[CSED211] Introduction to Computer Software Systems

Lecture 4: Machine-Level Basic

Prof. Jisung Park

2023.09.18

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, Assembly, and Machine Code
- Assembly Basics: Registers, Operands, Move
- Arithmetic & Logical Operations

Intel x86 Processors

- Dominate laptop/desktop/server markets

- Evolutionary 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
    - But only small subset encountered with Linux programs
  - Hard to match the performance of Reduced Instruction Set Computers (RISC)
    - But Intel has done just that!
    - In terms of speed. Less so for power consumption.

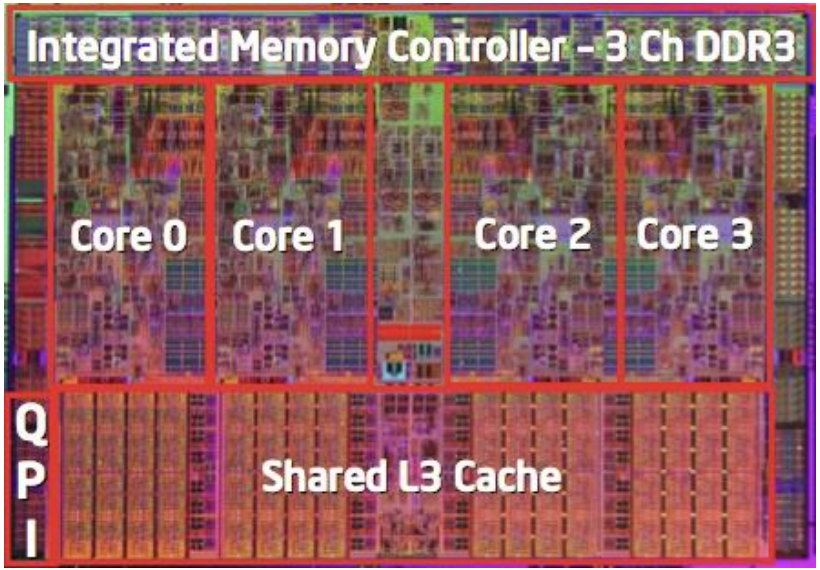
Intel x86 Evolution: Milestones

Name	Date	# of Transistors	Frequency [MHz]
8086	1978	29K	5-10
○ First 16-bit Intel processor. Basis for IBM PC & DOS			
○ 1MB address space			
386	1985	275K	16-33
○ First 32-bit Intel processor, referred to as IA32			
○ Added flat addressing, capable of running Unix			
Pentium 4E	2004	125M	2,800-3,800
○ First 64-bit Intel x86 processor, referred to as x86-64			
Core 2	2006	291M	1,060-3,500
○ First multi-core Intel processor			
Core i7	2008	731M	1,700-3,900
○ Four cores			

Intel x86 Processors: More History

- Machine Evolution

386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M
Core i7 Skylake	2015	1.9B
Core i9	2019	3.5B



- Added Features
  - Instructions to support multimedia operations (Pentium/MMX)
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores

Intel x86 Processors, cont.

- Past Generations

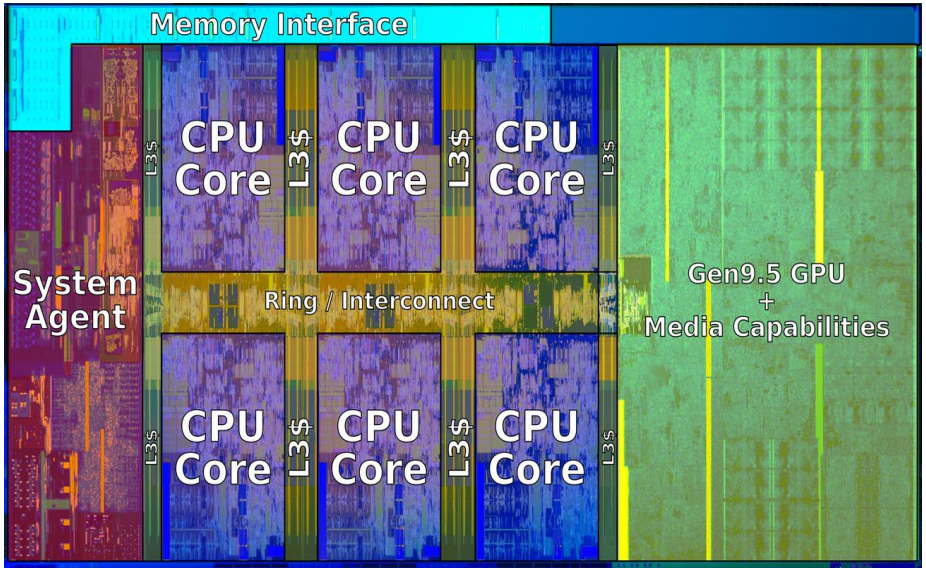
		Process technology
○ 1st Pentium Pro	1995	600 nm
○ 1st Pentium III	1999	250 nm
○ 1st Pentium 4	2000	180 nm
○ 1st Core 2 Duo	2006	65 nm

Process technology dimension
= width of narrowest wires
(10 nm ≈ 100 atoms wide)

