计算机组成原理
第二章“指令系统”

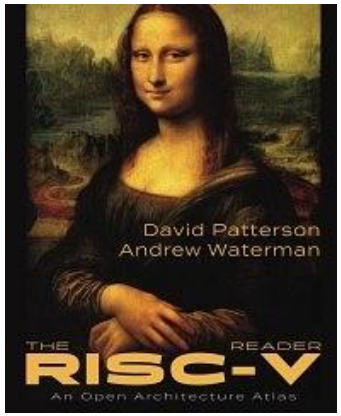
中科大11系

李曦

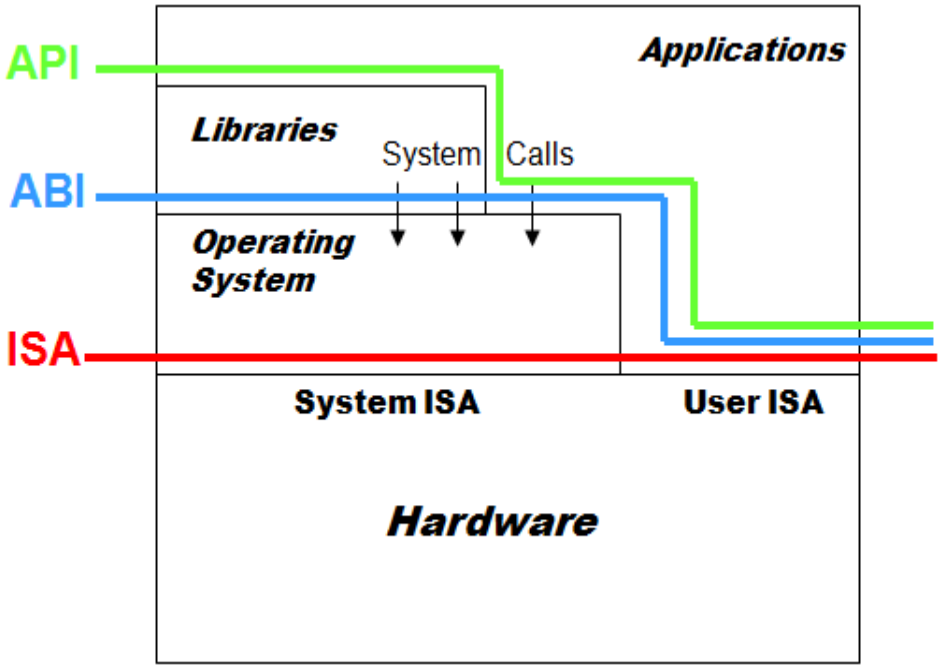
概要

- 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*

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



Architecture（ISA）& Interfaces



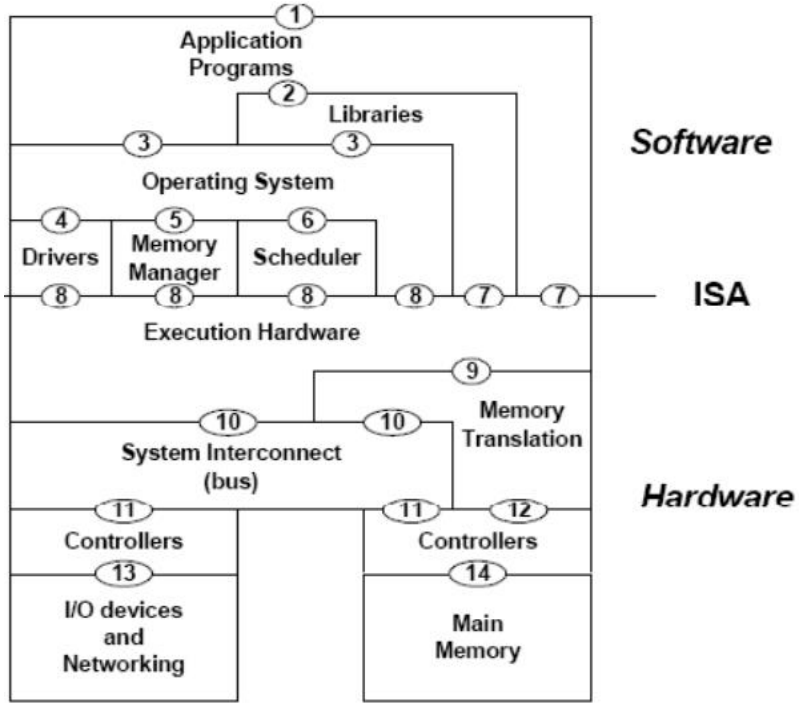
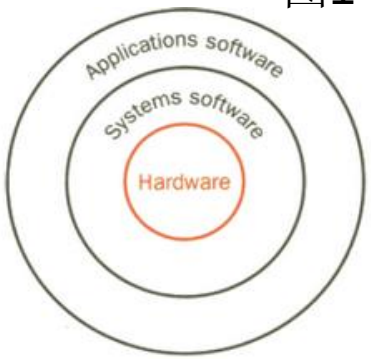


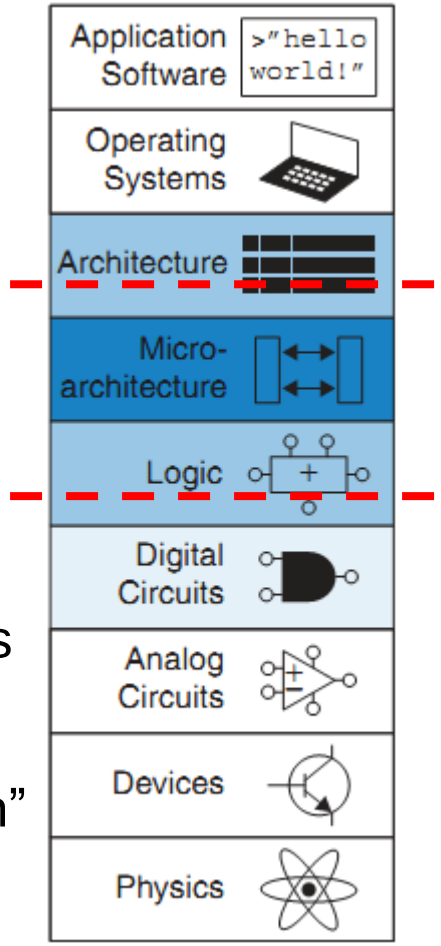
图1-3

- **API** – application programming interface
- **ABI** – application binary interface——HLL<->LLL
- **ISA** – instruction set architecture（p14）
  - Architecture: formal specification of a system's interface and the logical behavior of its visible resources.



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

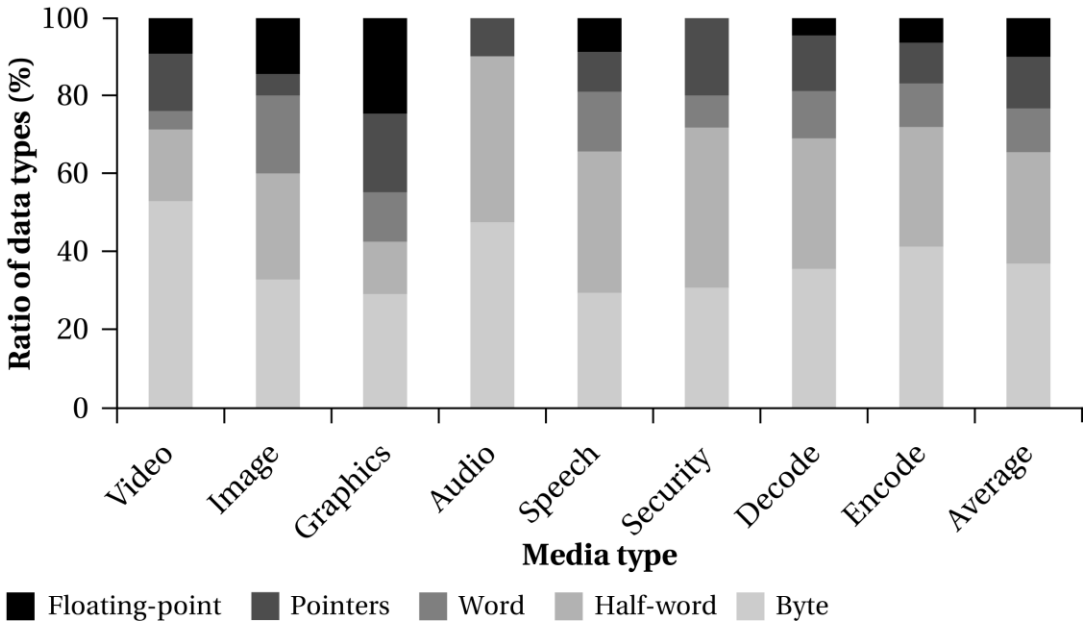


体系结构的8种属性

- 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*

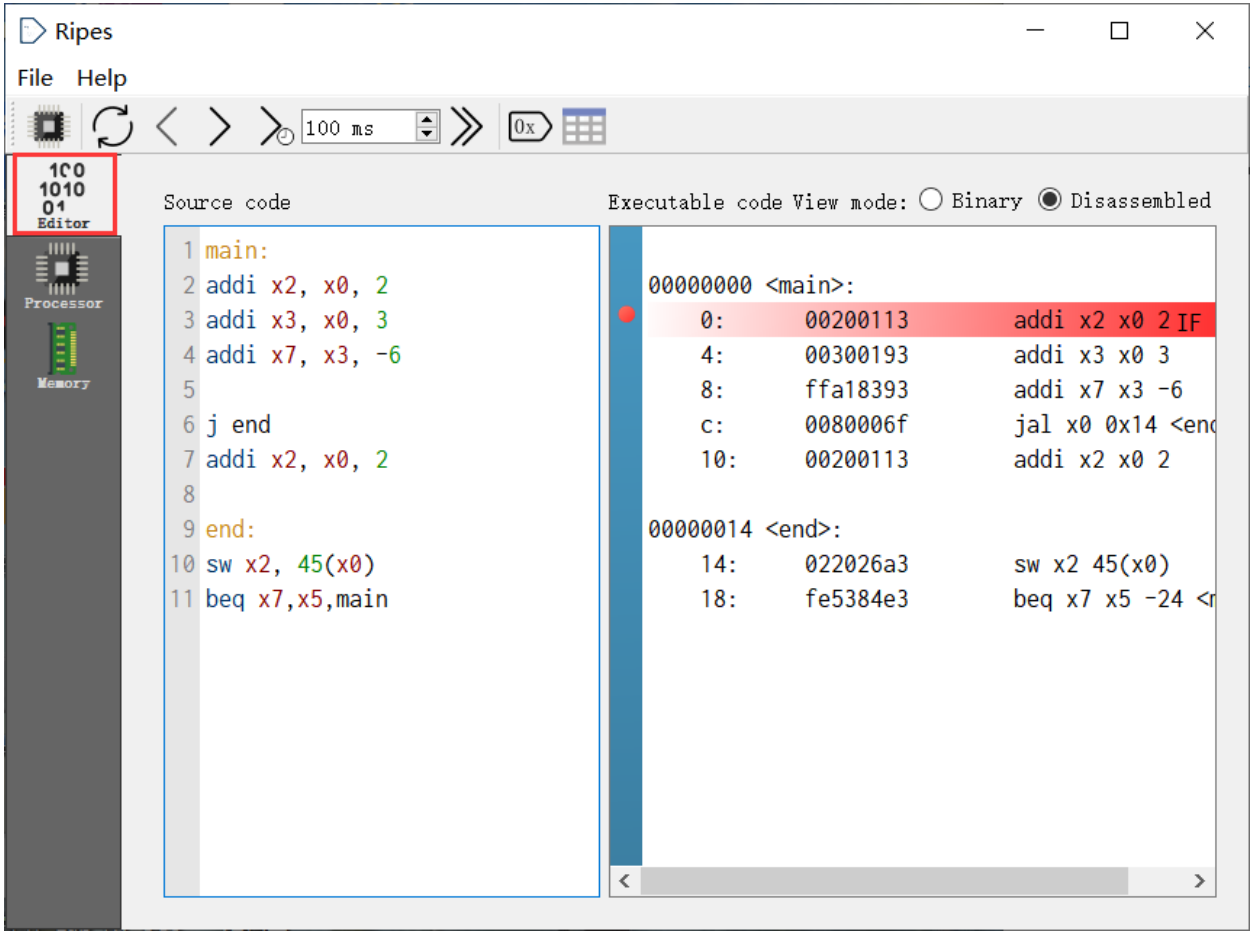
操作数（opr）

- 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）*
- 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

- 寄存器
- 存储器
  - 内存地址
  - 字长：sw
- 立即数
  - 进制表示
    - 十进制
    - 十六进制
      - 0x12345
    - 二进制
      - 0b1101
- 标号/行号
  - main，end



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

RV典型内存地址空间分配

- 段式
  - DATA
  - Stack/Heap
    - 栈：自高向低
    - 堆：自低向高
  - CODE
- I/O port?
- 硬盘
- 网络

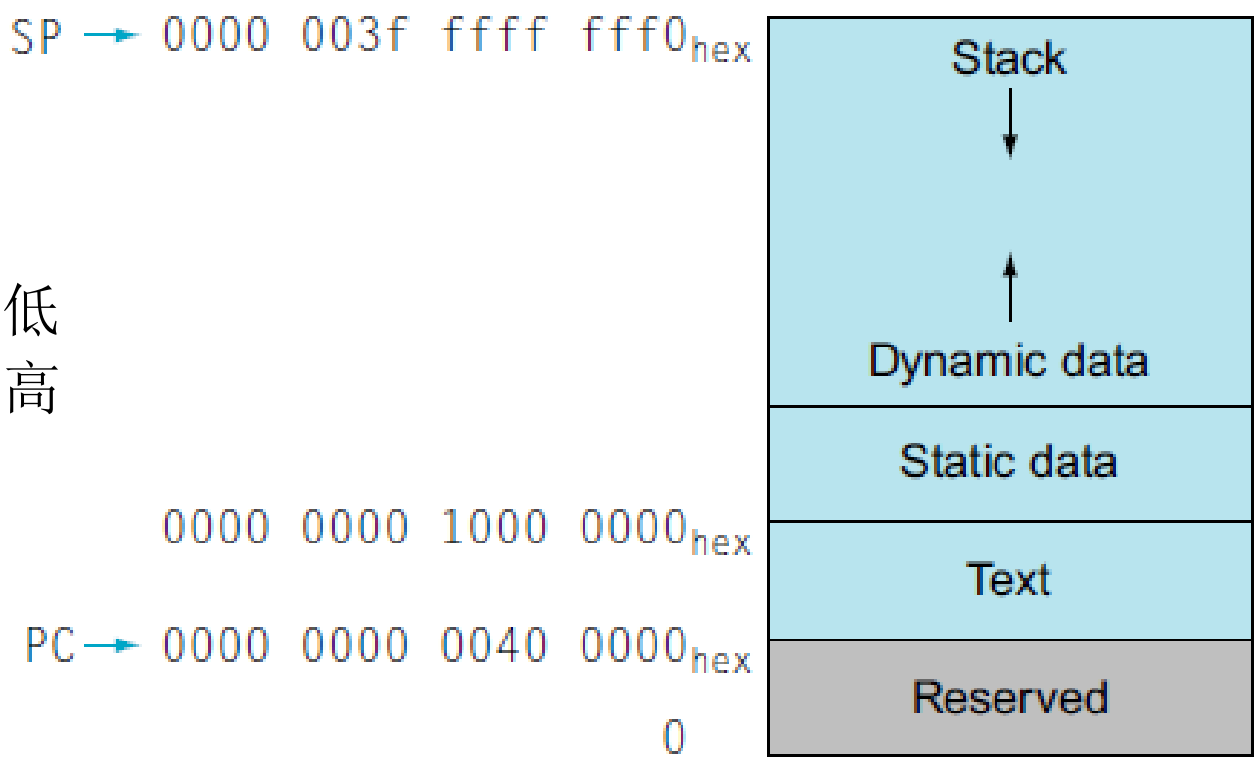


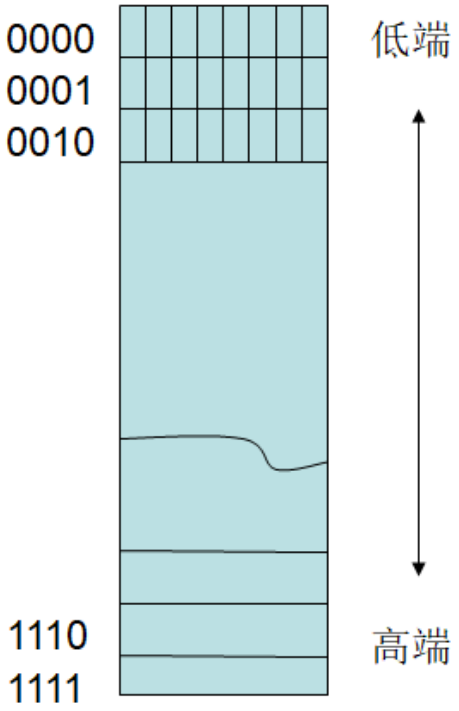
图2-13 用户地址空间划分

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）

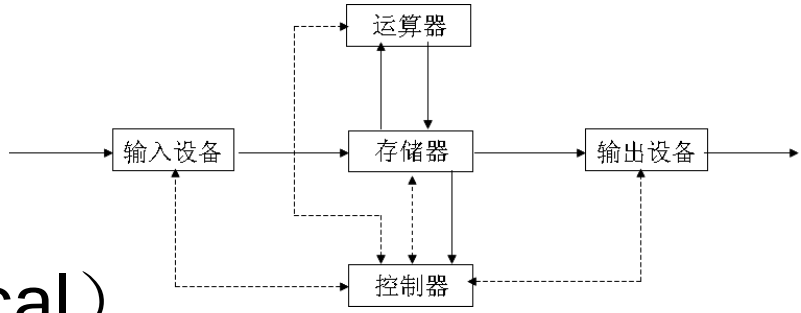
- 在数据不对准边界的计算机中，数据（例如一个字）可能在两个存储单元中。
  - 此时需要访问两次存储器，并对高低字节的位置进行调整后，才能取得一字。
- 边界对齐：
  - 字对齐：左移两位，按字访问
    - RV/x86不要求，MIPS要求
  - 半字：左移一位，按半字访问

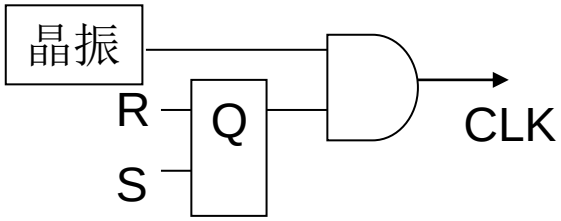
存储器 **地址（十进制）**

<table>
<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

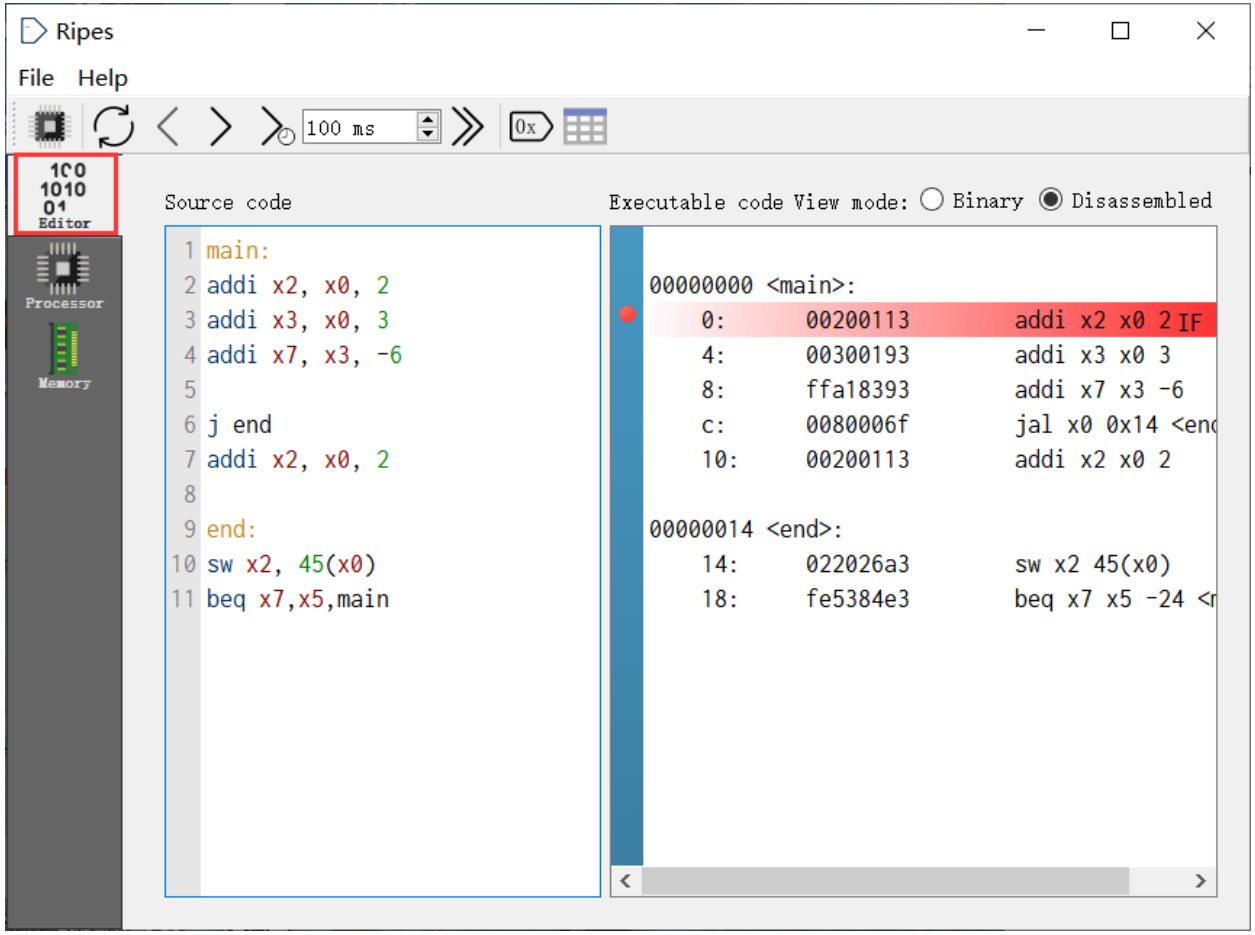
- 数据传递（data movement）
  - 访存：load，store，*mov*
  - I/O：i*n*，*out*
- 算逻运算（arithmetic & logical）
  - add，sub，and，not，or，xor，*dec，inc，cmp*
  - monadic & dyadic operations
- 移位操作
  - monadic operations：shl，shr，srl，srr
- 分支控制（tranfer of contral，Branch）
  - comparisons & conditional branches：beq，bnz
  - 无条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





