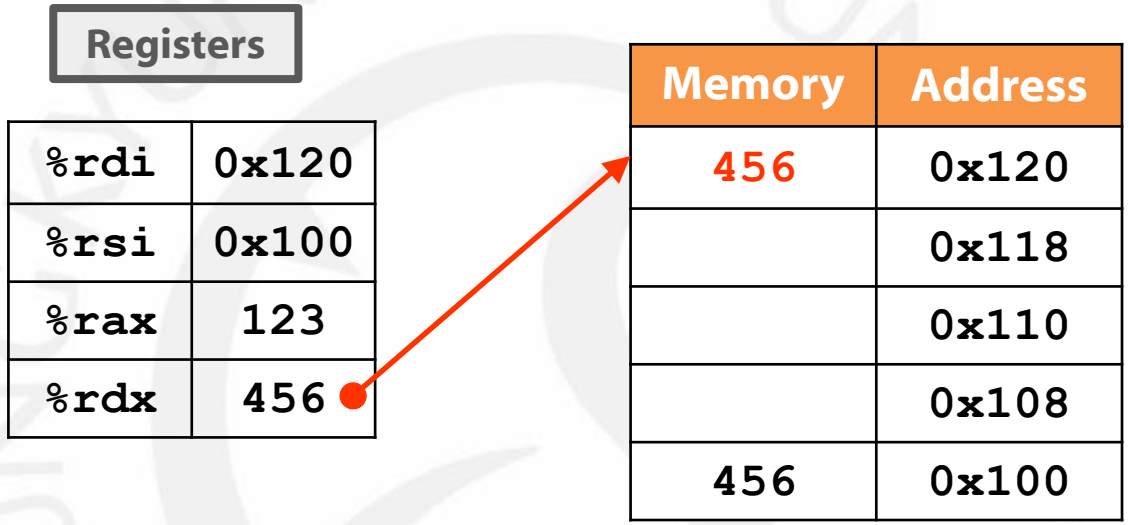
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