- Recent & Upcoming Generations

○ Nehalem	2008	45 nm
○ Sandy Bridge	2011	32 nm
○ Ivy Bridge	2012	22 nm
○ Broadwell	2014	14 nm
○ Skylake	2015	14 nm
○ Cannon Lake	2018	10 nm
○ Alder Lake	2021	10 nm

2018 State of the Art: Coffee Lake



- Mobile model: Core i7
  - 2.2–3.2 GHz
  - 45 W
- Desktop model: Core i7
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 W
- Server model: Xeon E
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 W

x86 Clones: 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. and other companies
  - Built Opteron: a tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- Recent Years
  - Intel got its act together
    - 1995-2011: lead semiconductor fab in the world
    - 2019: #2 largest by $$ (#1 is Samsung)
  - AMD has fallen behind
    - Relies on external semiconductor manufacturer GlobalFoundaries
    - ca. 2019 CPUs (e.g., Ryzen) are competitive again

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 EM64T 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
  - `$ gcc hello.c`
  - `$ gcc -m64 hello.c`

- Presentation
  - Book covers x86-64
  - Web aside on IA32: http://csapp.cs.cmu.edu/3e/waside.html
  - Lectures will only cover x86-64

Today: Machine Programming I: Basics

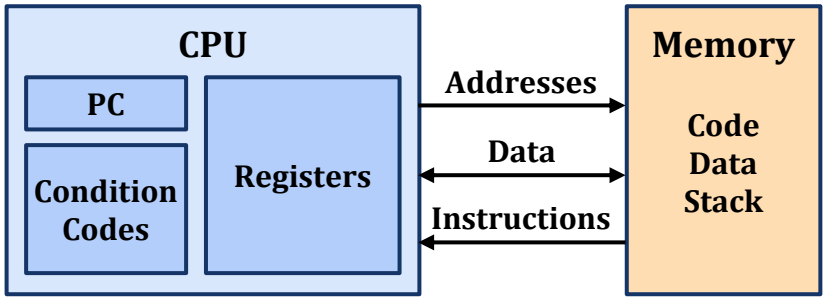
- History of Intel processors and architectures
- C, Assembly, and Machine Code
- Assembly Basics: Registers, Operands, Move
- Arithmetic & Logical Operations

Levels of Abstraction

C Programmer

C Code

Assembly Programmer



Computer Architect & HW Designer

Caches, clock freq, layout, …

Nice clean layers,
but beware…

Of course, you know that;
it's why you are taking this course

Definitions: ISA vs. Microarchitecture

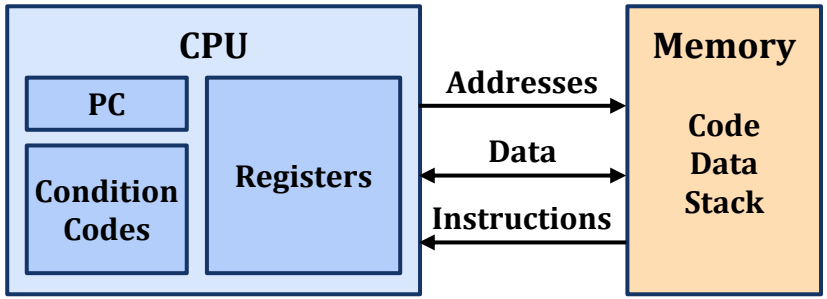
- ISA (instruction set architecture) or architecture: part of a processor design that one needs to understand for writing correct assembly/machine code
  - Examples: instruction set specification, registers.
  - Machine (binary) code: the byte-level programs that a processor executes
  - Assembly code: a text representation of machine code (using mnemonics)

- Microarchitecture: implementation of the architecture.
  - Examples: cache sizes and core frequency.

- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: used in almost all mobile phones
  - RISC-V: a new open-source ISA