指令示例

- ALU
  - 源操作数
    - 立即数
  - 目的操作数
- 分支
  - NPC
- 访存
  - 内存地址



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

- 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

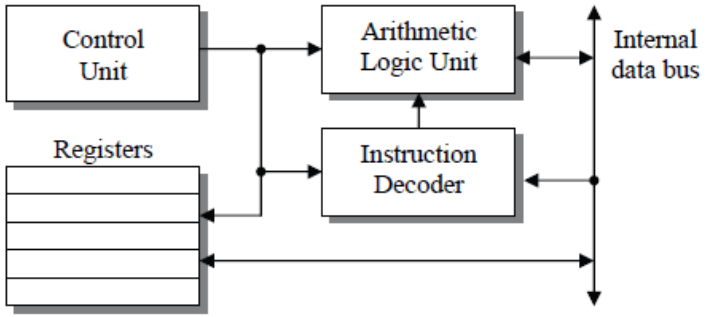
地址码：操作数，指令

- 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

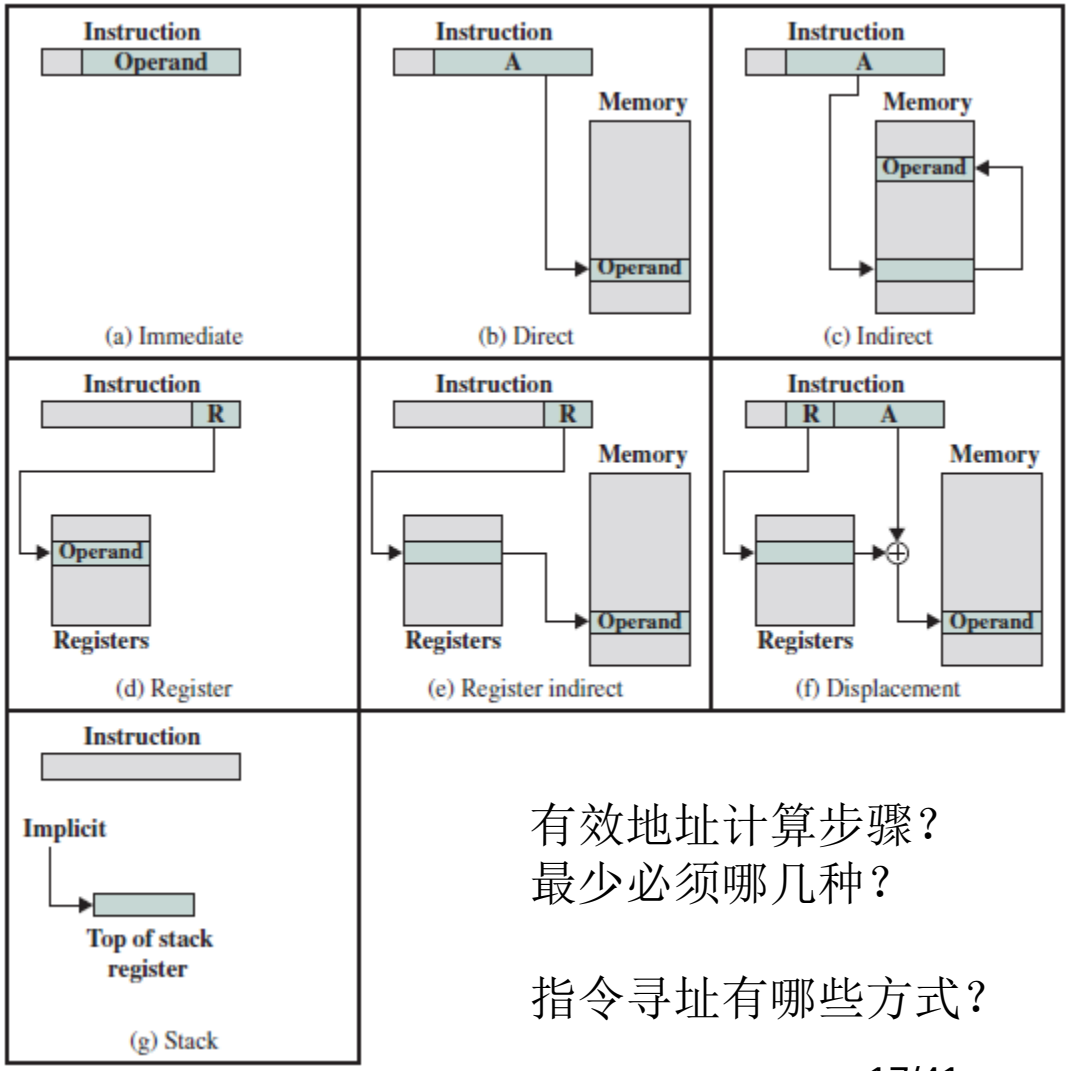
- 寻址方式：指令字和操作数的存储地址计算方式
- 指令寻址：现代CPU利用PC
  - 顺序执行：每执行一条指令，PC自动1
  - 跳转：更新PC，转移到目的地址执行
- 操作数寻址
  - 指令中给出“形式地址”
  - 有效地址：操作数在寄存器/内存中的物理地址
    - EA＝寻址方式＋形式地址

操作码	形式地址



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）*

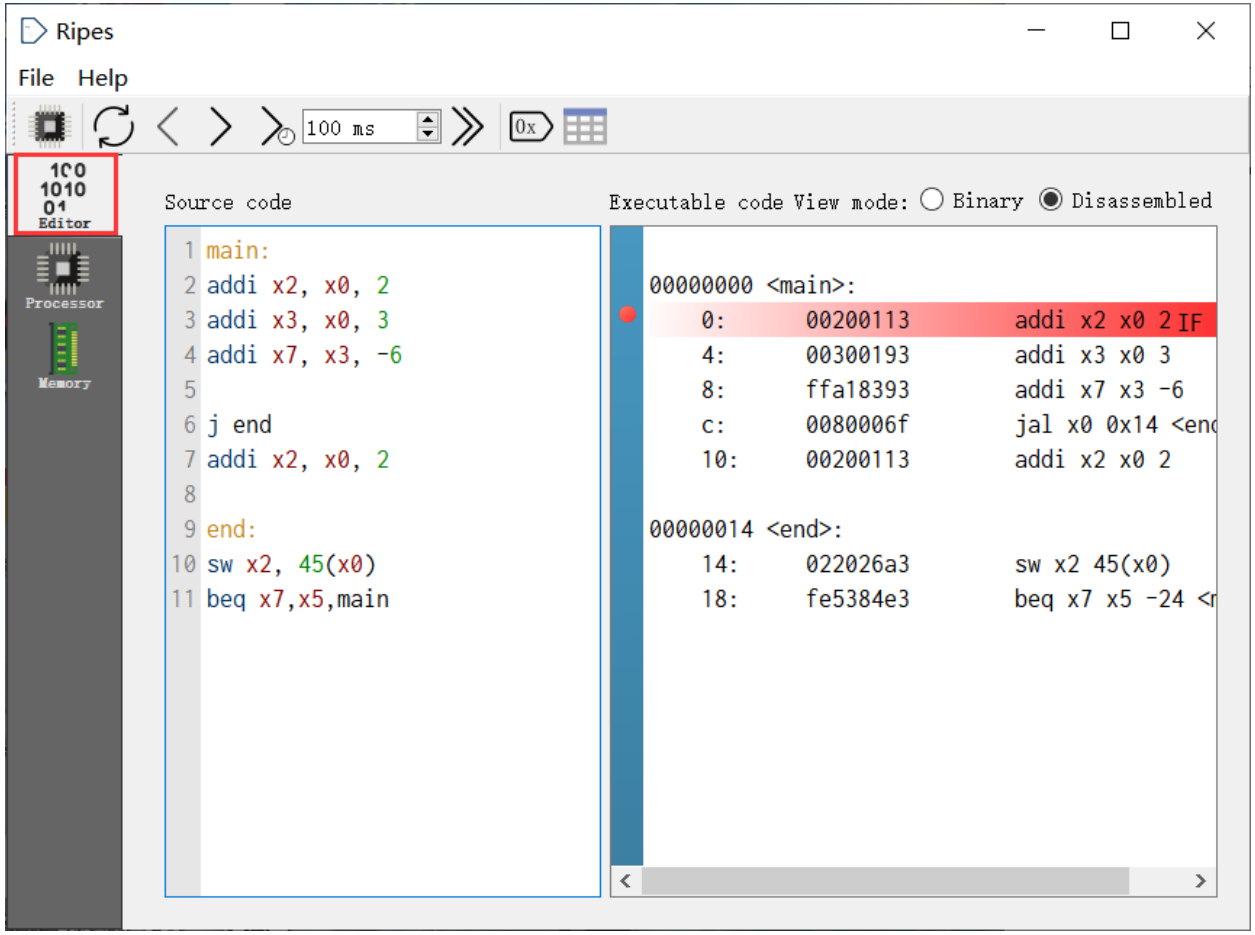


有效地址计算步骤？
最少必须哪几种？

指令寻址有哪些方式？

寻址方式示例

- addi
- j
- sw
- beq

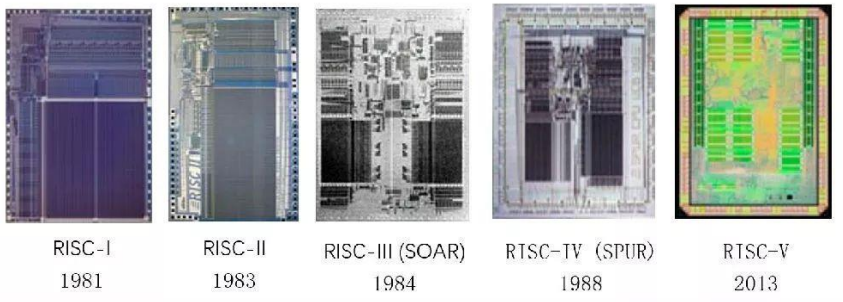


RISC-V ISA的特点

- 模块化：51+13+133
  - 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）+ 图2-37（14条）
  - 系统指令：同步，CSR操作，异常。图5-47，13条
  - RV32IMFD指令集： RV32I的基本扩展，133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

图2-38

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-V指令格式与寻址方式

图2-19

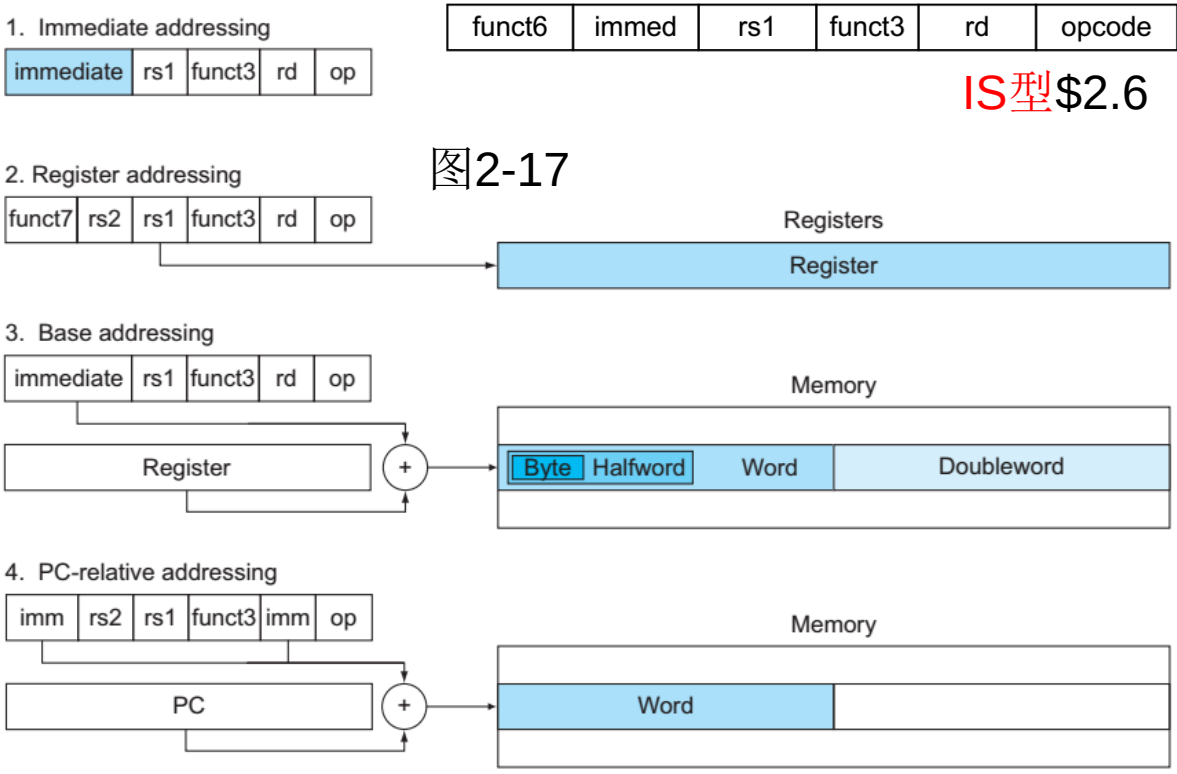
Name (Field size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	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

funct6	immed	rs1	funct3	rd	opcode

IS型$2.6

- 指令格式：6种
  - 基本：R/I/S/U
    - 7种：4+2+“IS”
    - 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，jal
    - 间接跳转：jalr x0，100(x1)

图2-17

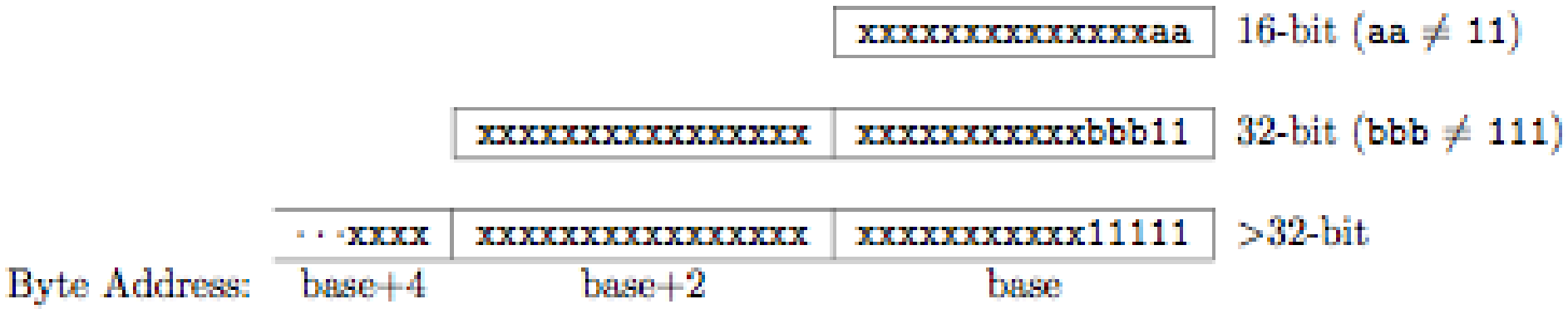


RV整数指令操作码

- 常用37条，图2-18
  - 按字长分类：访存指令
    - b，h，w，d
  - 按数据类型分类
    - i，u
  - 按指令格式分类
- 典型：按功能分类
  - 同功能：op同，funct不同
  - ALU，访存，分支
    - add：R-type
    - addi：I-type
    - lw：load，I-type
    - sw：store，S-type
    - beq：SB-type
    - jal（J-type），jalr（I-type）
- *51条指令示例：图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.

RV16/RV32/RV64的操作码域：7位



- Can support variable-length instructions.
- Base instruction set (RV32) always has fixed 32-bit instructions lowest two bits = 11_2
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)

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【二/十/16进制】，基址寻址【1000(x2)】
  - **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

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”
- data move：reg-reg

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

- 寄存器赋值

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

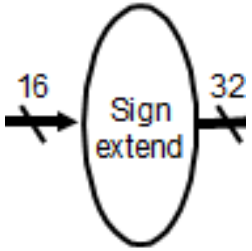
- Compare

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

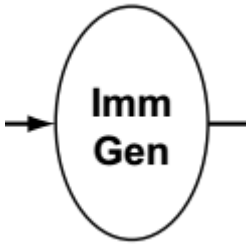
- Goto：beq x0，x0，Exit

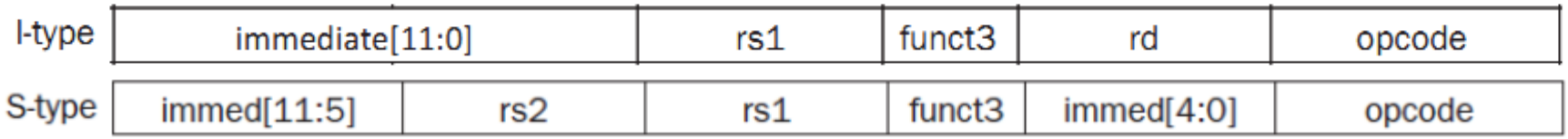
位扩展：短立即数=>32位立即数，$2.4

- 位扩展：高位填充，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/SB-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 goto nPC+7, else not taken。





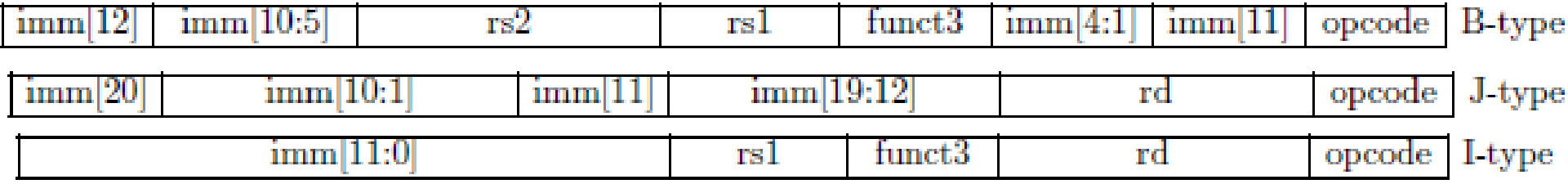
32位常数，“双指令序列”法

- 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
  - lui：取高20位——$2.18的auipc？
    - 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

转移指令的寻址方式$2.7，2.10.2，2.8

- 两类转移
  - 分支指令：if，for，while，case，goto
    - 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，间接转移，12位offset
      - beq x0，x0，Loop；B-type
  - 过程调用：x1为返回地址ra
    - 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）



其他RV指令（了解），$2.18，$5.14

- 图2-37
  - 14条

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

- lui，auipc
- lr.d，sc.d；$2.11

- 图5-47，系统指令，13条
  - 同步，CSR访问，系统调用
  - CSR：控制和状态寄存器

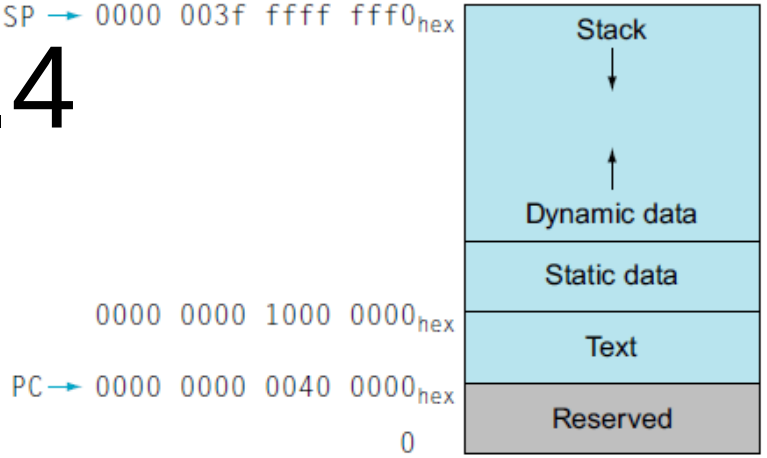
Type	Mnemonic	Name
Mem. Ordering	FENCE.I	Instruction Fence
	FENCE	Fence
	SFENCE.VMA	Address Translation Fence
CSR Access	CSRRWI	CSR Read/Write Immediate
	CSRRSI	CSR Read/Set Immediate
	CSRRCI	CSR Read/Clear Immediate
	CSRRW	CSR Read/Write
	CSRRS	CSR Read/Set
	CSRRC	CSR Read/Clear
System	ECALL	Environment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

示例：COD4图B.1.5

```
   #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);
}
```

示例：COD4图B.1.4

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

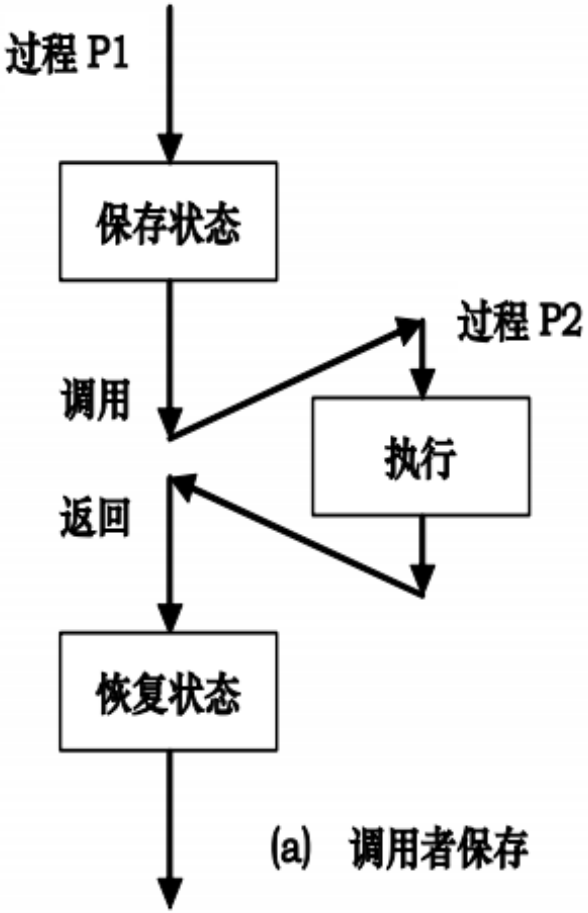


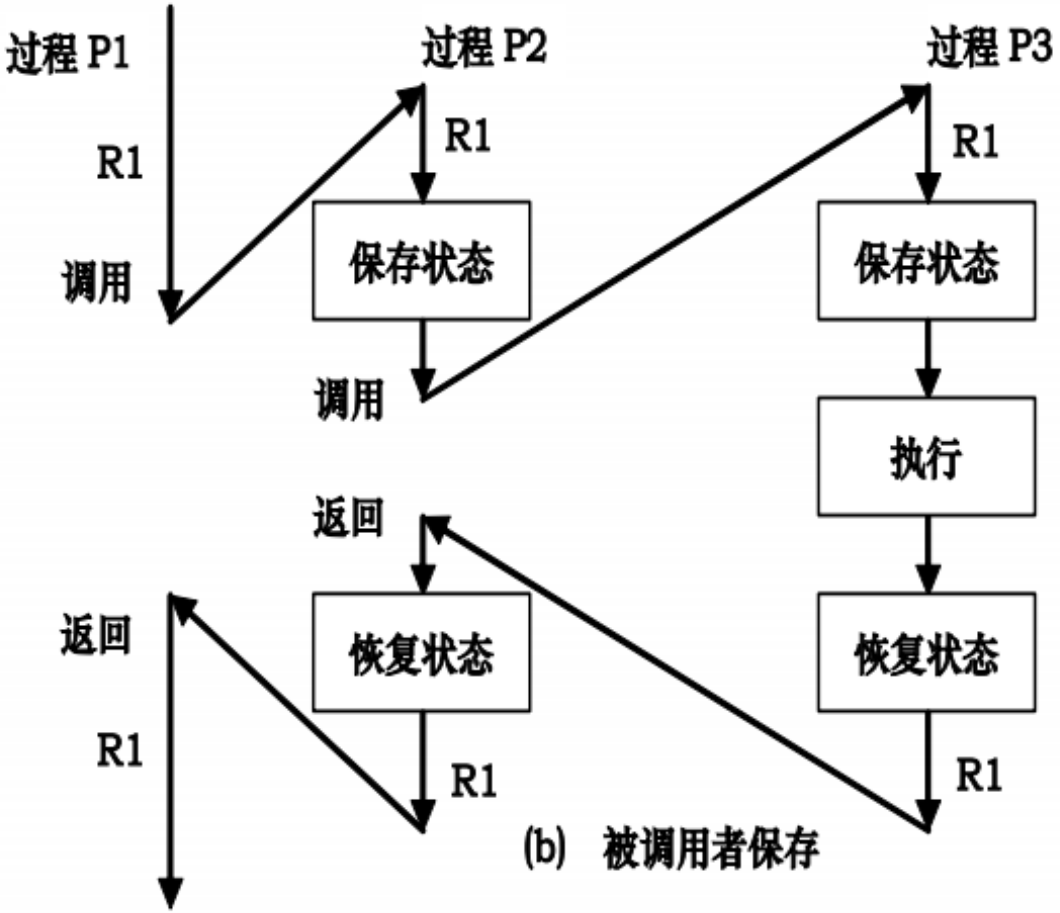
Name（ABI name）	Reg#	Usage
x0 (zero)	0	The constant value 0
x1 (ra)	1	Return address (link register)
x2 (sp)	2	Stack pointer
x3 (gp)	3	Global pointer
x4 (tp)	4	Thread pointer
x5-x7 (t0-t2)	5–7	Temporaries
x8-x9 (fp/s0-s1)	8-9	Frame pointer，Saved register
x10-x17 (a0-a7)	10–17	Arguments(a2-a7)/results(a0, a1)
x18-x27 (s2-s11)	18–27	Saved register
x28-x31 (t3-t6)	28–31	Temporaries

过程调用：断点，传参，现场保存（两种），$2.8

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

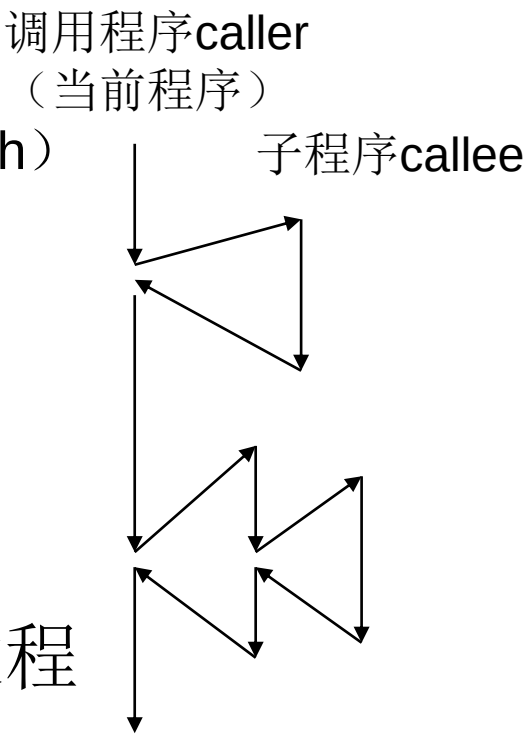
```
long long int fact (long long int n)
{       if (n < 1) return (1);
        else return (n * fact(n − 1));
}
```



