计算机组成原理
第二章“指令系统”

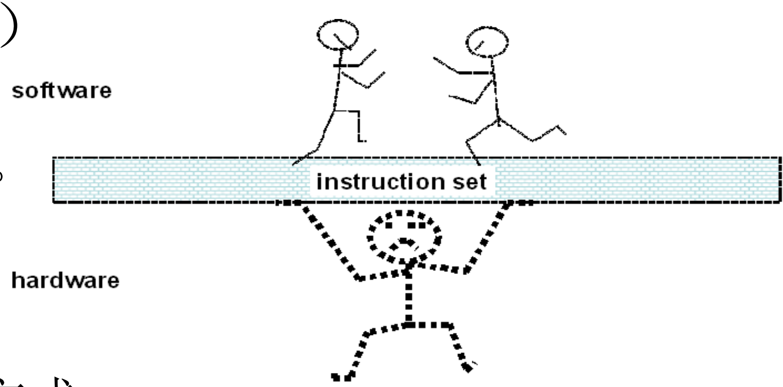
中科大11系
李曦

体系结构的8种属性

- 数据表示
  - 硬件能直接辨识和操作的数据类型和格式
- 寻址方式
  - 最小可寻址单位、寻址方式的种类、地址运算
- 寄存器组织
  - 操作寄存器、变址寄存器、控制寄存器及专用寄存器的定义、数量和使用规则
- 指令系统
  - 机器指令的操作类型、格式、指令间排序和控制机构
- *存储系统*
  - *最小编址单位、编址方式、主存容量、最大可编址空间*
- *输入输出*
  - *输入输出的连接方式、处理机/存储器与输入输出设备间的数据交换方式、数据交换过程的控制*
- *中断机构*
  - *中断类型、中断级别，以及中断响应方式等*
- *信息保护*
  - *信息保护方式、硬件信息保护机制*

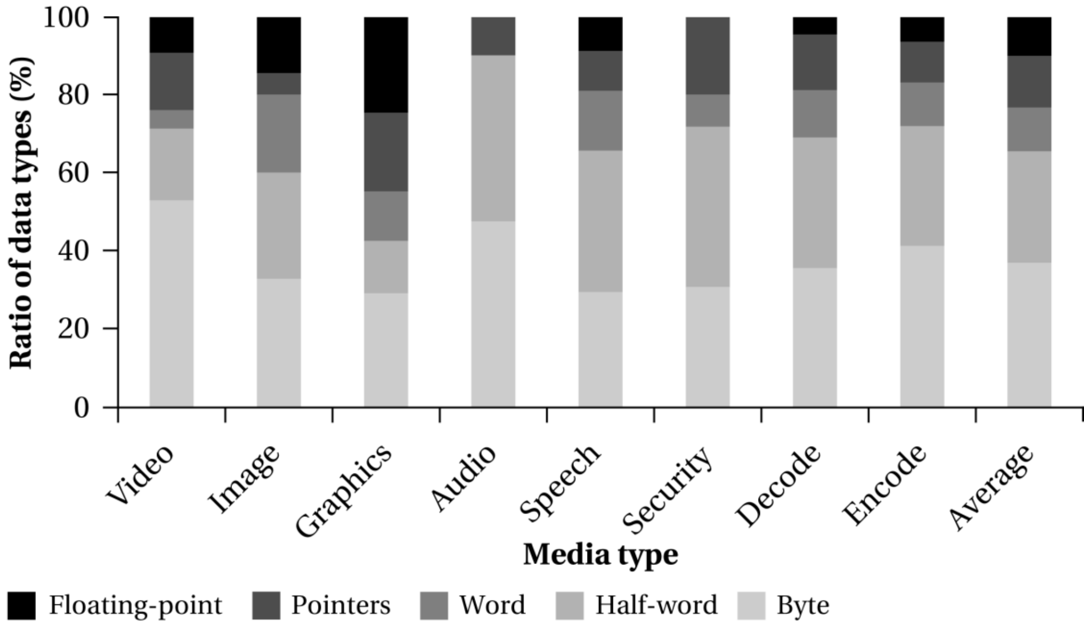
概要

- “程序控制”
  - 程序=顺序执行的指令流
- 指令系统：机器指令的集合
  - 汇编语言（Assemble Language）/机器语言
  - Instruction Set Architecture（ISA）
    - 分类：CISC、RISC、VLIW
    - 影响：处理器、C编译器、OS。。。
- 本章的内容
  - RV指令系统
    - 指令功能，指令格式与编码，寻址方式
  - 汇编程序设计：C语言的机器表示
    - 可执行程序生成：编译，汇编，链接，加载
  - 指令系统特征



操作数（opr）

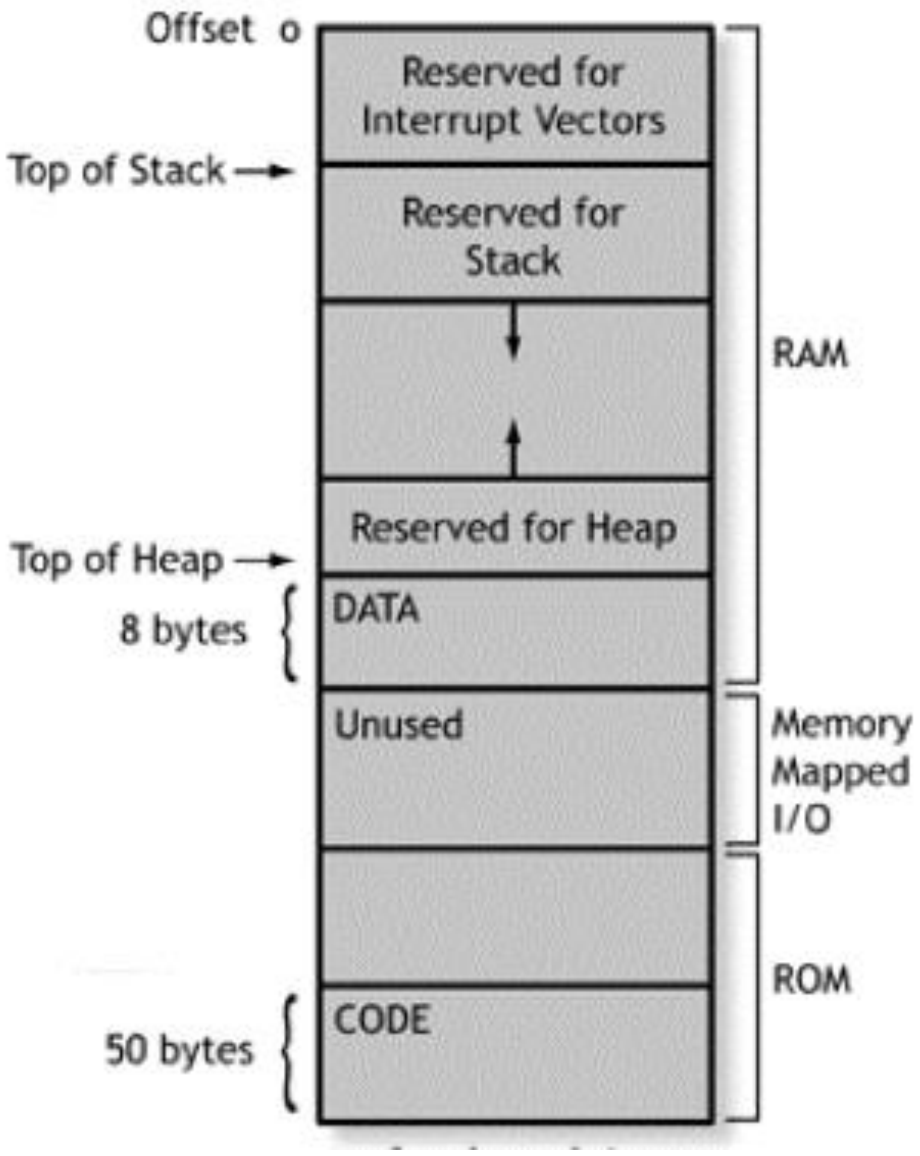
- 操作数类型：进制，编码，立即数（补码）
  - 地址：无符号整数。寄存器、内存、I/O端口ID
  - 数值：常数、定点数（有符号/无符号）、浮点数、逻辑值
  - 字符：ASCII、汉字内码
- 字长：“RV32I”——32位，*“RV64 ”——64位（“大立即数”）*
  - 字节
  - 半字：2B
  - 字：4B
  - *双字：8B（大立即数）*
- 存放位置
  - 寄存器
  - 主存
  - I/O端口
  - 外存？



典型内存地址空间分配

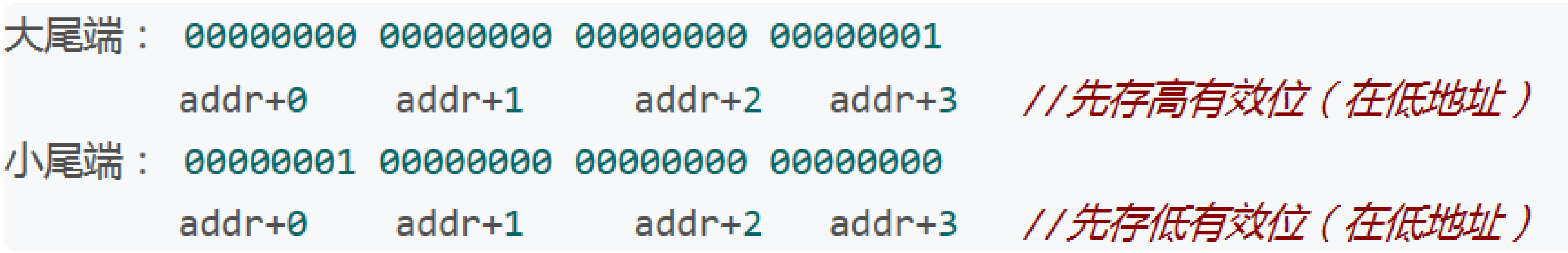
- 段式
  - DATA
  - CODE
  - Stack/Heap

- I/O port?
- 硬盘
- 网络



字存储顺序（Byte Ordering）

- 字存储顺序中，字节的次序有两种
  - 大尾端（big endness）：低地址，高字节
  - 小尾端（little endness）：低地址，低字节
    - X86和RV都为小端，ARM可以自主设置
- 00000000 00000000 00000000 00000001
  - 00000000 00000111 00000011 00000001?



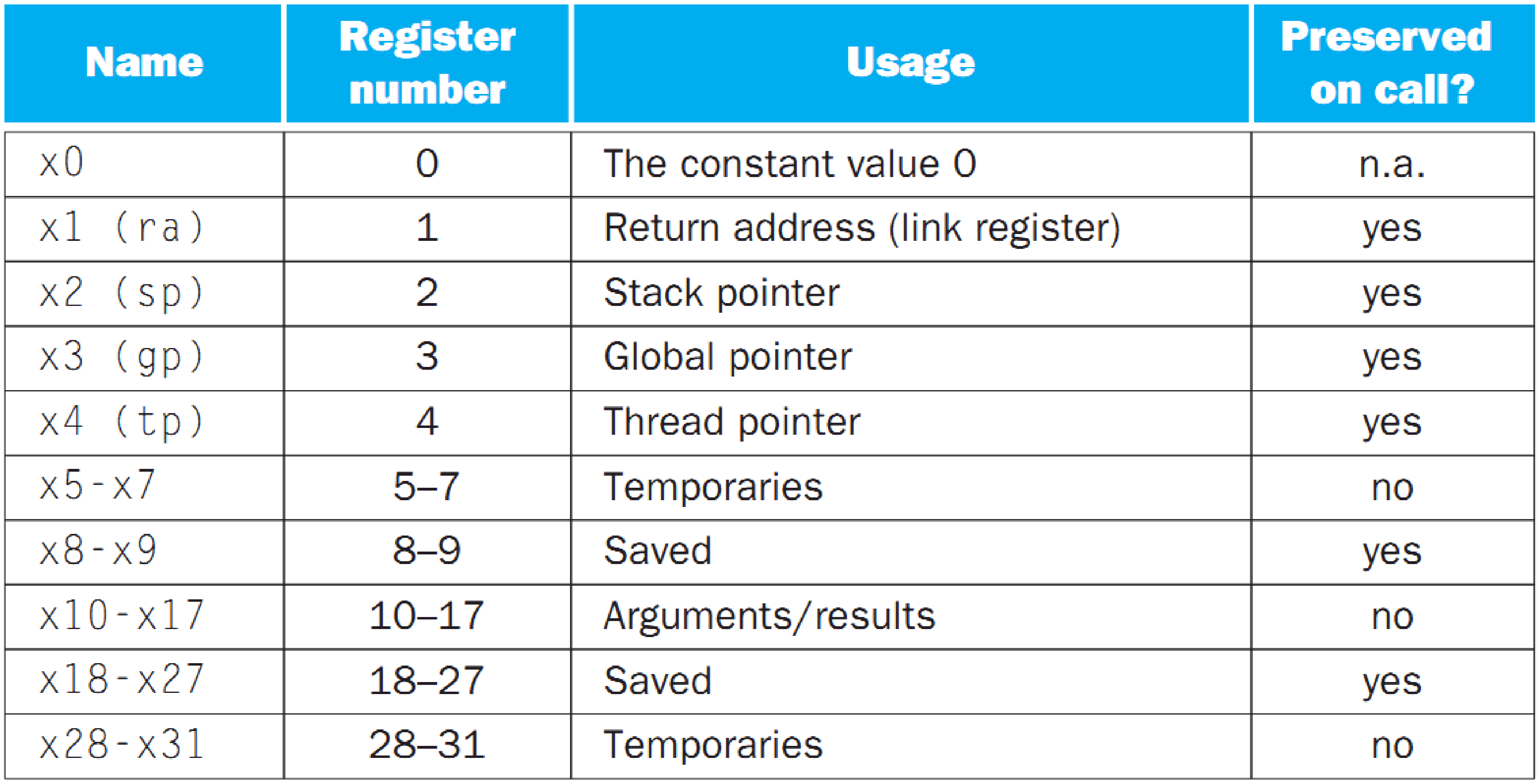
```
大尾端： 00000000 00000000 00000000 00000001
         addr+0   addr+1     addr+2   addr+3   //先存高有效位（在低地址）
小尾端： 00000001 00000000 00000000 00000000
         addr+0   addr+1     addr+2   addr+3   //先存低有效位（在低地址）
```

数据存放位置（Memory Alignment）

- 在数据不对准边界的计算机中，数据（例如一个字）可能在两个存储单元中。
  - 此时需要访问两次存储器，并对高低字节的位置进行调整后，才能取得一字。
- 边界对准：
  - 字对齐：左移两位，按字访问
    - RV/x86不要求，MIPS要求
  - 半字：左移一位，按半字访问

存储器			地址（十进制）
字（地址2）		半字（地址0）	0
字节（地址7）	字节（地址6）	字（地址4）	4
半字（地址10）		半字（地址8）	8

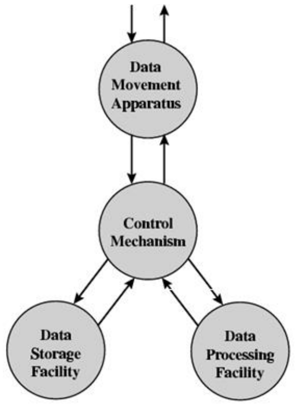
RV architected registers： RV64/RV32，图2-14



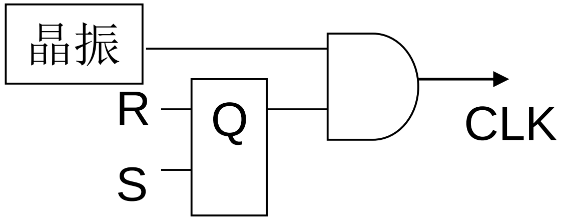
Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

- Preserved：在函数调用中保持不变

操作分类



- 数据传递（data movement）
  - 访存：mov，load，store
  - I/O：in，out
- 算逻运算（arithmetic & logical）
  - add，sub，and，not，or，xor，*dec，inc，cmp*
  - monadic & dyadic operations
- 移位操作
  - monadic operations：shl，shr，srl，srr
- 分支控制（tranfer of contral，Branch）
  - comparisons & conditional branches：beq，bnz，jmp
  - procedure call：call，ret，int，iret
- 系统指令
  - HLT，nop，wait，sti，cli，lock



指令字格式Machine Instruction Layout

- von Neumann：　“指令由**操作码**和**地址码**构成”
- 操作码：操作的性质
- 地址码：指令和操作数（operand）的存储位置

操作码域（op）	地址码域（addr）

- 指令字长度固定vs.可变：RISC（RV/MIPS/ARM）一般32位
  - 固定：规则，浪费空间
- 操作码长度固定vs.可变
  - 固定：译码简单，指令条数有限，RISC（RV/MIPS/ARM）
  - 可变：指令条数和格式按需调整，CISC（x86）
    - “扩展操作码技术”：调整op与addr域
      - 如果指令字长固定，则操作码长度增加，地址码长度缩短

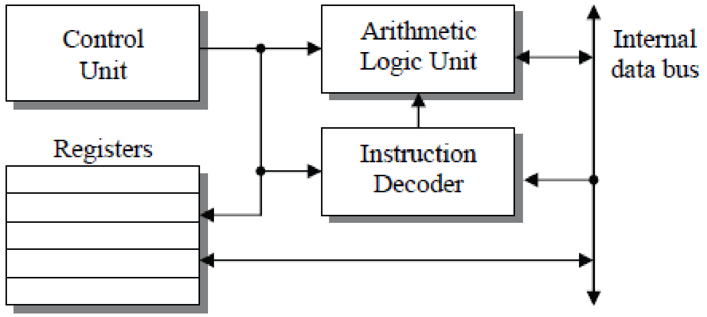
地址码：数据，指令

- 指定源操作数、目的操作数、下一条指令地址
  - 地址：寄存器、主存、I/O端口
- 地址码域格式
  - 4地址指令：op rs1, rs2, rd, ni
  - 3地址指令：op rs1, rs2, rd；　ni在PC中
  - 2地址指令：op rs1, rs2；　rd=rs1 or ACC
  - 1地址指令：op rs2；　rs1=ACC，rd=ACC
  - 0地址指令：op；　堆栈操作

寻址方式：指令的地址码

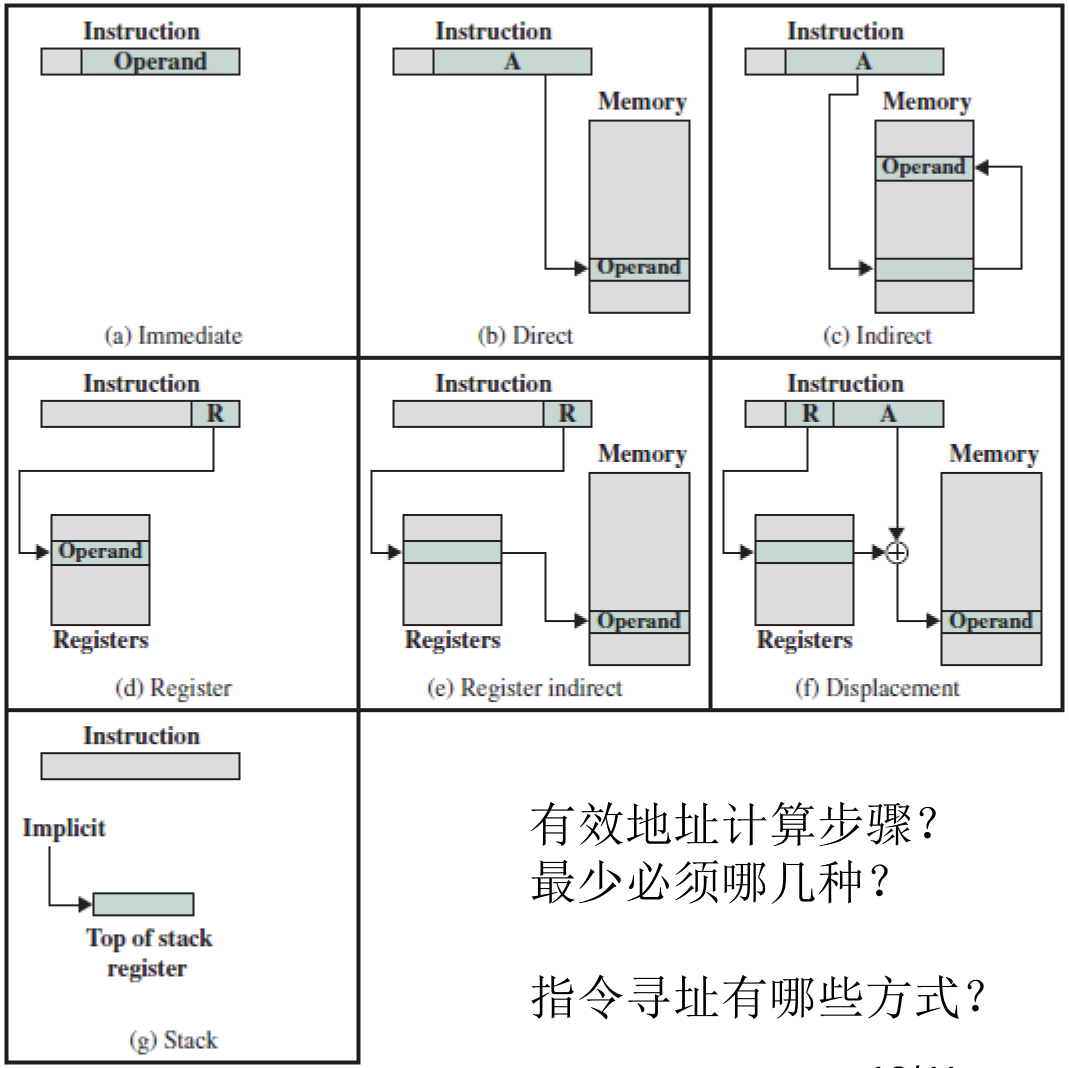
- 寻址方式：指令字和操作数的存储地址计算方式
- 指令寻址：现代CPU利用PC
  - 顺序执行：每执行一条指令，PC自动1
  - 跳转：更新PC，转移到目的地址执行
- 操作数寻址
  - 指令中给出“形式地址”
  - 有效地址：操作数在寄存器/内存中的物理地址
    - EA＝寻址方式＋形式地址

