

00000000	00000000	call	2084
00000001	00000001	jmp	15CA
00000002	00000002	call	17B4
00000003	00000003	mov	si, 7160
00000004	00000004	xor	di, di
00000005	00000005	mov	es, [7606]
00000006	00000006	mov	bx, 800F
00000007	00000007	xor	cx, cx
00000008	00000008	mov	bp, 0001
00000009	00000009	mov	dx, ds

# 目标指令集架构及汇编语言

黄波

bhuang@dase.ecnu.edu.cn

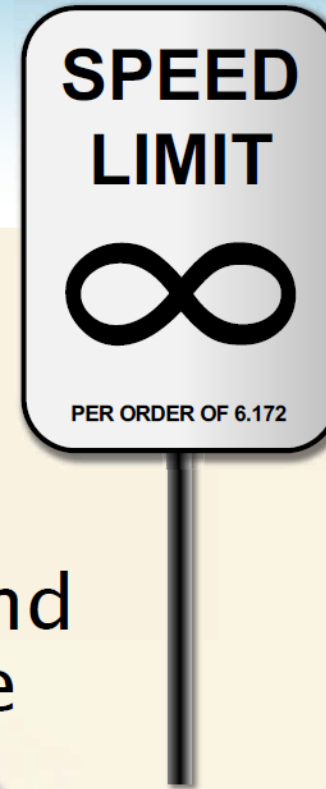
# 本次课的关注点

Scale up  
全栈思维



- **指令集架构** (Instruction Set Architecture, 简称ISA) 是计算机抽象模型的重要组成部分。作为硬件和软件之间的接口, 指令集架构定义了CPU如何被软件控制, 并对CPU的能力以及这些能力的实现方式进行了限定;
- 指令集架构定义了CPU支持的数据类型、寄存器、硬件管理内存的方式、虚拟内存等其它重要特性、CPU能够执行的指令、输入输出模型等.
- **理解指令集架构及汇编语言是编译器开发调试的基础**
- 理解目标平台的汇编指令集并进一步理解编译器如何针对目标指令集生成优化代码能够帮助程序开发人员写出更加高效的代码, 同时也能帮助程序开发人员更好地理解编译器的输出从而**有助于程序调试**。

6.172  
Performance  
Engineering  
of Software  
Systems



## LECTURE 4

# Assembly Language and Computer Architecture

Charles E. Leiserson

© 2008–2018 by the MIT 6.172 Lecturers



[https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6\\_172F18\\_lec4.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6_172F18_lec4.pdf)

# 从源代码到程序执行

## Source code fib.c

```
int64_t fib(int64_t n) {  
    if (n < 2) return n;  
    return (fib(n-1) + fib(n-2));  
}
```

```
$ clang fib.c -o fib
```

## Compilation

four stages {  
 Preprocessing  
 Compiling  
 Assembling  
 Linking

## Machine code fib

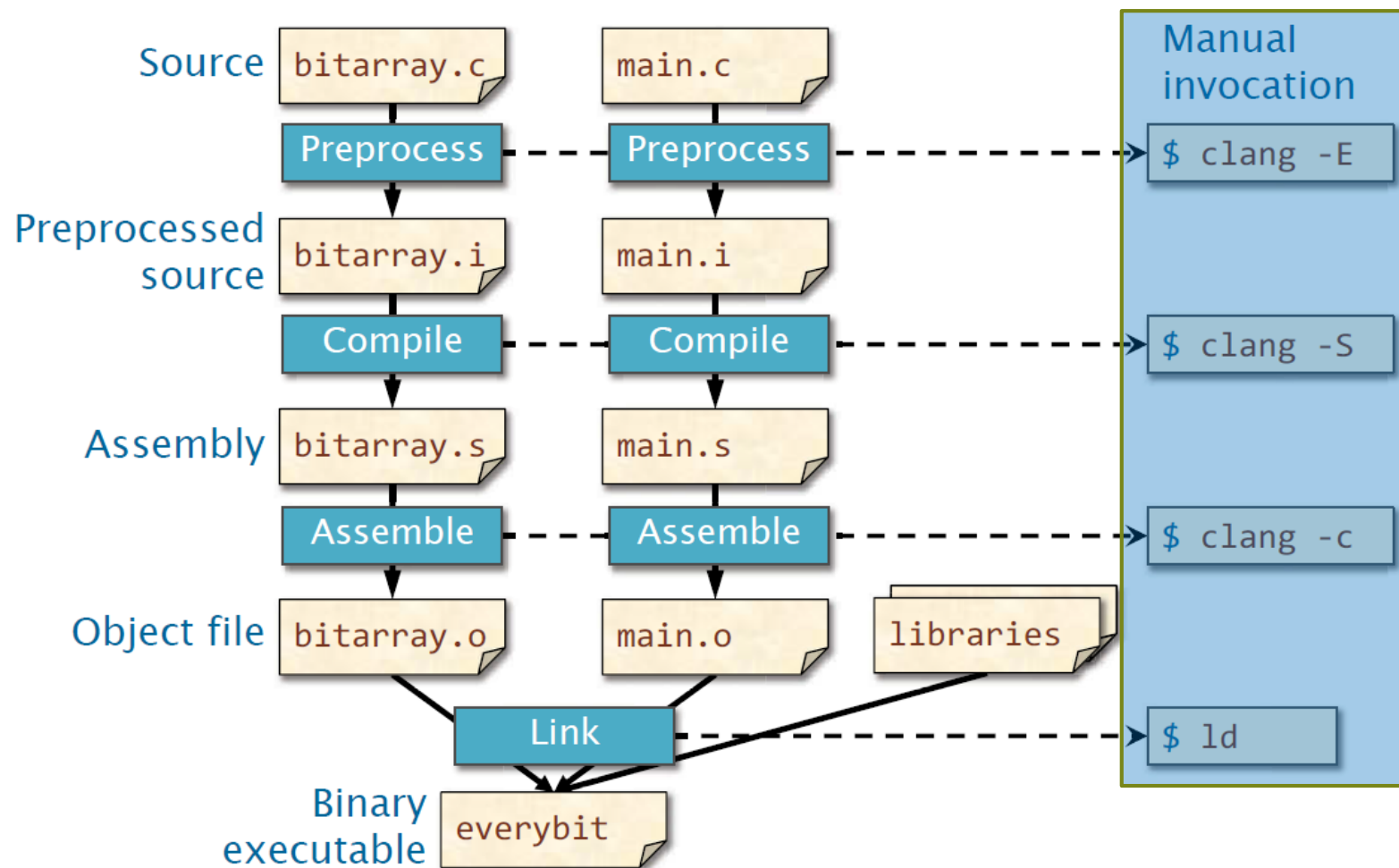
```
01010101 01001000 10001001  
11100101 01010011 01001000  
10000011 11101100 00001000  
10001001 01111101 11110100  
10000011 01111101 11110100  
00000001 01111111 00001000  
10001011 01000101 11110100  
10001001 01000101 11110000  
11101011 00011101 10001011  
01000101 11110100 10001101  
01111000 11111111 11101000  
11011011 11111111 11111111  
11111111 10001001 11000011  
10001011 01000101 11110100  
10001101 01111000 11111110  
11101000 11001110 11111111  
11111111 11111111 00000001  
11000011 10001001 01011101  
11110000 10001011 01000101  
11110000 01001000 10000011  
11000100 00001000 01011011  
11001001 11000011
```

