Chapter 2

Instructions: Language of the Computer

指令：机器的语言

目录

- **机器语言——指令集**
  - **操作码、操作数（寄存器、内存地址、小常量）**
- **数据的二进制表示**
  - **无符号数、有符号数（二进制补码）**
- **指令的二进制表示**
  - **算术逻辑、访存、控制、函数调用**
- **其他表示**
  - **字符串、常量、数组与指针**
- 并行与同步指令
- 程序的编译与运行

Character Data

- 单字节的字符集
  - ASCII码: 128 个字符
    - 95 个图形化可见字符, 33 个控制字符
  - Latin-1码: 256 个字符
    - ASCII码, +96 more graphic characters
- Unicode: 32位的字符集
  - 用于Java, C++宽字符集, …
  - 覆盖世界上的绝大多少字符, 以及符号
  - UTF-8, UTF-16: 变长编码

单字节/多字节 指令

- RISC-V 单字节/多字节 load/store指令
  - Load byte/halfword/word: 符号扩展到64 位 in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: 零扩展到64 位 in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store 最右边的8/16/32位
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

String Copy Example

- C code:

```
void strcpy (char x[], char y[])
{
  unsigned i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

String Copy Example

```
void strcpy (char x[], char y[])
{ unsigned i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

- RISC-V code:

```
strcpy:
    addi sp,sp,-8         // adjust stack for 1 double word
    sd   s2,0(sp)         // push s2
    add  s2,x0,x0         // i=0
L1:    add  t0,s2,a1      // t0 = addr of y[i]
    lbu  t1,0(t0)         // t1 = y[i]
    add  t2,s2,a0         // t2 = addr of x[i]
    sb   t1,0(t2)         // x[i] = y[i]
    beq  t1,x0,L2         // if y[i] == 0 then exit
    addi s2,s2,1          // i = i + 1
    jal  x0,L1            // next iteration of loop
L2:    ld   s2,0(sp)      // restore saved s2
    addi sp,sp,8          // pop 1 double word from stack
    jalr x0,0(x1)         // and return
```

练习

Which of the following is TRUE?

A. add a0,t0,4(x12) is valid in RV32
B. can byte address 8GB of memory with an RV32 word
C. imm must be multiple of 4 for **lw a0,imm(a0)** to be valid
D. None of the above

32位常量（大常量）

- 程序使用的绝大多数常量都很小
  - 12位立即数基本够了
- 偶尔需要使用 32-bit 常量

```
lui rd, constant
```

  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

U-Format for “Upper Immediate” instructions

31 ... 12	11 ... 7	6 ... 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, *rd*
- 一共两条U型指令
  - LUI – Load Upper Immediate
  - AUIPC – Add Upper Immediate to PC

LUI to create long immediates

- LUI 指令
  - 拷贝20位常量到bits [31:12] of rd, Extends bit 31 to bits [63:32], Clears bits [11:0] of rd to 0.
- ADDI指令
  - 设置低12位
- 合起来，在寄存器中存入一个32位的常量值

```
lui s2, 976  // 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

```
addi s2,s2,1280  // 0x500
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

思考：如果要得到0xDEADBEEF，如何？

一个特殊情况

如何存入 0xDEADBEEF?

LUI t0, 0xDEADB # t0 = 0xDEADB000

ADDI t0, t0, 0xEEF# t0 = ~~0xDEADAEEF~~

如果低12位的最高位为1会发生什么? Why?

- ADDI 12-bit immediate 总是有符号扩展的, if top bit is set, it would have subtracted 2^12. To compensate for this error, we need to add 1 into upper 20 bits

解决方案

How to set 0xDEADBEEF?

LUI t0, 0xDEADC   # t0 = 0xDEADC000

ADDI t0, t0, 0xEEF # t0 = 0xDEADBEEF

如果低12位的最高位为1，则需要提前给高20位的值加个1

伪指令:

li t0, 0xDEADBEEF # Creates two instructions

分支目标地址的寻址

- 分支指令指定
  - 操作码, 两个寄存器, 目标地址
- 绝大多数情况下，分支目标靠近分支指令本身
  - Forward or backward
- 分支指令格式为SB型, 类似 S-型指令:

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

imm[12]	imm [10:5]	rs2	rs1	funct3	imm [4:1]	imm[11]	opcode

- PC相对寻址
  - 目标地址 = PC + immediate × 2

RISC-V 中SB-型指令——分支指令

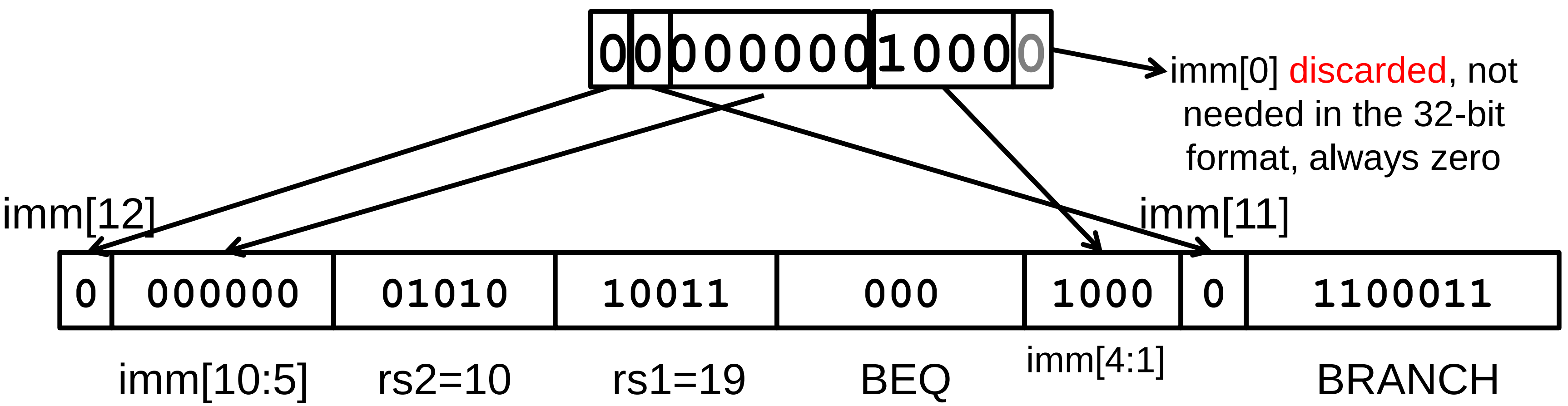
31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7

