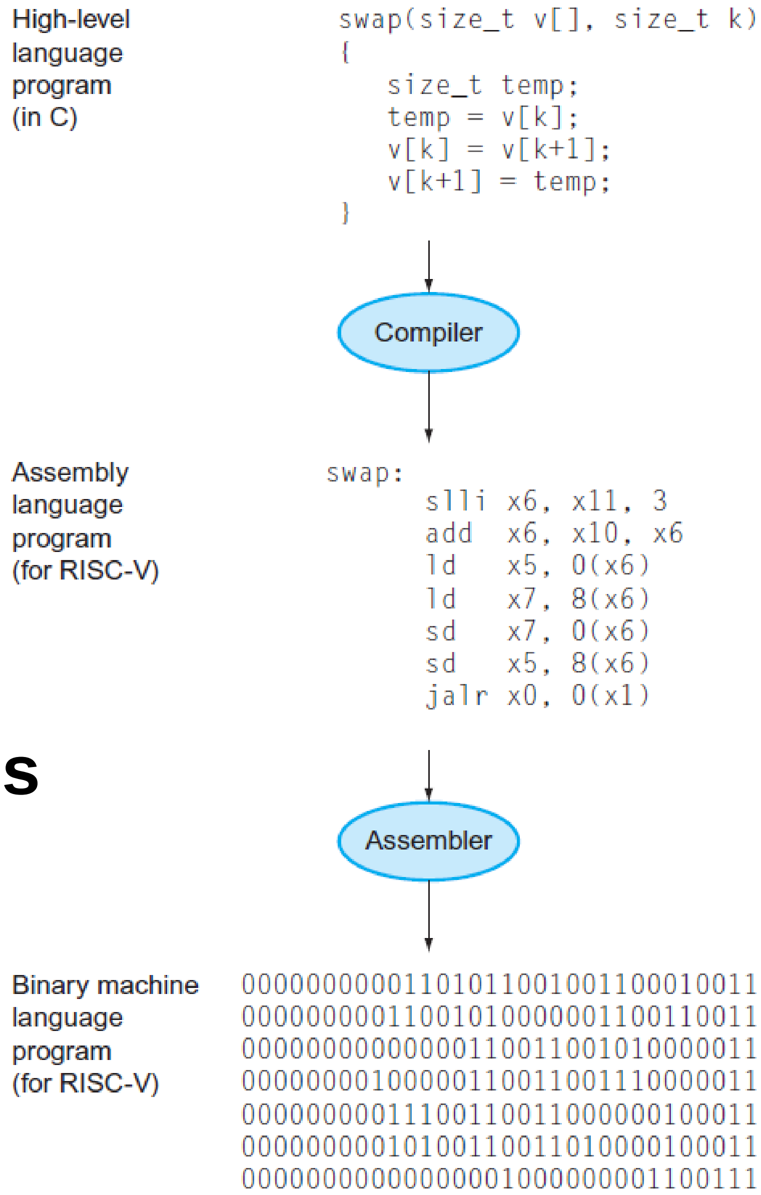
计算机组成原理
第二章“指令系统”

中科大11系

李曦

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【数制】


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

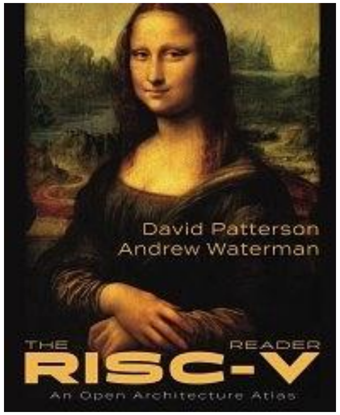
```
swap(size_t v[], size_t k)
{
   size_t temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
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
```

概要

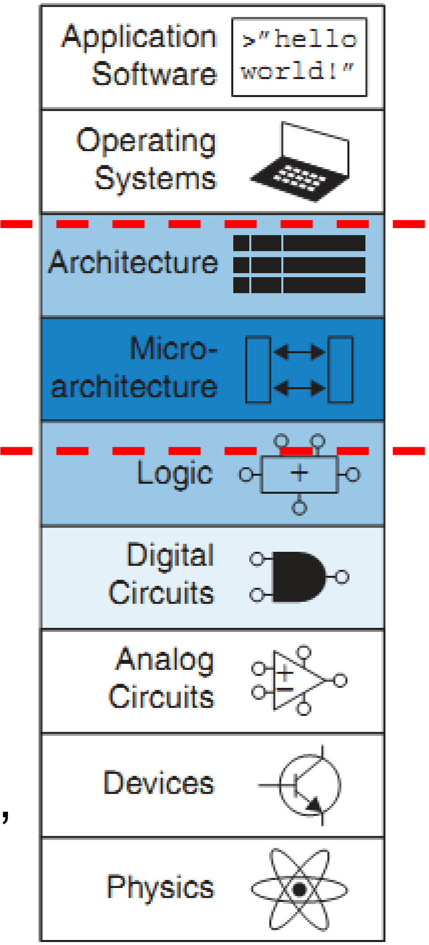
David Patterson，Andrew Waterman，*The RISC-V Reader: An Open Architecture Atlas*，2017



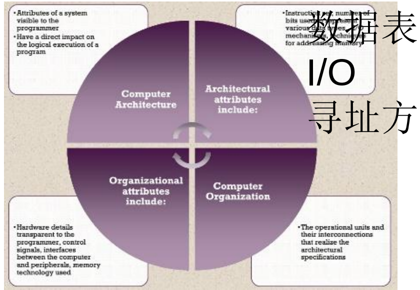
- 指令系统：机器指令的集合
  - “程序控制”
    - 程序=顺序执行的指令流
  - 机器语言，汇编语言（Assemble Language）
  - Instruction Set Architecture（ISA）
    - 分类：CISC、RISC、VLIW
    - 影响：处理器、C编译器、OS。。。
- 本章的内容
  - RV指令系统
    - 操作数：寄存器，存储器（大小尾端，对齐），常数/立即数；
    - 指令功能，指令格式与编码，寻址方式（操作数，下一条指令）
    - 分支指令$2.7，*大立即数处理$2.10*
    - 过程调用$2.8，$2.13
  - *汇编程序设计：COD4附录B*
  - 指令系统特征
  - *编译过程：$2.12*
    - *可执行程序生成：编译，汇编，链接，加载*

Instruction-Set Processor Design

- Architecture (ISA) *programmer/compiler view*
  - "functional appearance to its immediate user/system programmer"
  - Opcodes, addressing modes, architected registers, IEEE floating point
  - 机器语言
- Implementation (µArch) *processor designer view*
  - "logical structure or organization that performs the architecture"
  - functional units, pipelining, caches, physical registers
- Realization (chip) *chip/system designer view*
  - "physical structure that embodies the implementation"
  - Gates, cells, transistors, wires



体系结构（ISA）的8种属性

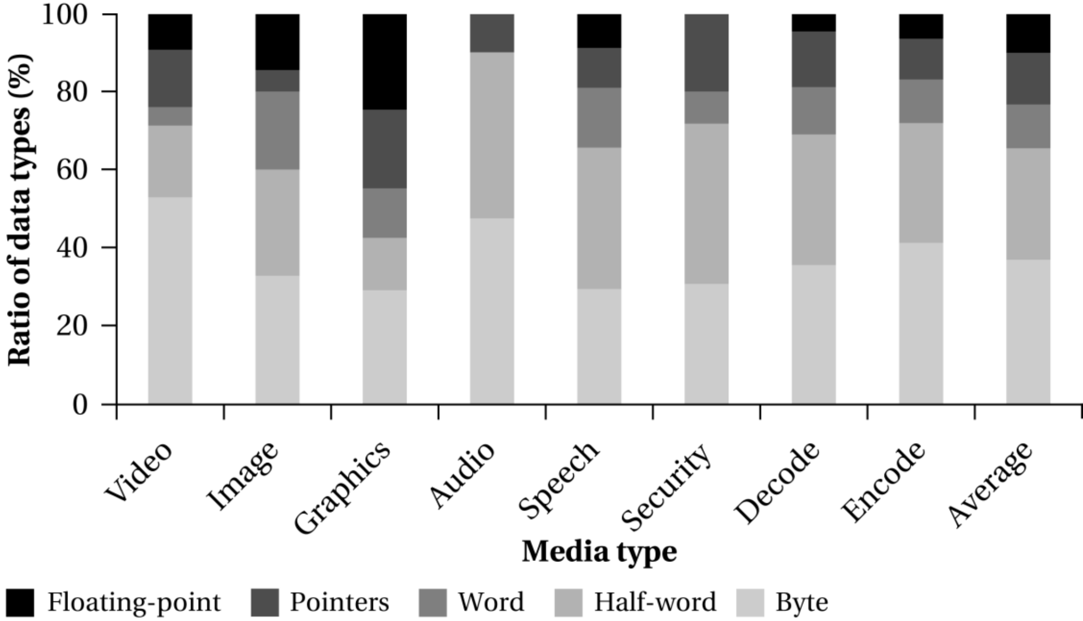

•Attributes of a system visible to the programmer
•Have a direct impact on the logical execution of a program
Computer Architecture
Architectural attributes include:
•Instruction set, number of bits used to represent various data types, I/O mechanisms, techniques for addressing memory
Organizational attributes include:
Computer Organization
•Hardware details transparent to the programmer, control signals, interfaces between the computer and peripherals, memory technology used
•The operational units and their interconnections that realize the architectural specifications

指令集
数据表示
I/O
寻址方式

- 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（专用）寄存器的定义、数量和使用规则
- 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【MCM？】
- 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。
- 输入输出
  - I/O互连方式、处理机/存储器与I/O设备间的数据交换方式和过程控制
- 中断（异常）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*

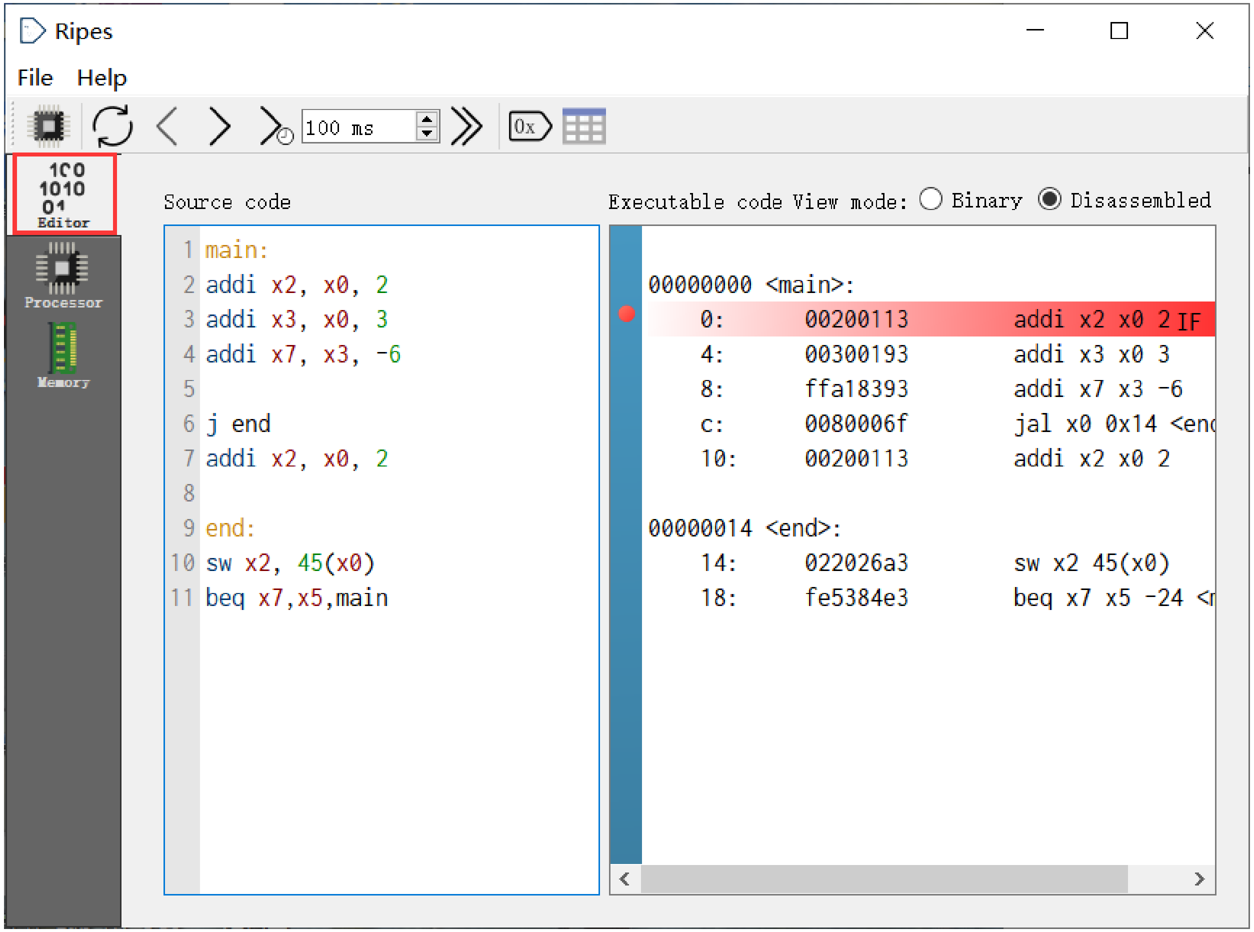
操作数（opr）： Data Representation，Number Systems

- 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/*RV64*”：32位/*64位*，“大立即数”
  - 字节
  - 半字：2B
  - 字：4B
  - *双字：8B*
- 物理操作数：存放位置
  - 寄存器
  - 主存
  - I/O端口
  - 外存？



指令字中的操作数

- 立即数
  - 进制表示
    - 十进制：2
    - 十六进制
      - 0x1F
    - 二进制
      - 0b1101
- 寄存器
- 存储器
  - 内存地址
  - 字长：sw
- 标号/行号
  - main，end



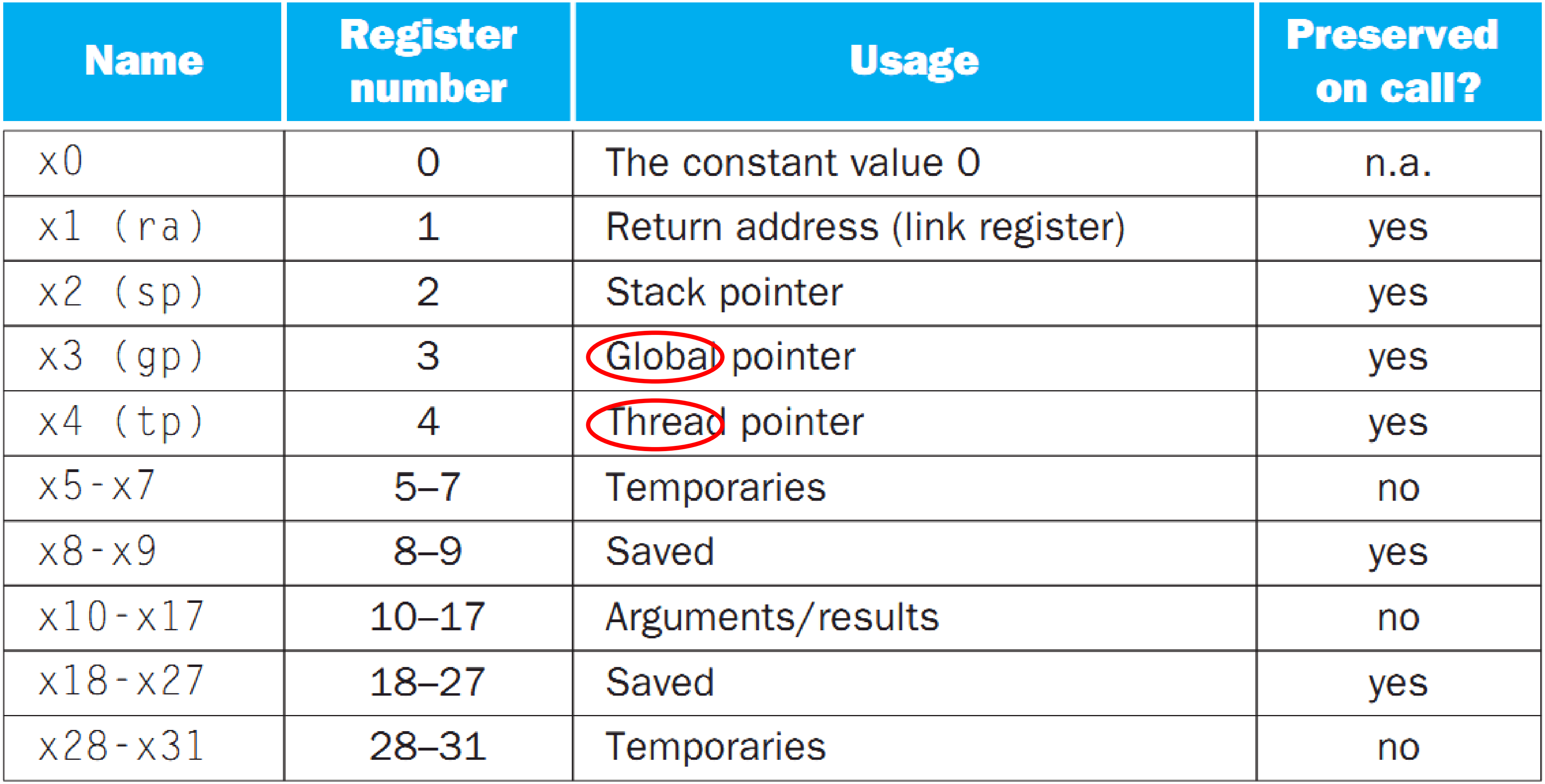
Source code

```
main:
addi x2, x0, 2
addi x3, x0, 3
addi x7, x3, -6

j end
addi x2, x0, 2

end:
sw x2, 45(x0)
beq x7,x5,main
```

Executable code View mode: ○ Binary ● Disassembled

```
00000000 <main>:
    0:      00200113        addi x2 x0 2 IF
    4:      00300193        addi x3 x0 3
    8:      ffa18393        addi x7 x3 -6
    c:      0080006f        jal x0 0x14 <end
    10:     00200113        addi x2 x0 2

00000014 <end>:
    14:     022026a3        sw x2 45(x0)
    18:     fe5384e3        beq x7 x5 -24 <
```

RV architected registers和ABI，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

ABI寄存器名，用途，过程调用时是否要保存到栈中

RV典型内存地址空间分配：段式

- 段式
  - DATA
    - static
    - Stack/Heap
      - 栈：自高向低
      - 堆：自低向高
  - CODE/Text
- 地址格式
  - 绝对地址
  - 相对地址：基址+偏移
    - offset，displacement
    - 基址寄存器bp
- 外设地址
  - I/O port?
  - 硬盘
  - 网络

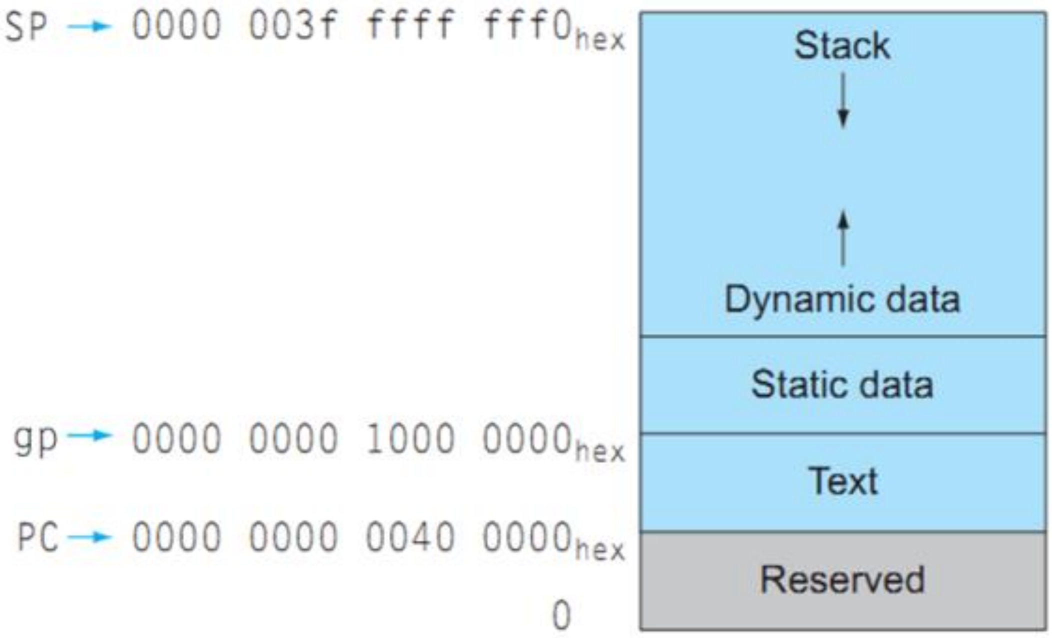

SP → 0000 003f ffff fff0hex
gp → 0000 0000 1000 0000hex
PC → 0000 0000 0040 0000hex
0
Stack
Dynamic data
Static data
Text
Reserved

图2-13 Linux用户地址空间划分约定

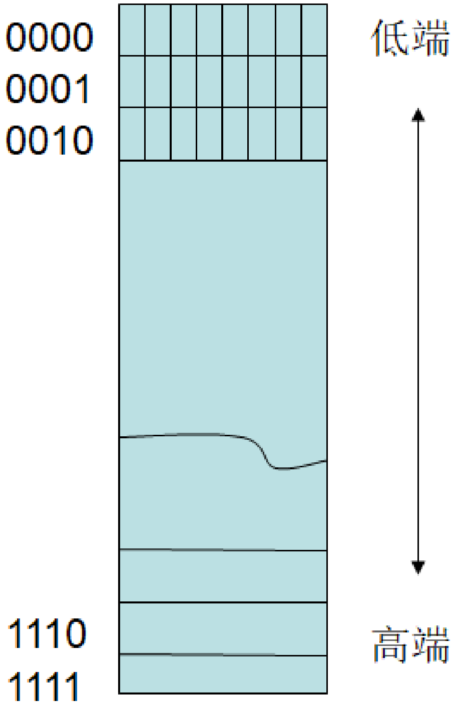
可寻址的总的内存空间大小？

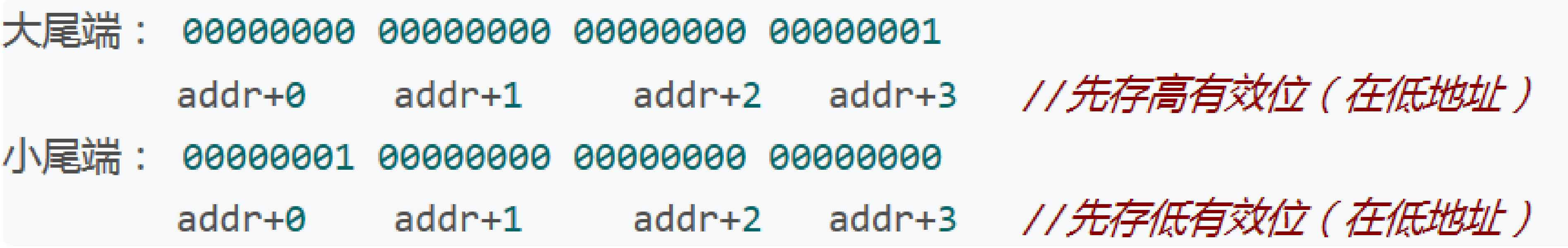
各段空间大小？

字存储顺序（Byte Ordering）$2.3.1

- 字存储的顺序中，字节的次序有两种
  - 大端/大尾端（big endness）
    - 低地址，高字节
  - 小端/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）
```

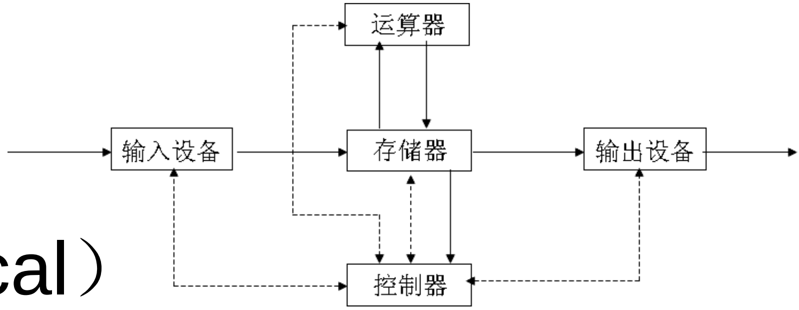
数据对齐（Memory Alignment）， $2.3.1

- 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。
- 边界对齐：RV/x86不要求，MIPS要求
  - 字对齐：地址左移两位，按字访问
  - 半字对齐：地址左移一位，按半字访问

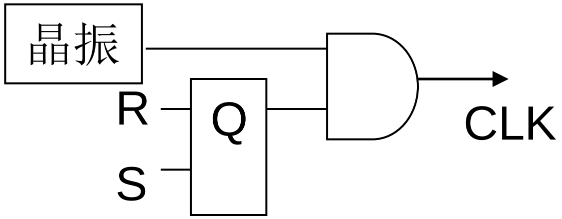
<table>
<tr><th colspan="3">存储器</th><th>地址（十进制）</th></tr>
<tr><td colspan="2">字（地址2）</td><td>半字（地址0）</td><td>0</td></tr>
<tr><td>字节（地址7）</td><td>字节（地址6）</td><td>字（地址4）</td><td>4</td></tr>
<tr><td colspan="2">半字（地址10）</td><td>半字（地址8）</td><td>8</td></tr>
</table>

操作分类

- 数据传递（data movement）
  - 访存：load，store，*mov*
  - I/O：*in*，*out*

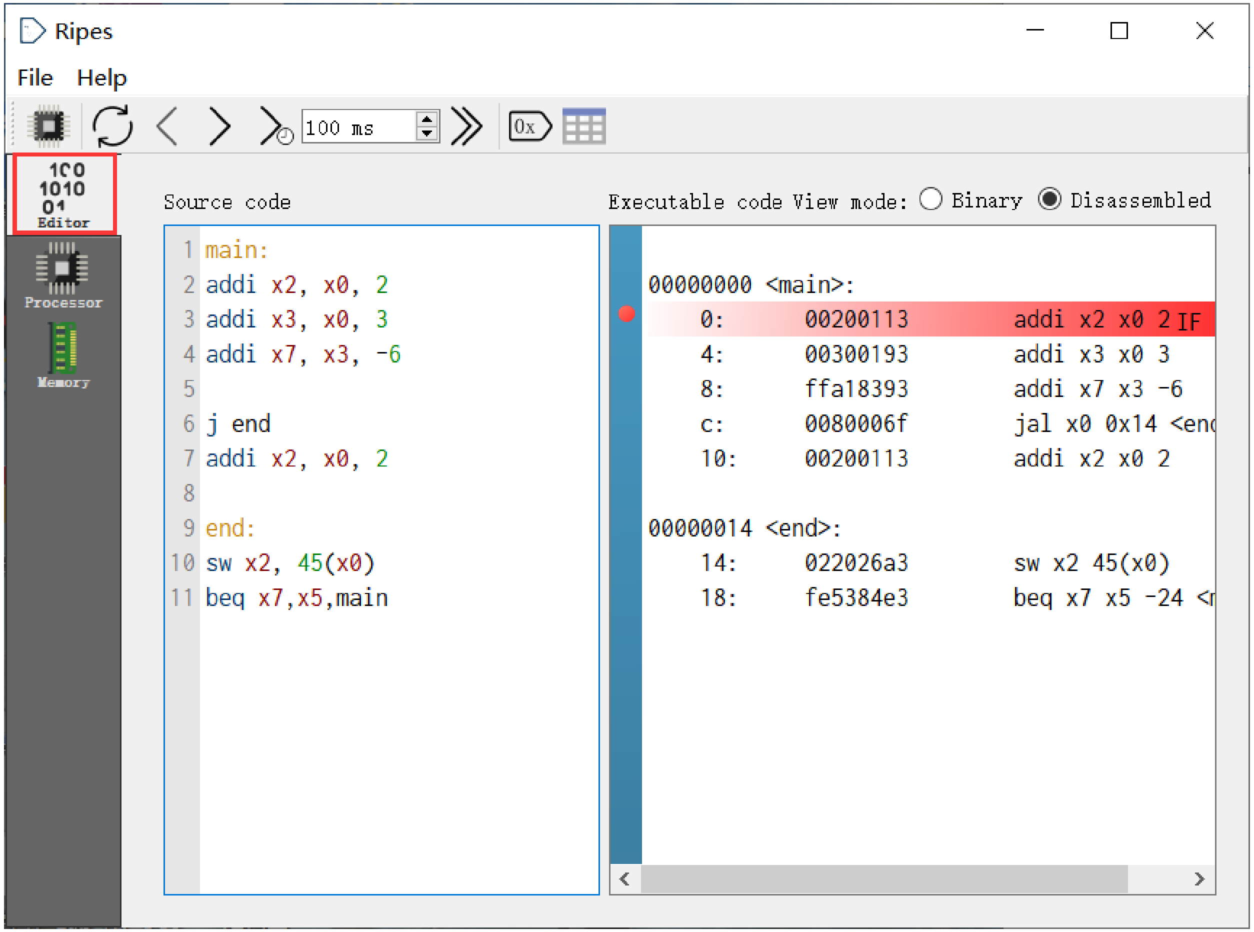


- 算逻运算（arithmetic & logical）
  - add，sub，and，not，or，xor，*dec*，*inc*，*cmp*
  - monadic & dyadic operations
- 移位操作
  - monadic operations：shl，shr，srl，srr
- 分支控制（tranfer of contral，Branch）
  - comparisons & conditional branches：beq，bnz
  - 无条件转移：jmp
  - procedure call：*call*，*ret*，*int*，*iret*
- 系统指令：nop，*sti*，*cli*，*lock*，*HLT*
- 多线程：原子，同步



指令示例：汇编指令，机器指令

- ALU：addi
  - 源操作数
    - 寄存器
    - 立即数
  - 目的操作数
    - 寄存器
- 分支：j，beq
  - NPC：行号
    - PC+1
    - 立即数
- 访存：sw
  - 内存地址
    - 基址+偏移



Source code