## Hardware interpretation

```
$ ./fib
```

## Execution

# “编译”过程的4个阶段



Q: 预处理阶段主要做哪些事情?

# 从源文件到汇编文件

## Source code fib.c

```
int64_t fib(int64_t n) {  
    if (n < 2) return n;  
    return (fib(n-1) + fib(n-2));  
}
```

```
$ clang -O3 fib.c -S
```

## Assembly Code fib.s

```
.globl fib  
fib:  # @fib  
    pushq  %r14  
    pushq  %rbx  
    pushq  %rax  
    xorl   %r14d, %r14d  
    cmpq   $2, %rdi  
    jl     .LBB0_3  
    movq   %rdi, %rbx  
.LBB0_2:  
    leaq   -1(%rbx), %rdi  
    callq  fib  
    leaq   -2(%rbx), %rdi  
    addq   %rax, %r14  
    cmpq   $3, %rbx  
    movq   %rdi, %rbx  
    jg     .LBB0_2  
.LBB0_3:  
    addq   %rdi, %r14  
    movq   %r14, %rax  
    addq   $8, %rsp  
    popq   %rbx  
    popq   %r14  
    retq
```

汇编语言提供了一种方便易读的  
对机器码的符号表示

<https://sourceware.org/binutils/docs/as/index.html>

<https://www.felixcloutier.com/x86/>



# 从汇编代码到可执行文件

## Assembly code fib.s

```
_fib:      .p2align    4, 0x90
          ## @fib
          pushq      %rbp
          movq       %rsp, %rbp
          pushq      %r14
          pushq      %rbx
          movq       %rdi, %rbx
          cmpq       $2, %rbx
          jge        LBB0_1
          movq       %rbx, %rax
          jmp        LBB0_3

LBB0_1:
          leaq       -1(%rbx), %rdi
          callq      _fib
          movq       %rax, %r14
          addq       $-2, %rbx
          movq       %rbx, %rdi
          callq      _fib
          addq       %r14, %rax

LBB0_3:
          popq       %rbx
          popq       %r14
          popq       %rbp
          retq
```

## Assembling

```
$ clang fib.s -o fib.o
```

## Machine code

```
01010101 01001000
10001001 11100101
01010011 01001000
10000011 11101100
00001000 10001001
01111101 11110100
10000011 01111101
11110100 00000001
01111111 00001000
10001011 01000101
11110100 10001001
01000101 11110000
11101011 00011101
10001011 01000101
11110100 10001101
01111000 11111111
11101000 11011011
11111111 11111111
11111111 10001001
11000011 10001011
01000101 11110100
```

可以对fib.s进行编辑并用clang来做汇编

# 反汇编

```
int64_t fib(int64_t n) {  
    if (n < 2)  
        return n;  
    return (fib(n-1) + fib(n-2));  
}
```

clang -g fib.c -o fib

fib

\$objdump -S fib

Q: 为什么“e8 b3 ff ff ff”是调用fib?

```
int64_t fib(int64_t n) {  
401110: 55                push %rbp  
401111: 48 89 e5          mov %rsp,%rbp  
401114: 48 83 ec 20       sub $0x20,%rsp  
401118: 48 89 7d f0       mov %rdi,-0x10(%rbp)  
    if (n < 2)  
40111c: 48 83 7d f0 02    cmpq $0x2,-0x10(%rbp)  
401121: 0f 8d 0d 00 00 00 jge 401134 <fib+0x24>  
        return n;  
401127: 48 8b 45 f0       mov -0x10(%rbp),%rax  
40112b: 48 89 45 f8       mov %rax,-0x8(%rbp)  
40112f: e9 34 00 00 00    jmpq 401168 <fib+0x58>  
        return (fib(n-1) + fib(n-2));  
401134: 48 8b 45 f0       mov -0x10(%rbp),%rax  
401138: 48 2d 01 00 00 00 sub $0x1,%rax  
40113e: 48 89 c7          mov %rax,%rdi  
401141: e8 ca ff ff ff    callq 401110 <fib>  
401146: 48 8b 4d f0       mov -0x10(%rbp),%rcx  
40114a: 48 81 e9 02 00 00 00 sub $0x2,%rcx  
401151: 48 89 cf          mov %rcx,%rdi  
401154: 48 89 45 e8       mov %rax,-0x18(%rbp)  
401158: e8 b3 ff ff ff    callq 401110 <fib>  
40115d: 48 8b 4d e8       mov -0x18(%rbp),%rcx  
401161: 48 01 c1          add %rax,%rcx  
401164: 48 89 4d f8       mov %rcx,-0x8(%rbp)  
}  
401168: 48 8b 45 f8       mov -0x8(%rbp),%rax  
40116c: 48 83 c4 20       add $0x20,%rsp  
401170: 5d                pop %rbp  
401171: c3                retq  
401172: 66 2e 0f 1f 84 00 00 nopw %cs:0x0(%rax,%rax,1)  
401179: 00 00 00
```



