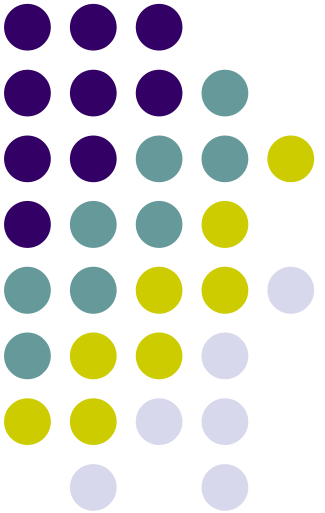
Computer Organization: A Programmer's Perspective

Machine-Level Programming (1: Introduction)



Gal A. Kaminka
galk@cs.biu.ac.il

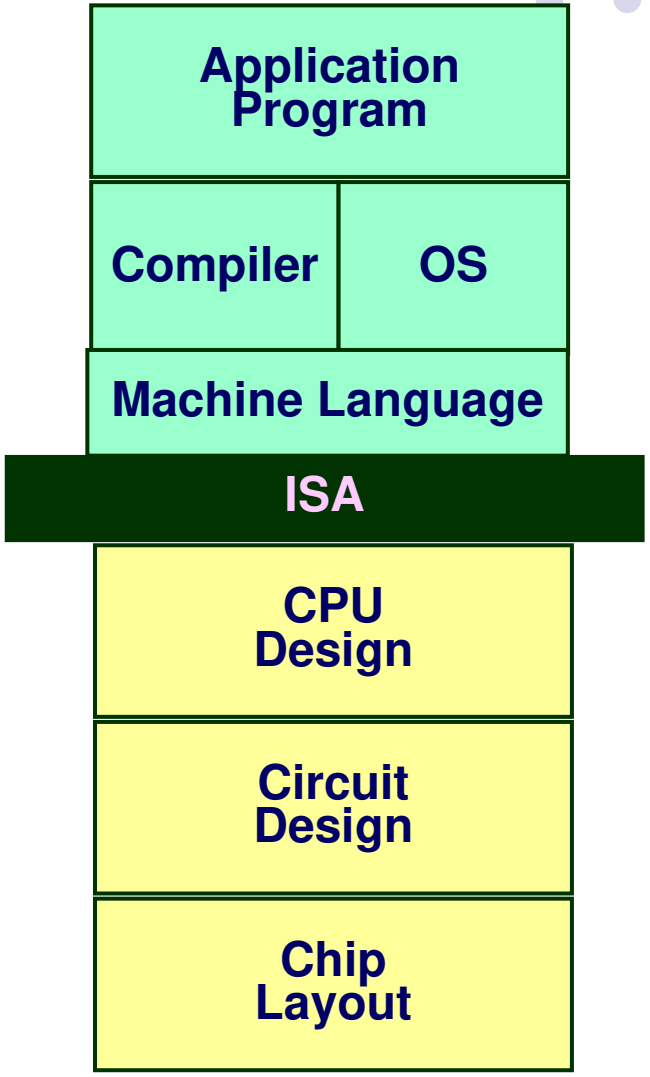
Instruction Set Architecture

Assembly Language View

- Processor state
  - Registers, memory, …
- Instructions
  - `addl`, `movl`, `leal`, …
  - How instructions are encoded as bytes

Layer of Abstraction

- Above: how to program machine
  - Processor executes instructions in a sequence
- Below: what needs to be built
  - Use variety of tricks to make it run fast
  - E.g., execute multiple instructions simultaneously