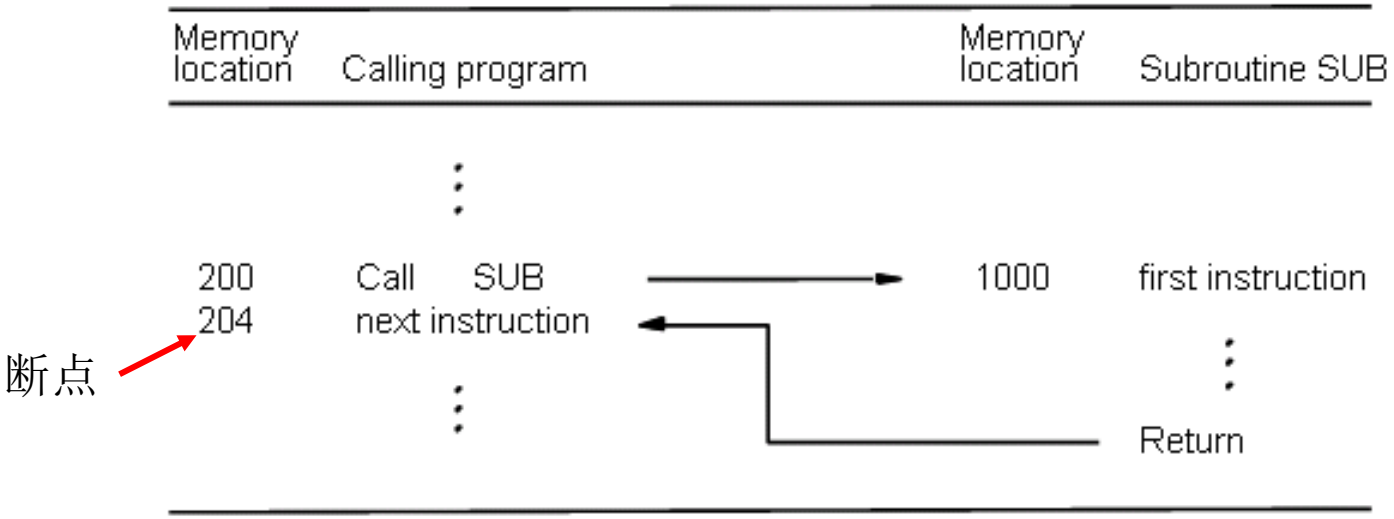


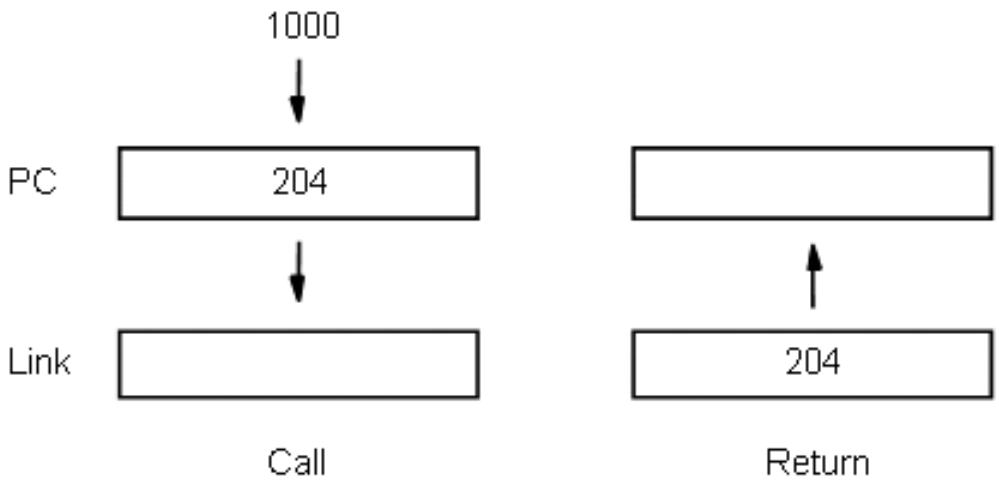
过程调用procedure calling

- 步骤：约定被调函数保存状态
  - 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
      - 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：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
- 保留寄存器
- 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：链接寄存器x1/返回地址ra





RV architected registers：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	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

图2-14

- Preserved：在函数调用中应保持不变

数组排序：swap()，sort()，$2.13

- 过程调用：参数传递，call/jal，return/jalr

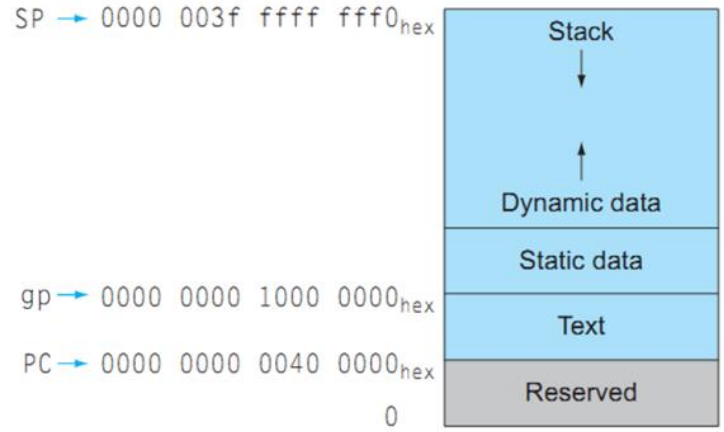
```
void swap(int v[], size_t k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

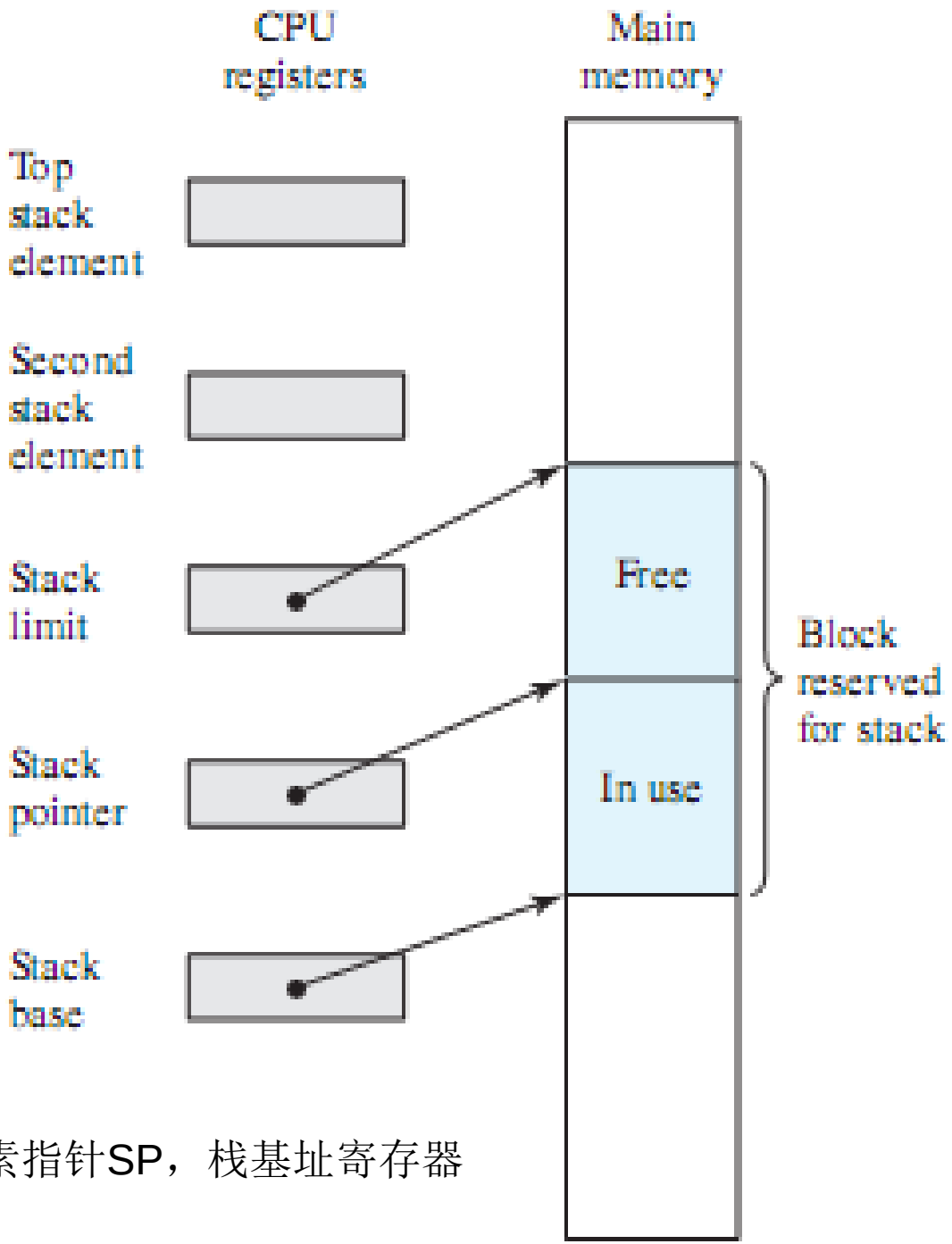
```
swap:
  slli   x6, x11, 2   // reg x6 = k * 4
  add    x6, x10, x6  // reg x6 = v + (k * 4)
  lw     x5, 0(x6)    // reg x5 (temp) = v[k]
  lw     x7, 4(x6)    // reg x7 = v[k + 1]
  sw     x7, 0(x6)    // v[k] = reg x7
  sw     x5, 4(x6)    // v[k+1] = reg x5 (temp)
  jalr   x0, 0(x1)    // return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

```
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);
        }
   }
}
```

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

stack





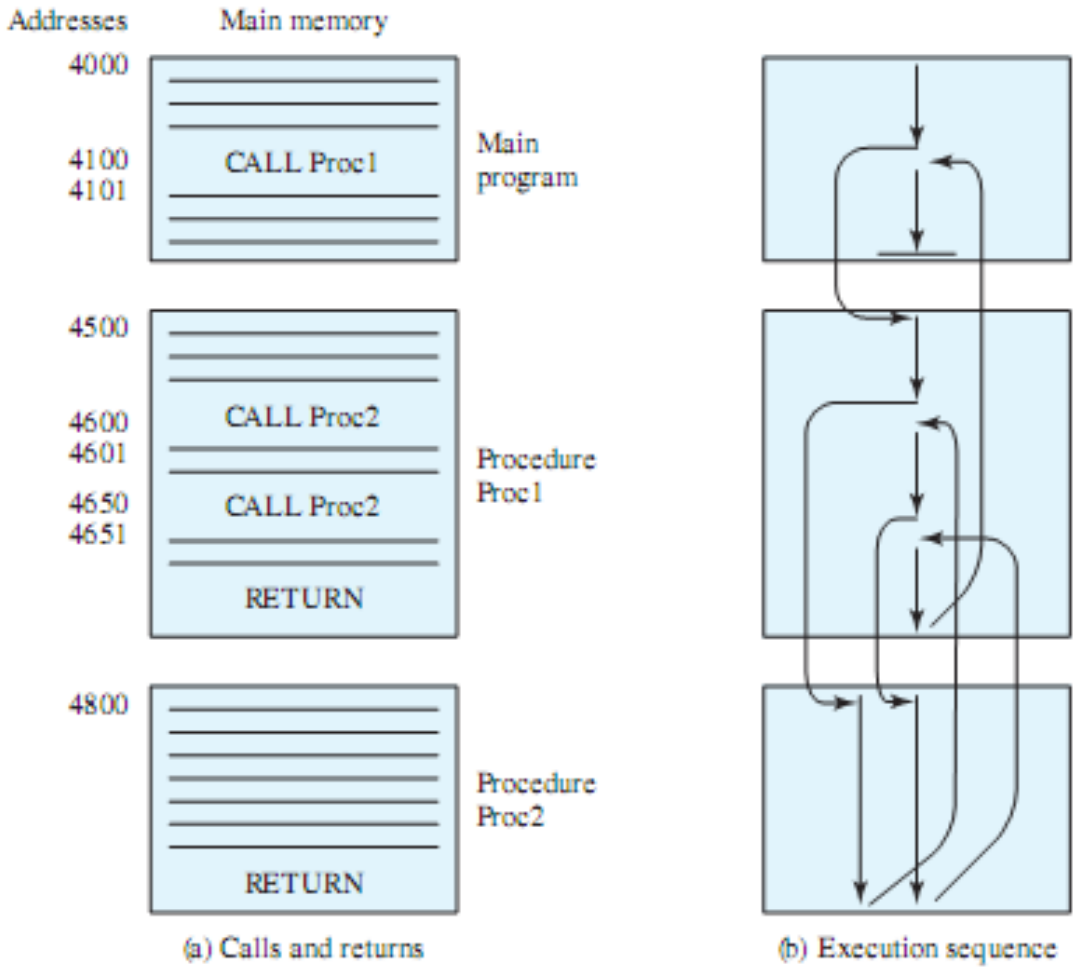
常见CPU寄存器：栈顶元素指针SP，栈基址寄存器

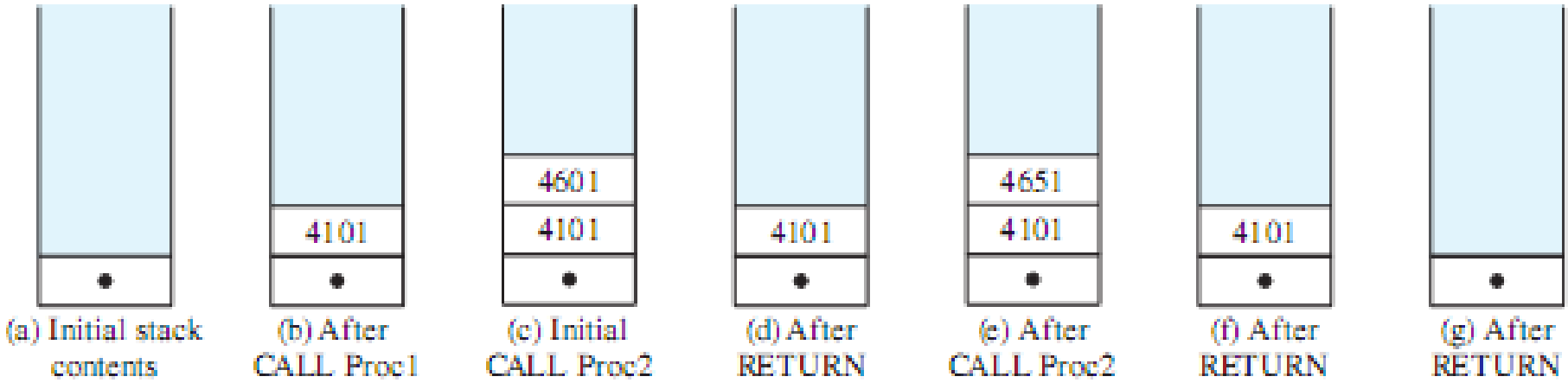
Use of Stack to Implement Nested Procedures

注意：

满堆栈

仅保存了断点





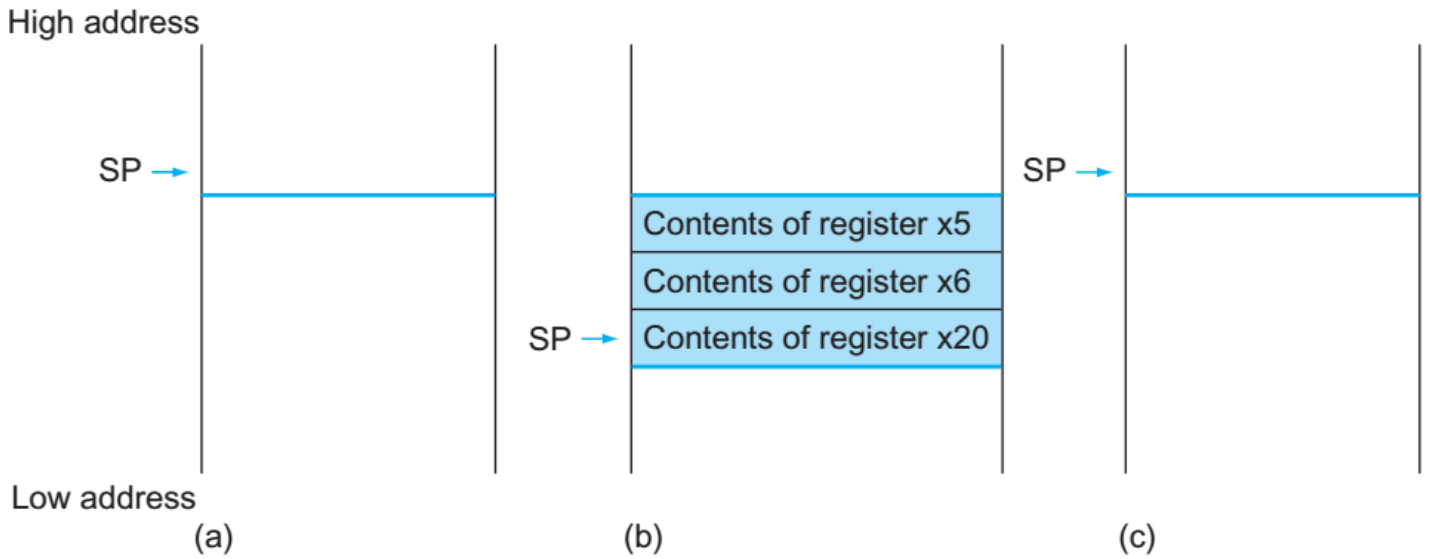
RV堆栈操作：push/pop，图2-10

入栈

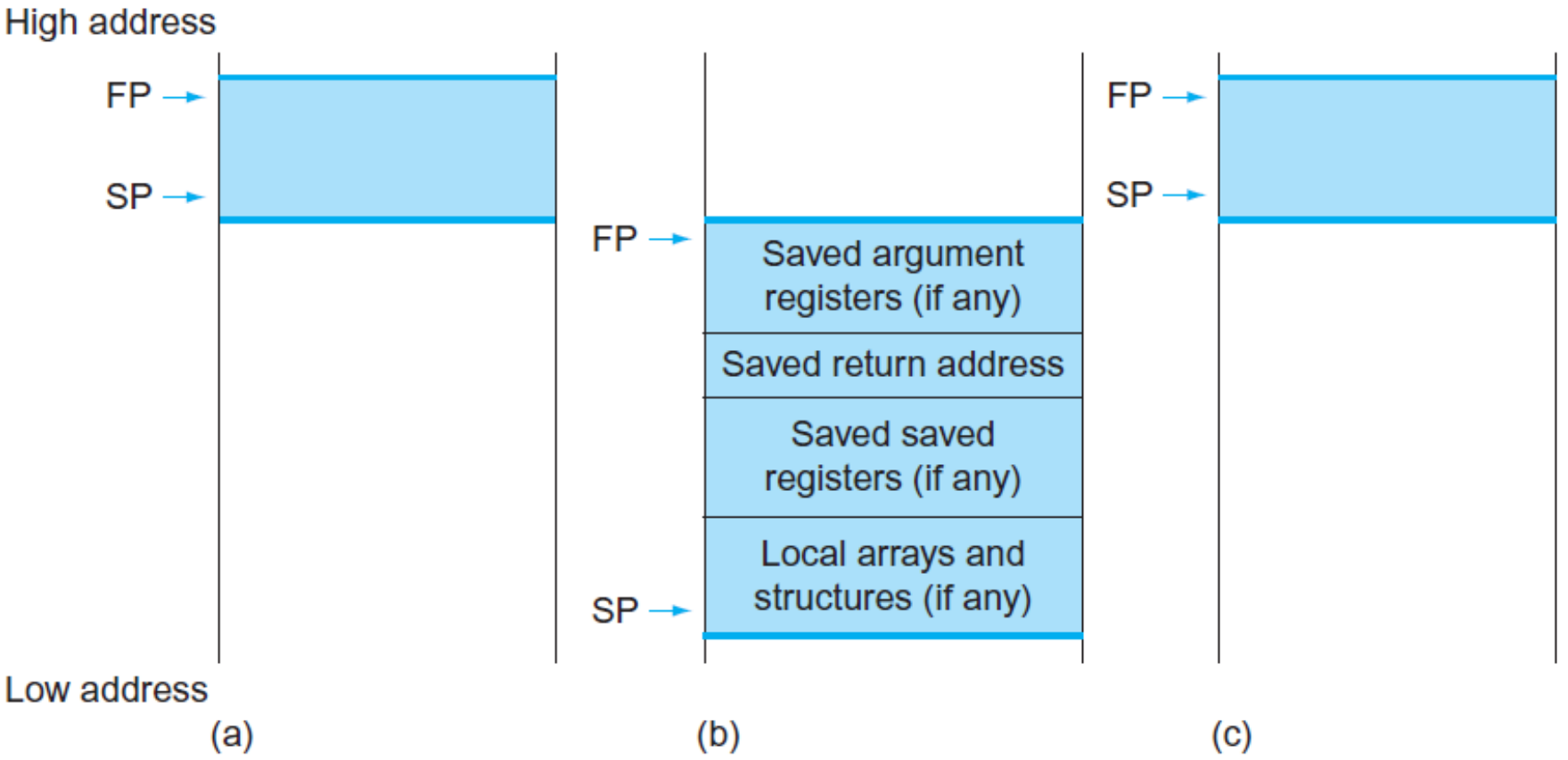
```
addi sp, sp, -24          // adjust stack to make room for 3 items
sd   x5, 16(sp)           // save register x5 for use afterwards
sd   x6, 8(sp)            // save register x6 for use afterwards
sd   x20, 0(sp)           // save register x20 for use afterwards
```

出栈

```
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
```

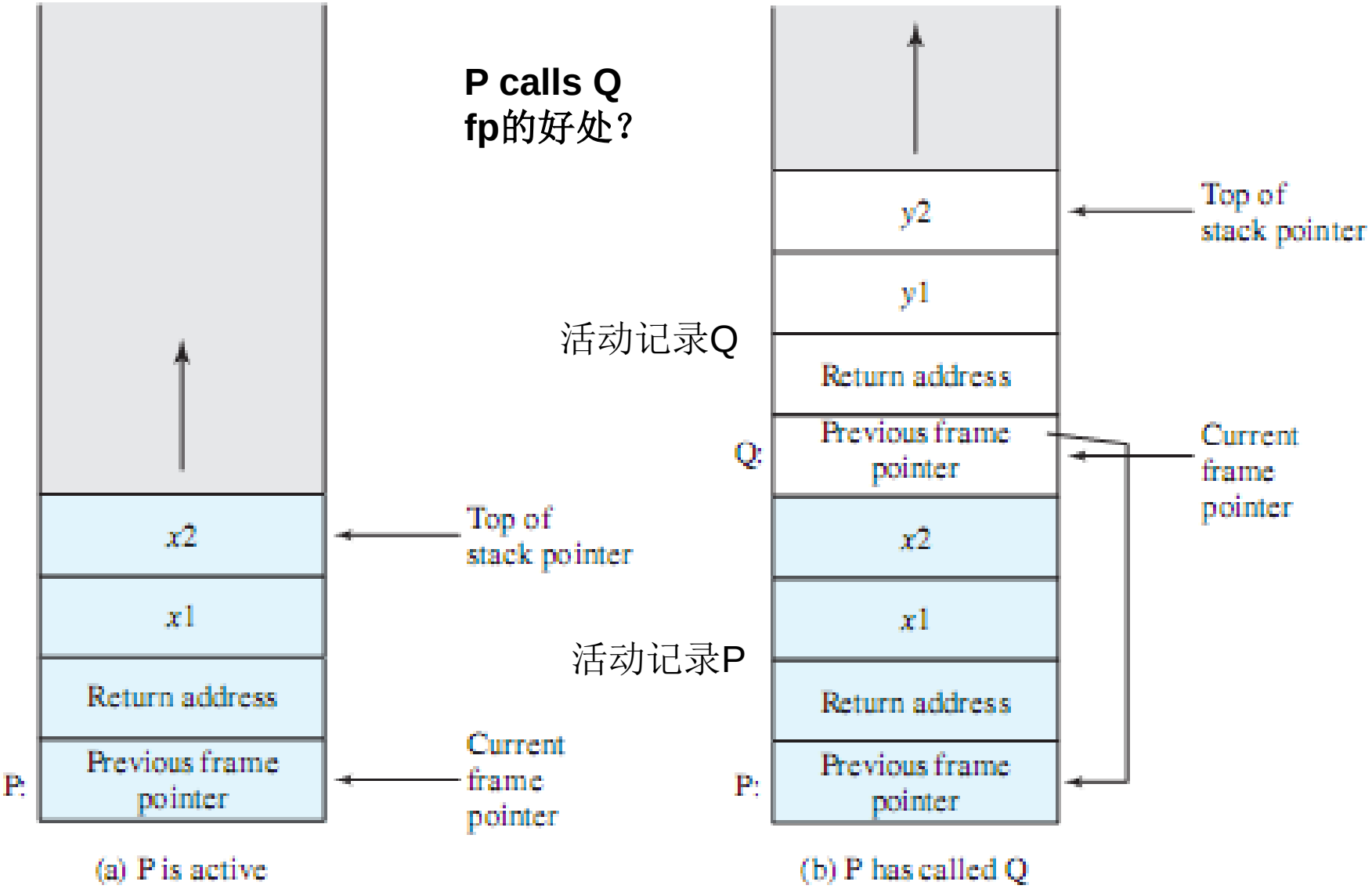


RV的过程帧（栈帧），图2-12



- 参数，断点，保留寄存器，局部变量

stack frame：活动记录，帧指针fp



RV calling conventions

- 传参和返回值：x10~x17
  - 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 register：返回ra
- 状态（现场）保存策略：callee负责保存

