

**CSCI-UA.0201**

# **Computer Systems Organization**

## **Machine Level – Assembly (x86-64) basics**

Thomas Wies

wies@cs.nyu.edu

<https://cs.nyu.edu/wies>

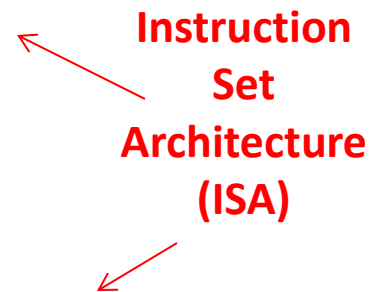
# Intel x86 Processors

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
- Complex instruction set computer (CISC)
  - Many instructions, many formats
  - By contrast, ARM architecture (in most cell phones and tablets) is a reduced instruction set computer (RISC)

# Intel x86 Evolution: Milestones

<i>Name</i>	<i>Transistors</i>	<i>MHz</i>
• 8086 <small>(1978)</small>	29K	5-10
– First 16-bit processor. Basis for IBM PC & DOS		
– 1MB address space		
• 386 <small>(1985)</small>	275K	16-33
– First 32 bit processor , referred to as <b>IA32</b>		
– Capable of running Unix		
• Pentium 4F <small>(2004)</small>	125M	2800-3800
– First 64-bit processor, referred to as <b>x86-64</b>		
• Core i7 <small>(2008)</small>	731M	2667-3333
• Xeon E7 <small>(2011)</small>	2.2B	~2400

Instruction Set Architecture (ISA)



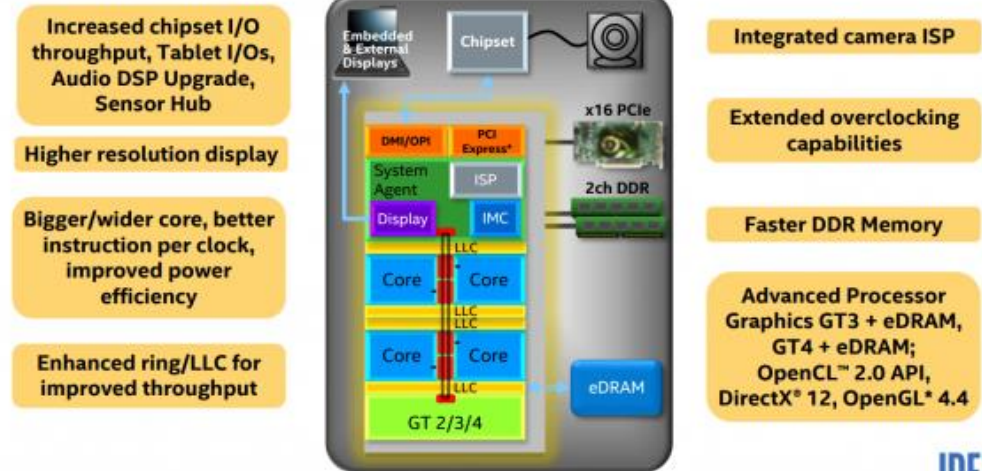
We will cover x86-64.