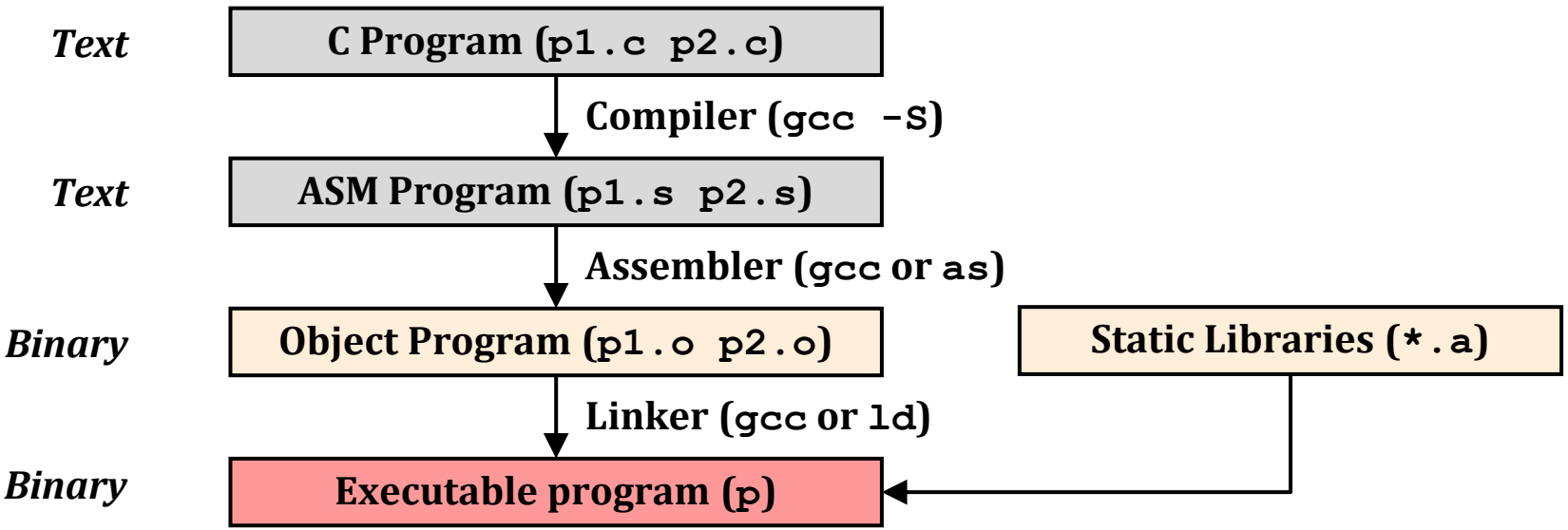
Assembly Programmer's View

- Programmer-visible states
  - PC: program counter
    - Address of the next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - Register file
    - Stores (heavily) used program data
  - Condition codes
    - Store status information about the most recent arithmetic operation
    - Used for conditional branching
  - Memory
    - Byte-addressable array
    - Code, user data, and (some) OS data
    - Includes stack used to support procedures



Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc –O1 p1.c p2.c -o p`
  - Use basic optimizations `(-O1)`
  - 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;
    return;
}
```

Generated x86-64 Assembly

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

- Obtained via command `gcc -O -S sum.c`
- Produces file `sum.s`
- Note: will get very different results on different machines (e.g., Andrew Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.

Assembly Characteristics: Data Types

- 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

Assembly Characteristics: Operations

- Perform arithmetic functions on register or memory data

- Transfer data between a register and memory
  - Load data from memory into register
  - Store register data into memory

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code

Object Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    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
  - 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

Machine Instruction Example

```
*dest = t;
```

- C Code
  - Store value **`t`** where designated by **`dest`**

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

- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

    `t:` Register **`%rax`**

    `dest:` Register **`%rbx`**

    `*dest:` Memory **`M[%rbx]`**

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

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

Disassembling Object Code (53:05)

Disassembled `sumstore`

```
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 Code

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

Disassembled `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
```

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

What Can be Disassembled?

```
% objdump -d WINWORD.EXE
WINWORD.EXE:    file format pei-i386

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

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by Microsoft End User License Agreement

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

Today: Machine Programming I: Basics

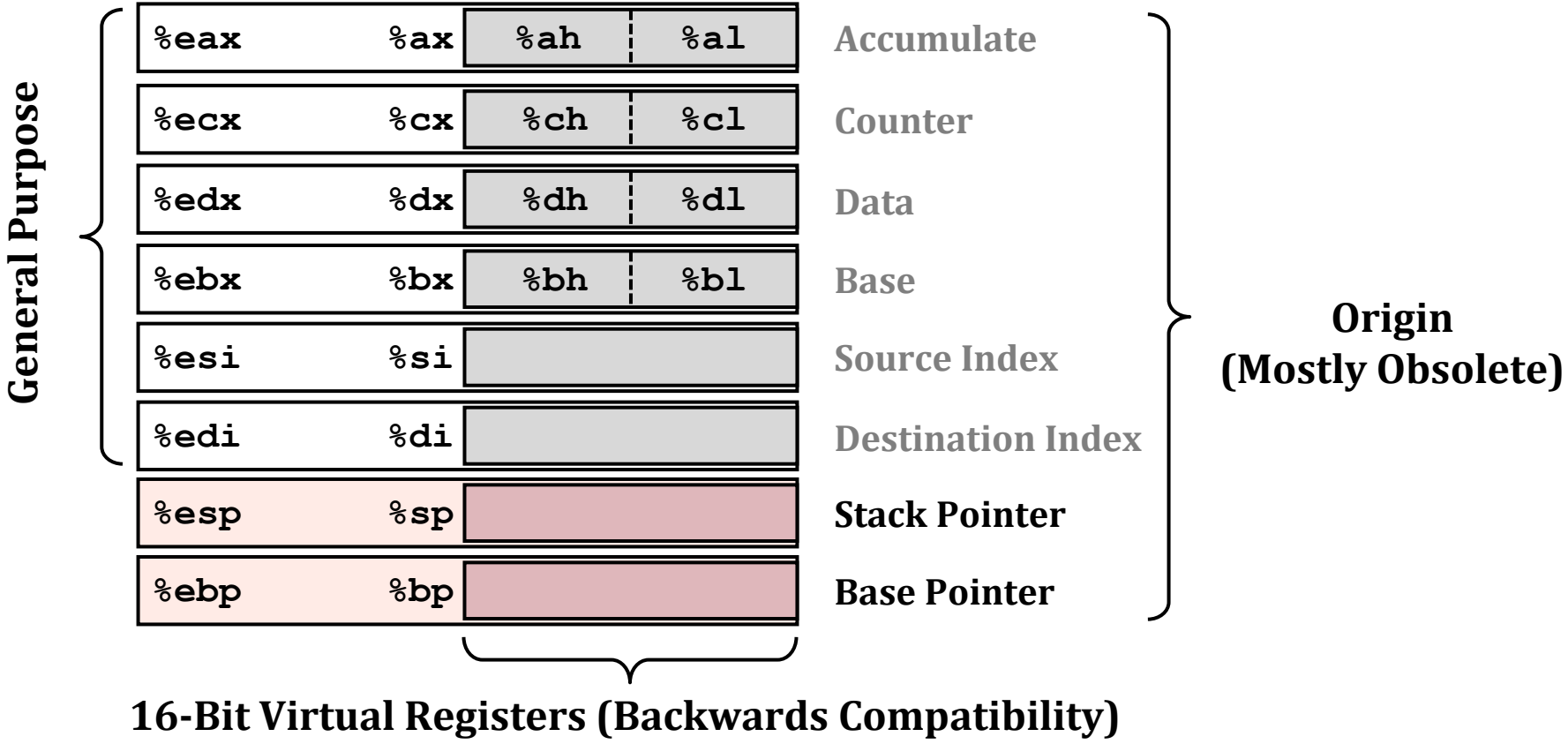
- History of Intel processors and architectures
- C, Assembly, and Machine Code
- Assembly Basics: Registers, Operands, Move
- Arithmetic & Logical Operations