- SB-型几乎跟S-型指令一样, 有2个源寄存器 (rs1/rs2) 和 12位常量
- 但是常量代表的值的范围为：-4096 to +4094 in 2-byte increments
- 12位常量本质上编码了13-位 有符号的字节偏移(lowest bit of offset is always zero, so no need to store it)

分支指令的例子

`beq   s2,t0, offset = 16 bytes`

13-bit immediate, imm[12:0], with value 16



跳转目标地址的寻址

- Jump and link (`jal`) saves PC+4 in register rd (the return address)
  - “j” 是一条伪指令：JAL x0, imm，放弃了保存返回地址
- 目标地址使用20-位立即数来支持更大范围的跳转:
  - $\pm 2^{20}$ byte addresses，$\pm 2^{19}$ locations, 2-byte addresses
- 跟分支指令相似，优化了立即数的编码来节省硬件成本
- UJ format:

	imm[10:1]		imm[19:12]	rd	opcode
imm[20]		imm[11]		5 bits	7 bits

- 还可以利用jalr支持32位的长距离跳转：
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

JAL的例子

- 无条件跳转

j 伪指令

j Label = jal x0, Label # Discard return address

  - vs jal ra, Label

- 函数调用

Call function within ±2^{18} 32-bit instructions of PC

jal ra, FuncName

JALR 指令 (I-型)

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
offset[11:0]	base	0	dest	JALR

- **JALR rd, rs, immediate**
  - Writes PC+4 to **rd** (return address)
  - Sets PC = **rs + immediate**
  - 跟load指令一样，使用12位常量编码字节地址
    - ***no*** multiplication by 2 bytes

JALR的例子

\# 返回指令：ret and jr 伪指令

ret = jr ra = jalr x0, ra, 0

\# long call指令-绝对地址：

lui x1, <hi20bits>

jalr **ra**, x1, <lo12bits>

\# long jump指令-相对地址：Jump PC-relative with 32-bit offset

auipc x1, <hi20bits> # Adds upper immediate value to PC寄存器

\# and places result in x1

jalr **x0**, x1, <lo12bits> # 注意：有符号扩展

\# 间接跳转/间接函数调用：return under indirect jump

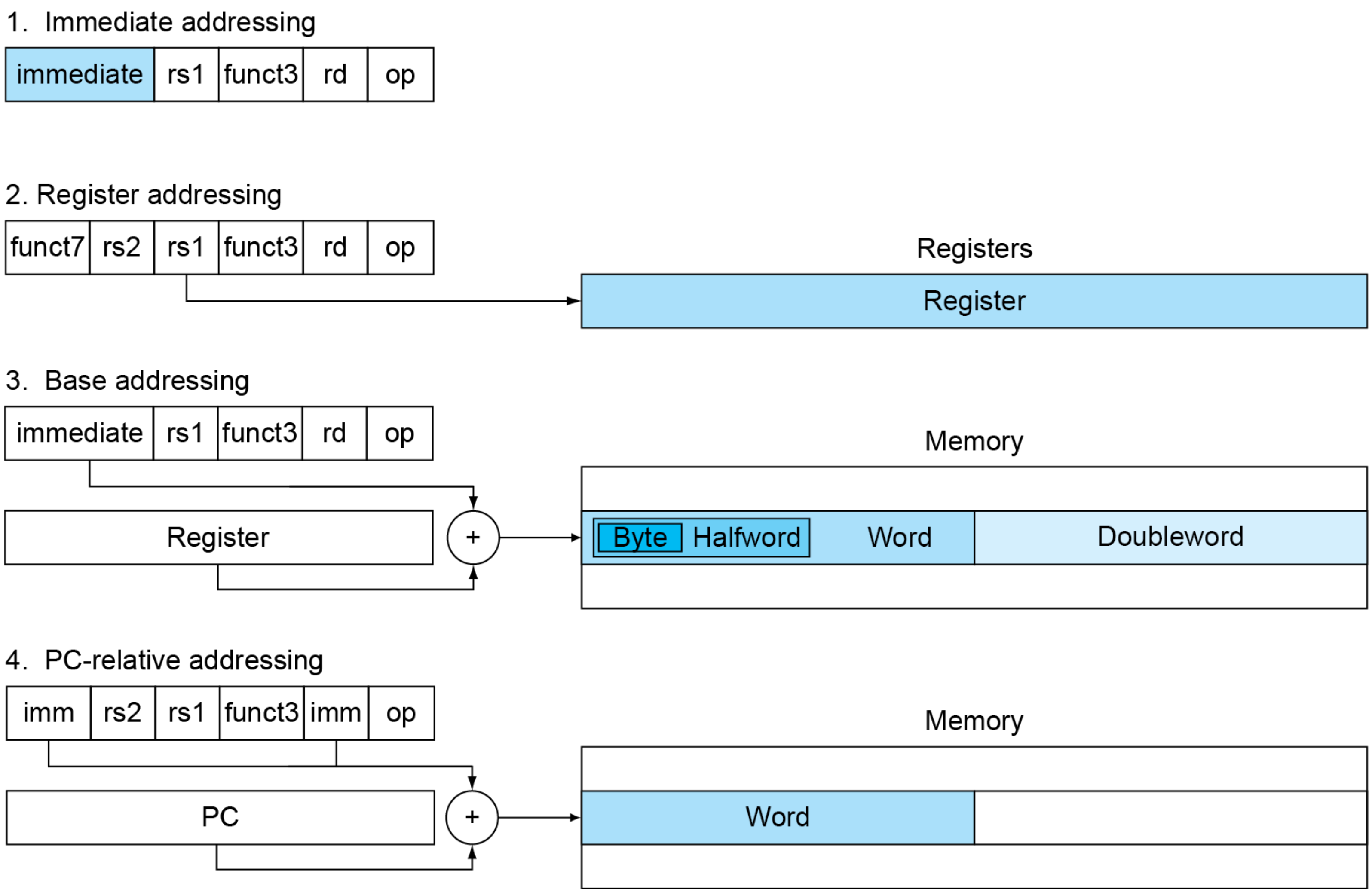
- with/without saving return address into ra