# Example from 2015

## Intel Skylake

- 4-8 cores
- Integrated graphics
- 2.4-4.0 GHz
- Integrated I/O
- ~35W-95W

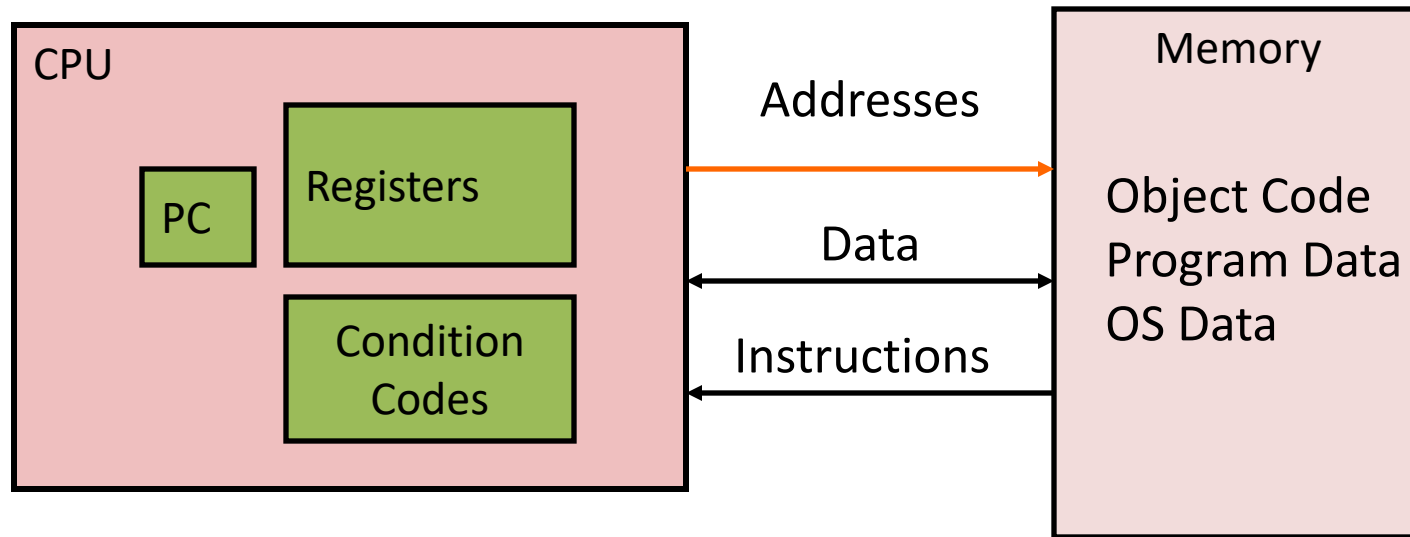
### Intel's Skylake Microarchitecture



7

Intel Next Generation Microarchitecture Code Name Skylake

# Assembly Programmer's View



- Execution context

- **PC**: Program counter

- Address of next instruction
    - Called "RIP" (x86-64)

- **Registers**

- Heavily used program data

- **Condition code registers**

- Store status information about most recent arithmetic or logical operation
    - Used for conditional branching

- **Memory**

- Byte addressable array
    - Code and user data
    - Stack to support procedures

# Assembly Data Types

- "Integer" data of 1, 2, or 4 bytes
  - Represent either data values
  - or addresses
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No arrays or structures

# 3 Kinds of Assembly Operations

- Perform arithmetic on register or memory data
  - Add, subtract, multiply, ...
- 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**