X86-64 Integer Registers

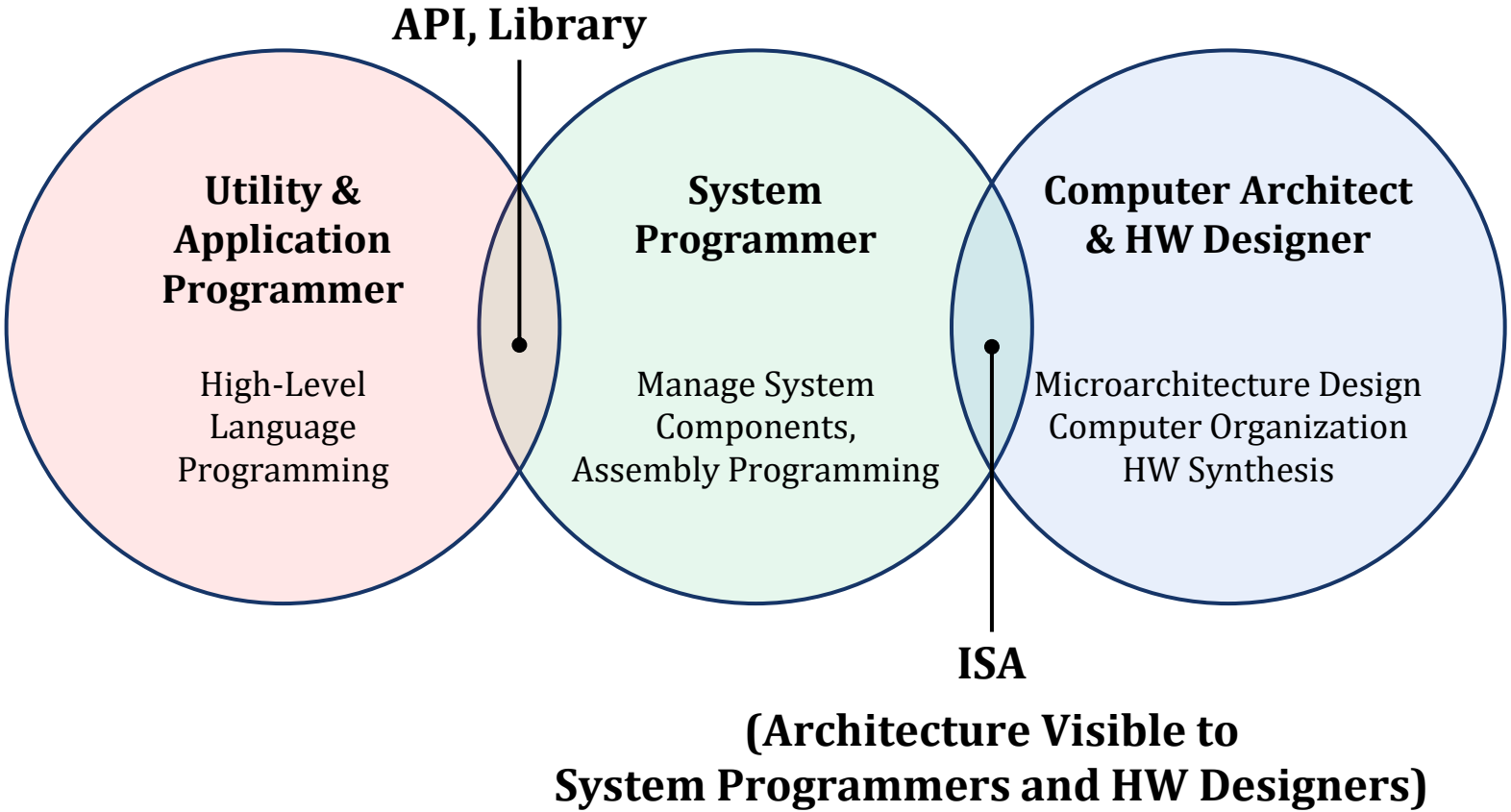
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