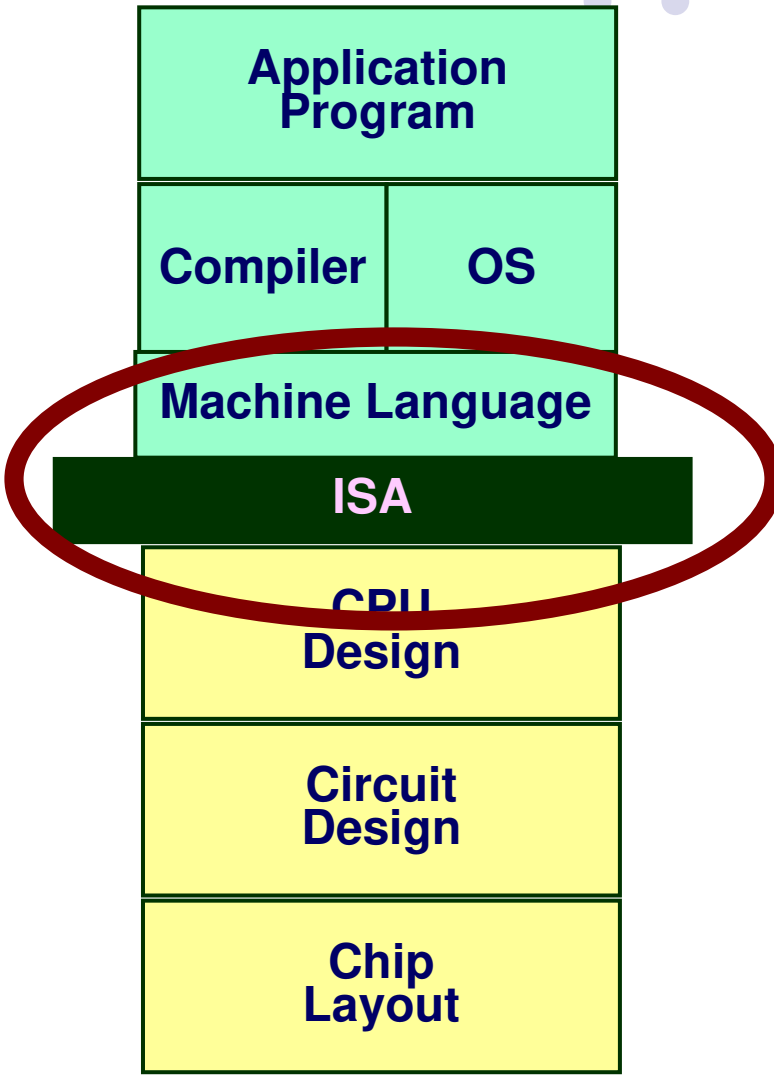
Instruction Set Architecture

Assembly Language View

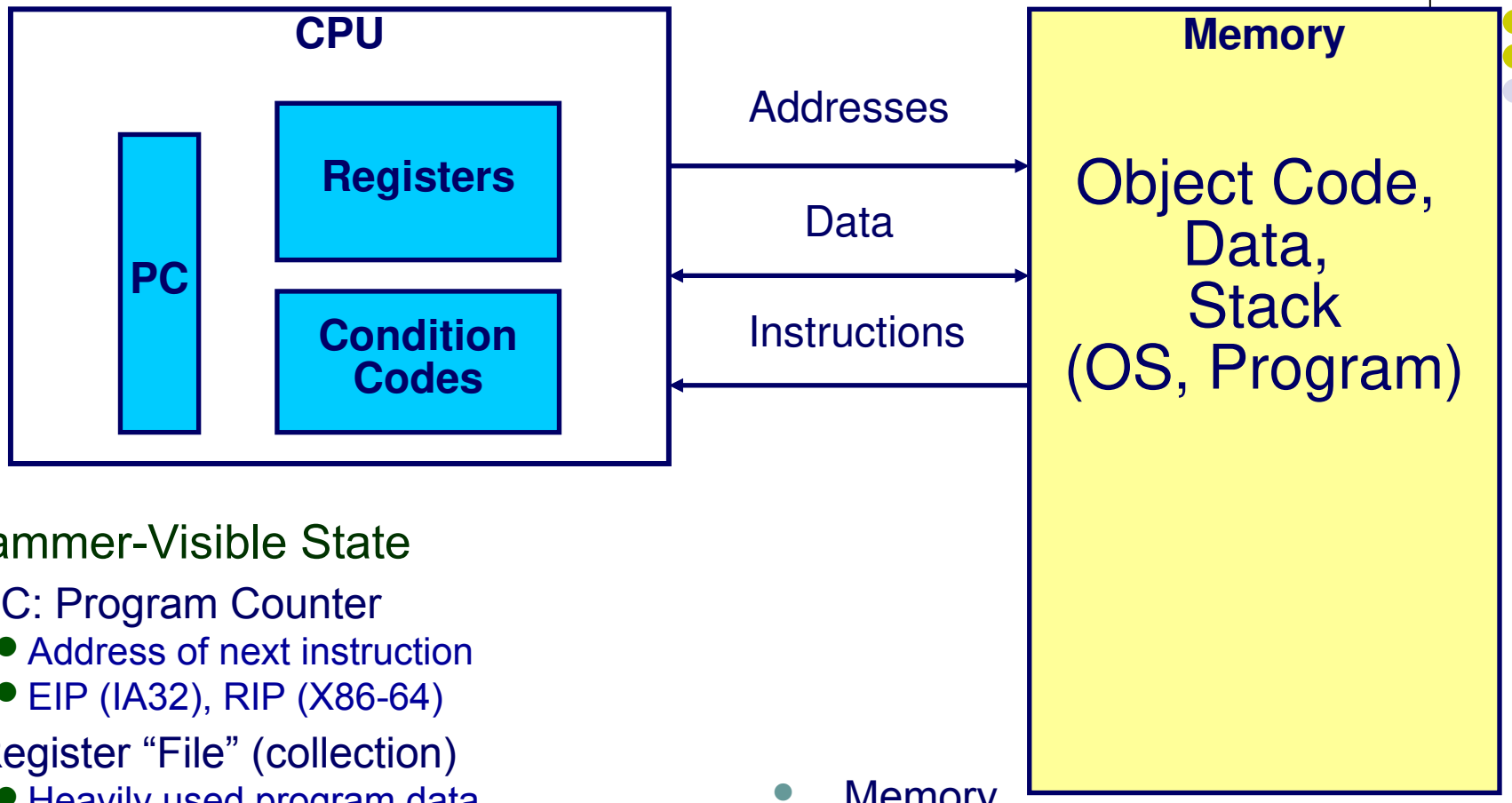
- Processor state
  - Registers, memory, …
- Instructions
  - `addl`, `movl`, `leal`, …
  - How instructions are encoded as bytes

Layer of Abstraction

- Above: how to program machine
  - Processor executes instructions in a sequence
- Below: what needs to be built
  - Use variety of tricks to make it run fast
  - E.g., execute multiple instructions simultaneously