操作码	形式地址



操作数寻址方式

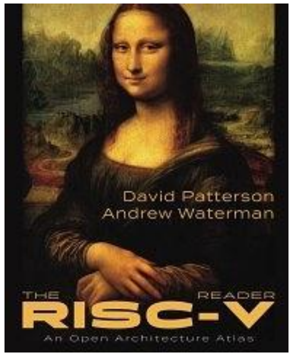
- 常见约10种
  - 立即寻址（a）
  - 直接寻址（b）
  - 间接寻址（c）
  - 寄存器寻址（d）
  - 寄存器间接寻址（e）
  - 基址寻址（f）
    - BP+offset
  - PC相对寻址（f）
    - PC+offsset
  - 堆栈寻址（g）
  - *变址寻址（d+f）*
    - Index：x86的si/di
  - *隐含寻址（如g）*



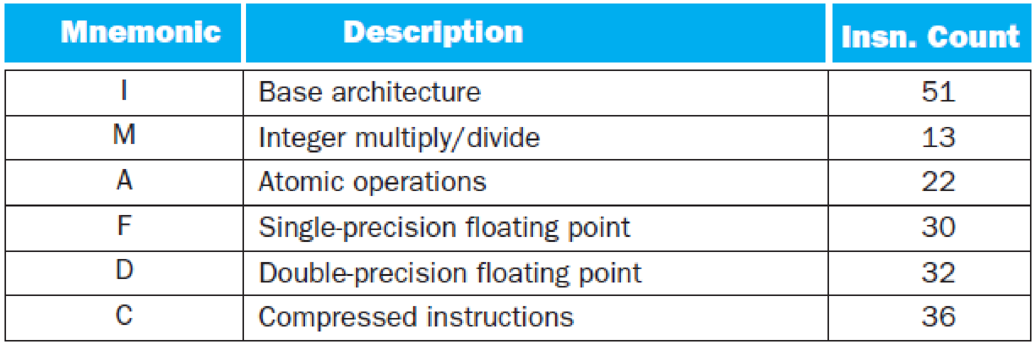
有效地址计算步骤？
最少必须哪几种？

指令寻址有哪些方式？

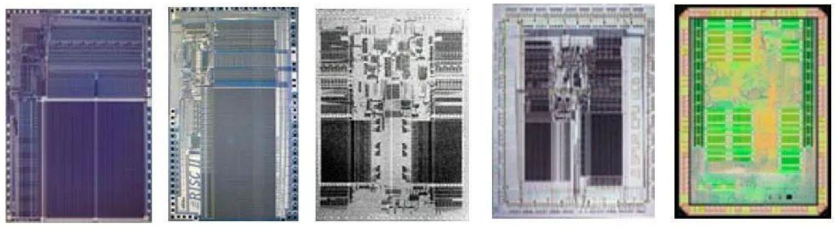
RISC-V ISA的特点



- 模块化：51+133+13=197条
  - RV32I指令集：支持完整软件栈，永远不变
    - 共51条：图2-1（37条，重点关注）+ 图2-37（14条）
  - RV32IMFD指令集： RV32I的基本扩展，图2-38，133条
  - 特殊指令：同步，CSR操作，异常。图5-47，13条
- 约束
  - 成本：芯片面积
  - 简洁
  - 性能：时间，功耗
  - 架构与实现分离
  - 扩展性：操作码域空间
  - 程序大小
  - 易于编程/编译/链接



Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36



David Patterson，Andrew Waterman，The RISC-V Reader: An Open Architecture Atlas，2017

RISC-V指令格式与寻址方式

图2-19

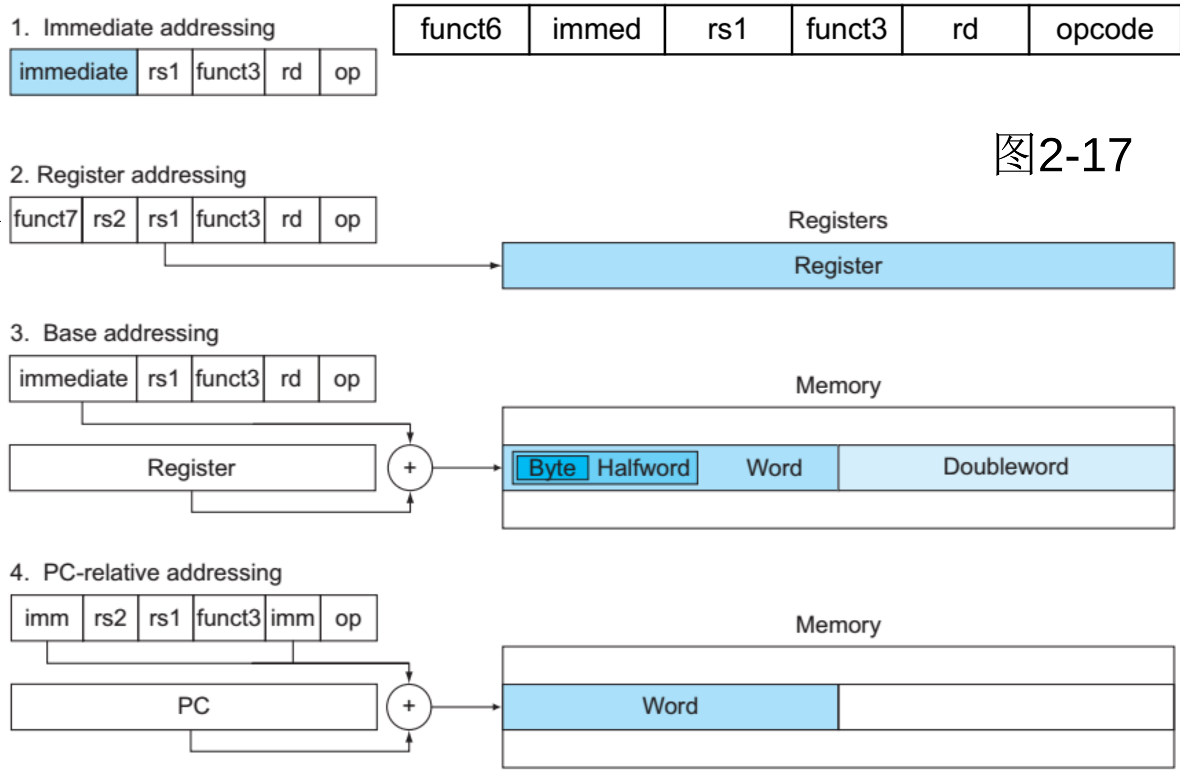
Name	Field						Comments
(Field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

funct6	immed	rs1	funct3	rd	opcode

- 指令格式：6种
  - 基本：R/I/S/U
    - 7种：4+2+“1”
    - IS-type：funct6，立即数移位
  - 规整：Reg和Imm位置固定
    - B/J-type的立即数域？
  - op码与类型绑定

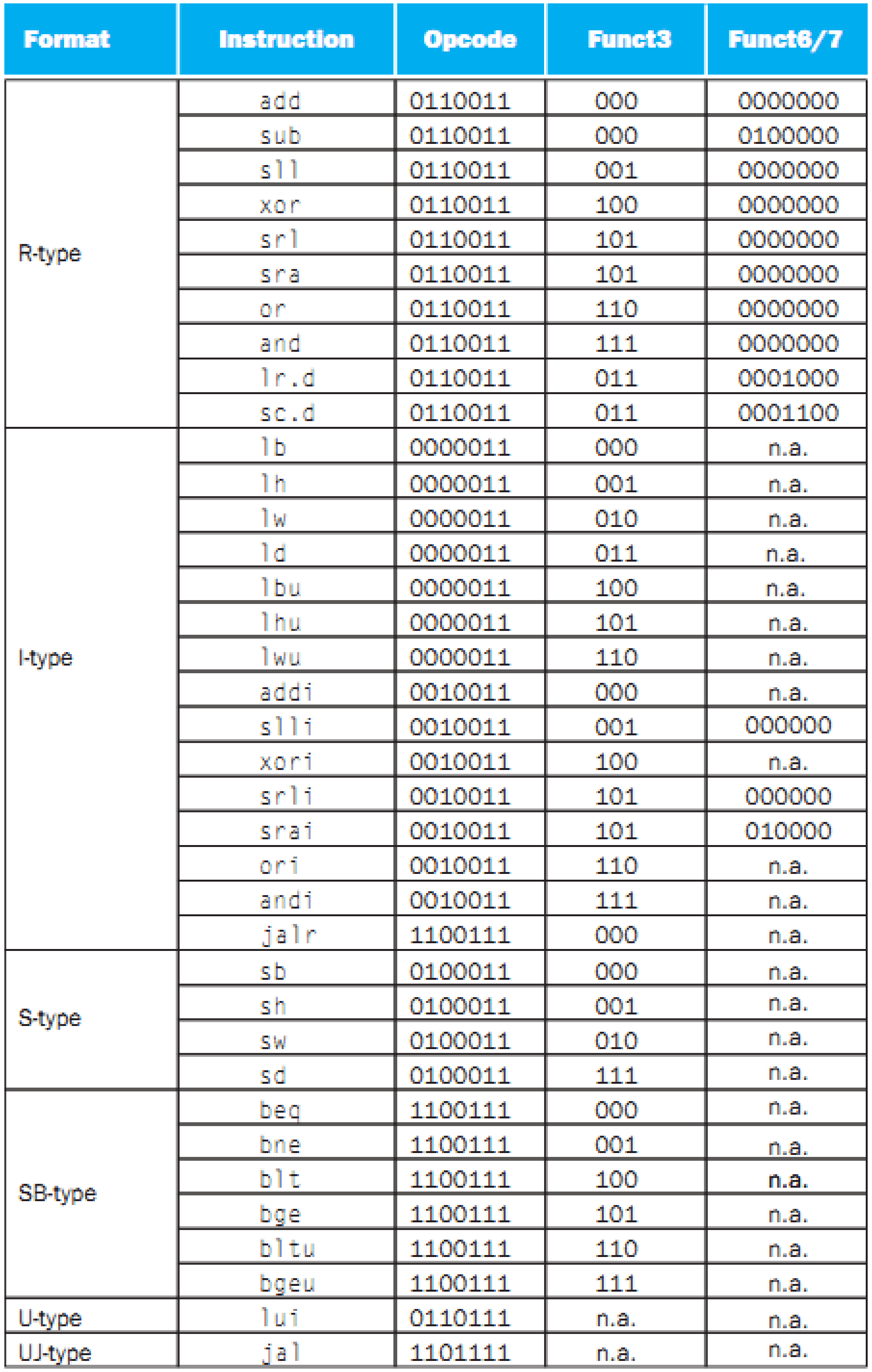
- 寻址方式：4种
  - 本质：Imm，Reg，Base
  - 指令寻址方式
    - PC相对寻址：beq，auipc
    - 间接跳转：jalr x0，100(x1)

图2-17



RV整数指令操作码

- 常用37条，图2-18
  - 按字长分类
    - b，w，d，h
  - 按数据类型分类
    - i，u
  - 按指令格式分类
- 典型：按功能分类
  - ALU
    - add：R-type
  - 访存
    - ld：load，I-type
    - sd：store，S-type
  - 分支
    - beq：SB-type
    - jal，jalr
- 总51条：图2-1;2-37;3-12



Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

RV示例：指令格式，寻址方式，图2-6

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
ld (load doubleword)	001111101000		00010	011	00001	0000011	ld x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1, 1000(x2)

- 汇编指令寻址方式表示
  - 寄存器寻址-名，立即数寻址-十进制/16进制，基址寻址-1000(rs1)
  - **算逻**指令均为寄存器寻址，load/store为基址寻址
- 机器指令与汇编指令中源操作数和目的操作数的位置对应关系
  - 汇编指令：x2/x3为源操作数rs1/rs2，x1为目的操作数rd
  - 注意S-type：x1=rs2（源），x2=rs1（基址），rs2 => mem[rs1+1000]

$zero：x0寄存器，$2.3.2

- x0固定为“0”
- data move：reg-reg

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

- 寄存器赋值

```
    addi  $v0,$zero,1 # return 1
```

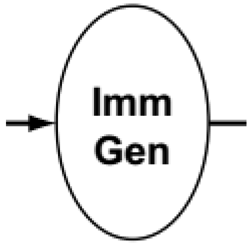
- Compare

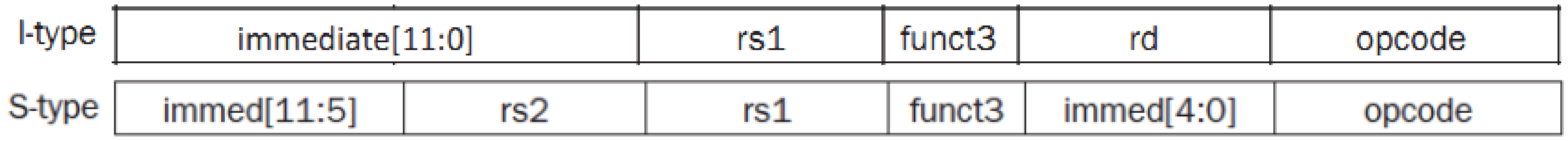
```
slti  $t0,$a0,1        # test for n < 1
beq   $t0,$zero,L1     # if n >= 1, go to L1
```

- Goto：beq x0，x0，Exit

位扩展：短立即数=>长立即数，$2.4

- 位扩展：从较小的数据类型转换成较大的类型
  - 无符号扩展（zero extension）：高位补0
    - 逻辑运算
  - 符号扩展（sign extension）：高位补1，补码
    - 算术运算，地址偏移
- 需求：I/S-type，短立即数12位=>32位
  - addi $s3,$s3,4；$s3 = $s3 + 4
  - lw $t1, offset($t2)；$t1=M[$t2+offset]
  - beq $1, $3, 7；if($1=$3)then taken, else not taken

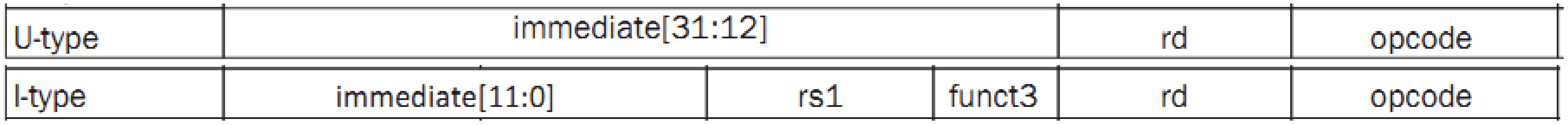




I-type	immediate[11:0]		rs1	funct3	rd	opcode
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode

生成32位常数，“双指令序列”法，$2.10.1

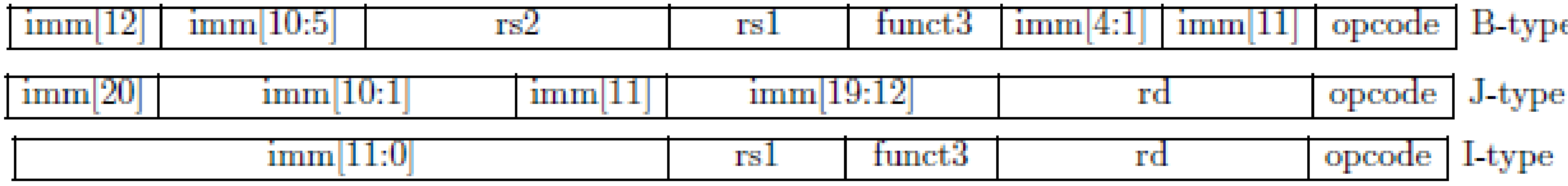
- 计算
  - 取20位立即数：取左移12位后的20位立即数
    - lui：load upper immediate，U-type
      - 加载20位立即数到[31:12]，符号扩展[63:32]=[31]，[11:0]=0
      - 例：lui x5，0x12345；x5=0x1234 5000
  - +低12位
    - addi：I-type
- 长跳转：寻址32位地址空间
  - lui：取高20位
    - 例： lui x5，0x12345；——$2.18的*auipc*？
  - jalr：jump & link reg，I-type
    - 高20位+低12位，——注意：间接跳转，非PC相对寻址！
    - 例：jalr x1，100(x5)；x1=PC+4，goto x5+100



U-type	immediate[31:12]			rd	opcode
I-type	immediate[11:0]	rs1	funct3	rd	opcode

转移指令的寻址方式$2.7，2.8，2.10.2

- 两类转移
  - 分支指令：if，while，case
    - 条件分支：beq rs1，rs2，L1；PC相对，12位offset
    - 无条件分支：可多种方式
      - jal x0，Label；J-type，PC相对，20位offset
      - jalr x0，100(x5)；I-type，基址，12位offset
      - beq x0，x0，Loop；B-type
  - 过程调用：x1为返回地址
    - Calling：jal x1, ProcedureAddress；NPC=>ra，转
    - Return：jalr x0, 0(x1)； “间接跳转”，基于$x1而非PC
- 转移范围：near（12位，20位），far（32位）
  - 远程转移：32位（lui高20位，jalr低12位）



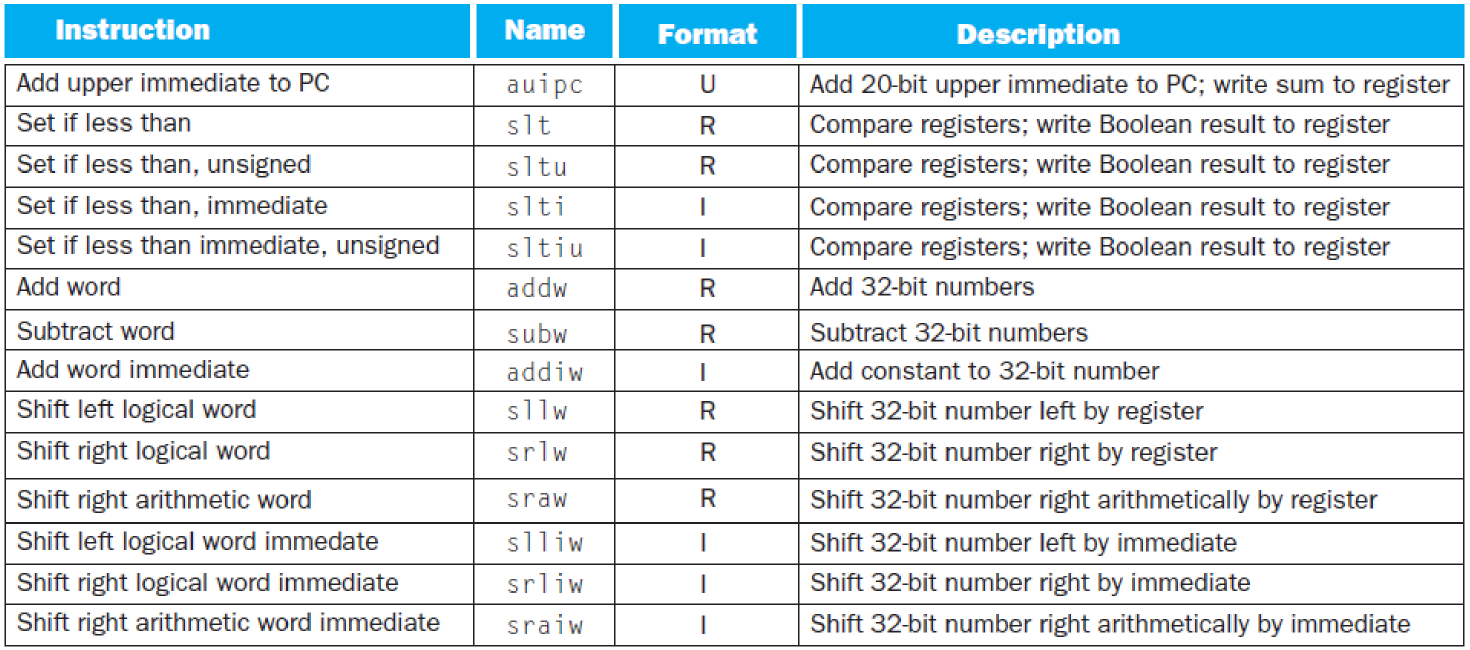
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

imm[11:0]	rs1	funct3	rd	opcode	I-type

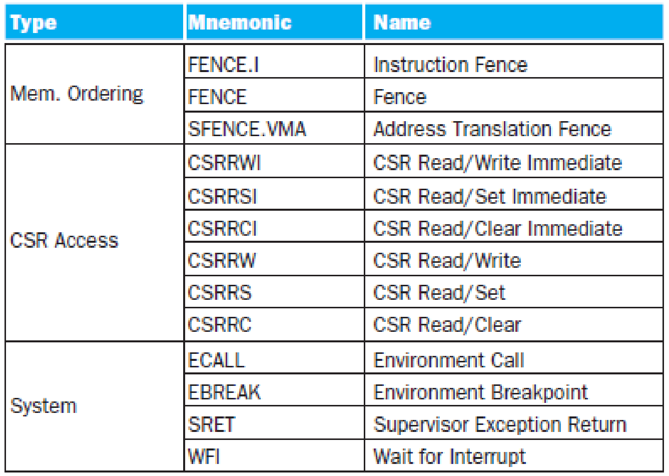
其他RV指令（了解），$2.18，$5.14

- 图2-37



Instruction	Name	Format	Description
Add upper immediate to PC	auipc	U	Add 20-bit upper immediate to PC; write sum to register
Set if less than	slt	R	Compare registers; write Boolean result to register
Set if less than, unsigned	sltu	R	Compare registers; write Boolean result to register
Set if less than, immediate	slti	I	Compare registers; write Boolean result to register
Set if less than immediate, unsigned	sltiu	I	Compare registers; write Boolean result to register
Add word	addw	R	Add 32-bit numbers
Subtract word	subw	R	Subtract 32-bit numbers
Add word immediate	addiw	I	Add constant to 32-bit number
Shift left logical word	sllw	R	Shift 32-bit number left by register
Shift right logical word	srlw	R	Shift 32-bit number right by register
Shift right arithmetic word	sraw	R	Shift 32-bit number right arithmetically by register
Shift left logical word immedate	slliw	I	Shift 32-bit number left by immediate
Shift right logical word immediate	srliw	I	Shift 32-bit number right by immediate
Shift right arithmetic word immediate	sraiw	I	Shift 32-bit number right arithmetically by immediate