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

Integer Registers in IA32



High-level language, Assembler, ISA



High-level & Assembly Comparison

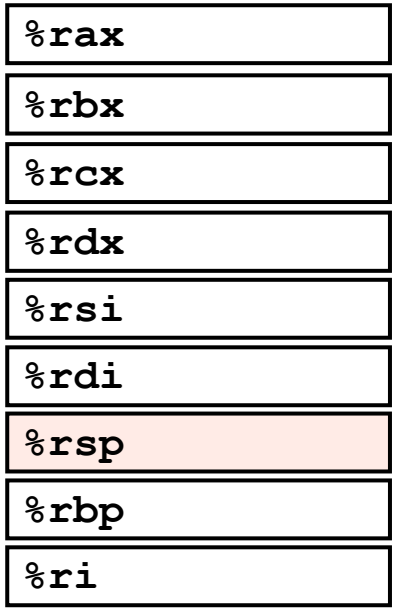
High-level language (C, C++, Java…)	Assembly language
Assignment constructs arithmetic, logical operations, data movement, data conversion	**Data handling constructs** arithmetic, logical operations, data movements, data comparison
Control flow constructs if-then-else, switch, call, return, exception handling	**Control flow constructs** branch (conditional & unconditional), call, return, tabled jump (indirect branch), int (interrupt)
Repetition for, while, do-while	**None** have to implement using data and conditional branch

Moving Data

- Moving Data

  `movq src, 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 address modes



movq Operand Combinations

	Source	Dest	Example	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;`

No memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- Normal: (R) - Mem[Reg[R]]
  - Register R specifies memory address
  - Pointer dereferencing in C

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

- Displacement: D(R) - Mem[Reg[R]+D]
  - Register R specifies start of memory address
  - Constant displacement D specifies offset

```
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;
}
```

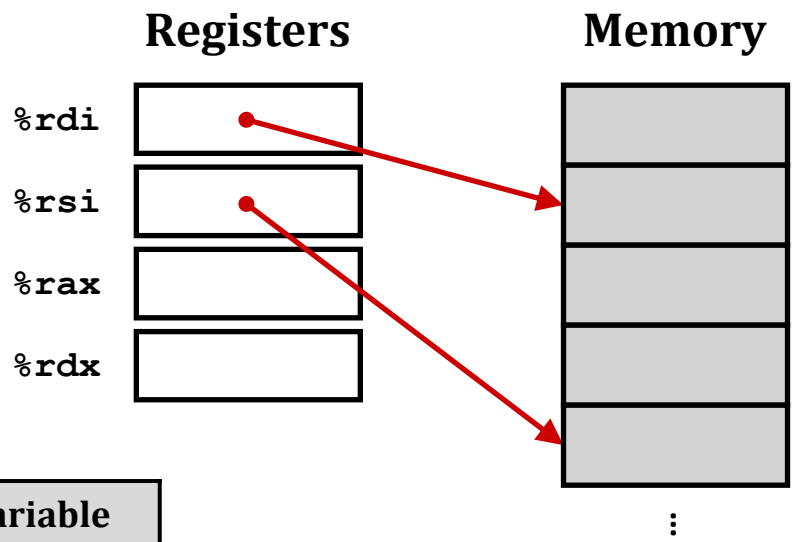
```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

Understanding Swap()

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

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

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

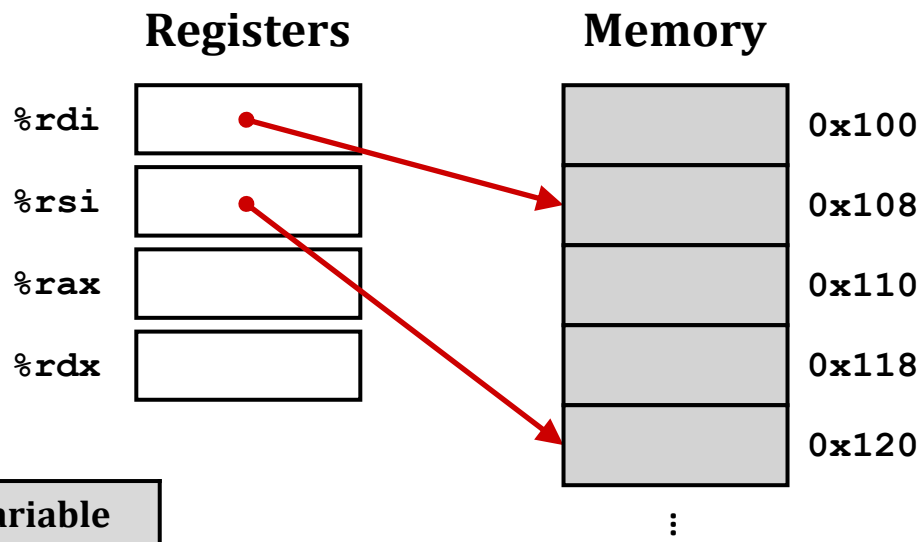


Understanding Swap()