Assembly Programmer's View

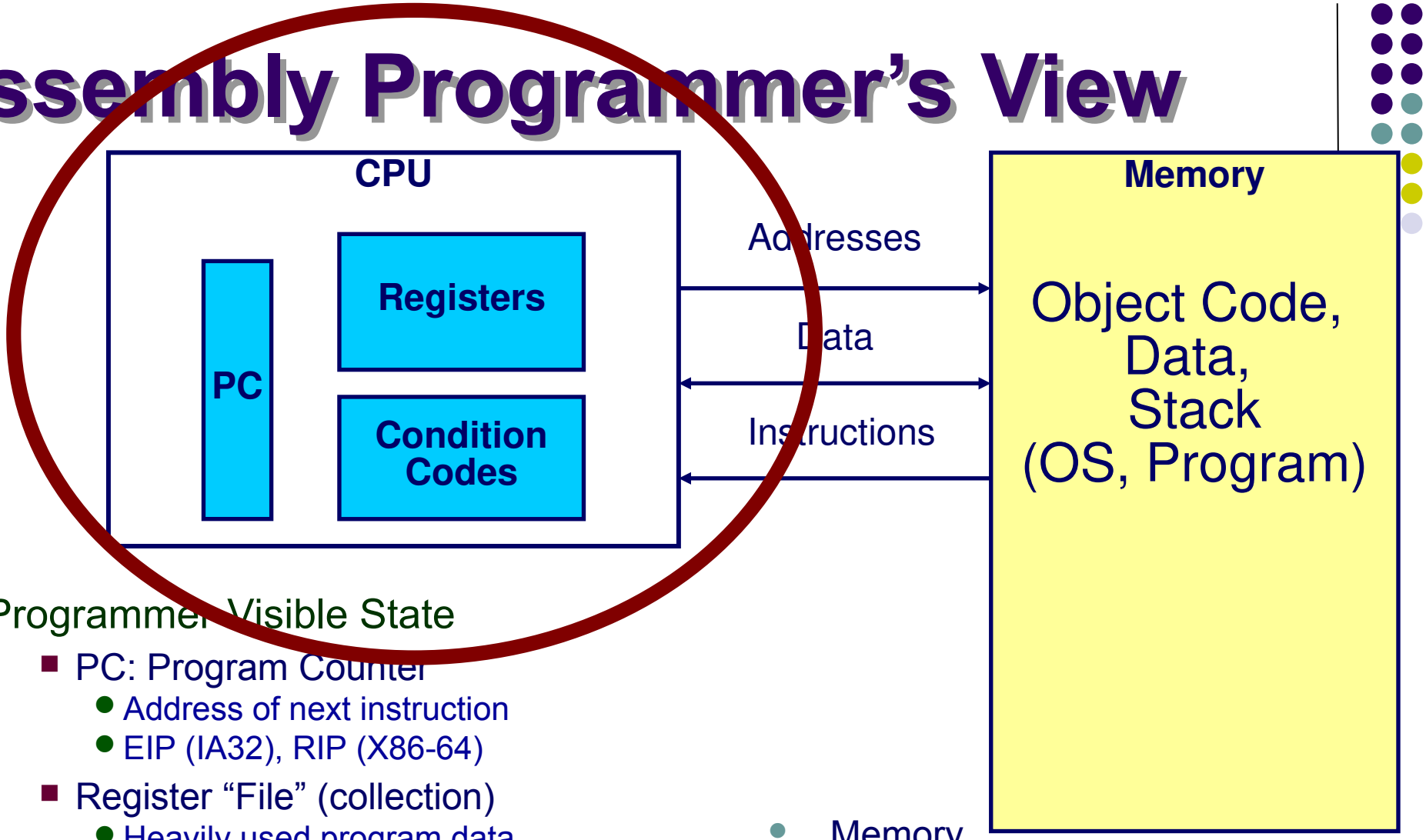


Programmer-Visible State

- PC: Program Counter
  - Address of next instruction
  - EIP (IA32), RIP (X86-64)
- Register "File" (collection)
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

- Memory
  - Byte addressable array
  - Code, user data, OS data
  - Includes stack used to support procedures
  - OS controls permissions

Assembly Programmer's View

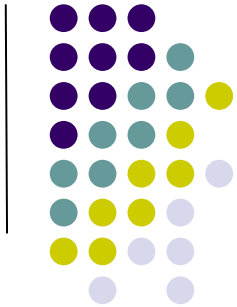


Programmer-Visible State

- PC: Program Counter
  - Address of next instruction
  - EIP (IA32), RIP (X86-64)
- Register "File" (collection)
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

- Memory
  - Byte addressable array
  - Code, user data, OS data
  - Includes stack used to support procedures
  - OS controls permissions

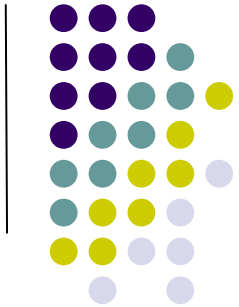
Abstract View of CPU



```
void cpu(void) {
    int iregs[ ] ;  // word registers
    bit flags[ ]  ;  // 1-bit flags:  carry, zero, overflow, …
    float fregs[ ] ;  // floating point – not all CPUs

    address register IP =  FIRST_ADDRESS
    while (true) {
      (instruction) ← get_next_instruction(IP)
      (iregs, fregs, flags, IP) ← execute(instruction,iregs,fregs,flags)
    }
}
```