# 期望

汇编语言涉及到指令集架构的很多知识，比较繁琐与复杂。从编译优化的角度上来说，希望同学们至少需要掌握以下几点：

- 理解编译器如何把C的语言结构转换成语义等价的X86-64的汇编指令；
- 在指令集架构手册的帮助下，掌握熟练读懂X86-64汇编程序的能力；
- 理解常见汇编指令模式对程序性能可能产生的影响；
- 掌握对编译器生成的汇编程序进行简单修改的能力；
- 理解跟汇编指令对应的编译器内部函数（intrinsic function），并能够在C程序中通过调用编译器内部函数来利用指令集架构中的一些汇编指令，从而达到实现特殊功能或者提升程序性能的目的；
- 在一些特殊需要的情况下，知道如何从头开始编写汇编代码。

# 指令集架构

指令集架构规定了汇编语言的语法和语义，常见的指令集架构有：

- Intel指令集架构 (X86、X86-64等)
- ARM指令集架构 (ARMv7、AArch64等)
- RISV指令集架构
- MIPS指令集架构
- ...

某个指令集架构是M位的一般有如下几层含义：

- CPU可以处理的数据的最大位数是M位
- 主要的通用寄存器和PC(Program Counter)寄存器的宽度都是M位宽
- 地址（指针）的宽度也是M位，寻址的地址空间为 $[0, 2^M]$
- *这里不考虑向量化指令中对多个数据并行处理的情况*

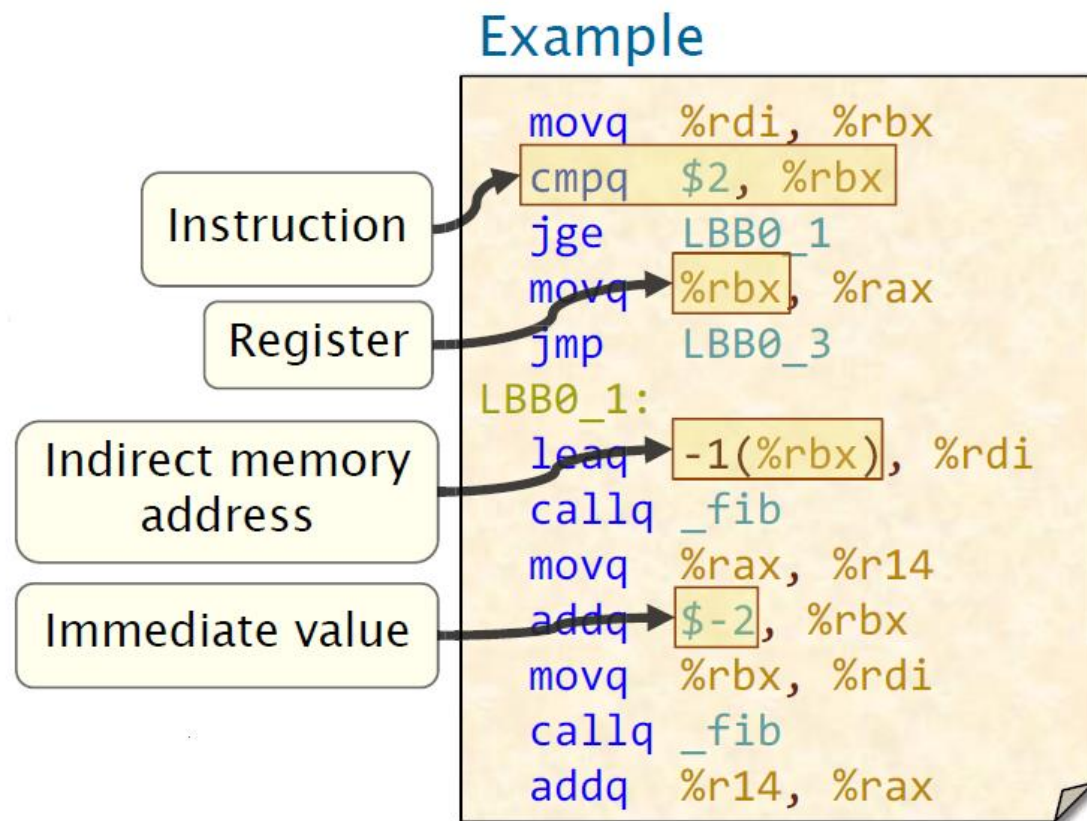
# X86-64指令架构

## 代表厂家:

- Intel、AMD、威盛、海光、兆芯

任何一门汇编语言，都必须明确定义这个汇编能够支持的

- 数据类型
- 寄存器
- 指令
- 内存寻址方式



# X86-64 数据类型

C declaration	C constant	x86-64 size (bytes)	Assembly suffix	x86-64 data type
char	'c'	1	b	Byte
short	172	2	w	Word
int	172	4	l or d	Double word
unsigned int	172U	4	l or d	Double word
long	172L	8	q	Quad word
unsigned long	172UL	8	q	Quad word
char *	"6.172"	8	q	Quad word
float	6.172F	4	s	Single precision
double	6.172	8	d	Double precision
long double	6.172L	16(10)	t	Extended precision