```
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

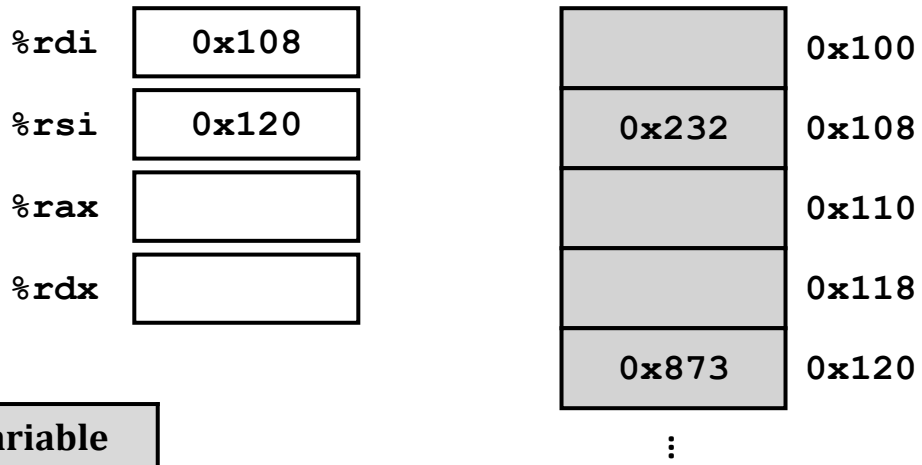


Understanding Swap()

```
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

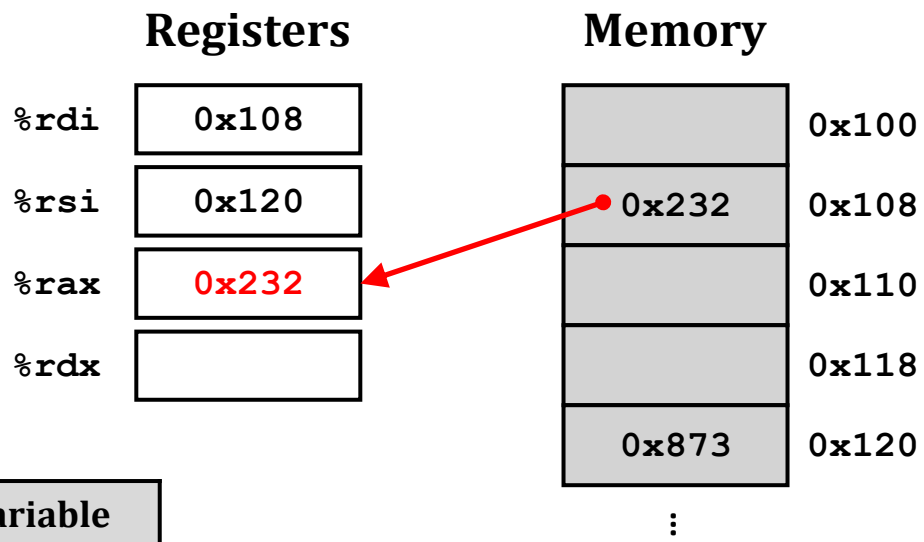


Understanding Swap()