Abstract View of CPU

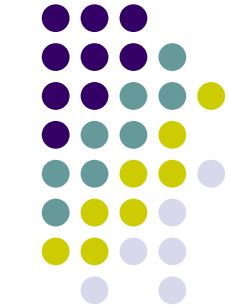


```
void cpu(void) {
   int iregs[ ] ;  // word registers
   bit flags[ ]  ;  // 1-bit flags:  carry, zero, overflow, …
   float fregs[ ] ;  // floating point – not all CPUs

   address register IP =  FIRST_ADDRESS
   while (true) {
     (instruction) ← get_next_instruction(IP)
     (iregs, fregs, flags, IP) ← execute(instruction,iregs,fregs,flags)
   }
}
```

ISA changes available registers, available instructions

Compiling Into Assembly



C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Obtain with command

```
gcc -O -S sum.c
```

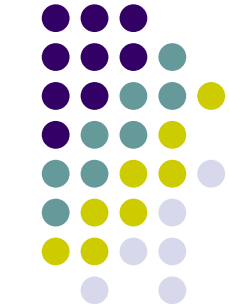
Produces file sum.s

Generated Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

IA32 assembly!

Compiling Into Assembly



C Code

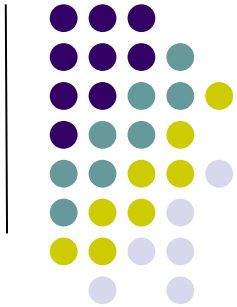
```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

registers: ebp, esp, eax

Compiling Into Assembly



C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

instructions: movl, addl, pushl, popl, ret

Compiling Into Assembly

C Code (sum.c)

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated Assembly

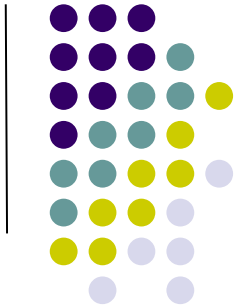
```
sum:
   leal    (%rdi,%rsi), %eax
   ret
```

AMD64/x86-64 assembly!

Different ISA

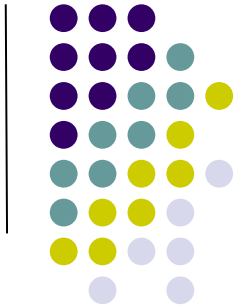
- **Different registers (rdi, rsi)**
- **Some commands different**

ISA vs processor (CPU)



- ISA defines the expected behavior and capabilities
  - How many registers? What type? What size?
  - What instructions are available?
  - What addressing modes (memory access) available?
  - What condition codes available? How can they be checked?
  - …
- CPUs implement ISA
  - For example, Intel and AMD compete on same ISA implementation
    - The competition includes other companies, too
  - CPUs may offer better speed on some instructions or on all
  - May use less or more energy
  - Be more robust to radiation, temperature, etc.

X86 ISA Evolution



Name	Date	|Word|	# General Regs
X86-16	1978	16 bits	4 (+4 limited)

- e.g., Intel 8086 16-bit processor. Basis for IBM PC & DOS