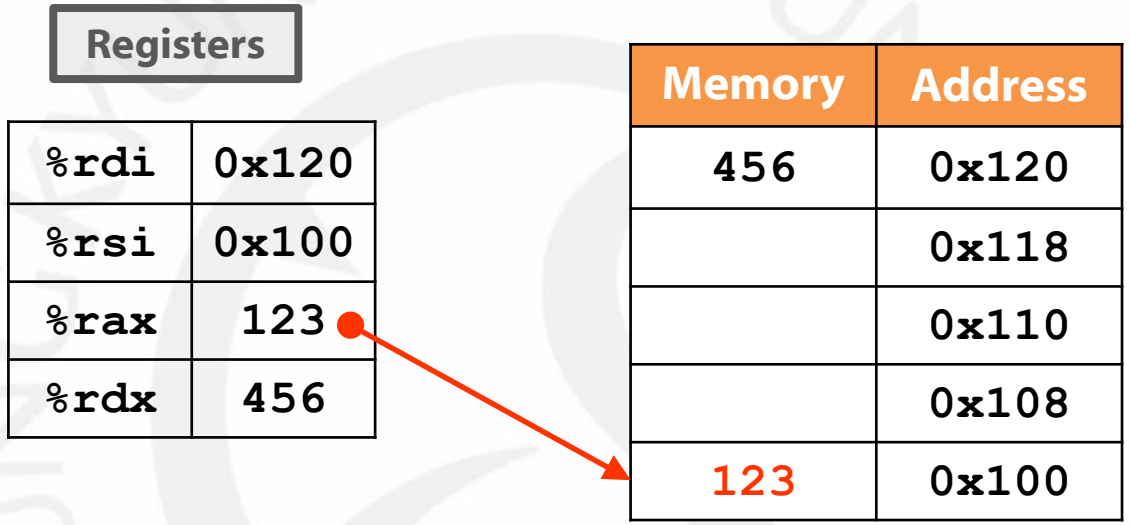

```
swap:
    movq    (%rdi), %rax # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```




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




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




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

Address Computation Examples

- Practice Problem 3.1

%rdx	0xf000

%rcx	0x100

Expression	Computation	Address
0x8 (%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4 × 0x100	0xf400
0x80 (,%rdx,2)	2 × 0xf000 + 0x80	0x1e080

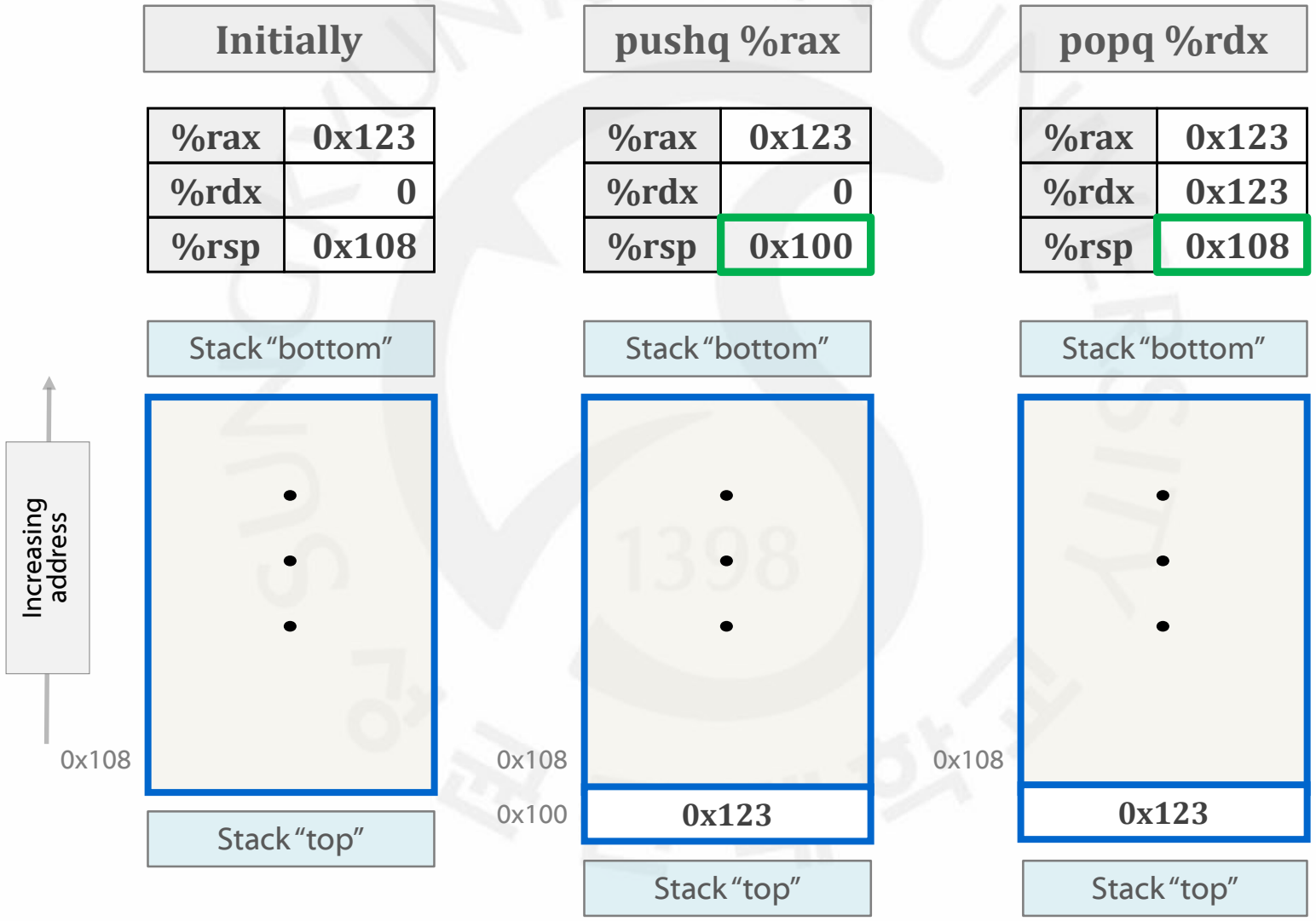
Operations on Stack

`pushq S`	`R[%rsp] ← R[%rsp] - 8;` `M[R[%rsp]] ← S`	Push quad word
`popq D`	`D ← M[R[%rsp]];` `R[%rsp] ← R[%rsp] + 8`	Pop quad word

- Stack is stored in some region of memory
- Stack grows downward

Stack Operation

Value 0x123 remains at memory location until it is overwritten



Data Movement

- Pointer dereferencing *
  - `long x = *xp;`
    - Read value stored in location designated by `xp` and store it as a local variable named `x`
  - `*xp = y;`
    - Write value `y` at location designated by `xp`
- Two features
  - A pointer in C is simple an address
  - A local variable is often kept in a register

```
long exchange (long *xp, long y)
{
  long x = *xp;

  *xp = y;
  return x;
}
```

```
long a = 4;
long b = exchange(&a, 3)
printf("a=%ld, b=%ld\b", a,b);
```

```
movq (%rdi),%rax # 1st argument at %rdi
movq %rsi,(%rdi) # 2nd argumentat %rsi
```


Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- **Arithmetic & logical operations**

Address Computation Instruction

- `leaq` *Src, Dst*
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
- Uses
  - Computing addresses **without** a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k` = 1, 2, 4, or 8

- Example