```
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

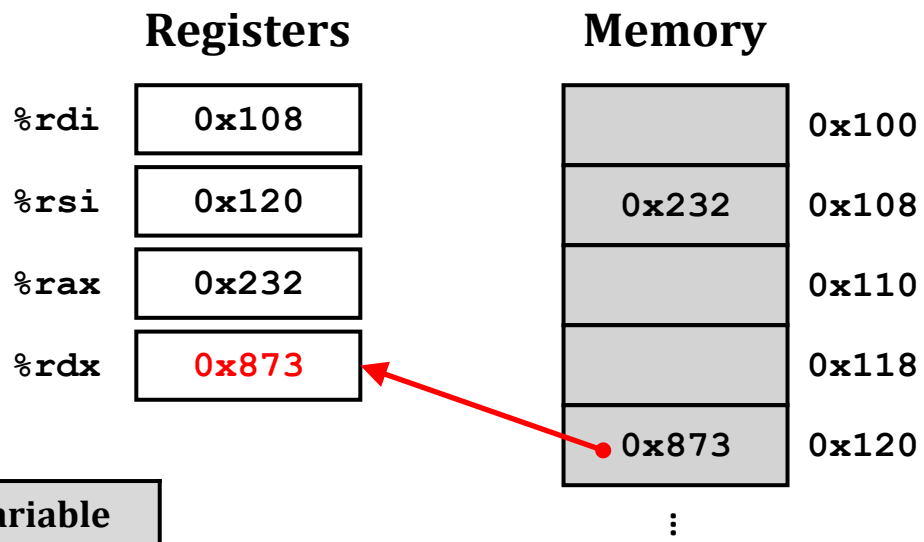


Understanding Swap()

```
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

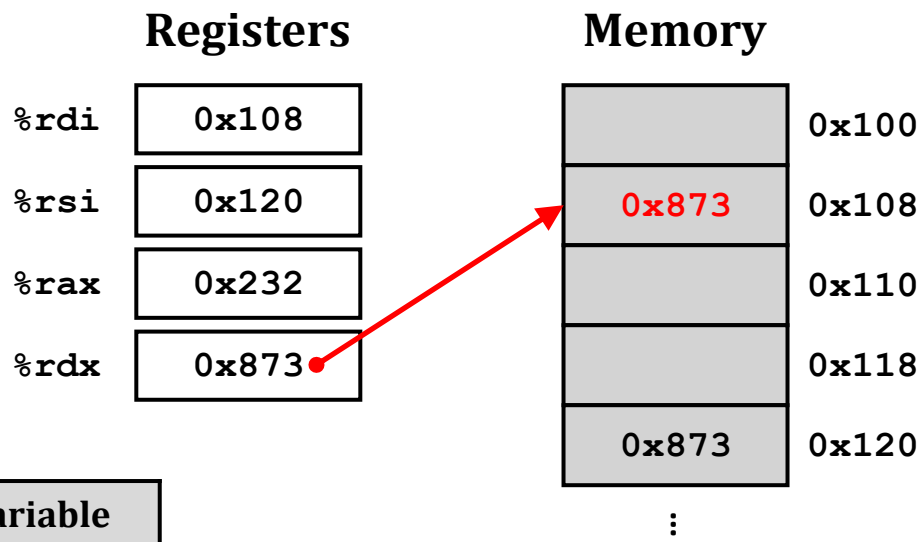


Understanding Swap()