  - Limited to 1MB address space. DOS only gives you 640K
- CPUs produced by Intel AMD, VIA Technologies, NEC
- 8088, 80186, 80286, … all extending. # of general registers same
- Registers: AX, BX, CX, DX, …

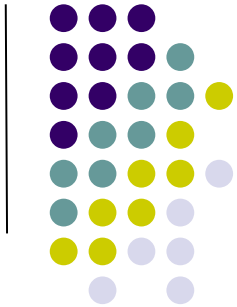
IA32(1)	1985	32 bits	8 (+6 limited)

- Intel 386, AMD Am386
- Extended to 32 bits. Added “flat addressing”, modern OS support: capable of running Unix

IA32(2)	1989	32 bits	8 (+6 limited) +8 FP (80 bits)

- Floating Point in hardware!
- Intel 486, but also by IBM, AMD, TI, … Combined 386+387 co-processor

X86 ISA Evolution



Name	Date	|Word|	# General Regs
IA32(3)	1997+	32	8 (+6 limited) + 8 FP (80 bits) + 8 64bit integers (XMM)

- SSE: SIMD instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data
- Later: SSE2, SSE3, … and other instruction extensions

X86-64/AMD64	2000	64	16 (+ 6 limited) + 16 128bit (XMM) + 8 FP, ...

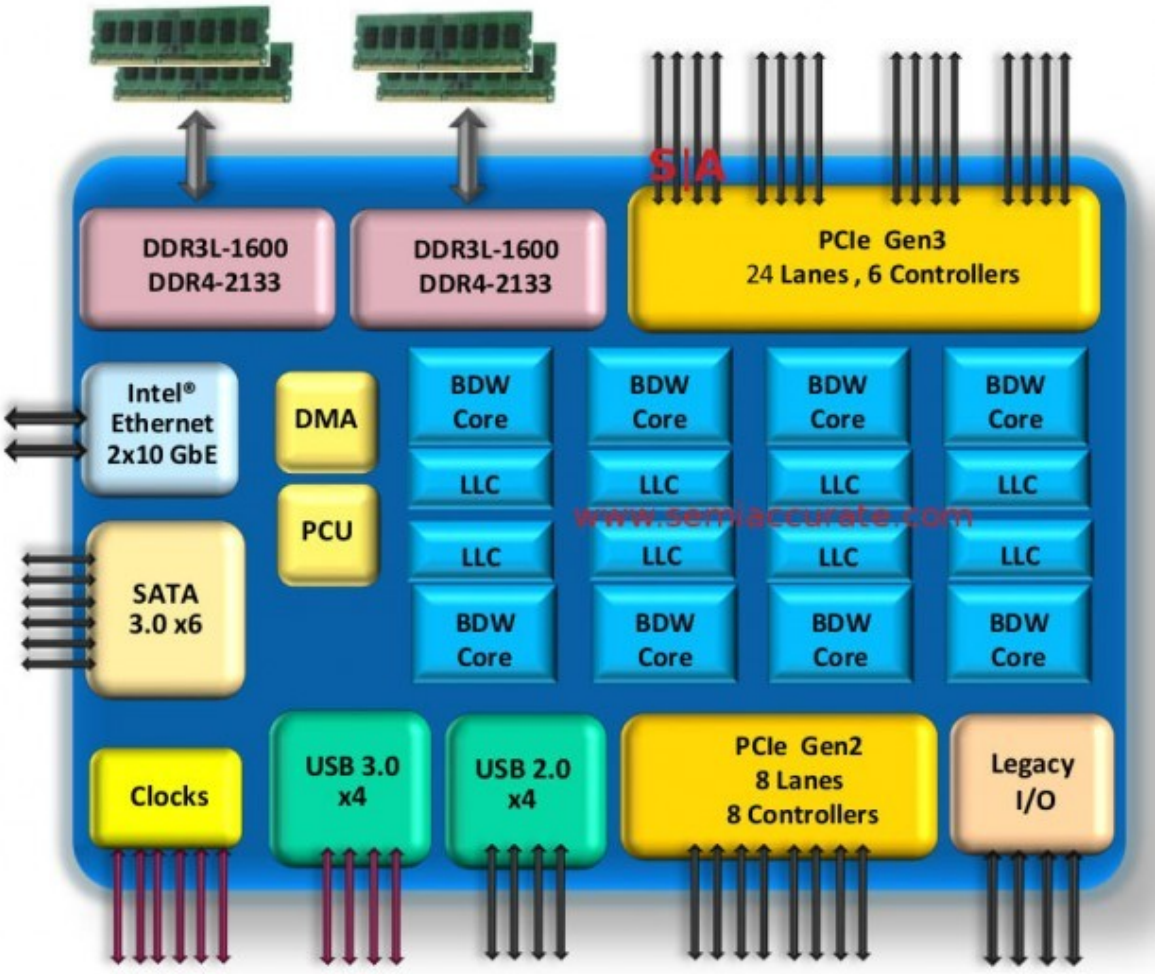
- Capable of running 16/32 bit binaries of older architecture (backwards compatible)
- CPUs by AMD, Intel, VIA

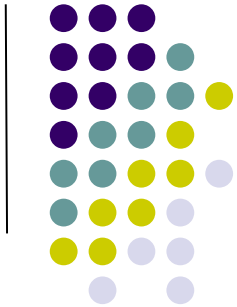
Later additions: multiple cores (2006-), virtualization (2007-), AES, …

2015 State of the Art: Core i7 Broadwell

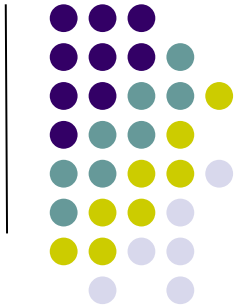
- **Desktop Model**
  - 4 cores
  - Integrated graphics
  - 3.3-3.8 GHz
  - 65W
- **Server Model**
  - 8 cores
  - Integrated I/O
  - 2-2.6 GHz
  - 45W

Same ISA





Assembly Characteristics



Minimal Data Types in Registers

- "Integer" data of 1, 2, 4, 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Exception: SIMD registers (XMM in X86)

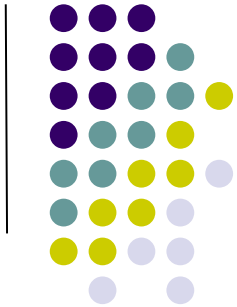
Limited Number of Registers

- Typically a few dozen (or less)

Code: Byte sequences that encode instructions

- Choice of instruction determines type of data!

Assembly Characteristics: Operations



- **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

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;
}
```

Generated Assembly

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

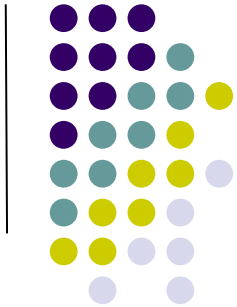
AMD64/x86-64 assembly!