回顾JAL 和JALR指令

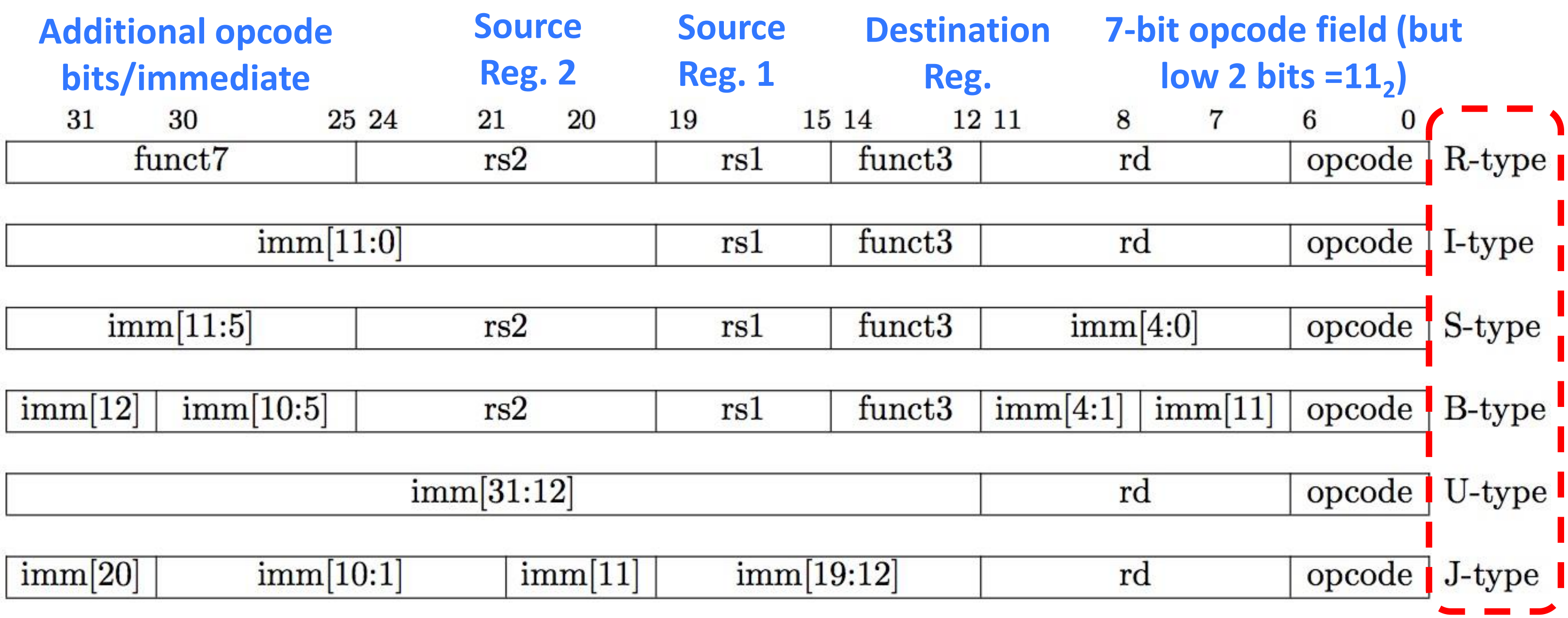
- JAL
  - 调用一个函数: jal ra, Func
    - 保存返回地址: PC + 4 -> ra
    - 跳转: Func (PC + imm) -> PC
  - 跳转: jal x0, Label
    - (PC + imm) -> PC
- JALR
  - 间接调用 a function: jalr ra, x1, imm
    - PC + 4 -> ra;   (X1 + imm) -> PC
  - 间接跳转: jalr x0, x1, imm
  - 返回: jalr x0, ra, 0
  - long call:

RISC-V 寻址模式总结

addi
add
ld/sd
jal
beq



RISC-V指令格式总结



- 指令的内存地址按照4字节对齐
- 立即数的符号位总是在第31位
- 寄存器的位置绝对不变
- 操作码在最右边

数组 vs. 指针

- 数组的索引计算需要
  - 索引i乘以元素的宽度
  - 再加上数组的基地址
- 指针就是内存地址
  - 可以避免索引计算

例子: 清空一个数组

<pre>clear1(long array[], long size) {
  long i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}</pre>
	<pre>clear2(long *array, int size) {
	  int *p;
	  for (p = &array[0]; p < &array[size];
	       p = p + 8)
	    *p = 0;
	}</pre>

<pre>   li   x5,0   // i = 0
loop1:
   slli t1,x5,3   // t1 = i * 8
   add t2,a0,t1 // t2 = address of
array[i]
   sd   x0,0(t2)   // array[i] = 0
   addi x5,x5,1   // i = i + 1
   blt x5,a1,loop1 // if (i<size)
                      // go to loop1</pre>
	<pre> mv x5,a0 // p = address of
	array[0] mv 将一个寄存器拷贝到另一个寄存器
	   slli t1,a1,3 // t1 = size * 8
	   add t2,a0,t1 // t2 = address of
	array[size]
	loop2:
	   sd x0,0(x5) // Memory[p] = 0
	   addi x5,x5,8 // p = p + 8
	   bltu x5,t2,loop2
	                  // if (p<&array[size])
	                  // go to loop2</pre>

Array vs. Ptr

- 乘法可以替换成移位“strength reduced”
- 数组版本，需要在循环内部移位
  - 索引的计算需要增加i移位，加基地址，计算内存地址
- 指针版本
  - 直接增加指针的值
- 编译器可以自动优化成指针版本

目录

- **机器语言——指令集**
  - **操作码、操作数（寄存器、内存地址、小常量）**
- **数据的二进制表示**
  - **无符号数、有符号数（二进制补码）**
- **指令的二进制表示**
  - **算术逻辑、访存、控制、函数调用**
- **其他表示**
  - **字符串、常量、数组与指针**
  - **例子**
- **并行与同步指令**
- 程序的编译与运行

并发与同步

- 为什么需要同步——共享变量
  - 两个并发线程：下列2种情形，谁会开心?

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  spit out cash;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  spit out cash;
}
```

```
balance = get_balance(account);
balance -= amount;
```

余额：100
100-10 = 90

context switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
spit out cash;
```

读余额：100
100-10 = 90
写余额：90
收现金：10

context switch

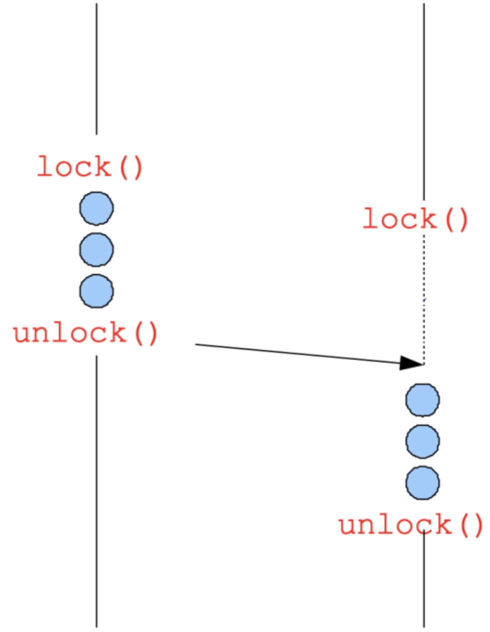
```
put_balance(account, balance);
spit out cash;
```

写余额：90
收现金：10

问题：没有确保“读-计算-写回”的原子性!

并发与同步

- 如何实现同步
- 利用**原子锁**，确保一个事务的**原子性**
  - 原子锁lock()的语义
    - 如果当前是开锁状态，则授权上锁(...后续可以工作)
    - 如果当前是上锁状态，则不能授权
  - 原子锁unlock()的语义
    - 开锁走人
  - 能否实现同步？



并发与同步

- 如何实现原子锁
  - 需求
    - 检查锁状态load—开锁状态，则授权上锁store
    - 至少需要保证load&store的原子化
  - 挑战
    - CPU调度以**指令**为基本单位。对于调度来说，单条指令是原子的；多条指令则不是
  - ~~方案：一条指令完成?~~
    - 该指令会很复杂
  - 方案：多条指令完成（类似于同步库函数）
    - load-reserved + store-conditional，构成automic

并发与同步

- Load-reserved and StoreConditional
  - load-reserved <==> LoadLinked
    - load
    - 同时reserve register记录load时的memory address
  - StoreConditional
    - 如果自load以来没被写过，则store并返回1；否则返回0

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

并发与同步

- lock的C实现

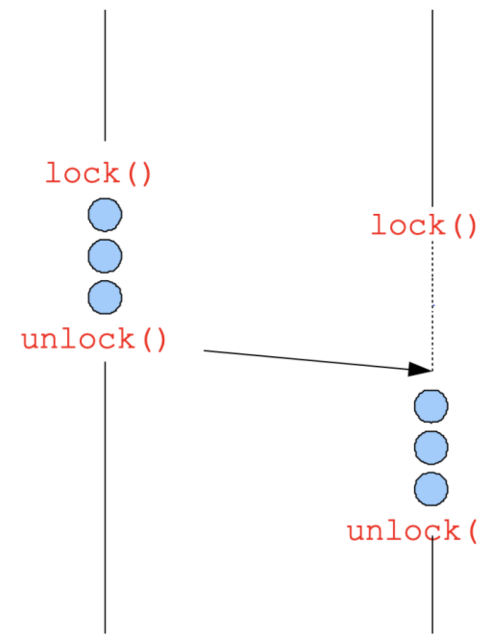
```c
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag) ||
           !StoreConditional(&lock->flag, 1))
        ; // spin
}
```