```
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

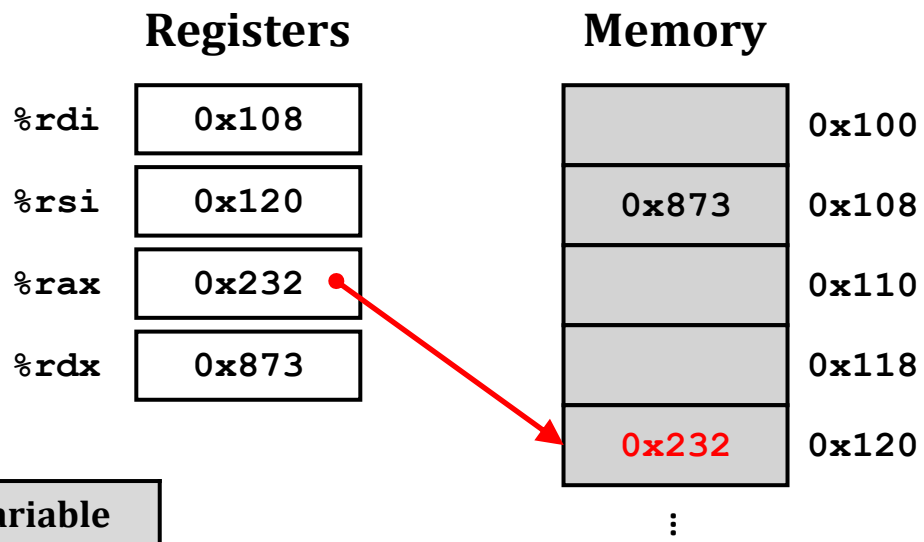


Understanding Swap()

```
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



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

Convert from Assembly to Machine Code

- One assembly code to one machine code
- Consists of
  - Operation code (Opcode)
  - Source operand
  - Destination operand

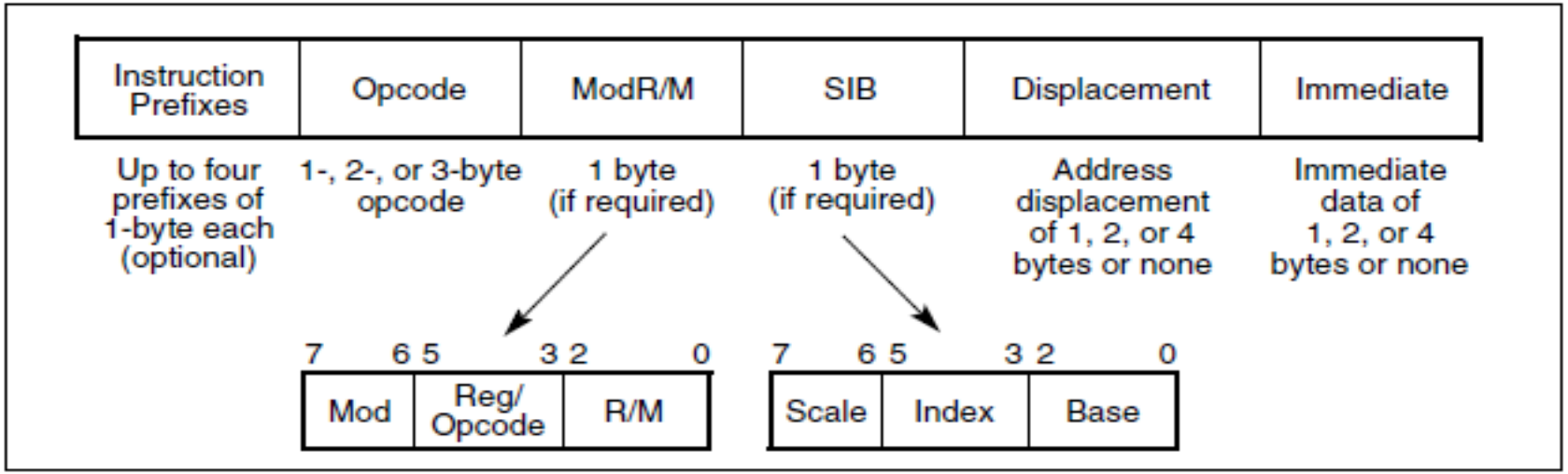


Figure 2-1. IA-32 Instruction Format

Convert from Assembly to Machine lang.

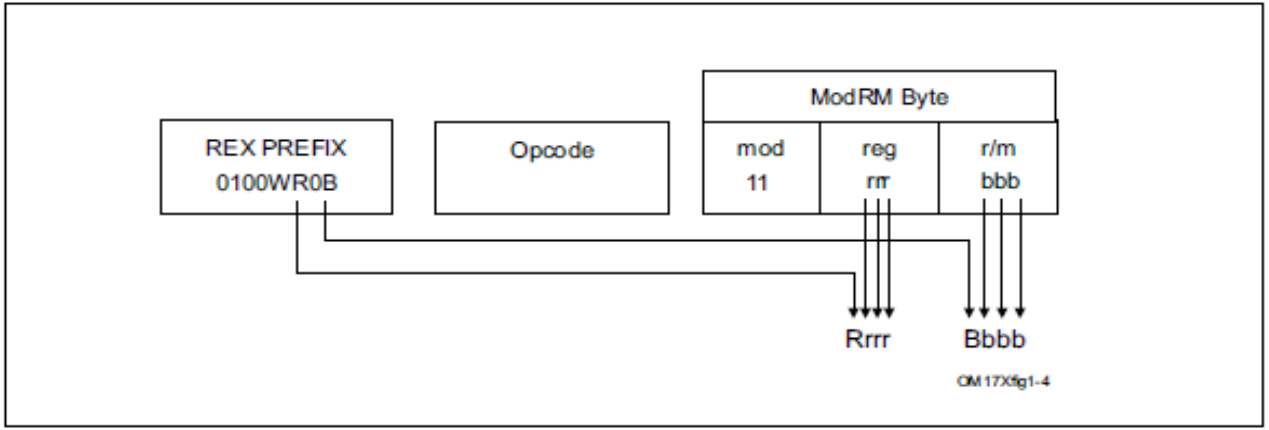


Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

```
movq %rdx, %rbx
```

```
0x40059e:  48 89 d3
```

EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, Assembly, and 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 $x + k \times y$
    - k = 1, 2, 4, or 8
- Example

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

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:

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

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

Some Arithmetic Operations

- One-operand instructions

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

- See book for more instructions

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;
}
```

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

```
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

Memory Operands and `LEA`

- In most instructions, a memory operand accesses memory

Assembly	C Equivalent
`mov 6(%rbx,%rdi,8), %ax`	`ax = *(rbx + rdi*8 + 6)`
`add 6(%rbx,%rdi,8), %ax`	`ax += *(rbx + rdi*8 + 6)`
`xor %ax, 6(%rbx,%rdi,8)`	`*(rbx + rdi*8 + 6) ^= ax`

- **`LEA`** is special: it does not access memory

Assembly	C Equivalent
`lea 6(%rbx,%rdi,8), %rax`	`rax = rbx + rdi*8 + 6`

Why Use `LEA`?

- CPU designers' intention: calculate a pointer to an object
  - An array element, perhaps
  - e.g., to pass just one array element to another function

Assembly	C Equivalent
`lea (%rbx,%rdi,8), %rax`	`rax = &rbx[rdi]`

- Compiler authors like to use it for ordinary arithmetic
  - It can do complex calculations in one instruction
  - It is one of the only three-operand instructions the x86 has
  - It does not touch the condition codes (will come back to this)

Assembly	C Equivalent
`lea (%rbx,%rbx,2), %rax`	`rax = rbx * 3`

Sidebar: Instruction Suffixes

- Most x86 instructions can be written with or without a suffix

```
imul    %rcx,   %rax
imulq   %rcx,   %rax
```

No difference!

- The suffix indicates the operation size
  - `b` = byte, `w` = short (2 bytes), `l` = int (4 bytes), `q` = long (8 bytes)
  - If present, it must match register names

- Assembly output from the compiler (`gcc -S`) usually has suffixes

- Disassembly dumps (`objdump -d, gdb 'disas'`) usually omit suffixes

- Intel's manuals always omit the suffixes

Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - 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

More Information

- Textbook
- Intel processors (Wikipedia)
- Intel microarchitectures

[CSED211] Introduction to Computer Software Systems

Lecture 4: Machine-Level Basic

Prof. Jisung Park

2023.09.18