Obtain with command

```
gcc -S sum.c
```

Produces file `sum.s`

Object Code from Assembly



Machine Code example

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

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

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Image of executable code (almost)
  - Missing linkages between code in different files

- **Run 'gcc -c sum.s'**
- **Examine result using objdump**

Machine Instruction Example

```
*dest = t;
```

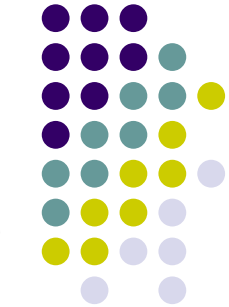
```
movq %rax, (%rbx)
```

```
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:
    t: Register **%rax**
    dest: Register **%rbx**
    ***dest:** Memory **M[%rbx]**
- **Object Code**
  - 3-byte instruction
  - Stored at address **0x40059e**

Disassembling Object Code

`objdump -d sumstore.o`



Disassembled

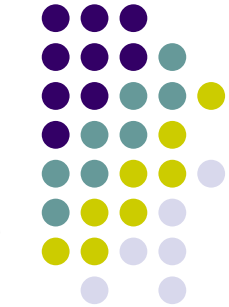
```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3             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

Disassembler

`objdump -d sum`

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

Alternate Disassembly



Object

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

Disassembled

```
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
```

Within gdb Debugger

`gdb sum`

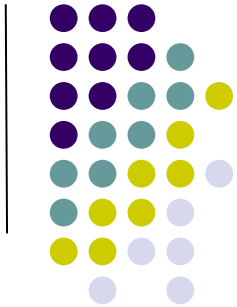
`disassemble sumstore`

- Disassemble procedure

`x/14xb sumstore`

- Examine the 14 bytes starting at `sumstore`

What Can be Disassembled?



- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Reverse engineering forbidden by Microsoft End User License Agreement

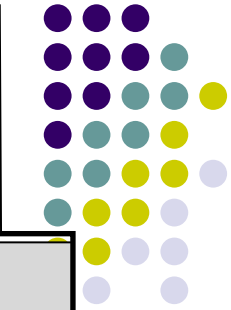
```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55              push   %ebp
30001001:  8b ec           mov    %esp,%ebp
30001003:  6a ff           push   $0xffffffff
30001005:  68 90 10 00 30  push   $0x30001090
3000100a:  68 91 dc 4c 30  push   $0x304cdc91
```