```
main:
addi x2, x0, 2
addi x3, x0, 3
addi x7, x3, -6

j end
addi x2, x0, 2

end:
sw x2, 45(x0)
beq x7,x5,main
```

Executable code View mode: ○ Binary ● Disassembled

```
00000000 <main>:
    0:      00200113        addi x2 x0 2 IF
    4:      00300193        addi x3 x0 3
    8:      ffa18393        addi x7 x3 -6
    c:      0080006f        jal x0 0x14 <end
    10:     00200113        addi x2 x0 2

00000014 <end>:
    14:     022026a3        sw x2 45(x0)
    18:     fe5384e3        beq x7 x5 -24 <r
```

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，opr）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），RV（16/32/…）
- “扩展操作码技术”：调整op域与addr域的宽度
  - 如果指令字长固定，则操作码长度增加，地址码长度缩短

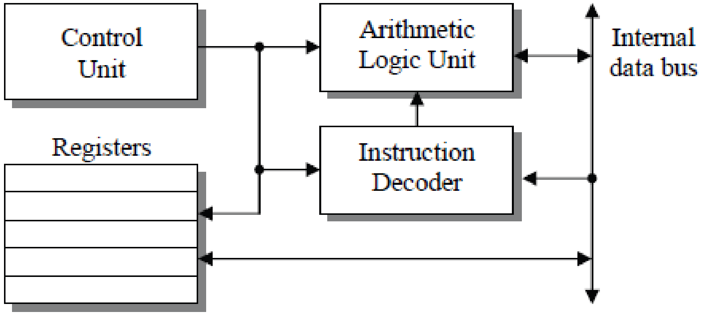
地址码：操作数，指令

- 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：指令的地址码域

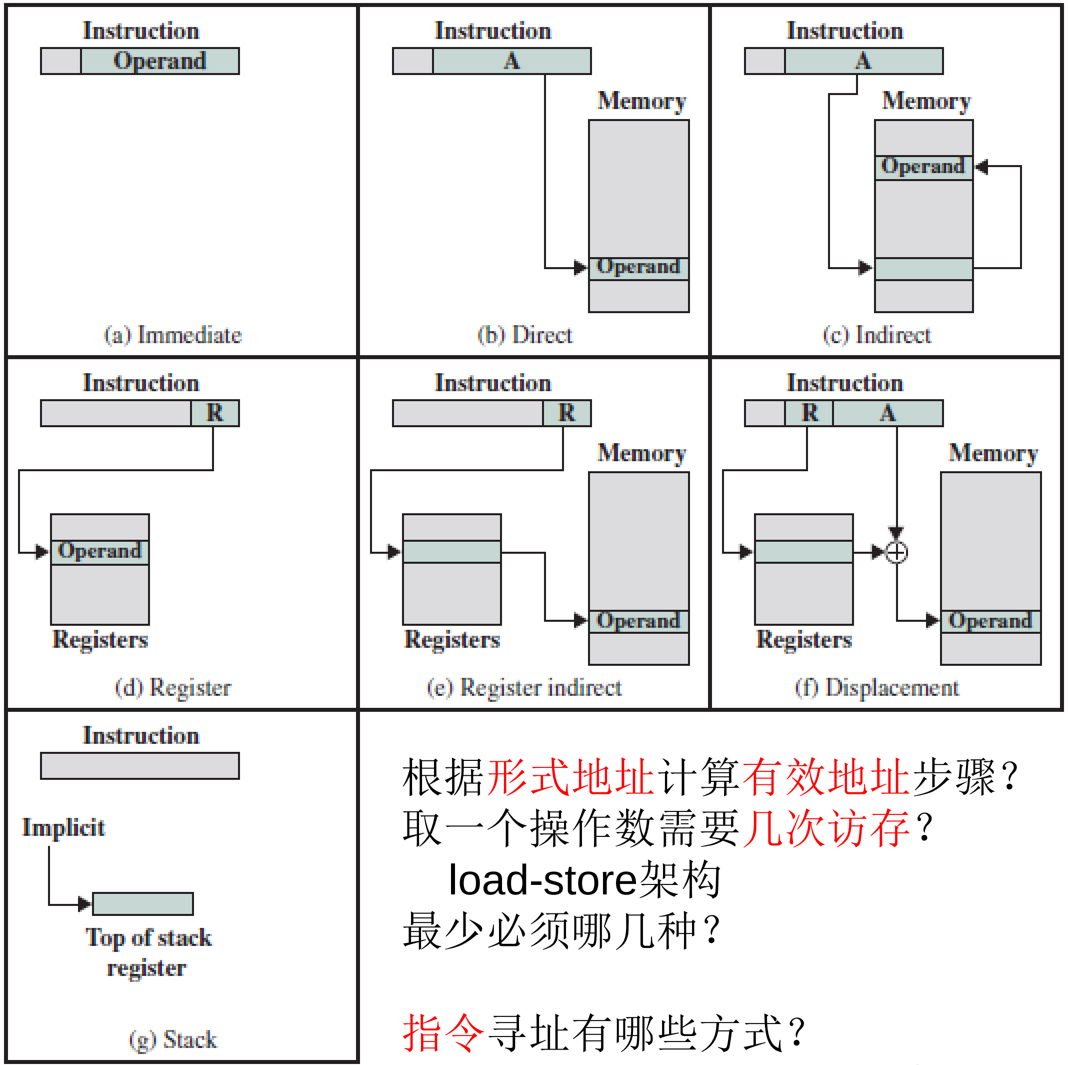
- 寻址方式：指令字和操作数的存放地址的计算方式
- 指令寻址：现代CPU利用PC
  - 顺序执行：每执行一条指令，PC自动1
  - 跳转：更新PC，转移到目的地址执行
- 操作数寻址
  - 指令中给出“形式地址”
  - 有效地址：操作数在寄存器/内存中的物理地址
    - EA＝寻址方式＋形式地址

操作码	形式地址



典型寻址方式：操作数，下一条指令

- 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+offset
  - 堆栈寻址（g）
  - *变址寻址（d+f）*
    - Index：x86的si/di
  - *隐含寻址（如堆栈）*



根据形式地址计算有效地址步骤？

取一个操作数需要几次访存？

load-store架构

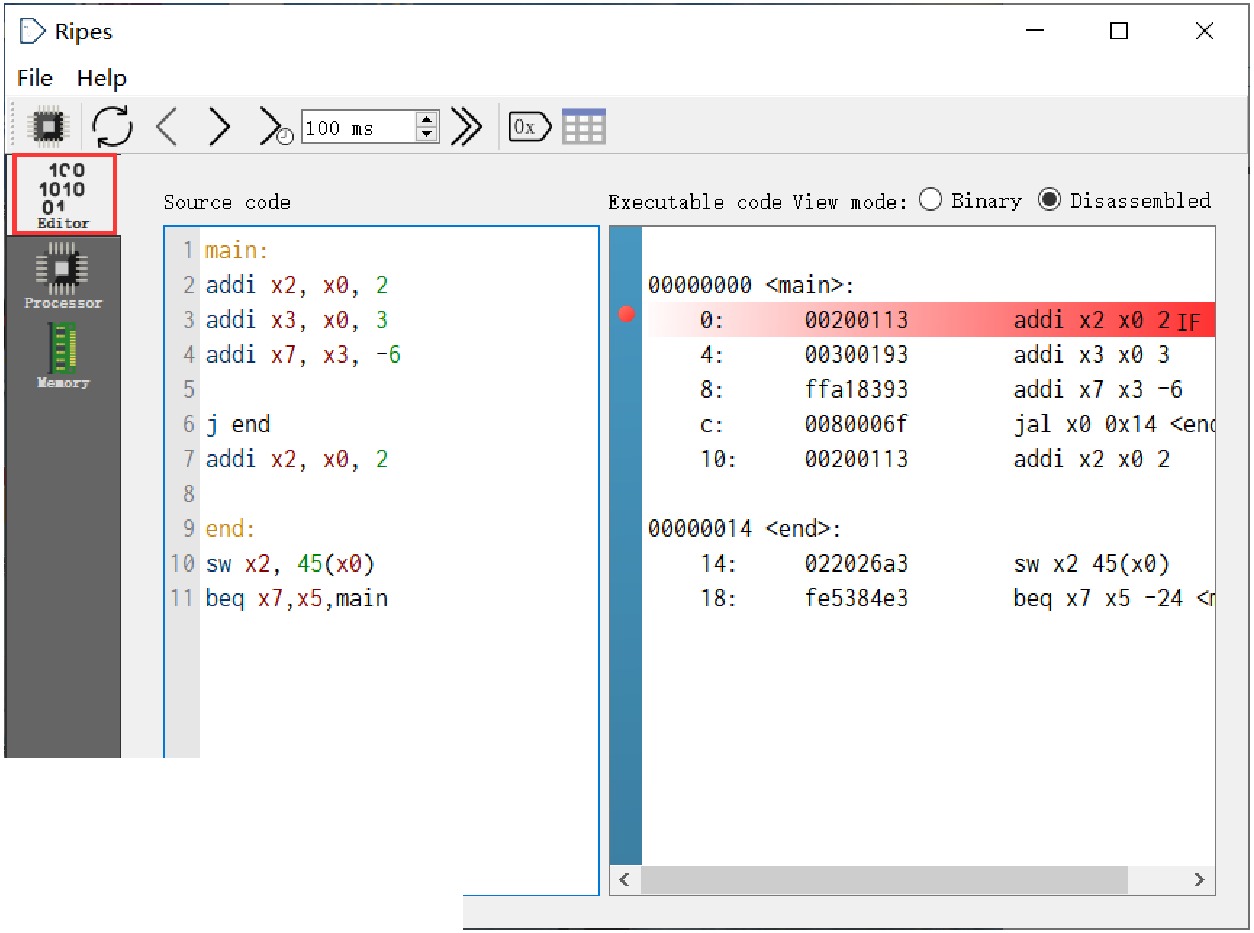
最少必须哪几种？

指令寻址有哪些方式？

寻址方式示例：操作数，下一条指令

- addi
- j/jal
- sw
- beq

- C程序
  - 存储模型
    - 名，地址，寄存器？
  - 执行模型：控制流
    - 行号？



Executable code View mode: ○ Binary ● Disassembled

Source code

```
main:
addi x2, x0, 2
addi x3, x0, 3
addi x7, x3, -6

j end
addi x2, x0, 2

end:
sw x2, 45(x0)
beq x7,x5,main
```

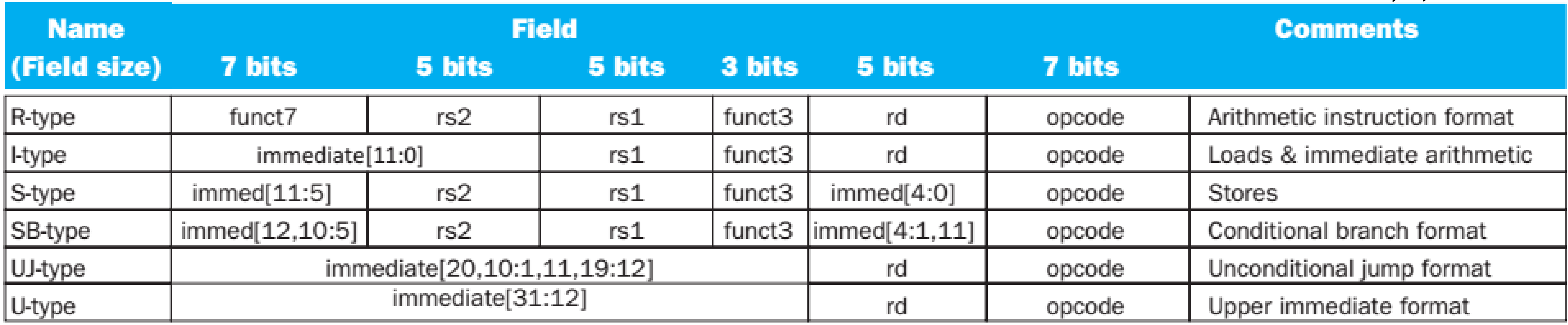
Executable code

```
00000000 <main>:
    0:     00200113     addi x2 x0 2 IF
    4:     00300193     addi x3 x0 3
    8:     ffa18393     addi x7 x3 -6
    c:     0080006f     jal x0 0x14 <end
    10:    00200113     addi x2 x0 2

00000014 <end>:
    14:    022026a3     sw x2 45(x0)
    18:    fe5384e3     beq x7 x5 -24 <m
```

```
   #include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

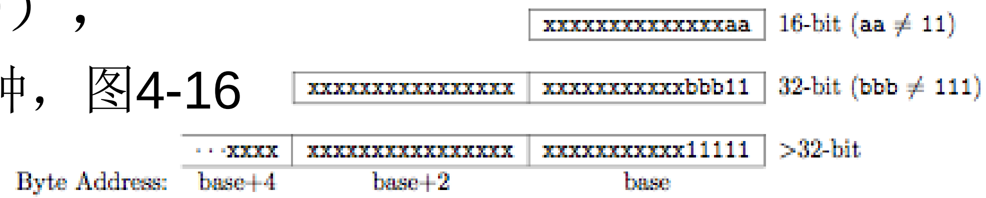
RISC-V指令格式与操作码

图4-16



Name (Field size)	Field 7 bits	 5 bits	 5 bits	 3 bits	 5 bits	 7 bits	Comments
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

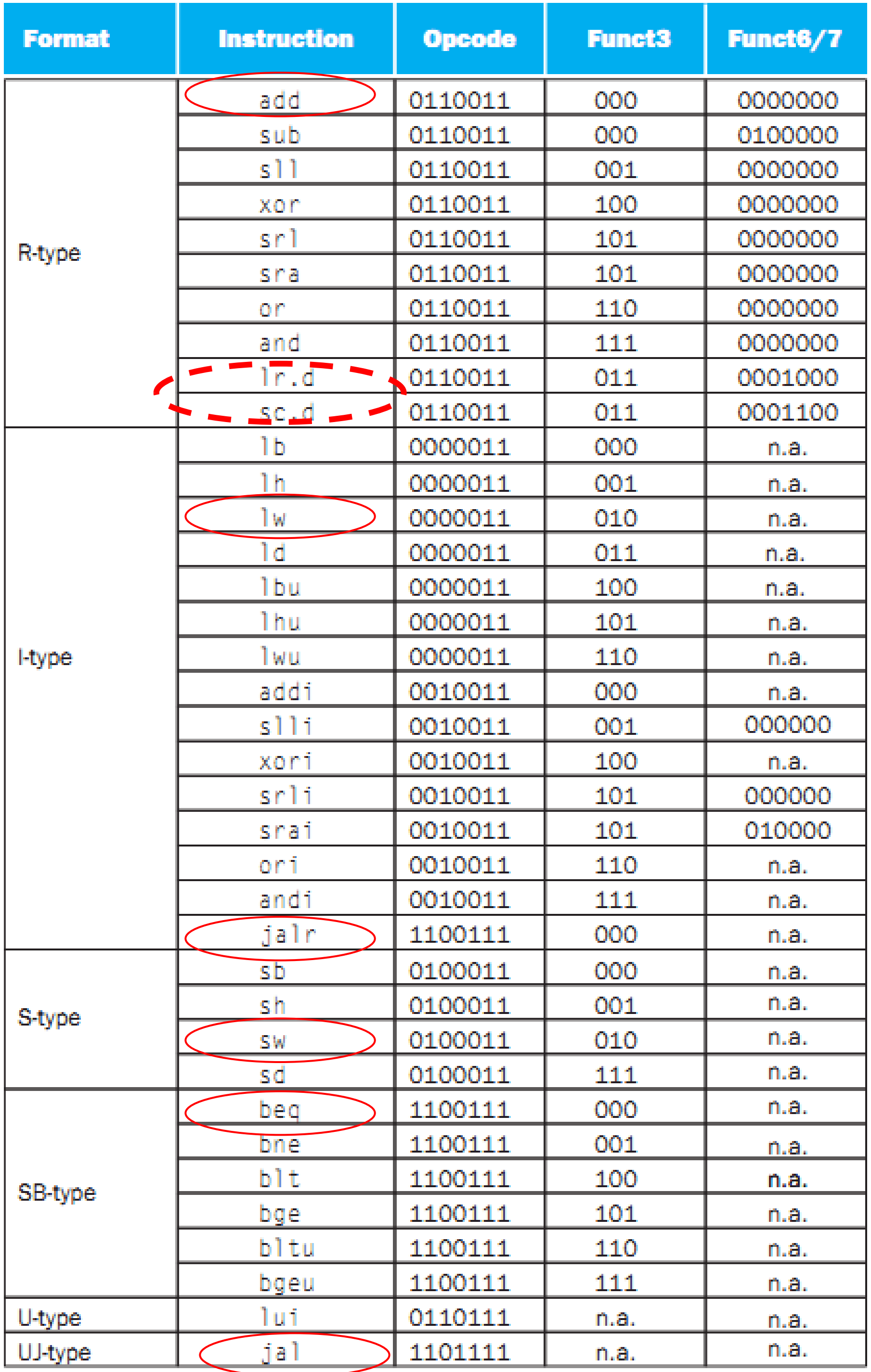
- 格式：基本R/I/S/U（图2-19），
  - 功能与类型（格式）绑定，6种，图4-16
  - 规整：Reg和Imm位置固定



- B/J-type的立即数域（循环移位）【$4.4.2，图4-17，4-18】
  - "减少数据通路上的MUX数量和MUX端口数，改善时钟周期"
- 有限支持可变长指令字：Opcode的bbbaa
  - Can support variable-length instructions【当前仅RV16/RV32】

RV指令操作码

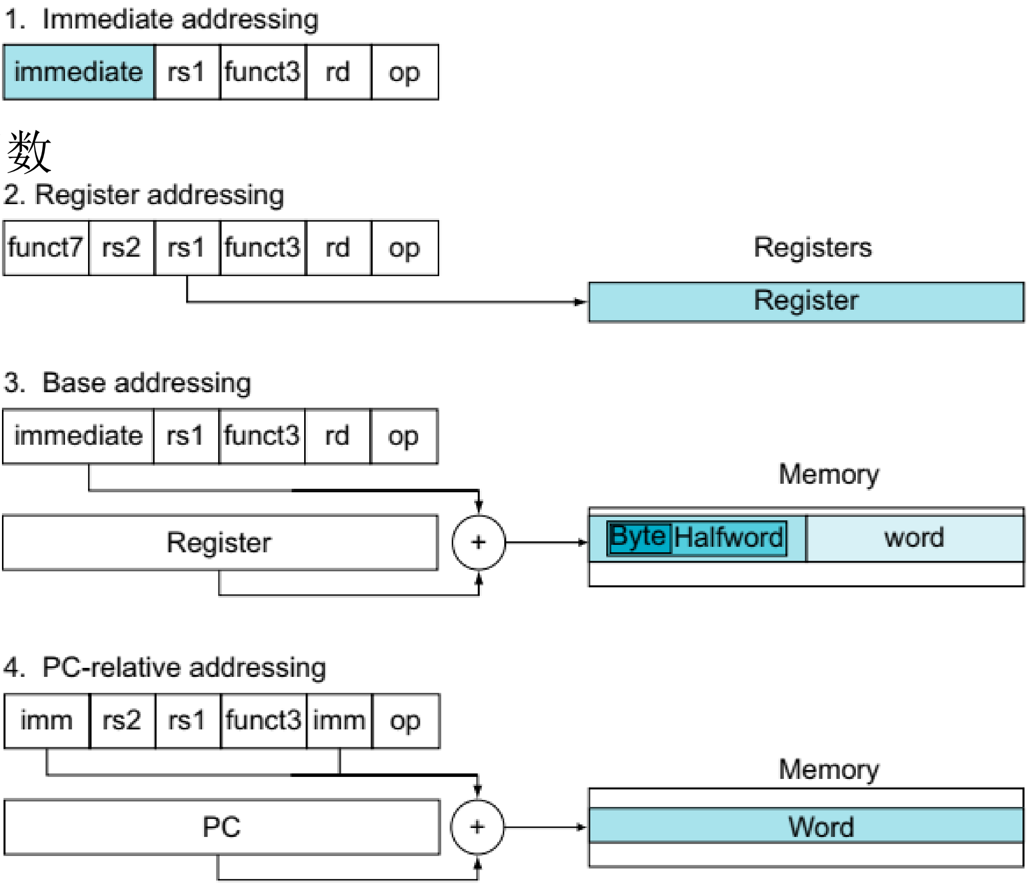
- RV32I：常用，图2-18
  - 按字长分类：访存指令
    - b，h，w，d
  - 按数据类型分类：访存，移位
    - i（立即数），u（无符号数）
  - 按指令格式分类
  - 示例：图3-12
- 典型指令：按功能分类
  - 同功能：op同，funct不同
  - 示例：ALU，访存，分支
    - add：R-type
    - addi：I-type
    - lw：load，I-type
    - sw：store，S-type
    - beq：SB-type
    - jal（UJ-type），jalr（I-type）



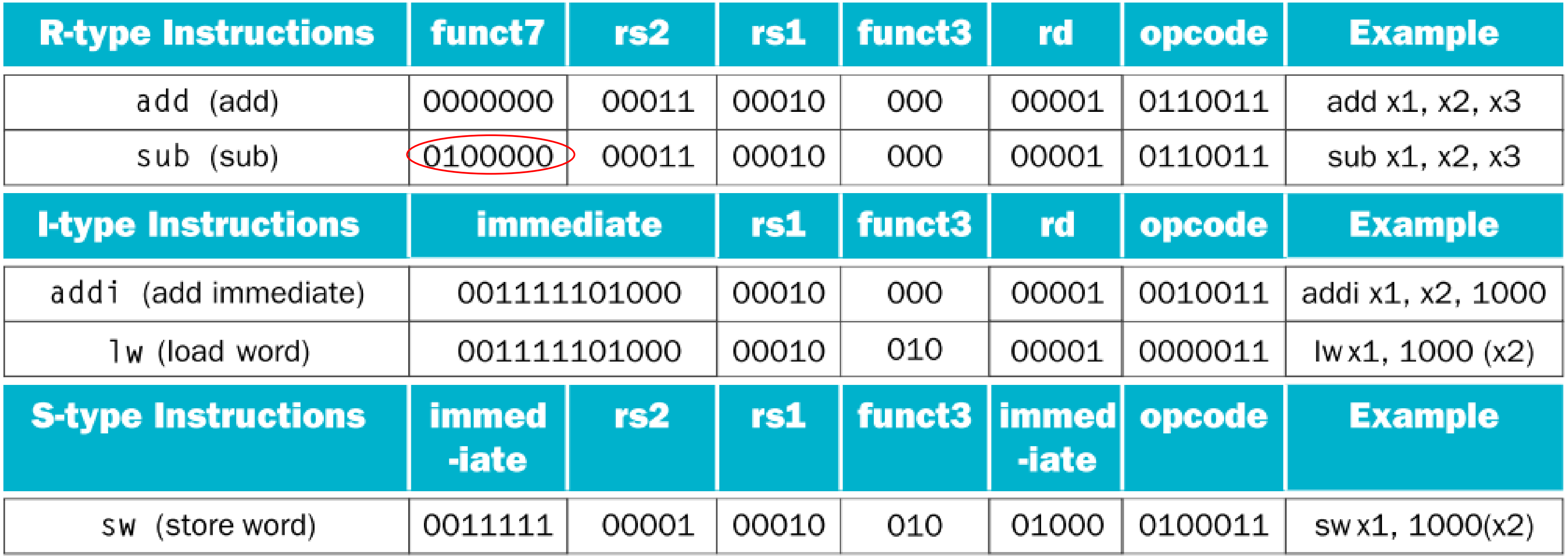
Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	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.

RISC-V寻址方式，图2-17

- 4(5)种：本质【HW view】3种，Imm，Reg，Base
- opr寻址方式
  - 立即寻址
    - 一条指令只能有一个立即数
    - addi x1, x2, 1000
  - 寄存器寻址
  - 基址寻址
    - 一条指令只能有一个
    - lw x1, 1000(x2)
- 指令字寻址方式
  - PC相对寻址
    - beq，jal
  - 间接跳转：indirect
    - jalr x0, 100(x1)



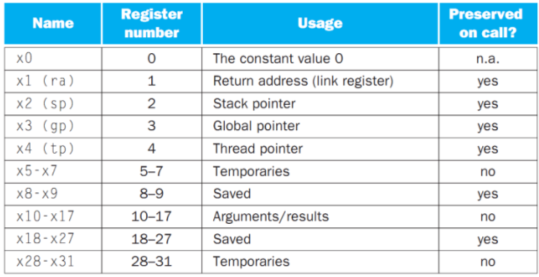
例：RV指令格式，寻址方式，图2-6



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
lw (load word)	001111101000		00010	010	00001	0000011	lw x1, 1000 (x2)
S-type Instructions	**immed-iate**	**rs2**	**rs1**	**funct3**	**immed-iate**	**opcode**	**Example**
sw (store word)	0011111	00001	00010	010	01000	0100011	sw x1, 1000(x2)

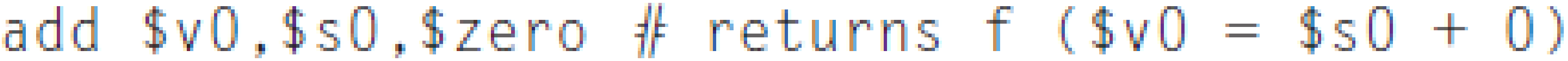
- 汇编指令操作数寻址方式表示
  - 寄存器寻址【编号，别名】，立即寻址【二/十/16进制】，基址寻址【1000(x2)】
  - **算逻**指令均为寄存器寻址或立即寻址，load/store为基址寻址
- 机器指令与汇编指令：源操作数和目的操作数的位置对应关系
  - 汇编指令：x2/x3为源操作数rs1/rs2，x1为目的操作数rd
  - 注意S-type：rs2 = x1（待写入MEM，源），rs1 = x2（基址），rs2 => mem[rs1+1000]
- **Load-Store架构**：ALU操作为Reg-Reg或Reg-Imm型，只能Load/Store访存（I/O）
  - add x1, x1, 1000(x2)　//非法

$zero：x0寄存器，$2.3.2



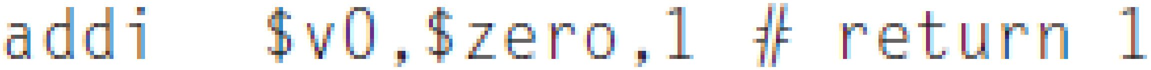
Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27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

- x0固定为“0”（hardwired）
  - data move：reg-reg



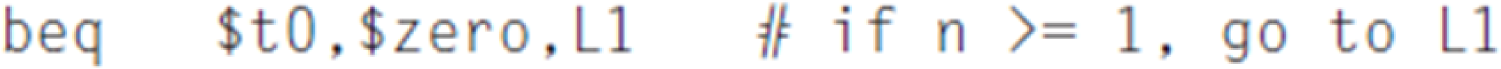
```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

  - 寄存器赋值



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

  - Compare



```
beq  $t0,$zero,L1   # if n >= 1, go to L1
```

  - Goto

    beq x0，x0，Exit   // ≠ jal x1, Exit?

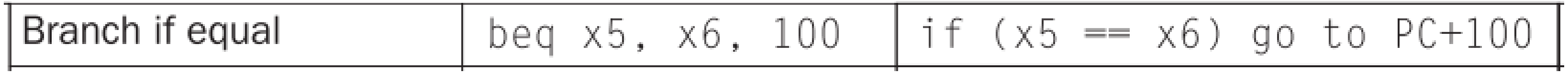
  - nop

    addi x0, x0, 0；占用资源最少的nop实现方法

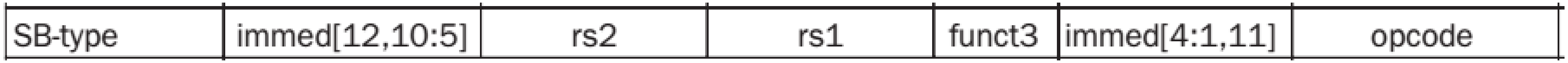
分支指令寻址方式：跳转范围？

$2.7，2.8，2.10.2，4.4

- 条件分支：相对分支，PC+12位offset（+/-）
  - beq，bne，…

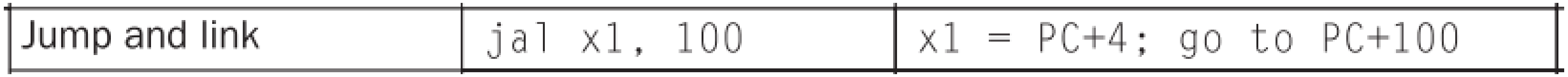


Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100

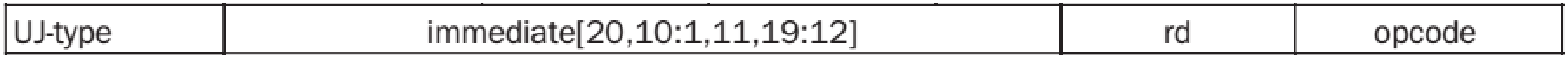


SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

- 无条件分支，+/-，过程调用call
  - jal：PC-relative分支， 20位offset，Calling，x1 = ra

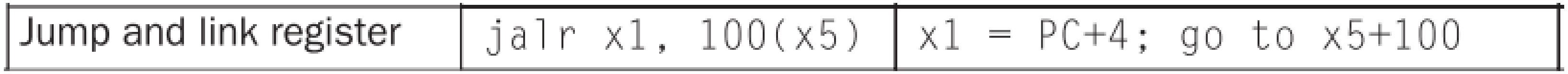


Jump and link	jal x1, 100	x1 = PC+4; go to PC+100

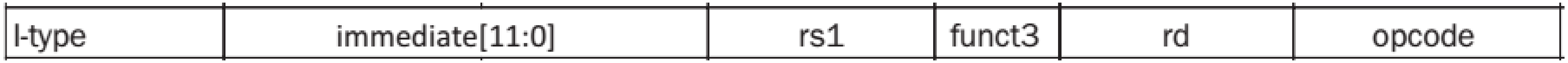


UJ-type	immediate[20,10:1,11,19:12]	rd	opcode

  - jalr：间接（indirect）跳转，12位offset，Return



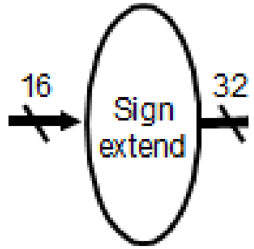
Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100

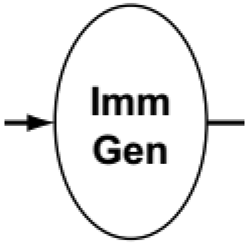


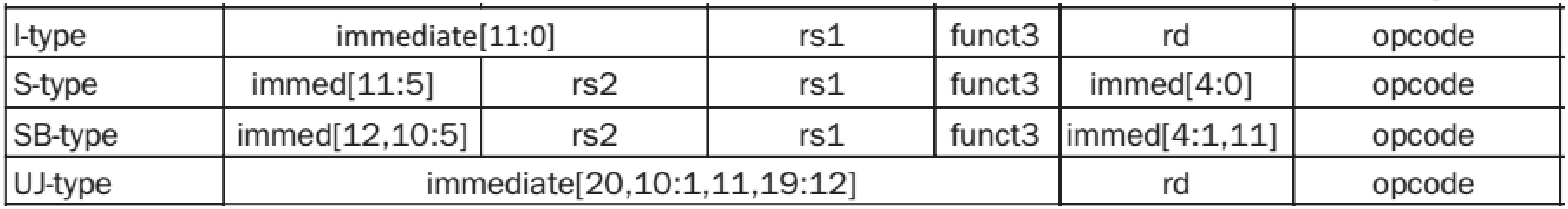
I-type	immediate[11:0]	rs1	funct3	rd	opcode

位扩展：短立即数=>32位立即数，$2.4

- 需求：I/S/SB/UJ-type，12位/20位短立即数=>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 goto nPC+7, else not taken
  - jal x0, 100；x0 = 0, goto PC+100
- 位扩展：高位填充
  - 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



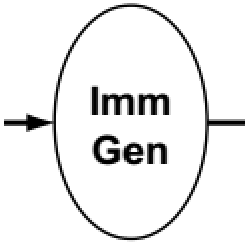




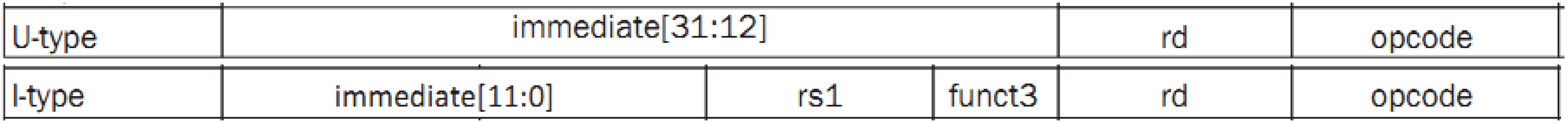
I-type	immediate[11:0]		rs1	funct3	rd	opcode
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode

32位常数生成，“双指令序列”法，了解

- 计算：大立即数，$2.10.1
  - 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]，[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，$2.10.2，——*$2.18的auipc？*
  - lui：取高20位
    - 例： lui x5，0x12345；
  - jalr：jump & link reg，I-type
    - 高20位+低12位，——基于x5间接跳转，不是PC相对寻址！
    - 例：jalr x1，100(x5)；x1=PC+4，goto x5+100



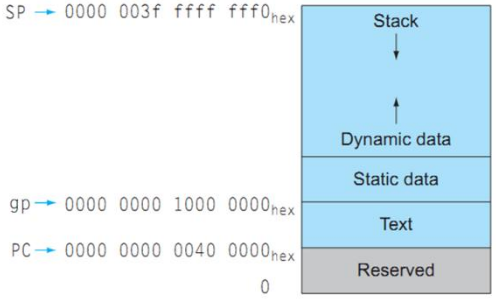
U-type	immediate[31:12]			rd	opcode
I-type	immediate[11:0]	rs1	funct3	rd	opcode

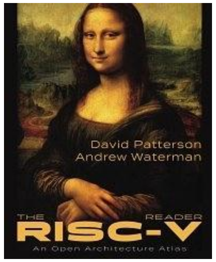
RV汇编程序结构：段式，《P&W》例

对照图2-25：
sort函数

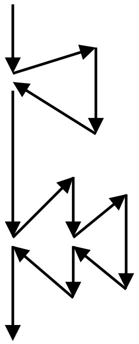
```
   .text                      # Directive: enter text section
   .align 2                   # Directive: align code to 2^2 bytes
   .globl main                # Directive: declare global symbol main
main:                         # label for start of main
   addi sp,sp,-16             # allocate stack 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
```

```c
#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



过程调用：调用，返回



- 例，数组排序：sort()，swap()，$2.13.1
  - call/jal，return/jalr

```
void sort (int v[], size_t int n)
{
    size_t 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);
          }
    }
}
```

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

Pass parameters and call	`addi x10, x21, 0      # first swap parameter is v`
	`addi x11, x20, 0      # second swap parameter is j`
	`jal x1, swap          # call swap`

```
swap:

  slli   x6, x11, 2   // reg x6 = k * 4
  add    x6, x10, x6  // reg x6 = v + (k * 4)
  lw     x5, 0(x6)    // reg x5 (temp) = v[k]
  lw     x7, 4(x6)    // reg x7 = v[k + 1]
  sw     x7, 0(x6)    // v[k] = reg x7
  sw     x5, 4(x6)    // v[k+1] = reg x5 (temp)
  jalr   x0, 0(x1)    // return to calling routine
```

调用：

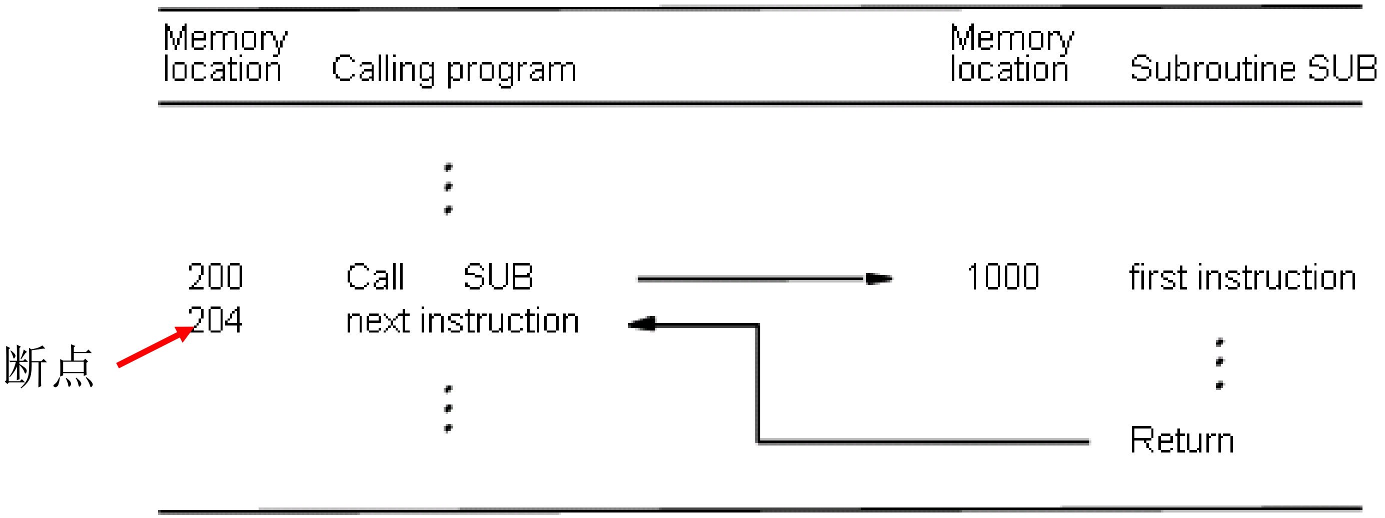
传参，保存断点，保存现场，控制权转移

现场：寄存器/栈/内存

返回：

回传结果，恢复现场，控制权转移

保存断点：单级call/return指令（伪指令）



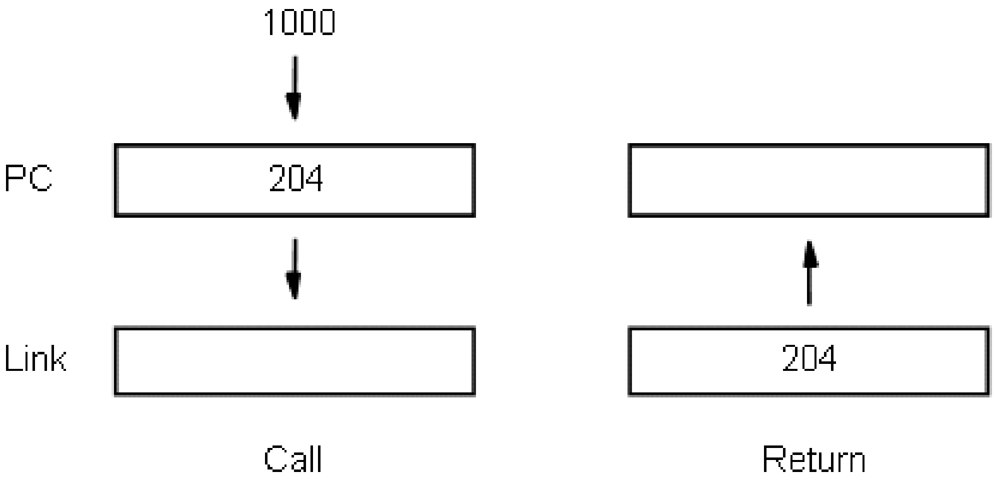
Memory location | Calling program | Memory location | Subroutine SUB

200 Call SUB → 1000 first instruction

204 next instruction

Return

断点



1000

PC 204

Link

Call

204

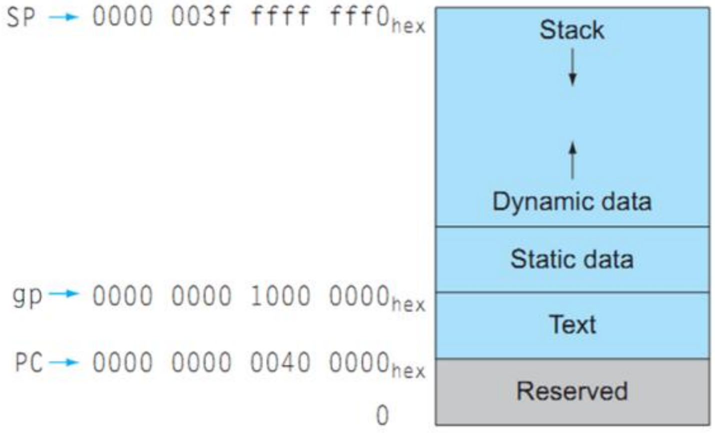
Return

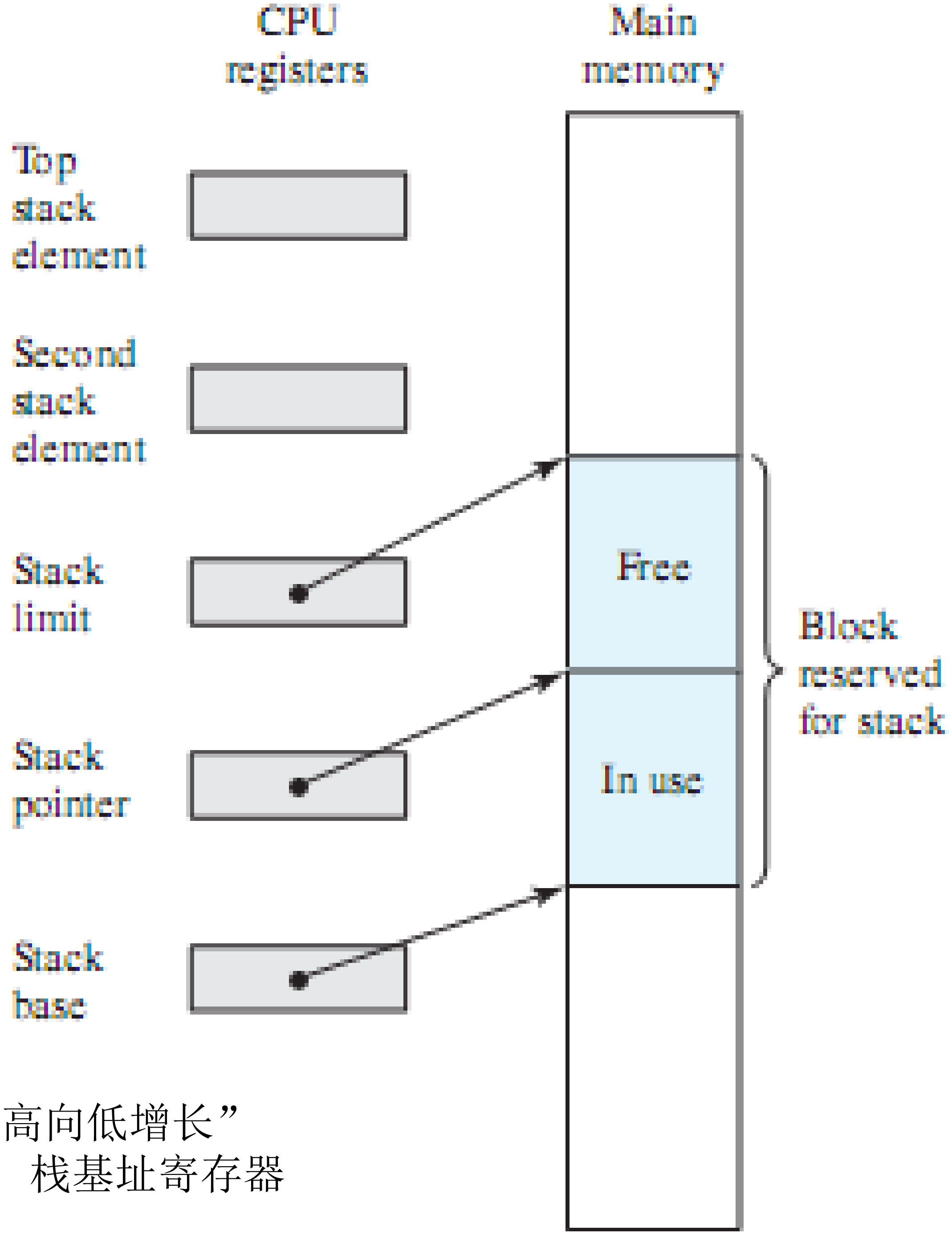
jal x1, 1000
jalr x0，0(x1)

链接寄存器x1 = npc
= 返回地址ra

Stack

多级过程调用：断点，传参


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



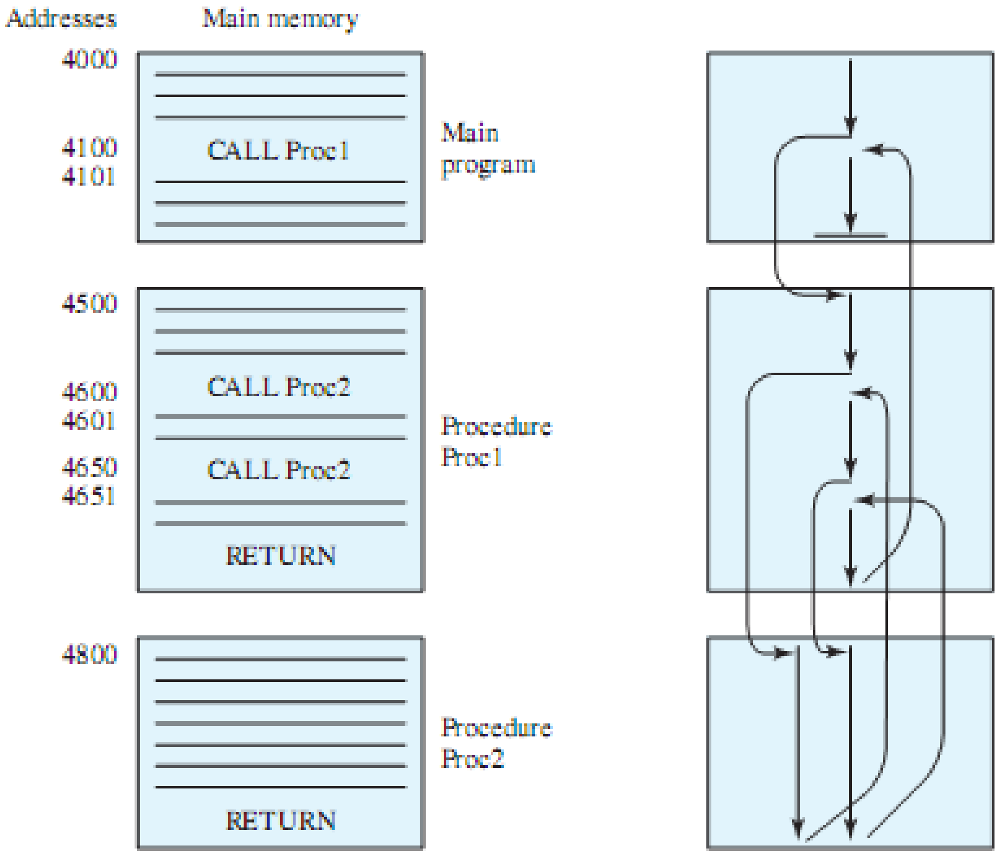
$2.8.1： “按照历史惯例，栈从高向低增长”
常见CPU寄存器：栈顶指针SP，栈基址寄存器

Use of Stack to Implement Nested Procedures

注意：

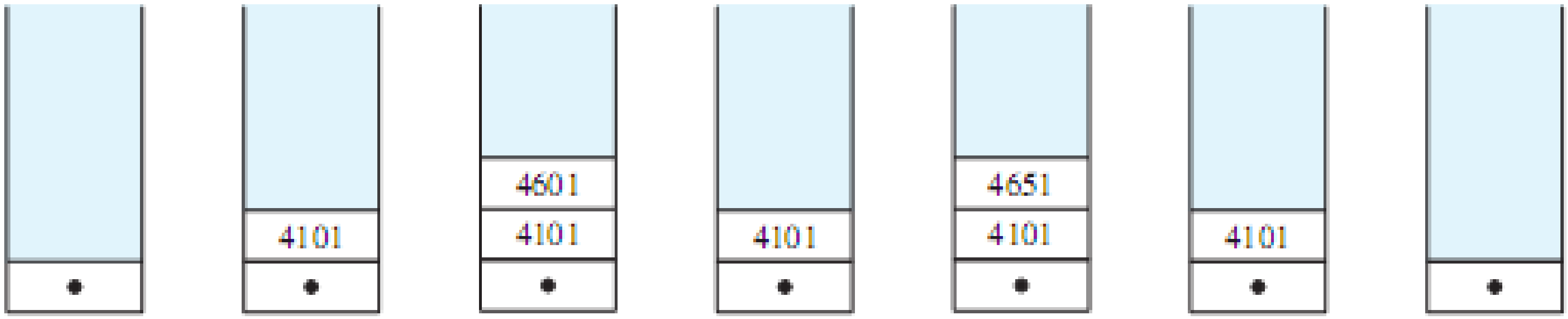
示例仅保存了断点，无传参

空堆栈/满堆栈?



(a) Calls and returns

(b) Execution sequence



(a) Initial stack contents

(b) After CALL Proc1

(c) Initial CALL Proc2

(d) After RETURN

(e) After CALL Proc2

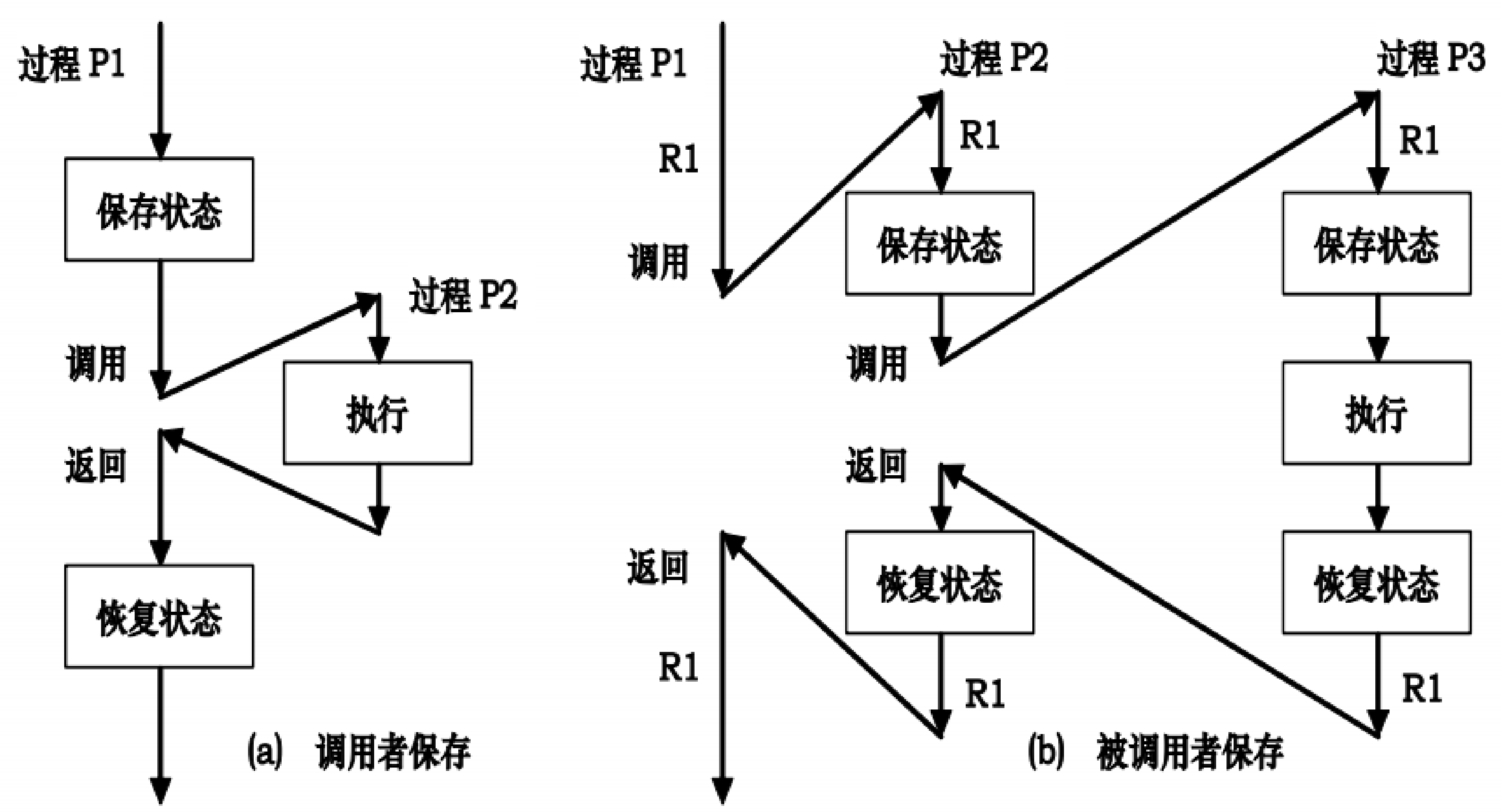
(f) After RETURN

(g) After RETURN

过程调用：保存现场（状态），两种机制，$2.8

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

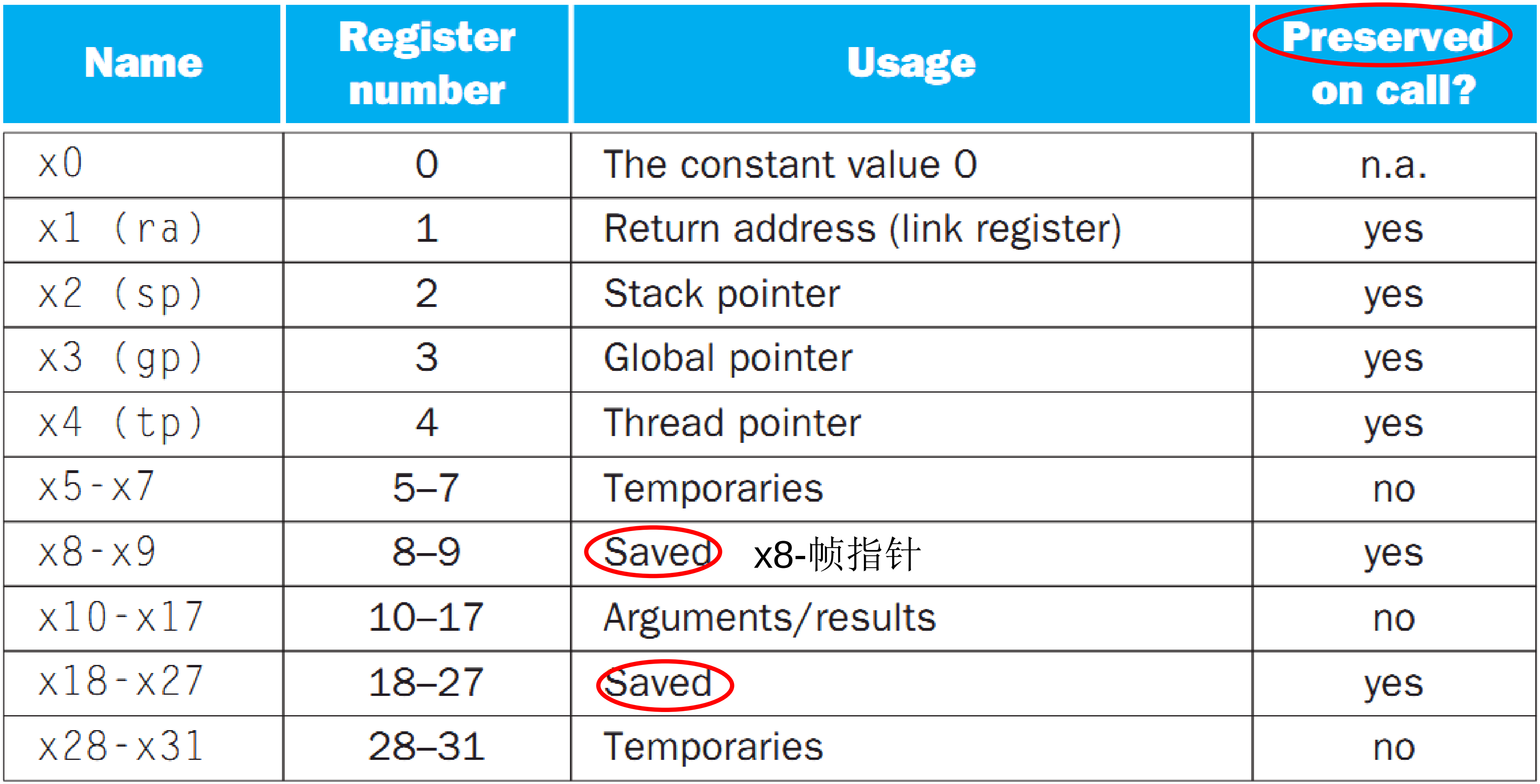
```
long long int fact (long long int n)
{		if (n < 1) return (1);
		else return (n * fact(n − 1));
}
```



(a) 调用者保存

(b) 被调用者保存

保存现场：RV的保留寄存器



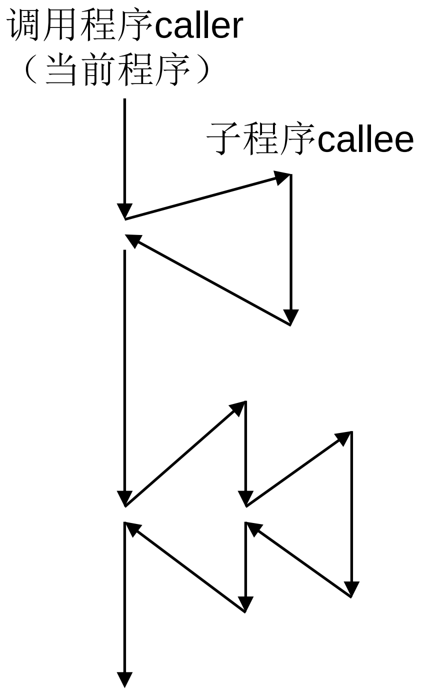
Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved x8-帧指针	yes
x10-x17	10–17	Arguments/results	no
x18-x27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

- Preserved：在函数调用和返回过程中应保持不变

图2-14

RV过程调用procedure calling

- 约定“由被调函数（子程序）保存状态（现场）”
  - 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——原子操作（由一条指令完成）
      - 保存断点（nPC）：返回点
      - 将控制交给子过程：使PC指向子过程入口
  - Callee
    - 保存现场（状态）
      - 保留寄存器：将过程内将要使用的通用Reg入栈
      - 程序的内存数据区？
    - 计算，并将结果放在caller可以访问的位置
    - 恢复现场：出栈
    - 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（jal）/return（jalr）
- 类型：叶子过程，嵌套过程，递归过程



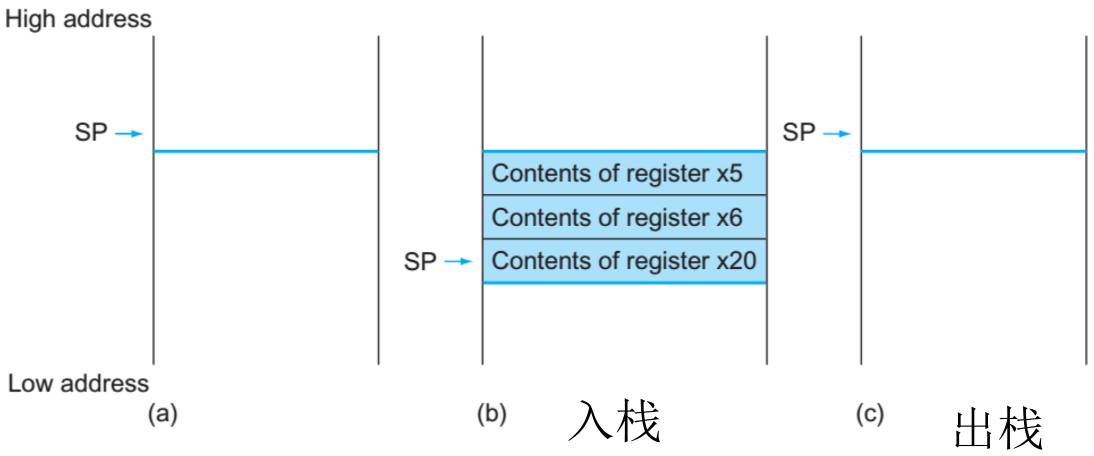
RV32堆栈操作：push/pop， 图2-10

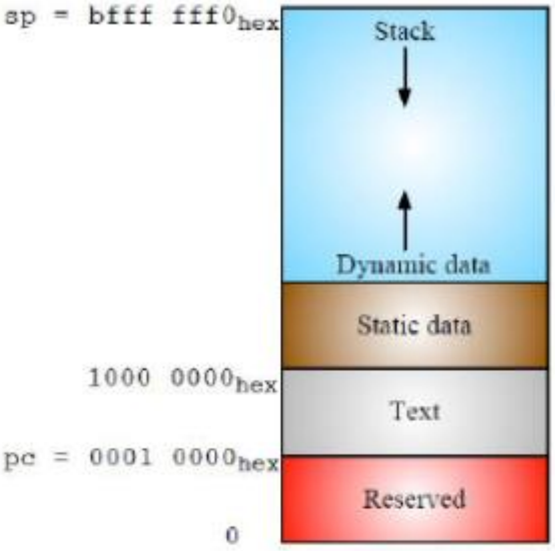
入栈

```
addi  sp, sp, -12            // adjust stack to make room for 3 items
sw    x5, 8(sp)      // save register x5 for use afterwards
sw    x6, 4(sp)              // save register x6 for use afterwards
sw    x20, 0(sp)             // save register x20 for use afterwards
```

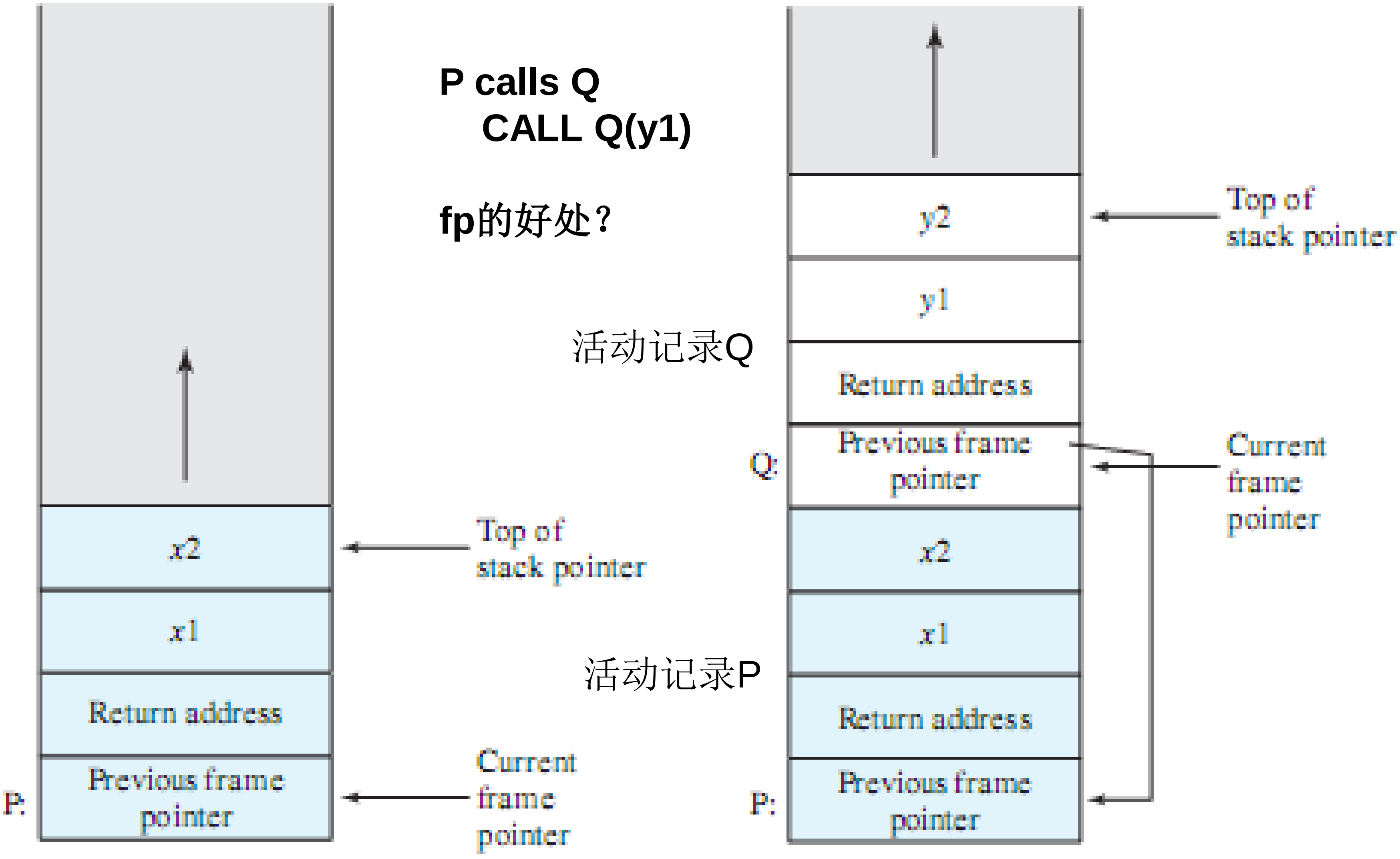
出栈

```
lw x20, 0(sp)    // restore register x20 for caller
lw x6, 4(sp)     // restore register x6 for caller
lw x5, 8(sp)     // restore register x5 for caller
addi sp, sp, 12  // adjust stack to delete 3 items
```





stack frame：“活动”记录，帧指针fp



(a) P is active

(b) P has called Q

P calls Q
CALL Q(y1)

fp的好处？

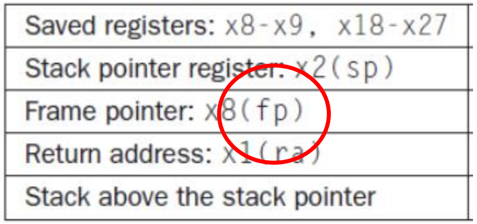
活动记录Q

活动记录P

RV的过程帧：活动记录（现场+）

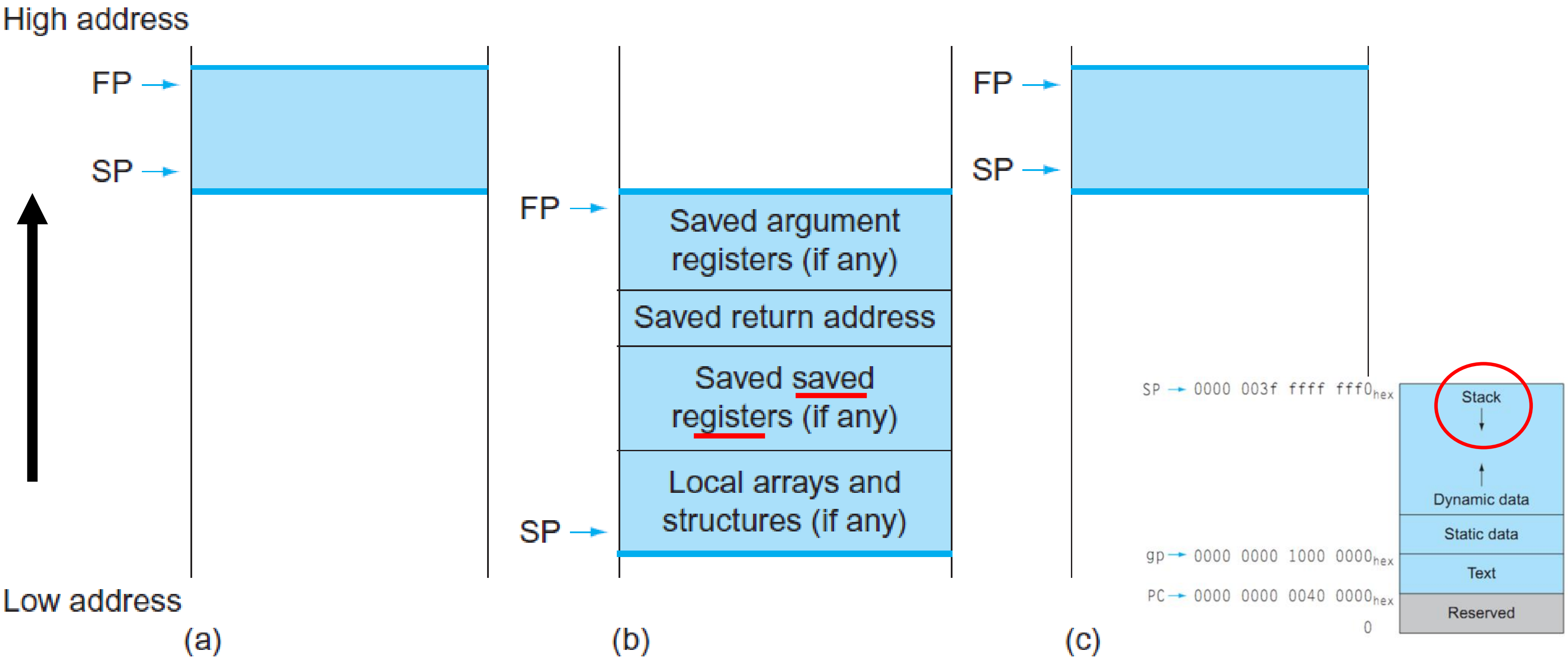
- 栈帧：参数，断点（返回地址），保留寄存器，局部变量
- 帧指针fp = 帧基址
  - 初始（a）：sp = fp
  - 结束（c）：sp = fp

图2-11



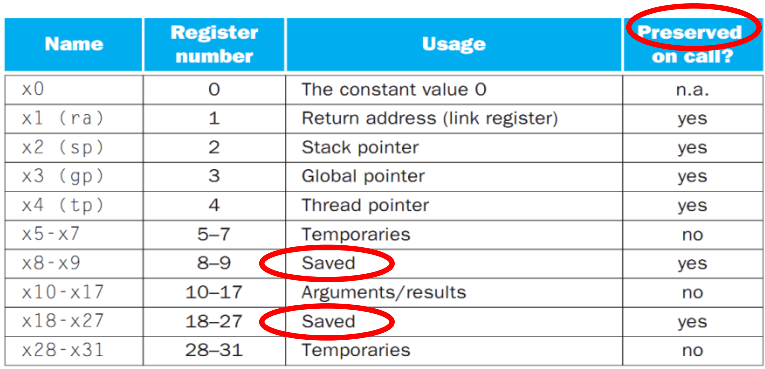
Saved registers: x8-x9, x18-x27
Stack pointer register: x2(sp)
Frame pointer: x8(fp)
Return address: x1(ra)
Stack above the stack pointer

图2-12


High address
FP
SP
Low address
(a)
FP
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
SP
(b)
FP
SP
(c)
SP → 0000 003f ffff fff0hex
Stack
Dynamic data
Static data
Text
Reserved
gp → 0000 0000 1000 0000hex
PC → 0000 0000 0040 0000hex
0

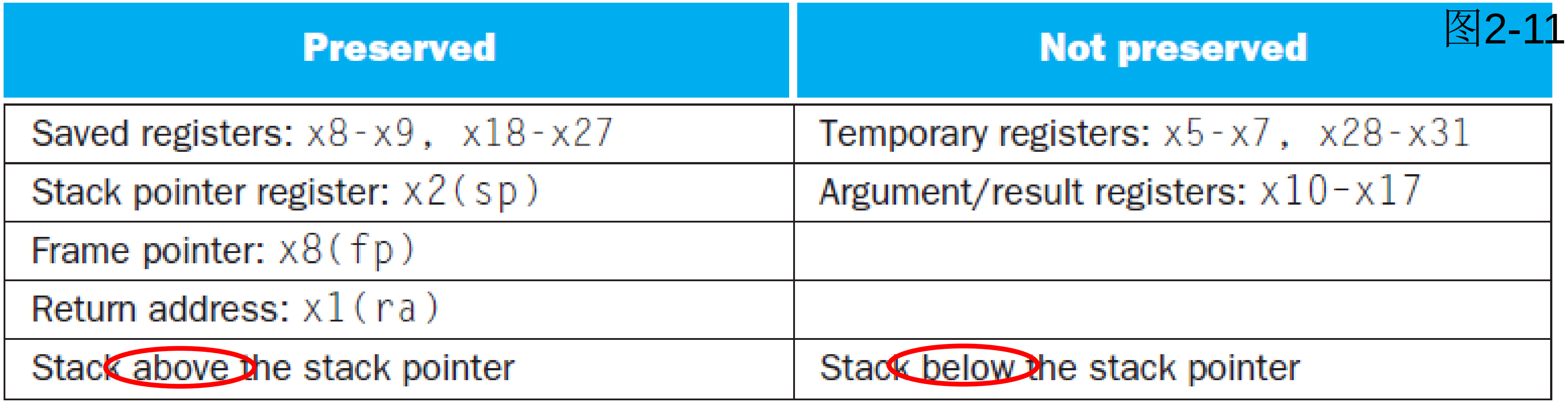
RV calling conventions，$2.8.2

- 传参和返回值：x10~x17【8个】
  - a0–a1: 函数变量或返回值
  - a2–a7：函数变量
- 断点ra：x1
- 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-and-link register：返回ra。断点（当前子函数的nPC）保存于x0（被抛弃）
- 现场保存策略：preserved由callee保存，notPreserved由caller负责



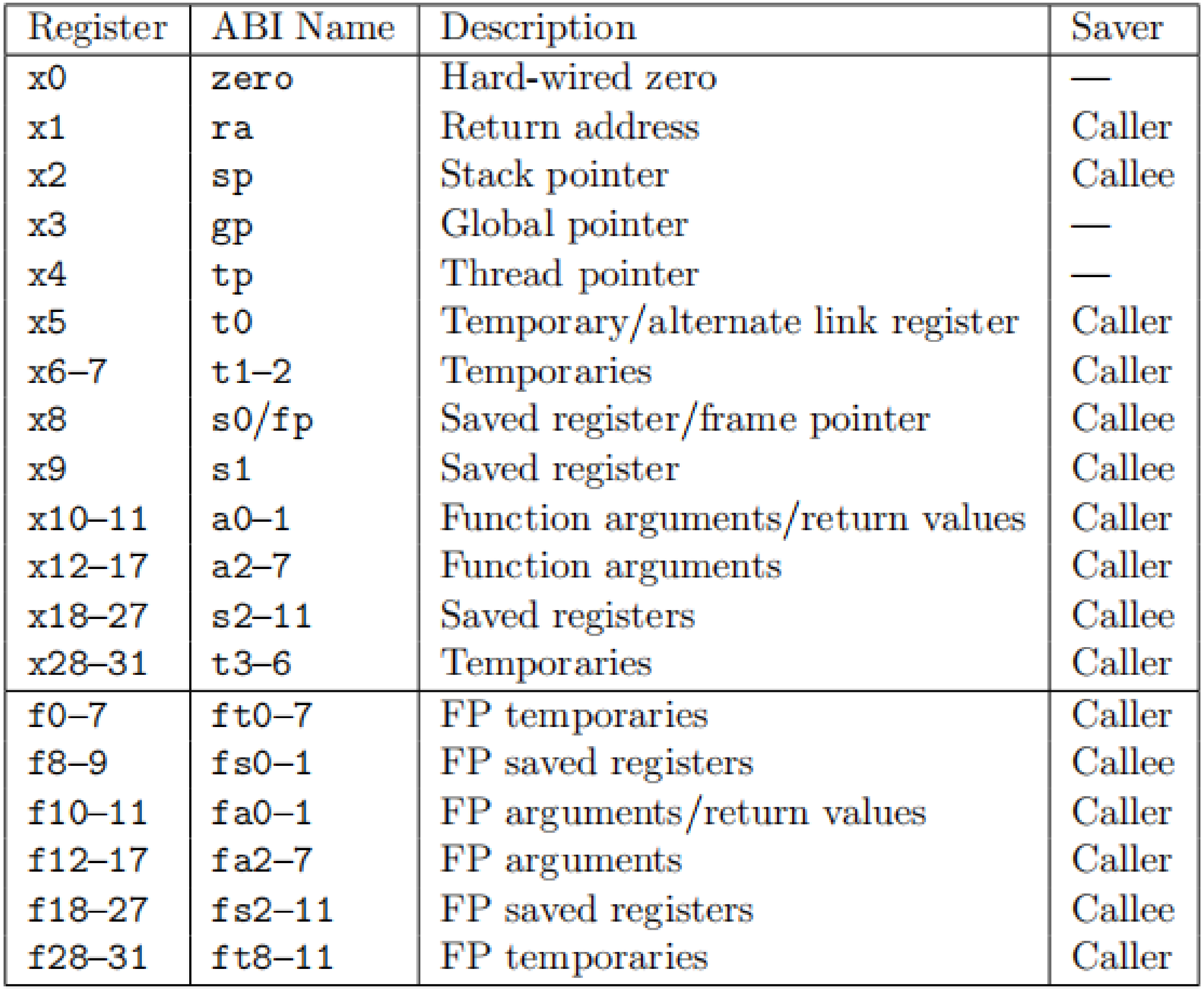
Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27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

图2-14



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

图2-11

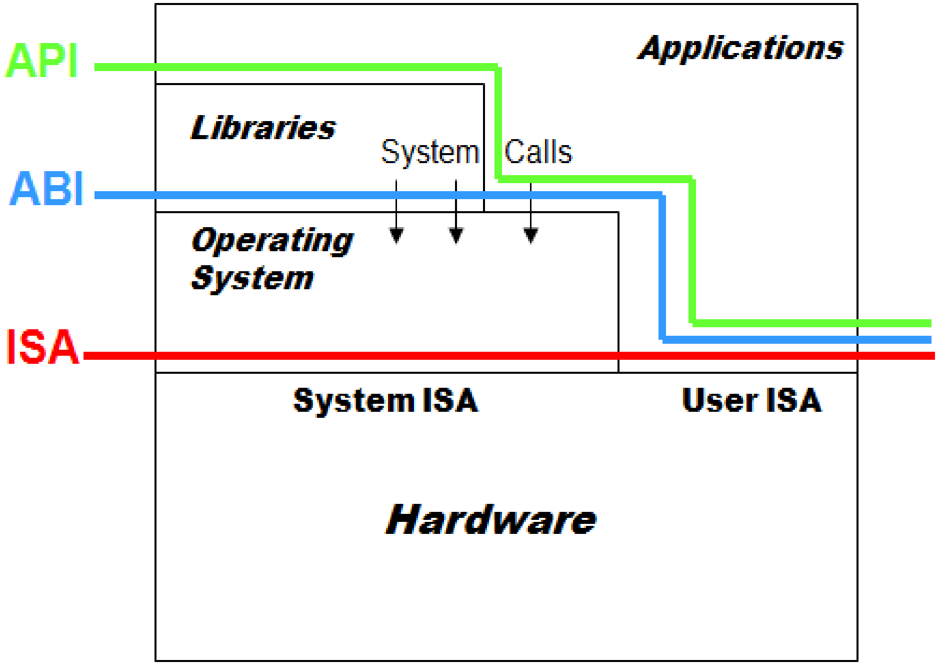


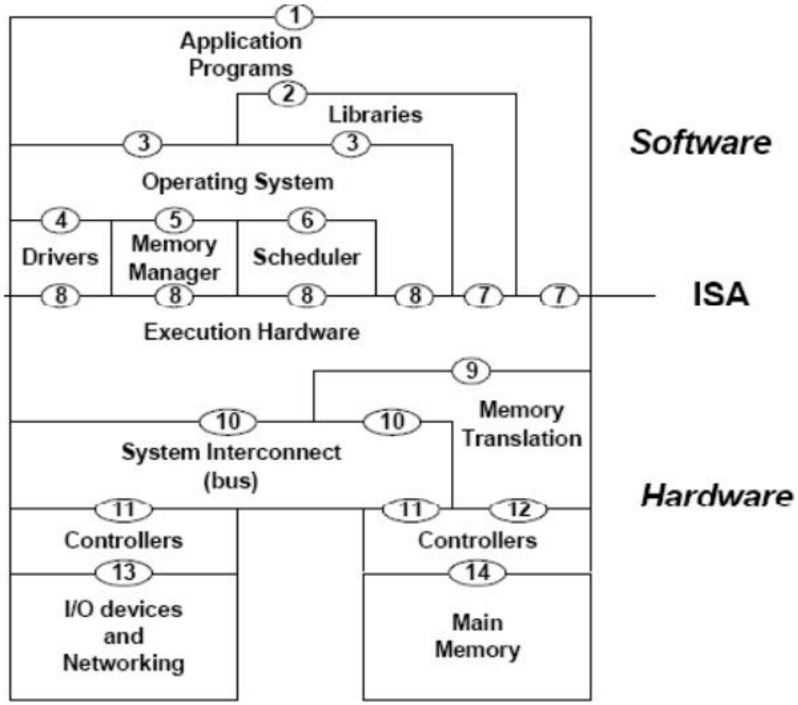
Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	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
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 25.1: Assembler mnemonics for RISC-V integer and floating-point registers, and their role in the first standard calling convention.

The RISC-V Instruction Set Manual，Volume I: Unprivileged ISA，2019
Ch25：RISC-V Assembly Programmer’s Handbook

Architecture（ISA）& Interfaces

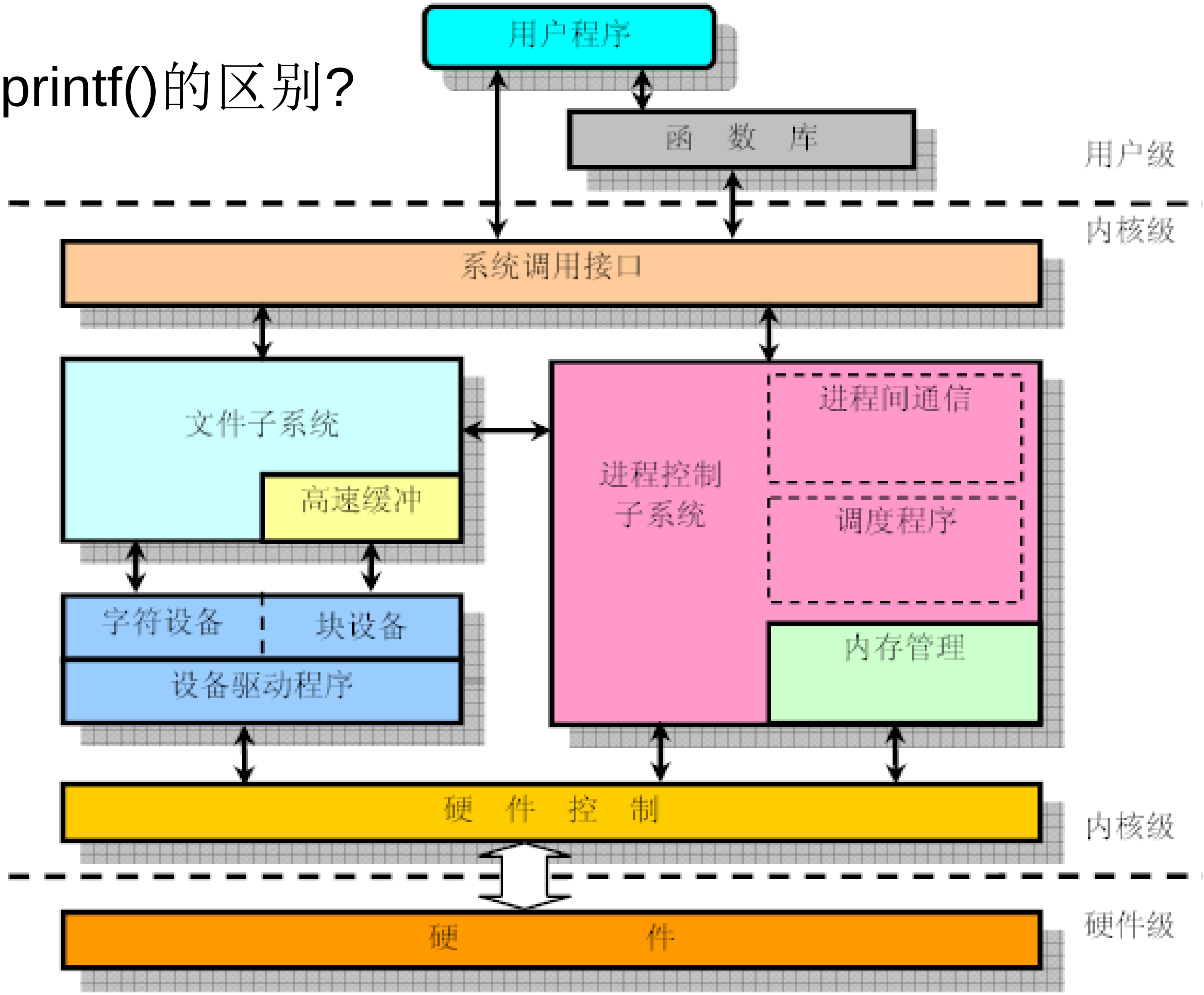




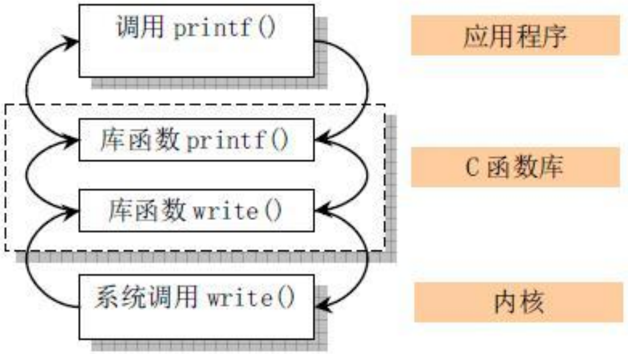
- **API** – application programming interface
- **ABI** – application binary interface = SysCall+UserISA（$1.4.3）
- **ISA** – instruction set architecture（$1.4.3）
  - 简称Architecture: formal specification of a system’s interface and the logical behavior of its visible resources.

C语言和OS服务：APIs、libc、syscalls?

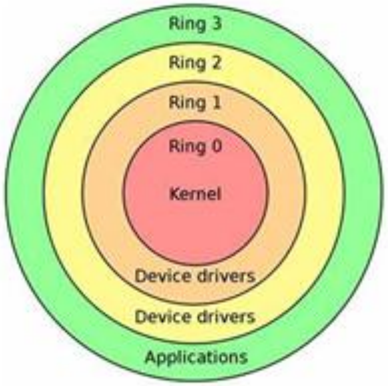
abs()与printf()的区别?



System calls

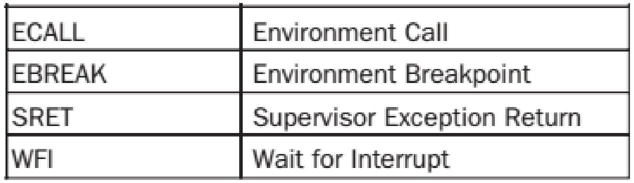


- OS服务：API
  - various names：trap/exception, svc, soft interrupt
- Why
  - Certain operations require specialized knowledge
    - I/O设备，PCIe总线，USB
  - protection：多任务共享，复用

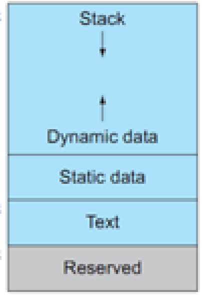


- What
  - A special machine instruction that causes an soft-interrupt/exception
    - 机器状态切换（protection）：用户态，内核态
    - 需保存PSW/CSR
  - RV：环境调用指令ecall
    - 系统态（supervisor，hypervisor，机器态）
    - Ripes提供哪些API服务?
  - x86系统调用（system calls）：int16，int32
    - BIOS，Windows：显示、键盘、磁盘、文件、打印机、时间

RV ecall指令，图5-47

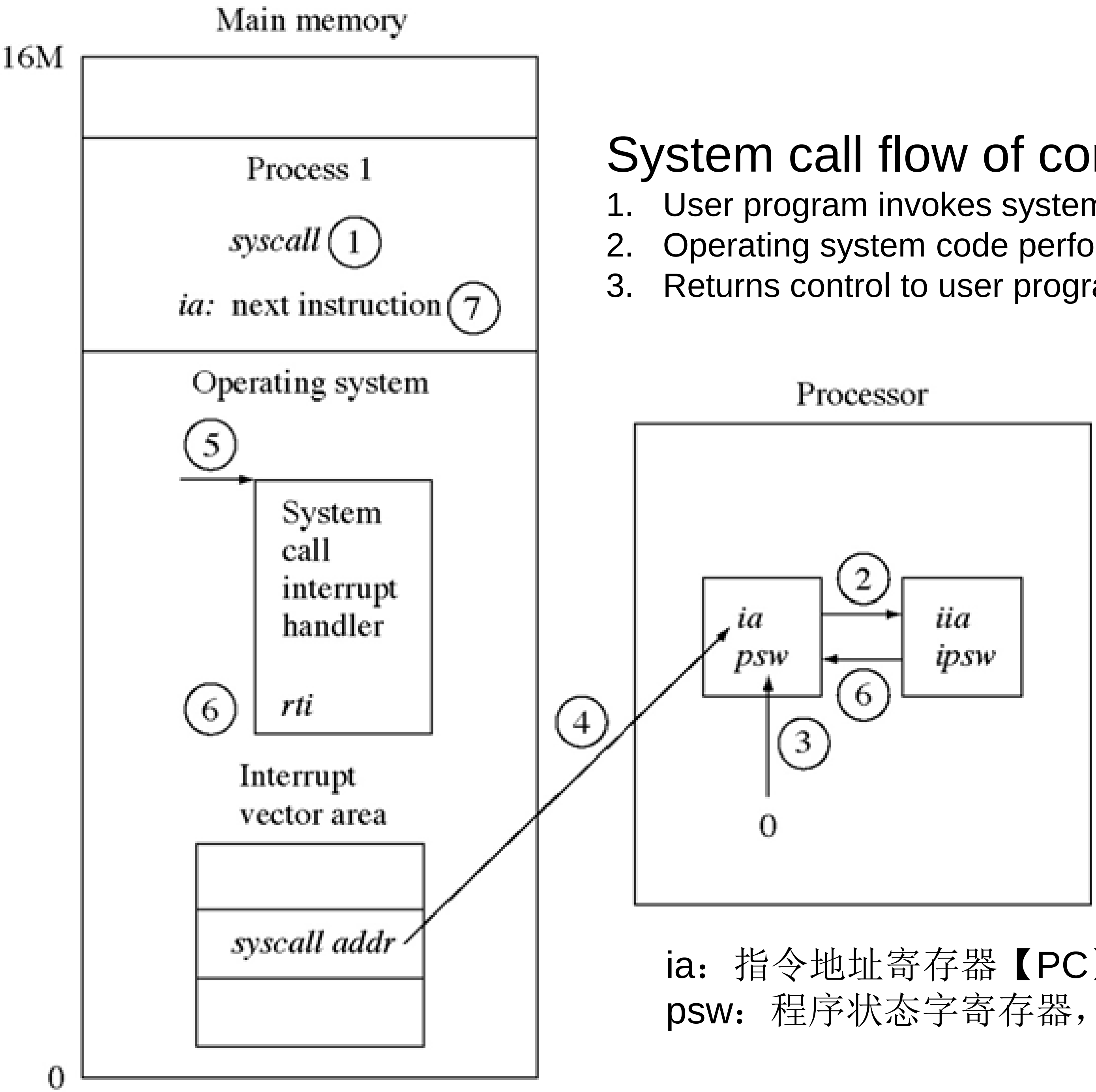


ECALL	Environment Call
EBREAK	Environment Breakpoint
SRET	Supervisor Exception Return
WFI	Wait for Interrupt



System call flow of control

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



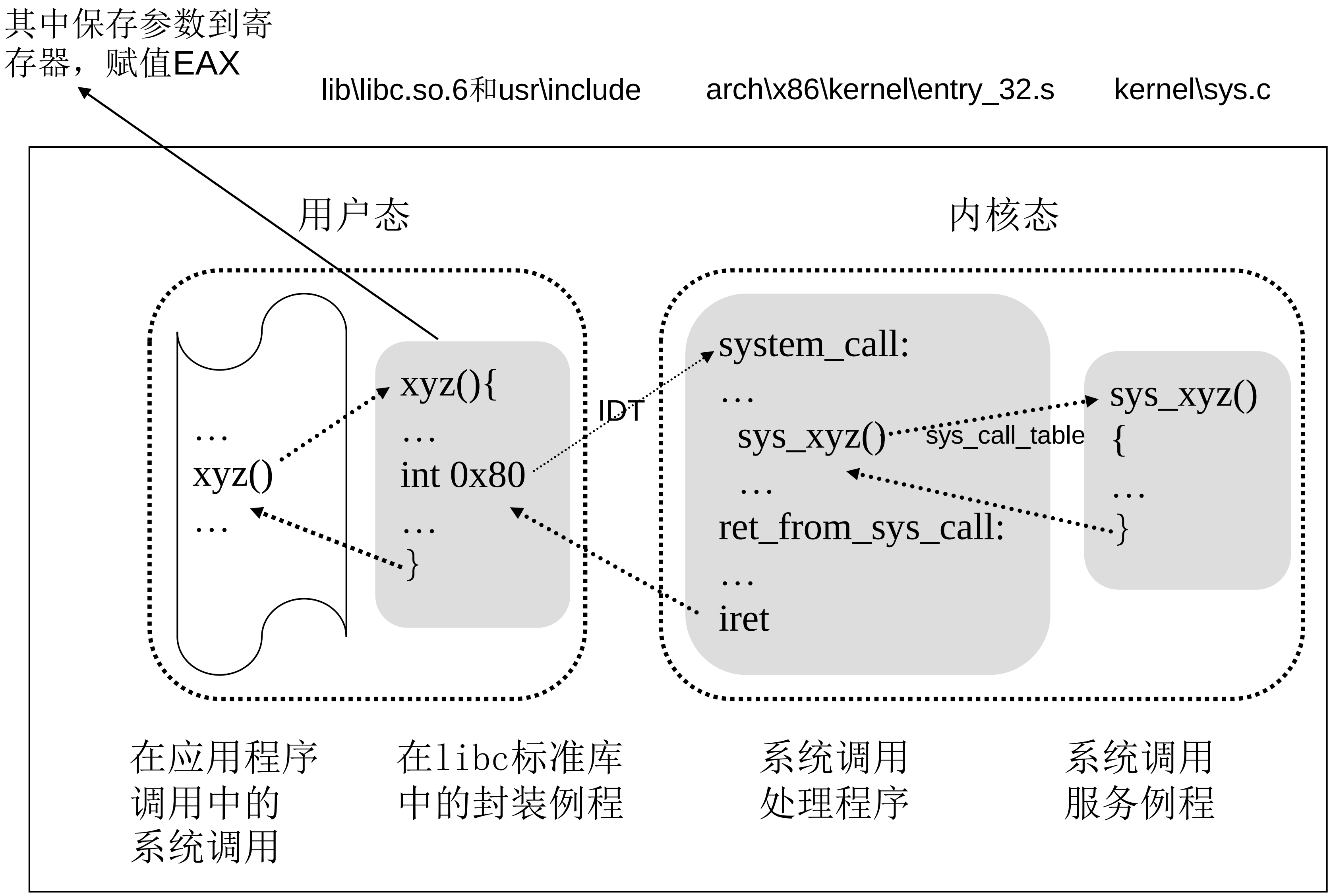
2/3/4由syscall指令完成！
 2：保存断点和psw
 3：psw赋0，进入内核态
 4：syscall入口赋给PC

5：进入系统函数
6/7由中断返回指令iret完成
 6：恢复调用前状态
 7：从断点处继续执行

API存储位置：
sys_call_table（interrupt vector area）
sys_call_ISR（system call interrupt handler）

ia：指令地址寄存器【PC】，保存于iia寄存器
psw：程序状态字寄存器，保存于ipsw寄存器

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()
{
…
}
```

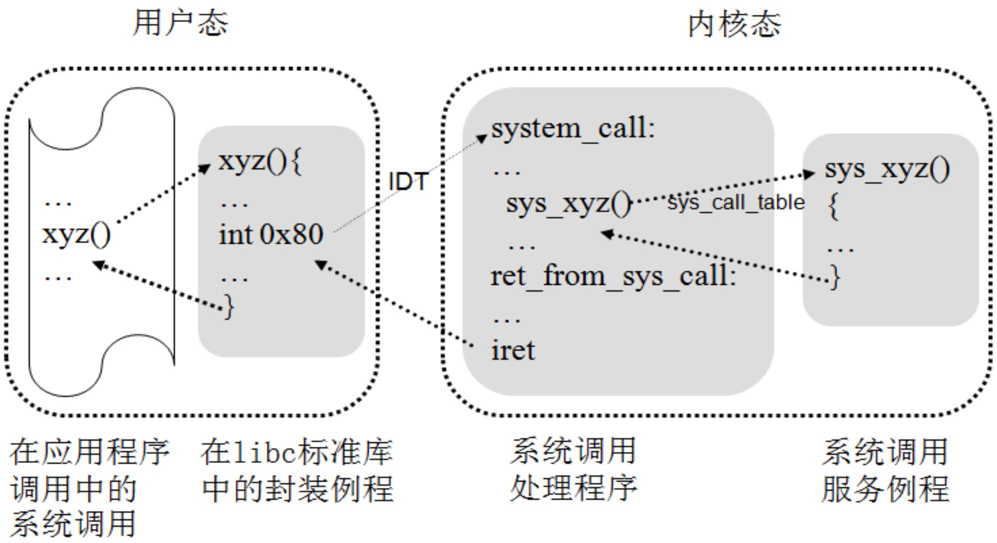
在应用程序调用中的系统调用

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

陷阱/系统调用：应用程序调用OS服务



用户态

内核态

```
…
xyz()
…
```

```
xyz(){
…
int 0x80
…
}
```

IDT

```
system_call:
…
  sys_xyz()
…
ret_from_sys_call:
…
iret
```

sys_call_table

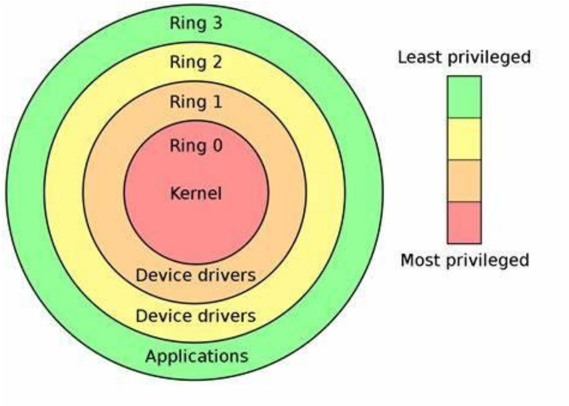
```
sys_xyz()
{
…
}
```

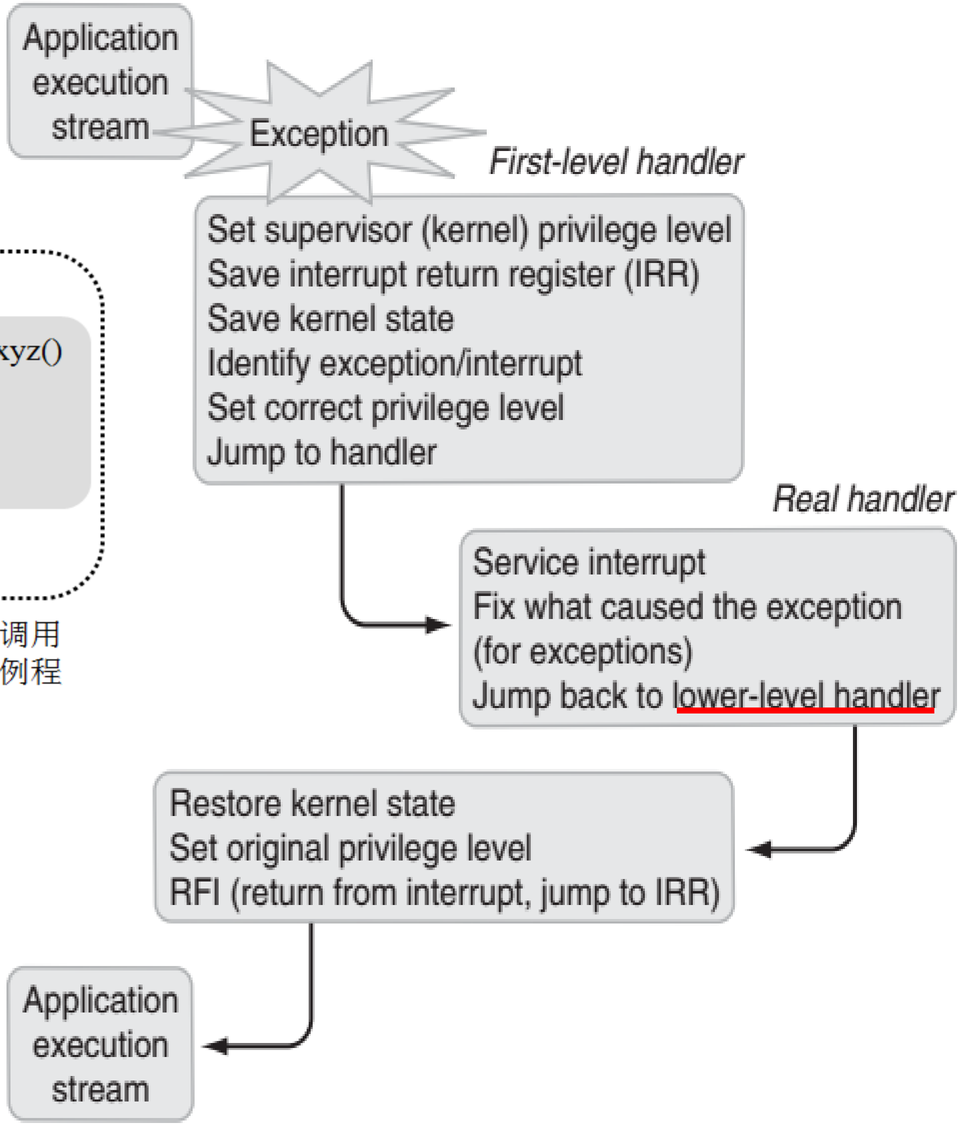
在应用程序调用中的系统调用

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





Application execution stream

Exception

First-level handler

Set supervisor (kernel) privilege level
Save interrupt return register (IRR)
Save kernel state
Identify exception/interrupt
Set correct privilege level
Jump to handler

Real handler

Service interrupt
Fix what caused the exception (for exceptions)
Jump back to lower-level handler

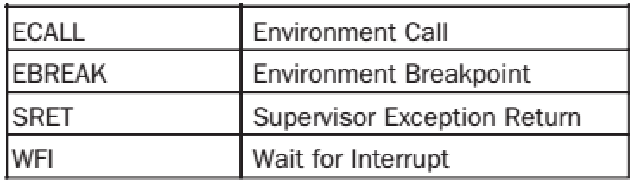
Restore kernel state
Set original privilege level
RFI (return from interrupt, jump to IRR)

Application execution stream

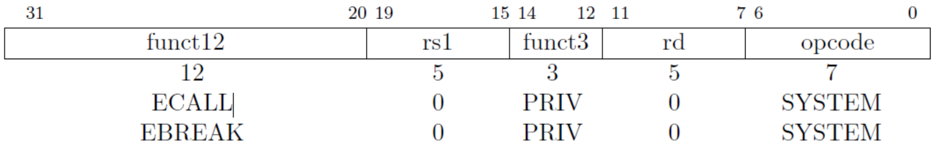
RV ecall指令

- RISCV特权级：User/Supervisor/Hypervisor/Machine
- 在 U-mode（用户模式）下执行 ecall 指令
  - 特权级由 User-mode提升为 Supervisor-mode（内核态）
  - 当前指令地址保存到sepc特权寄存器
  - 设置 scause 特权寄存器
  - 跳转到 stvec 特权寄存器指向的指令地址
  - ecall通过寄存器a0-a7传递参数
    - a0和a1不仅传入，还要传出：a0错误码，a1返回值
    - a6传递功能ID，a7传递扩展ID；
- 异常返回：mret，sret，uret，——指令格式？
  - 异常程序完成后，从异常服务程序中退出，并返回主程序

图5-47

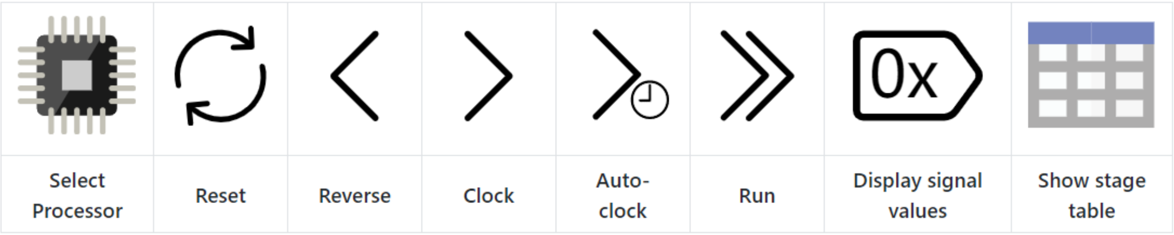


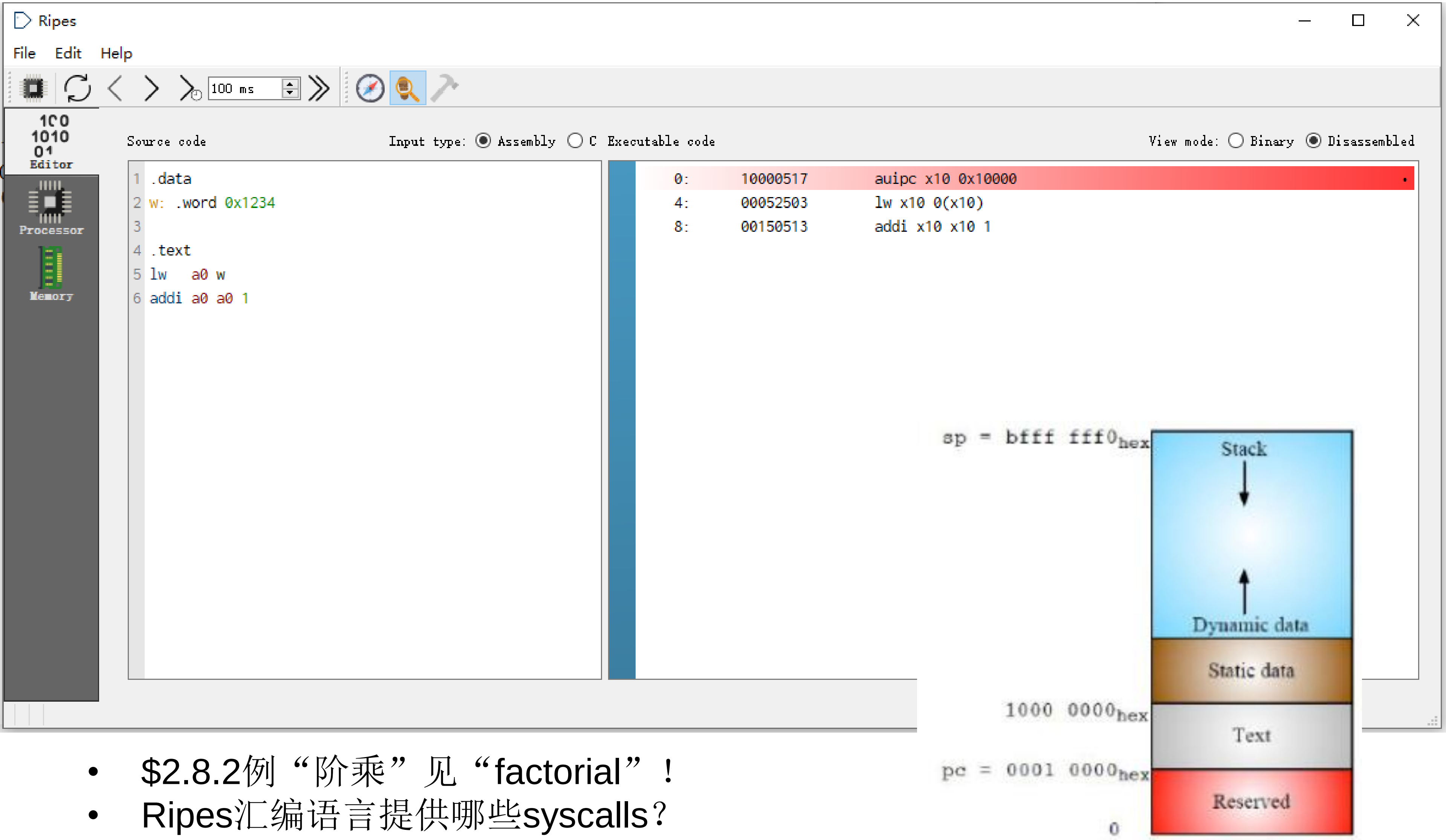
ECALL	Environment Call
EBREAK	Environment Breakpoint
SRET	Supervisor Exception Return
WFI	Wait for Interrupt



31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

Ripes汇编





Source code Input type: ● Assembly ○ C

```
.data
w: .word 0x1234

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

Executable code View mode: ○ Binary ● Disassembled

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

$sp = \text{bfff fff0}_{hex}$

1000 0000_{hex}

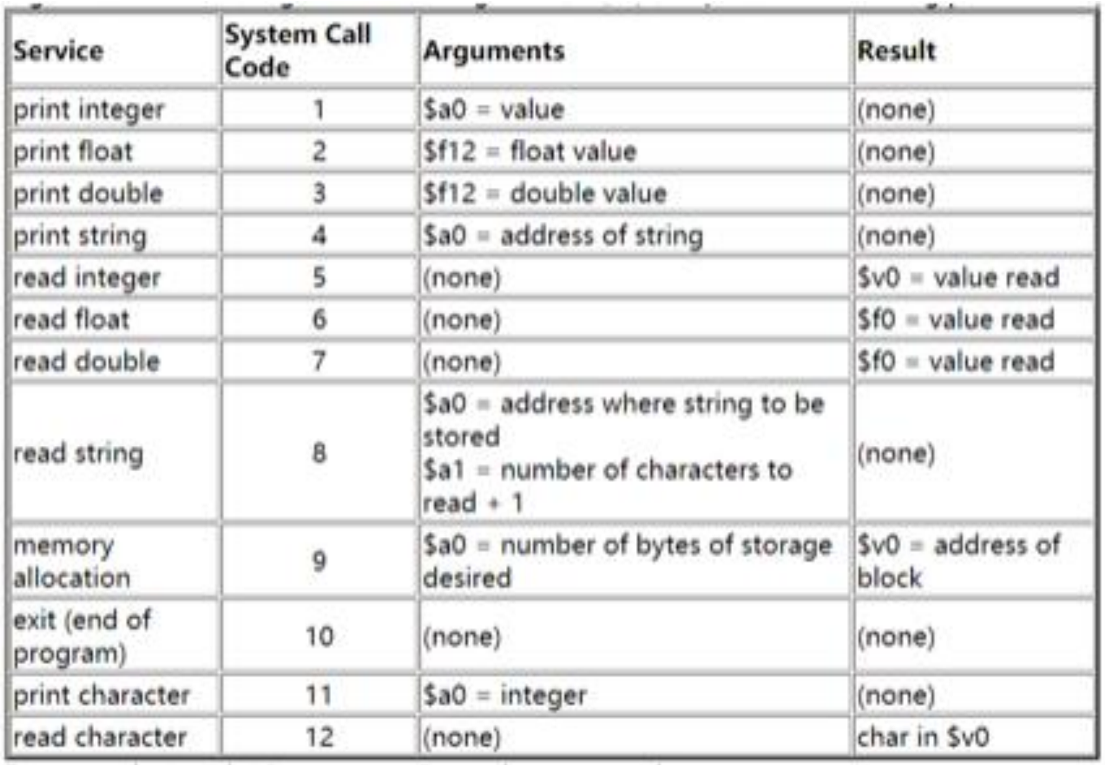
$pc = \text{0001 0000}_{hex}$

0

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

SPIM的系统调用：COD4附录B.9

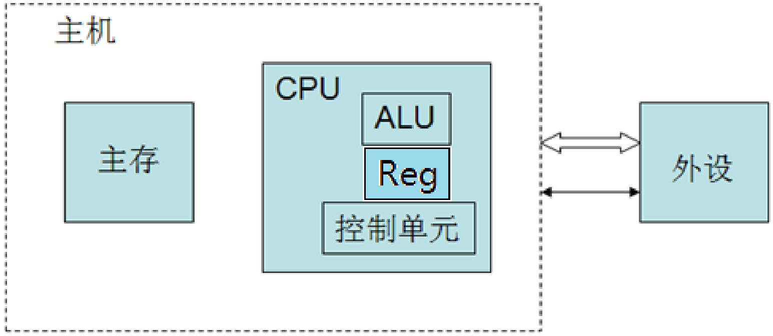
- SPIM：MIPS-32仿真器
  - 汇编程序调试、执行
  - 标准设备I/O服务
- SYSCALL Step
  - $v0=srv#
  - $a0~3=arg
  - syscall
  - $v0=返回值
- Ripes类似

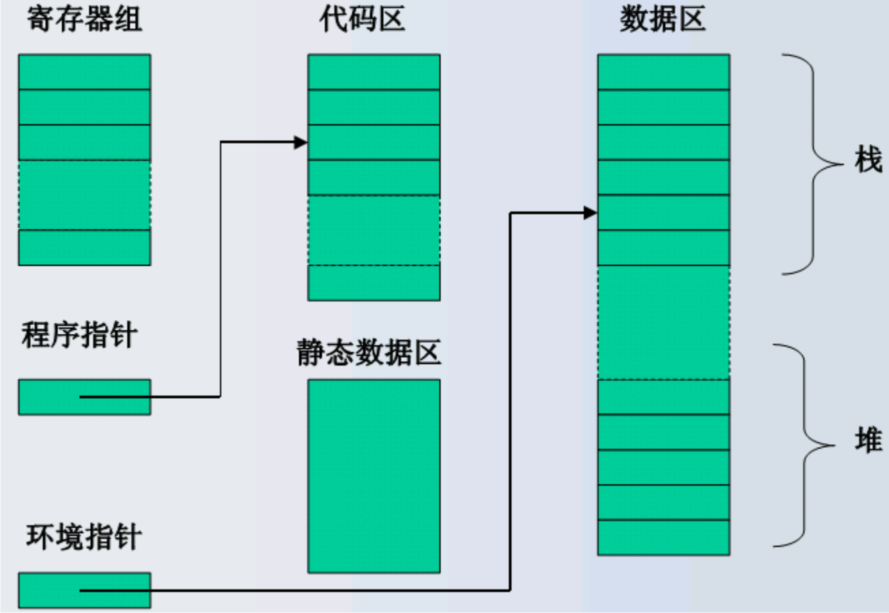


Service	System Call Code	Arguments	Result
print integer	1	$a0 = value	(none)
print float	2	$f12 = float value	(none)
print double	3	$f12 = double value	(none)
print string	4	$a0 = address of string	(none)
read integer	5	(none)	$v0 = value read
read float	6	(none)	$f0 = value read
read double	7	(none)	$f0 = value read
read string	8	$a0 = address where string to be stored $a1 = number of characters to read + 1	(none)
memory allocation	9	$a0 = number of bytes of storage desired	$v0 = address of block
exit (end of program)	10	(none)	(none)
print character	11	$a0 = integer	(none)
read character	12	(none)	char in $v0

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
    - 数据存储：reg， mem
    - Move，ALU，分支，I/O
    - 整数，*浮点*，伪指令（RV1图2-40）
  - 寻址方式：操作数，目标指令
- 程序
  - 段式内存分配：数据、代码、堆栈
  - 寄存器分配：寄存器使用约定
  - 过程调用/系统调用约定
    - » 堆栈，栈帧
- *可执行程序生成*





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

见$2.13.2，图2-25

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)

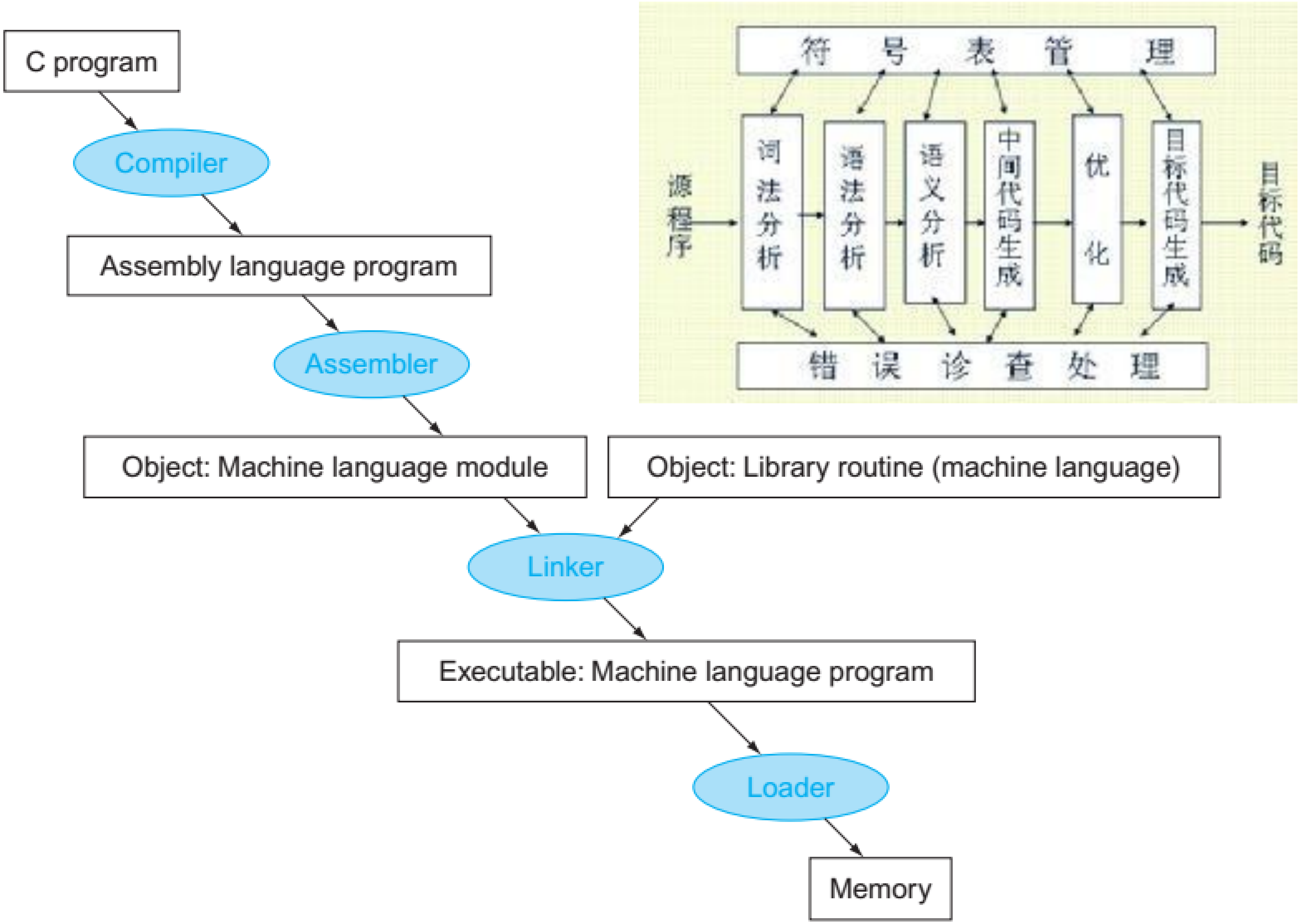
①

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

可执行程序格式、生成与执行

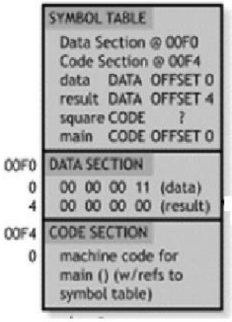
了解，自学

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
符号表管理
源程序
词法分析
语法分析
语义分析
中间代码生成
优化
目标代码生成
目标代码
错误诊查处理

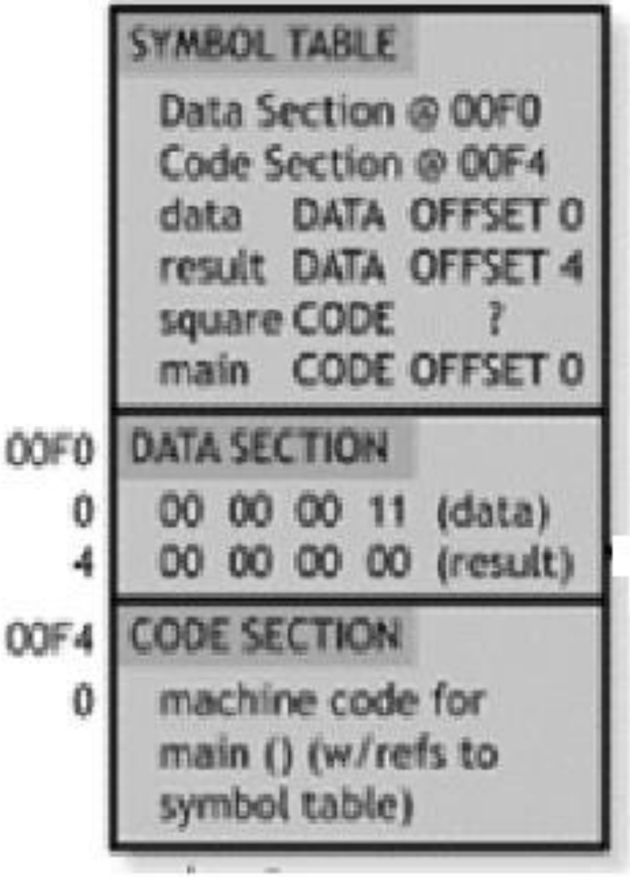
The Assembly Process：生成.obj

- 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



SYMBOL TABLE

Data Section @ 00F0
Code Section @ 00F4

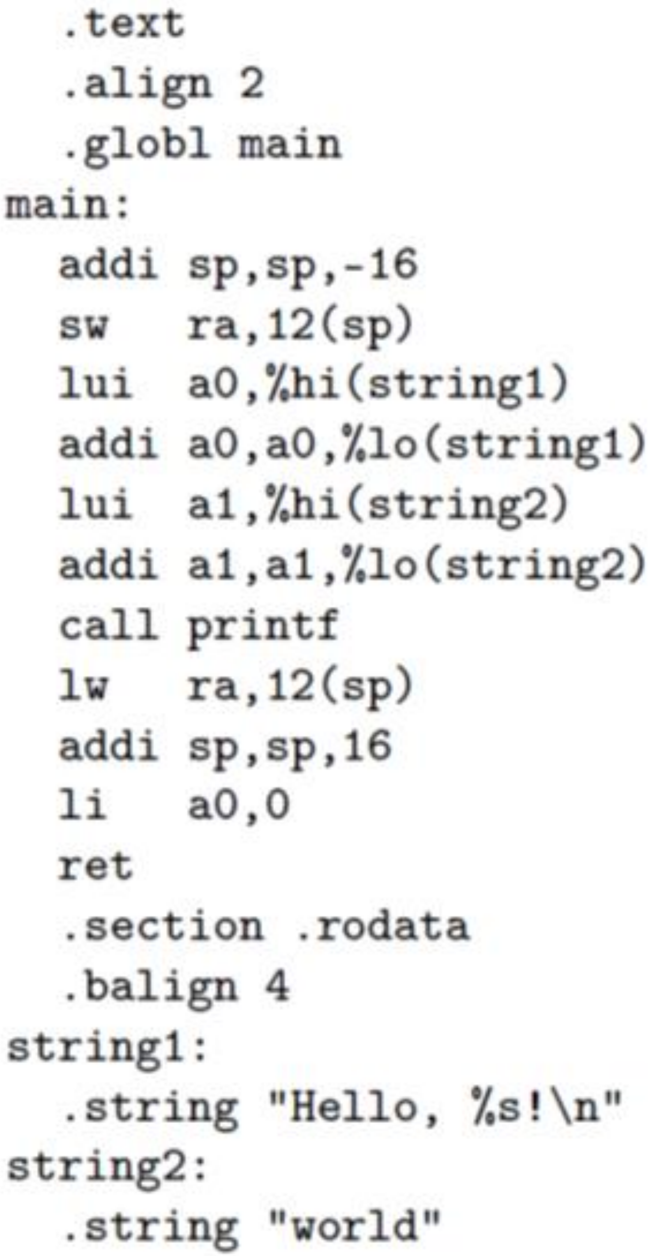
data	DATA	OFFSET 0
result	DATA	OFFSET 4
square	CODE	?
main	CODE	OFFSET 0

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)



```
    .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"
```

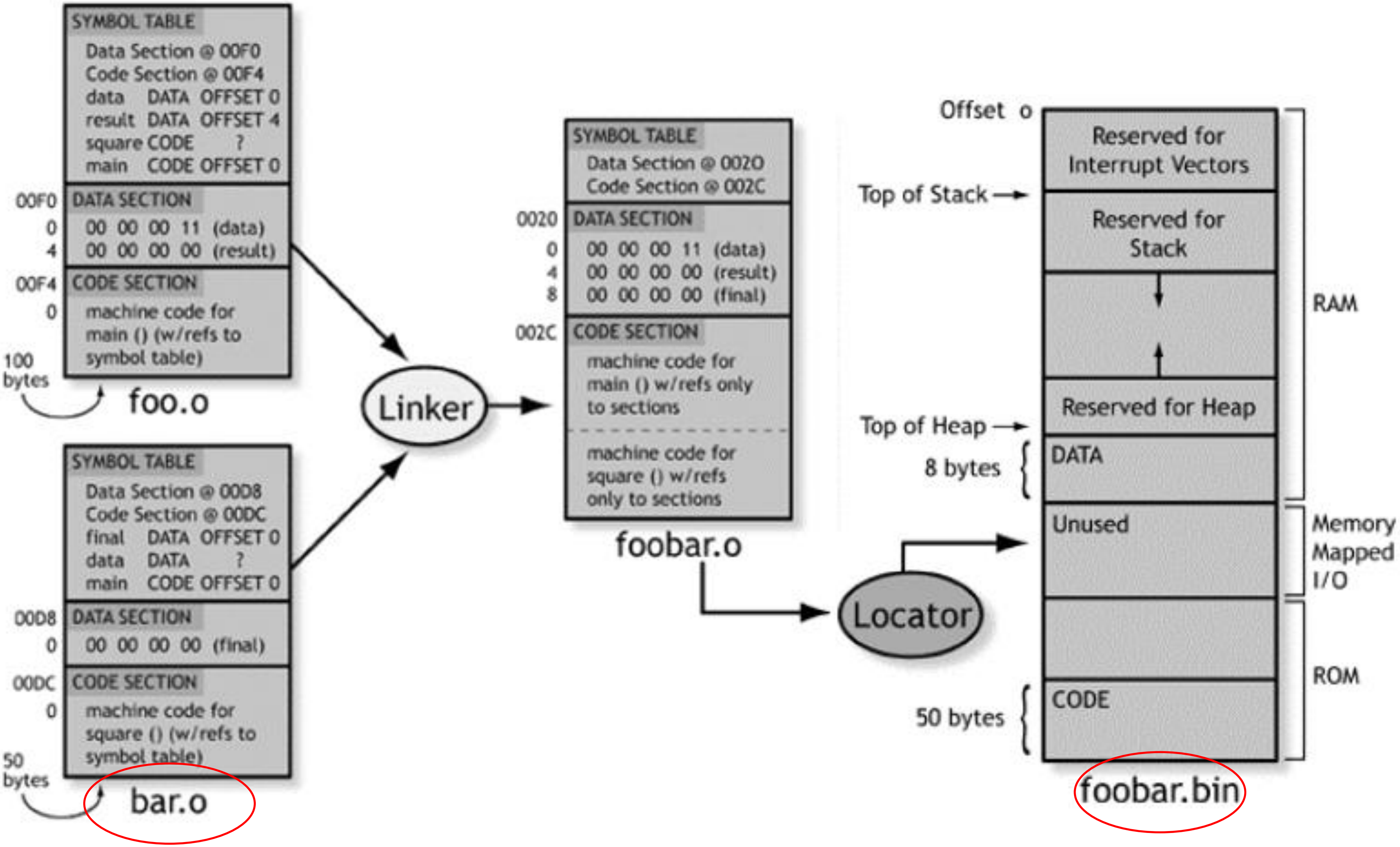
符号表：
全局定义和外部引用

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

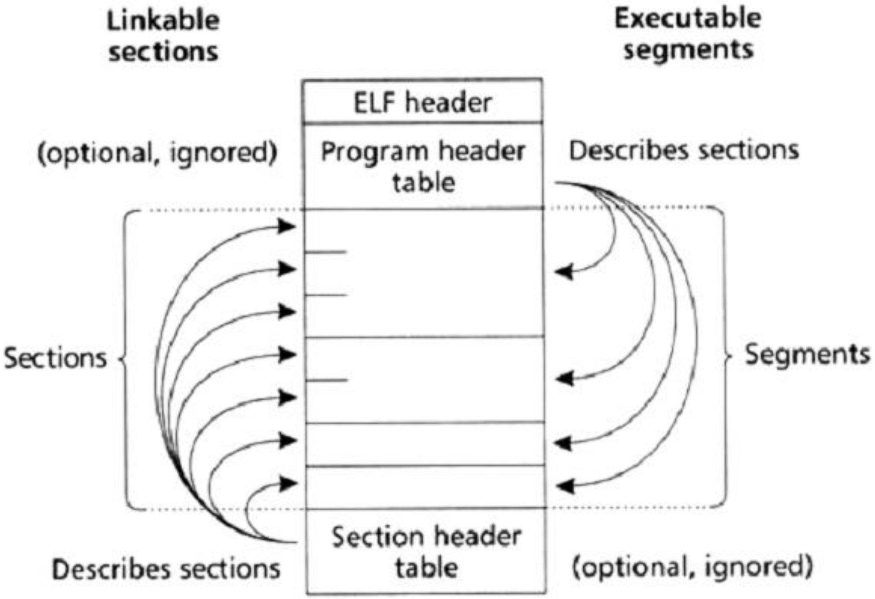
The Linker：合并各段

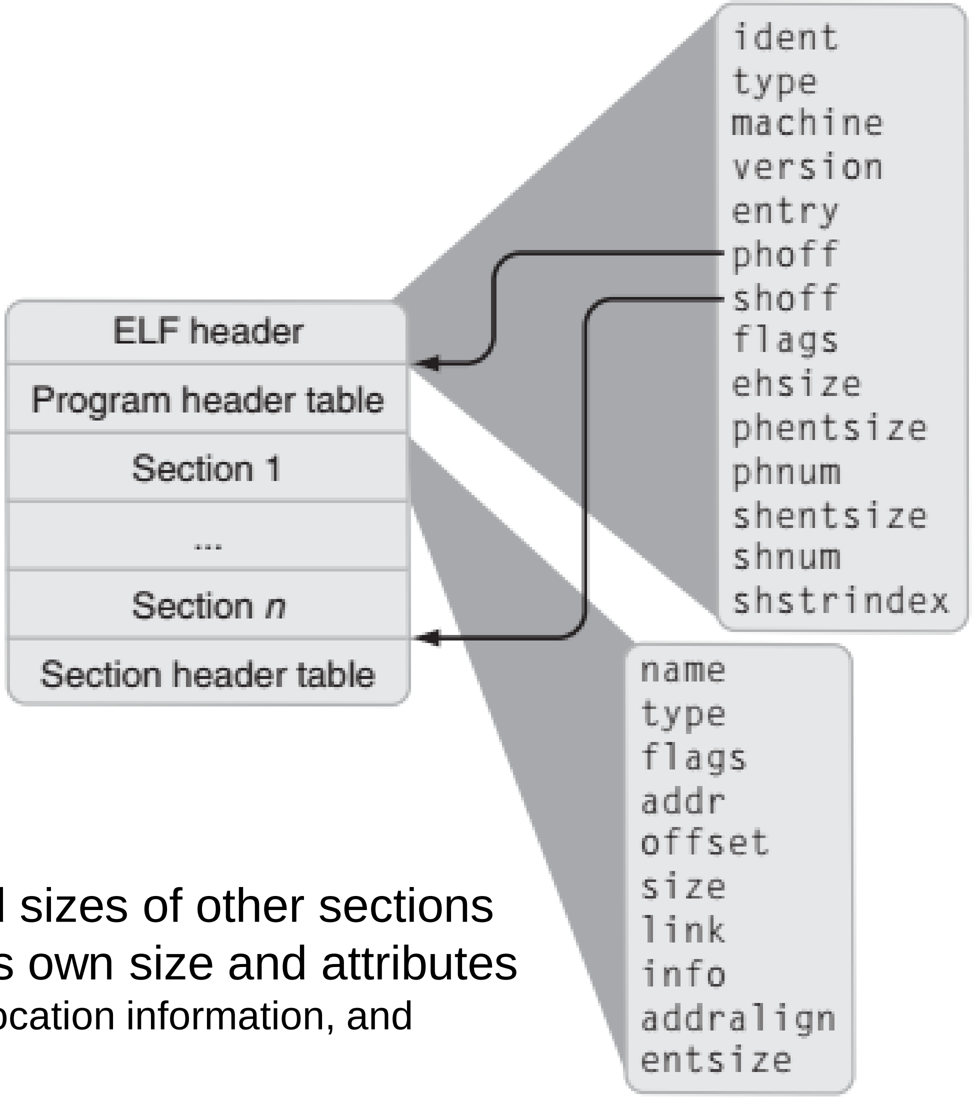
- Combines object files into object program（exe）
  - Constructs map of full program in memory using length information in each object file
    - Map determines addresses of all names
  - Instructions referring to external names are finalized with addresses determined by map
- Libraries：Subroutines
 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

Linking and Locating



ELF 格式目标文件结构





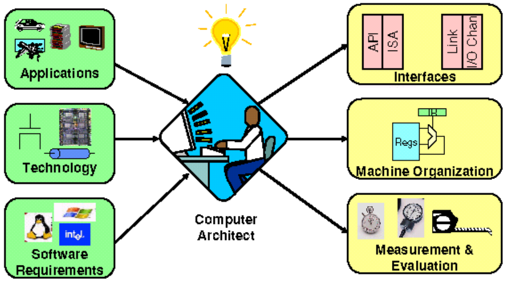
- the ELF specification
  - ELF header tell the offsets and sizes of other sections
  - Each section then describes its own size and attributes
    - symbol tables, string tables, relocation information, and dynamic linking information

Loading/Executing Object Programs

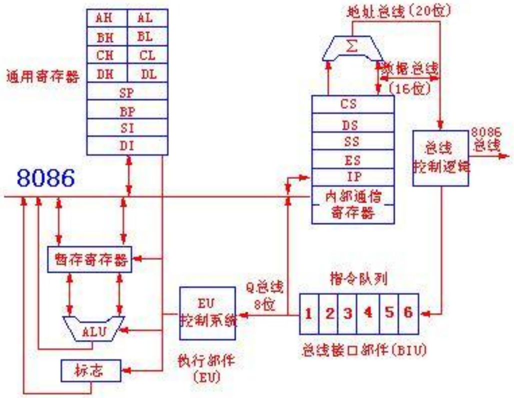
- 将映像文件从磁盘加载到内存
  - 读取文件头来确定各段大小
  - 创建虚拟地址空间
  - 将代码和初始化的数据复制到内存中
    - 或设置页表项来处理缺页
  - 在栈上建立参数
  - 初始化寄存器（包括sp、 fp、 gp）
  - 跳转到启动例程
    - 将参数复制到x10等等并调用main函数
    - 当main函数返回时，进行exit系统调用

ISA分类与进化

影响早期ISA设计的因素



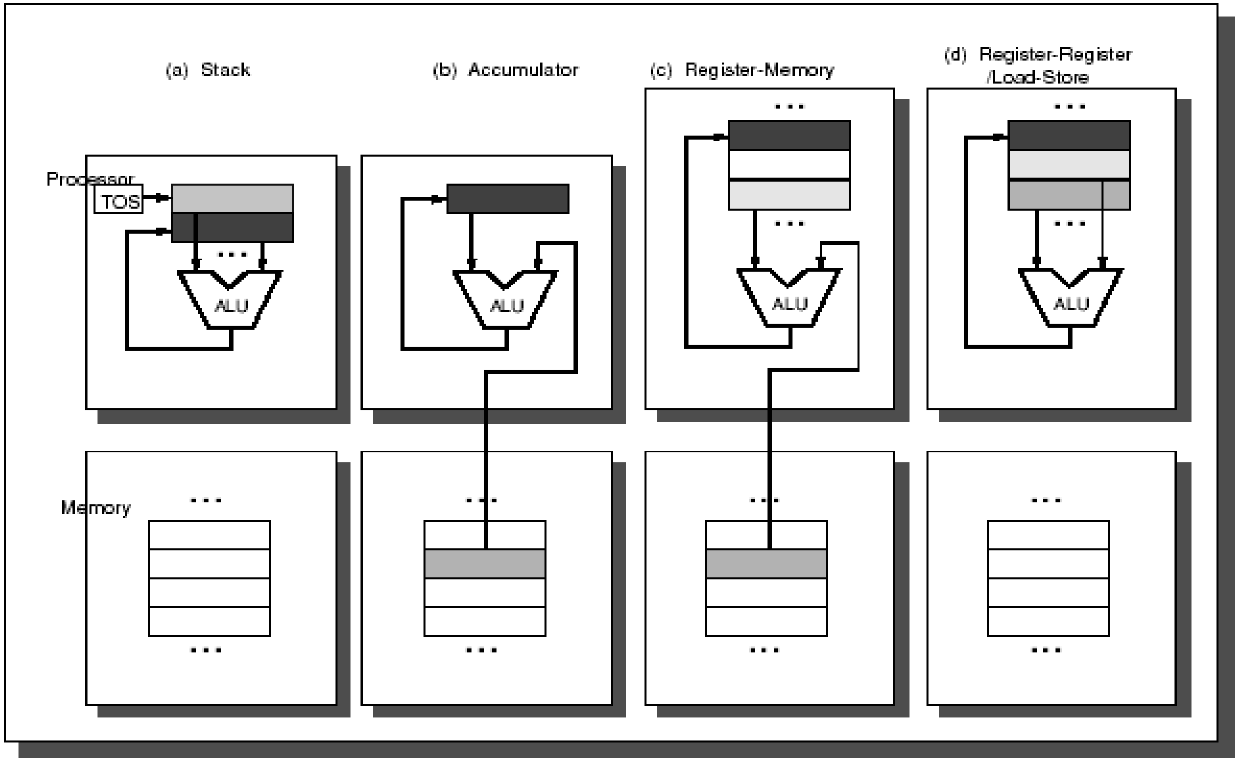
- 内存小而慢，能省则省
  - 某个完整系统只需几K字节
  - 指令长度不等、执行多个操作的指令
- 寄存器贵，少
  - 操作基于存储器
  - 多种寻址方式
- 编译技术尚未出现
  - 程序是以机器语言或汇编语言设计
  - 当时的看法是硬件比编译器更易设计
    - 为了便于编写程序，计算机架构师造出越来越复杂的指令，完成高级程序语言直接表达的功能
    - 进化中的痕迹：X86中的串操作指令



机器结构与ISA分类

- 指令格式和寻址方式越复杂，则越灵活高效
  - 性能：操作数的存放位置（访存是瓶颈）
  - 权衡：硬件设计复杂度、指令系统的兼容性
- 机器结构：processor designer view
  - stack
  - Accumulator
  - register-mem
  - register-register
- ISA分类：programmer/compiler view
  - CISC：以机器指令实现高级语言功能（reg-mem）
  - RISC：采用load/store体系，运算基于寄存器（reg-reg）
  - VLIW：兼容性差，硬件简单，低功耗

ISA Classes (processor designer view)

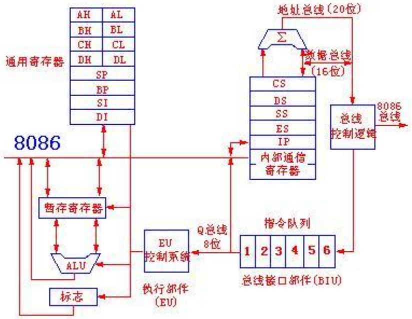


Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

H&P第6版：附录A，图A.1，四类ISA中操作数的位置

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！≤两百条指令
  - 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（SIMD），低功耗，兼容性差

CISC例，X86指令格式，图2-39

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

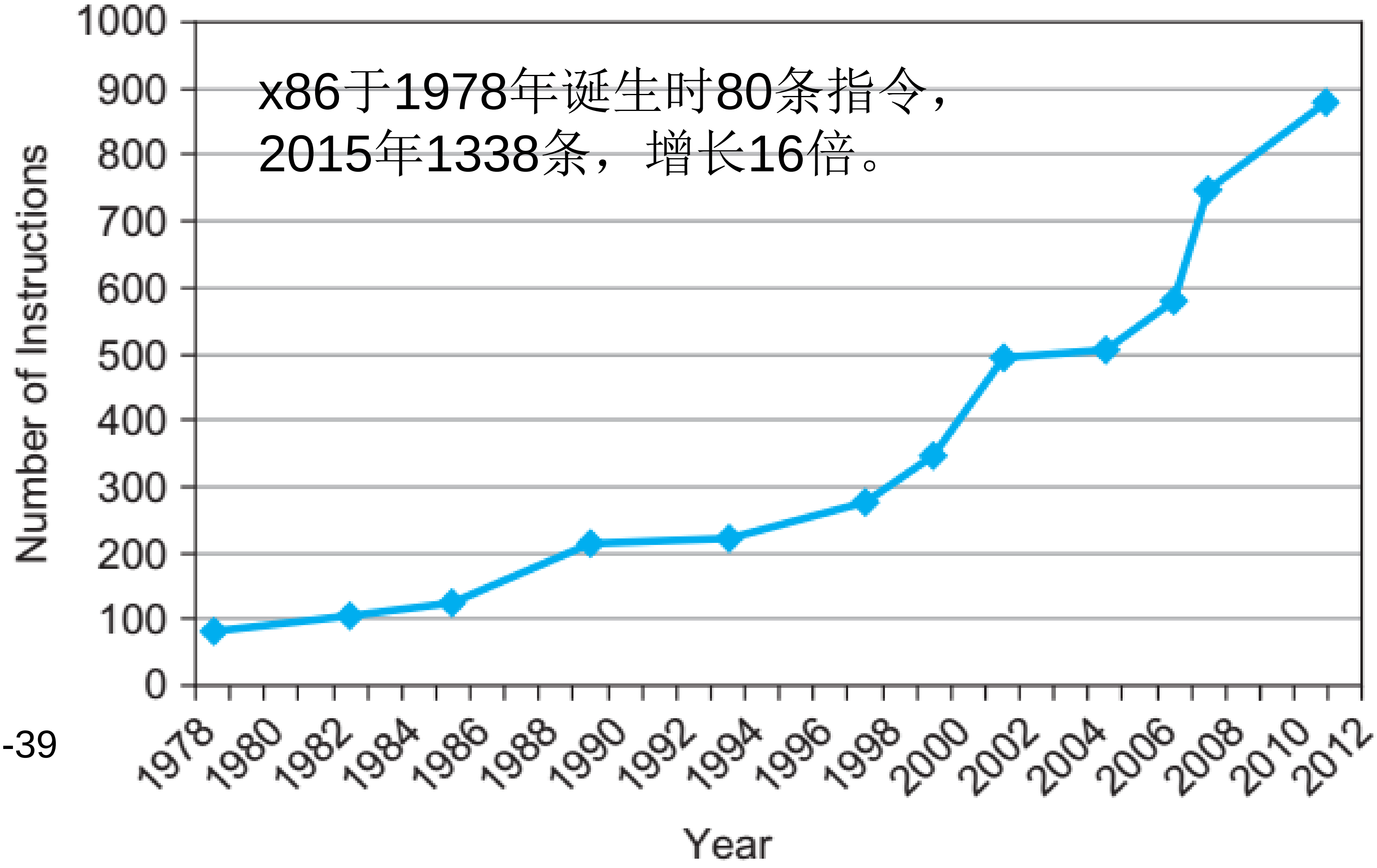
e. ADD EAX, #6765

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



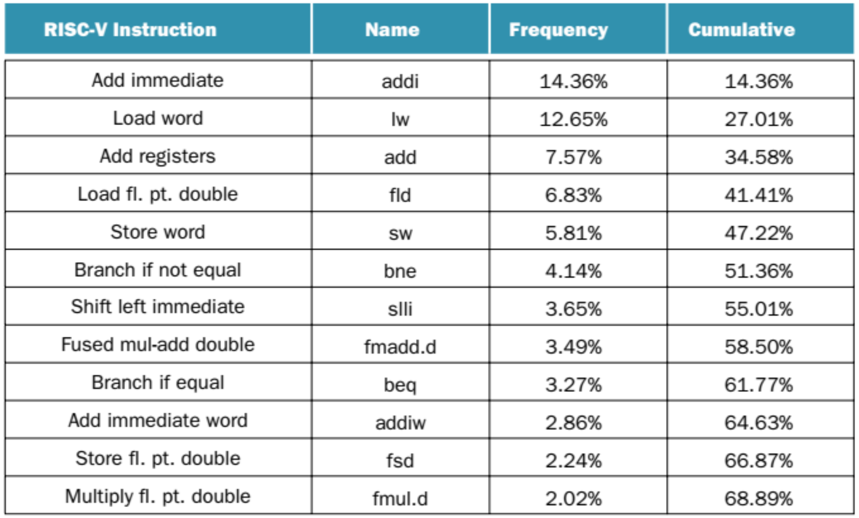
x86于1978年诞生时80条指令，
2015年1338条，增长16倍。

图2-39

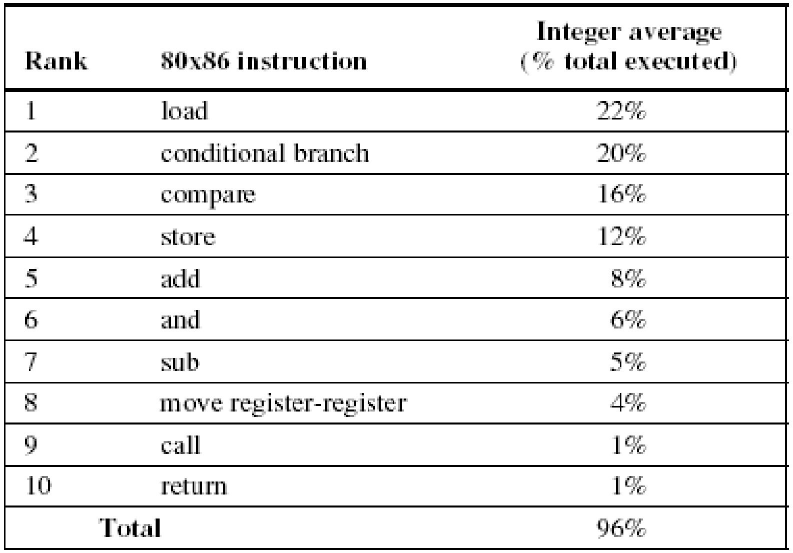
指令分布：RV，x86

- SPEC CPU2006

图3-22

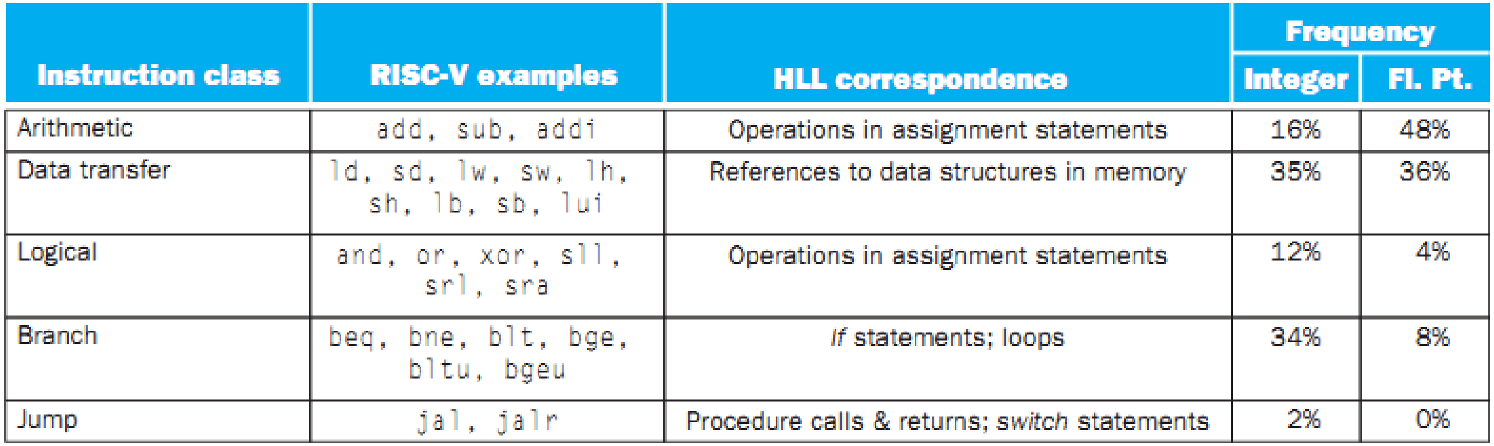


RISC-V Instruction	Name	Frequency	Cumulative
Add immediate	addi	14.36%	14.36%
Load word	lw	12.65%	27.01%
Add registers	add	7.57%	34.58%
Load fl. pt. double	fld	6.83%	41.41%
Store word	sw	5.81%	47.22%
Branch if not equal	bne	4.14%	51.36%
Shift left immediate	slli	3.65%	55.01%
Fused mul-add double	fmadd.d	3.49%	58.50%
Branch if equal	beq	3.27%	61.77%
Add immediate word	addiw	2.86%	64.63%
Store fl. pt. double	fsd	2.24%	66.87%
Multiply fl. pt. double	fmul.d	2.02%	68.89%



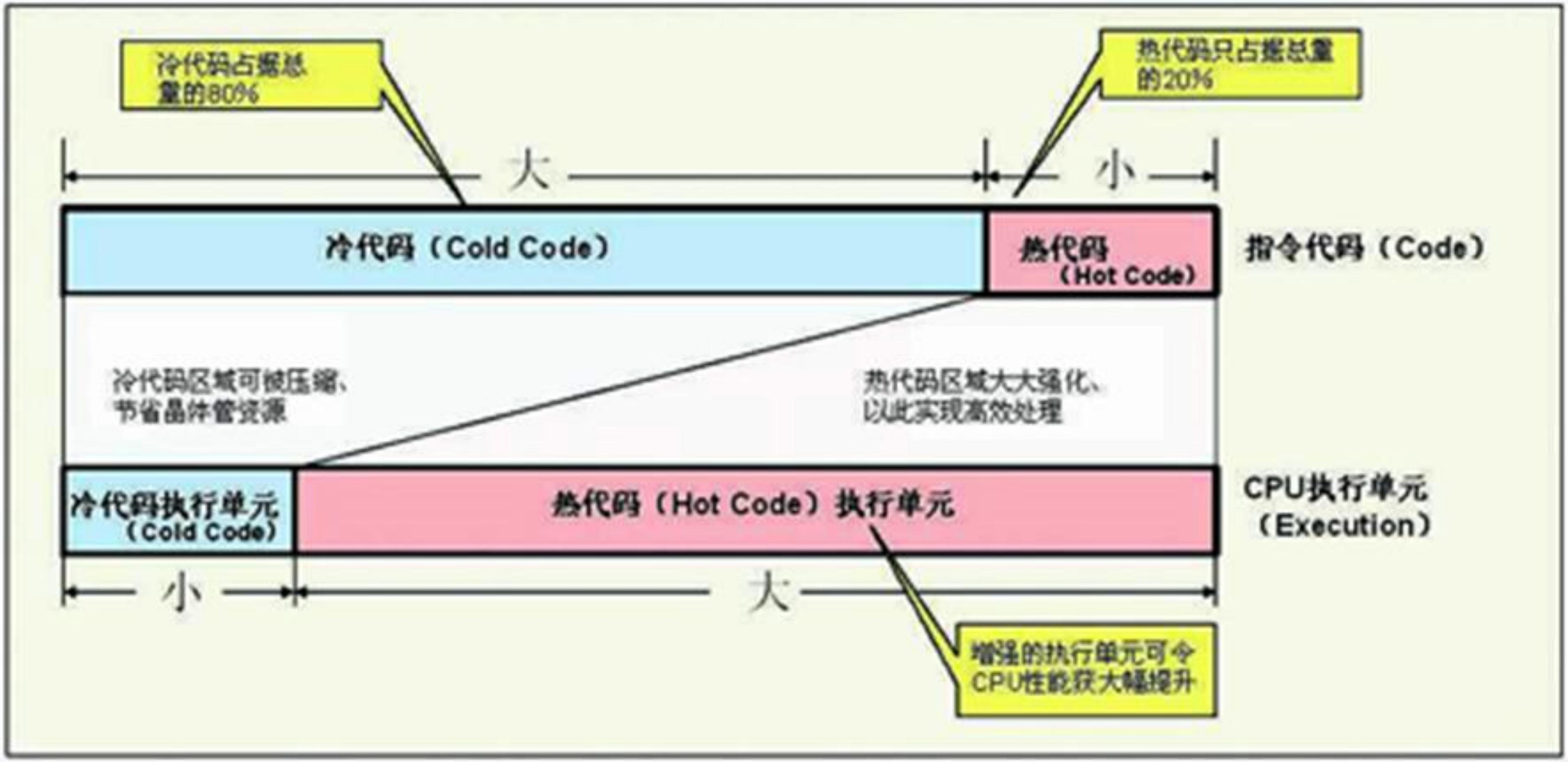
Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

图2-48



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：二八定律



- 1975年IBM John Cocke（1987图灵奖）提出“二八定律”，应精简指令数量，一条复杂指令可用多条简单指令代替，且应放弃微程序技术。
  - 1980年完成了第一个采用RISC 架构【llxx：应称“精简指令集”架构，RISC是UCB提出的 】的计算机原型IBM ® 801。
- UCB David Patterson（2017图灵奖）和Carlo H. Sequin 1980年总结Cocke思想，提出“RISC”一词
  - 1982完成RISC-I处理器

CISC machine vs RISC machine

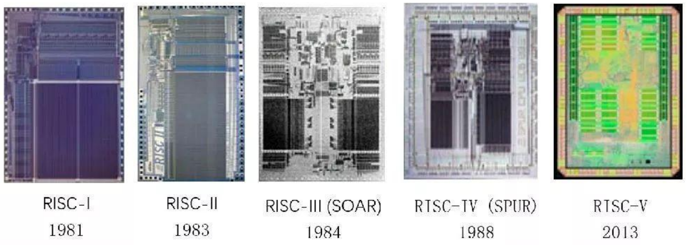
- Instructions are of variable format.
- There are multiple instructions and addressing modes.
- Complex instructions take many different cycles.
- Any instruction can reference memory.
- There is a single set of registers.
- No instructions are pipelined.
- A microprogram is executed for each native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

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

- Microprocessor w/o Interlocked Piped Stages
  - interlock单元：数据依赖问题【两个动作不许同时执行】
    - 互锁：检测，推迟后续指令执行（互锁状态）
    - 无互锁：尽量利用软件办法避免
    - 无互锁困难，低效：R4000以后使用interlock
- Patterson总结Cocke思想，提出RISC指令集，1980
  - 4个Design Principles
    - Simplicity favors regularity
    - Smaller is faster
    - *Make the common case fast*
    - Good design demands good compromises
  - 1981/1982完成RISC-I处理器
- Hennessy完成MIPS，1983？
- 图灵奖： Cocke 1987， Hennessy+Patterson 2017



RISC-I 1981

RISC-II 1983

RISC-III (SOAR) 1984

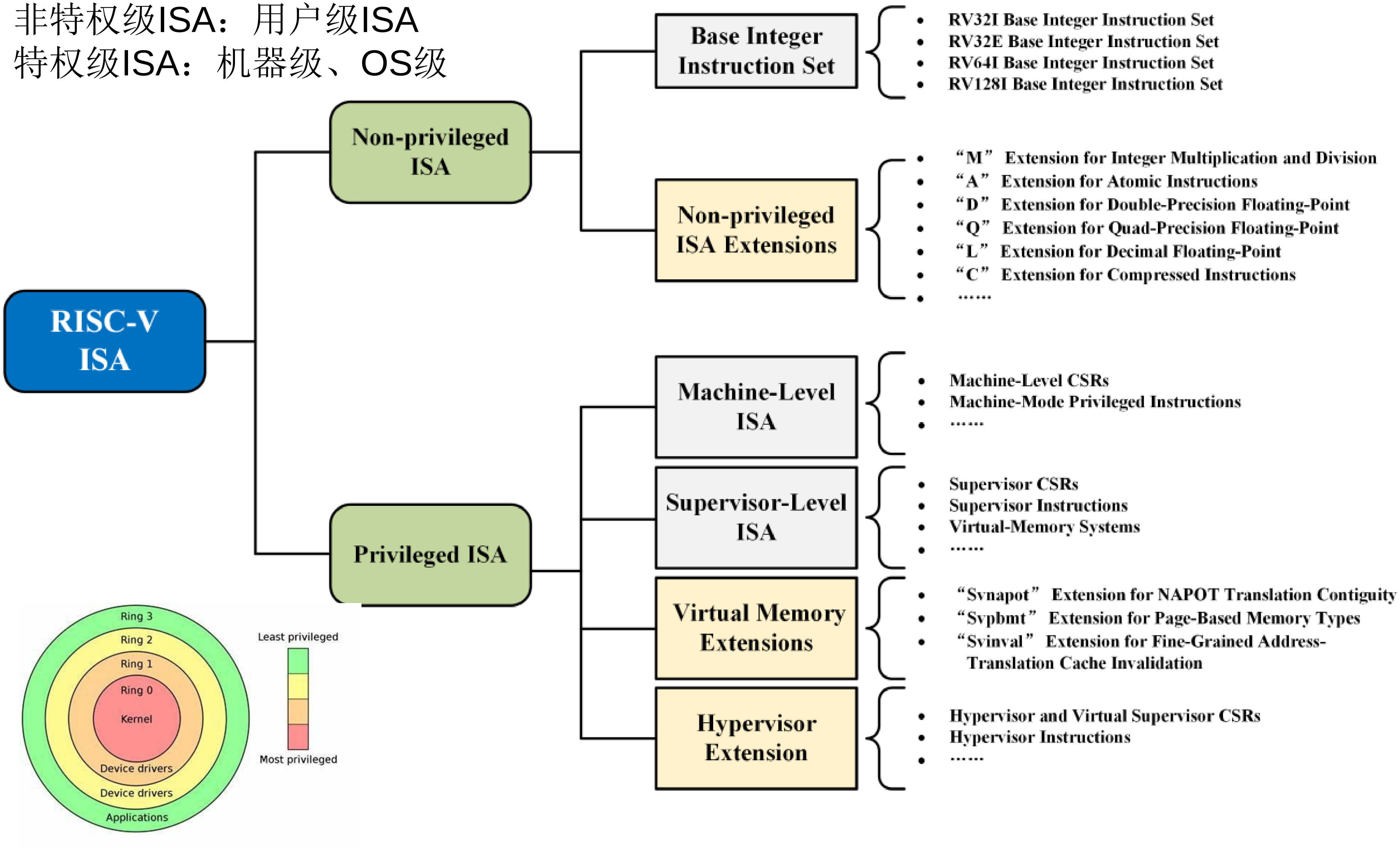
RISC-IV (SPUR) 1988

RISC-V 2013

The RISC-V ISA

非特权级ISA：用户级ISA

特权级ISA：机器级、OS级



RISC-V ISA

Non-privileged ISA

Base Integer Instruction Set

- RV32I Base Integer Instruction Set
- RV32E Base Integer Instruction Set
- RV64I Base Integer Instruction Set
- RV128I Base Integer Instruction Set

Non-privileged ISA Extensions

- “M” Extension for Integer Multiplication and Division
- “A” Extension for Atomic Instructions
- “D” Extension for Double-Precision Floating-Point
- “Q” Extension for Quad-Precision Floating-Point
- “L” Extension for Decimal Floating-Point
- “C” Extension for Compressed Instructions
- ……

Privileged ISA

Machine-Level ISA

- Machine-Level CSRs
- Machine-Mode Privileged Instructions
- ……

Supervisor-Level ISA

- Supervisor CSRs
- Supervisor Instructions
- Virtual-Memory Systems
- ……

Virtual Memory Extensions

- “Svnapot” Extension for NAPOT Translation Contiguity
- “Svpbmt” Extension for Page-Based Memory Types
- “Svinval” Extension for Fine-Grained Address-Translation Cache Invalidation

Hypervisor Extension

- Hypervisor and Virtual Supervisor CSRs
- Hypervisor Instructions
- ……

RISC-V ISA的特点，最少几条指令？

- RV32I+系统指令：可运行基本Linux
  - RV32I：图2-18（34）+ 图2-41（5）
    - 计算、控制、访存（load/store，lr/sc）
  - 系统指令：图5-47（13）
    - 同步+CSR+异常：fence、ecall
  - 《The RISC-V Instruction Set Manual》
- 指令集模块化：图2-42
  - RV32IMFD，RV32E
  - RV32A：amo【“指令参考表”】
- 约束：简洁，架构与实现分离
  - 成本：芯片面积
  - 性能：时间，功耗
  - 扩展性：操作码域空间
  - 程序大小
  - 易于编程/编译/链接

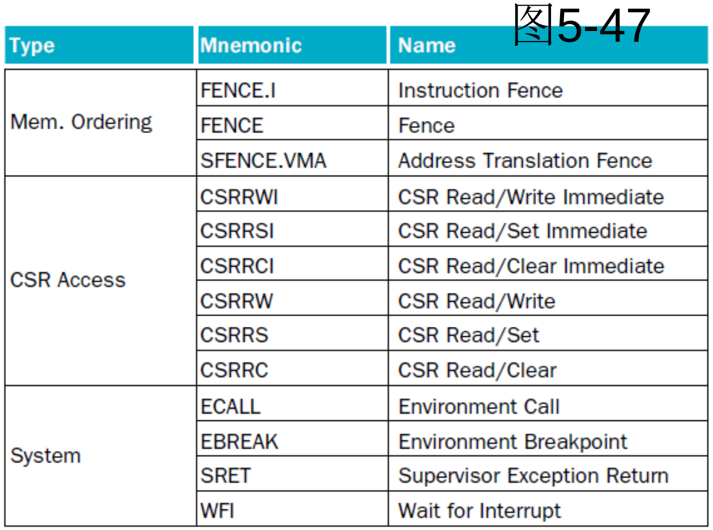


图5-47

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

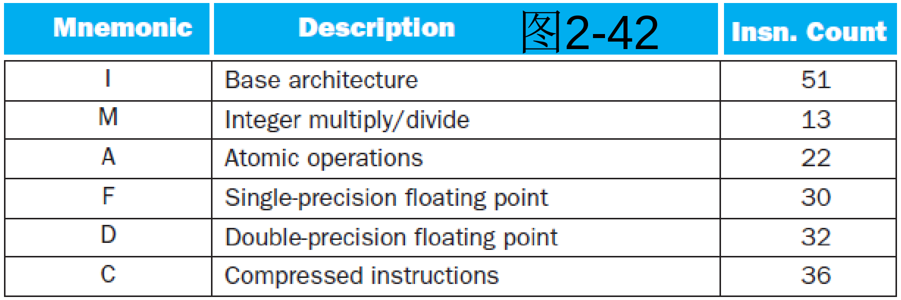
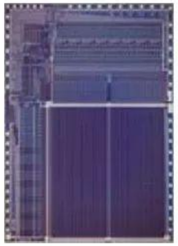


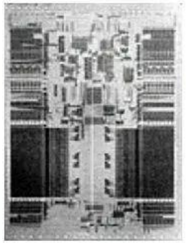
图2-42

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

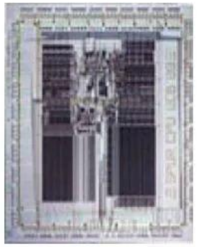


RISC-I
1981

RISC-II
1983



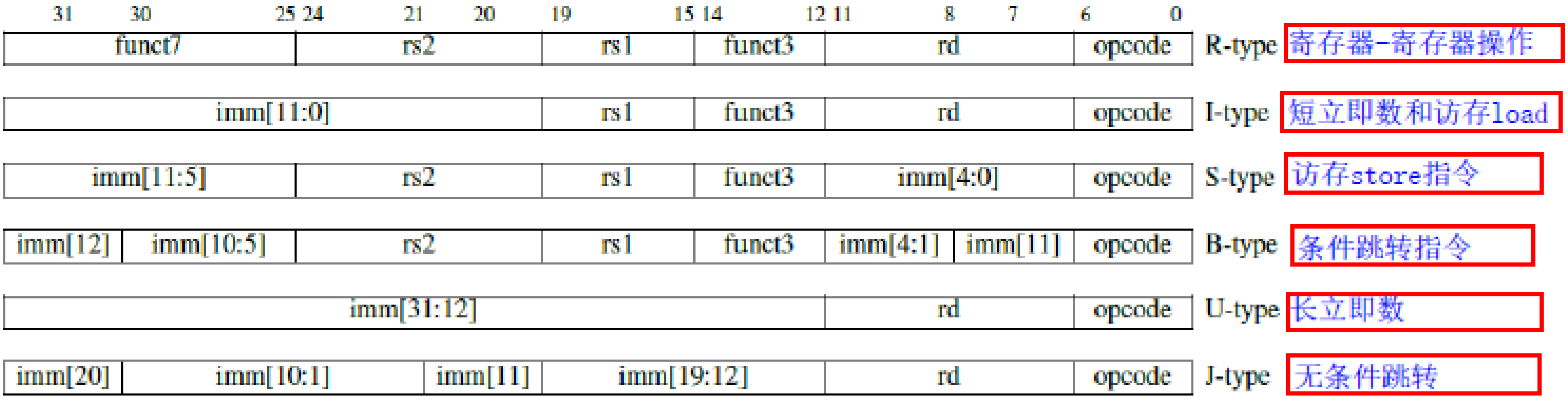
RISC-III (SOAR)
1984



RISC-IV（SPUR)
1988

RISC-V
2013

RV vs. MIPS：指令格式



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 *reg-reg*	op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)
	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)				

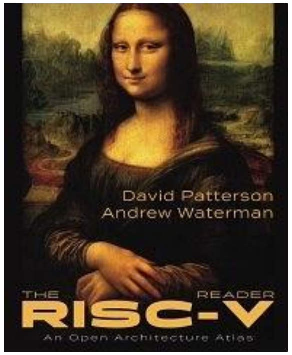
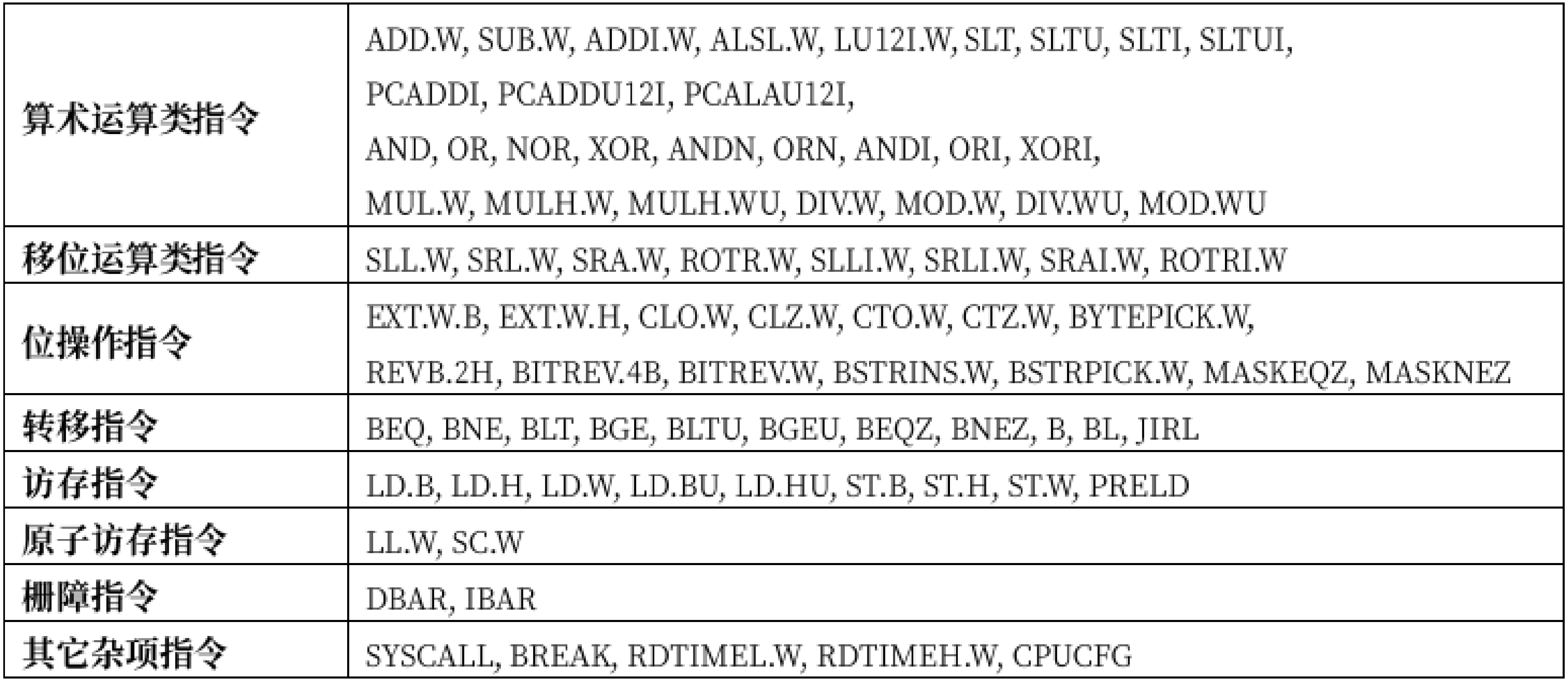


图2.29

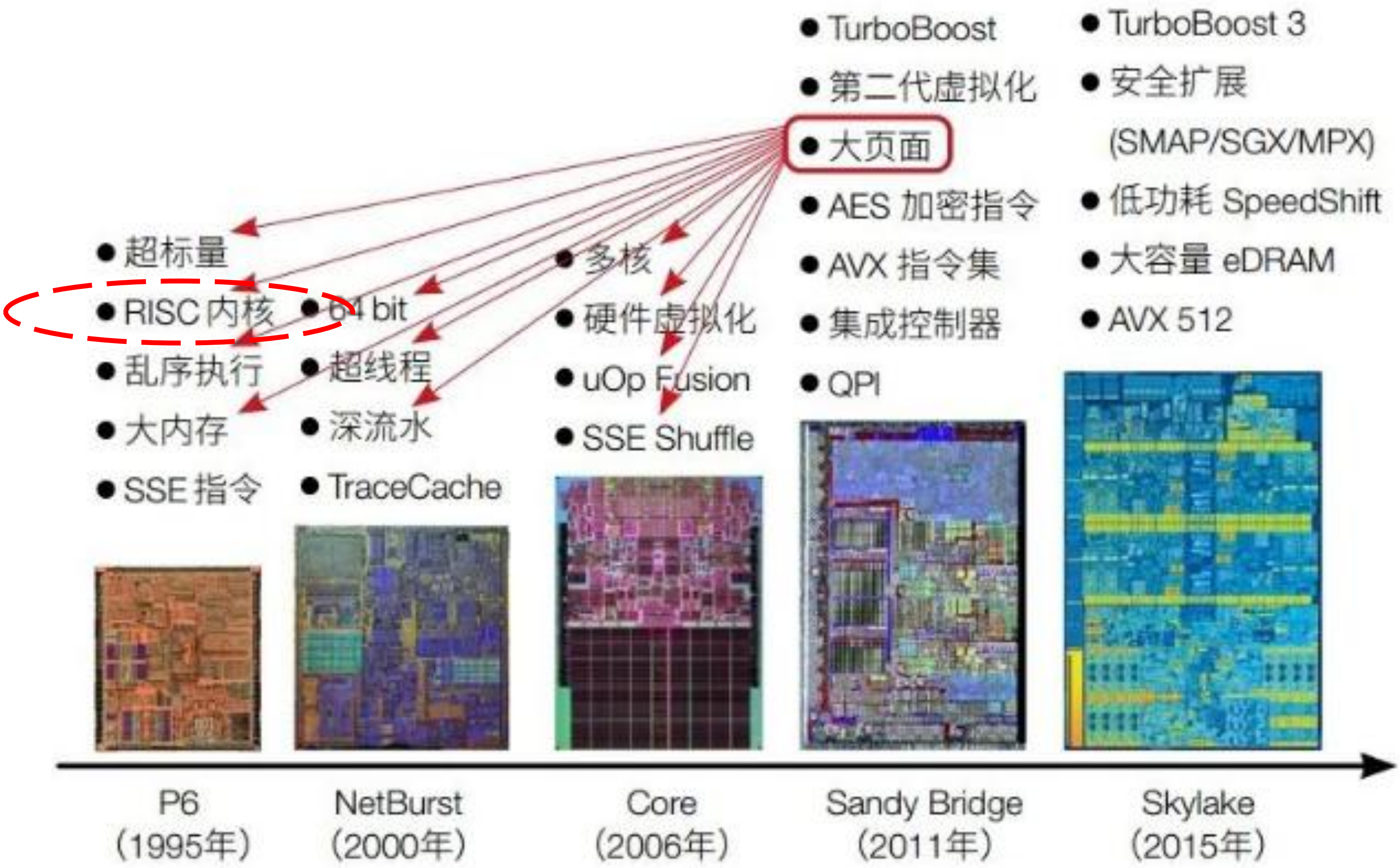
差别：1）操作码位数与位置；2）立即数位数与位置；2）rs/rd位置。

LA32基础整数指令



算术运算类指令	ADD.W, SUB.W, ADDI.W, ALSL.W, LU12I.W, SLT, SLTU, SLTI, SLTUI, PCADDI, PCADDU12I, PCALAU12I, AND, OR, NOR, XOR, ANDN, ORN, ANDI, ORI, XORI, MUL.W, MULH.W, MULH.WU, DIV.W, MOD.W, DIV.WU, MOD.WU
移位运算类指令	SLL.W, SRL.W, SRA.W, ROTR.W, SLLI.W, SRLI.W, SRAI.W, ROTRI.W
位操作指令	EXT.W.B, EXT.W.H, CLO.W, CLZ.W, CTO.W, CTZ.W, BYTEPICK.W, REVB.2H, BITREV.4B, BITREV.W, BSTRINS.W, BSTRPICK.W, MASKEQZ, MASKNEZ
转移指令	BEQ, BNE, BLT, BGE, BLTU, BGEU, BEQZ, BNEZ, B, BL, JIRL
访存指令	LD.B, LD.H, LD.W, LD.BU, LD.HU, ST.B, ST.H, ST.W, PRELD
原子访存指令	LL.W, SC.W
栅障指令	DBAR, IBAR
其它杂项指令	SYSCALL, BREAK, RDTIMEL.W, RDTIMEH.W, CPUCFG

Intel处理器架构演化（1995—2015 年）



- 超标量
- RISC 内核
- 乱序执行
- 大内存
- SSE 指令

P6
（1995年）

- 64 bit
- 超线程
- 深流水
- TraceCache

NetBurst
（2000年）

- 多核
- 硬件虚拟化
- uOp Fusion
- SSE Shuffle

Core
（2006年）

- TurboBoost
- 第二代虚拟化
- 大页面
- AES 加密指令
- AVX 指令集
- 集成控制器
- QPI

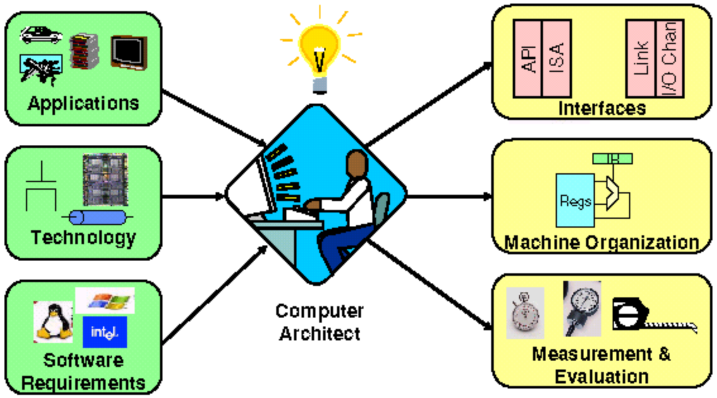
Sandy Bridge
（2011年）

- TurboBoost 3
- 安全扩展 (SMAP/SGX/MPX)
- 低功耗 SpeedShift
- 大容量 eDRAM
- AVX 512

Skylake
（2015年）

ISA: A Minimalist Perspective

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



- “最小”处理器？——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？执行过程？
  - C：CPI，IPC

OISC：the one instruction set computer

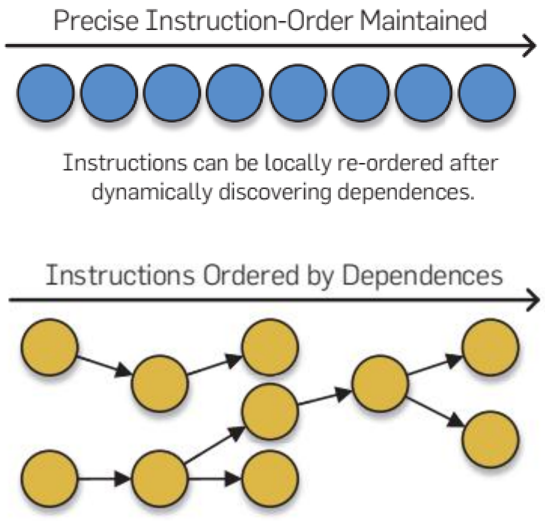
- OISC：the ultimate reduced instruction set computer
  - 一条SBN指令：substract and branch if negative
    - subleq *a*, *b*, *c*；Mem[*b*] = Mem[*b*] - Mem[*a*]，
      if (Mem[*b*] ≤ 0) goto *c*
- 应用：嵌入式处理器
  - 硬件极其简单
    - 程序员有充分的控制权
    - 优化由编译器完成
  - 灵活
    - 其他“指令”都可由该指令构造
    - 意味着用户可自定义指令集
    - 意味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

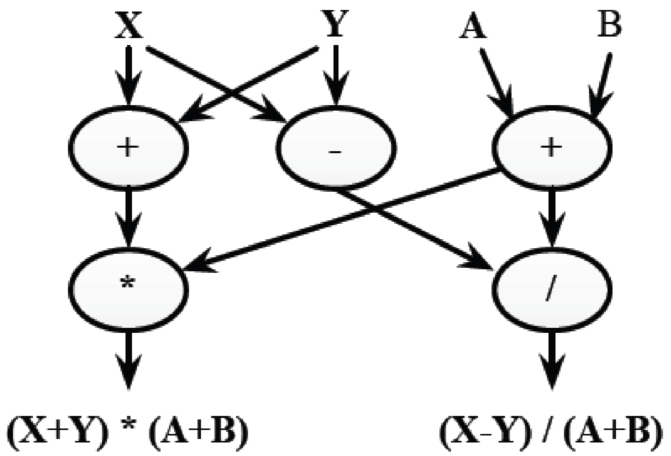
COMPUTER ARCHITECTURE:
A Minimalist Perspective

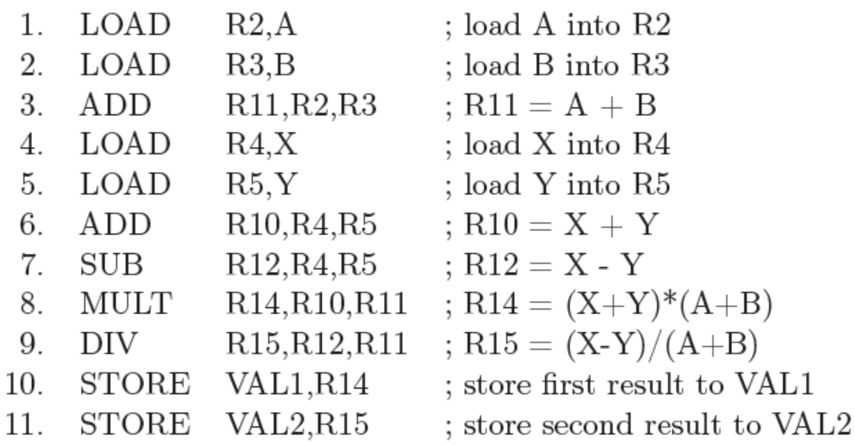
William F. Gilreath
Phillip A. Laplante

Springer

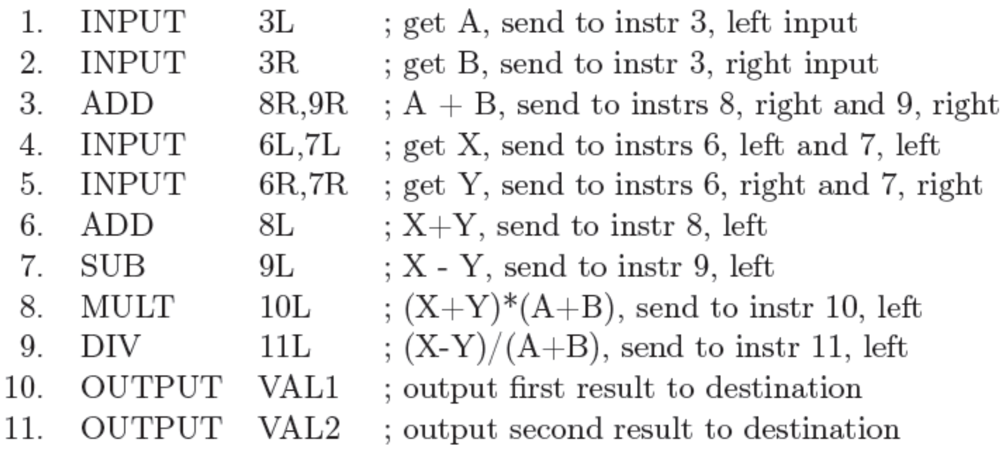
Von Neumann vs. dataflow


Precise Instruction-Order Maintained
Instructions can be locally re-ordered after dynamically discovering dependences.
Instructions Ordered by Dependences



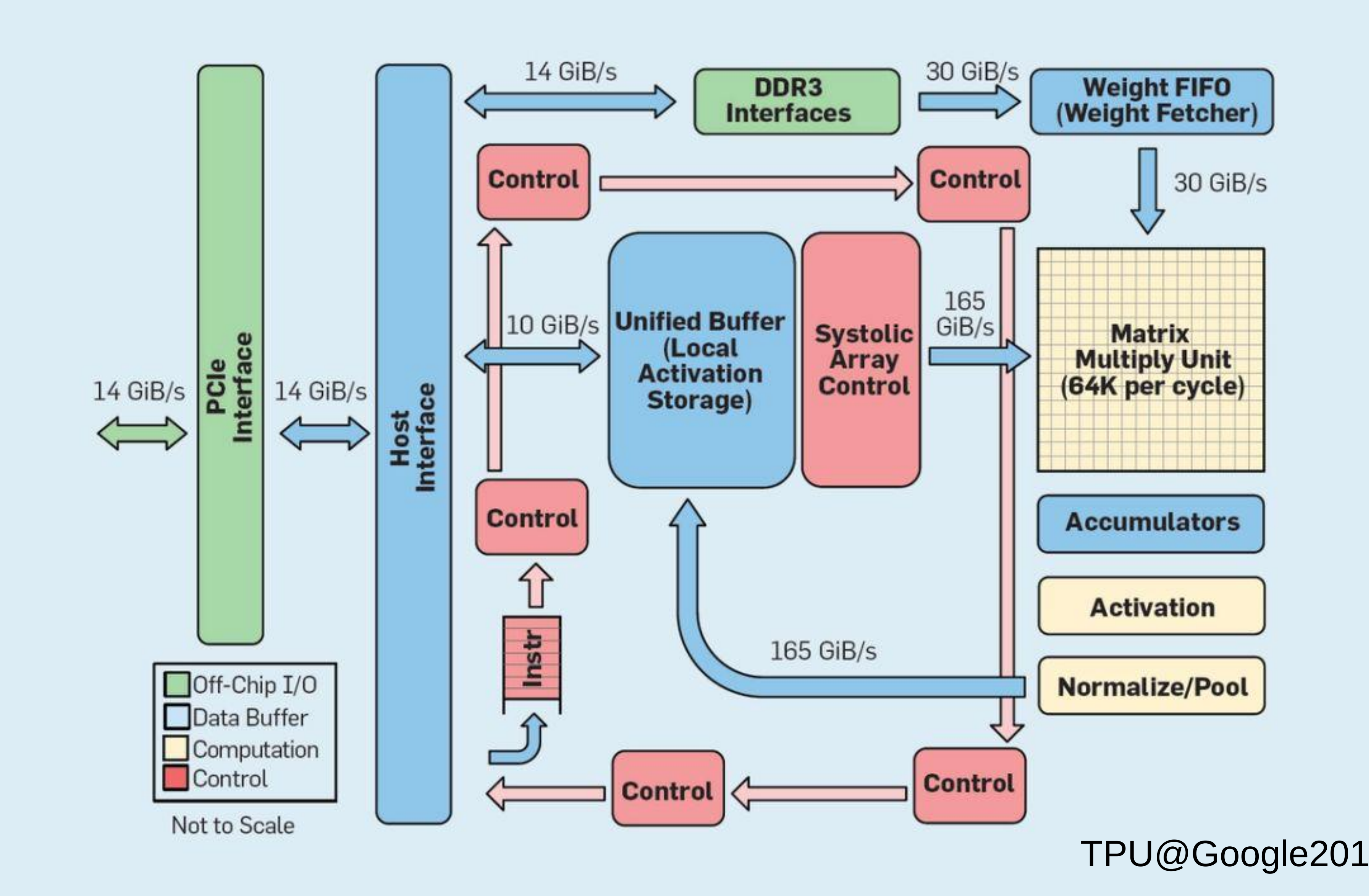


```
 1.  LOAD    R2,A          ; load A into R2
 2.  LOAD    R3,B          ; load B into R3
 3.  ADD     R11,R2,R3     ; R11 = A + B
 4.  LOAD    R4,X          ; load X into R4
 5.  LOAD    R5,Y          ; load Y into R5
 6.  ADD     R10,R4,R5     ; R10 = X + Y
 7.  SUB     R12,R4,R5     ; R12 = X - Y
 8.  MULT    R14,R10,R11   ; R14 = (X+Y)*(A+B)
 9.  DIV     R15,R12,R11   ; R15 = (X-Y)/(A+B)
10.  STORE   VAL1,R14      ; store first result to VAL1
11.  STORE   VAL2,R15      ; store second result to VAL2
```



```
 1.  INPUT   3L      ; get A, send to instr 3, left input
 2.  INPUT   3R      ; get B, send to instr 3, right input
 3.  ADD     8R,9R   ; A + B, send to instrs 8, right and 9, right
 4.  INPUT   6L,7L   ; get X, send to instrs 6, left and 7, left
 5.  INPUT   6R,7R   ; get Y, send to instrs 6, right and 7, right
 6.  ADD     8L      ; X+Y, send to instr 8, left
 7.  SUB     9L      ; X - Y, send to instr 9, left
 8.  MULT    10L     ; (X+Y)*(A+B), send to instr 10, left
 9.  DIV     11L     ; (X-Y)/(A+B), send to instr 11, left
10.  OUTPUT  VAL1    ; output first result to destination
11.  OUTPUT  VAL2    ; output second result to destination
```

计算机架构的未来：DSA，寒武纪、英伟达等



TPU@Google2015

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
  - main()与swap()状态保存异同？
  - 过程调用时程序的内存数据是否需要保存？
  - 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。CPI?