- lock的汇编实现：
  - Example 2: lock

```
        addi x12,x0,1          // copy locked value
again:  lr.d x10,(x20)         // read lock
        bne  x10,x0,again      // check if it is 0 yet
        sc.d x11,(x20),x12     // attempt to store
        bne  x11,x0,again      // branch if fails
```

  - Unlock:

```
        sd   x0,0(x20)         // free lock
```



Synchronization

- Two processors sharing an area of memory
  - e.g., P1 read&write, P2 read&write
    - the prevous bank example
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - **Atomic** read &write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., **atomic** swap of register ↔ memory
  - Or an **atomic** pair of instructions

Synchronization in RISC-V

- Load reserved: `lr.d rd,(rs1)`
  - Load from address in rs1 to rd
  - Place reservation on memory address
- Store conditional: `sc.d rd,rs2,(rs1)`
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the `lr.d`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

https://www.youtube.com/watch?v=fuHwmyZXnPA
https://www.youtube.com/watch?v=pcNCw8iAp8A

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again: lr.d t0,(s3)    // ; amount -> t0
       sc.d t0,(s3),x23 // t0 = status;  ; new
       amount ->
       bne  t0,x0,again  // branch if store failed
       addi x23,t0,0     // X23 = loaded value
```

- Example 2: lock

```
       addi x12,x0,1   // copy locked value
again: lr.d t0,(s3)    // read lock
       bne  t0,x0,again      // check if it is 0 yet
       sc.d t0,(s3),x12      // attempt to store
       bne  t0,x0,again      // branch if fails
```

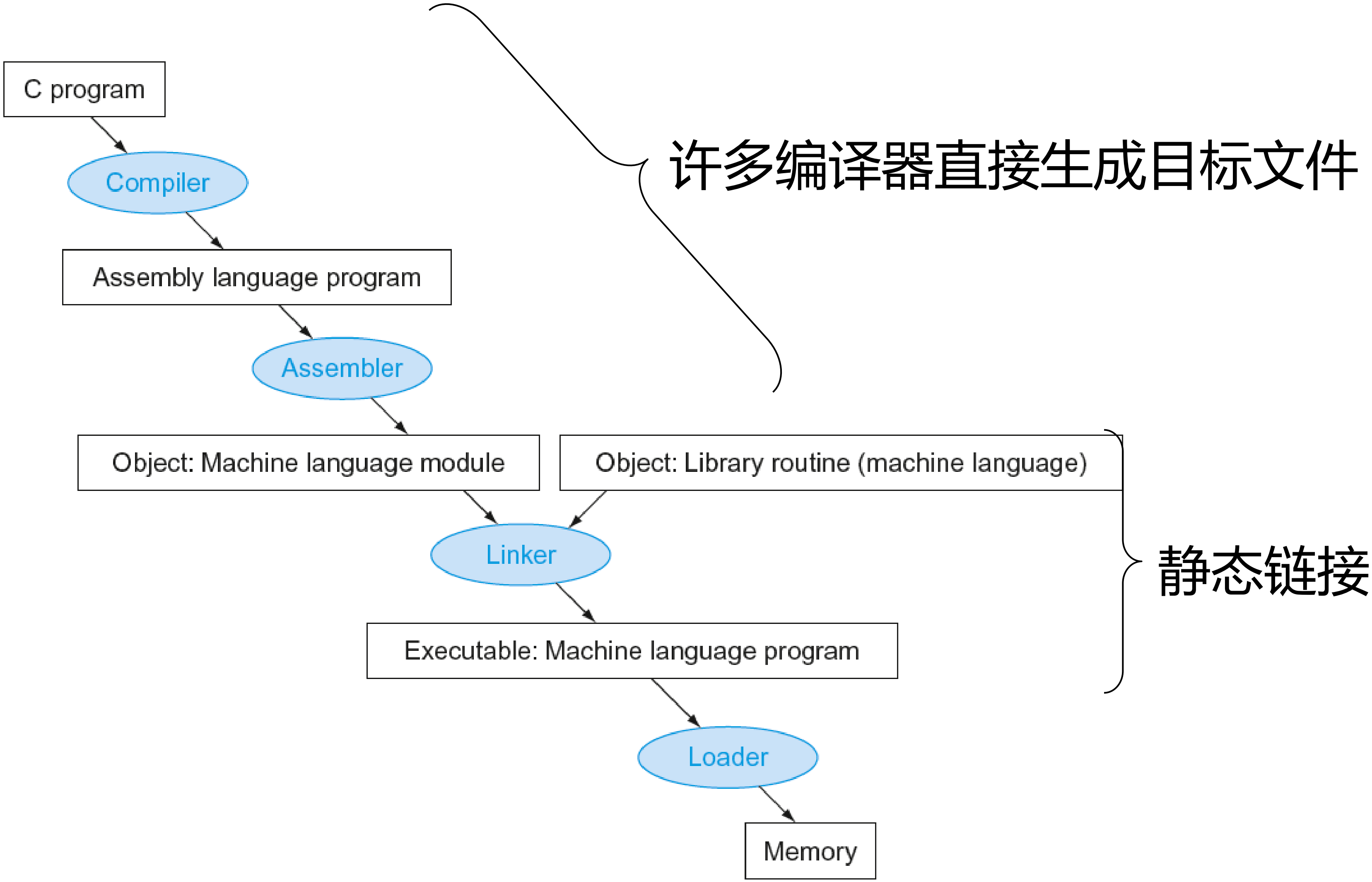
- Unlock:

```
       sd   x0,0(s3)   // free lock
```

目录

- 机器语言——指令集
  - 操作码、操作数（寄存器、内存地址、小常量）
- 数据的二进制表示
  - 无符号数、有符号数（二进制补码）
- 指令的二进制表示
  - 算术逻辑、访存、控制、函数调用
- 其他表示
  - 字符串、常量、数组与指针
- 并行与同步指令
- 程序的编译与运行

程序的翻译与启动


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
许多编译器直接生成目标文件
静态链接

生成一个目标模块文件

- 编译器将单个源代码文件翻译成单个机器指令文件
  - 称为目标模块（object module）
- 目标模块中，提供用于构建完整程序所需的信息
  - Header: 描述目标模块的内容
  - Text segment: 包含翻译后的机器指令
  - Static data segment: 全程序生命期的数据 (全局、静态数据)
  - Relocation info: 记录暂缺信息的部分（比如另一个模块中函数的地址），需要链接时修改
  - Symbol table: 本模块提供或引用的全局函数/数据
  - Debug info: 用于关联到源代码的位置信息

链接多个目标模块文件

- 生成一个可执行的镜像文件
  1. 合并多个模块中同名的segments
  2. 解析外部符号 (同时确定外部符号的地址)
  3. 修改依赖外部符号的指令/数据信息
- 可能需要动态链接
  - 如果外部符号的地址需要运行时才能动态确定

装载一个程序

- 将可执行镜像文件从磁盘装入内容
  1. 读取header信息，确定segment的位置和大小
  2. 创建虚拟地址空间（一个数据结构）
  3. 将代码和初始化数据**拷贝到内存**
     - 也可以利用页表缺页处理延迟拷贝
  4. 在栈上**准备参数**
  5. 初始化寄存器 (包括sp, fp, gp)
  6. 跳转到程序中的启动函数（startup routine）
     - 将命令行参数拷贝到 a0, … 寄存器，然后调用main
     - 当main函数返回时，调用 exit系统调用

动态链接

- 概念
  - 仅当调用发生时，才链接和装载库代码
  - 需要库函数允许在运行时重新分配地址的
- 优点
  - 允许运行时多个程序复用共同的库代码
    - 可以避免静态链接一次性链接所有依赖库导致的代码膨胀
  - 当库代码的版本更新时，可以在运行时自动重新链接到新版本

一个完整的例子：排序算法

- 展示一个C语言冒泡排序算法的汇编指令
- Swap函数(叶子函数)

```
void swap(long long int v[],
          long long int k)
{
  long long int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - 假设：v in a0, k in a1, temp in t0

swap函数

```
swap:
  slli t1,a1,3          // reg t1 = k * 8
  add  t1,a0,t1         // reg t1 = v + (k * 8)
  ld   t0,0(t1)         // reg t0 (temp) = v[k]
  ld   t2,8(t1)         // reg t2 = v[k + 1]
  sd   t2,0(t1)         // v[k] = reg t2
  sd   t0,8(t1)         // v[k+1] = reg t0 (temp)
  jalr x0,0(x1)         // return to calling routine
```

sort函数

- 非叶子函数 (调用swap)

```
void sort (long long int v[], long long
int n)
{
  long long int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
          j >= 0 && v[j] > v[j + 1];
          j -= 1) {
      swap(v,j);
    }
  }
}
```

  - 假设：v in a0, n in a1, i in s2, j in s3

外层循环

- 外层循环的框架:
  - for (i = 0; i <n; i += 1) {

v in a0, n in a1, i in s2, j in s3

```
    li    s2,0                     // i = 0
for1tst:
    bge   s2,a1,exit1          // go to exit1 if s2 ≥ a1
    (i≥n)

    (循环体)

    addi  s2,s2,1    // i += 1
    j     for1tst              // 跳转到外层循环的条件检测
exit1:
```

内层循环

- 内层循环的框架: v in a0, n in a1, i in s2, j in s3
  - for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j – = 1) { swap(v,j);

```
   addi  s3,s2,-1      // j = i –1
for2tst:
     blt   s3,x0,exit2   // go to exit2 if s3 < 0 (j < 0)
     slli  t0,s3,3       // reg t0 = j * 8
     add   t0,a0,t0      // reg t0 = v + (j * 8)
     ld    t1,0(t0)       // reg t1 = v[j]
     ld    t2,8(t0)       // reg t2 = v[j + 1]
     ble   t1,t2,exit2    // go to exit2 if t1 ≤ t2
     mv    s4, a0        // copy parameter a0 into s4, 保护
   caller-save寄存器
     mv    s5, a1        // copy parameter a1 into s5, 保护
   caller-save寄存器
     mv    a0, s4        // first swap parameter is v, 传参数
     mv    a1, s3        // second swap parameter is j, 传参
   数
   jal   x1,swap       // call swap
   addi  s3,s3,-1      // j -= 1
   j     for2tst       // 跳转到内层循环的条件检测
```

保护Registers

- callee保护寄存器:

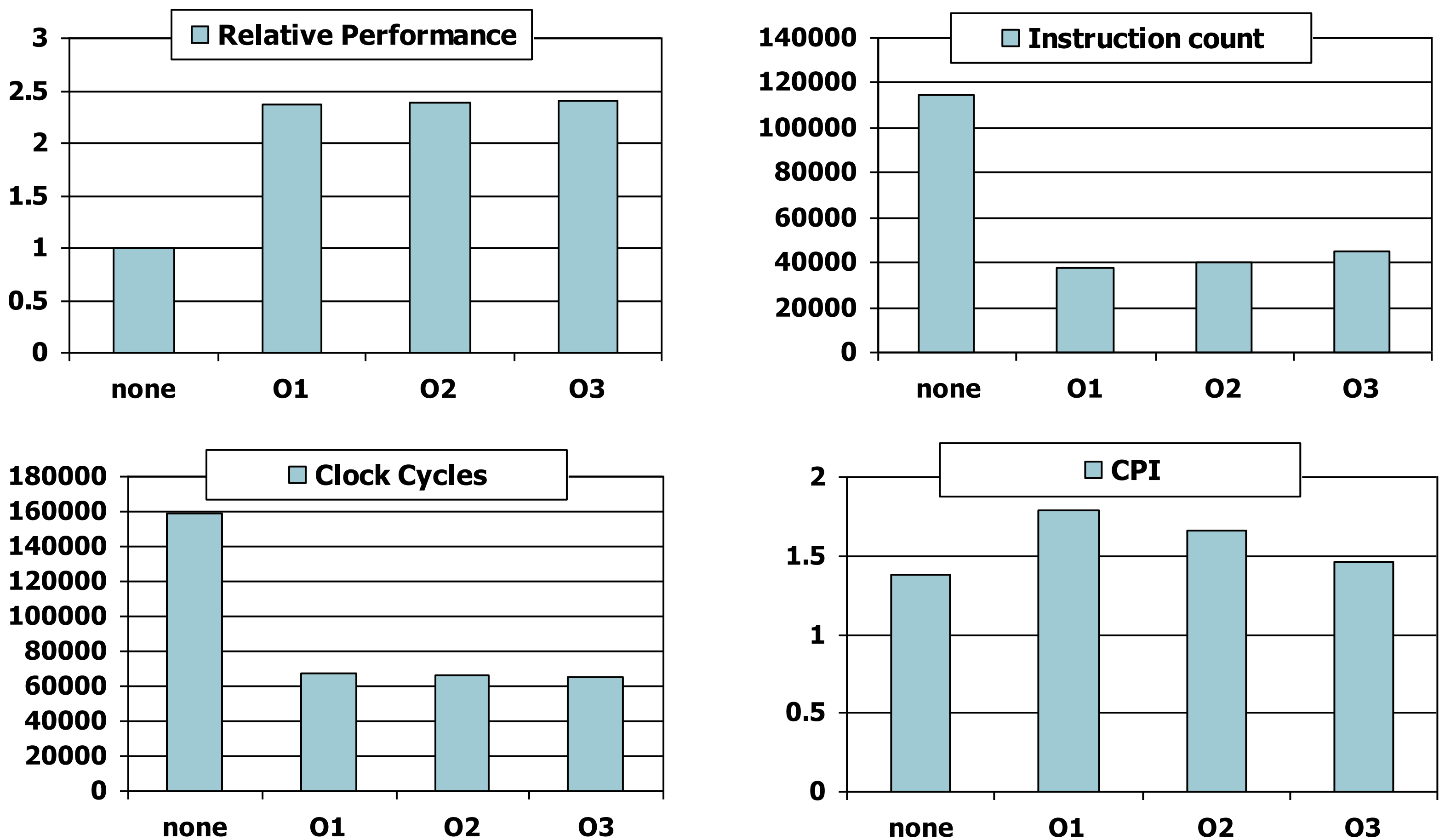
```
addi  sp,sp,-40   // 为5个寄存器分配了40字节的空间（此例子中寄存器为64位宽）
sd    x1,32(sp)   // save x1 on stack
sd    s5,24(sp) // save s5 on stack
sd    s4,16(sp) // save s4 on stack
sd    s3,8(sp)  // save s3 on stack
sd    s2,0(sp)  // save s2 on stack
```

- callee恢复寄存器:

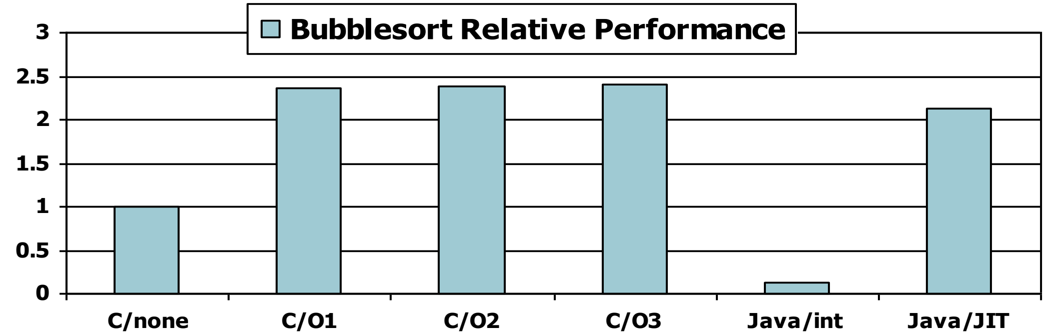
```
exit1:
  ld    s2,0(sp)         // restore s2 from stack
  ld    s3,8(sp)  // restore s3 from stack
  ld    s4,16(sp) // restore s4 from stack
  ld    s5,24(sp) // restore s5 from stack
  ld    x1,32(sp)  // restore x1 from stack
  addi  sp,sp, 40  // restore stack pointer
  jalr  x0,0(x1)
```