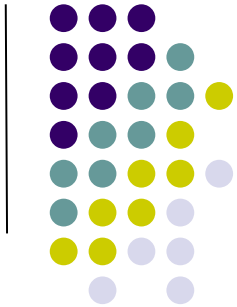
x86-64 Integer Registers

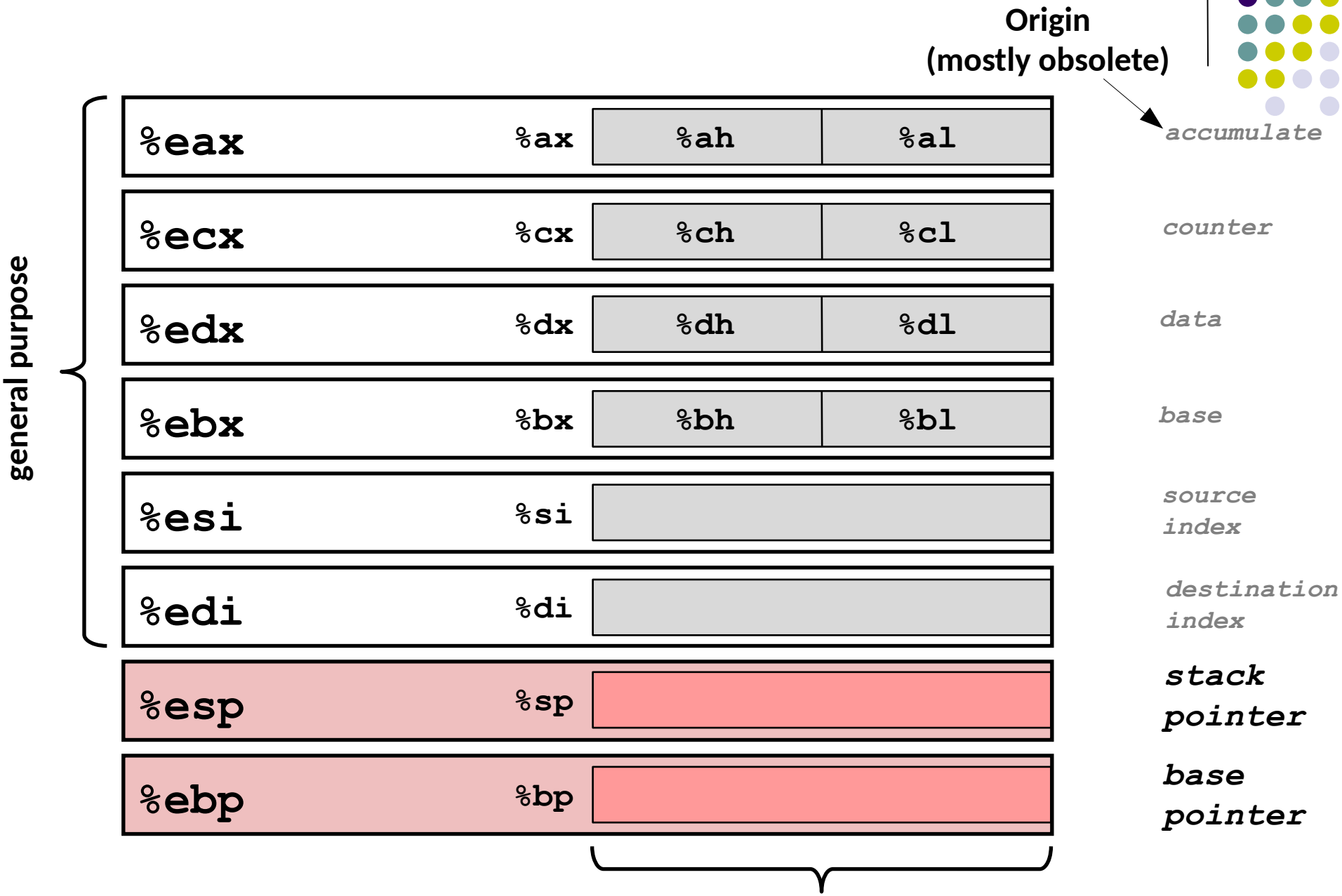


%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers



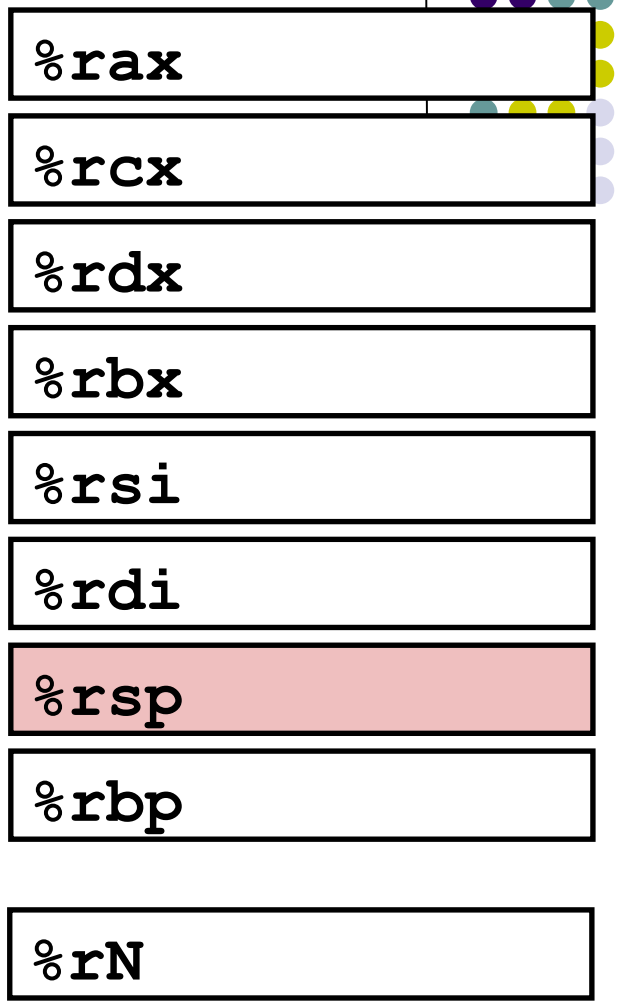


Moving Data

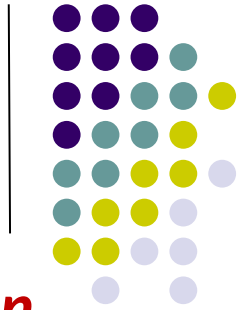
- **Moving Data**

  `movq` *Source*, *Dest*:

- **Operand Types**
  - ***Immediate:*** Constant integer data
    - Example: `$0x400`, `$-533`
    - Like C constant, but prefixed with `'$'`
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: `%rax`, `%r13`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: `(%rax)`
    - Various other "address modes"



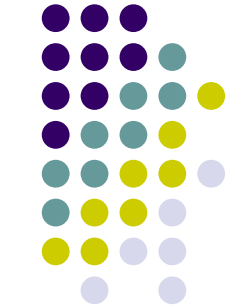
`movq` Operand Combinations



Cannot do memory-memory transfer with a single instruction

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Simple Memory Addressing Modes



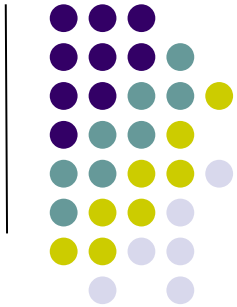
- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Pointer dereferencing in C: "(%rcx)" is really " *rcx " in C notation

```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset (think of "*(rbp+8)" in C)

```
movq 8(%rbp),%rdx
```

Example of Simple Addressing Modes

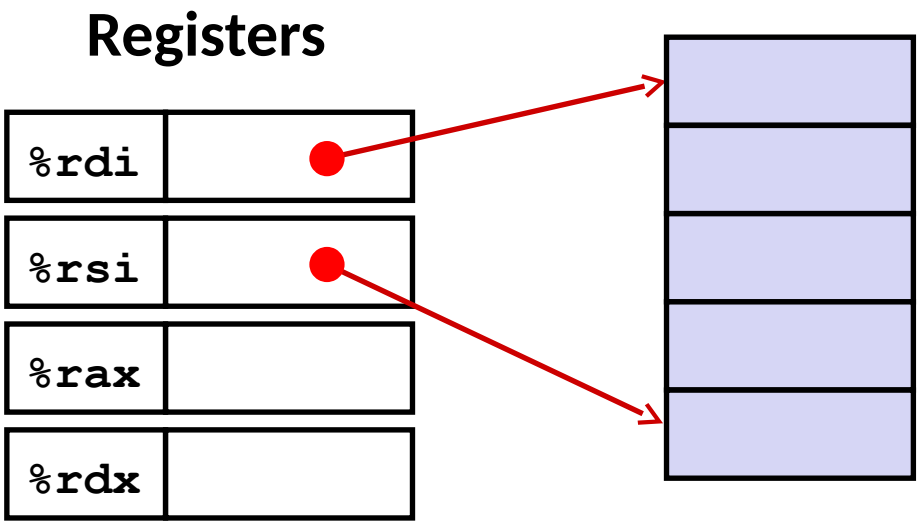