# x86-64 寄存器

Number	Width (bits)	Name(s)	Purpose
16	64	(many)	General-purpose registers
6	16	%ss,%[c-g]s	Segment registers
1	64	RFLAGS	Flags register
1	64	%rip	Instruction pointer register
7	64	%cr[0-4,8], %xcr0	Control registers
8	64	%mm[0-7]	MMX registers
1	32	mxcsr	SSE2 control register
16	128	%xmm[0-15]	XMM registers (for SSE)
	256	%ymm[0-15]	YMM registers (for AVX)
8	80	%st([0-7])	x87 FPU data registers
1	16	x87 CW	x87 FPU control register
1	16	x87 SW	x87 FPU status register
1	48		x87 FPU instruction pointer register
1	48		x87 FPU data operand pointer register
1	16		x87 FPU tag register
1	11		x87 FPU opcode register

lscpu结果中的flags

```
fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush
mmx fxsr sse sse2 ss ht syscall nx pdpe1
gb rdtscp lm constant_tsc rep_good nopl
xtopology cpuid pni pclmulqdq ssse3 fma
cx16 pcid sse4_1 sse4_2 movbe popcnt aes
xsave avx flbc rdrand hypervisor lahf_l
m abm 3dnowprefetch invpcid_single ssbd
ibrs ibpb stibp ibrs_enhanced fsgsbase b
mil avx2 smep bmi2 erms invpcid rdseed a
dx smap clflushopt xsaveopt xsavec xgetb
v1 xsaves flush_lld arch_capabilities
```

← AVX-512 is not mentioned here!

# X86-64 通用寄存器

64-bit name	32-bit name	16-bit name	8-bit name(s)
%rax	%eax	%ax	%ah, %al
%rbx	%ebx	%bx	%bh, %bl
%rcx	%ecx	%cx	%ch, %cl
%rdx	%edx	%dx	%dh, %dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b



# x86-64 寄存器别名

对于某个64位通用寄存器，这个通用寄存器和它的别名寄存器之间其实是有重叠的

## General-purpose register layout

%rax							
				%eax			
						%ax	
						%ah	%al
Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0

只有%rax、%rbx、%rcx和%rdx拥有能够单独访问各寄存器之次低位字节（Byte 1）的别名寄存器

# RFLAG (64位 EFLAG) 寄存器

Bit(s)	Abbreviation	Description
0	CF	Carry
1		<i>Reserved</i>
2	PF	Parity
3		<i>Reserved</i>
4	AF	Adjust
5		<i>Reserved</i>
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12-63		<i>System flags or reserved</i>

最近的算术逻辑运算产生了进位或借位

最近的算术逻辑运算结果为零

最近的算术逻辑运算产生的结果的符号位为1

最近的算术运算产生了溢出

# RFLAG (64位 EFLAG) 寄存器

Bit(s)	Abbreviation	Description
0	CF	Carry
1		<i>Reserved</i>
2	PF	Parity
3		<i>Reserved</i>
4	AF	Adjust
5		<i>Reserved</i>
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12-63		<i>System flags or reserved</i>

算术和逻辑运算更新RFLAGS寄存器中各标志位的值

Decrement `%rbx`, and set **ZF** if the result is 0.

Example:

```
decq %rbx  
jne .LBB7_1
```

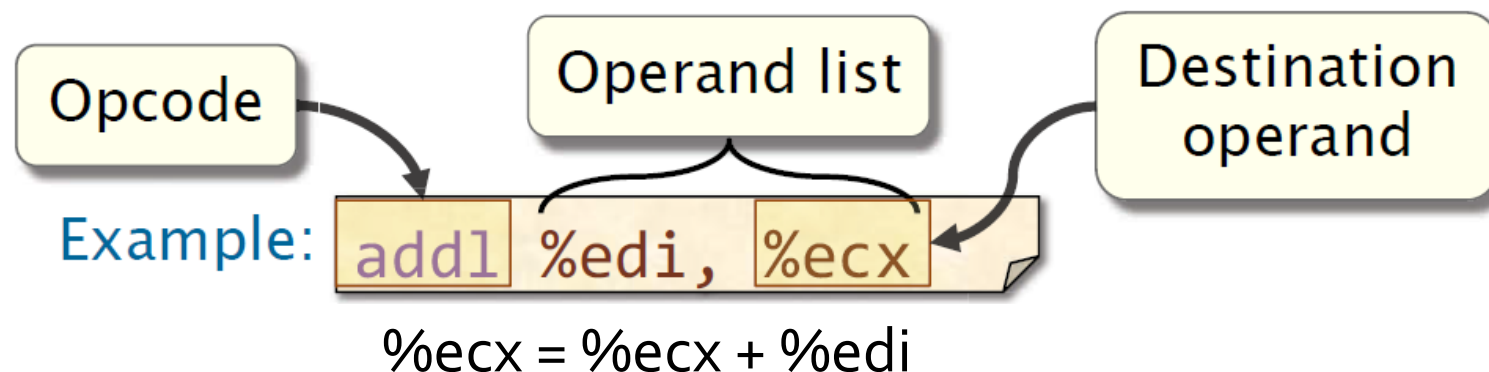
Jump to label `.LBB7_1` if **ZF** is not set.

正确理解RFLAGS标志寄存器以及各指令对RFLAGS标志寄存器内各个标志位的影响是编译器生成有效准确的控制逻辑的基础。

# x86-64指令格式

指令格式： <操作码> <操作数列表>

- 操作码是一个短助记符，表明指令类型
- 操作数列表可以包含最多3个操作数，各操作数值间以逗号分割，也可以为空（即不包含操作数）
- 通常情况下，操作数列表中所有的操作数都是源操作数，其中有一个源操作数也有可能同时是目的操作数



Q: RISC ISA和CISC ISA的区别?