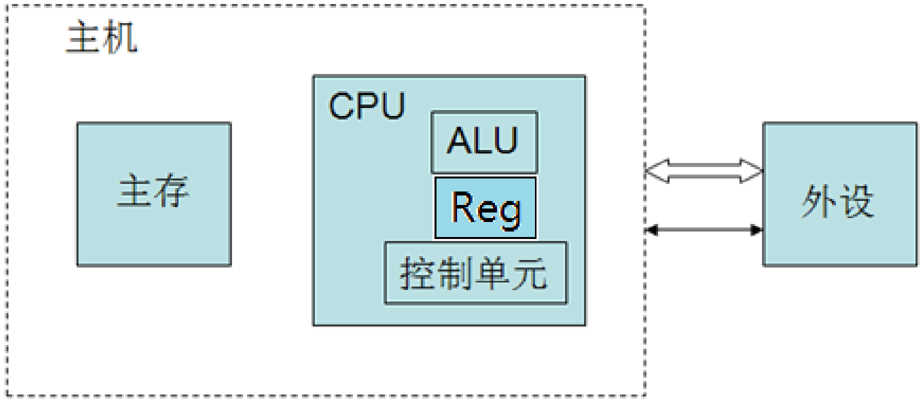
- 图5-47
  - 同步，CSR访问，系统调用
  - CSR：控制和状态寄存器

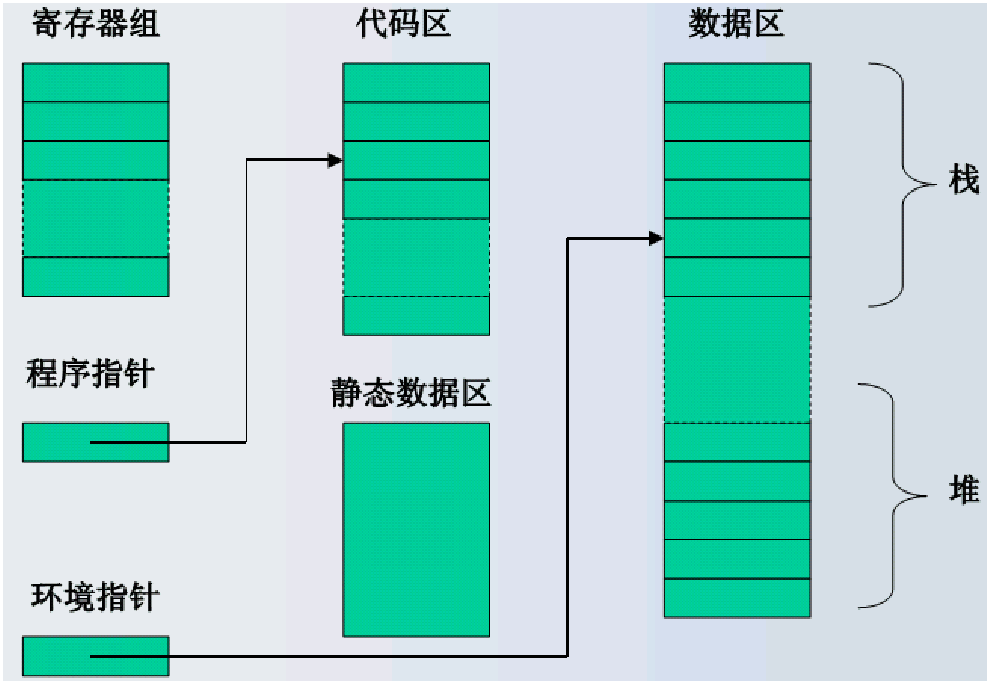


Type	Mnemonic	Name
Mem. Ordering	FENCE.I	Instruction Fence
	FENCE	Fence
	SFENCE.VMA	Address Translation Fence
CSR Access	CSRRWI	CSR Read/Write Immediate
	CSRRSI	CSR Read/Set Immediate
	CSRRCI	CSR Read/Clear Immediate
	CSRRW	CSR Read/Write
	CSRRS	CSR Read/Set
	CSRRC	CSR Read/Clear
System	ECALL	Environment Call
	EBREAK	Environment Breakpoint
	SRET	Supervisor Exception Return
	WFI	Wait for Interrupt

汇编语言程序设计要点：显式与约定

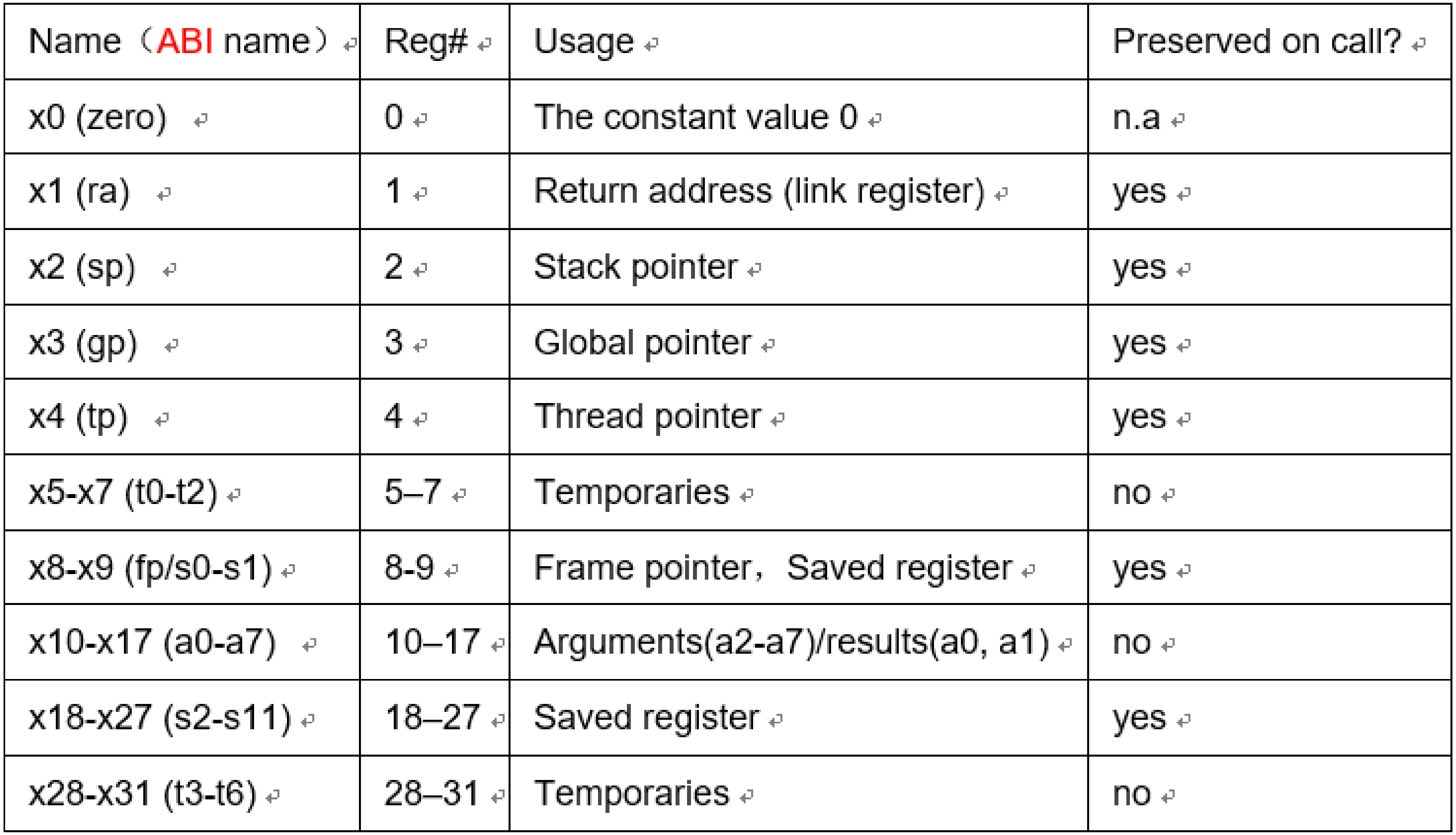
- 机器模型：对程序员显式可见
- ISA
  - 指令集
    - Move，ALU，分支，I/O
    - 整数指令，*浮点指令*，伪指令
  - 寻址方式：操作数，目标指令
  - 寄存器使用约定
  - 内存分配：数据、代码、堆栈
- 程序结构
- 过程调用/系统调用约定
  - 堆栈，栈帧
- 可执行程序生成





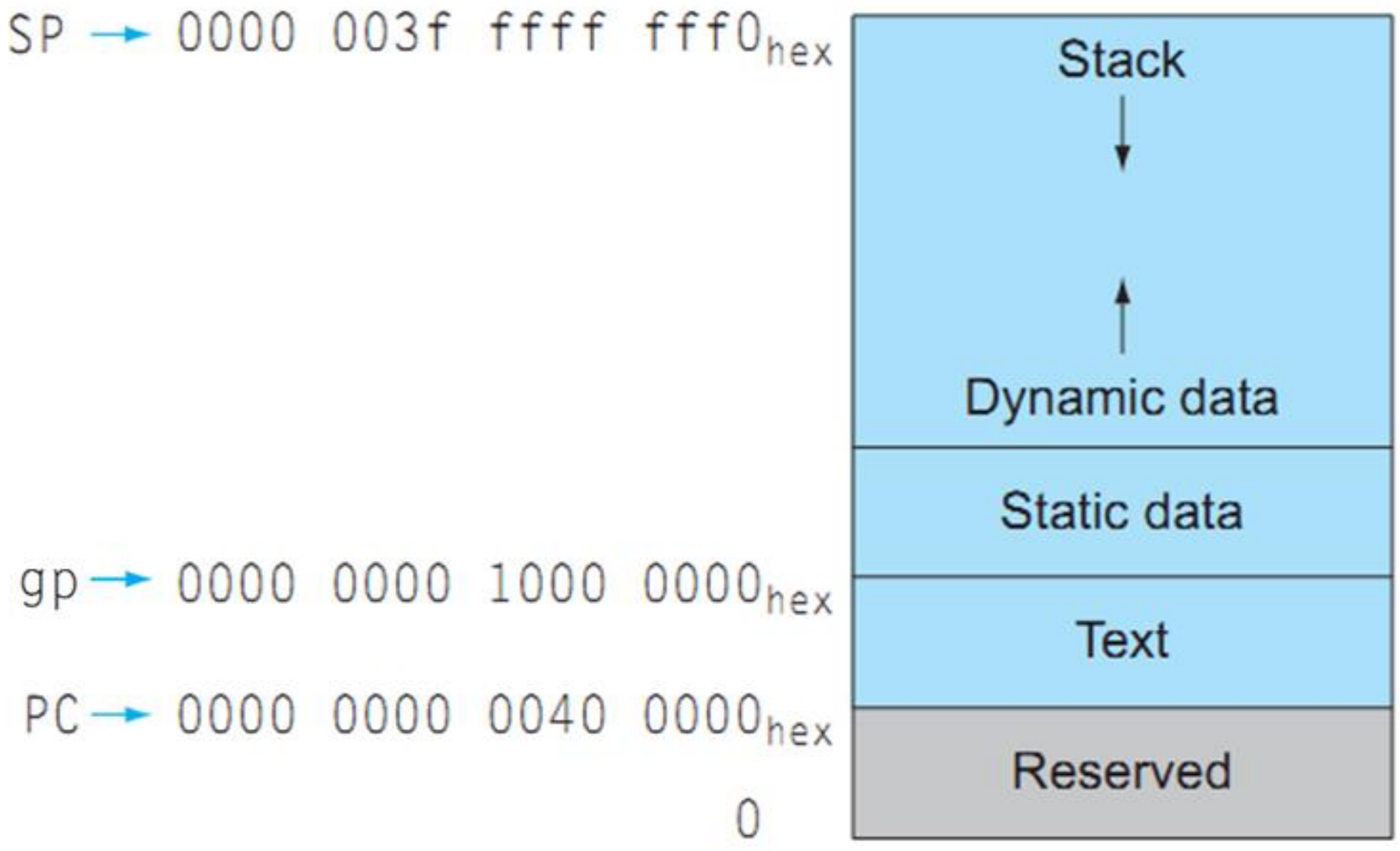
Policy of Use Conventions for registers

RV64/RV32，图2-14（注意：寄存器别名）



Name（ABI name）	Reg#	Usage	Preserved on call?
x0 (zero)	0	The constant value 0	n.a
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7 (t0-t2)	5–7	Temporaries	no
x8-x9 (fp/s0-s1)	8-9	Frame pointer，Saved register	yes
x10-x17 (a0-a7)	10–17	Arguments(a2-a7)/results(a0, a1)	no
x18-x27 (s2-s11)	18–27	Saved register	yes
x28-x31 (t3-t6)	28–31	Temporaries	no

RV内存分配约定，图2-13

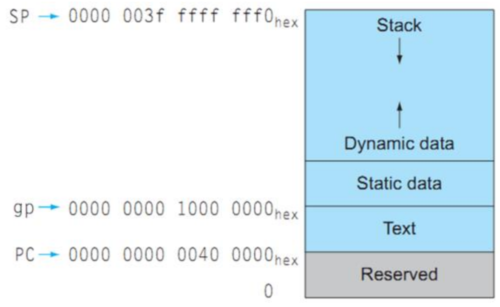

SP → 0000 003f ffff fff0hex
gp → 0000 0000 1000 0000hex
PC → 0000 0000 0040 0000hex
0
Stack
Dynamic data
Static data
Text
Reserved

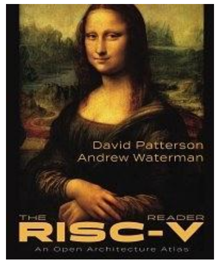
P&W中，RV32I的pc从0x0001 0000开始

RV汇编程序结构：《P&W》例

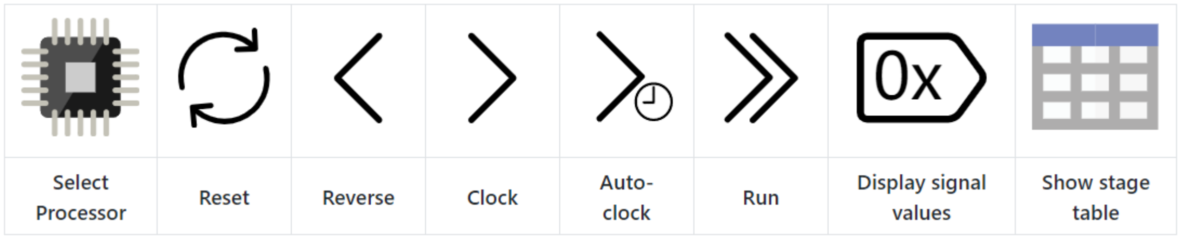
```
   .text                         # Directive: enter text section
   .align 2                      # Directive: align code to 2^2 bytes
   .globl main                   # Directive: declare global symbol main
main:                            # label for start of main
   addi sp,sp,-16                # allocate stack frame
   sw   ra,12(sp)                # save return address
   lui  a0,%hi(string1)          # compute address of
   addi a0,a0,%lo(string1)       #   string1
   lui  a1,%hi(string2)          # compute address of
   addi a1,a1,%lo(string2)       #   string2
   call printf                   # call function printf
   lw   ra,12(sp)                # restore return address
   addi sp,sp,16                 # deallocate stack frame
   li   a0,0                     # load return value 0
   ret                           # return
   .section .rodata              # Directive: enter read-only data section
   .balign 4                     # Directive: align data section to 4 bytes
string1:                         # label for first string
   .string "Hello, %s!\n"        # Directive: null-terminated string
string2:                         # label for second string
   .string "world"               # Directive: null-terminated string
```

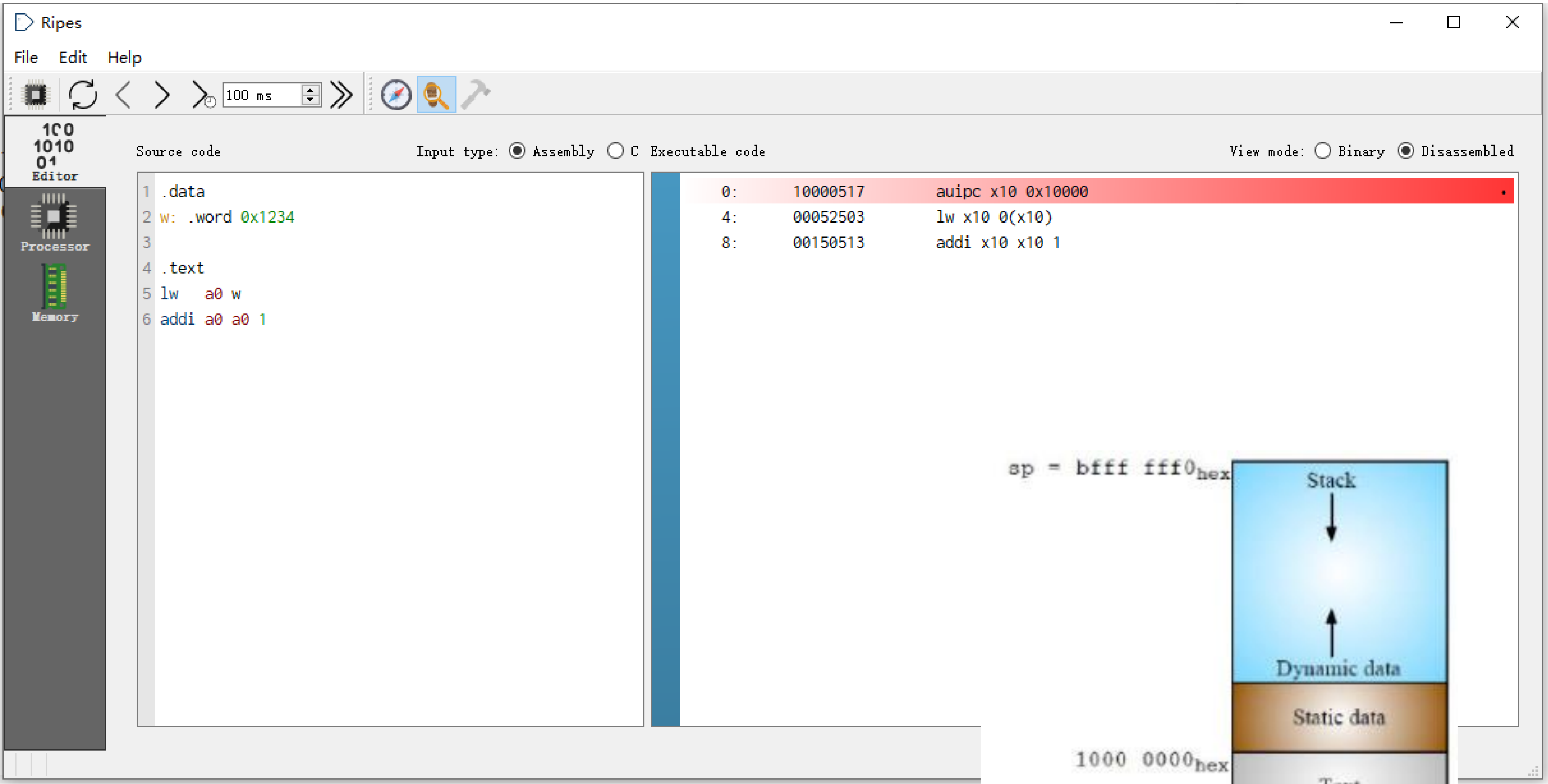
```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```


SP → 0000 003f ffff fff0hex
Stack
Dynamic data
Static data
gp → 0000 0000 1000 0000hex
Text
PC → 0000 0000 0040 0000hex
Reserved
0



Ripes汇编



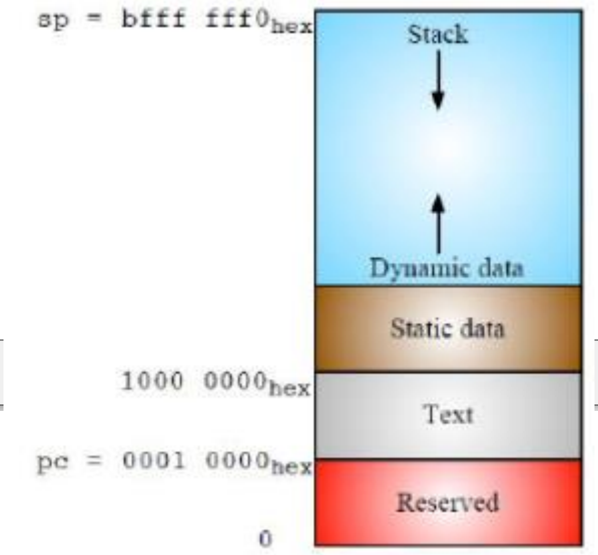


Source code   Input type: [✓ Assembly] C   Executable code   View mode: Binary [✓ Disassembled]