```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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

Understanding Swap()

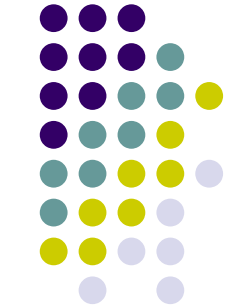
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

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

Complete Memory Addressing Modes



- **Most General Form**

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]

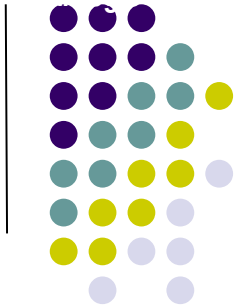
  (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address 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

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**

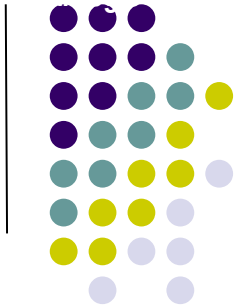
```
long m12(long x)
{
  return x*12;
}
```

`%rdi` contains x

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Some Arithmetic Operations



- **Two Operand Instructions:**

FormatComputation

`addq`	Src,Dest	Dest = Dest + Src	
`subq`	Src,Dest	Dest = Dest - Src	
`imulq`	Src,Dest	Dest = Dest * Src	
`salq`	Src,Dest	Dest = Dest << Src	Also called shlq
`sarq`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrq`	Src,Dest	Dest = Dest >> Src	Logical
`xorq`	Src,Dest	Dest = Dest ^ Src	
`andq`	Src,Dest	Dest = Dest & Src	
`orq`	Src,Dest	Dest = Dest | Src	

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

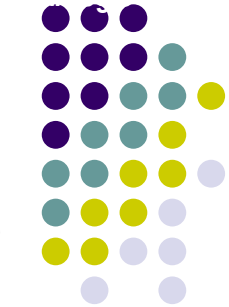
Some Arithmetic Operations

- **One operand instructions**

Format	Computation	
`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest - 1
`negq`	Dest	Dest = -Dest
`notq`	Dest	Dest = ~Dest

- **Lots more**

Arithmetic Expression Example (64 bit)



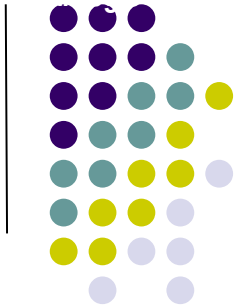
```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Interesting Instructions

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

Understanding Arithmetic Expression Example

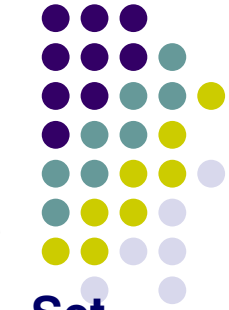


```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx  # rdx = 3y
   salq    $4, %rdx             # t4 (3y*16)
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z, 3y
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Another Example



```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

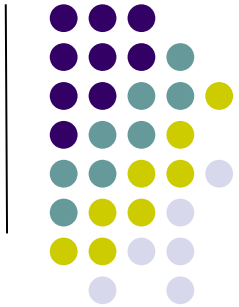
Body

Finish

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y     (t1)
sarl $17,%eax          eax = t1>>17  (t2)
andl $8185,%eax        eax = t2 & 8185
```

Whose Assembler?



Intel/Microsoft Format

```
lea   eax,[ecx+ecx*2]
sub   esp,8
cmp   dword ptr [ebp-8],0
mov   eax,dword ptr [eax*4+100h]
```

GAS/Gnu Format

```
leal (%ecx,%ecx,2),%eax
subl $8,%esp
cmpl $0,-8(%ebp)
movl $0x100(,%eax,4),%eax
```

Intel/Microsoft Differs from GAS

- Operands listed in opposite order

  `mov` Dest, Src `movl` Src, Dest

- Constants not preceded by '$', Denote hex with 'h' at end

  `100h` `$0x100`

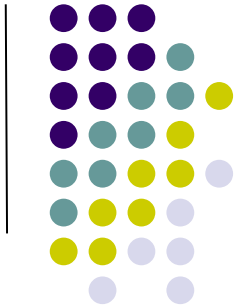
- Operand size indicated by operands rather than operator suffix

  `sub` `subl`

- Addressing format shows effective address computation

  `[eax*4+100h]` `$0x100(,%eax,4)`

Halt and Catch Fire (HCF)



- פקודות אסמבלי ידועות בשמות מוזרים (LEA? באמת?)
- במחשב IBM SYSTEM/360, התגלה באג במעבד שגרם לו להתקע
- בצחוק, המתכנתים החליטו לסמן את הפקודה ב HCF
- יותר מאוחר, היא הפכה לפקודה רשמית
  - למטרות בדיקה עצמית – לא עושה שום דבר הרסני
- ועוד אחר כך, לכינוי כללי לפקודות מכונה שגורמות למחשב להתקע
  - קרה מאז בהרבה מעבדים, כולל מעבדי X86
- ועוד אחר כך: סדרת טלוויזיה מרתקת על תחילת עידן המחשב האישי
  - אבל זה לאחרי הסמסטר, בסדר?