# AT&T汇编格式 vs. Intel汇编格式

“<op> A, B” 到底是什么意思？

AT&T Syntax $B \leftarrow B <op> A$	Intel Syntax $A \leftarrow A <op> B$
<code>movl \$1, %eax</code>	<code>mov eax, 1</code>
<code>addl (%ebx,%ecx,0x2), %eax</code>	<code>add eax, [ebx+ecx*2h]</code>
<code>subq 0x20(%rbx), %rax</code>	<code>sub rax, [rbx+20h]</code>

很多开源工具（比如clang、objdump、perf等）基本上都沿用AT&T格式

Intel的相关文档基本都采用Intel的汇编指令语法格式

# 常见的X86-64汇编指令操作码

Type of operation		Examples
Data movement	Move	mov
	Conditional move	cmov
	Sign or zero extension	movs, movz
	Stack	push, pop
Arithmetic and logic	Integer arithmetic	add, sub, mul, imul, div, idiv, lea, sal, sar, shl, shr, rol, ror, inc, dec, neg
	Binary logic	and, or, xor, not
	Boolean logic	test, cmp
Control transfer	Unconditional jump	jmp
	Conditional jumps	j<condition>
	Subroutines	call, ret



# 条件操作指令

条件操作指令（跳转指令和条件数据传送指令）通常用一到两个字符作为指令后缀来表明跳转的条件码（Condition Code）：

```
cmpq $4096, %r14  
jne.LBB1_1
```

当且仅当寄存器%r14的值不等于4096时  
才会跳转到标号.LBB1\_1上的代码

# 条件码

条件码	条件码的含义	如何检查RFLAGS 中的标志位
a	if above	CF = 0 and ZF = 0
ae	if above or equal	CF = 0
c	on carry	CF = 1
e	if equal	ZF = 1
ge	if greater or equal	SF = OF
ne	if not equal	ZF = 0
o	on overflow	OF = 1
z	if zero	ZF = 1

**Q:** 为什么条件码e或者ne需要检查Z (Zero) 标志位?

**答案:** 处理器硬件一般都用减法来比较两个整数操作数

**Q:** jae和jge有什么区别?

# 操作码后缀

操作码后缀除了可以表示条件码之外，还可以表示指令操作的数据类型

- 对于数据移动指令、算术运算指令和逻辑运算指令，这些指令操作码后的单个字符后缀表明的就是数据类型
- 如果这些指令操作码后面没有跟后缀，通常可以从操作数寄存器来推断操作的数据类型

## Example

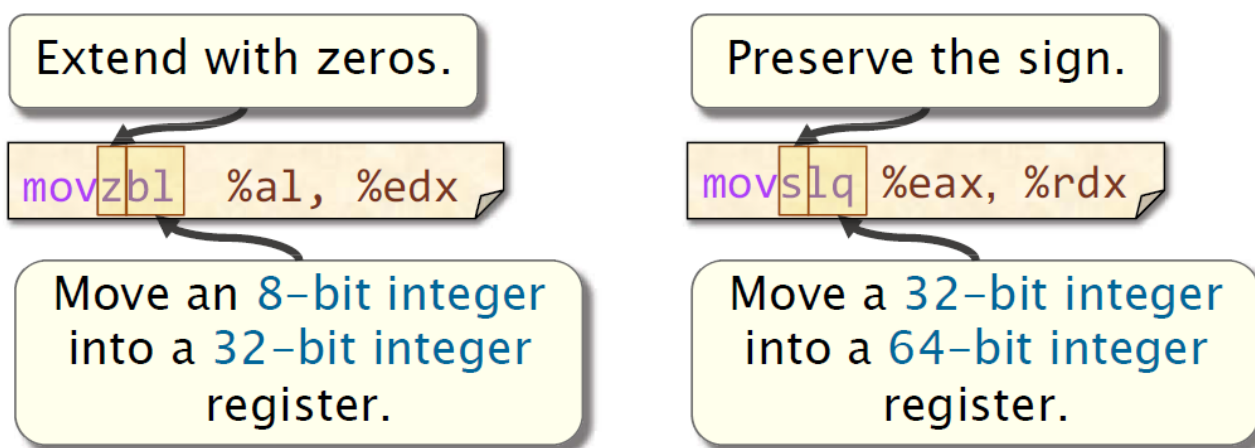
`movq -16(%rbp), %rax`

Moving a 64-bit integer.

# 标明符号扩展或零扩展的操作码后缀

对于符号扩展（Sign-extension）指令或者零扩展（Zero-extension）指令，它们的操作码需要用到两个数据类型后缀，分别表示源操作数的数据类型以及扩展结果的数据类型

Examples:



在X86-64的指令集架构中，32位操作的结果会被缺省零扩展成64位，这个跟8位操作及16位操作的缺省符号扩展是不一样的！


# x86-64 寻址方式

操作数的寻址方式确定了一条汇编指令该如何读取指令最终操作的值。在X86-64指令集架构中，一条汇编指令最多只允许一个操作数通过内存地址来访问取值，常见的寻址方式有两大类：

- **直接寻址 (Direct Addressing Modes)**

- 立即数(Immediate), e.g. "movq \$172, %rdi"
- 寄存器(Register), e.g. "movq %rcx, %rdi"
- 直接内存访问(Direct memory), e.g. "movq 0x172, %rdi"

- **间接寻址(Indirect Addressing Modes)**

- 寄存器间接寻址(Register indirect), e.g. "movq (%rax), %rdi"
- 寄存器索引寻址(Register indexed), e.g. "movq 172(%rax), %rdi"
- 指令指针相对寻址(Instruction-pointer relative), e.g. "movq 172(%rip), %rdi"
- 基索引表位移寻址(Base indexed scale displacement), 