```
.data
w:  .word 0x1234

.text
lw   a0 w
addi a0 a0 1
```

```
0:    10000517    auipc x10 0x10000
4:    00052503    lw x10 0(x10)
8:    00150513    addi x10 x10 1
```

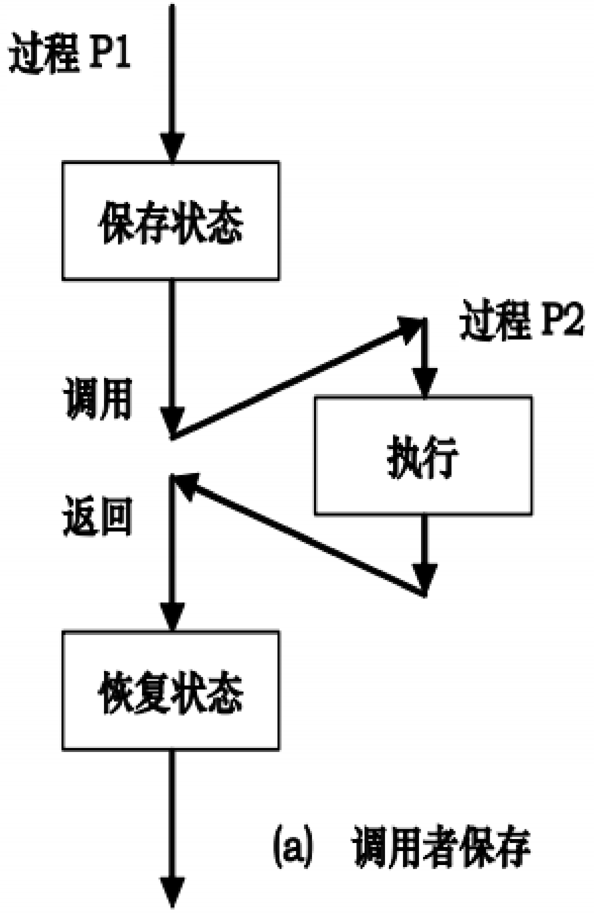


- Ripes汇编语言提供syscalls
- $2.8.2例“阶乘”见“factorial”！

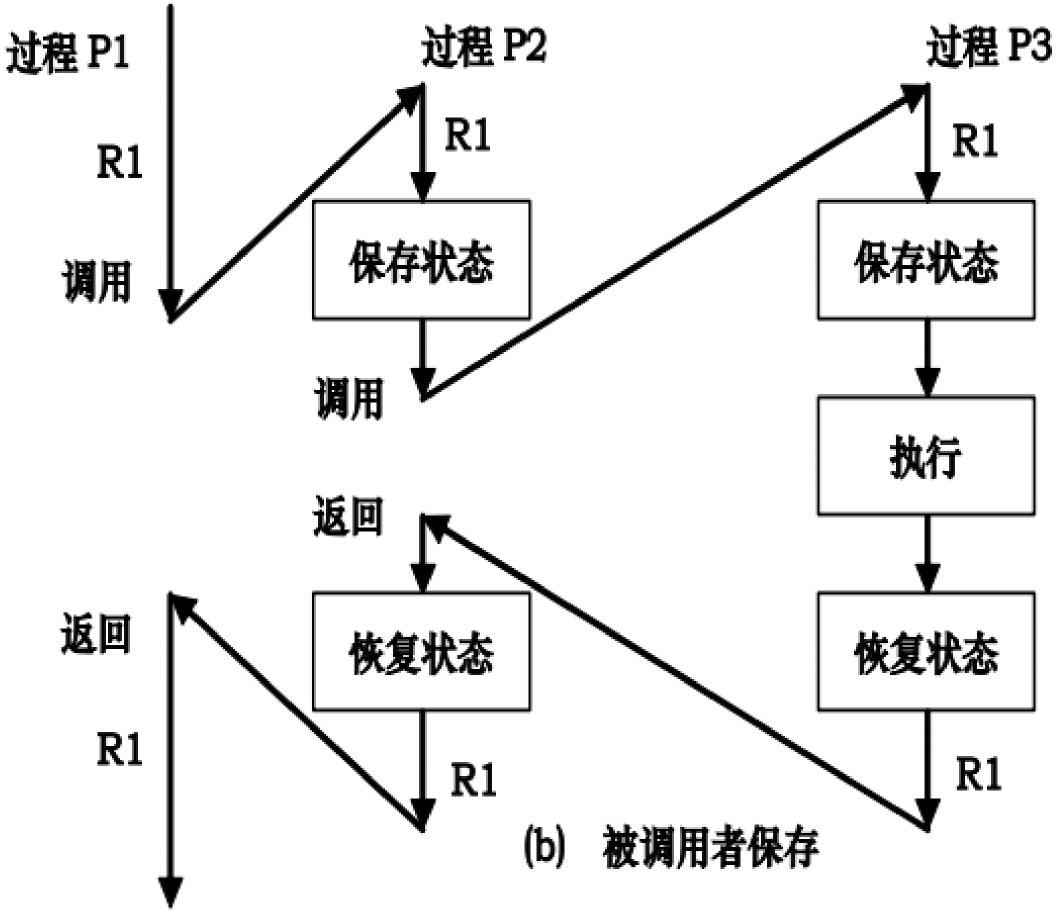
过程调用：现场保存，参数传递，$2.8

```
int leaf (int g, int h, int i, int j)
{           int f;
            f = (g + h) − (i + j);
            return f;
}
```

```
long long int fact (long long int n)
{           if (n < 1) return (1);
            else return (n * fact(n − 1));
}
```



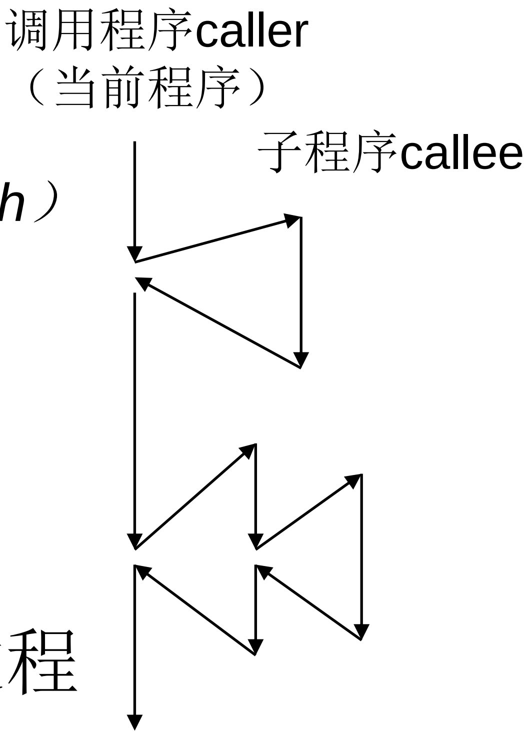
(a) 调用者保存



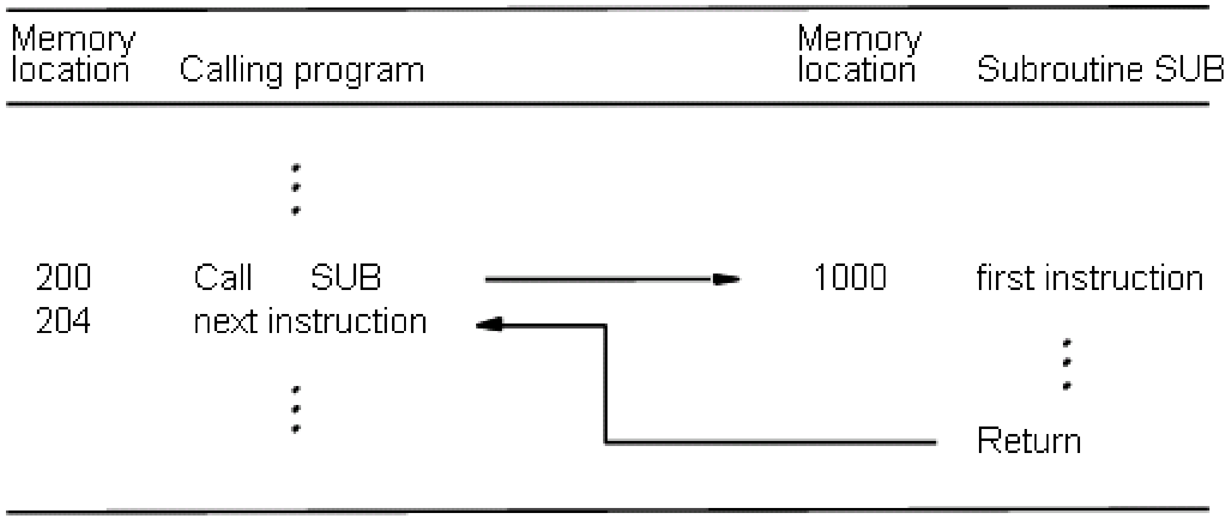
(b) 被调用者保存

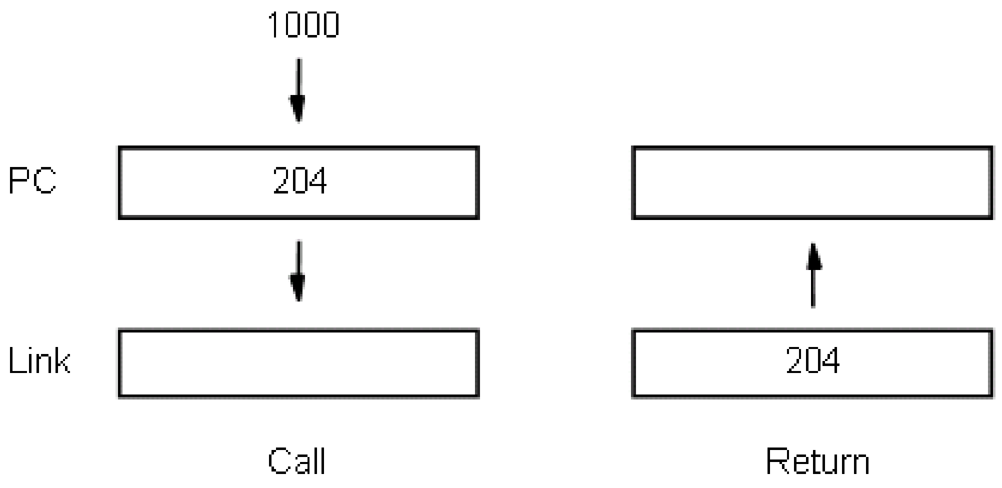
过程调用procedure calling

- 步骤
  - Caller
    - 参数传递：将参数放在子过程可访问的位置：寄存器/栈/内存
    - 控制转移：Call子过程
      - 保存返回点（断点，nPC）
      - 将控制交给子过程：使PC指向子过程入口
  - Callee
    - *保存现场：将过程内须使用的reg入栈（push）*
    - 计算，并将结果放在caller可以访问的位置
    - *恢复现场：出栈（pop）*
    - 子过程Return：返回Caller的返回点（断点）
      - 将控制交回调用程序：PC = nPC
- 控制转移指令：call/return
- 类型：叶子过程，嵌套过程，递归过程

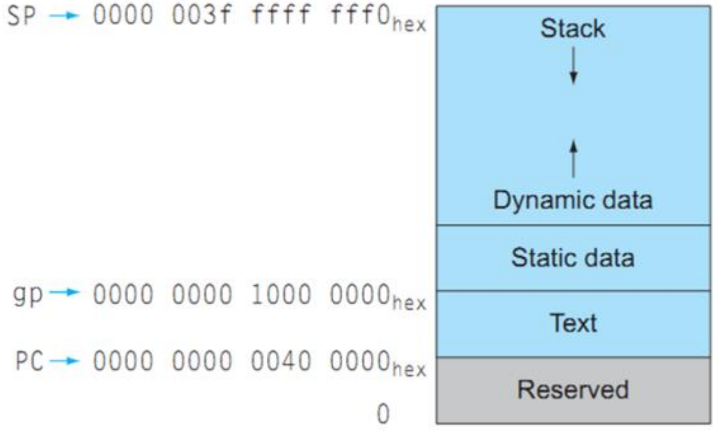


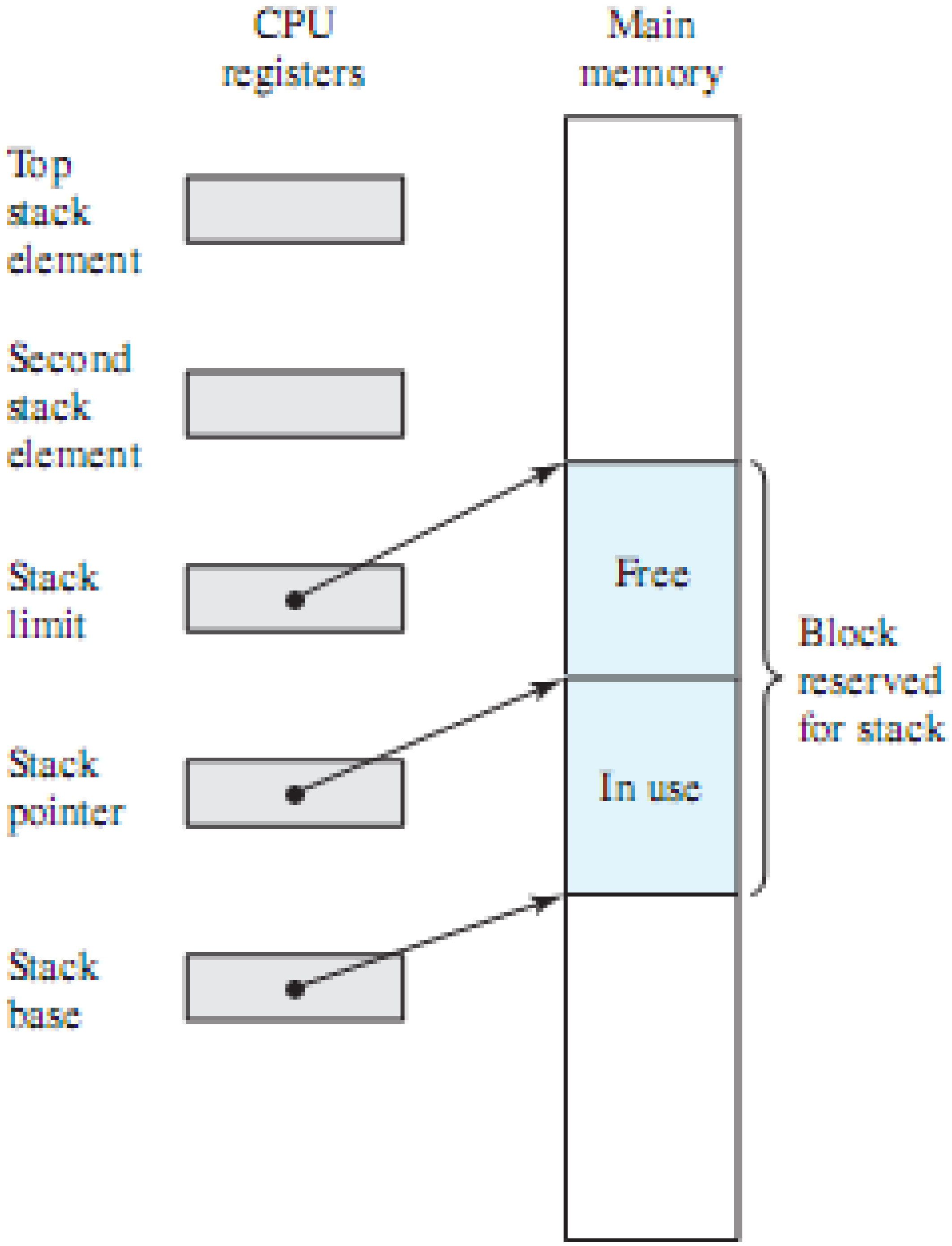
单级call/return：断点保存于链接寄存器ra/x1





stack

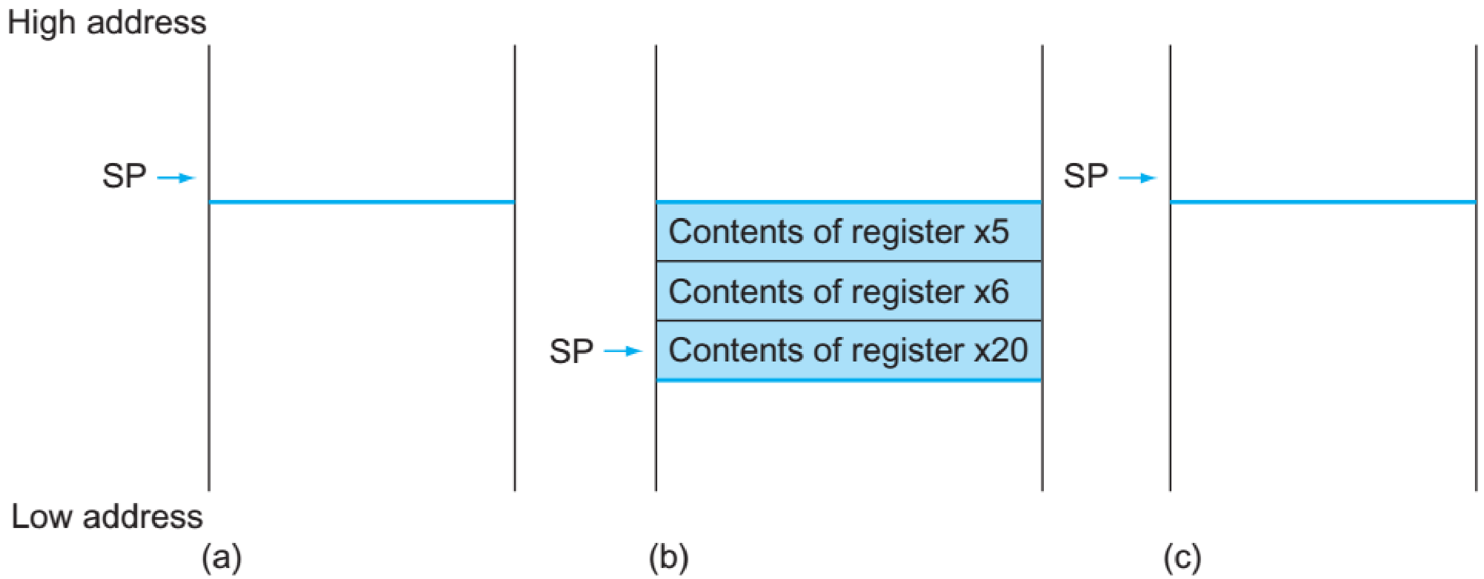

SP → 0000 003f ffff fff0hex
gp → 0000 0000 1000 0000hex
PC → 0000 0000 0040 0000hex
0
Stack
Dynamic data
Static data
Text
Reserved


CPU registers
Main memory
Top stack element
Second stack element
Stack limit
Stack pointer
Stack base
Free
In use
Block reserved for stack

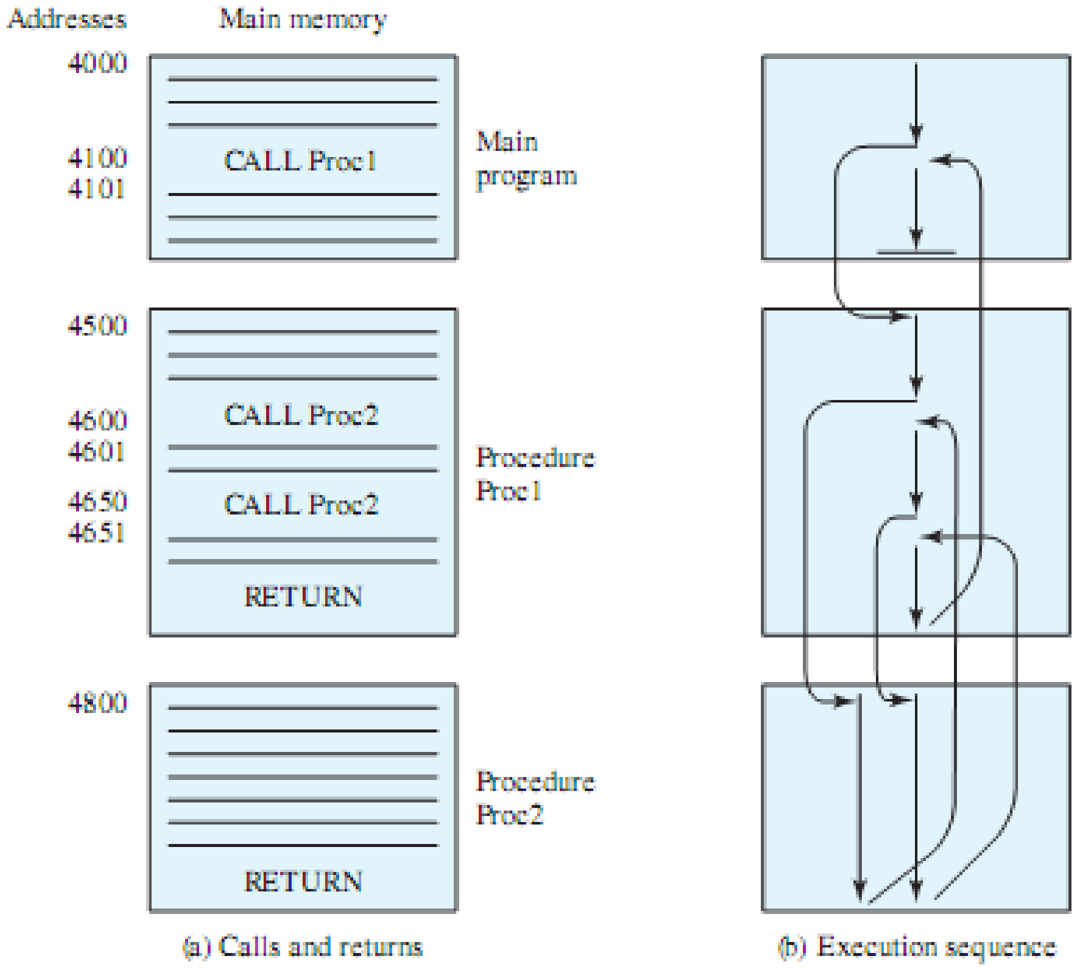
RV堆栈操作：push/pop，图2-10

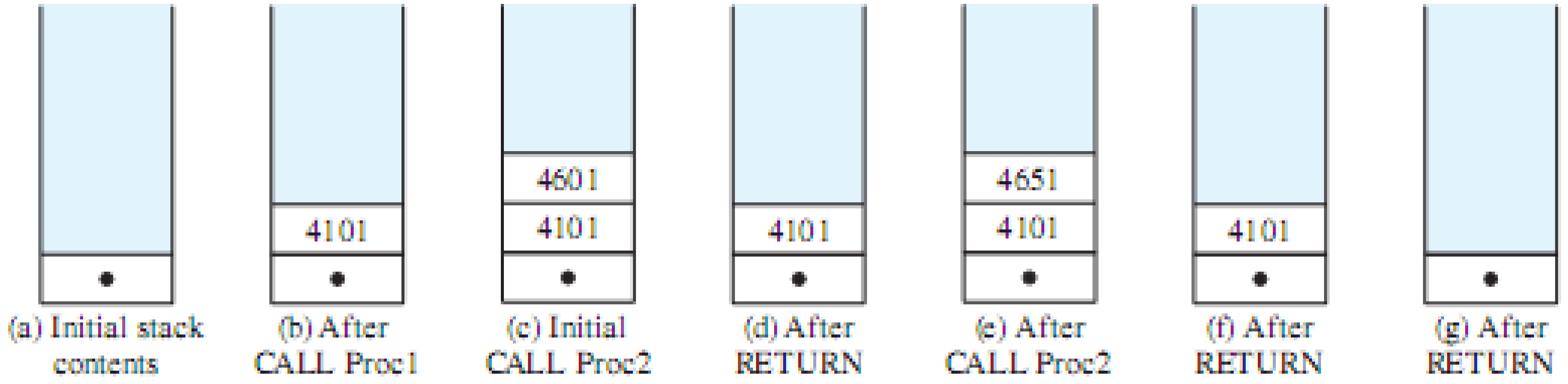
```
addi sp, sp, -24          // adjust stack to make room for 3 items
sd   x5, 16(sp)           // save register x5 for use afterwards
sd   x6, 8(sp)            // save register x6 for use afterwards
sd   x20, 0(sp)           // save register x20 for use afterwards
```

```
  ld x20, 0(sp)     // restore register x20 for caller
  ld x6, 8(sp)      // restore register x6 for caller
  ld x5, 16(sp)     // restore register x5 for caller
  addi sp, sp, 24  // adjust stack to delete 3 items
```



Use of Stack to Implement Nested Procedures

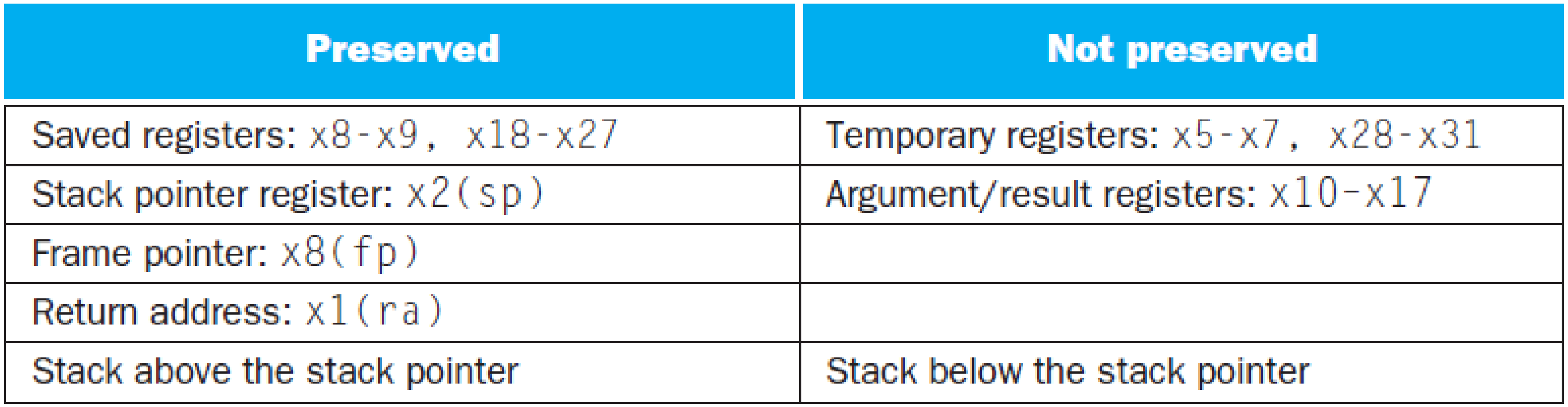




RV calling conventions

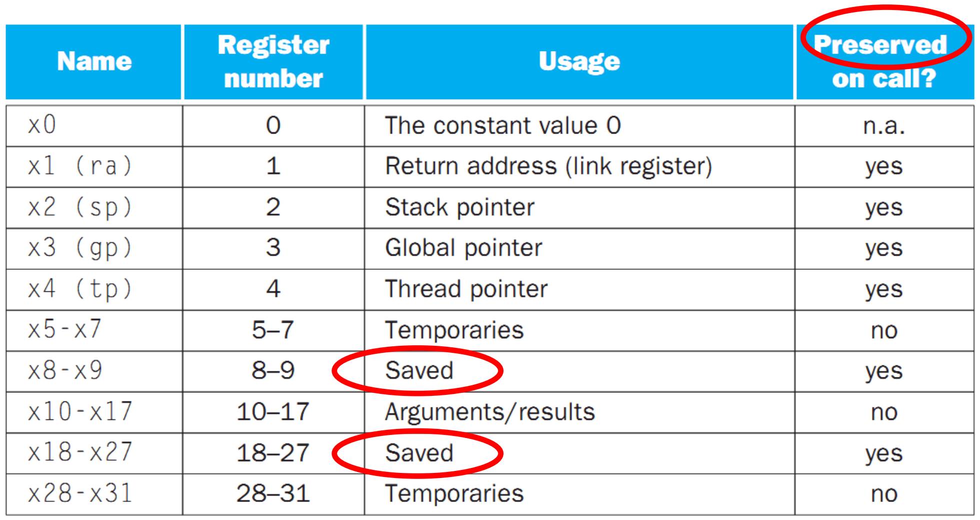
- for procedure calling
  - a2–a7：five argument registers in which to pass parameters
  - a0–a1: two value registers in which to return values
  - ra: one return address register to return to the point of origin
- call：jal x1，ProcessAddress； PC相对寻址
  - jump-and-link：跳转，并自动保存断点（nPC）至$ra
- return：jalr x0，0(ra)；间接跳转
  - jump register：返回ra
- 状态（现场）保存策略：callee负责

图2-11



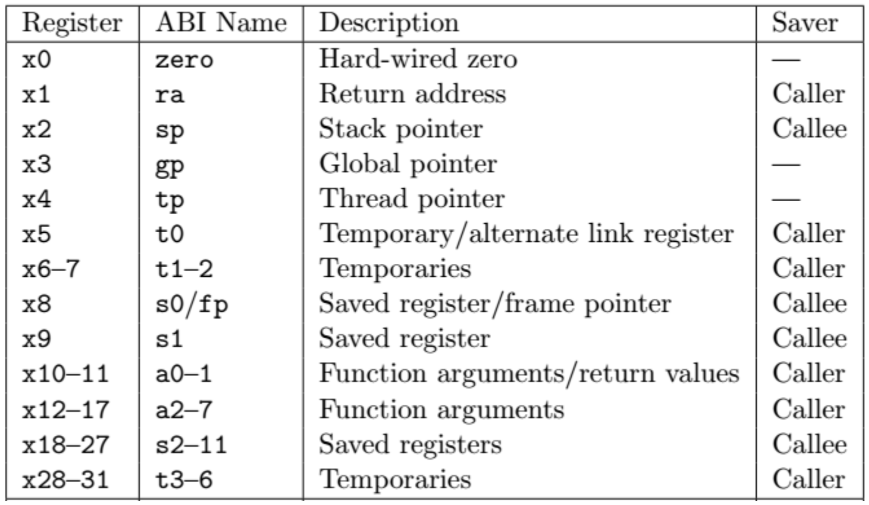
Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

RV Arch’ed Regs



Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

RV 图2-14



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Table 25.1

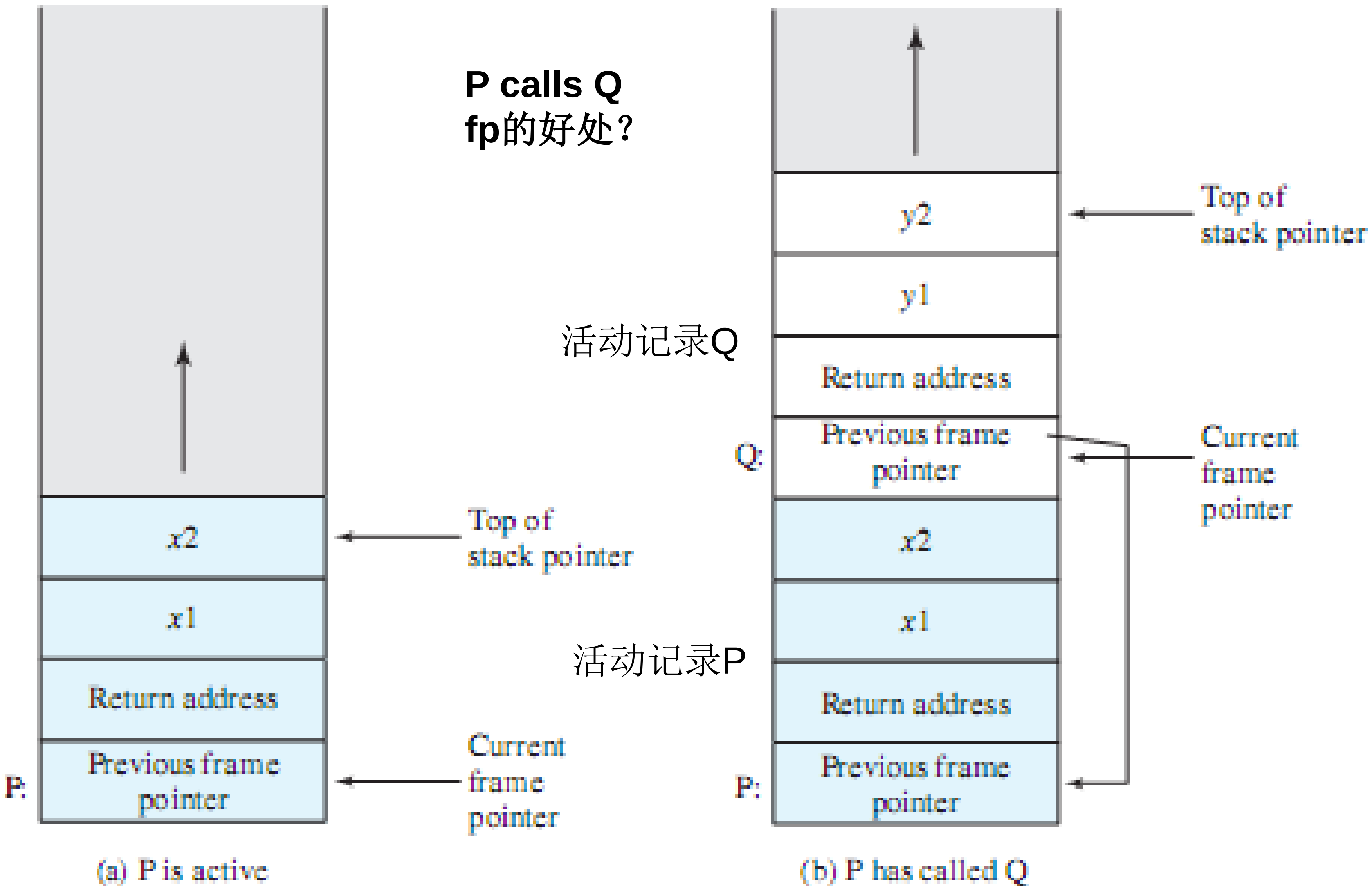
The RISC-V Instruction Set Manual Volume I

过程调用时可用a0~a7，s0~s11，t0~t6

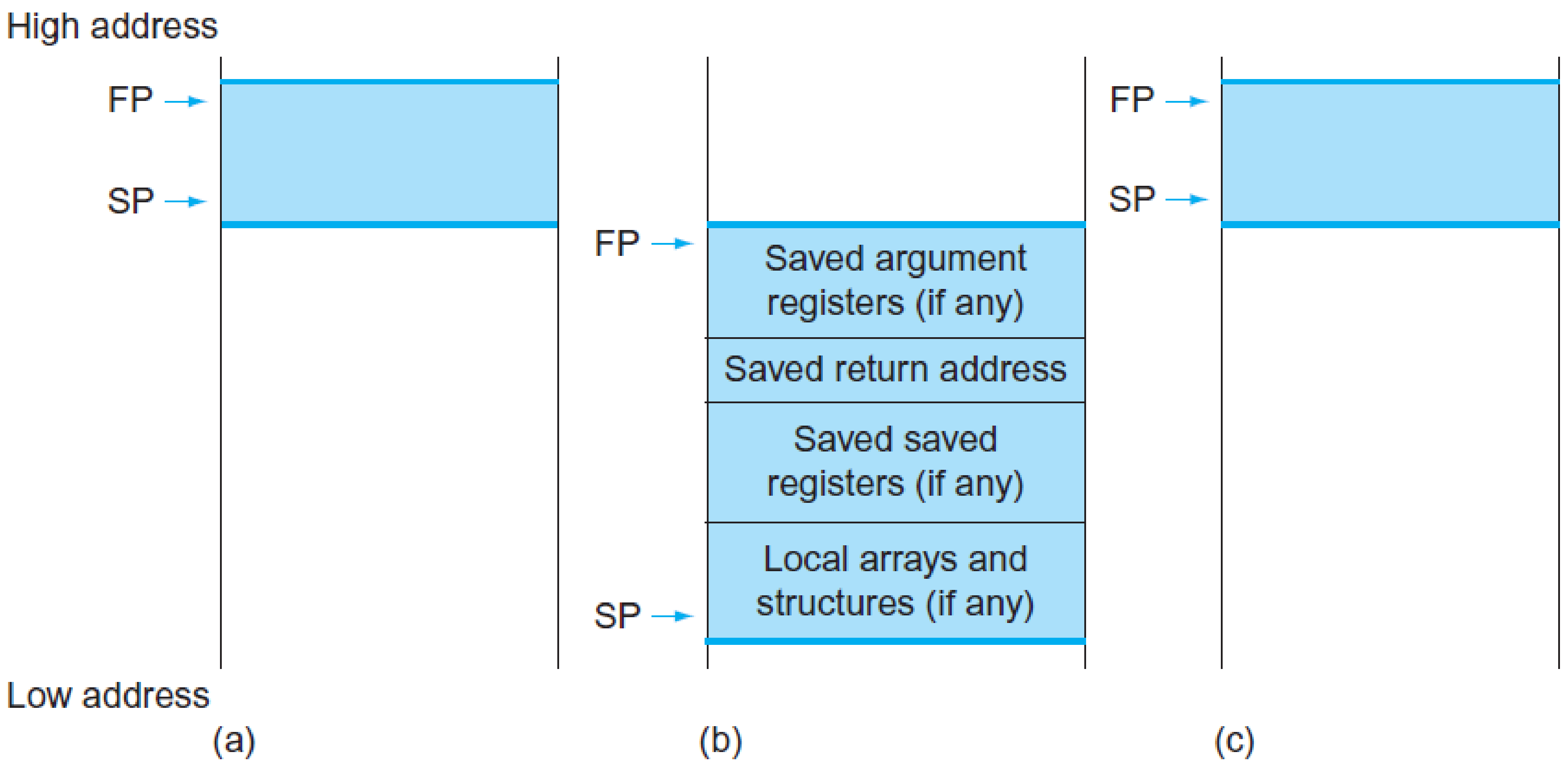
ABI calling convention：——减少代码量，提升性能
- **a/t**-registers are caller-saved
- **s**-regs are callee-saved and **preserve** their contents across function calls

stack frame：活动记录，帧指针fp

P calls Q
fp的好处？



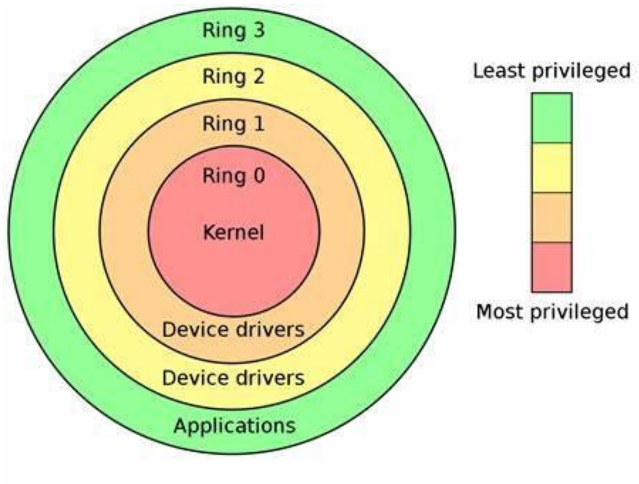
RV的过程帧（栈帧），图2-12



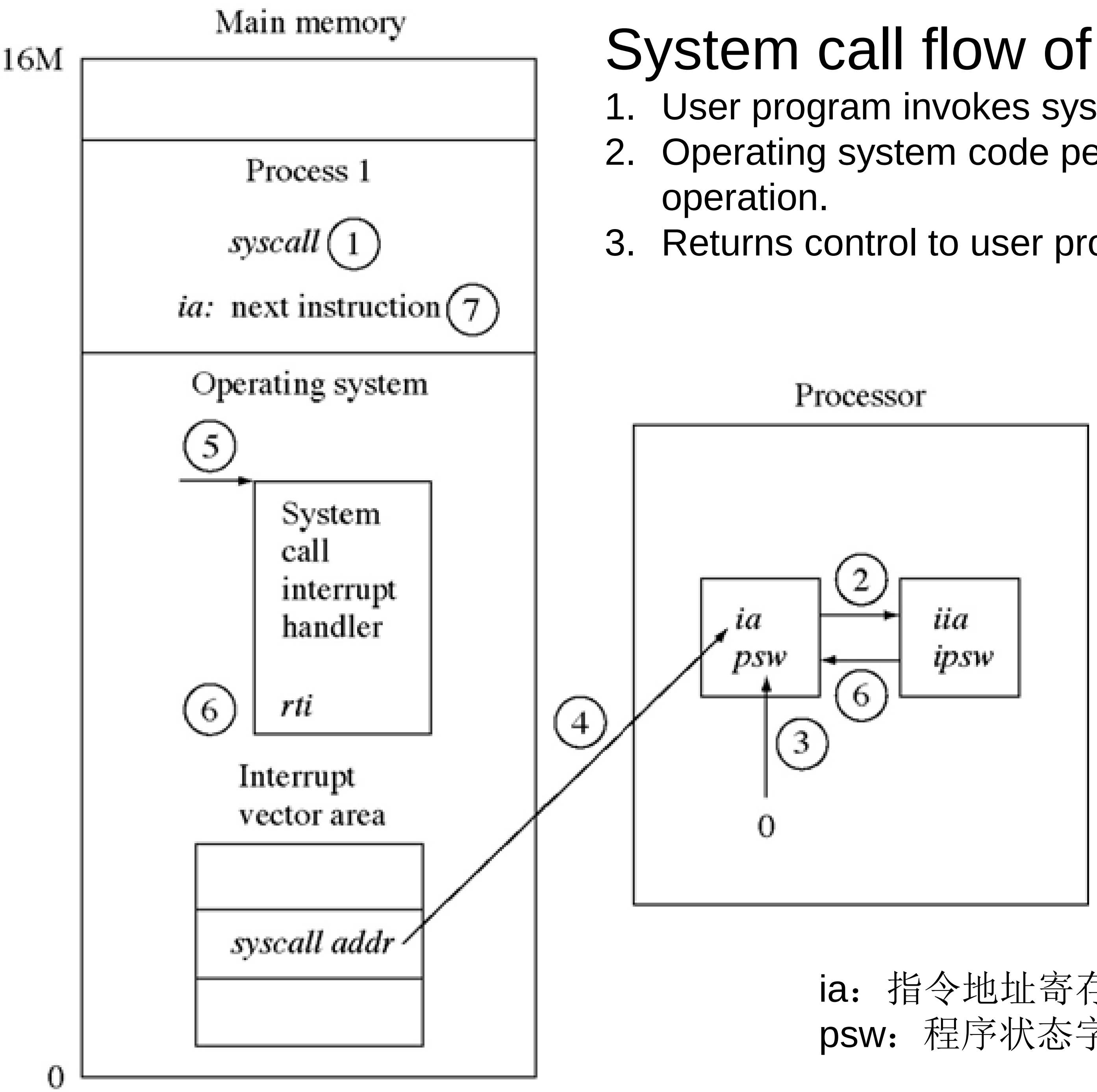
High address
FP
SP
FP
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
SP
FP
SP
Low address
(a)
(b)
(c)

System calls

- OS服务：various names
  - trap/exception, svc, soft interrupt
- Why：Certain operations require
  - specialized knowledge
    - I/O设备，PCIe总线，USB
  - protection：多任务共享
- What



  - A special machine instruction that causes an soft-interrupt/exception
    - *产生状态切换（protection）：需保存PSW*
  - RV：环境调用指令ecall。Ripes提供哪些服务?
  - x86系统调用（system calls）：int16，int32
    - BIOS，Windows：显示、键盘、磁盘、文件、打印机、时间



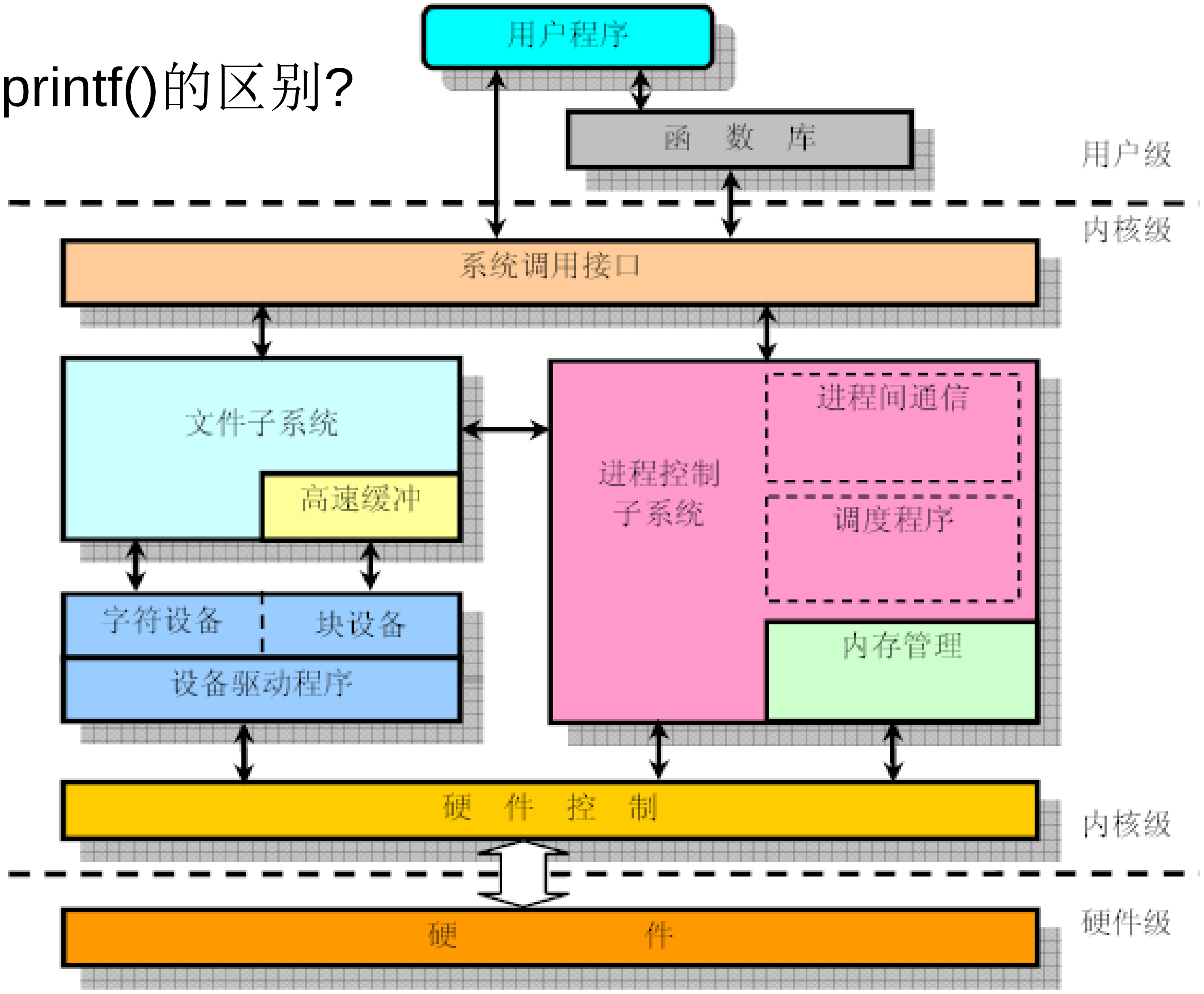
System call flow of control

1. User program invokes system call.
2. Operating system code performs operation.
3. Returns control to user program.

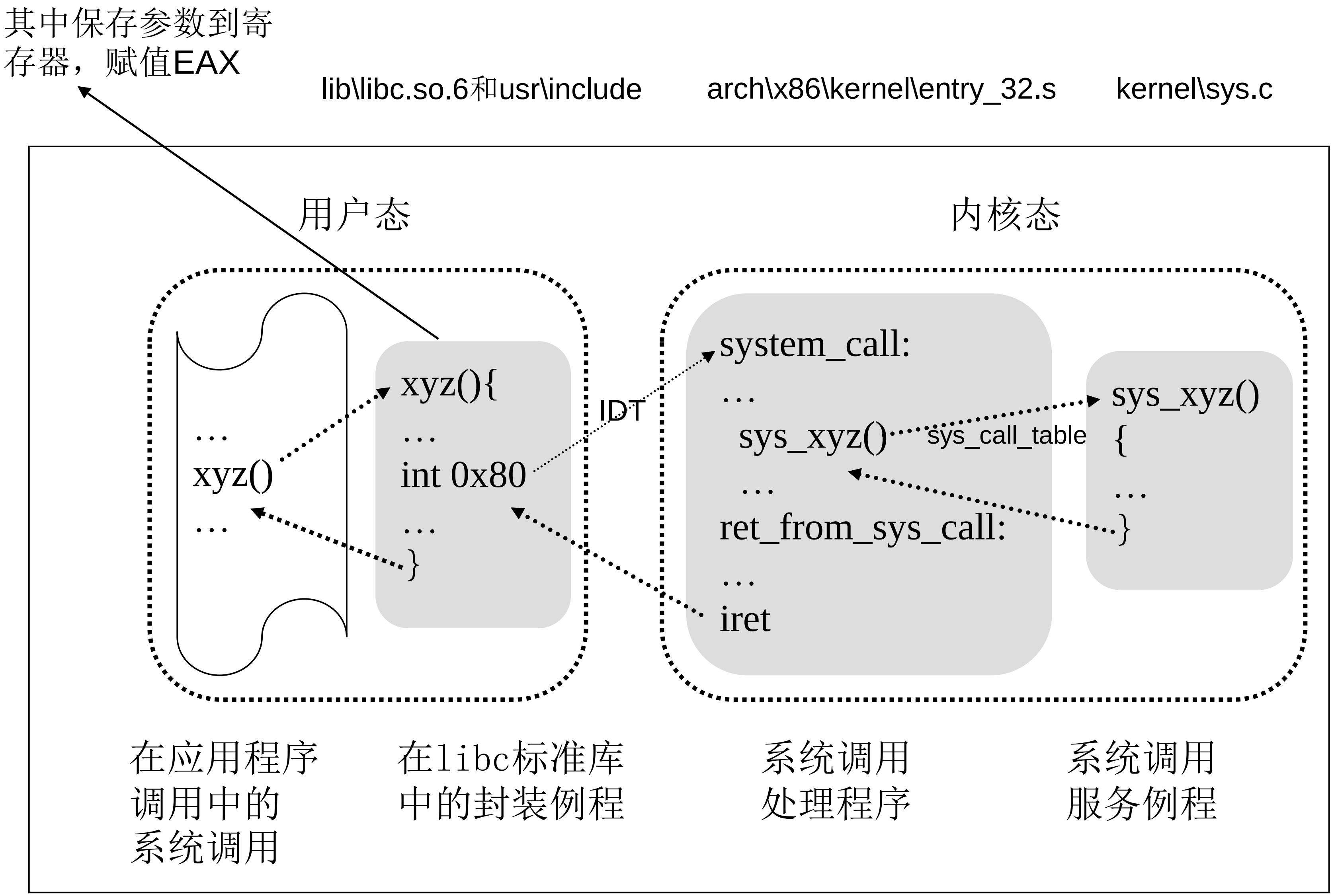
ia：指令地址寄存器
psw：程序状态字寄存器

C语言和OS服务：APIs、lib、syscalls?

abs()与printf()的区别?



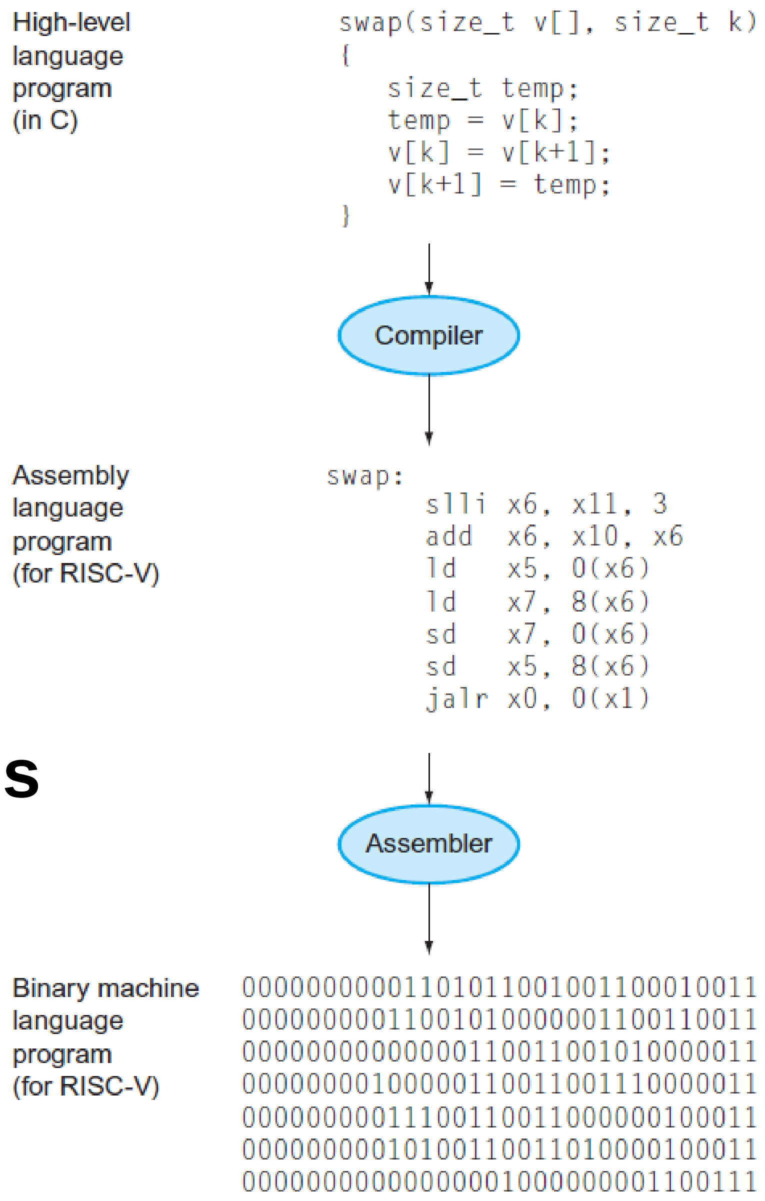
系统调用：x86调用门


其中保存参数到寄存器，赋值EAX
lib\libc.so.6和usr\include
arch\x86\kernel\entry_32.s
kernel\sys.c
用户态
内核态
…
xyz()
…
xyz(){
…
int 0x80
…
}
IDT
system_call:
…
sys_xyz()
…
ret_from_sys_call:
…
iret
sys_call_table
sys_xyz()
{
…
}
在应用程序调用中的系统调用
在libc标准库中的封装例程
系统调用处理程序
系统调用服务例程

可执行程序生成与执行

High Level to Assembly，图1-4

- **High Level Lang (C, etc.)**
  - Statements
  - Variables
  - Operators
  - func, proc, methods
- **Assembly Language**
  - Instructions
  - Registers
  - Memory segments/sections
- **Data Representation**
- **Number Systems**

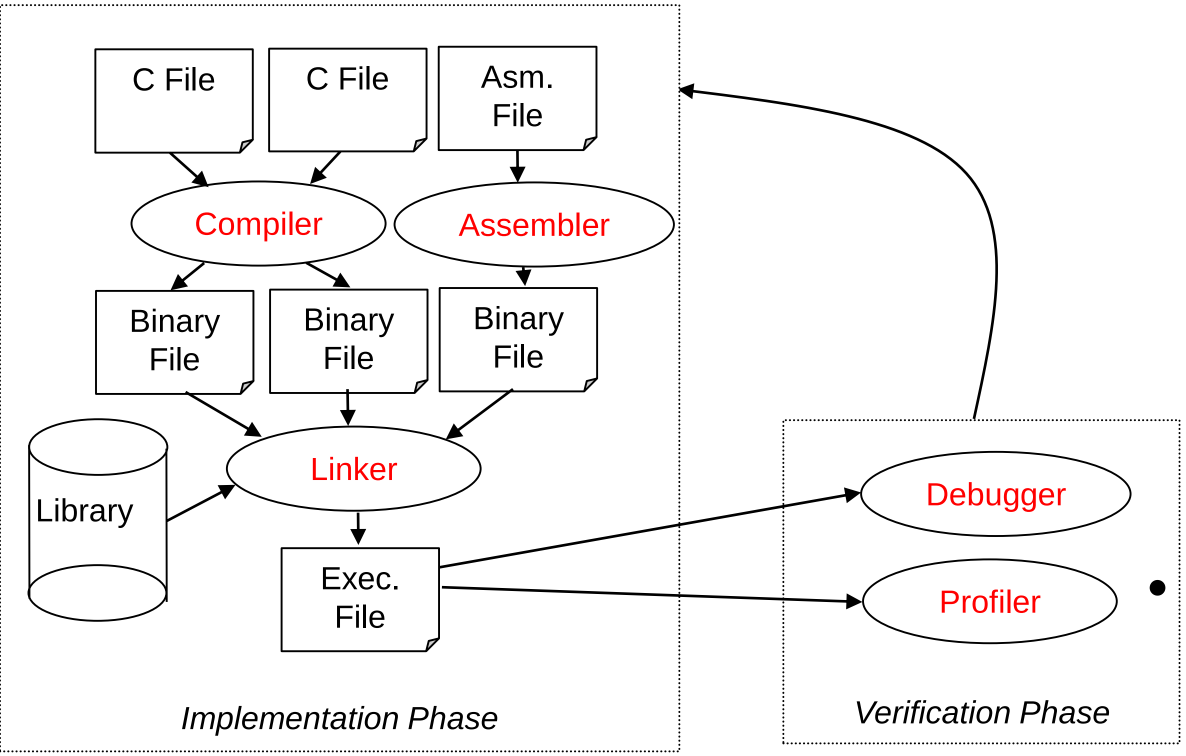

High-level language program (in C)
Compiler
Assembly language program (for RISC-V)
Assembler
Binary machine language program (for RISC-V)

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

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```

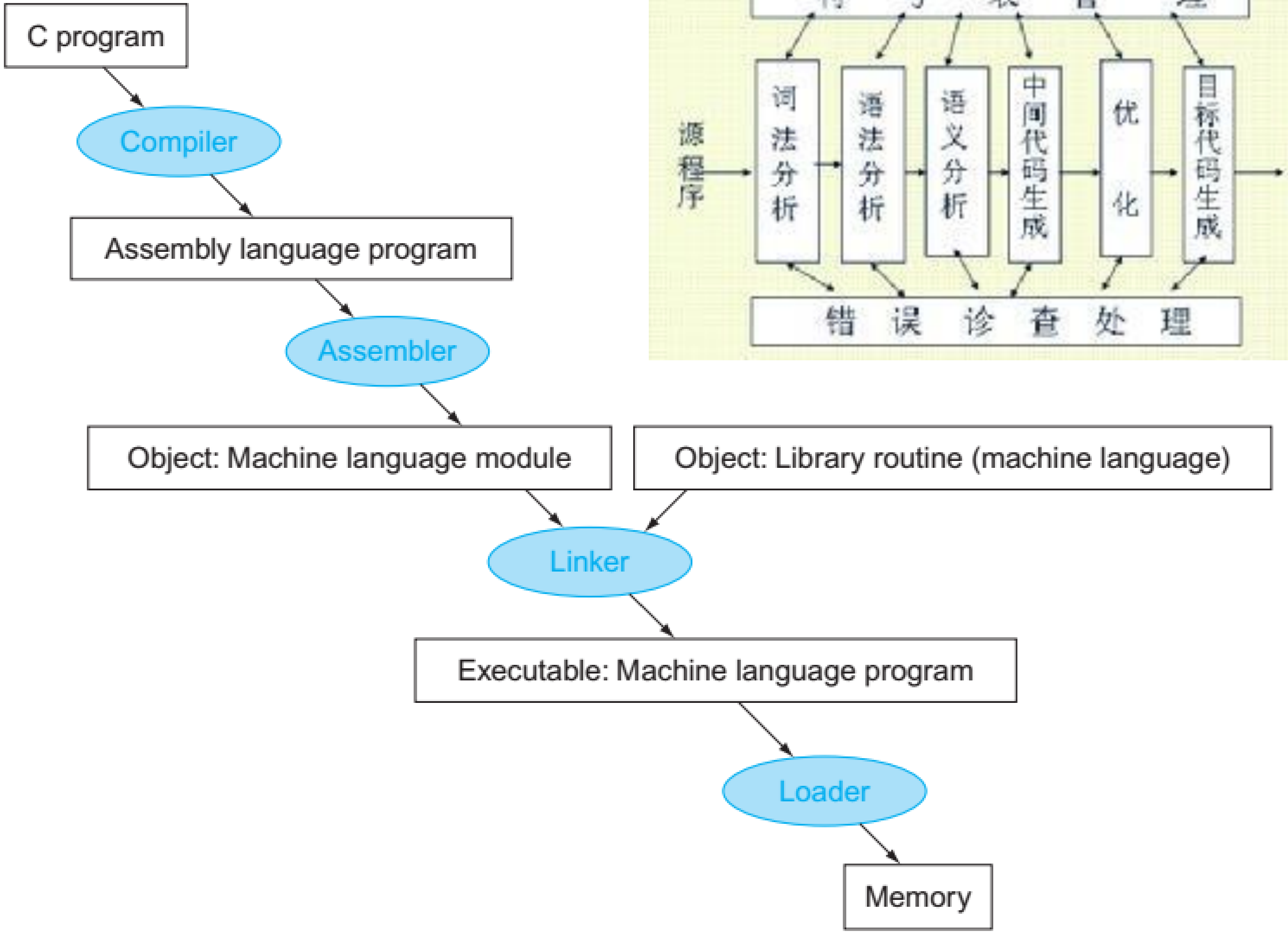
```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

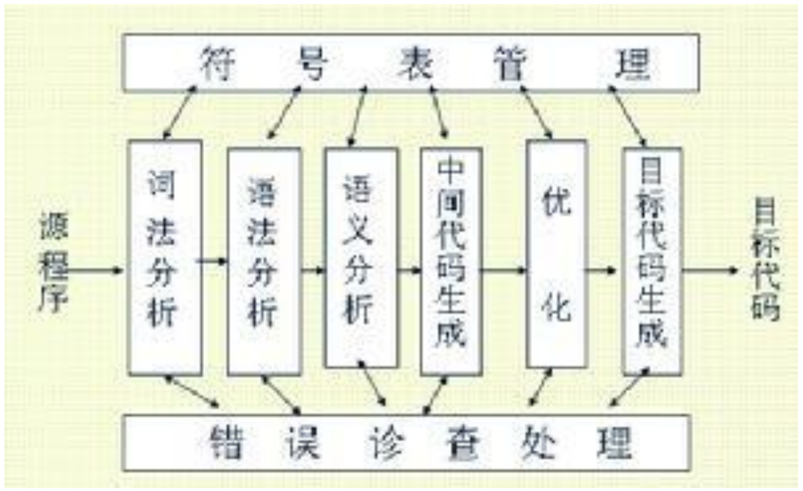
Program Development Process



- *Implementation Phase*
  - editor
  - Compilers
    - Cross compiler
      - Runs on one processor, but generates code for another
  - Assemblers
  - Linkers
- *Verification Phase*
  - Debuggers
  - Profilers

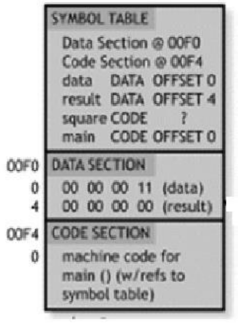
A translation hierarchy for C，FIG 2.20


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory



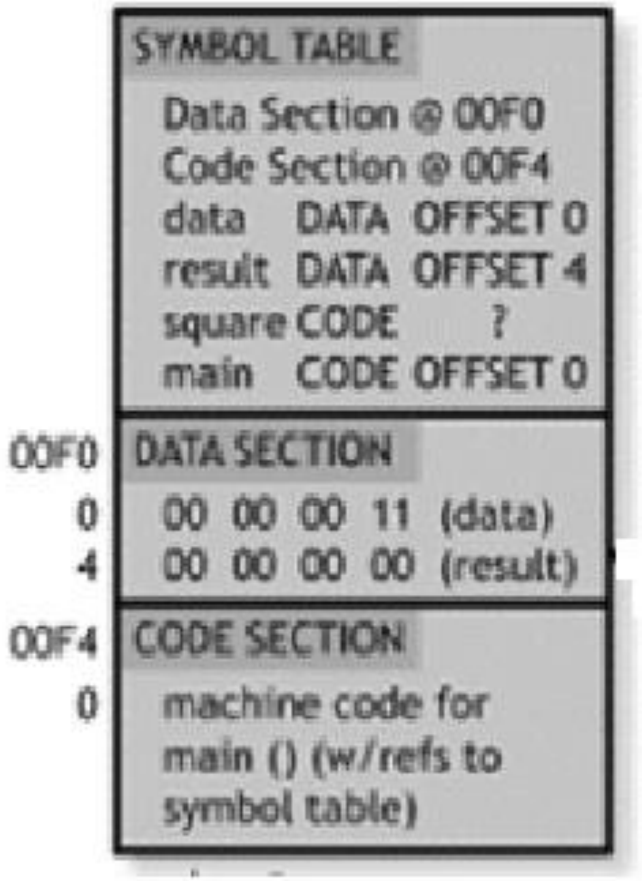
The Assembly Process：生成.obj

- Assembler translates source file to *object code*（common object file format, COFF）
  - Recognizes *mnemonics* for OP codes
  - Interprets *addressing modes* for operands
  - Recognizes *directives* that define constants and allocate space in memory for data
  - *Labels* and *names* placed in symbol table
- 关键问题：Consider forward branch to label in program
  - Offset cannot be found without target address
- Let assembler make two passes over program
  - 1st pass: generate all machine instructions, and enter labels/addresses into symbol table
    - Some instructions incomplete but sizes known
  - 2nd pass: calculate unknown branch offsets using address information in symbol table



```
      SYMBOL TABLE
        Data Section @ 00F0
        Code Section @ 00F4
        data    DATA  OFFSET 0
        result  DATA  OFFSET 4
        square  CODE     ?
        main    CODE  OFFSET 0
00F0  DATA SECTION
   0    00  00  00  11  (data)
   4    00  00  00  00  (result)
00F4  CODE SECTION
   0    machine code for
        main () (w/refs to
        symbol table)
```

.obj与Symbol Table



```
  .text
  .align 2
  .globl main
main:
  addi sp,sp,-16
  sw   ra,12(sp)
  lui  a0,%hi(string1)
  addi a0,a0,%lo(string1)
  lui  a1,%hi(string2)
  addi a1,a1,%lo(string2)
  call printf
  lw   ra,12(sp)
  addi sp,sp,16
  li   a0,0
  ret
  .section .rodata
  .balign 4
string1:
  .string "Hello, %s!\n"
string2:
  .string "world"
```

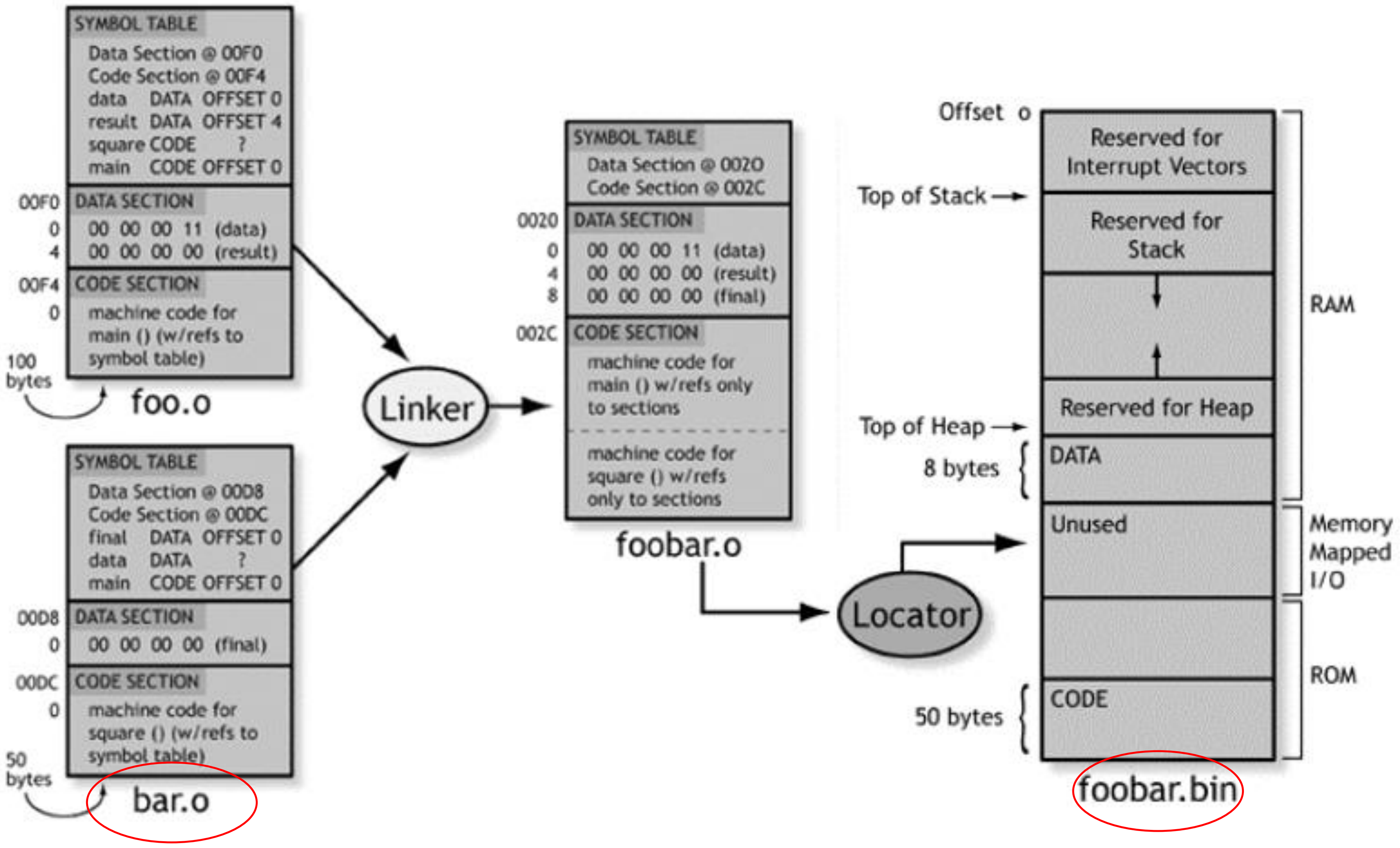
符号表：
全局定义和外部引用

directives： 内存地址指针
Labels： 程序地址标号
names： 段名，变量名

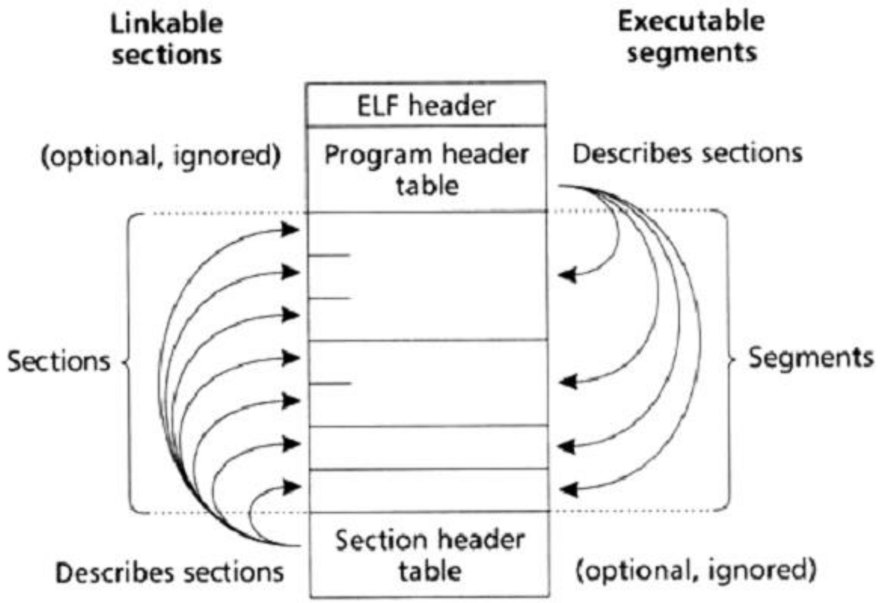
The Linker：合并各段

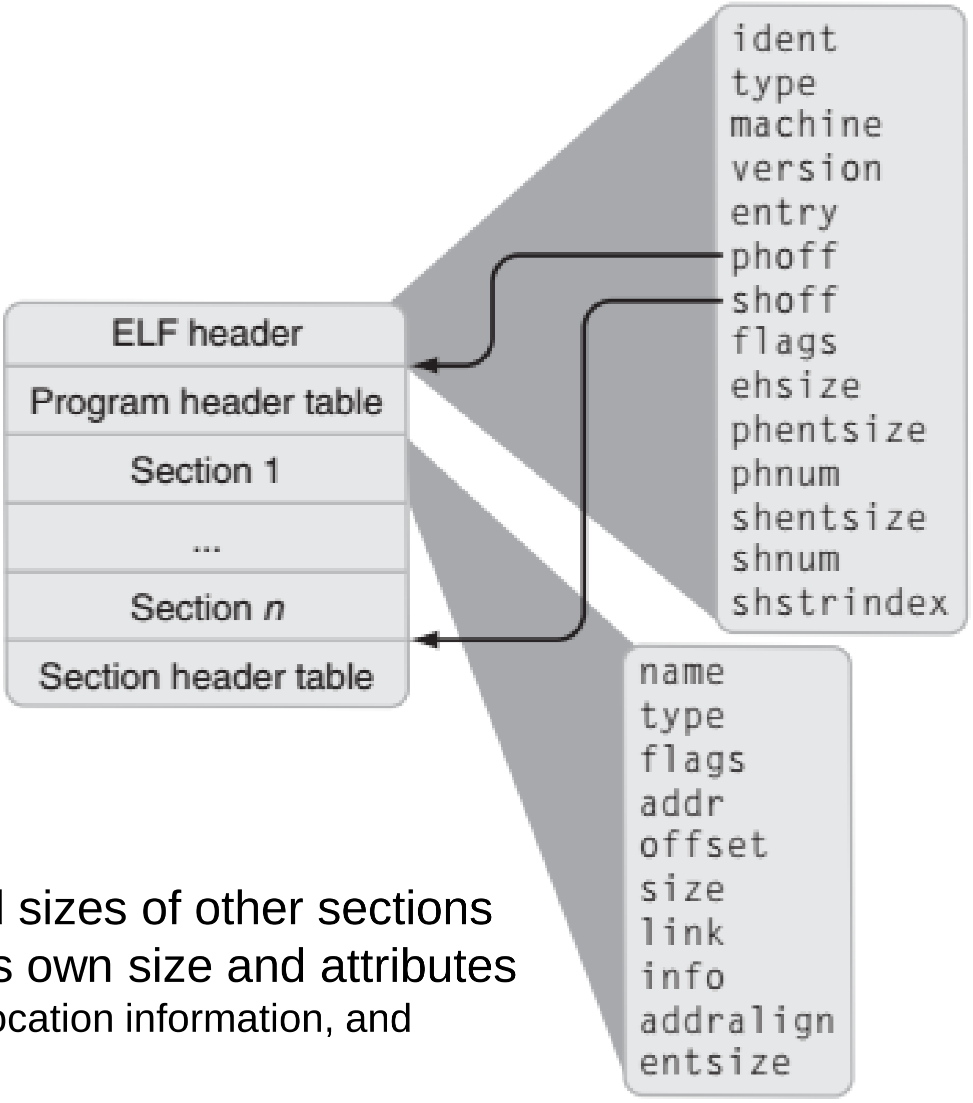
- Combines object files into object program（exe）
  - Constructs map of full program in memory using length information in each object file
    - Map determines addresses of all names
  - Instructions referring to external names are finalized with addresses determined by map
- Libraries：Subroutines
  - includes name information to aid in resolving references from calling program

Linking and Locating



ELF 格式目标文件结构





- the ELF specification
  - ELF header tell the offsets and sizes of other sections
  - Each section then describes its own size and attributes
    - symbol tables, string tables, relocation information, and dynamic linking information

Loading/Executing Object Programs

- 将映像文件从磁盘加载到内存
  - 读取文件头来确定各段大小
  - 创建虚拟地址空间
  - 将代码和初始化的数据复制到内存中
    - 或设置页表项来处理缺页
  - 在栈上建立参数
  - 初始化寄存器（包括sp、 fp、 gp）
  - 跳转到启动例程
    - 将参数复制到x10等等并调用main函数
    - 当main函数返回时，进行exit系统调用

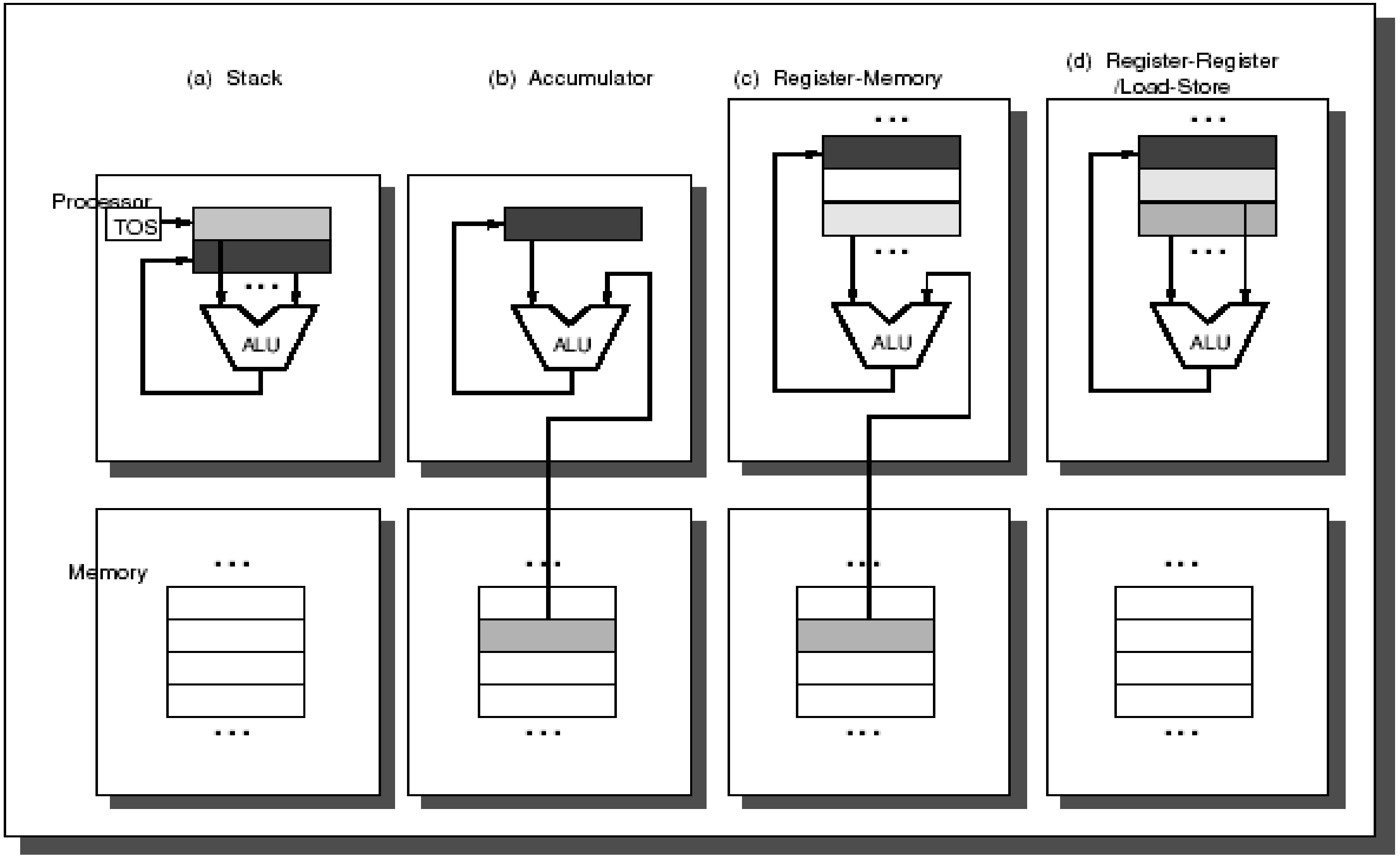
影响早期ISA设计的因素

- 内存小而慢，能省则省
  - 某个完整系统只需几K字节
  - 指令长度不等、执行多个操作的指令
- 寄存器贵，少
  - 操作基于存储器
  - 多种寻址方式
- 编译技术尚未出现
  - 程序是以机器语言或汇编语言设计
  - 当时的看法是硬件比编译器更易设计
    - 为了便于编写程序，计算机架构师造出越来越复杂的指令，完成高级程序语言直接表达的功能
    - 进化中的痕迹：X86中的串操作指令

ISA分类

- 指令格式和寻址方式越复杂，则越灵活高效
  - 权衡：硬件设计复杂度、指令系统的兼容性
- 机器实现角度：processor designer view
  - stack
  - Accumulator
  - register-mem
  - register-register
- 程序员角度：programmer/compiler view
  - CISC：以机器指令实现高级语言功能
  - RISC：采用load/store体系，运算基于寄存器（register-register）
  - VLIW：兼容性差，硬件简单，低功耗

ISA Classes (processor designer view)



H&P：附录B，图B.1

ISA分类（programmer prospective）

- CISC：硬件换性能！——上千条指令
  - 以机器指令实现高级语言功能
  - 指令译码复杂
    - 指令格式、字长不一（x86从1byte～6bytes）
    - 寻址方式多
  - 访存开销大：寄存器少，任何指令都可以访存
- RISC：简化硬件，优化常用操作！—百余条指令
  - 指令字长固定，格式规则，种类少，寻址方式简单
  - 减少访存，设置大量通用寄存器，运算基于寄存器
    - 为了提高性能，需要减少访存次数，因此寄存器寻址性能最高。
    - 采用load/store体系，只有load/store指令访存。
  - 采用Superscalar、Superpipeling等技术，提高IPC
- VLIW：空间换时间，低功耗，兼容性差

The CISC's eight principles:

- Instructions are of variable format.
- There are multiple instructions and addressing modes.
- Complex instructions take many different cycles.
- Any instruction can reference memory.
- There is a single set of registers.
- No instructions are pipelined.
- A microprogram is executed for each native instruction.
- Complexity is in the microprogram and hardware.

X86指令格式，图2-35

a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r-m postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

Growth of x86 instruction set over time

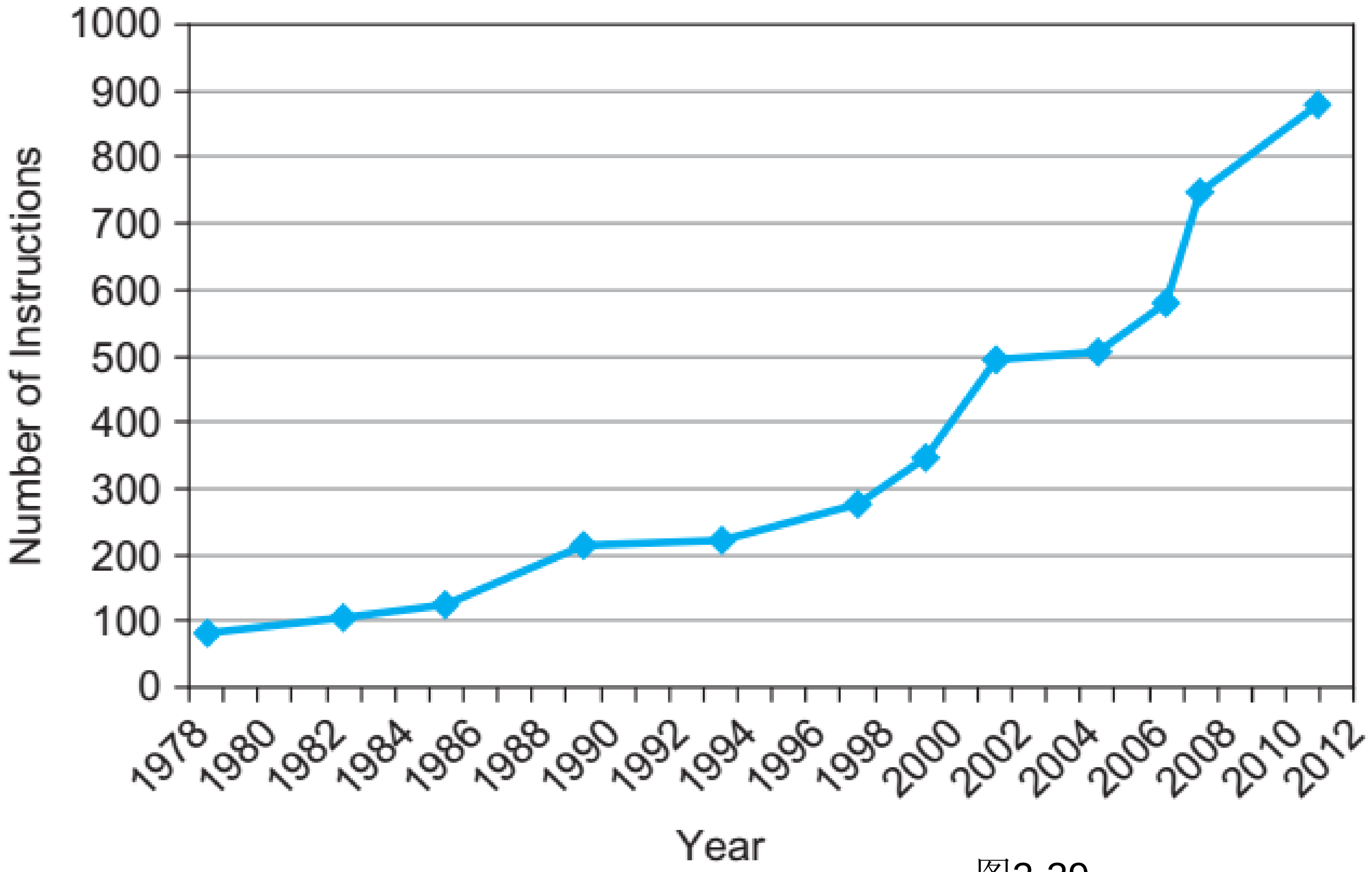
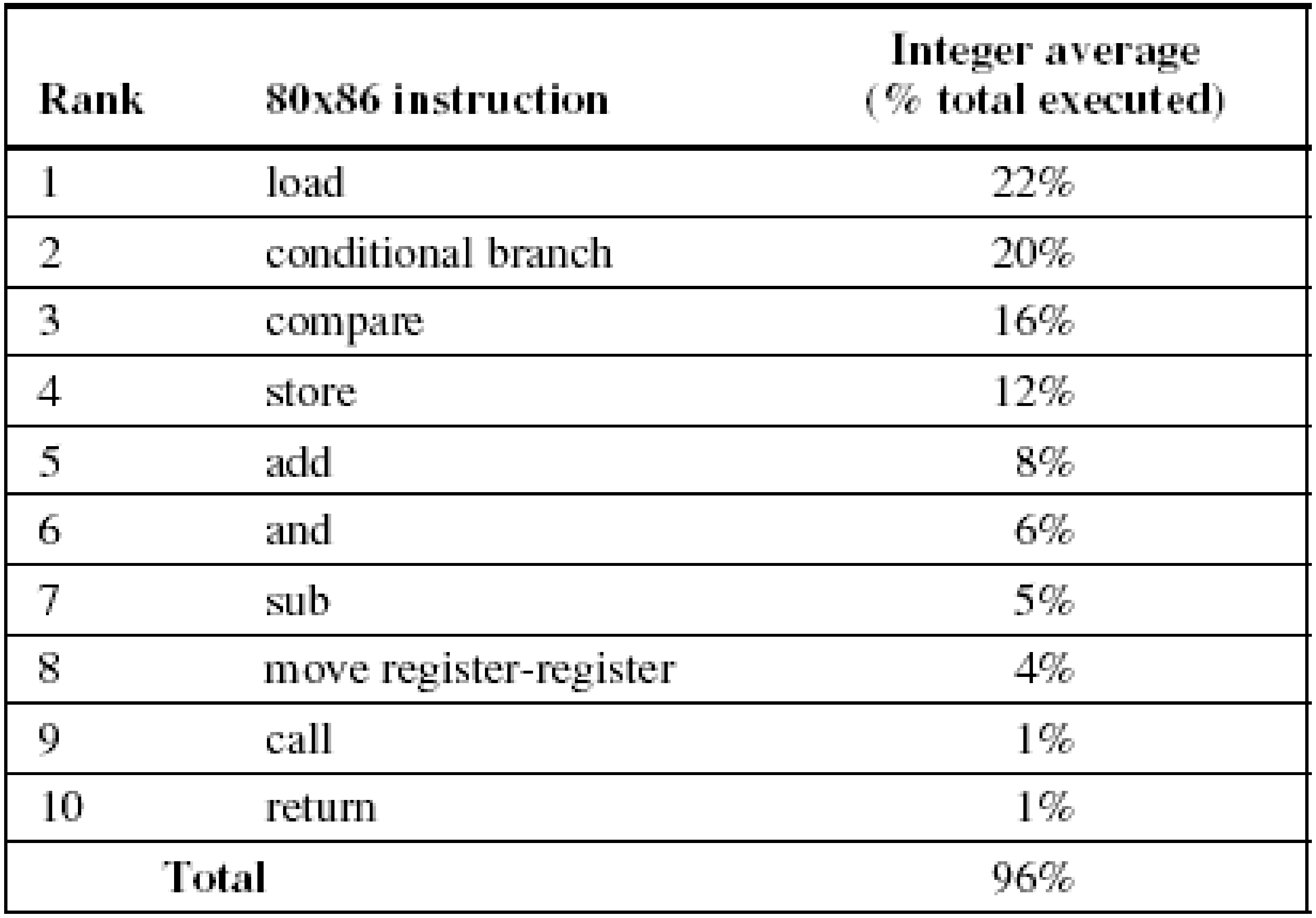


图2-39

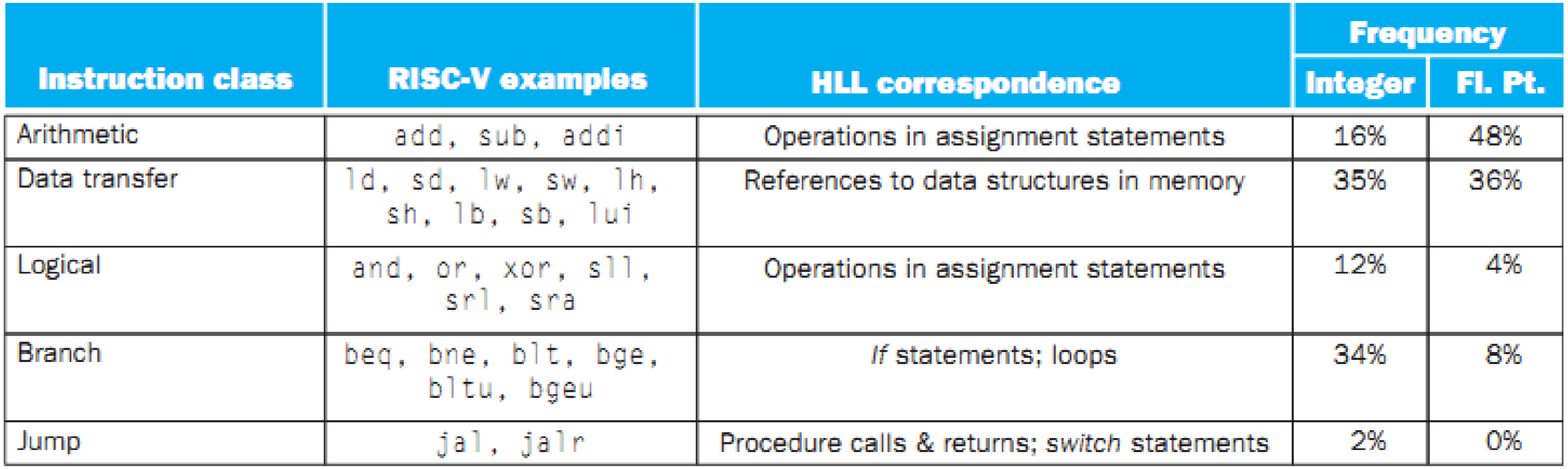
X86 Instruction Distribution



Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

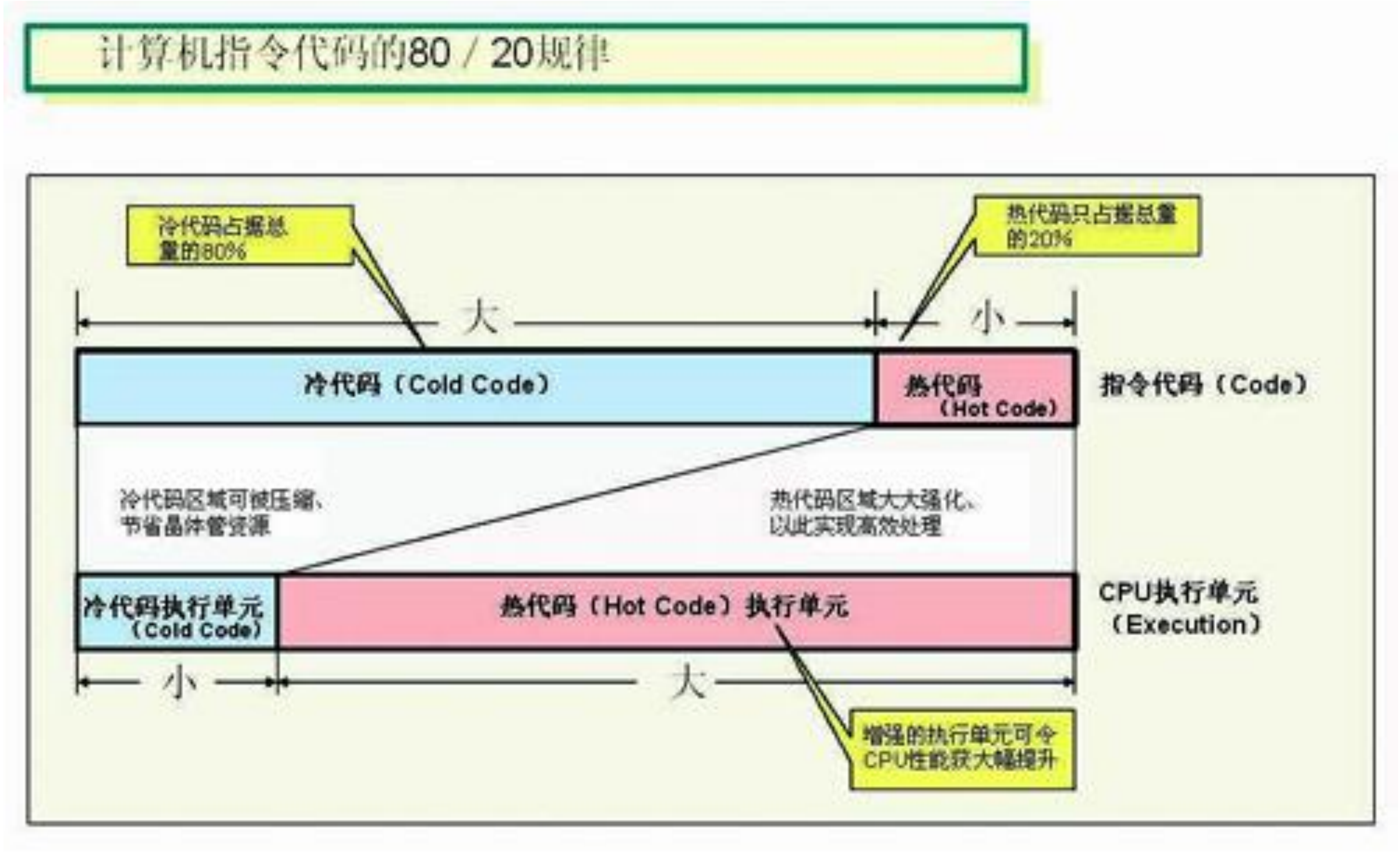
RV指令分布，图2-41

- SPEC CPU2006



Instruction class	RISC-V examples	HLL correspondence	Frequency	
			Integer	Fl. Pt.
Arithmetic	`add, sub, addi`	Operations in assignment statements	16%	48%
Data transfer	`ld, sd, lw, sw, lh, sh, lb, sb, lui`	References to data structures in memory	35%	36%
Logical	`and, or, xor, sll, srl, sra`	Operations in assignment statements	12%	4%
Branch	`beq, bne, blt, bge, bltu, bgeu`	*If* statements; loops	34%	8%
Jump	`jal, jalr`	Procedure calls & returns; *switch* statements	2%	0%

RISC的理论基础


计算机指令代码的80 / 20规律
冷代码占据总量的80%
热代码只占据总量的20%
大
小
冷代码（Cold Code）
热代码（Hot Code）
指令代码（Code）
冷代码区域可被压缩、节省晶体管资源
热代码区域大大强化、以此实现高效处理
冷代码执行单元（Cold Code）
热代码（Hot Code）执行单元
CPU执行单元（Execution）
小
大
增强的执行单元可令CPU性能获大幅提升

The RISC’s eight principles:

- Fixed-format instructions.
- Few instructions and addressing modes.
- Simple instructions taking one clock cycle.
- LOAD/STORE architecture to reference memory.
- Large multiple-register sets.
- Highly pipelined design.
- Instructions executed directly by hardware.
- Complexity handled by the compiler and software.

CISC machine vs RISC machine

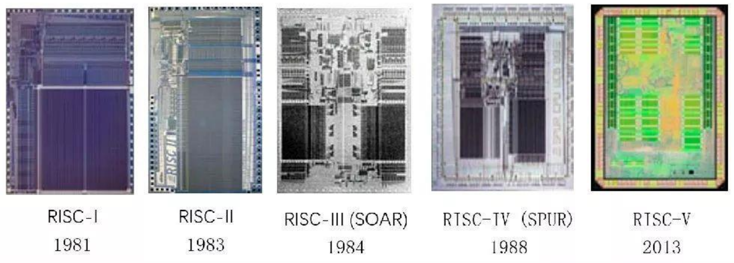
- Instructions are of variable format.
- There are multiple instructions and addressing modes.
- Complex instructions take many different cycles.
- Any instruction can reference memory.
- There is a single set of registers.
- No instructions are pipelined.
- A microprogram is executed for each native instruction.
- Complexity is in the microprogram and hardware.

- Fixed-format instructions.
- Few instructions and addressing modes.
- Simple instructions taking one clock cycle.
- LOAD/STORE architecture to reference memory.
- Large multiple-register sets.
- Highly pipelined design.
- Instructions executed directly by hardware.
- Complexity handled by the compiler and software.

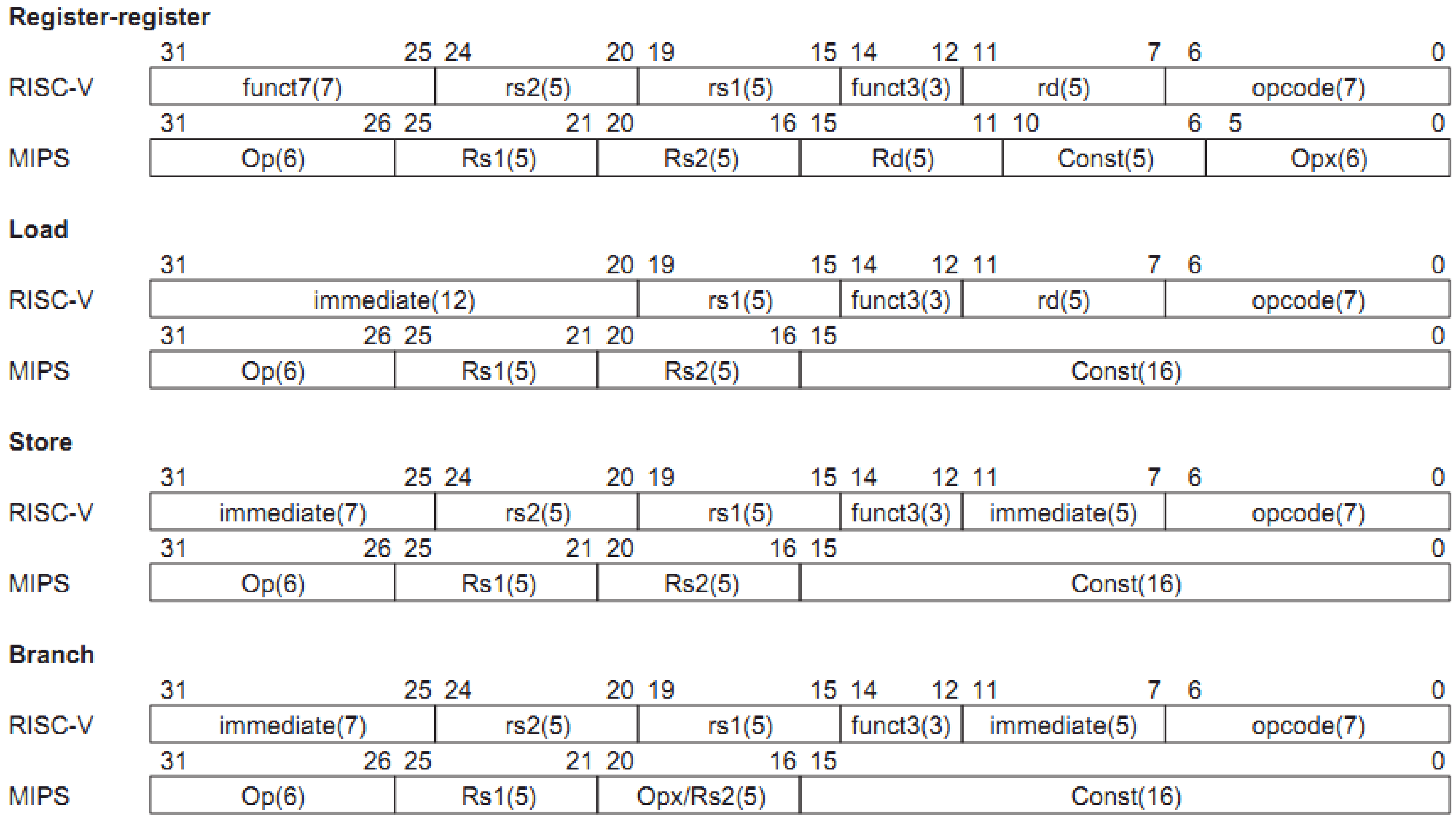
MIPS is simple, elegant.

- MIPS：无互锁流水级的微处理器 (Microprocessor w/o Interlocked Piped Stages)
  - interlock单元：检测RAW相关，推迟后续指令执行（互锁状态）
    - 尽量利用软件办法避免流水线中的数据依赖问题
    - R4000以后开始使用interlock
- 1980，Patterson提出RISC指令集
  - *4个Design Principles*：
    - Simplicity favors regularity！
    - Make the common case fast！
- 1983，Hennessy完成第一个RISC处理器MIPS。

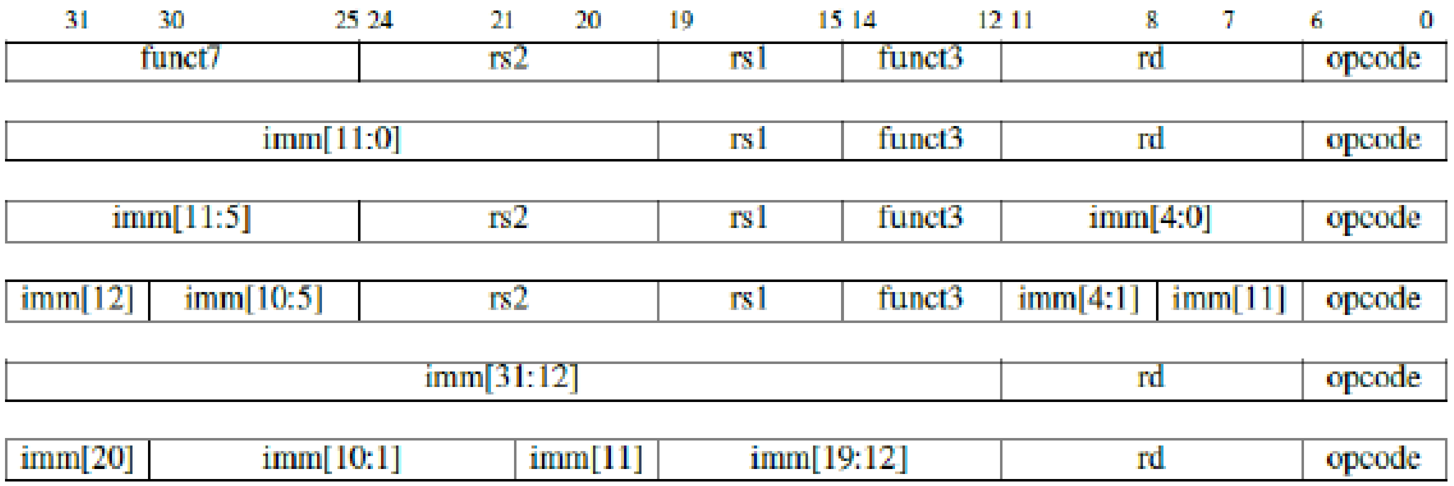
Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.



RV vs. MIPS，图2.29

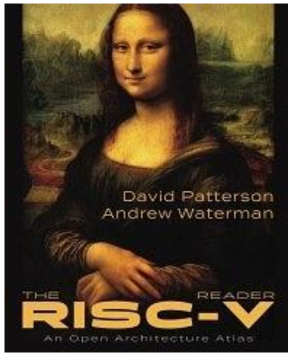

Register-register
31 25 24 20 19 15 14 12 11 7 6 0
RISC-V
funct7(7) rs2(5) rs1(5) funct3(3) rd(5) opcode(7)
31 26 25 21 20 16 15 11 10 6 5 0
MIPS
Op(6) Rs1(5) Rs2(5) Rd(5) Const(5) Opx(6)
Load
31 20 19 15 14 12 11 7 6 0
RISC-V
immediate(12) rs1(5) funct3(3) rd(5) opcode(7)
31 26 25 21 20 16 15 0
MIPS
Op(6) Rs1(5) Rs2(5) Const(16)
Store
31 25 24 20 19 15 14 12 11 7 6 0
RISC-V
immediate(7) rs2(5) rs1(5) funct3(3) immediate(5) opcode(7)
31 26 25 21 20 16 15 0
MIPS
Op(6) Rs1(5) Rs2(5) Const(16)
Branch
31 25 24 20 19 15 14 12 11 7 6 0
RISC-V
immediate(7) rs2(5) rs1(5) funct3(3) immediate(5) opcode(7)
31 26 25 21 20 16 15 0
MIPS
Op(6) Rs1(5) Opx/Rs2(5) Const(16)

RV vs. MIPS



31	30 25	24 21	20	19 15	14 12	11 8	7	6 0		
funct7		rs2		rs1	funct3	rd		opcode	R-type	寄存器-寄存器操作
imm[11:0]				rs1	funct3	rd		opcode	I-type	短立即数和访存load
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type	访存store指令
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type	条件跳转指令
imm[31:12]						rd		opcode	U-type	长立即数
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type	无条件跳转

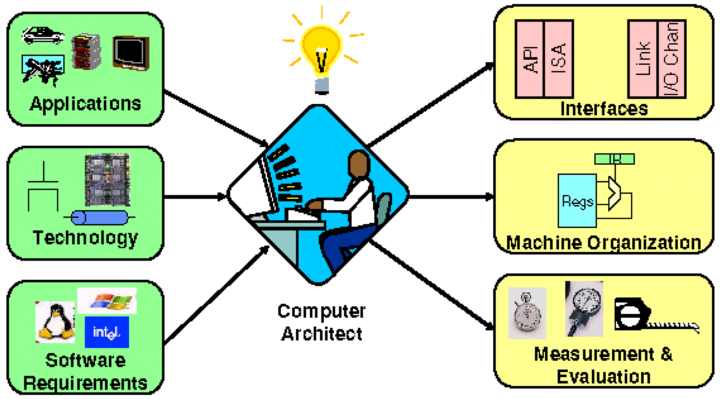
R-type	op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)
reg-reg	op(6 bits)	rs(5 bits)	rt(5 bits)	immediate(16 bits)		
I-type *reg-mm*	op(6 bits)	rs(5 bits)	rt(5 bits)	addr(16 bits)		
J-type	op(6 bits)	rs(5 bits)	rt(5 bits)	addr(16 bits)		
	op(6 bits)	addr(26 bits)				



- 1）立即数在高位；2）rs/rd位置固定。

ISA: A Minimalist Perspective

- **ISA design decisions must take into account:**
  - **technology**
  - **machine organization**
  - **programming languages**
  - **compiler technology**
  - **operating systems**



- “最小”计算机？——快速原型☺，ABC
  - “A”：由哪些部件构成？
  - “B”：需要哪几条指令？需要哪些寻址方式？

OISC：the one instruction set computer

- OISC：the ultimate reduced instruction set computer
  - 一条SBN指令：substract and branch if negative
    - subleq *a*, *b*, *c*；Mem[*b*] = Mem[*b*] - Mem[*a*]，
      if (Mem[*b*] ≤ 0) goto *c*
- 应用：嵌入式处理器
  - 硬件极其简单
    - 程序员有充分的控制权
    - 优化由编译器完成
  - 灵活
    - 其他“指令”都可由该指令构造
    - 意味着用户可自定义指令集
    - 意味着可适用于任何领域
  - 低功耗

COMPUTER ARCHITECTURE:
A Minimalist Perspective

William F. Gilreath
Phillip A. Laplante

Springer

小结

- 作业
  - 2.9，2.24，2.35，2.40
- 思考（选一）
  - CPU的ISA要定义哪些内容？
    - 见Yale Patt附录A
  - Windows系统中可执行程序的格式？
- 实验报告：2周
  - 基于RV汇编，设计一个冒泡排序程序，并用Ripes工具调试执行。
  - 可选：测量冒泡排序程序的执行时间。

Bubble sort (trace)

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	5	3

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	3	4	5	10

Basic idea:

① $j \leftarrow n-1$ (index of last element in A)

② If $A[j] < A[j-1]$, swap both elements

③ $j \leftarrow j-1$, goto ② if $j > 0$

④ Goto ① if a swap occurred

参考Ripes仿真器的“Console printing”代码？

①

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	5	③

②

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	③ ↔	5

③

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	③	5

②

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	③ ↔	10	5

③

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	③	10	5

②

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	③ ↔	4	10	5

③

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	③	4	10	5

②

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
③ ↔	3	4	10	5

④ Swap occured? (Yes, goto ①)

①

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	3	4	10	⑤