编译器优化对性能的影响

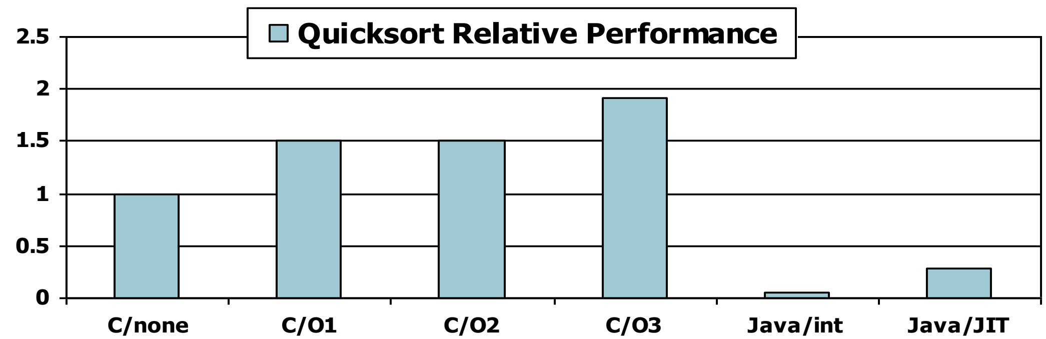
Compiled with gcc for Pentium 4 under Linux

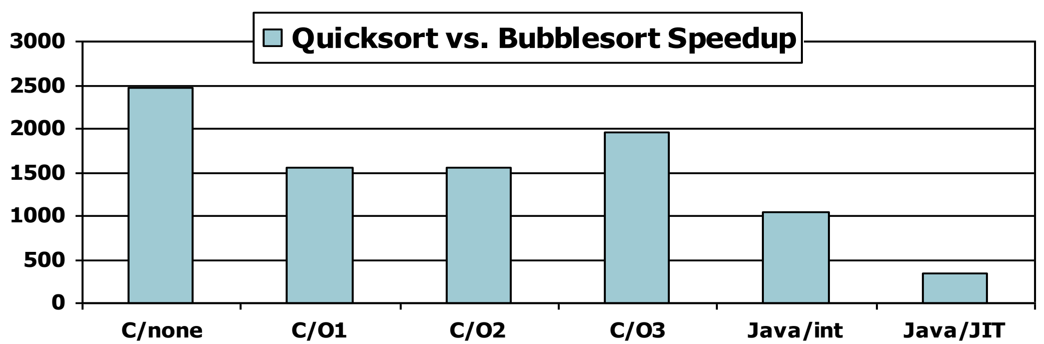


编程语言与算法对性能的影响



sort 100,000
randomized data





经验与教训

- 指令的总数 和 CPI 不能单独作为性能指标
- 编译优化的效果，受到算法的影响
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

其他RISC-V指令

- 其他基本的整数指令 (RV64I)
  - auipc rd, immed // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
  - addw, subw, addiw: 32-bit add/sub
  - sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift
- 32-bit variant: RV32I
  - registers are 32-bits wide, 32-bit operations

指令集的扩展

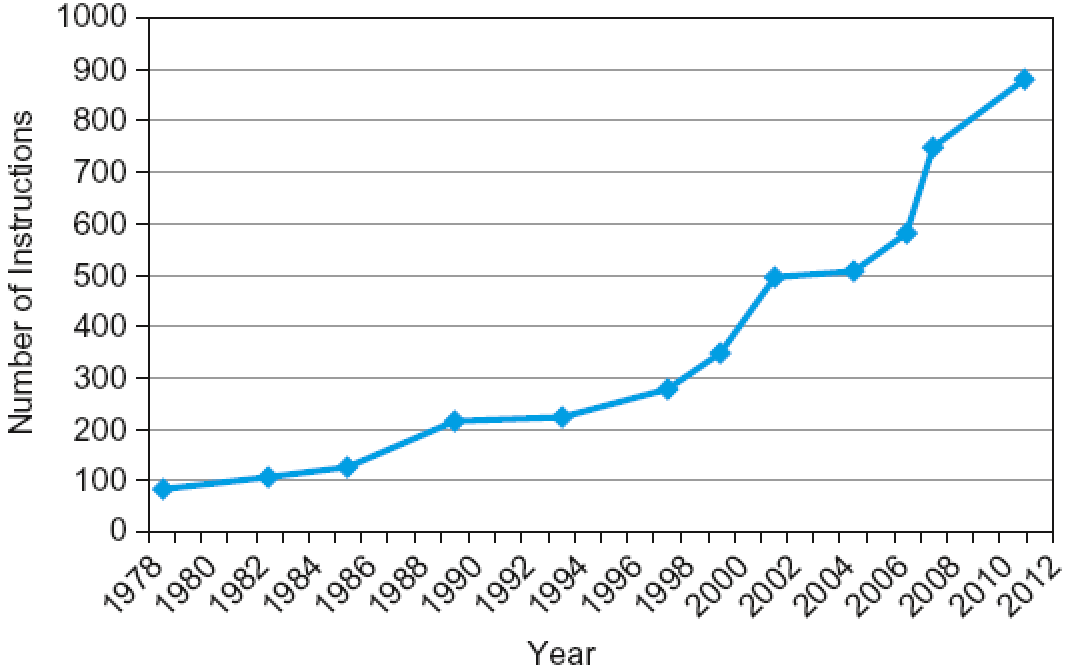
- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 对高频使用的的指令，采取16-位编码

误解

- 强大的指令 ⇒ 更好的性能
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- 汇编编程可以获得更好的性能
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

误解

- 后向兼容 ⇒ 指令集不再变化
  - 但是，确实导致指令集的膨胀



x86 instruction set

总结

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises
- Make the common case fast
- Layers of software/hardware
  - Compiler, assembler, hardware
- RISC-V: typical of RISC ISAs