# 基索引表位移寻址

这种寻址方式是X86-64指令集架构间接寻址的最通用方式，常见表达如下：

$\text{Displacement}(\%Base\_GPR, \%Index\_GPR, Scale)$

其中：

- Displacement和Scale都是常量
- Scale的值可以是1,2,4或者8
- 假设通用寄存器%Base\_GPR的值为Base，通用寄存器%Index\_GPR的值为Index，则上面这个寻址方式所表达的地址是： $Base + Index * Scale + Displacement$
- 如果Displacement缺失，相当于Displacement等于0
- 如果%Index\_GPR缺失，相当于Index等于0；
- 如果Scale缺失，相当于Sale等于1

以指令“`movq 172(%rdi, %rdx, 8), %rax`”为例，源操作数是地址  
“ $(\%rdi) + (\%rdx) * 8 + 172$ ”上存储的64位整数

Q: 这种寻址方式适合对C语言中的哪类数据结构进行访问？



# 跳转指令的寻址方式

**跳转指令：**无条件跳转指令、条件跳转指令

**直接跳转指令：**跳转目标作为指令的一部分编码给出，表示跳转目标的方式可以用绝对地址（立即数地址），也可以用相对地址（相对于当前指令指针的偏移量）

```

cmpq    $2, %rbx
jge     LBB0_1
movq    %rbx, %rax
jmp     LBB0_3
LBB0_1:
leaq    -1(%rbx), %rdi

```

```

a: 48 83 fb 02    cmpq    $2, %rbx
e: 7d 05          jge     5 <_fib+0x15>
; }
10: 48 89 d8      movq    %rbx, %rax
13: eb 1b        jmp     27 <_fib+0x30>
; return (fib(n-1) + fib(n-2));
15: 48 8d 7b ff    leaq    -1(%rbx), %rdi

```

**间接跳转指令：**可以用寄存器中的值作为跳转目标（比如“`jmp *%rax`”指令），也可以用寄存器指向的内存中的值为跳转目标（比如“`jz *(%rax)`”指令）

# 汇编技巧：寄存器初始化

Q: 这条汇编指令在干啥？

```
xor %rax, %rax
```

答案：实现对寄存器%rax进行赋零初始化

# 汇编技巧：判断寄存器的值是否为零

“**test** A, B,” 计算A与B的逻辑与，然后把结果丢弃，但保留RFLAGS的值改变

Q：这些**test**汇编指令在干啥？

```
test %rcx, %rcx  
je 400c0a <mm+0xda>
```

```
test %rax, %rax  
cmovne %rax, %r8
```

答案：检查寄存器内的值是否为0.

Bit	Abbreviation	Description
0	CF	Carry
2	PF	Parity
4	AF	Adjust
6	ZF	Zero
7	SF	Sign
11	OF	Overflow

# 汇编技巧：空指令

- 在X86-64汇编语言中，有一个特殊的指令叫做**空指令**（汇编操作码为**nop**，No Operation的缩写），也叫无操作指令。这条指令的主要作用是通过在汇编程序中填充nop指令来达到指令对齐的目的，同时清除由上一个算术逻辑指令执行后在RFLAGS中设置的标志位。
- 为了达到内存对齐的目的，编译器有时会生成一些“神奇”的“空指令代码串”

```

100000ae5: 31 ed          xor    %ebp,%ebp
100000ae7: 31 d2          xor    %edx,%edx
100000ae9: 49 89 c4       mov    %rax,%r12
100000aec: b8 00 ca 9a 3b mov    $0x3b9aca00,%eax
100000af1: 66 66 66 66 66 66 2e data16 data16 data16 data16 data16 nopw %cs:0x0(%rax,%rax,1)
100000af8: 0f 1f 84 00 00 00 00
100000aff: 00
100000b00: 8d 75 01       lea    0x1(%rbp),%esi

```

使得后面的“`lea 0x1(%rbp),%esi`”指令的起始地址满足了16字节对齐的要求

# 浮点运算指令

浮点运算是程序中常用的运算，特别是在高性能计算中  
标量浮点运算

- X87指令支持单精度、双精度和扩展精度的标量浮点运算
- SSE和AVX也支持单精度和双精度的标量浮点运算
- X87标量浮点运算指令的执行由X87浮点运算单元（FPU, Floating-Point Unit）负责，SSE标量浮点运算指令在SIMD运算单元中执行。X87用一个寄存器栈来表示浮点寄存器，运算一般只针对栈顶（以及次栈顶）上存储的浮点数，比较难进行指令调度优化

# 标量浮点运算的SSE指令

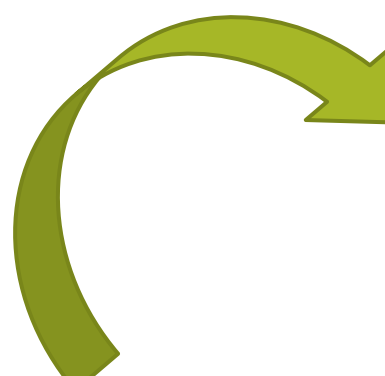
当代编译器一般都倾向于生成SSE标量浮点指令而不太倾向去生成X87标量浮点运算指令，原因：编译优化相对简单。

- SSE标量浮点运算指令的操作码跟X86-64的操作码很相似
- 操作数寄存器用的是XMM寄存器
- SSE指令的操作码用两个字母组成的后缀来对操作数的数据类型进行编码，其中第一个字母用于区分是单个标量（s）还是组装起来的向量（p），第二个字母用于区分是单精度浮点数（s）还是双精度浮点数（d）

Assembly suffix	Data type
ss	One single-precision floating-point value (float)
sd	One double-precision floating-point value (double)
ps	Vector of single-precision floating-point values
pd	Vector of double-precision floating-point values