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

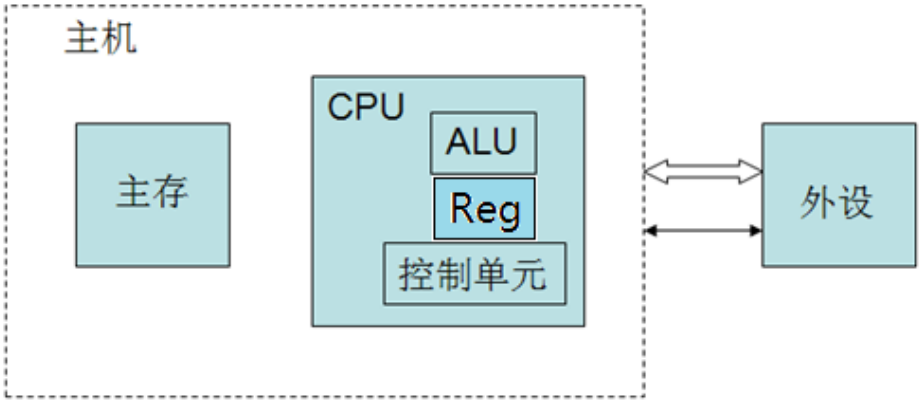
图2-14

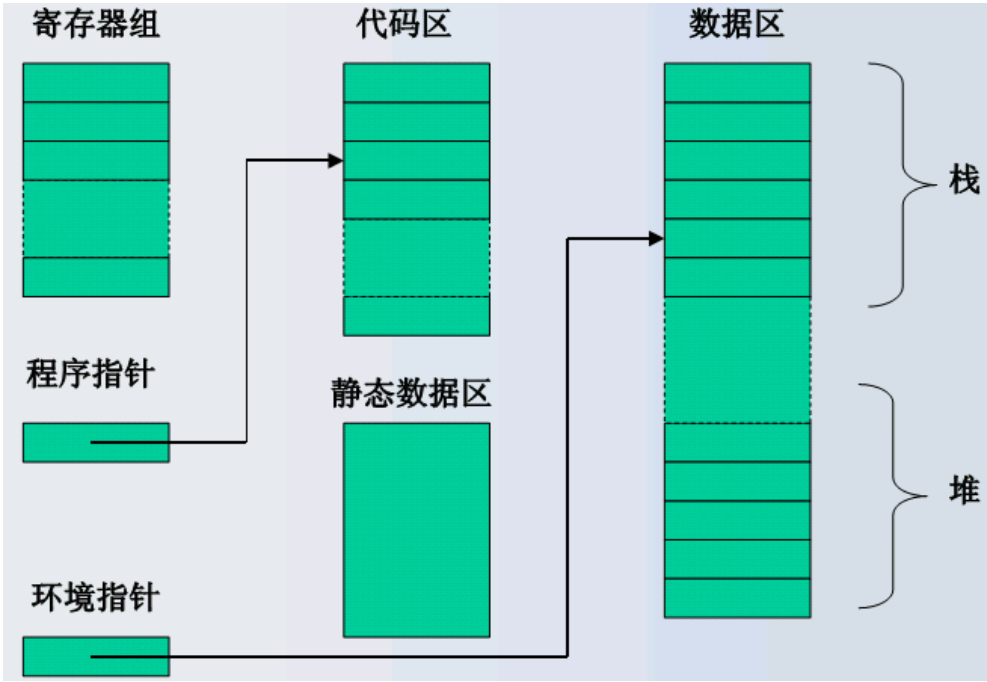
图2-11

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

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，*浮点*，伪指令（图2-40）
  - 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



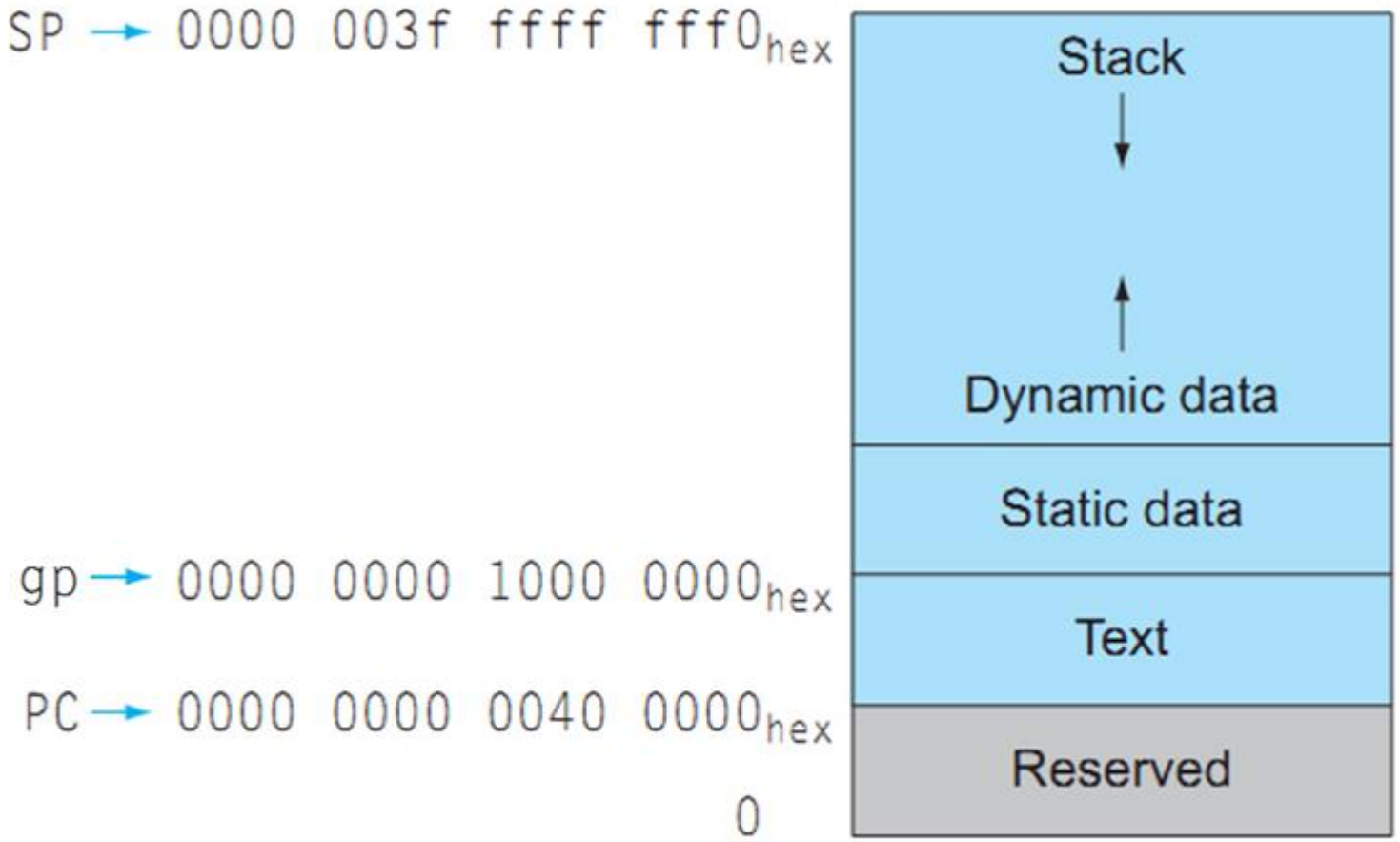


Policy of Use Conventions for registers

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

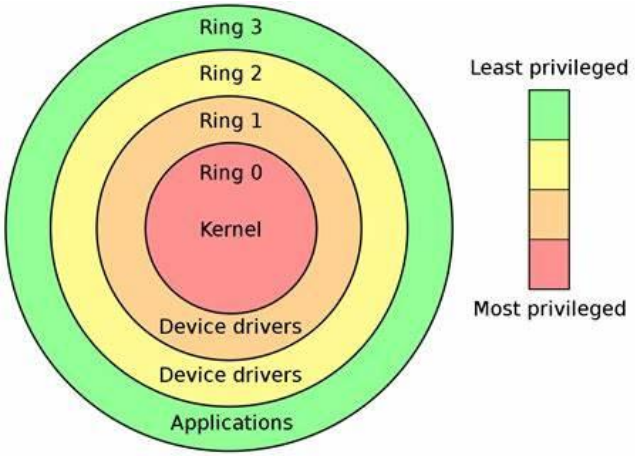
RV内存分配约定，图2-13



P&W中，RV32I的pc从0x0001 0000开始

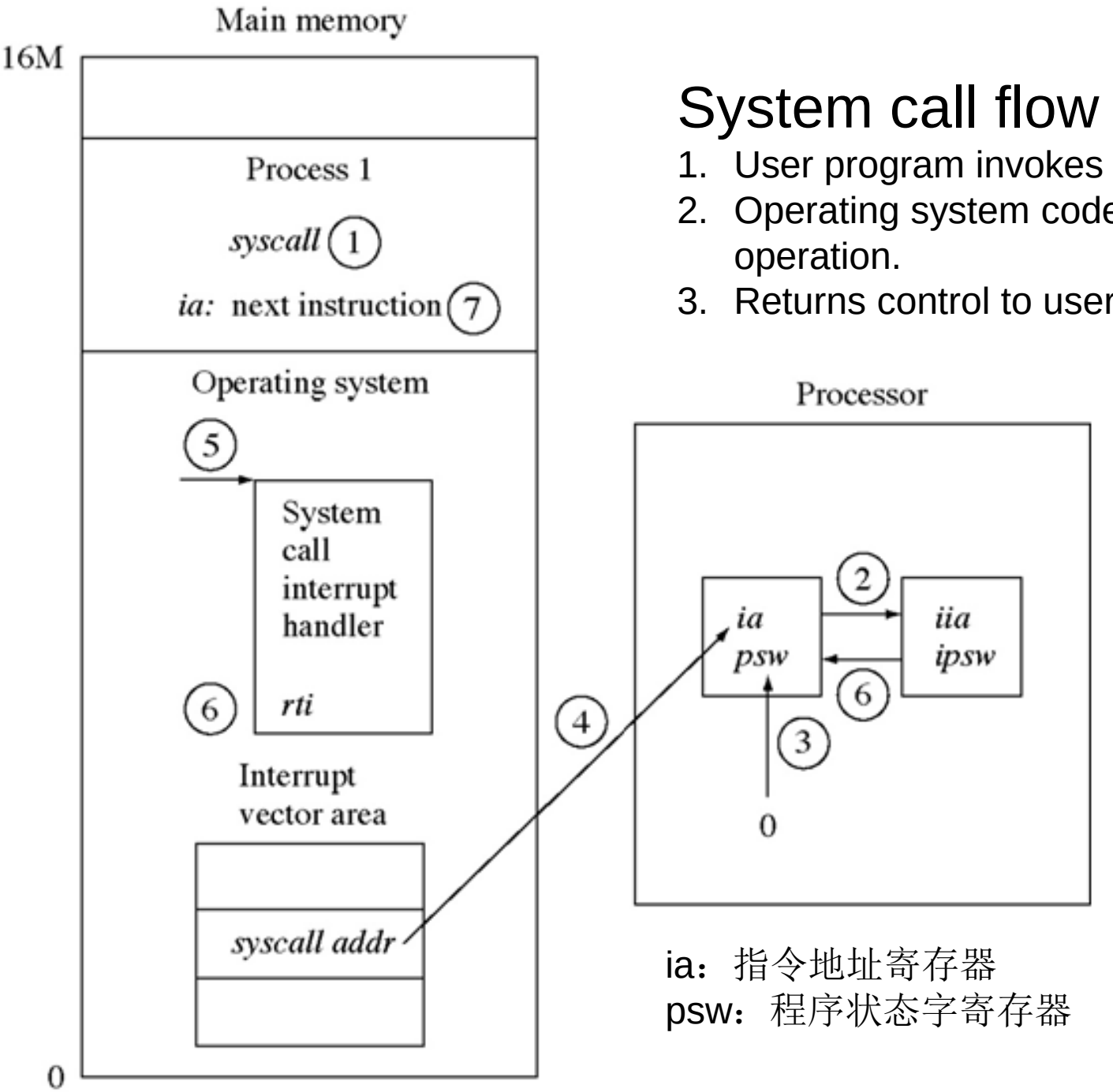
System calls

- 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.



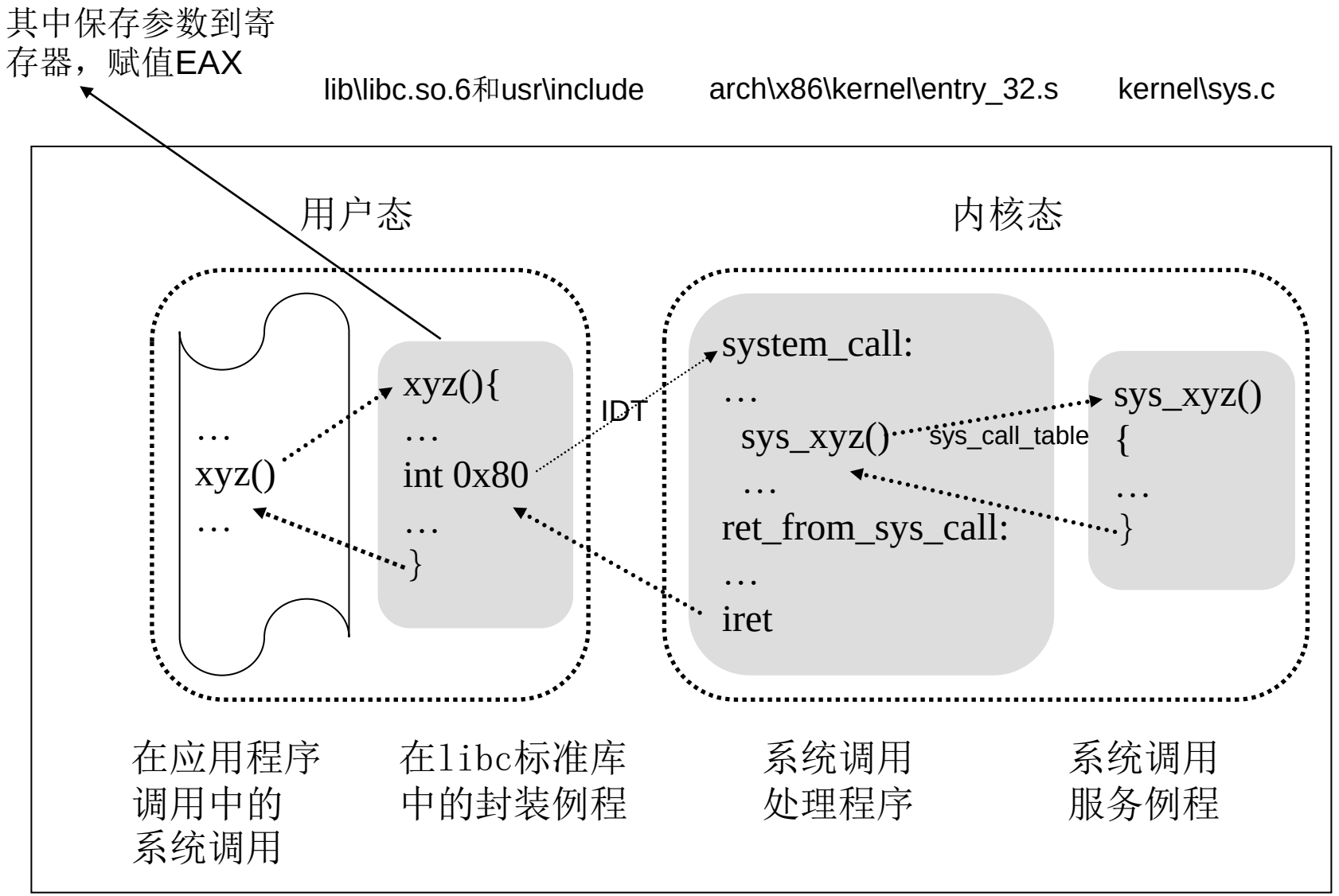
ia：指令地址寄存器
psw：程序状态字寄存器

2/3/4由软中断指令int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

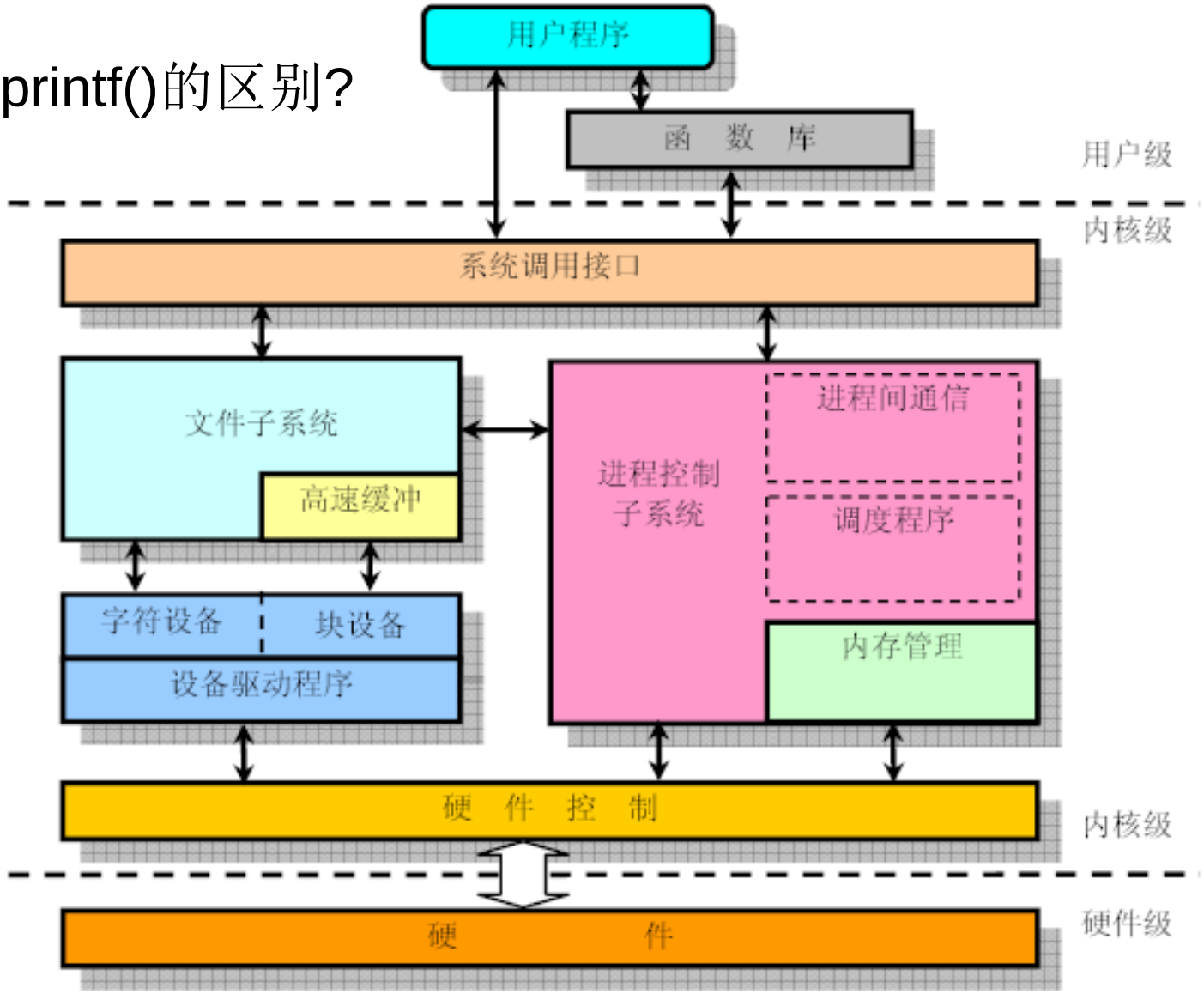
sys_call_table
sys_call_ISR

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

C语言和OS服务：APIs、libc、syscalls?

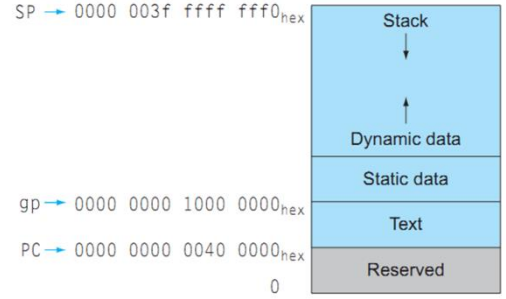
abs()与printf()的区别?

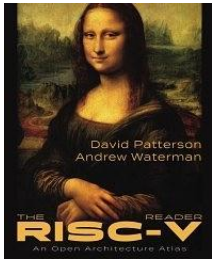


RV汇编程序结构：《P&W》例

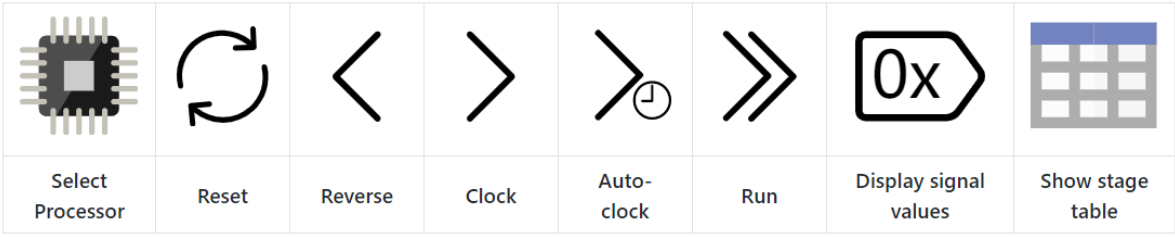
```
  .text                    # Directive: enter text section
  .align 2                 # Directive: align code to 2^2 bytes
  .globl main              # Directive: declare global symbol main
main:                      # label for start of main
  addi sp,sp,-16           # allocate stack frame
  sw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





Ripes汇编



Ripes

File Edit Help

100 ms

Editor

Processor

Memory

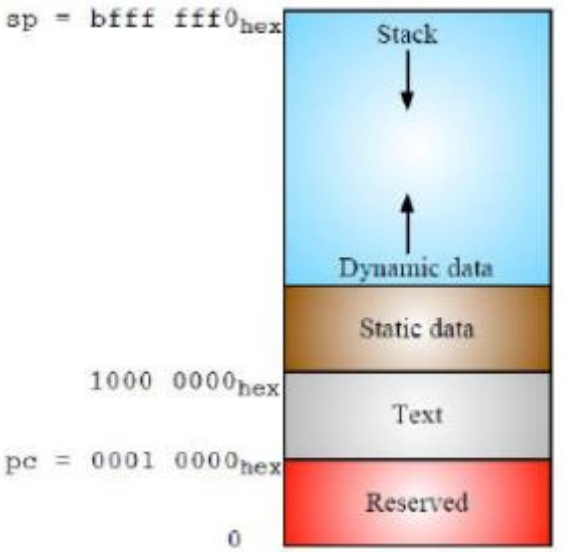
Source code

Input type: Assembly C Executable code

View mode: Binary Disassembled

```
1 .data
2 w: .word 0x1234
3
4 .text
5 lw   a0 w
6 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”！

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

COD-RV图5-47 syscall指令

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

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

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

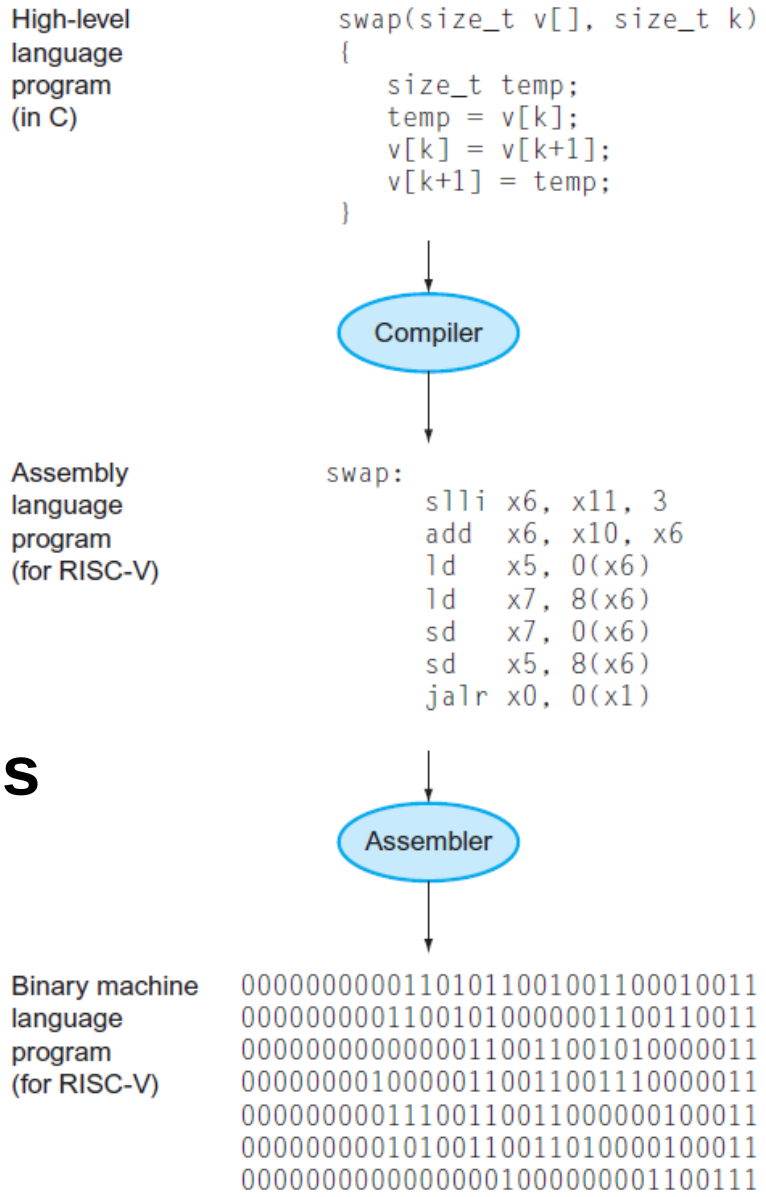
④ Swap occured? (Yes, goto ①)

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

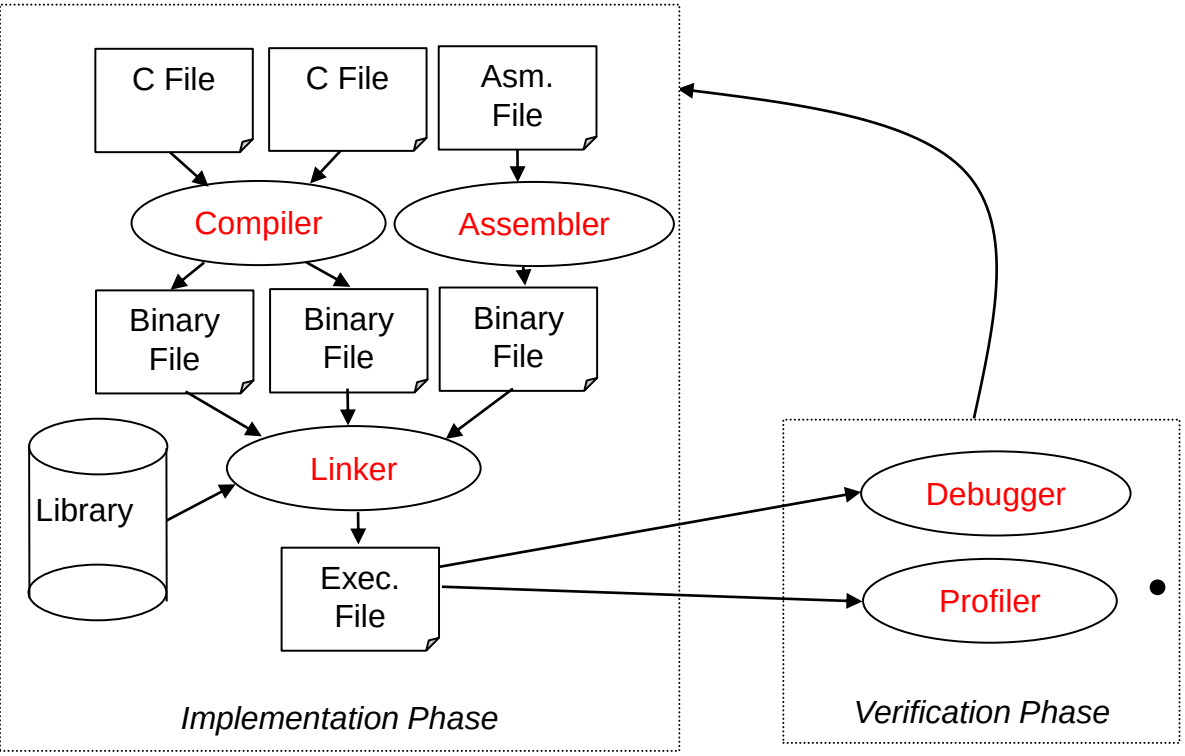
可执行程序生成与执行

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**

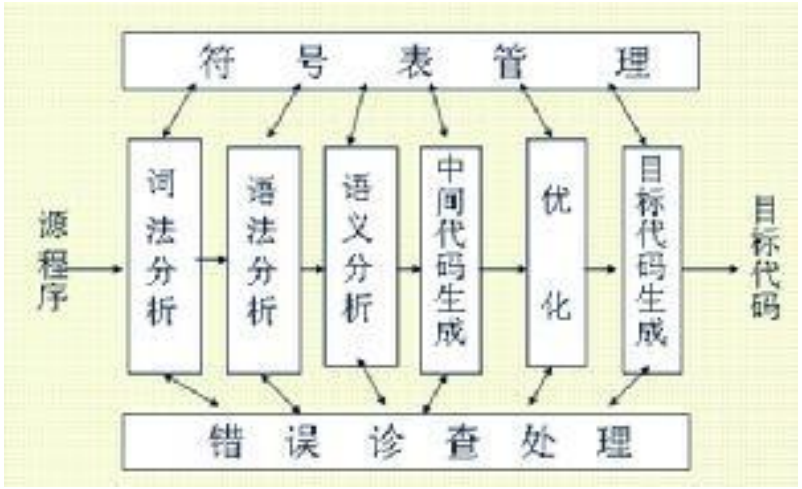


Program Development Process



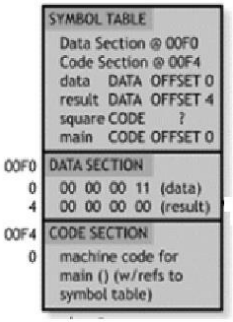
- *Implementation Phase*
  - editor
  - Compilers
    - Cross compiler
      - Runs on one processor, but generates code for another
  - Assemblers
  - Linkers
- *Verification Phase*
  - Debuggers
  - Profilers

A translation hierarchy for C，FIG 2.20

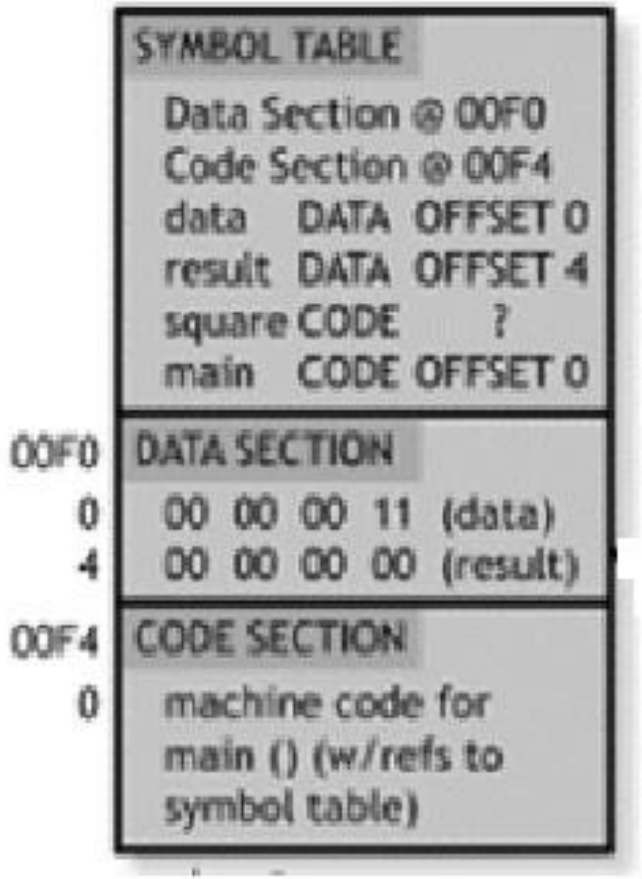


The Assembly Process：生成.obj

- 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



.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"
```

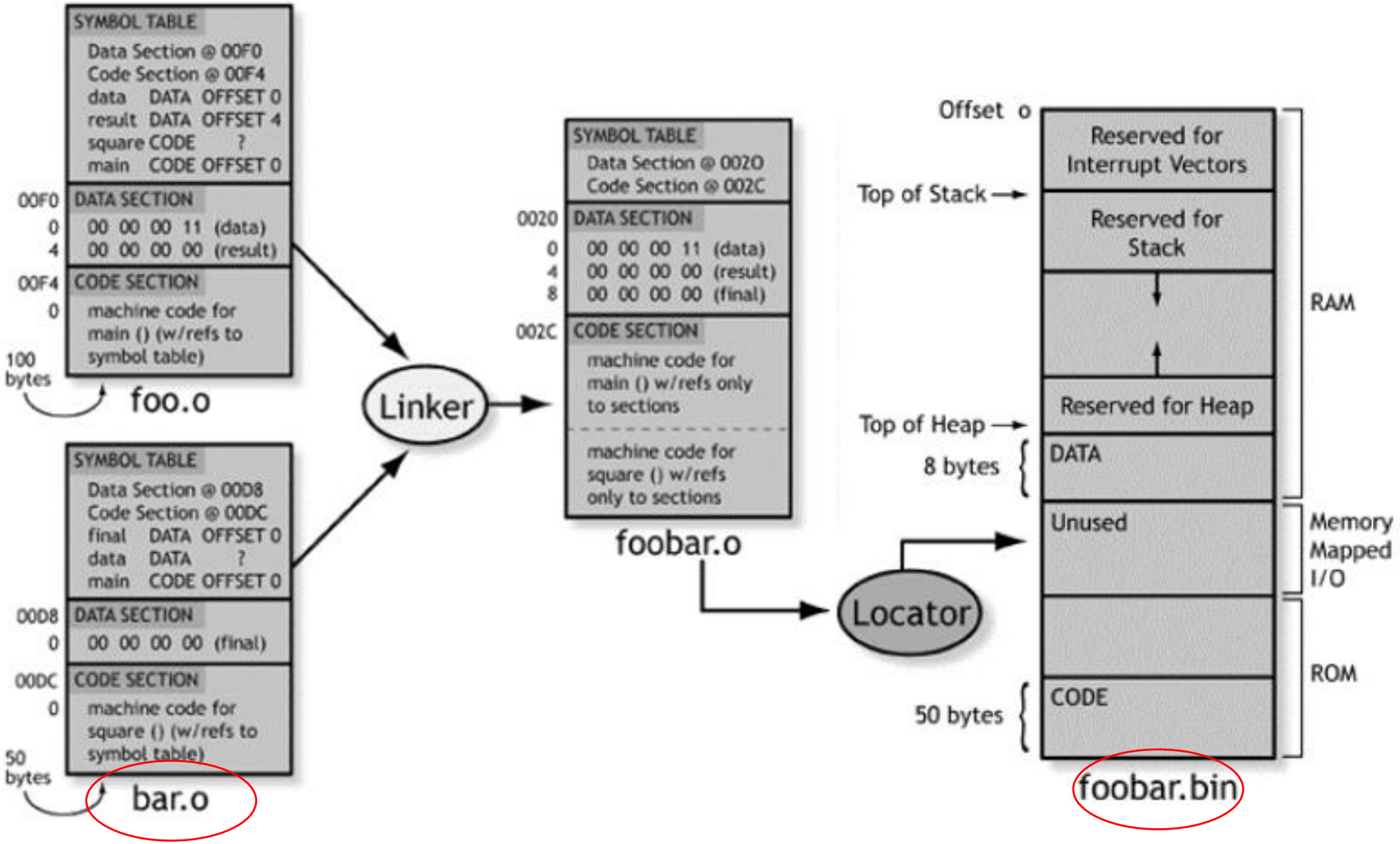
符号表：
全局定义和外部引用

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

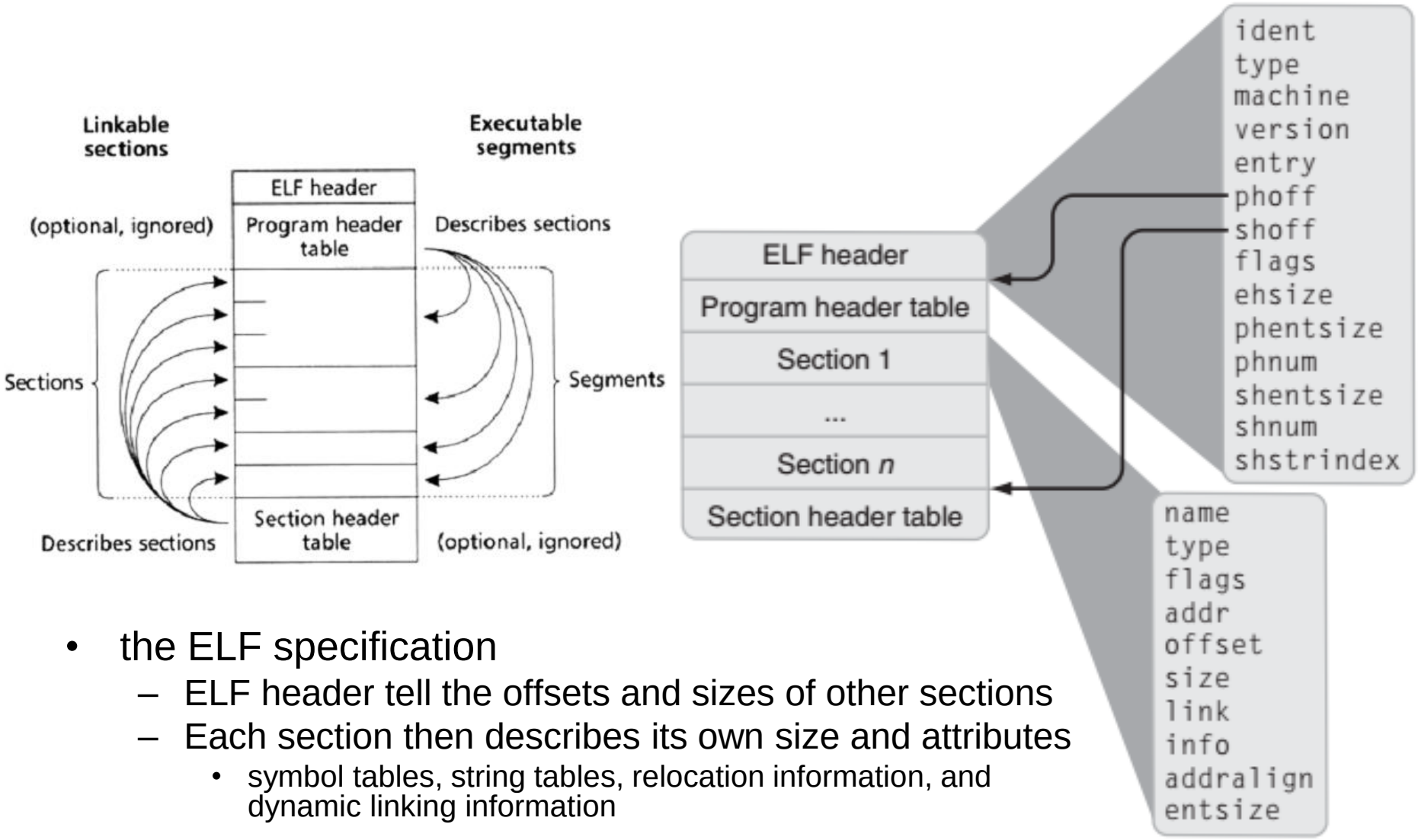
The Linker：合并各段

- 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

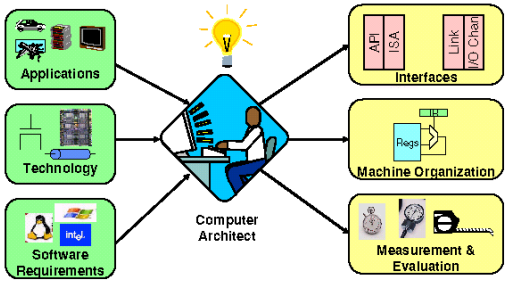
Loading/Executing Object Programs

- 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

ISA分类与进化

影响早期ISA设计的因素

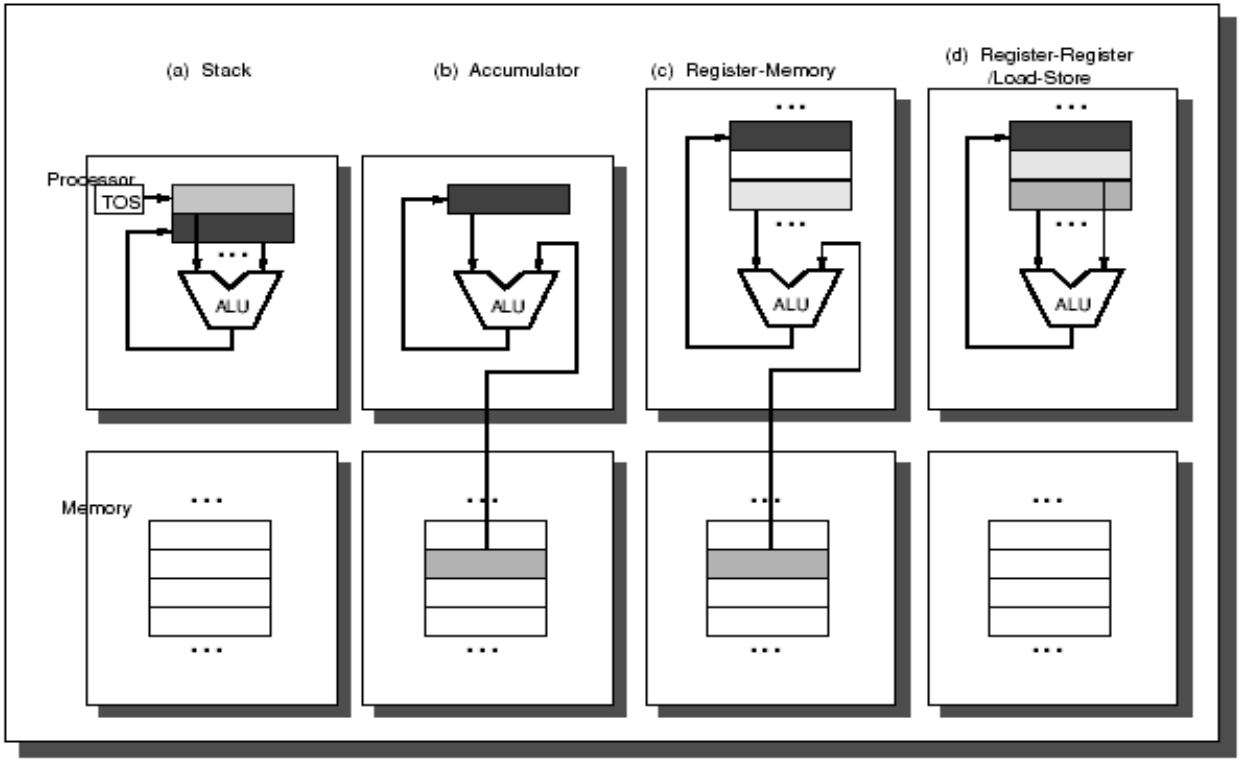
- 内存小而慢，能省则省
  - 某个完整系统只需几K字节
  - 指令长度不等、执行多个操作的指令
- 寄存器贵，少
  - 操作基于存储器
  - 多种寻址方式
- 编译技术尚未出现
  - 程序是以机器语言或汇编语言设计
  - 当时的看法是硬件比编译器更易设计
    - 为了便于编写程序，计算机架构师造出越来越复杂的指令，完成高级程序语言直接表达的功能
    - 进化中的痕迹：X86中的串操作指令



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）

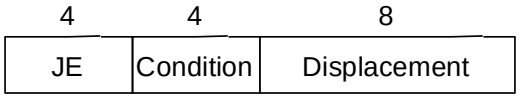
- CISC：硬件换性能！—上千条指令
  - 以机器指令实现高级语言功能
  - 指令译码复杂
    - 指令格式、字长不一（x86从1byte～6bytes）
    - 寻址方式多
  - 访存开销大：寄存器少，任何指令都可以访存
- RISC：简化硬件，优化常用操作！—两百条指令
  - 指令字长固定，格式规则，种类少，寻址方式简单
  - 减少访存，设置大量通用寄存器，运算基于寄存器
    - 为了提高性能，需要减少访存次数，因此寄存器寻址性能最高。
    - 采用load/store体系，只有load/store指令访存。
  - 采用Superscalar、Superpipeling等技术，提高IPC
- VLIW：空间换时间，低功耗，兼容性差

The CISC's eight principles:

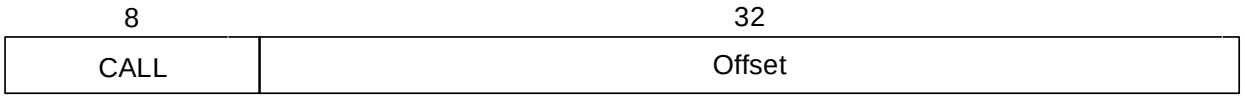
- Instructions are of variable format.
- There are multiple instructions and addressing modes.
- Complex instructions take many different cycles.
- Any instruction can reference memory.
- There is a single set of registers.
- No instructions are pipelined.
- A microprogram is executed for each native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

X86指令格式，图2-35

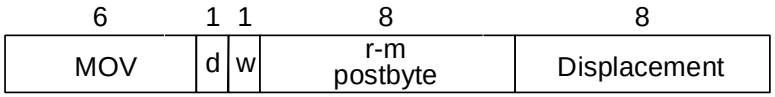
a. JE EIP + displacement



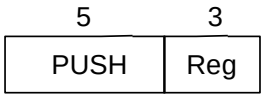
b. CALL



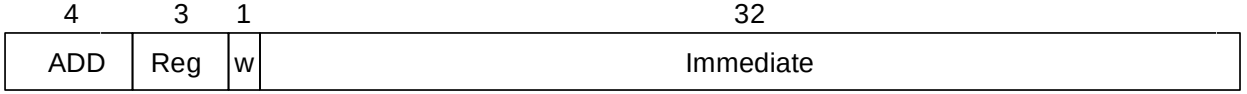
c. MOV EBX, [EDI + 45]



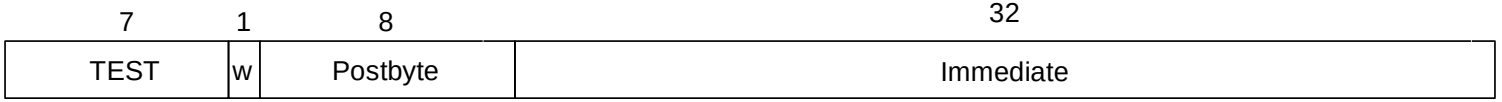
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



Growth of x86 instruction set over time

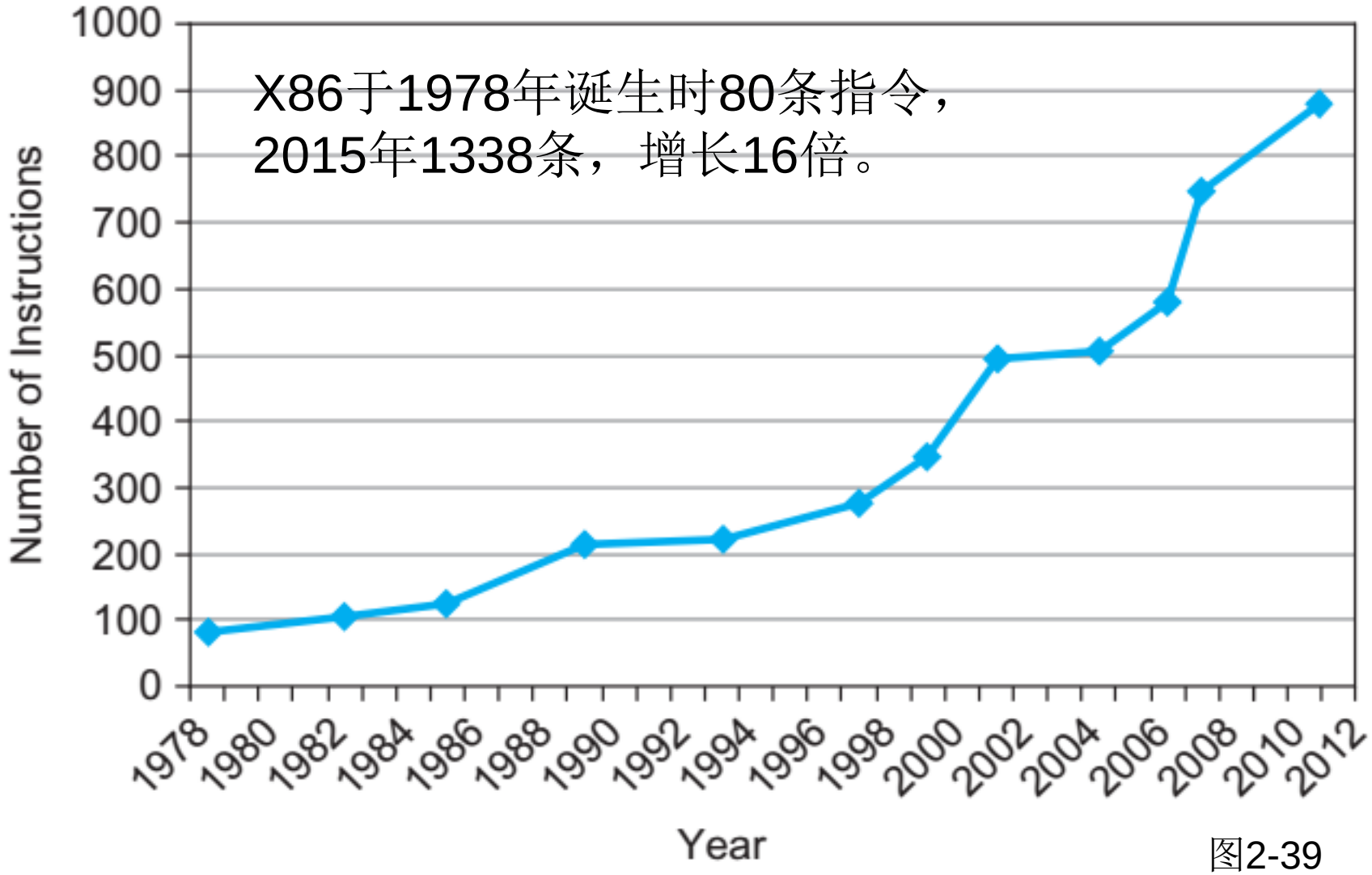


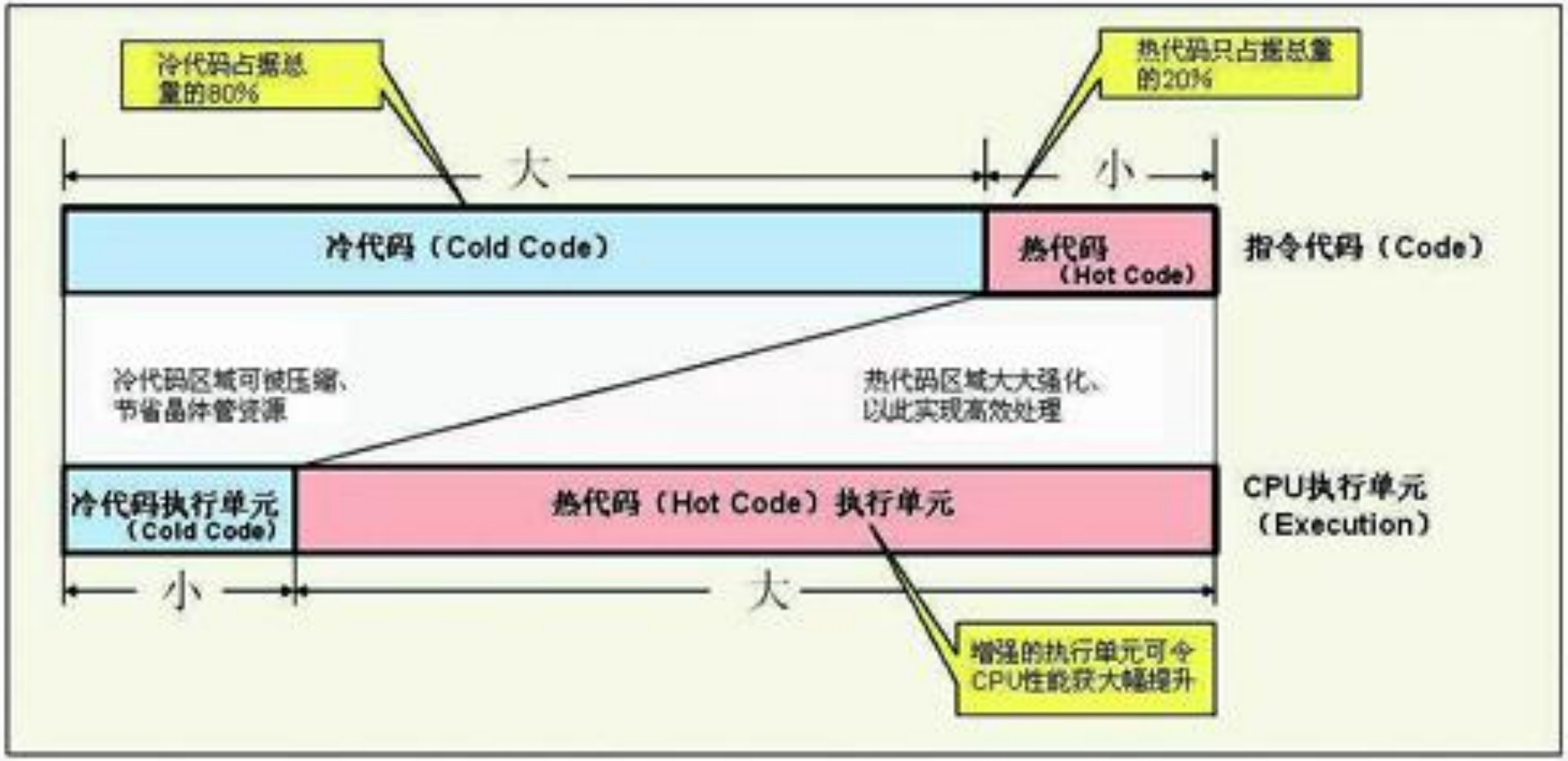
图2-39

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%

RISC的理论基础

计算机指令代码的80 / 20规律



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.

RV指令分布，图2-41

- SPEC CPU2006

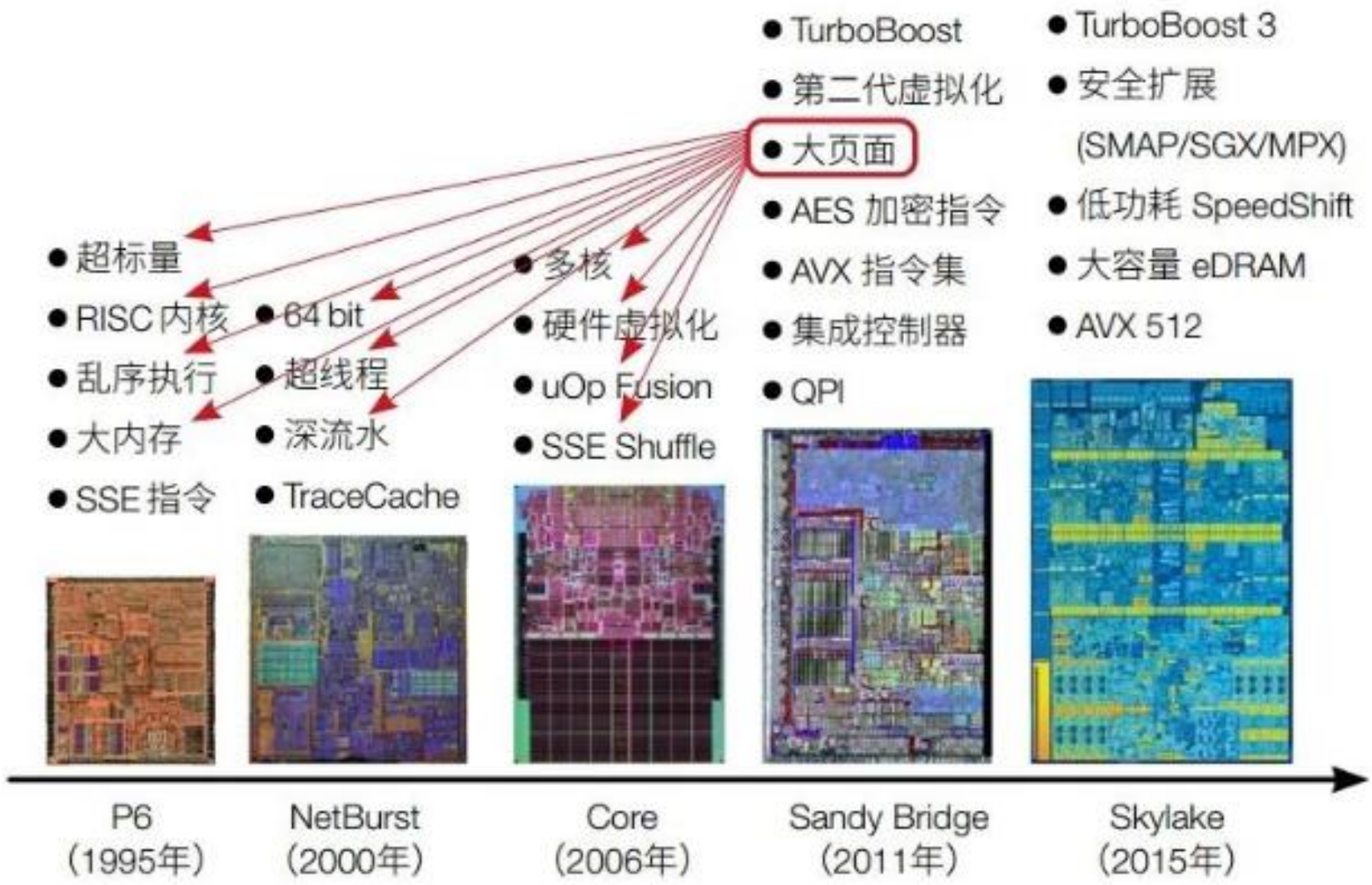
Instruction class	RISC-V examples	HLL correspondence	Frequency	
			Integer	Fl. Pt.
Arithmetic	add, sub, addi	Operations in assignment statements	16%	48%
Data transfer	ld, sd, lw, sw, lh, sh, lb, sb, lui	References to data structures in memory	35%	36%
Logical	and, or, xor, sll, srl, sra	Operations in 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%

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.

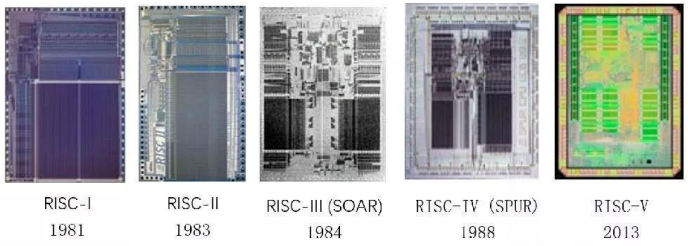
- Fixed-format instructions.
- Few instructions and addressing modes.
- Simple instructions taking one clock cycle.
- LOAD/STORE architecture to reference memory.
- Large multiple-register sets.
- Highly pipelined design.
- Instructions executed directly by hardware.
- Complexity handled by the compiler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

Intel处理器架构演化（1995—2015 年）

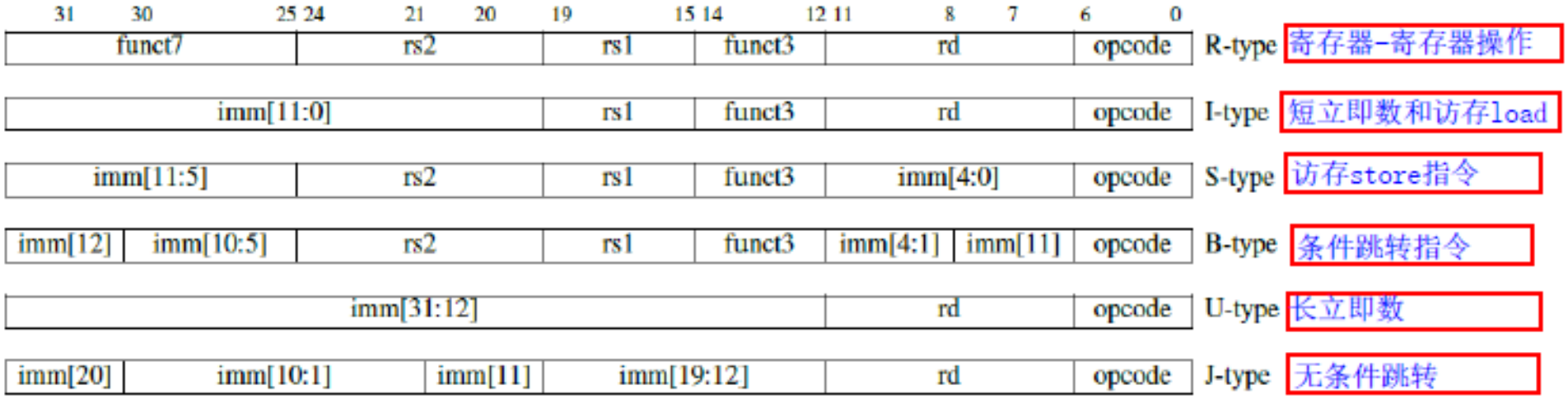


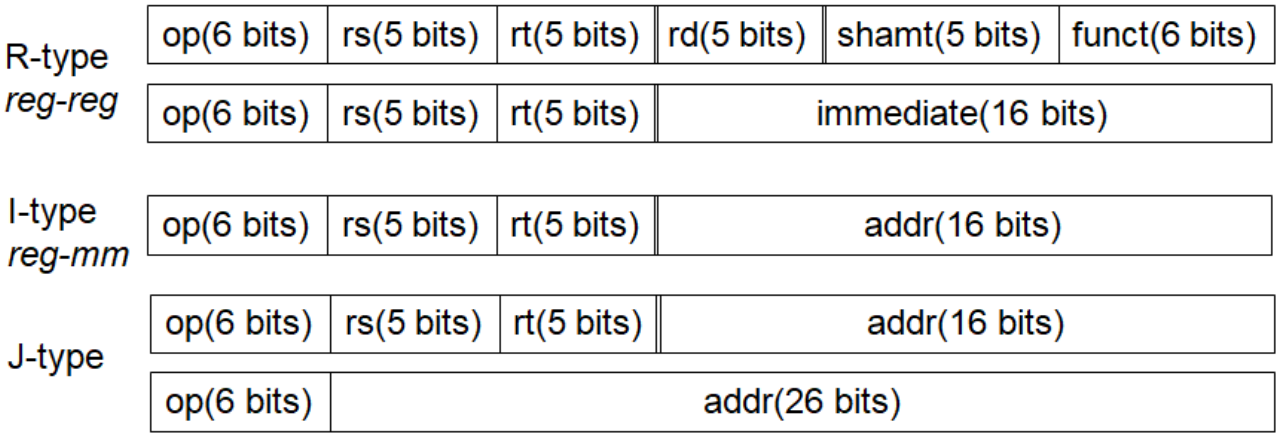
MIPS is simple, elegant.

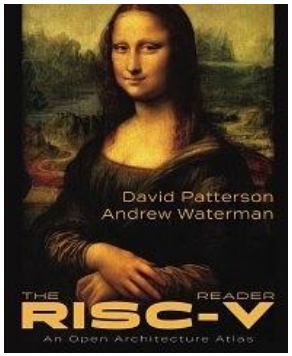
- 无互锁流水的微处理器
  - Microprocessor w/o Interlocked Piped Stages
  - interlock单元：数据依赖问题
    - 互锁：检测，推迟后续指令执行（互锁状态）
    - 无互锁：尽量利用软件办法避免
    - 无互锁困难，低效：R4000以后使用interlock
- Patterson提出RISC指令集，1980
  - 4个Design Principles
    - Simplicity favors regularity
    - Smaller is faster
    - *Make the common case fast*
    - Good design demands good compromises
- Hennessy完成MIPS，1983？
  - 第一个RISC处理器



RV vs. MIPS：指令格式，图2.29



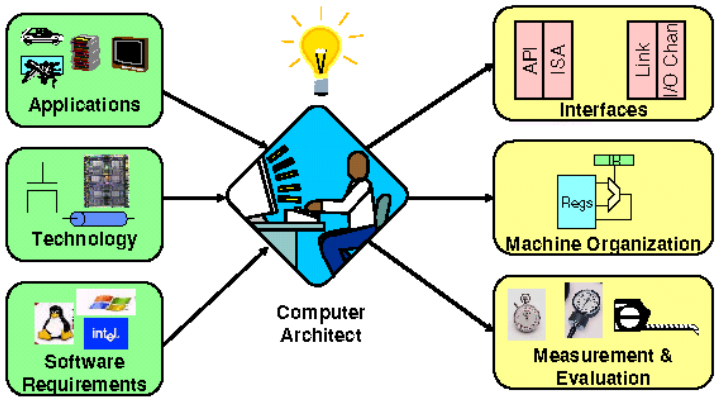




差别：1）操作码位数与位置；2）立即数位数与位置；2）rs/rd位置。

ISA: A Minimalist Perspective

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



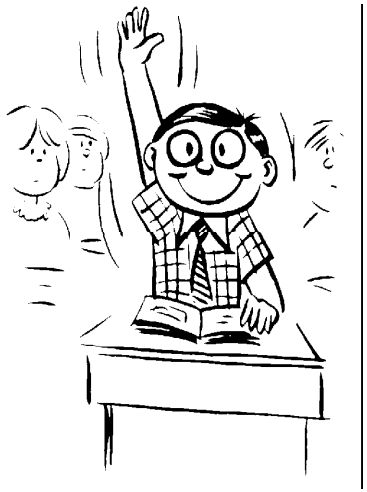
- “最小”计算机？——快速原型☺，ABC
  - A：由哪些部件构成？
  - B：需要哪几条指令？需要哪些寻址方式？执行过程？
  - C：

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*

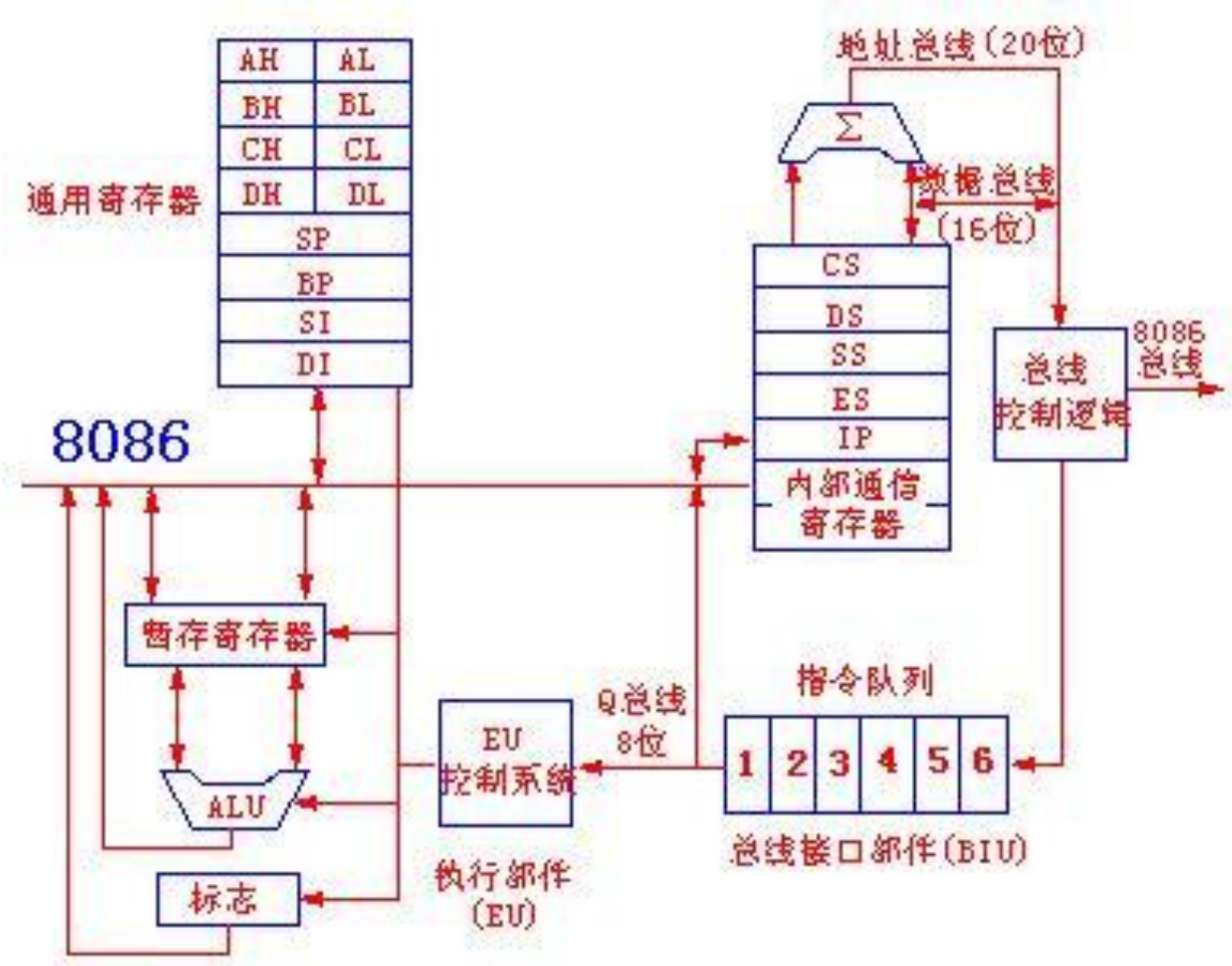
- 应用：嵌入式处理器
  - 硬件极其简单
    - 程序员有充分的控制权
    - 优化由编译器完成
  - 灵活
    - 其他“指令”都可由该指令构造
    - 意味着用户可自定义指令集
    - 意味着可适用于任何领域
  - 低功耗

小结



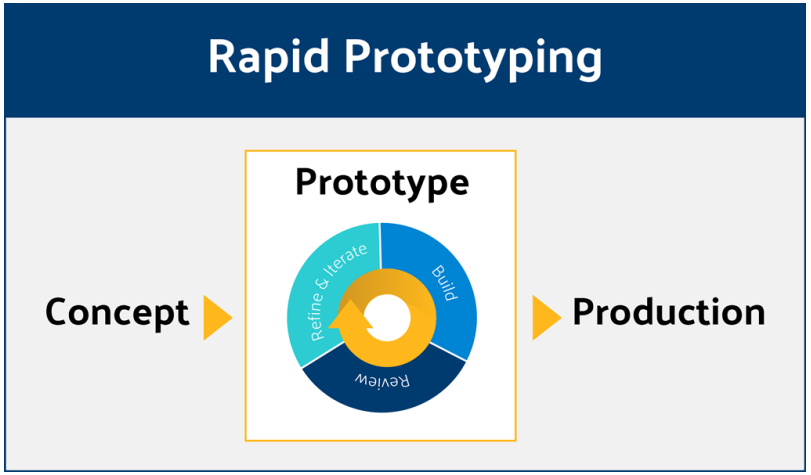
- 作业
  - 2.9，2.24，2.35，2.40
- 思考（选一）
  - CPU的ISA要定义哪些内容？见Yale Patt附录A
  - main()与swap()状态保存异同？
  - Windows系统中可执行程序的格式？
- 实验报告：2周
  - 基于RV汇编，设计一个冒泡排序程序，并用Ripes工具调试执行。
  - 可选：测量冒泡排序程序的执行时间。

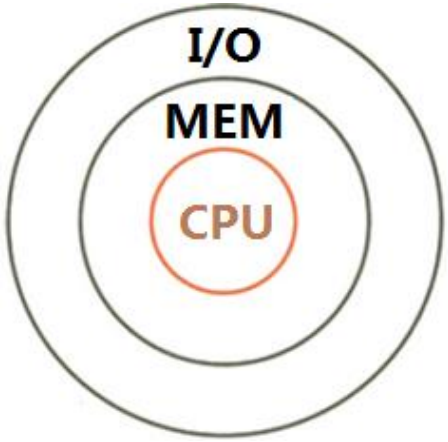
8086处理器逻辑结构



汇编语言计算机设计思路

- 快速原型
  - 概念、原型、产品
    - 构建、评估、求精
  - 冯机模型
    - 命令式，顺序执行
  - 自顶向下，自内而外
    - CPU + MEM
- Processor/CPU
  - ISA
    - 指令：原子操作
  - uArch
    - 数据通路，控制器





边界对准问题（Memory Alignment）

- 为了便于硬件实现，通常要求多字节的数据在存储器的存放方式能满足“边界对准”的要求。
- 字对齐：左移两位，按字访问

存储器

				地址（十进制）
字（地址0）				0
字（地址4）				4
字节（地址11）	字节（地址10）	字节（地址9）		8
字节（地址15）	字节（地址14）	字节（地址13		12
半字（地址18）		半		16
半字（地址22）		半		20
双字（地址24）				24
双字				28
双字（地址32）				32
双字				36

在对准边界的32位字长的计算机中，半字地址是2的整数倍，字地址是4的整数倍，双字地址是8的整数倍。当所存数据不能满足此要求时，可填充一个至多个空白字节。

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

例： Byte Ordering & Memory Alignment

大众点评 笔试中的一道题（也是全国考研题）：

计算机存储器按字节编址，采用小端方式存放数据。假定int型和short型长度分别为32位和16位，并且数据按边界对齐存储。某C语言程序段如下：

```
struct {
    int     a;
    char    b;
    short   c;
} record;
record.a=273;
```

若record首地址为0xC008，则地址0xC008中内容及record.c的地址是

A. 0x00、0xC00D

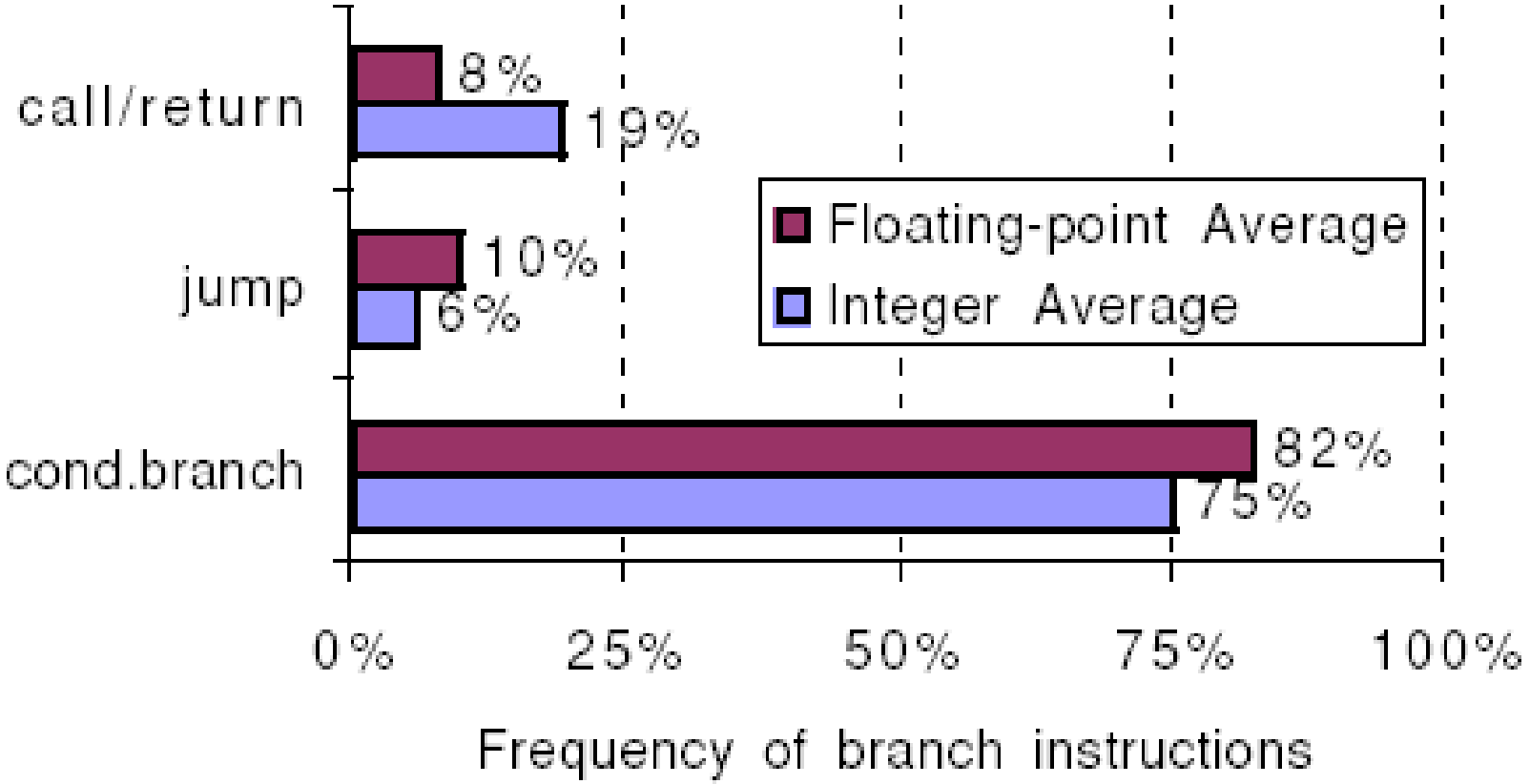
B . 0x11、0xC00E

C . 0x11、0xC00D

D . 0x00、0xC00E

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1+R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	`Add R1,–(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Control Instruction Distribution



Ex: beq $s4, $s5, Lab1

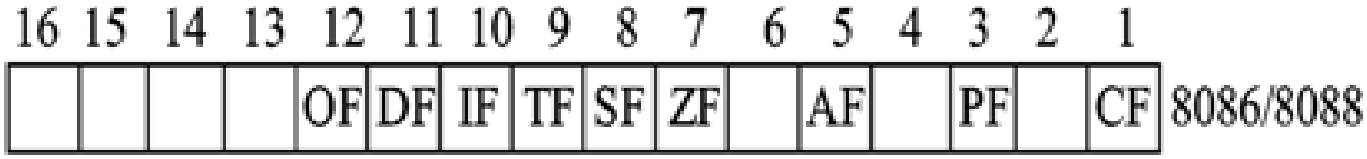
分支比较的实现方式（***Branch Conditions***）

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Special bits are set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

Condition: (Z, N, C, O)

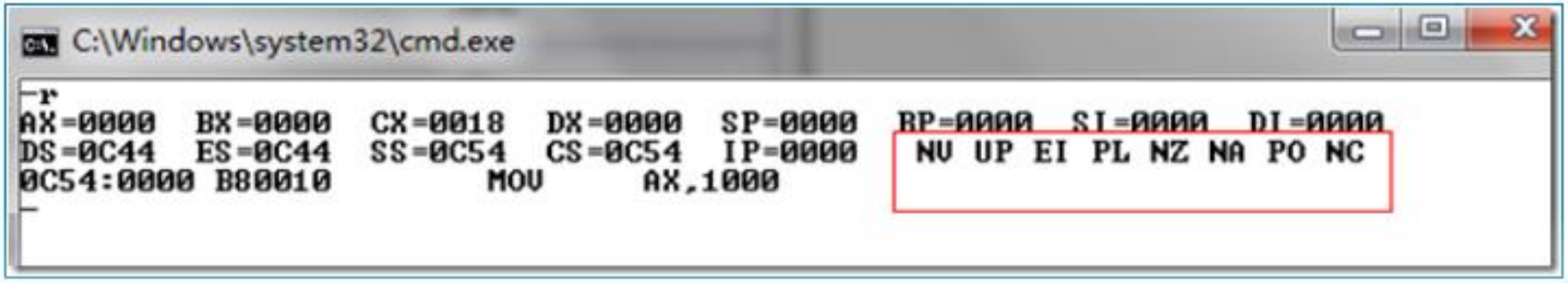
MIPS的beq？

8086标志寄存器：系统当前状态与控制



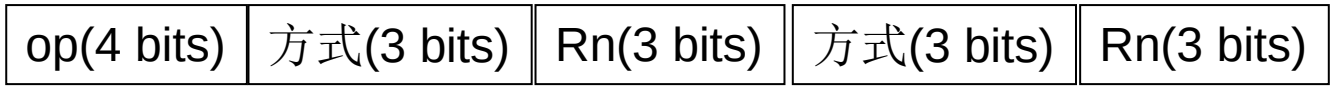
标志位
CF 进位（有/否）
PF 奇偶（偶/奇）
AF 半进位
ZF 全零（是/否）
SF 符号（负/正）
IF 中断（允许/禁止）
DF 方向（增量/减量）
OF 溢出（是/否）

标志位名称	值为 1 时的标记	值为 0 时的标记
OF	OV	NV
SF	NG	PL
ZF	ZR	NZ
PF	PE	PO
CF	CY	NC
DF	DN	UP



VAX11/780机器指令格式与编码

- 16位机，有16种寻址方式，共303条指令
  - 16个寄存器，指令字不定长（1～54bytes）
- 例：



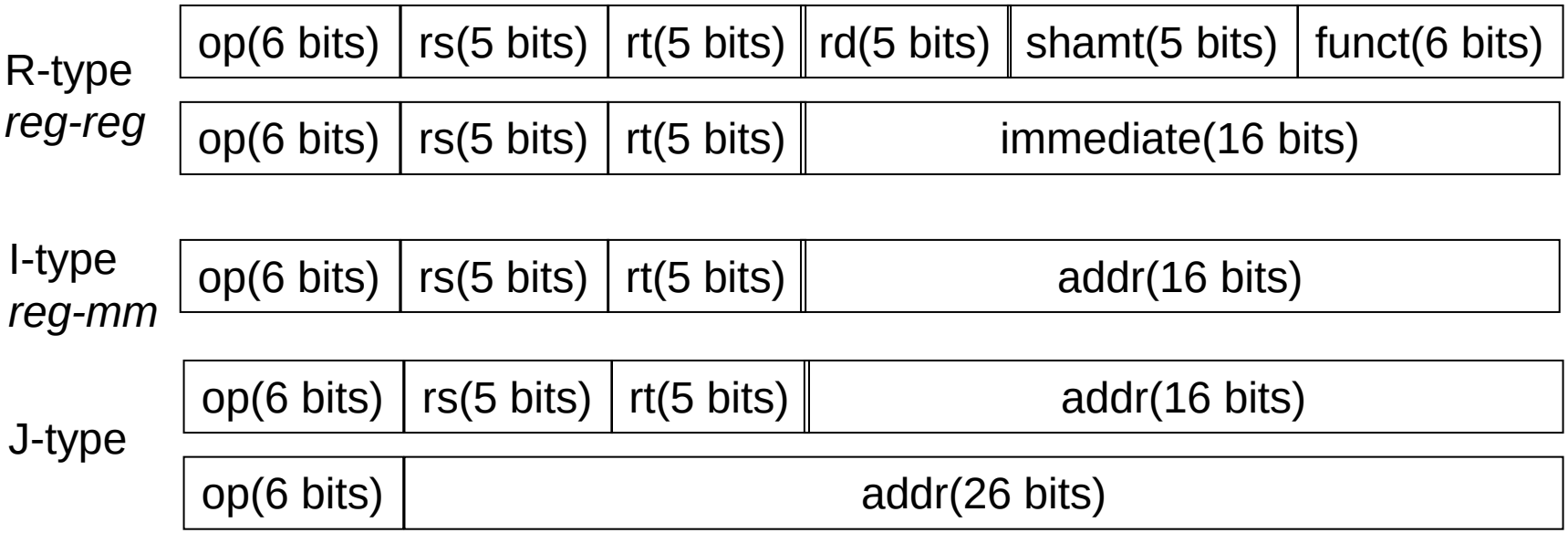
  - Add R2, R4; R2+R4 -> R4, “06 02 04” (8进制)
  - Add [R2], R1; [R2] + R1 -> R1, 06 12 01
  - Add #1000, R1; 1000+R1 -> R1

寄存器寻址

寄存器间接寻址

MIPS指令格式

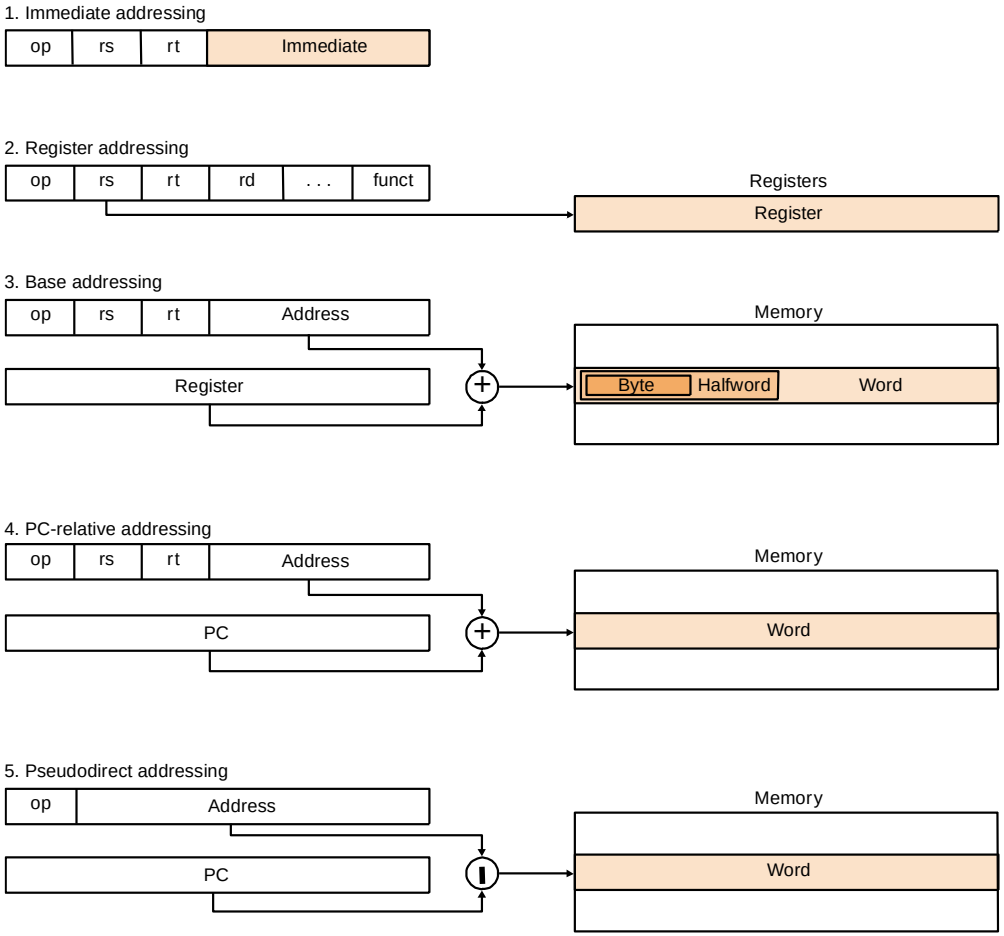
- 100余条指令（COD中33条），共32个通用寄存器
- 指令格式：定长32位
  - R-type：arithmetic instruction
  - I-type：data transfer
  - J-type：branch instruction(conditional & unconditional)
- MIPS：Simple（<=regularity）= Elegant！！！



MIPS寻址模式(3+2)

- 立即寻址：I-type
- 寄存器寻址：R-type
- 基址寻址：I-type

- PC相对寻址：J-type
- 伪直接寻址：J-type
  - pseudo-direct addressing
  - 26位形式地址左移2位，与PC的高4位拼接



MIPS 指令示例

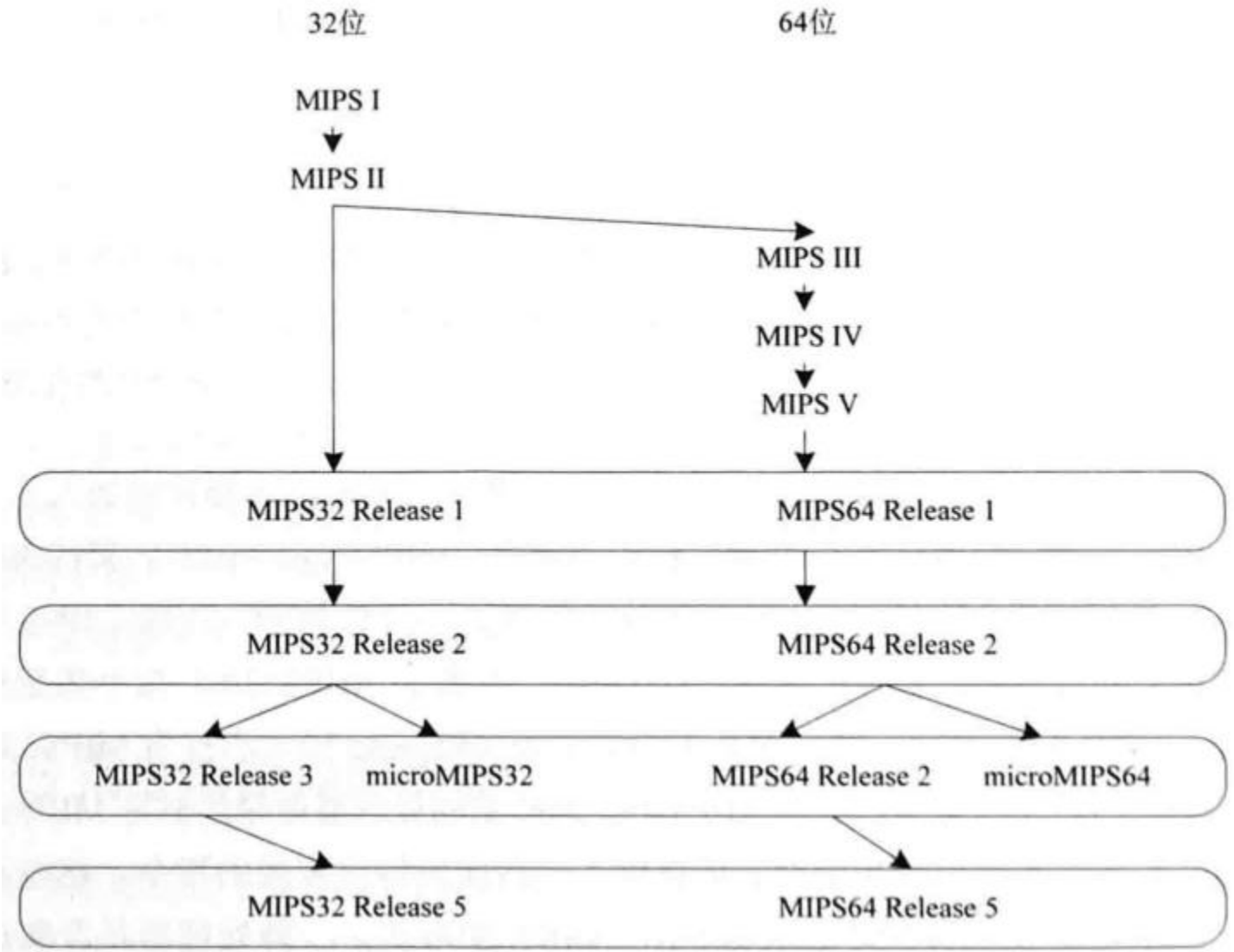
```
add $t0,$s1,$s2
```

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

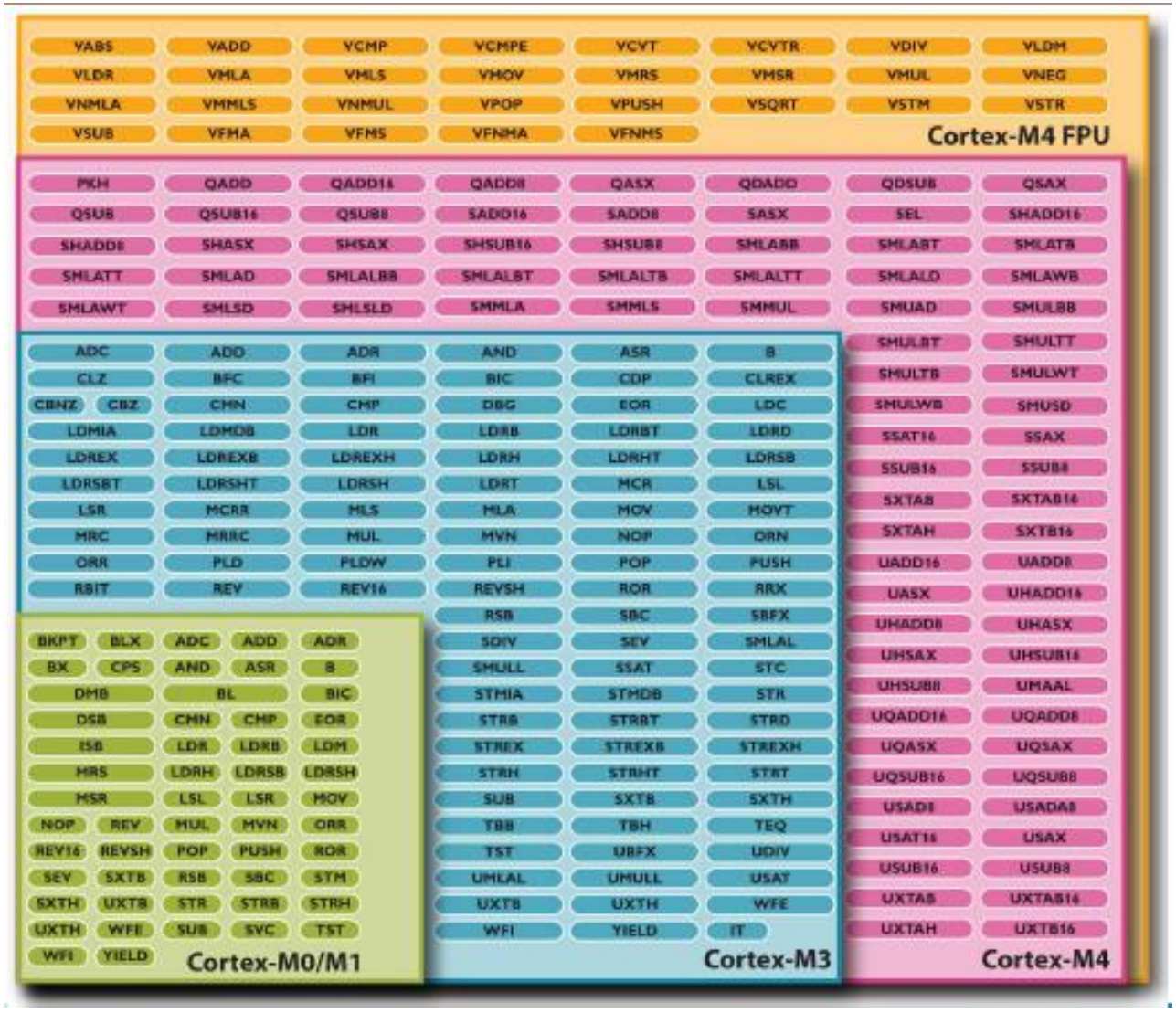
```
sll  $t2,$s0,4   # reg $t2 = reg $s0 << 4 bits
```

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

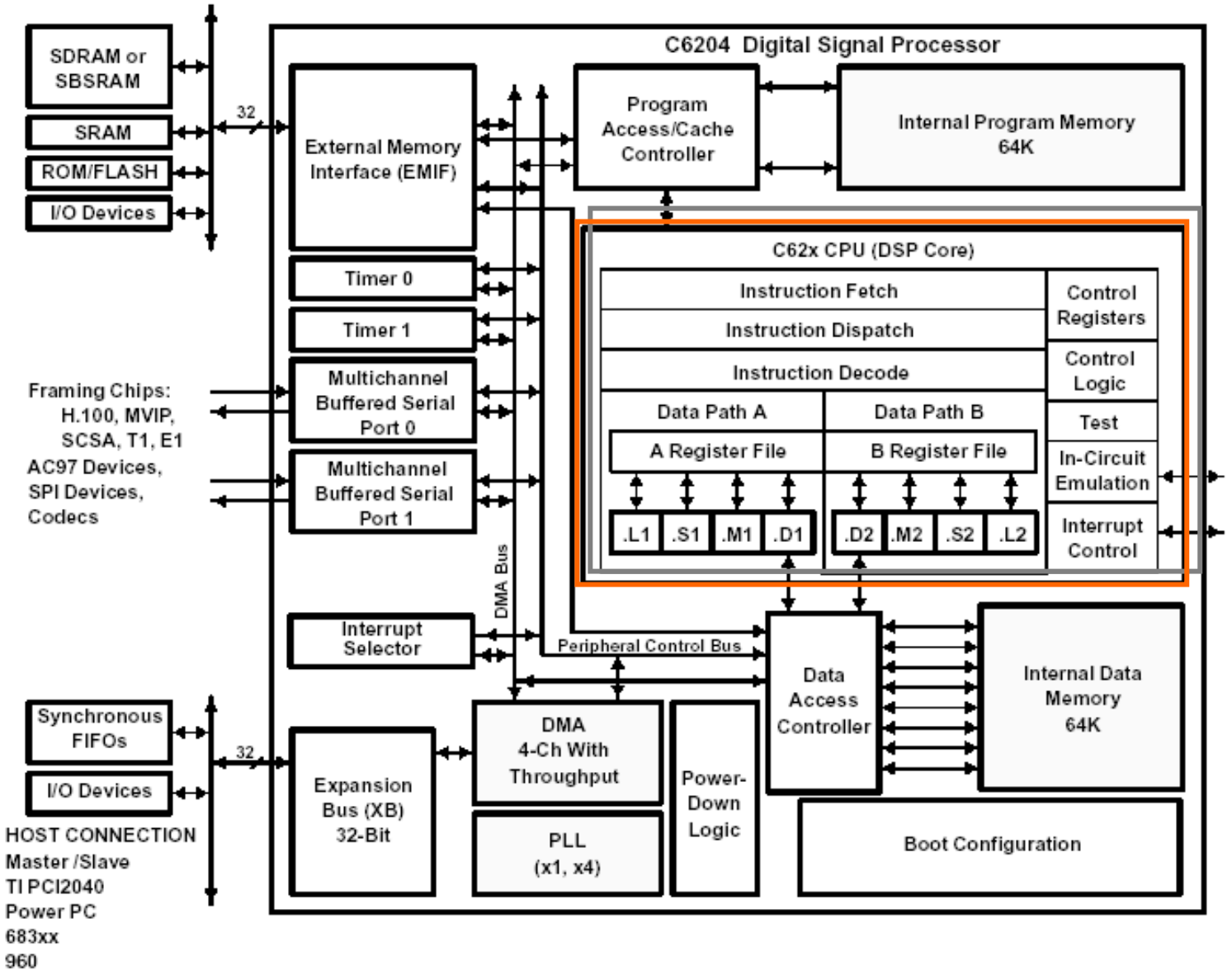
MIPS ISA的演变


32位
64位
MIPS I
MIPS II
MIPS III
MIPS IV
MIPS V
MIPS32 Release 1
MIPS64 Release 1
MIPS32 Release 2
MIPS64 Release 2
MIPS32 Release 3
microMIPS32
MIPS64 Release 2
microMIPS64
MIPS32 Release 5
MIPS64 Release 5

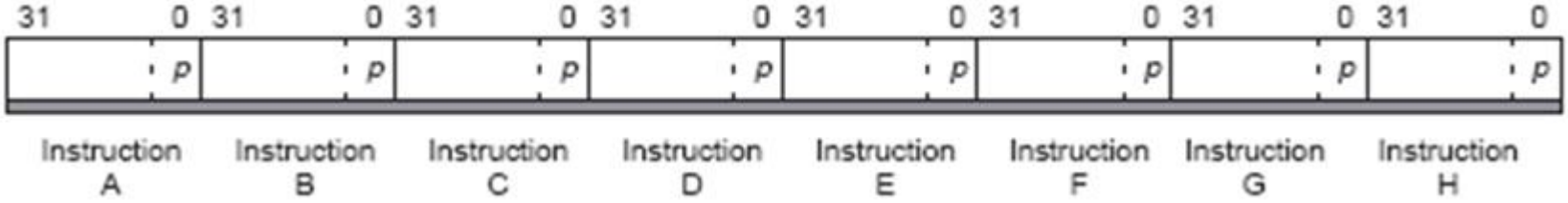
Cortex-M0/M1/M3/M4指令集



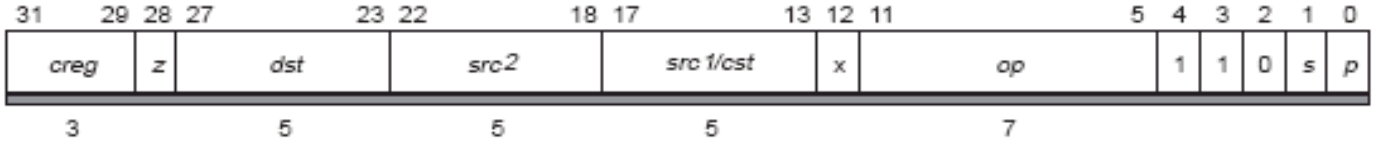
TMS320C64x：VLIW



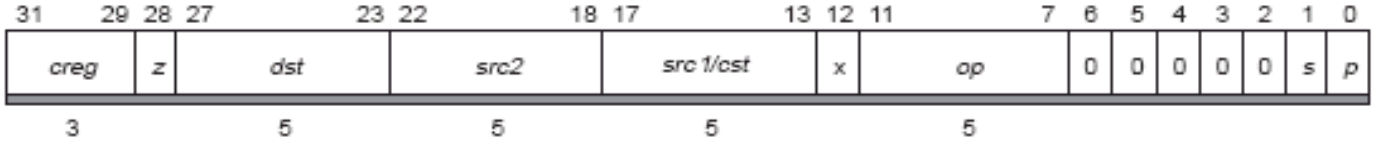
TMS320C64x指令字



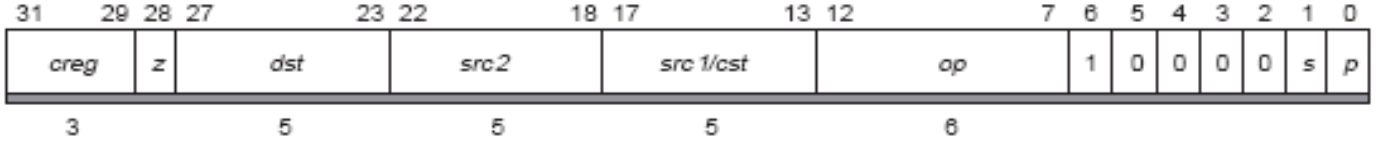
Operations on the .L unit



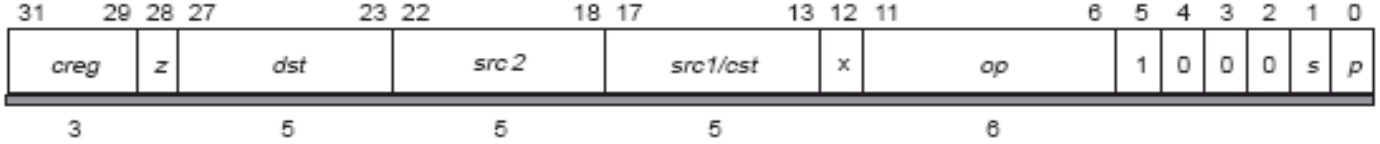
Operations on the .M unit



Operations on the .D unit



Operations on the .S unit



ISA Classes (uArch prospective)

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

$$C = A + B$$

Compact Code & Stack Architecture

- 60's，编译技术发挥寄存器的效率很困难，因此某些设计者完全放弃寄存器，而采用堆栈执行模型。
- 基于堆栈的操作可以有效的缩短指令字长，可以减少存储和传输的开销
  - JAVA虚拟机即采用堆栈模型
  - FORTH，仪器控制语言
- *Stack在哪儿？*

Accumulator Architectures

- 早期硬件太昂贵，所以使用的寄存器很少。
  - 实际上只有一个用于算术指令的寄存器——被称为“Accumulator”，所以这种结构被称为“Accumulator Architectures”
    - 如EDSAC（第一台存储程序计算机，1949）。
- 采用“memory-based operand-addressing mode”。
  - 进化：采用一些专用寄存器，如数组指针、堆栈指针、乘除运算寄存器等
    - 如x86，称为“extended accumulator arch”

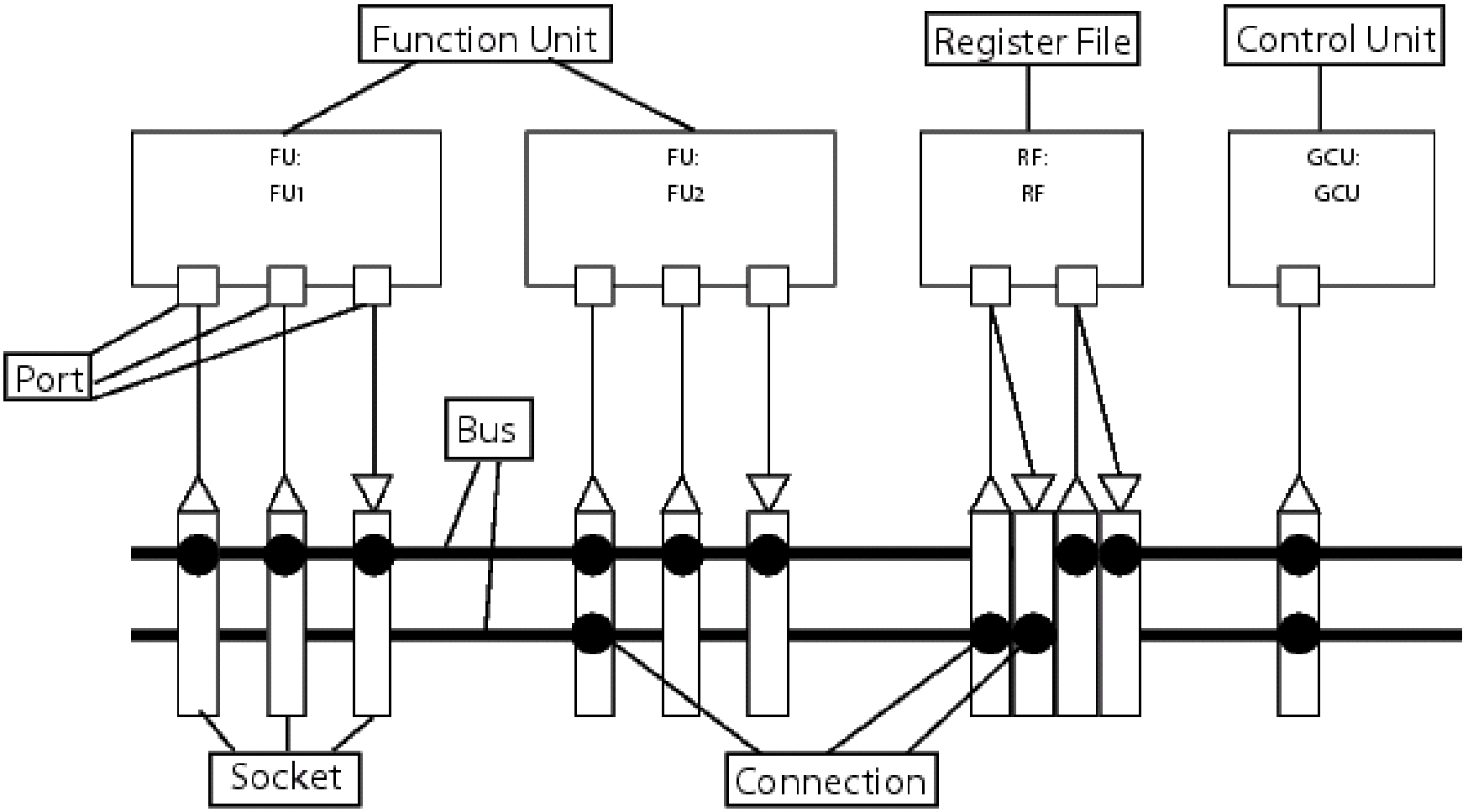
General-Purpose Register Architecture

- 通用：所有寄存器可以用于任何目的
- 两类
  - Register-memory Arch
    - 允许一个操作数在内存中
    - x86：Register-memory
  - Load-store or register-register architecture
    - 要求所有操作数都在寄存器中
    - MIPS：Load-store

Transport triggered architecture

- originally called a "move machine”
  - 程序直接控制处理器内部的传输总线
- uses only the **MOVE** instruction
  - 计算：由memory-mapped ALU完成
    - 向功能单元的触发端口写数据将触发功能单元完成计算
      - 故曰transport triggered
  - 跳转：由memory-mapped PC完成
  - 适于实现ASIP
- 上市产品：MAXQ微控制器
  - 适合数模混合应用场景

TTA的典型结构——冯机？

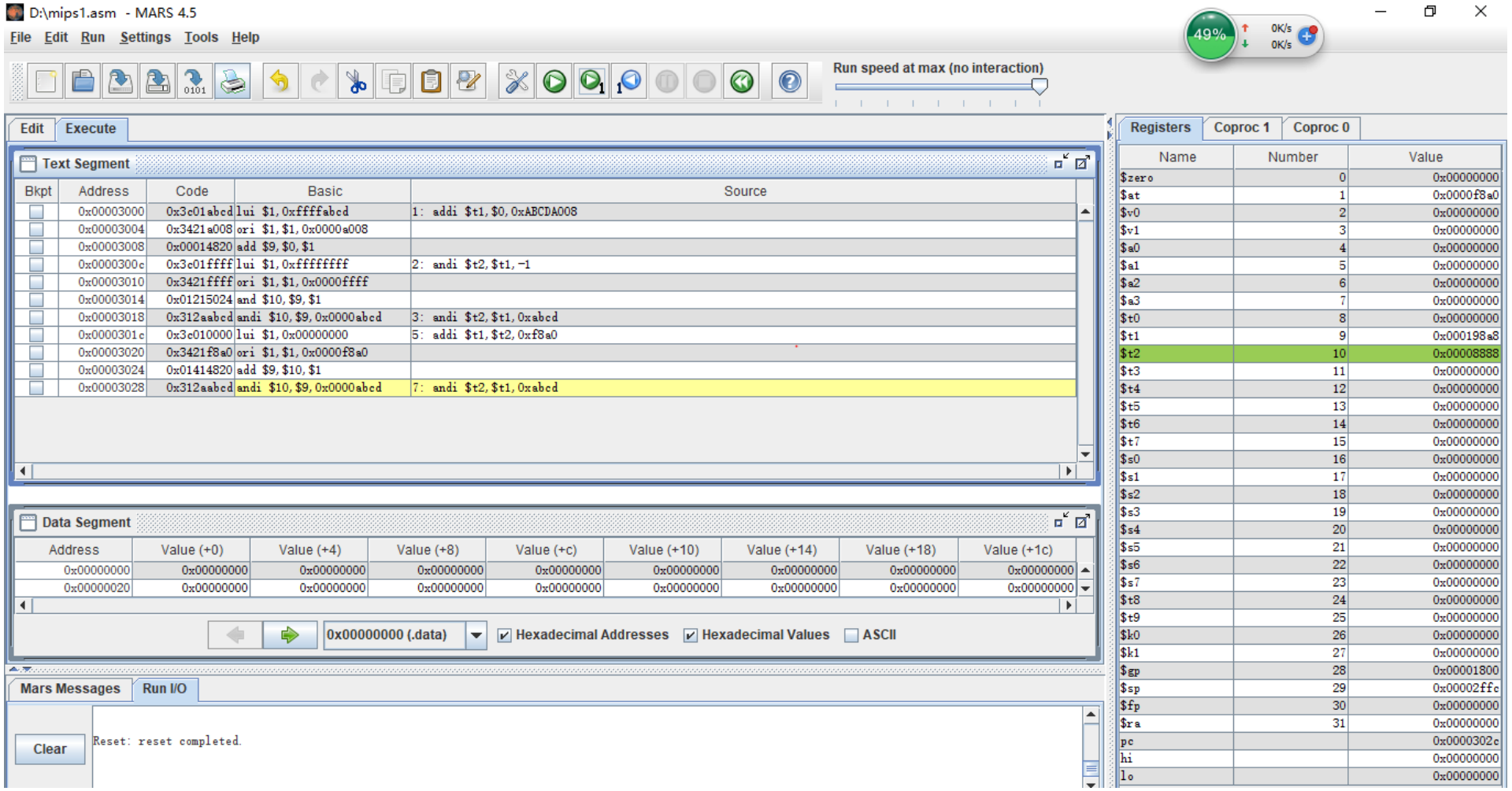

Function Unit
Register File
Control Unit
FU:
FU1
FU:
FU2
RF:
RF
GCU:
GCU
Port
Bus
Socket
Connection

SPIM的系统调用：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

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

MARS模拟器


D:\mips1.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
Text Segment
Bkpt Address Code Basic Source
0x00003000 0x3c01abcd lui $1, 0xffffabcd 1: addi $t1, $0, 0xABCDA008
0x00003004 0x3421a008 ori $1, $1, 0x0000a008
0x00003008 0x00014820 add $9, $0, $1
0x0000300c 0x3c01ffff lui $1, 0xffffffff 2: andi $t2, $t1, -1
0x00003010 0x3421ffff ori $1, $1, 0x0000ffff
0x00003014 0x01215024 and $10, $9, $1
0x00003018 0x312aabcd andi $10, $9, 0x0000abcd 3: andi $t2, $t1, 0xabcd
0x0000301c 0x3c010000 lui $1, 0x00000000 5: addi $t1, $t2, 0xf8a0
0x00003020 0x3421f8a0 ori $1, $1, 0x0000f8a0
0x00003024 0x01414820 add $9, $10, $1
0x00003028 0x312aabcd andi $10, $9, 0x0000abcd 7: andi $t2, $t1, 0xabcd
Data Segment
Address Value (+0) Value (+4) Value (+8) Value (+c) Value (+10) Value (+14) Value (+18) Value (+1c)
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000020 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 (.data)
Hexadecimal Addresses
Hexadecimal Values
ASCII
Mars Messages Run I/O
Clear
Reset: reset completed.
Registers Coproc 1 Coproc 0
Name Number Value
$zero 0 0x00000000
$at 1 0x0000f8a0
$v0 2 0x00000000
$v1 3 0x00000000
$a0 4 0x00000000
$a1 5 0x00000000
$a2 6 0x00000000
$a3 7 0x00000000
$t0 8 0x00000000
$t1 9 0x000198a8
$t2 10 0x00008888
$t3 11 0x00000000
$t4 12 0x00000000
$t5 13 0x00000000
$t6 14 0x00000000
$t7 15 0x00000000
$s0 16 0x00000000
$s1 17 0x00000000
$s2 18 0x00000000
$s3 19 0x00000000
$s4 20 0x00000000
$s5 21 0x00000000
$s6 22 0x00000000
$s7 23 0x00000000
$t8 24 0x00000000
$t9 25 0x00000000
$k0 26 0x00000000
$k1 27 0x00000000
$gp 28 0x00001800
$sp 29 0x00002ffc
$fp 30 0x00000000
$ra 31 0x00000000
pc 0x0000302c
hi 0x00000000
lo 0x00000000

MIPS汇编指令

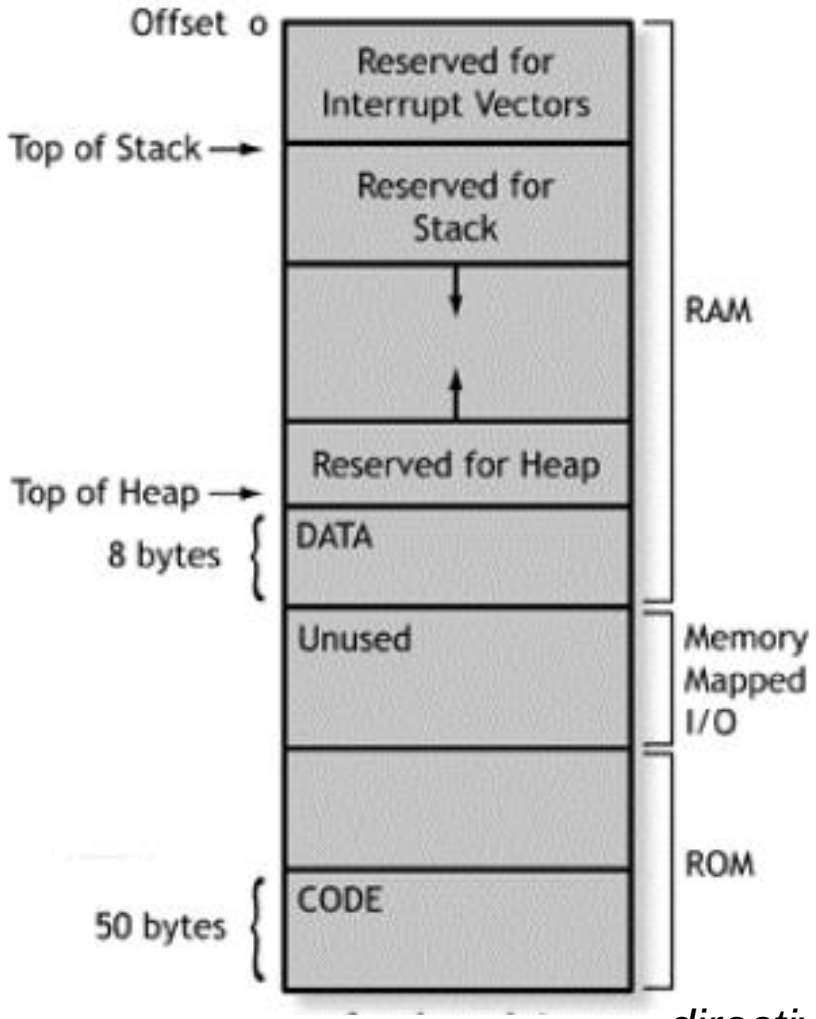
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Policy of Use Conventions for registers

REGISTER	NAME	USAGE
$0	$zero	常量0(constant value 0)
$1	$at	保留给汇编器(Reserved for assembler)
$2-$3	$v0-$v1	函数调用返回值(values for results and expression evaluation)
$4-$7	$a0-$a3	函数调用参数(arguments)
$8-$15	$t0-$t7	暂时的(或随便用的)
$16-$23	$s0-$s7	保存的(或如果用，需要SAVE/RESTORE的)(saved)
$24-$25	$t8-$t9	暂时的(或随便用的)
$28	$gp	全局指针(Global Pointer)
$29	$sp	堆栈指针(Stack Pointer)
$30	$fp	帧指针(Frame Pointer)
$31	$ra	返回地址(return address)

- $26..$27：$k0..$k1，为OS存放EPC

程序结构



	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

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

#1

First, design your system so that the code is measurable!

Measure execution time as part of your standard testing.
Do not only test the functionality of the code!

Learn both coarse-grain and fine-grain techniques to measure execution time.

Use coarse-grain measurements for analyzing real-time properties

Use fine-grain measurements for optimizing and fine-tuning

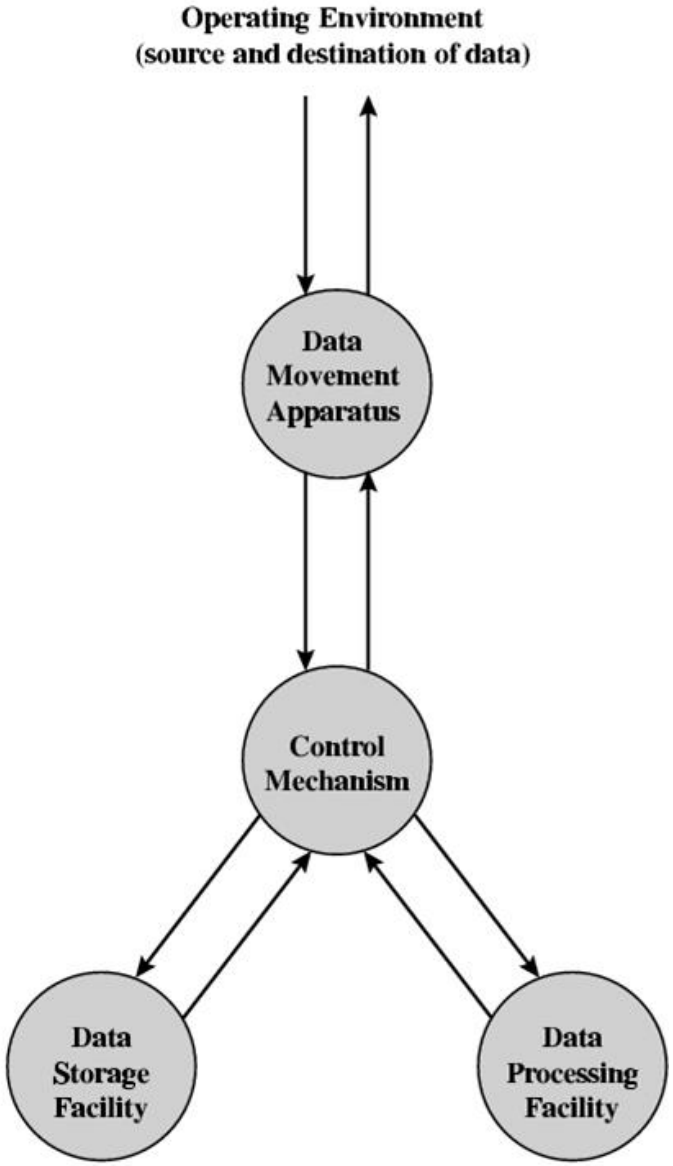
No measurements of execution time!

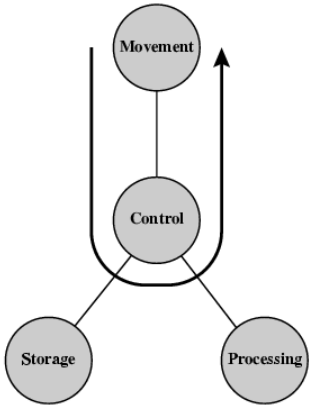
WCET：程序执行时间

- 执行代码所花费的处理器时间
  - 取决于程序的结构、目标处理器、执行环境
    - 分支路径、循环控制参数、数据存储位置
    - ILP 、Cache、中断
  - 不考虑上下文切换

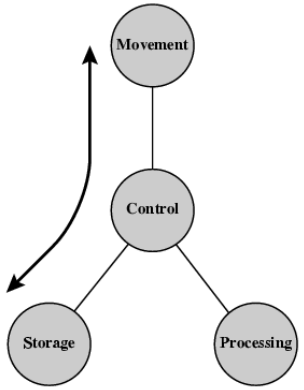
- 见CSAPP第九章“测量程序执行时间”

计算机基本行为：计算、访存、I/O

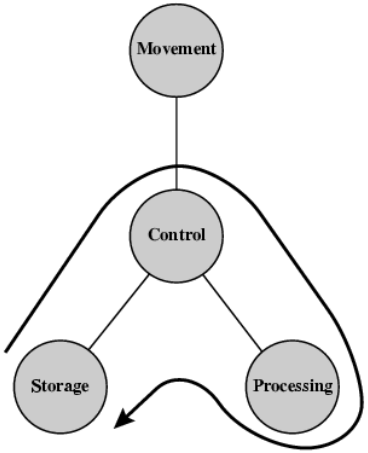




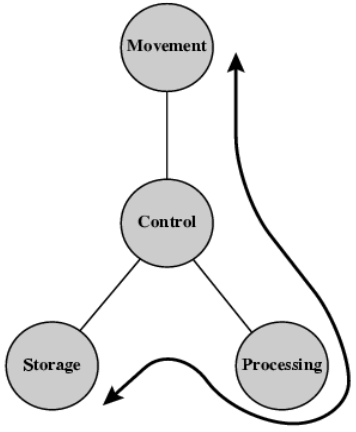
Data Movement



Data Storage



Processing from/to storage



Processing from Storage to I/O