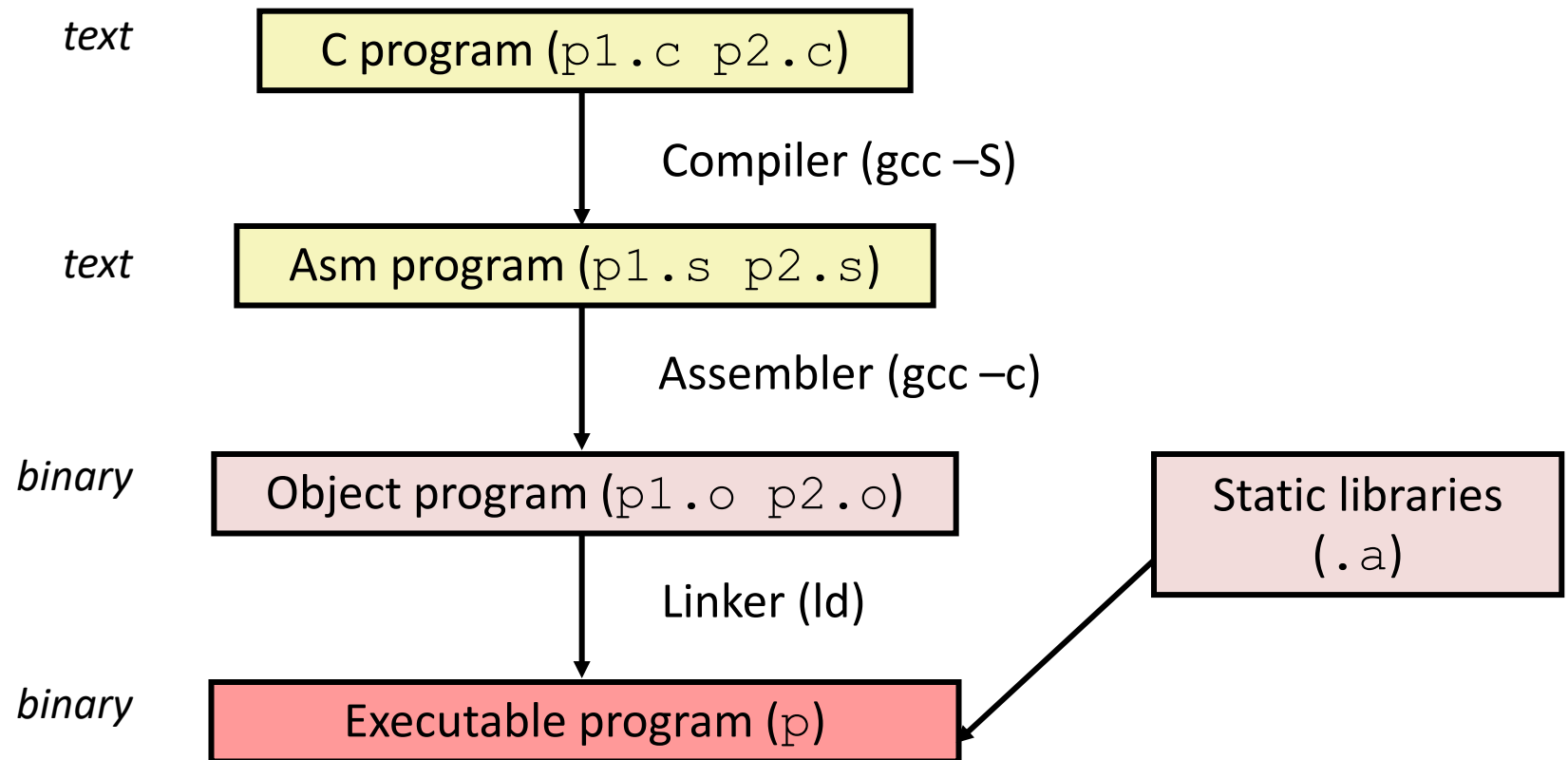
# Turning C into Object Code

– Code in files `p1.c` `p2.c`

Optimization level

Output file is p

– Compile with command: `gcc -Og p1.c p2.c -o p`





# Compiling Into Assembly

C Code (sumstore.c)

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

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

Generated x86-64 Assembly

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

Obtain with command

```
gcc -Og -S sumstore.c
```

Produces file `sumstore.s`

*Warning:* Will get very different results on different machines due to different versions of gcc, different compiler settings, and different hardware architecture.

# Object Code

Code for `sumstore`

`0x0400595:`

`0x53`

`0x48`

`0x89`

`0xd3`

`0xe8`

`0xf2`

`0xff`

`0xff`

`0xff`

`0x48`

`0x89`

`0x03`

`0x5b`

`0xc3`

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - 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
- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address `0x0400595`

# Machine Instruction Example

```
*dest = t;
```

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

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

- C Code
  - (look at `sumstore.c` 2 slides back)
  - 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

## Disassembled

```
000000000400595 <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  
0x48  
0x89  
0x03  
0x5b  
0xc3

Disassembled

Dump of assembler code for function sumstore:

```
0x000000000400595 <+0>:  push    %rbx
0x000000000400596 <+1>:  mov     %rdx,%rbx
0x000000000400599 <+4>:  callq  0x400590 <plus>
0x00000000040059e <+9>:  mov     %rax,(%rbx)
0x0000000004005a1 <+12>: pop     %rbx
0x0000000004005a2 <+13>:  retq
```

- Within gdb Debugger

**gdb** sum

**disassemble sumstore**

– Disassemble procedure

**x/14xb sumstore**

– Examine the 14 bytes starting at sumstore

# x86-64 Integer Registers

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

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

# Some History: Integer Registers (IA32)

			Origin (mostly obsolete)	
general purpose	%eax	%ax	%ah   %al	accumulate
	%ecx	%cx	%ch   %cl	counter
	%edx	%dx	%dh   %dl	data
	%ebx	%bx	%bh   %bl	base
	%esi	%si		source index
	%edi	%di		dest. index
	%esp	%sp		stack pointer
	%ebp	%bp		base pointer

16-bit virtual registers  
(backwards compatibility)

# Moving Data



# Moving Data

- Moving Data

`movq Source, Dest`

- Operand Types

– **Immediate:** Constant integer data

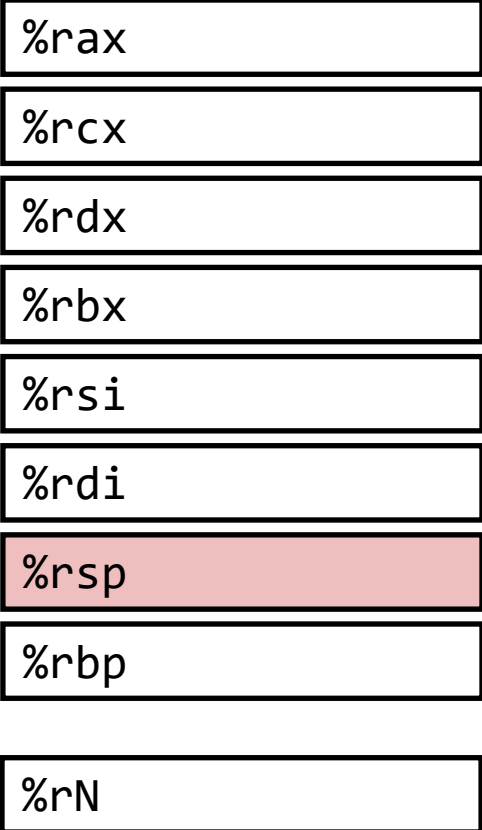
- Example: `$0x400`, `$-533`
- Like C constant, but prefixed with ``$'`

– **Register:** One of 16 integer registers

- Example: `%rax`, `%r13`
- But `%rsp` reserved for special use
- Others have special uses for particular instructions (later on that)

– **Memory:** 8 consecutive bytes of memory at address given by register

- Simplest example: `(%rax)`
- We will see various other "address modes" later.



# movq Operand Combinations

	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;

*No memory-to-memory instruction*

# movq

C Declaration	Intel Data Type	Assembly code suffix	Size (bytes)
Char	Byte	b	1
Short	Word	w	2
Int	Double Word	l	4
Long	Quad Word	q	8
Pointer	Quad Word	q	8

# Special Type of mov

- `movz S,R`  $\rightarrow$  `R = ZeroExtend(S)`
  - `movzbw` (zero extend byte to word)
  - `movzbl` (zero extend byte to double word)
  - `movzbq` (zero extend byte to quad word)
  - `movzwl` (zero extend word to double word)
  - `movzwq` (zero extend word to quad word)
- `movs S,R`  $\rightarrow$  `R = SignExtend(S)`
  - `movsbw` (sign extend byte to word)
  - `movsbl` (sign extend byte to double word)
  - `movsbq` (sign extend byte to quad word)
  - `movswl` (sign extend word to double word)
  - `movswq` (sign extend word to quad word)
  - `movslq` (sign extend double word to quad word)
- **S**: memory or register    **R**: register