# 标量浮点运算的SSE指令示例

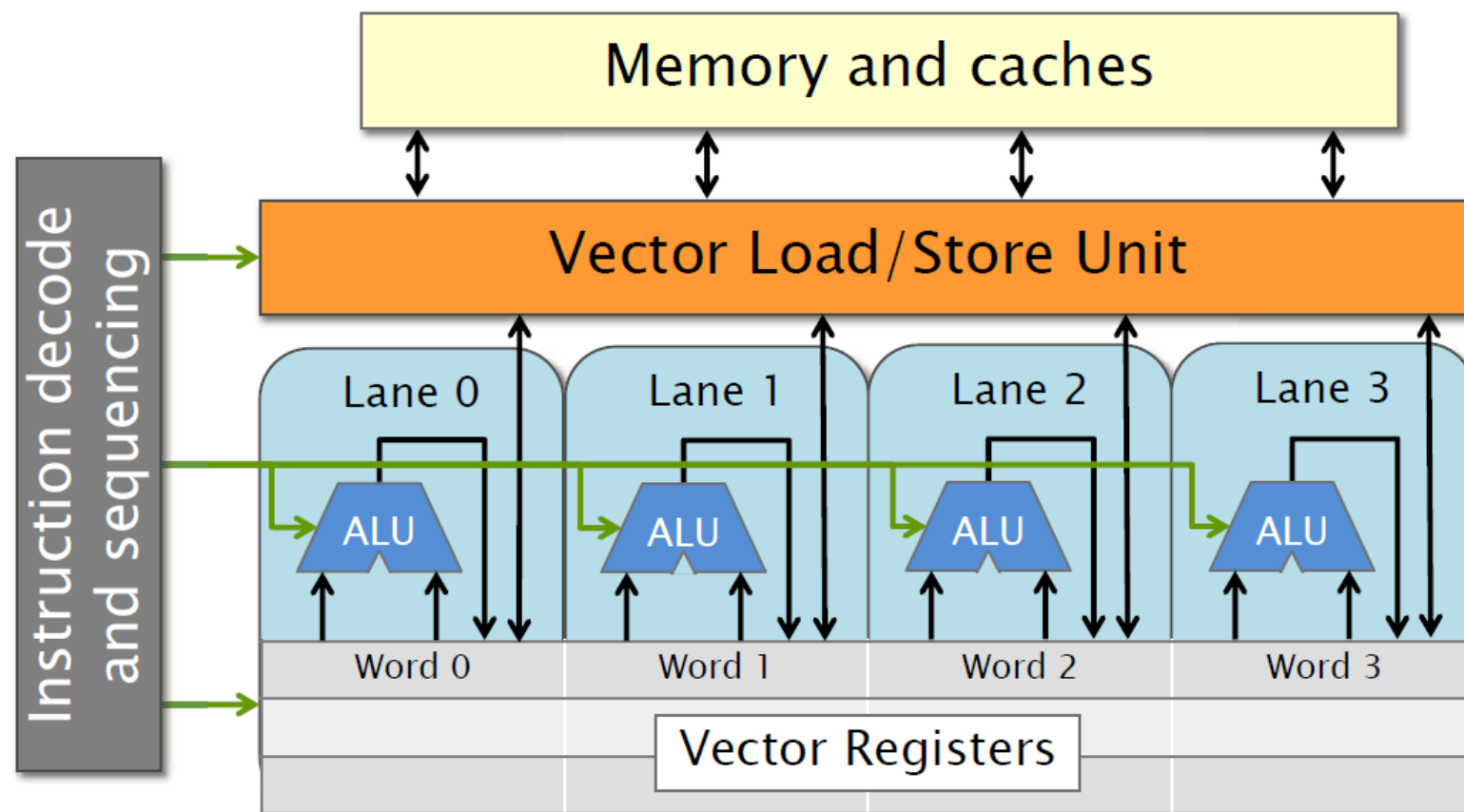


```
movsd (%rcx, %rsi, 8), %xmm1  
mulsd %xmm0, %xmm1  
addsd (%rax,%rsi, 8), %xmm1  
movsd %xmm1, (%rax,%rsi, 8)
```

第三条指令“`addsd (%rax,%rsi, 8), %xmm1`”是一条标量双精度浮点的加法指令（`add`后面的`s`表示单个标量运算，`s`后面的`d`表示双精度浮点数据类型），指令的具体操作是从地址“`(%rax)+(%rsi)*8`”中读取一个双精度浮点数，和寄存器`%xmm1`中的双精度浮点数相加后，把结果保存在`%xmm1`寄存器内。

# Vector Hardware

向量化指令提供了同时对多个操作数进行操作的能力，如下图所示，图中的向量化器件可以对向量化寄存器中的4个元素同时进行操作



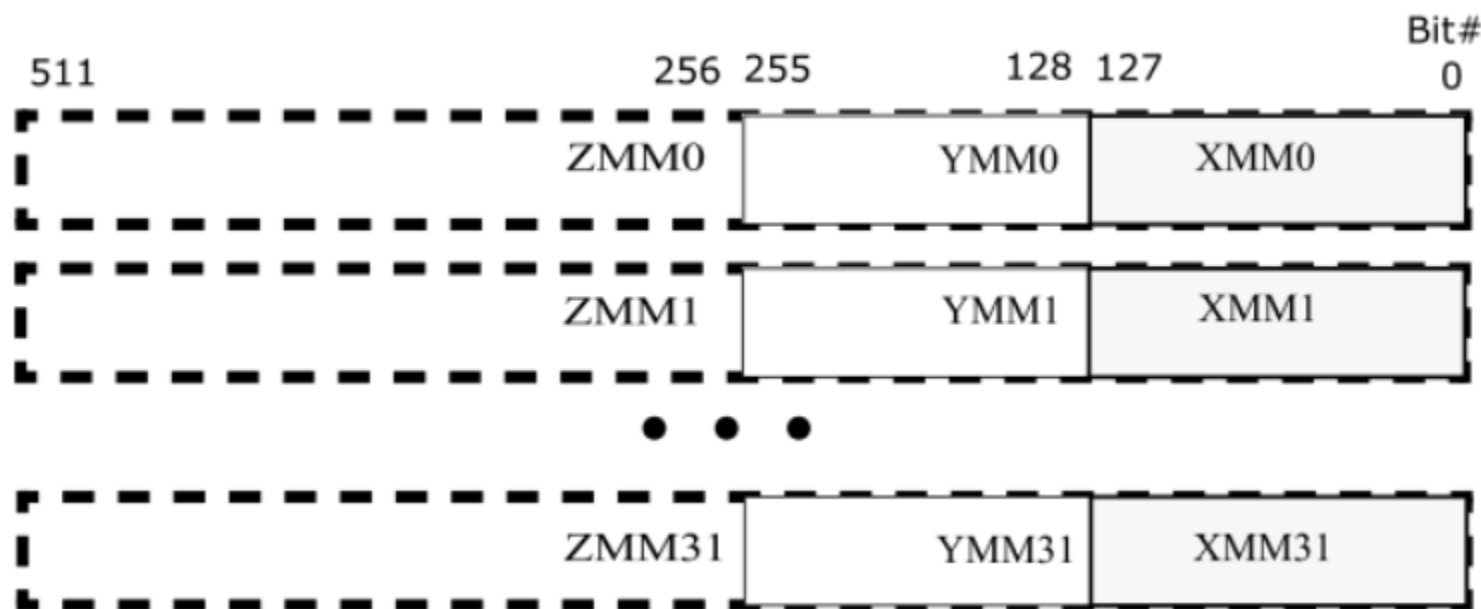
# 向量化指令集

当代的X86-64 架构支持多种向量化指令集：

- **SSE指令**：指令的操作对象是128位的XMM向量化寄存器（共32个，%xmm0-%xmm31），每条SSE指令最多有2个操作数
- **AVX指令**：浮点SIMD运算指令从128位扩展到256位，操作的是YMM向量化寄存器（共32个，%ymm0-%ymm31）。每个YMM向量化寄存器有256位，并且每条AVX向量化指令可以有3个操作数（2个源操作数和一个目的操作数）。以AVX指令“vaddpd %ymm0,%ymm1,%ymm2”为例，它表示的是对%ymm0和%ymm1中的4个double元素同时求和，并把结果存在%ymm2中的相应元素内
- **AVX2指令**：在AVX指令的基础上，把整数SIMD指令的宽度也扩展至256位，同时增加了2个新FMA（Fused-Multiply-Add）单元和一些新的指令
- **AVX512 (AVX3) 指令**：把向量化指令宽度扩展到512位，每条指令可以同时操作的数据成倍增多，向量化寄存器ZMM的也扩展到512位（共32个，%zmm0-%zmm31）；AVX512也增加了一些新的向量操作

# X86-64中向量化寄存器之间的关系

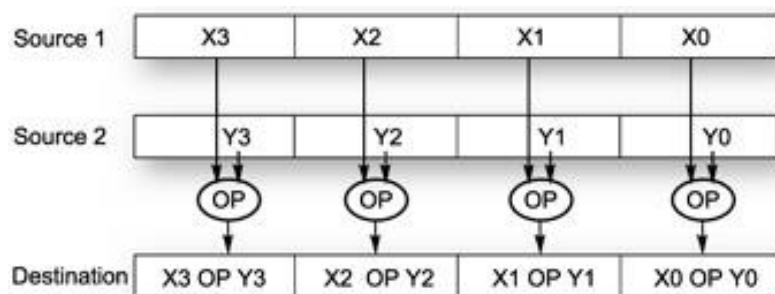
别名：向量化寄存器ZMM<sub>i</sub>的低256位是向量化寄存器YMM<sub>i</sub>，向量化寄存器YMM<sub>i</sub>的低128位是向量化寄存器XMM<sub>i</sub>



# 向量化指令

通常来说，向量化寄存器内所有的向量元素都被执行同样的操作：

- 如果一条向量化指令的两个操作数都是向量化寄存器，则通常的向量化操作方式是一个寄存器的第*i*个元素和另外一个向量化寄存器的第*i*个元素进行操作，运算的结果存储在目标向量化寄存器的第*i*个元素上
- 在向量化指令中，内存操作的地址需要保持一定方式的对齐，具体对齐的方式跟架构相关，通常的对齐方式是地址需要等于向量寄存器长度（按字节计算）的倍数
- 有些架构支持更复杂的向量操作（插入、抽取、置换、分散、聚集等）



a3	a2	a1	a0
*	*	*	*
b3	b2	b1	b0
a3*b3+a2*b2		a1*b1+a0*b0	

PMADDWD: 16b x 16b -> 32b Multiply Add

# SSE与AVX向量操作码

SSE和AVX指令的操作码跟传统的X86-64指令操作码非常相似，只有一些小差别

- 一般SSE指令前面的字母v前缀表明这条指令是AVX/AVX2/AVX512指令
- 标量指令（比如说addq）前面的p前缀表明这条指令是向量化指令

	SSE	AVX/AVX2
浮点运算	addpd	vaddpd
整数运算	paddq	vpaddq

充分利用向量化指令是编译器优化的一个重要方向，编译器通常通过循环向量化以及组合多个没有访问相关性的相同数据类型标量操作来达到向量优化的目的

# Summary



## 一个编译器后端的典型任务

- 指令选择
- 指令调度
- 指令操作数映射
- 寄存器分配
- 遵循函数调用规范
- 优化
- 汇编的文本化打印和显示

**预习LLVM IR**

<https://llvm.org/docs/LangRef.html>