```
long m12(long x)
{
  return x*12;
}
```

```
leaq (%rdi,%rdi,2), %rax  # t ← x + x × 2
salq $2, %rax             # return t ≪ 2
```


Using leal for Arithmetic Expressions

```
long arith (long x, long y, long z) {
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

- Interesting Instructions
  - **`leaq`**: address computation
  - **`salq`**: shift
  - **`imulq`**: multiplication
    - But, only used once

```
arith:
 leaq  (%rdi,%rsi),%rax      # rax = x+y  (t1)
 addq   %rdx, %rax             # rax = z+t1  (t2)
 leaq(%rsi,%rsi,2),%rdx    # rdx = 3y
 salq$4,%rdx                   # rdx = 48y(t4)
 leaq4(%rdi,%rdx),%rcx     # rcx = x+4+t4(t5)
 imulq %rcx, %rax              # rax = t5×t2  (rval)
 ret
```


Some Arithmetic Operations

- Two Operand Instructions
- Watch out for argument order!
- No distinction between signed and unsigned **`int`** (why?)

`add Src,Dest`	`Dest = Dest + Src`	
`sub Src,Dest`	`Dest = Dest - Src`	
`imul Src,Dest`	`Dest = Dest * Src`	
`sal Src,Dest`	`Dest = Dest << Src`	
`sar Src,Dest`	`Dest = Dest >> Src`	arithmetic
`shr Src,Dest`	`Dest = Dest >> Src`	logical
`xor Src,Dest`	`Dest = Dest ^ Src`	
`and Src,Dest`	`Dest = Dest & Src`	
`or Src,Dest`	`Dest = Dest | Src`	

Some Arithmetic Operations

- One operand instructions

```
incl (%esp)
subl %eax, %edx
```

- Decrements register `%edx` by the value in `%eax`
  - 2 operands cannot both be memory locations

`inc Dest`	`Dest = Dest + 1`	
`dec Dest`	`Dest = Dest - 1`	
`neg Dest`	`Dest = –Dest`	
`not Dest`	`Dest = ~Dest`	

- Shift amount w is encoded as either an *immediate* or a *single* byte
  - Low-order `m` bits of register `%cl`, where $2^m = w$
    - When register `%cl` has value `0xFF`, shift amount w is 7 bits for instruction `salb` and 15 bits for `salw` and 31 for `sall` and 63 for `salq`
  - High-order bits are ignored

Special Arithmetic Operations

- Multiplying two 64-bit signed or unsigned integers can yield 128 bits
- X86-64 instruction provides limited support for 128-bit numbers
  - Oct word (16B)

Instruction	Effect	Description
`imulq S`	`R[%rdx]:R[%rax] ← S × R[%rax]`	Signed full multiply
`mulq S`	`R[%rdx]:R[%rax] ← S × R[%rax]`	Unsigned full multiply
`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word
`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S;`	Signed divide
`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;` `R[%rax] ← R[%rdx]:R[%rax] ÷ S;`	Unsigned divide

Two Different Forms of `imul`

- Two operands
  - Generating a 64-bit product from two 64-bit operands
- One operand
  - Full 128-bit product of two 64-bit values
  - One argument must be in register %**rax**
  - Product is stored in register %**rdx** (high-order 64 bits) and register %**rax** (low-order 64 bits)
  - Little-endian

```
#include <inttypes.h>

typedef unsigned __int128 uint128_t;

void store_uprod
(unit128_t *dest, unit64_t x, uint64_t y) {
  *dest = x * (uint128_t) y;
}
```

```
store_uprod:

  movq %rsi,%rax
  mulq %rdx
  movq %rax,(%rdi)
  movq %rdx,8(%rdi)
  ret
```


64-bit Division

- Dividend of 128-bit quantity
  - Register `%rdx` (high-order) and register `%rax` (low-order)
- Divisor is given as operand
- Instruction stores quotient in register `%rax` remainder in register `%rdx`
- Most application dividend is given as 64 bit value stored in `%rax`
  - Bits of `%rdx` should be set to either
    - All zeros (unsigned) or sign bit of `%rax`
    - The latter is performed using instruction `cqto`

```
void remdiv
(long x, long y, long *qp, long *rp) {
  long q = x/y;
  long r = r%y;
  *qp = q;
  *rp = r;
}
```

```
remdiv:
  movq %rdx,%r8     #copy qp
  movq %rdi,%rax
  cqto
  idivq %rsi
  movq %rax,(%r8)   #store quotient
  movq %rdx,(%rcx)  #store remainder
  ret
```


Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code (instruction) and microcode
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation