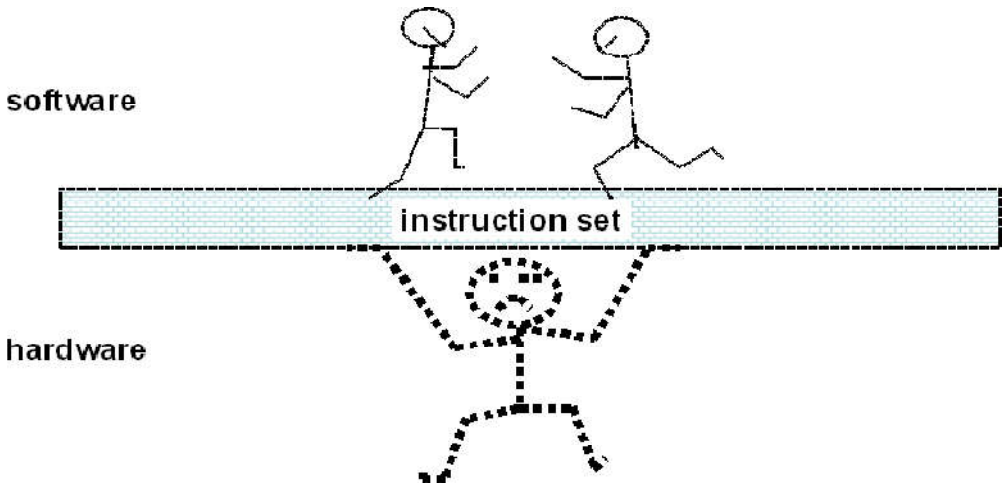
计算机组成原理
第二章“指令系统”

中科大11系

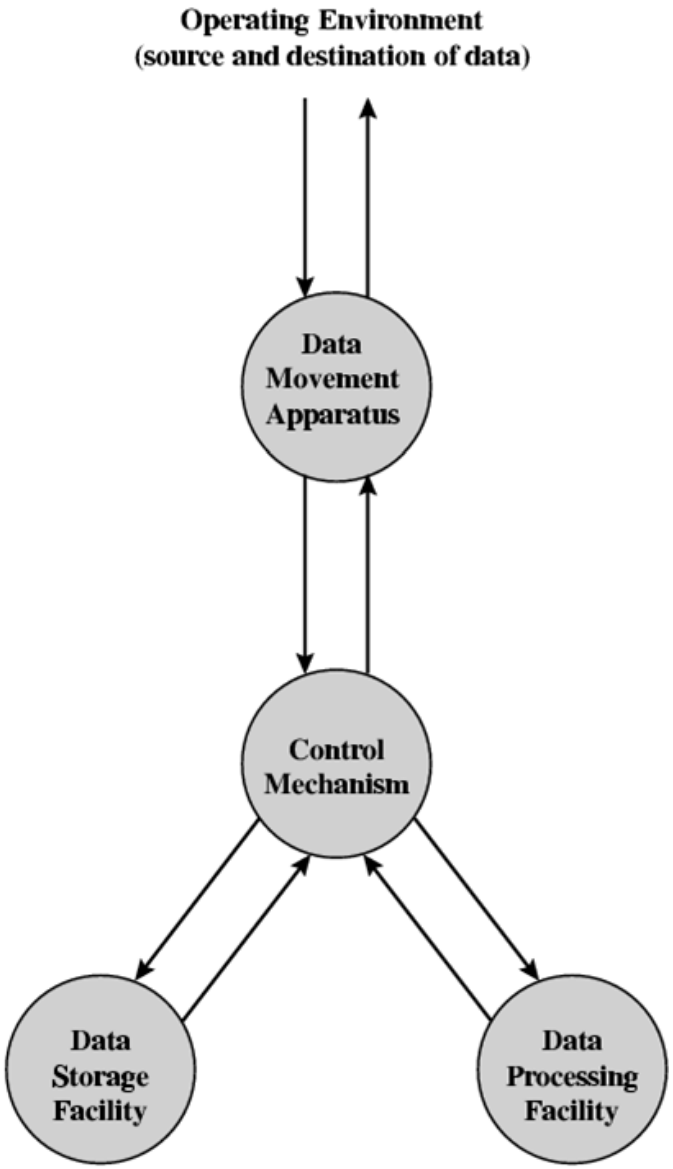
李曦

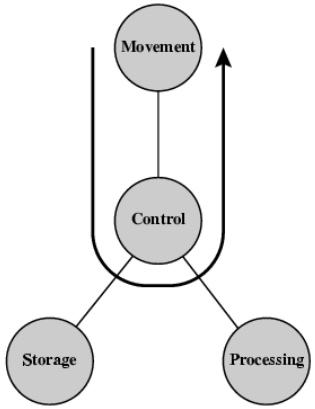
概要

- 指令系统：机器指令的集合
  - 汇编语言（Assemble Language）/机器语言
  - Instruction Set Architecture（ISA）
    - CISC、RISC、VLIW
    - 处理器、C编译器、OS
- 本章的内容
  - 指令功能
  - 指令格式
  - 寻址方式
  - 指令系统特征
  - 汇编程序设计
    - X86/MIPS（见下一节）

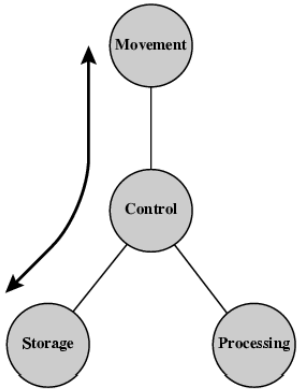


计算机基本功能（行为）

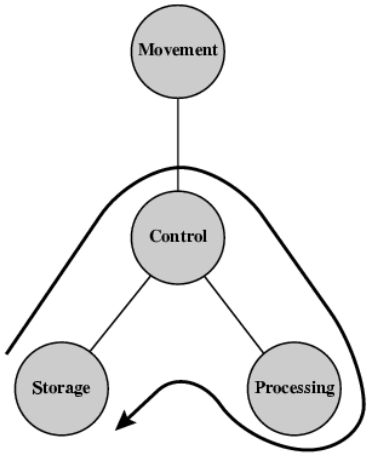




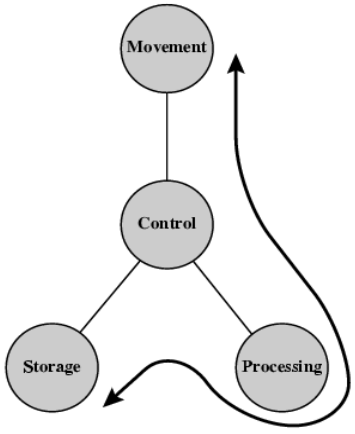
Data Movement



Data Storage



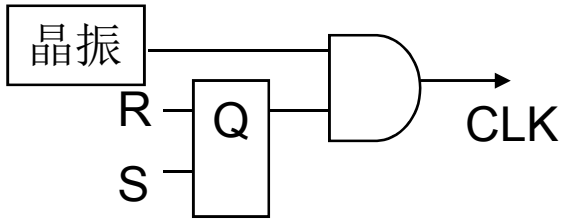
Processing
from/to storage



Processing
from Storage to I/O

操作分类

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



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)

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
  - 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（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

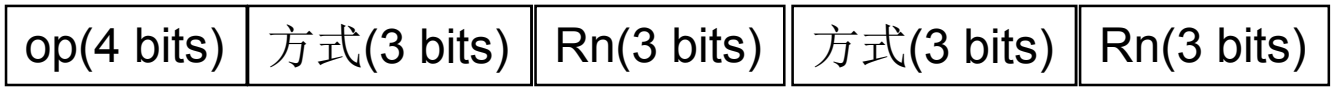
- 指定源操作数、目的操作数、下一条指令地址
  - 地址：主存、寄存器、I/O端口
- 地址码格式
  - 4地址指令：op rs1, rs2, rd, ni
  - 3地址指令：op rs1, rs2, rd；　ni在PC中
  - 2地址指令：op rs1, rs2；　　rd=rs1 or ACC
  - 1地址指令：op rs2；　　rs1=ACC，rd=ACC
  - 0地址指令：op；　　堆栈操作

操作数（opr）

- 含
  - 地址：无符号整数，计算offset等
  - 数据：定点数（有符号/无符号）、浮点数、逻辑值
  - 字符：ASCII、汉字内码
- 数据存储形式
  - 机器字长＝通用寄存器位数 = or ≠ 数据总线宽度
    - 32位机：字节、半字、字、双字
  - 字存储顺序
    - 小尾端（small endness）：低地址，低字节
    - 大尾端（big endness）：低地址，高字节
  - 边界对准（Memory Alignment、data alignment）
    - 数据从偶地址开始存放，空字节填充（data structure padding）

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：

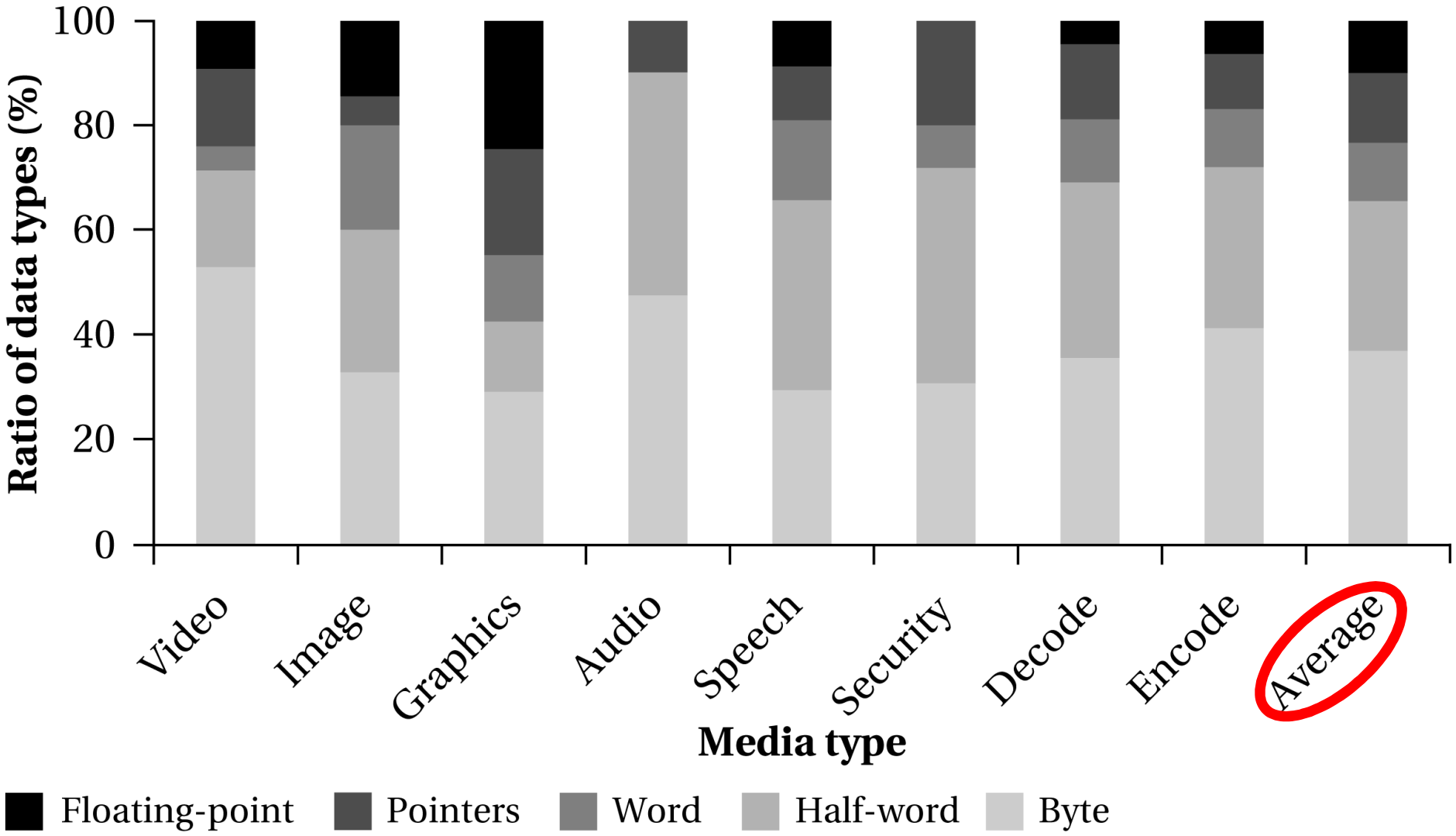


  - Add R2, R4; R2+R4 -> R4, “06 02 04” (8进制)
  - Add [R2], R1; [R2] + R1 -> R1, 06 12 01
  - Add #1000, R1; 1000+R1 -> R1

寄存器寻址

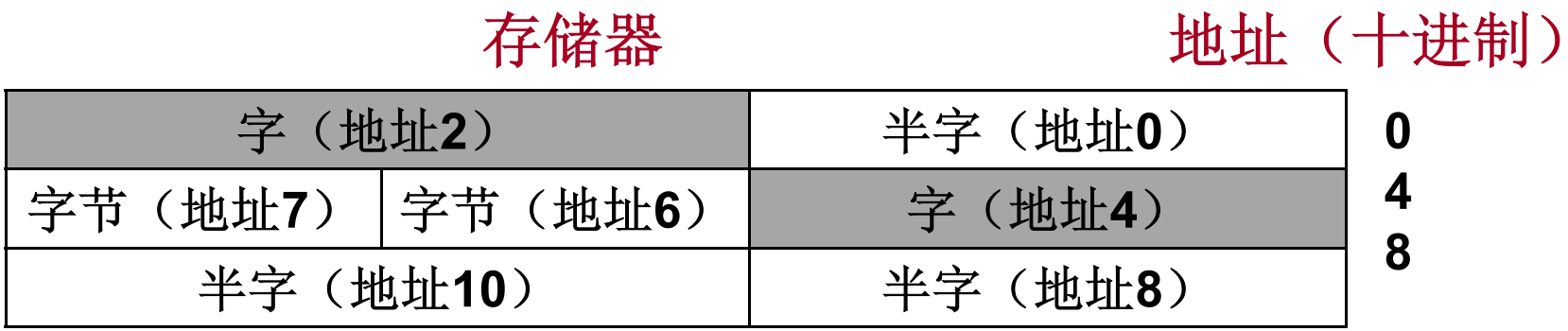
寄存器间接寻址

Operand characteristics in MediaBench (Fritts)



数据的存放方式

- 在数据不对准边界的计算机中，数据(例如一个字)可能在两个存储单元中。
- 此时需要访问两次存储器，并对高低字节的位置进行调整后，才能取得一字。



边界对准问题

- 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。

寻址方式

- 寻址方式：确定指令和操作数的存储地址的方式
- 指令寻址：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：操作数在内存中的物理地址
    - EA＝寻址方式＋形式地址

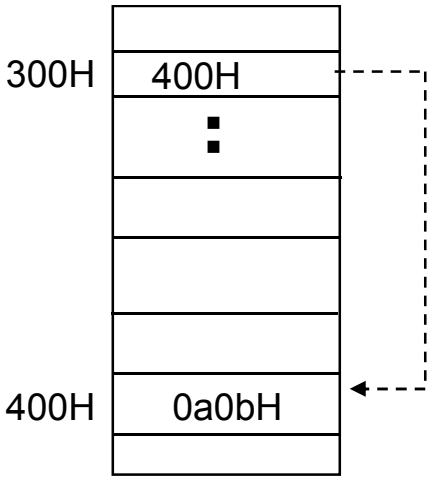
操作数寻址方式

- 常见约10种
  - 立即寻址
  - 直接寻址
  - 隐含寻址
  - 间接寻址
  - 寄存器寻址
  - 寄存器间接寻址
  - 基址寻址
  - 变址寻址
  - 相对寻址
  - 堆栈寻址

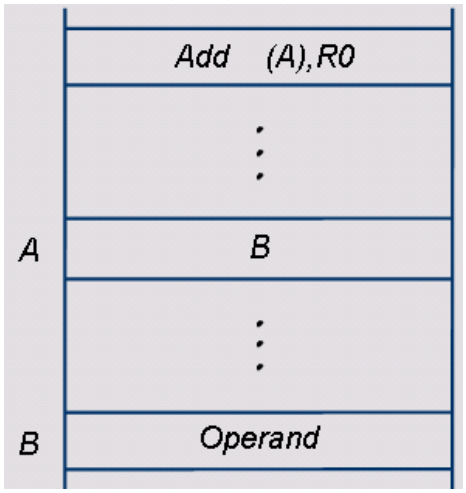
寻址方式（addressing mode）

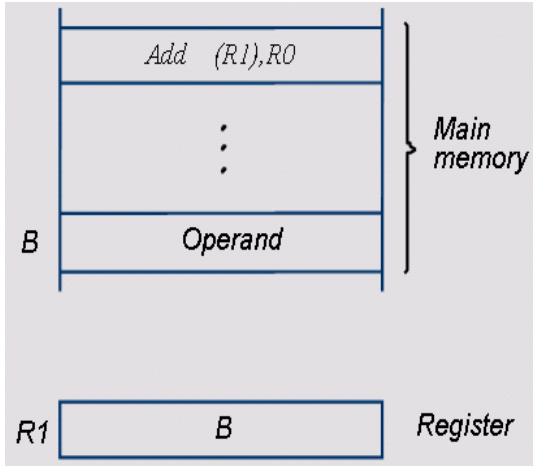
- 立即寻址(Immediate addressing)
  - 地址域中即为操作数
  - 表示为：op ＃xxxx
  - 立即数的范围与指令字长有关
  - 存储于内存中的指令段
- 直接寻址(direct addressing)
  - 有效地址＝形式地址
  - 形式地址位数确定寻址范围
  - op xxxx
- 隐含寻址(Implicitly)
  - 操作数在缺省的寄存器或ACC中

寻址方式（con't）



- 间接寻址（间址寻址， Indirect ）
  - 形式地址中给出有效地址的存储位置
  - 表示为：op [xxxx]
  - 寻址范围比**直接寻址**大
  - 可以“多次间址”
- 寄存器寻址(Register Direct)
  - 形式地址为某寄存器名（编号），该寄存器中存放操作数
  - op %r
- 寄存器间接寻址(Register Indirect)
  - 寄存器中为操作数的地址
  - 表示为：op [%r]





llxx@ustc.edu.cn

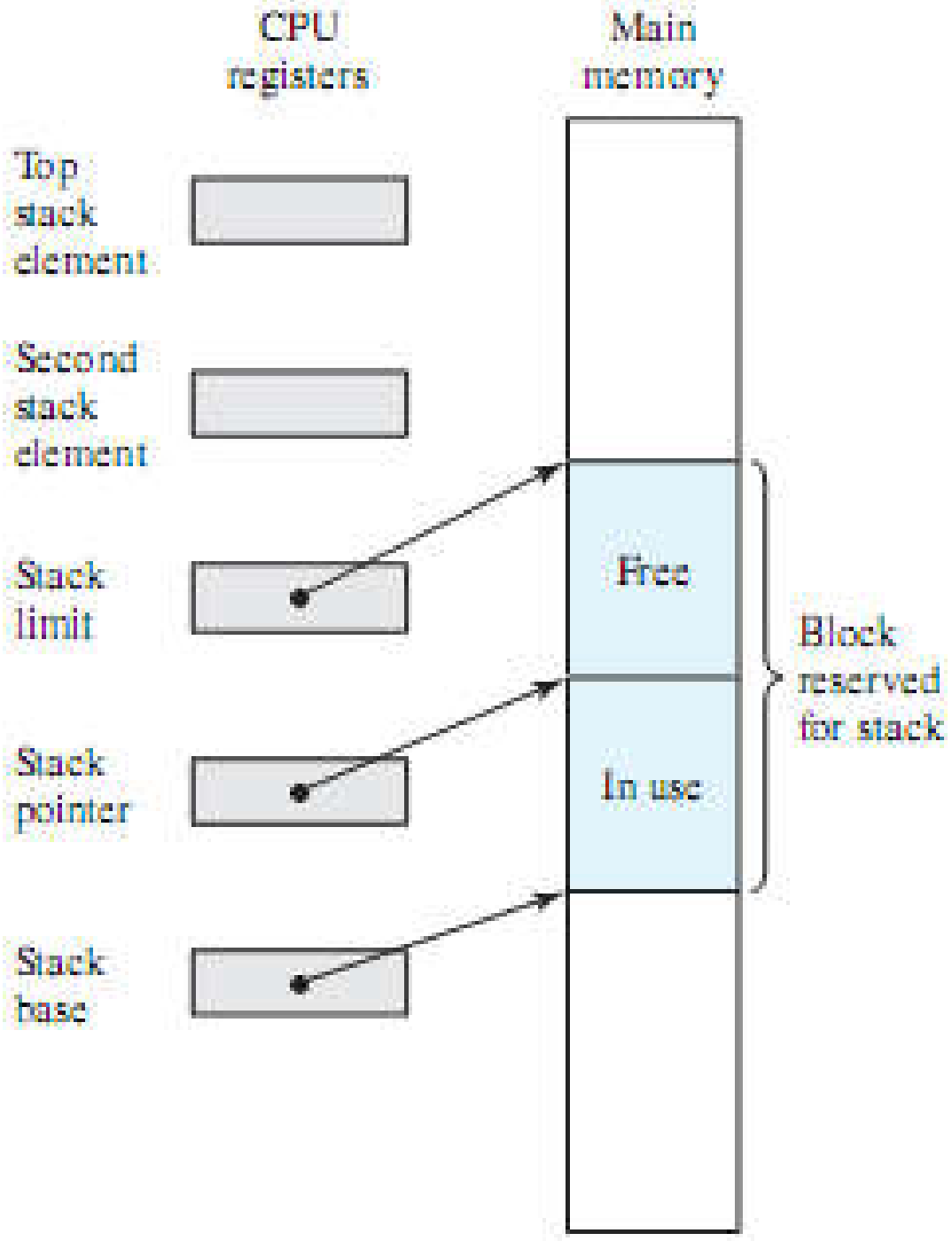
寻址方式（con't）

- 基址寻址(Base or Displacement addressing)
  - op xxxx[BR]
  - 以基址寄存器（BR）为基准进行寻址
  - EA＝形式地址（Disp）＋BR
  - 基址寄存器：专用、通用
    - 显式（通用寄存器）、隐式（专用寄存器）
- 变址寻址（index）
  - op xxxx[IR]
  - 以变址寄存器（ IR ）的值为基准寻址
- 基址寻址 vs. 变址寻址
  - 基址：BR由OS赋值，不变；形式地址可变
  - 变址：IR由程序员赋值，可变；形式地址不变
  - 用途不同：基址——段寻址，变址——数组指针

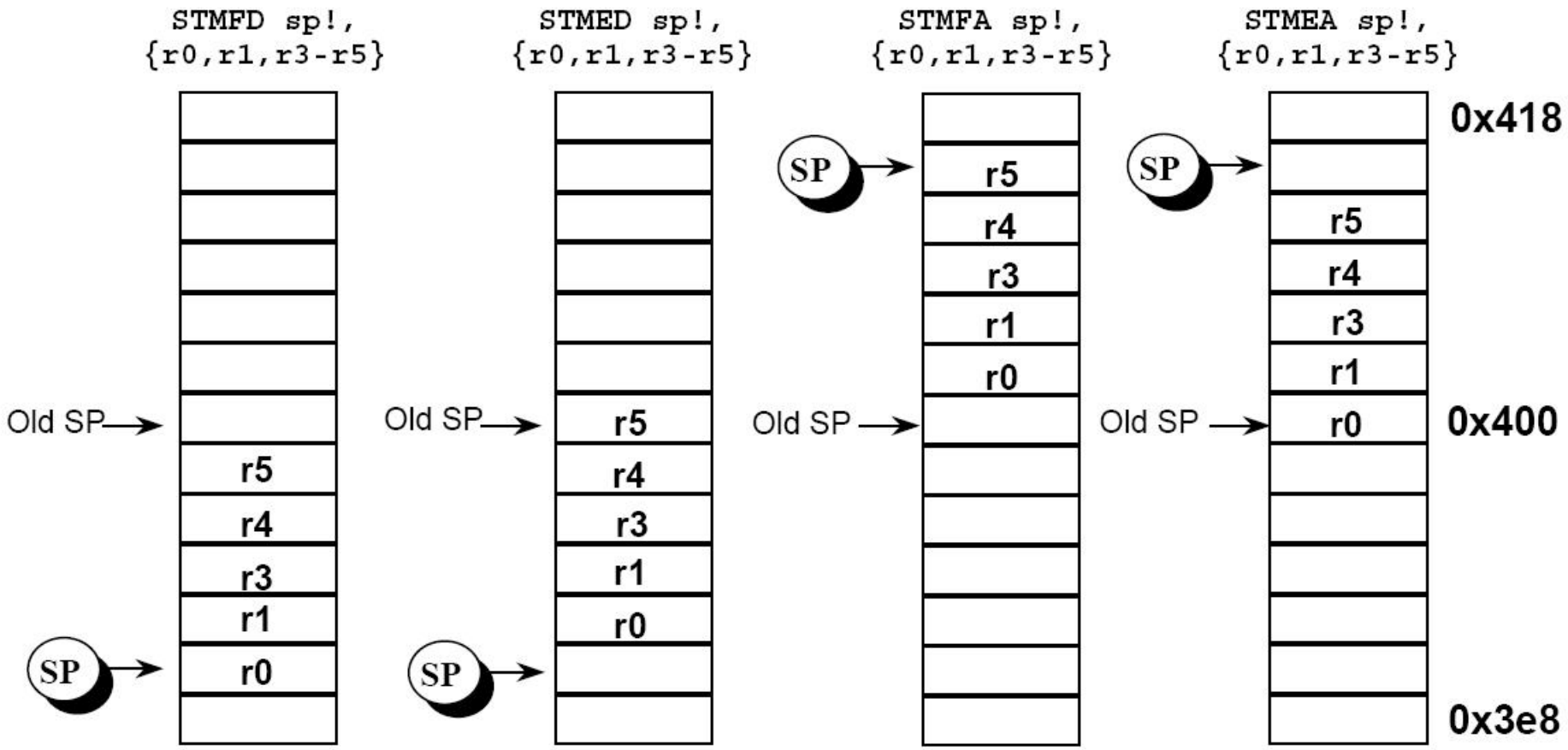
寻址方式（con't）

- 相对寻址（PC-relative addressing）
  - EA＝PC＋形式地址（disp）
  - 用于转移指令的目标计算
    - 位移量（Disp）：用补码，可正可负
- 堆栈寻址：专用于堆栈操作指令
  - 堆栈：可以为寄存器堆（Register File）或内存中的区域，由SP指示栈顶位置
  - 是一种隐含寻址

stack


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

栈操作图示


STMFD sp!,
{r0,r1,r3-r5}
STMED sp!,
{r0,r1,r3-r5}
STMFA sp!,
{r0,r1,r3-r5}
STMEA sp!,
{r0,r1,r3-r5}
Old SP
SP
r5
r4
r3
r1
r0
0x418
0x400
0x3e8

ISA分类

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

ISA分类（程序员角度）

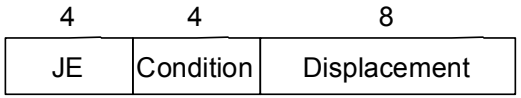
- CISC：硬件换性能！
  - 以机器指令实现高级语言功能
  - 指令字长不一（x86从1byte～6bytes）
  - 寻址方式多
  - ALU指令可以访存
- RISC：简化硬件，做好高频度的事！
  - 设置大量通用寄存器，运算基于寄存器
    - 为了提高性能，需要减少访存次数，因此寄存器寻址性能最高。
    - 采用load/store体系，只有load/store指令访存。
  - 指令字长固定，格式规则，种类少，寻址方式简单
  - 采用Superscalar、Superpipeling等技术，提高IPC
- VLIW：硬件简单，低功耗，兼容性差

The CISC's eight principles:

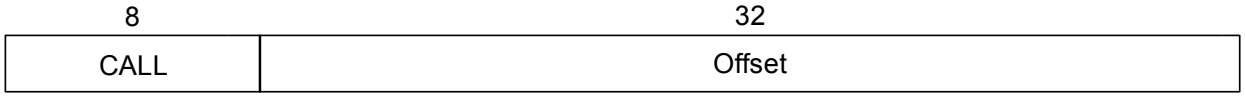
- Instructions are of variable format.
- There are multiple instructions and addressing modes.
- Complex instructions take many different cycles.
- Any instruction can reference memory.
- There is a single set of registers.
- No instructions are pipelined.
- A microprogram is executed for each native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

X86指令格式

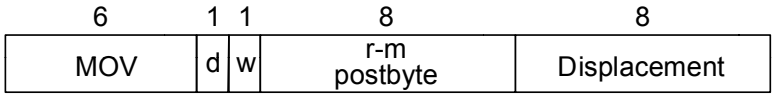
a. JE EIP + displacement



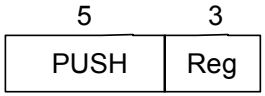
b. CALL



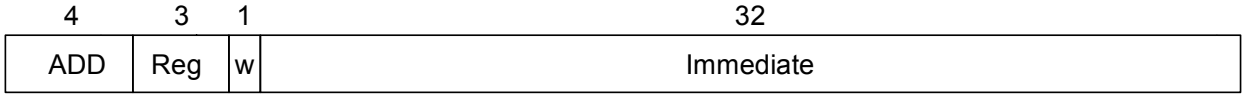
c. MOV EBX, [EDI + 45]



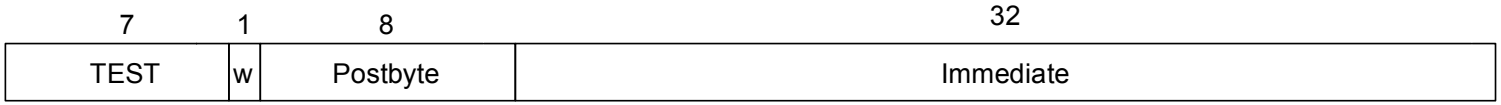
d. PUSH ESI



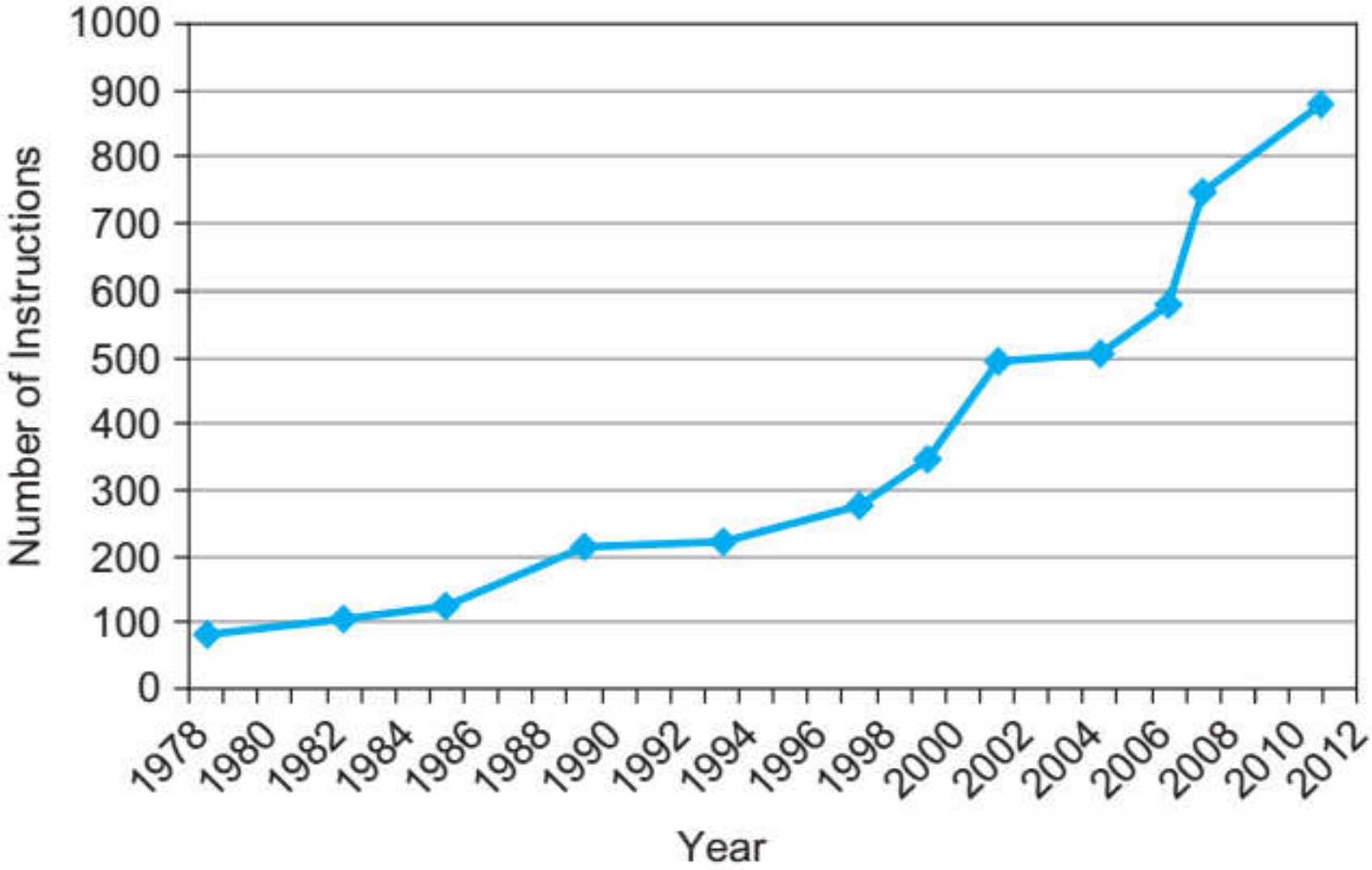
e. ADD EAX, #6765



f. TEST EDX, #42



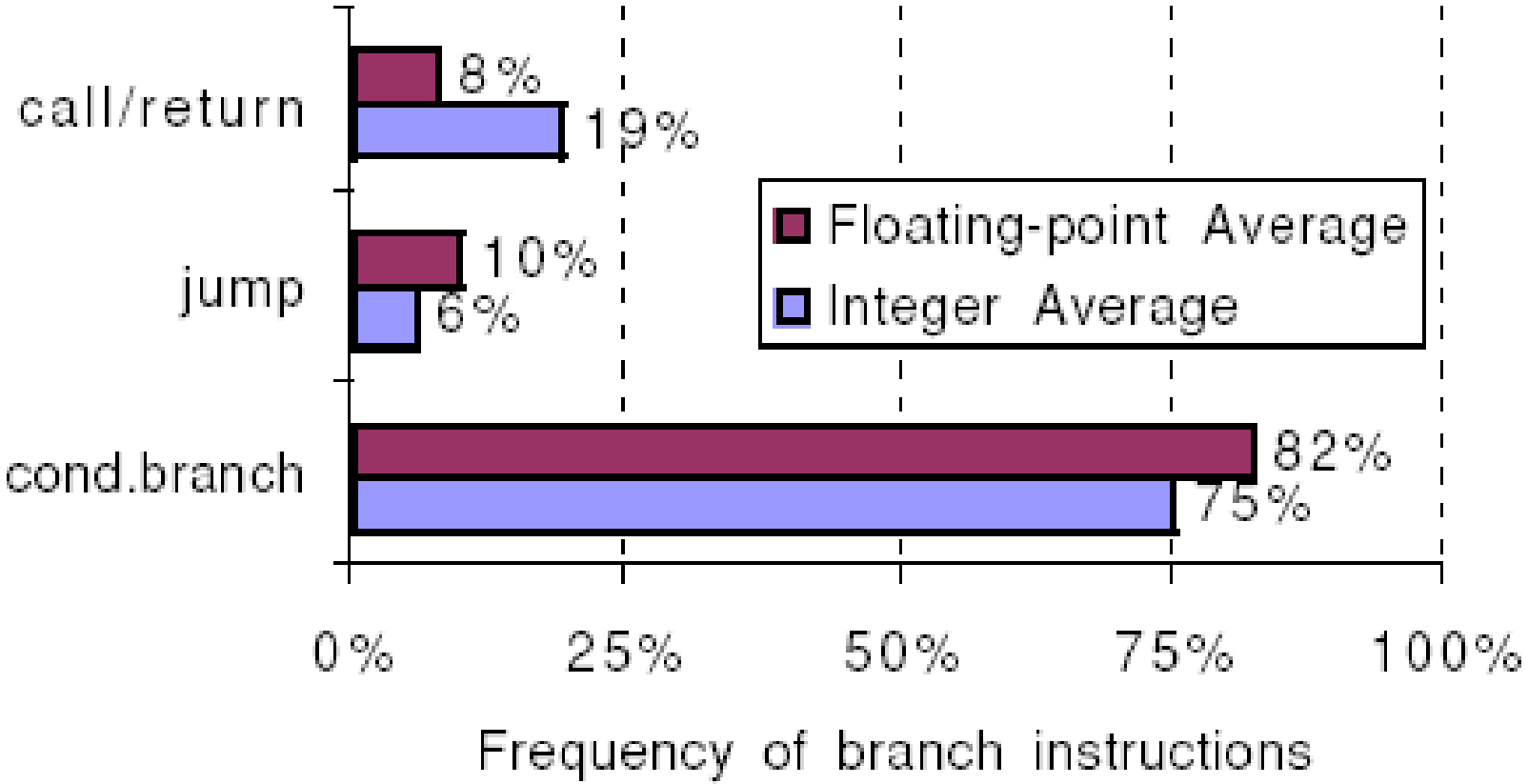
Growth of x86 instruction set over time



X86 Instruction Distribution

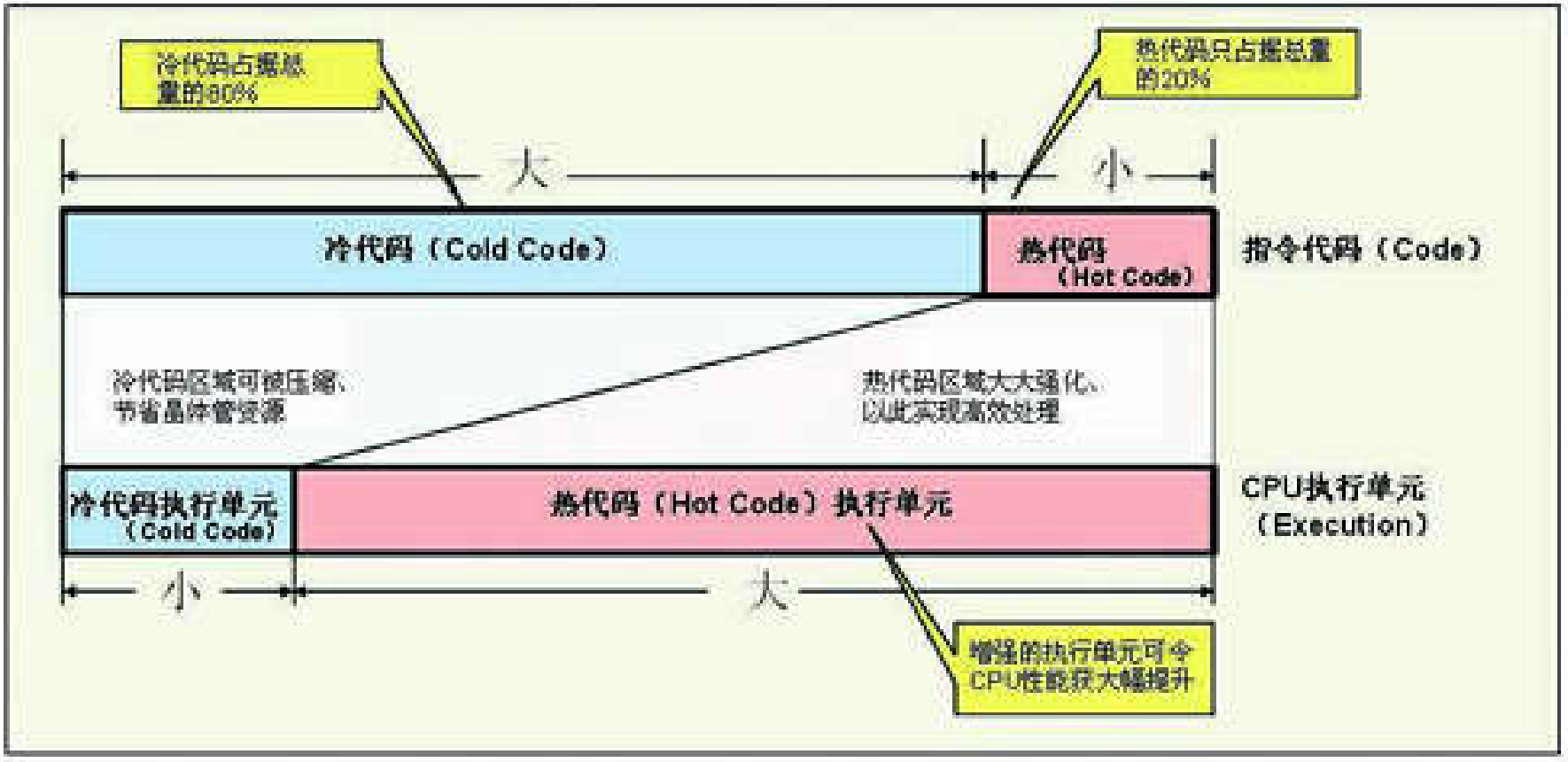
Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

Control Instruction Distribution



RISC的理论基础

计算机指令代码的80 / 20规律

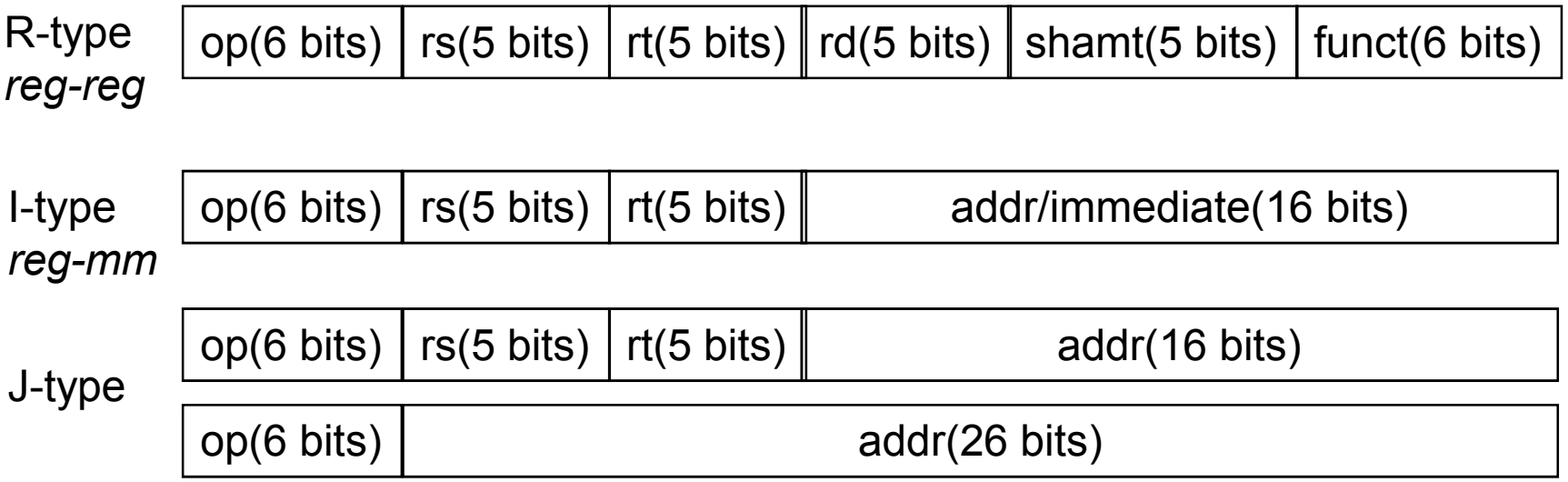


The RISC's eight principles:

- 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.

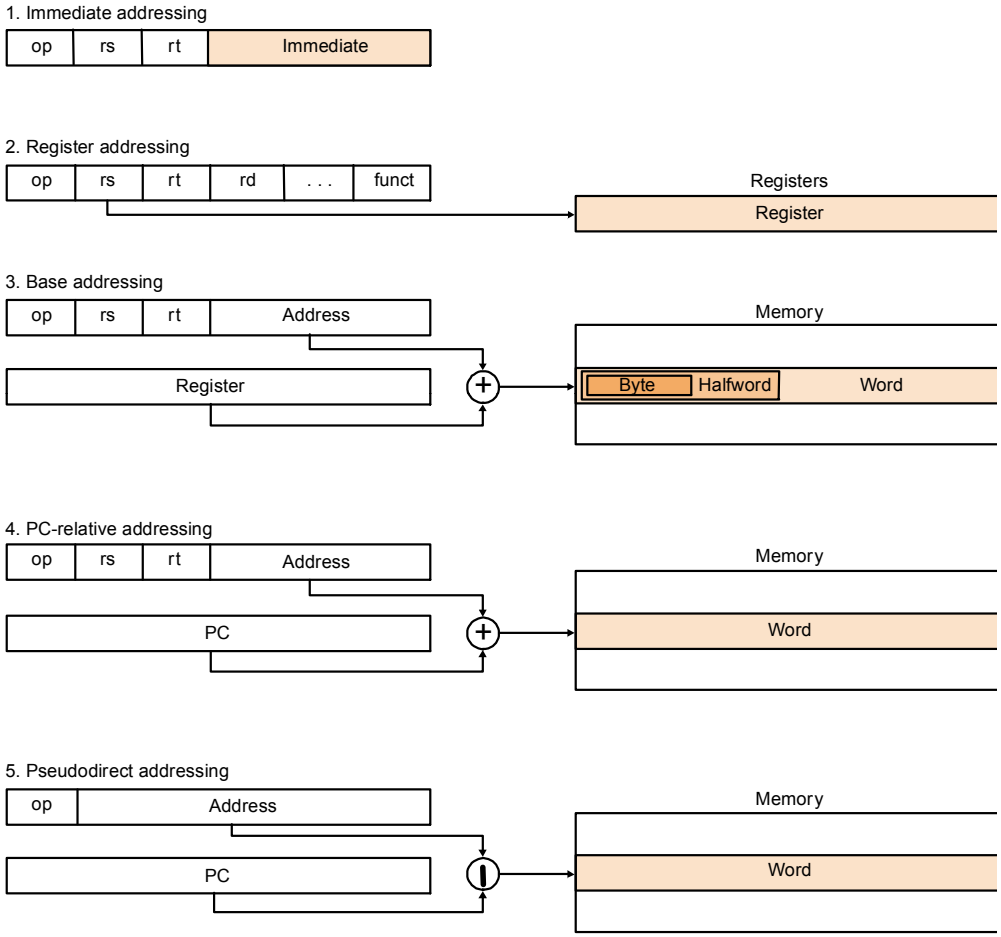
MIPS指令格式

- 100余条指令（Hennessy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)

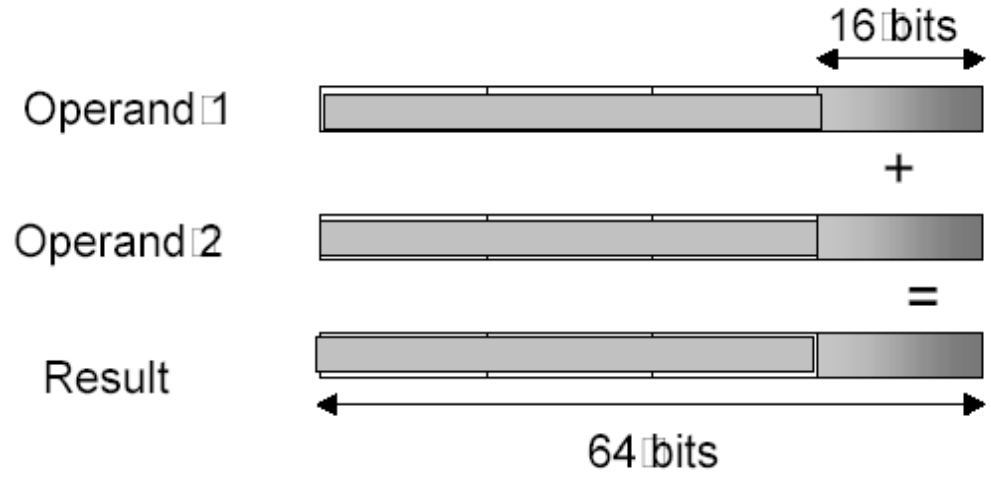


MIPS寻址模式

- 寄存器寻址：R-type
- 基址寻址：I-type
- 立即寻址： I-type
- 相对寻址：J-type
- 伪直接寻址：J-type
  - pseudodirect addressing
  - 26位形式地址左移2位，与PC的高4位拼接

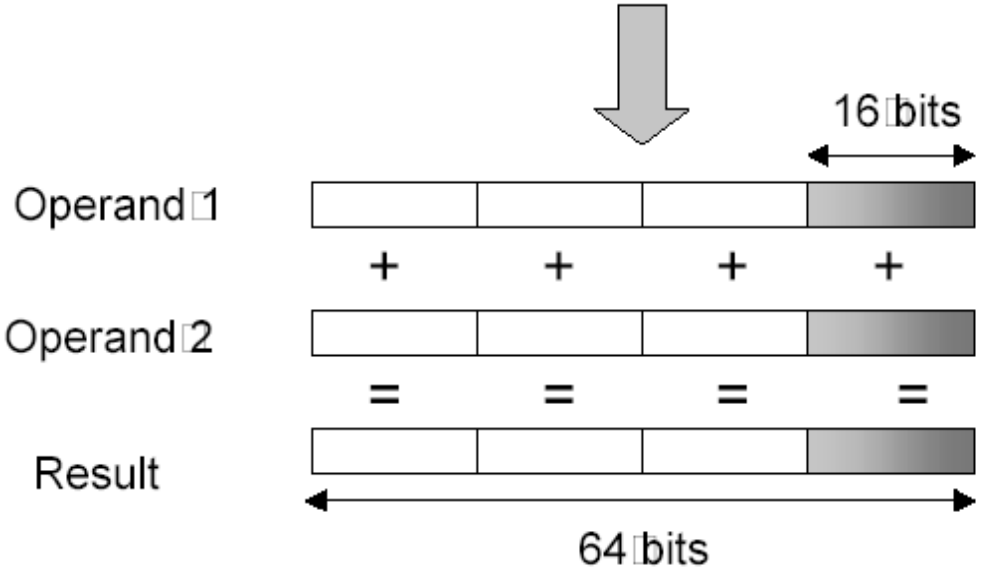


SIMD Instructions



48 bits are wasted!

Can we use them in any way?



4 operations in 1 cycle
SPEEDUP: 4X??

Called SIMD:
single-instruction multiple-data parallelism

SIMD Instructions

MD-technique

- Multiple data operands per operation

Vector instruction:

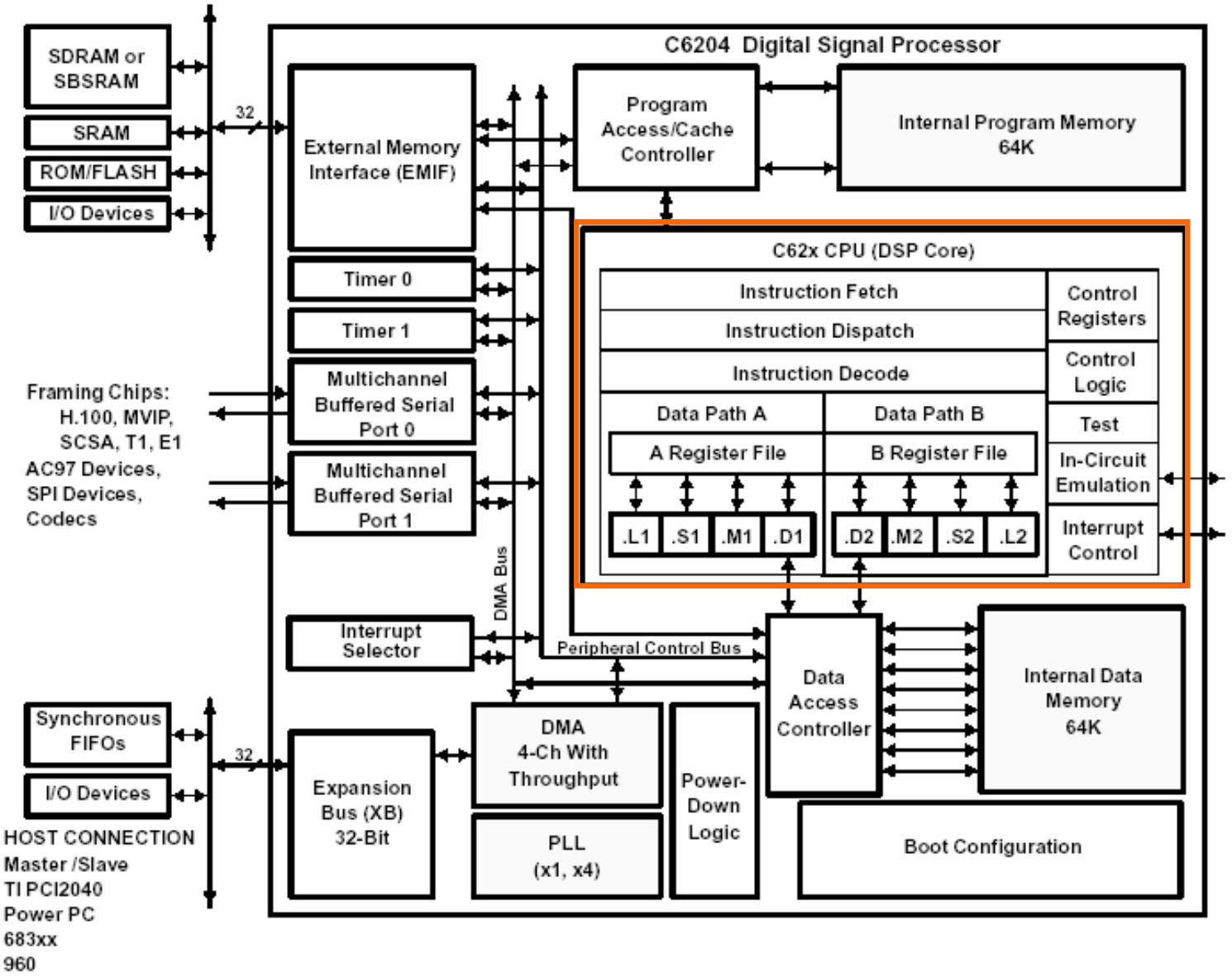
```
for (i=0, i++, i<64)
  c[i] = a[i] + 5*b[i];
```

$\vec{c} = \vec{a} + 5*\vec{b}$

Assembly:

```
set     vl,64
ldv     v1,0(r2)
mulvi   v2,v1,5
ldv     v1,0(r1)
addv    v3,v1,v2
stv     v3,0(r3)
```

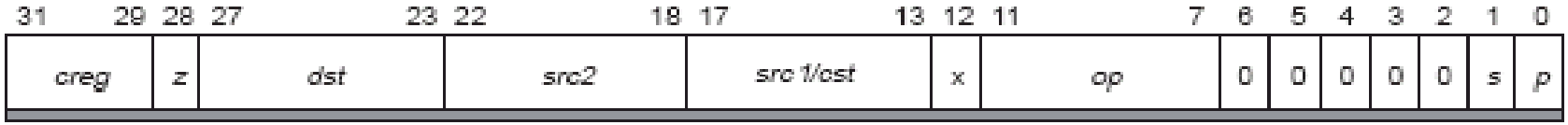
TMS320C64x：VLIW



TMS320C64x指令字

Operations on the .L unit

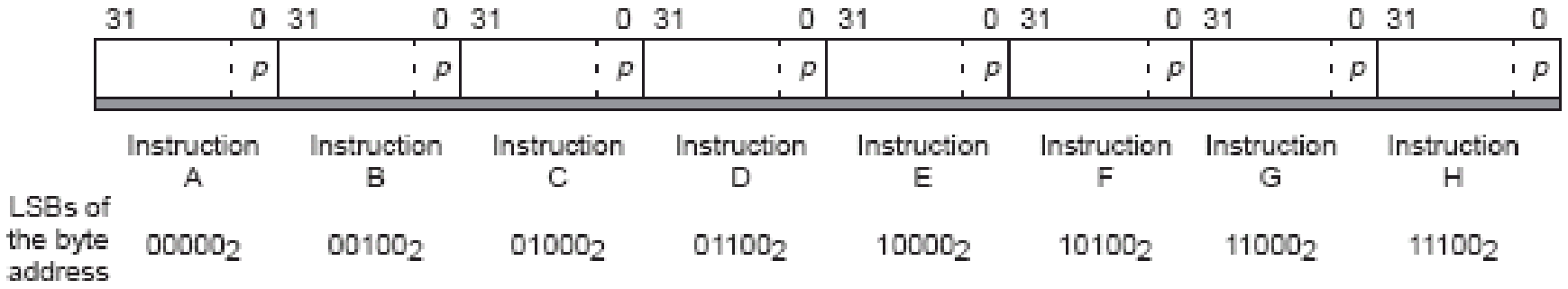
Operations on the .M unit

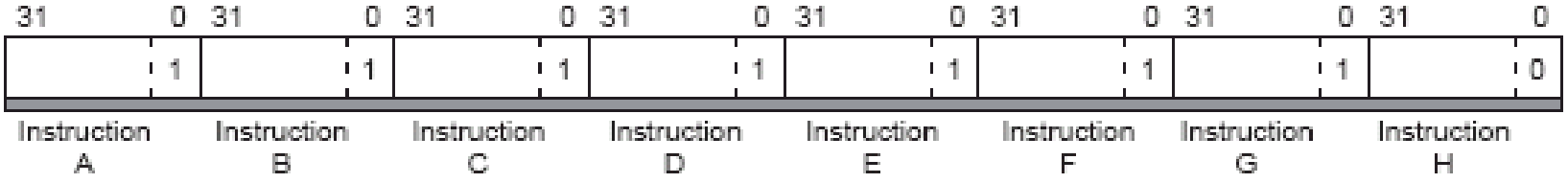


Operations on the .D unit

Operations on the .S unit

TMS320C64x：取指包（并行）

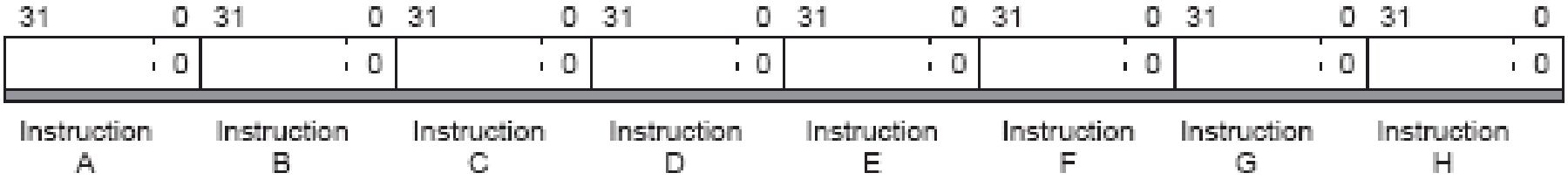




results in this execution sequence:

Cycle/Execute Packet	Instructions							
1	A	B	C	D	E	F	G	H

TMS320C64x：取指包（串行）



results in this execution sequence:

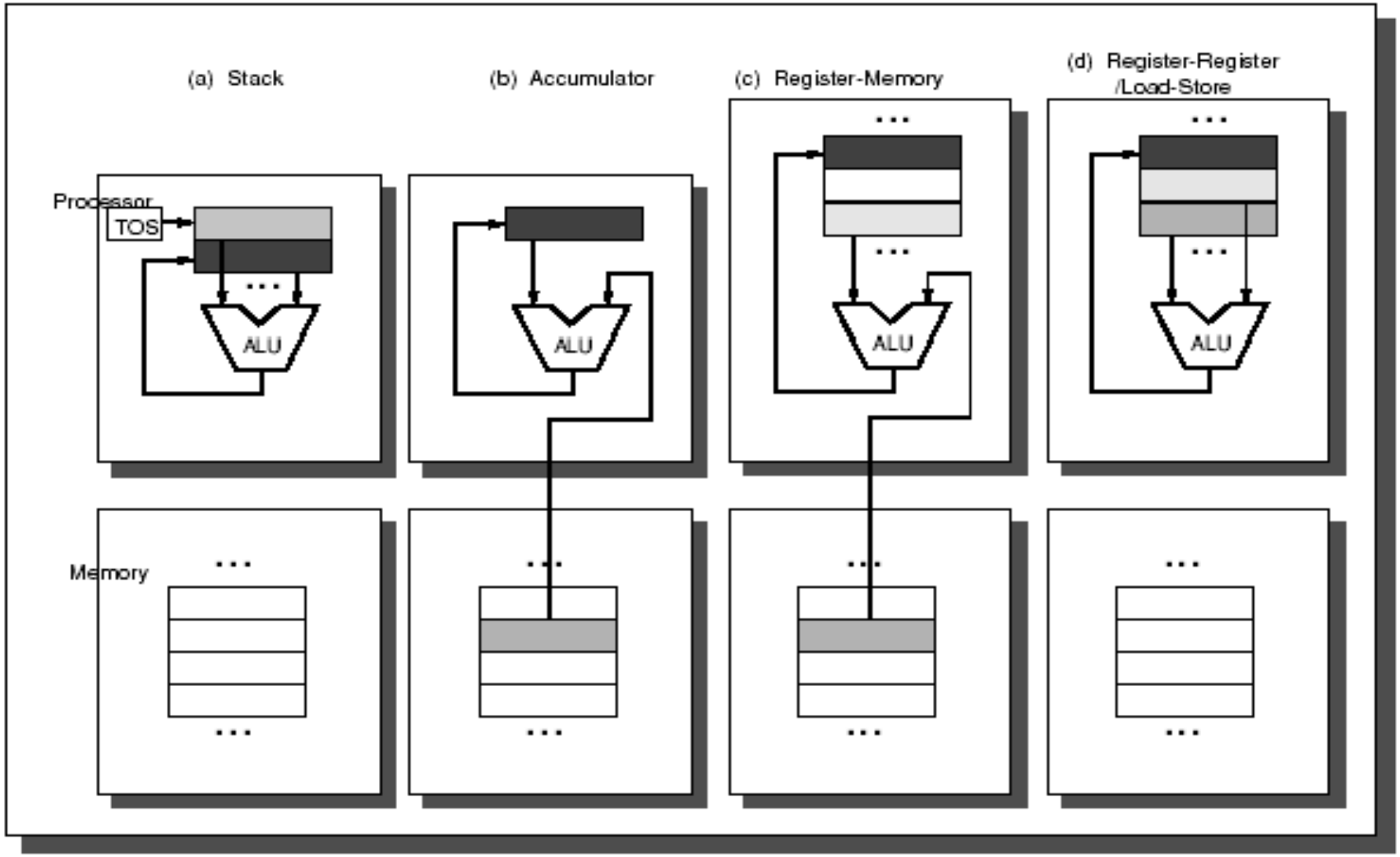
Cycle/Execute Packet	Instructions
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H

ISA Classes (CPU prospective)

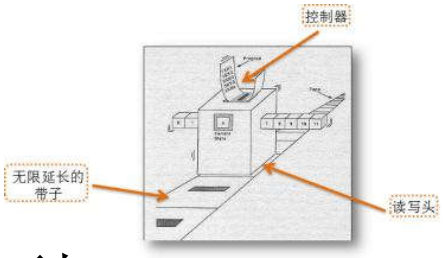
Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

$$C = A + B$$

ISA Classes (cont.)



More about MoC（register machine）



- 1950’s初的两个研究方向
  - 用图灵机刻画计算机
  - 建立与图灵机等价的“类计算机”模型
    - 指令顺序执行、条件跳转
    - Motivation
      - Emil Post问题： computability theory
      - David Hilbert问题： mathematics of formal systems
- register machine：a computer-like model
  - counter-machine：也称Minsky machine
    - "cut the tape" into many
      - the heads indicate "the tops of the stack“
      - A Turing machine can be simulated by two stacks

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即采用堆栈模型
  - FORTH

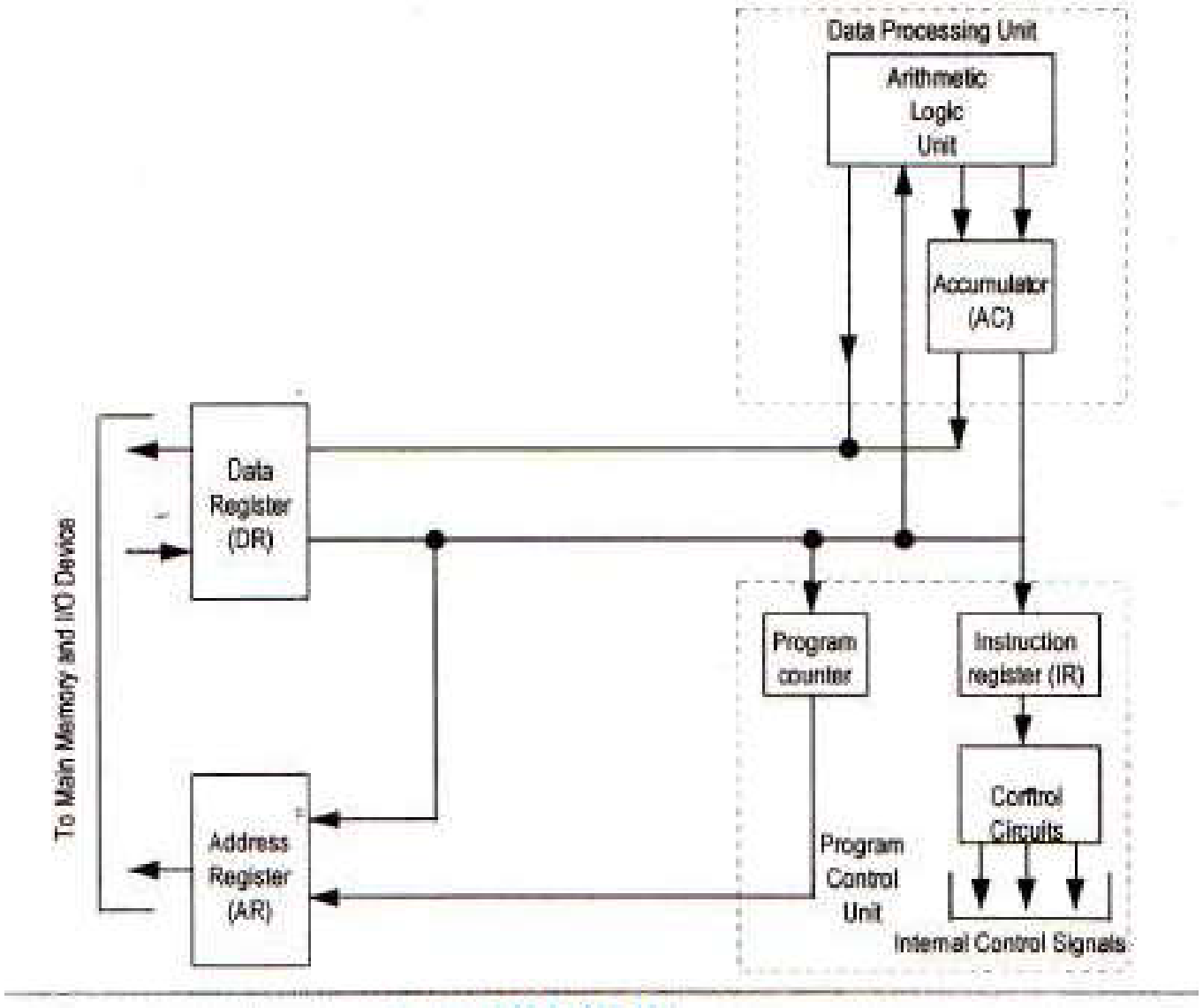
More about MoC（stack machine）

- a **stack machine** is a real or emulated computer
  - uses a pushdown stack rather than individual machine registers to evaluate each sub-expression in the program.
  - A stack computer is programmed with a reverse Polish notation instruction set.
  - The operands of the ALU are always the top two registers of the stack and the result from the ALU is stored in the top register of the stack.
- Dense machine code；More memory references
- The common alternative to stack machines are register machines
  - in which each instruction explicitly names the specific registers to use for operand and result values.

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（1949）。
- 采用“memory-based operand-addressing mode”。
  - 进化：采用一些专用的独占的寄存器，如数组指针、堆栈指针、乘除运算等
    - 如x86，称为“extended accumulator arch”

Block Diagram of an Accumulator Based CPU



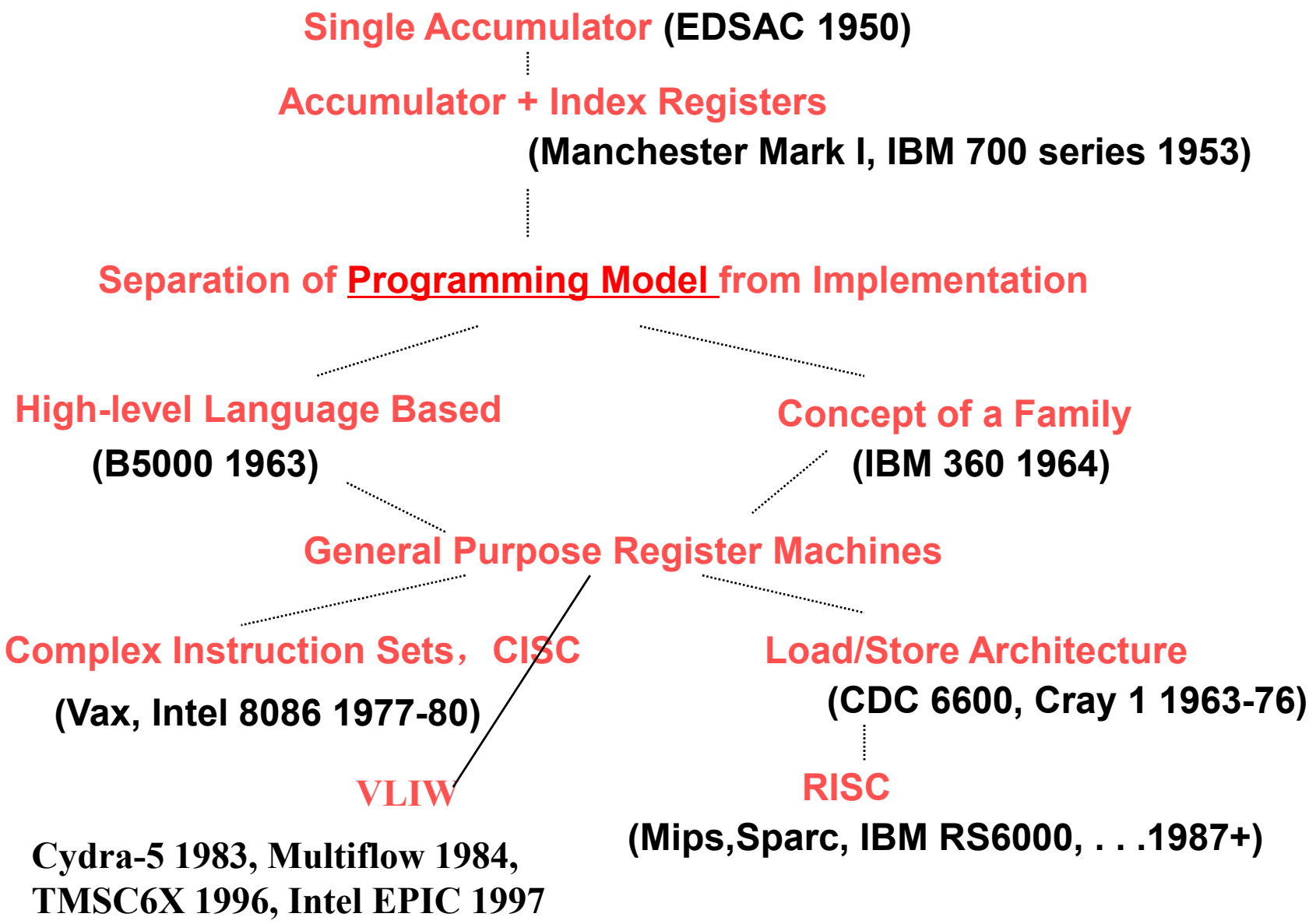
General-Purpose Register Architecture

- 所有寄存器可以用于任何目的
  - 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

High-level-language Computer Architecture

- 70's，使用高级语言编写的软件很少。
  - 在Unix之前，所有的商业软件都使用汇编语言。
  - 人们更关注于代码密度而不是编程语言和编译技术。
- 提倡“High-level-language Conputer Architecture”机器设计策略的目标是使硬件更靠近编程语言。
  - 例：Burroughs B5000
- 随着更高效的编程语言和编译器以及更大的存储空间使得这种思想逐渐消失。

Evolution of Instruction Sets

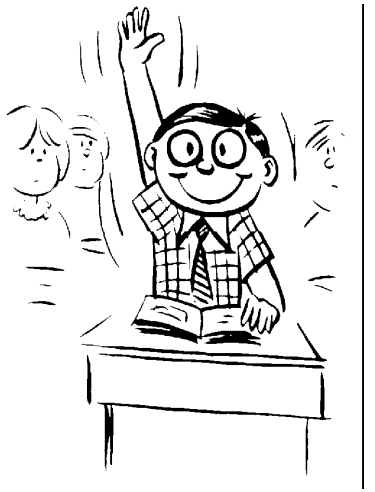


影响早期ISA设计的因素

- 内存小而慢，能省则省
  - 某个完整系统只需几千字节
  - 指令长度不等、执行多个操作的指令
- 寄存器贵，少
  - 操作基于存储器
  - 多种寻址方式
- 编译技术尚未出现
  - 程序是以机器语言或汇编语言设计
  - 当时的看法是硬件比编译器更易设计
    - 为了便于编写程序，计算机架构师造出越来越复杂的指令，完成高级程序语言直接表达的功能
    - 进化中的痕迹：X86中的串操作指令

ISA: A Minimalist Perspective

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



- 最小的计算机系统由哪些部件构成？
- 最小的ISA需要哪几条指令？
- 需要哪些寻址方式？

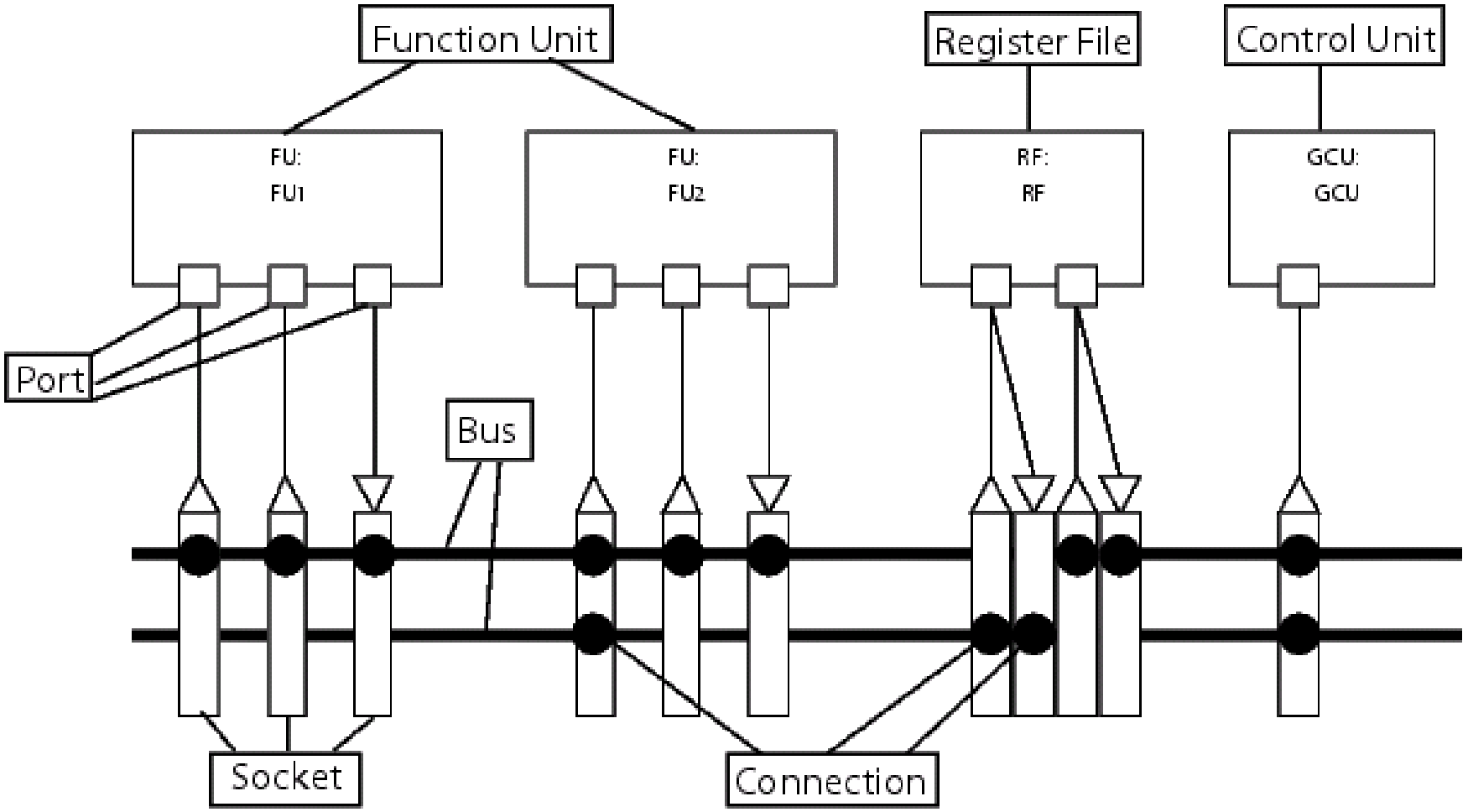
OISC：the one instruction set computer

- URISC体系：the ultimate reduced instruction set computer
  - 一条SBN指令：substract and branch if negative
- OISC的优势
  - 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
- 应用：嵌入式处理器

Transport triggered architecture

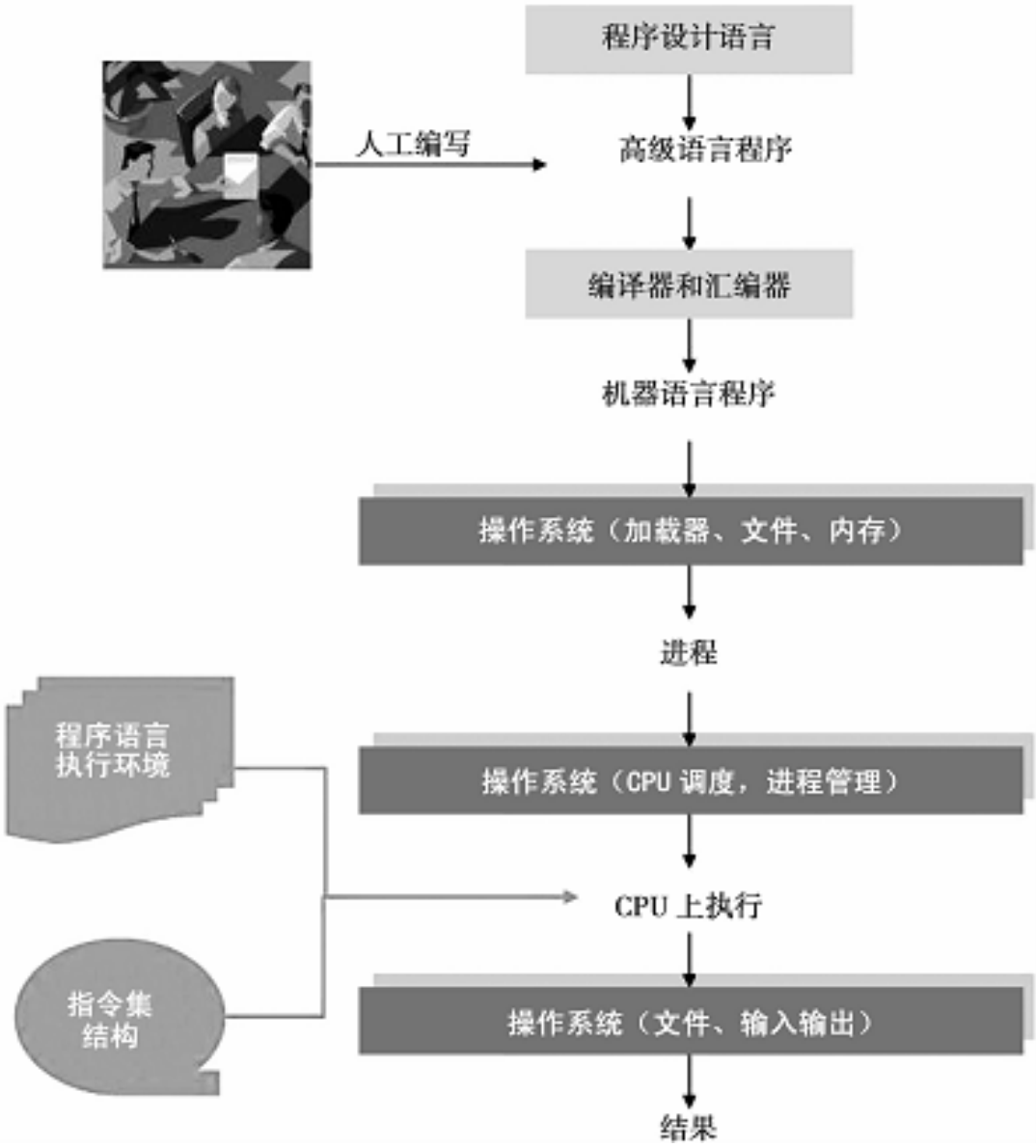
- originally called a "move machine”
  - 程序直接控制处理器内部的传输总线
- uses only the **MOVE** instruction
  - 计算：由memory-mapped ALU完成
    - 向功能单元的触发端口写数据将触发功能单元完成计算
      - 故曰transport triggered
  - 跳转：由memory-mapped PC完成
  - 适于实现ASIP
- 上市产品：MAXQ微控制器
  - 适合数模混合应用场景

TTA的典型结构


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

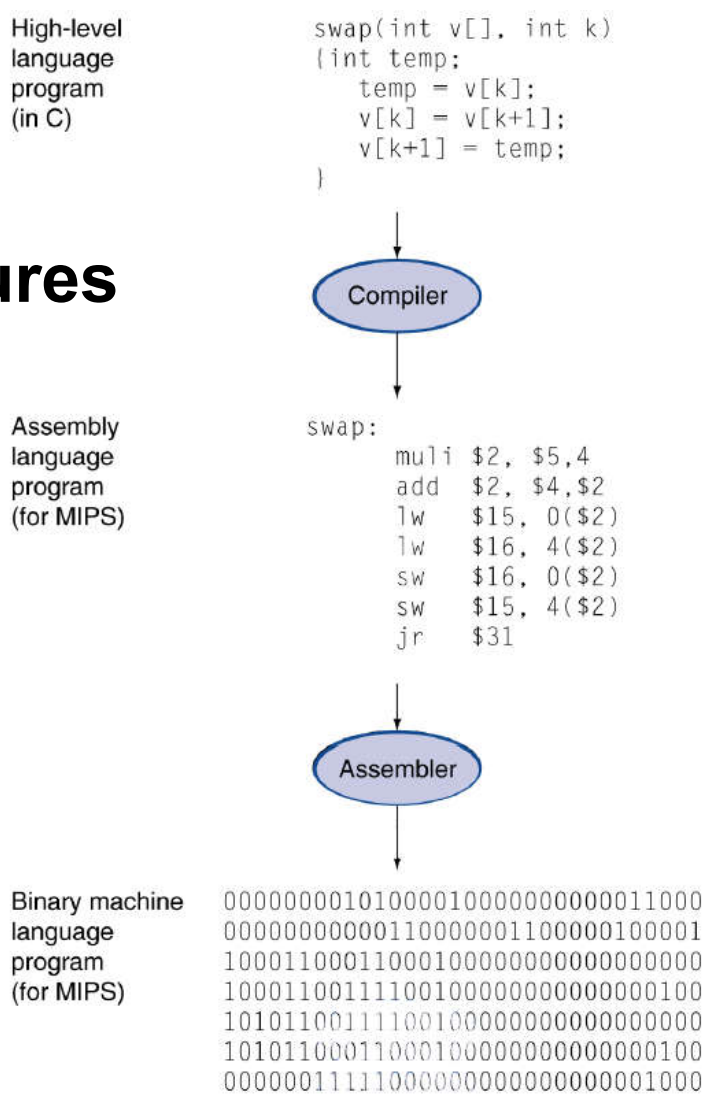
计算机解题过程

- 用户采用程序设计语言描述问题的求解过程，计算机在程序的控制下完成问题的求解
  - 计算机只能识别用0/1代码表示的程序
  - 用户需要使用高级语言编程



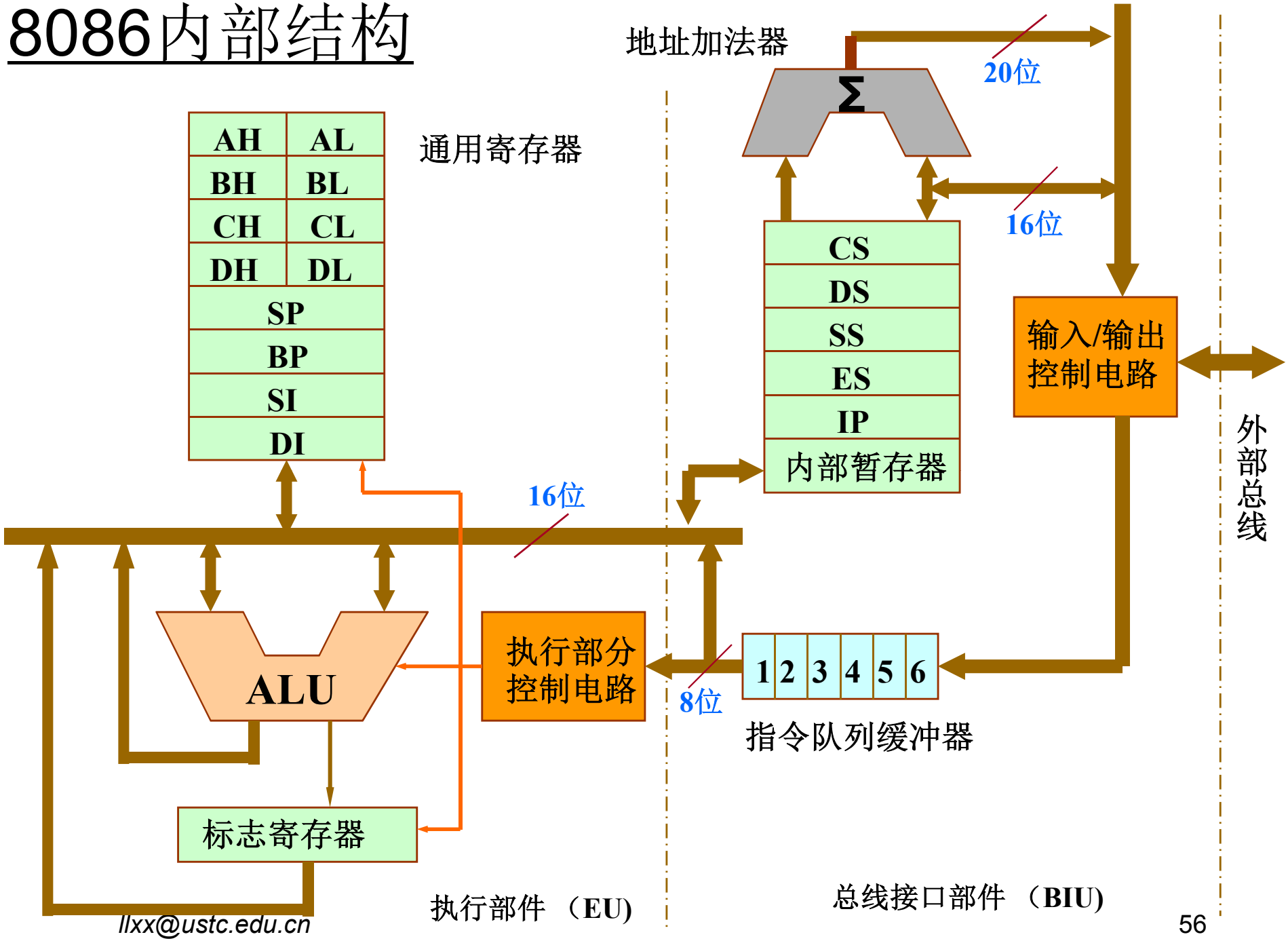
High Level to Assembly

- High Level Lang (C, C++, Fortran, Java, etc.)
  - Statements
  - Variables
  - Operators
  - Methods, functions, procedures
- Assembly Language
  - Instructions
  - Registers
  - Memory
- Data Representation
- Number Systems



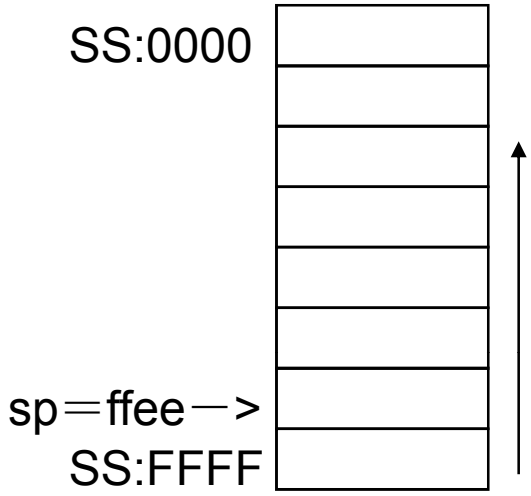
例：8086汇编语言

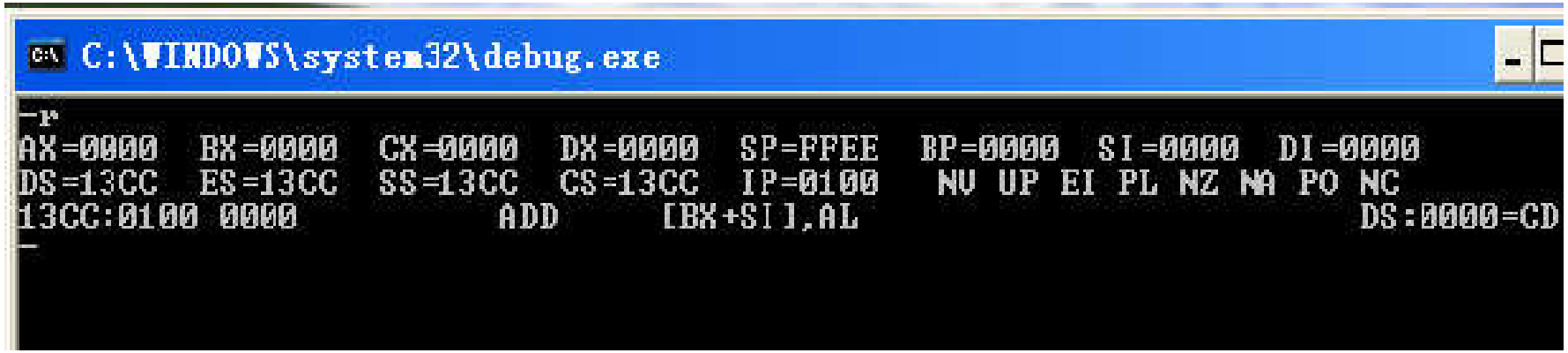
8086内部结构


地址加法器
Σ
20位
16位
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
通用寄存器
CS
DS
SS
ES
IP
内部暂存器
输入/输出
控制电路
外部总线
16位
ALU
执行部分
控制电路
1 2 3 4 5 6
8位
指令队列缓冲器
标志寄存器
执行部件 （EU)
总线接口部件 （BIU)

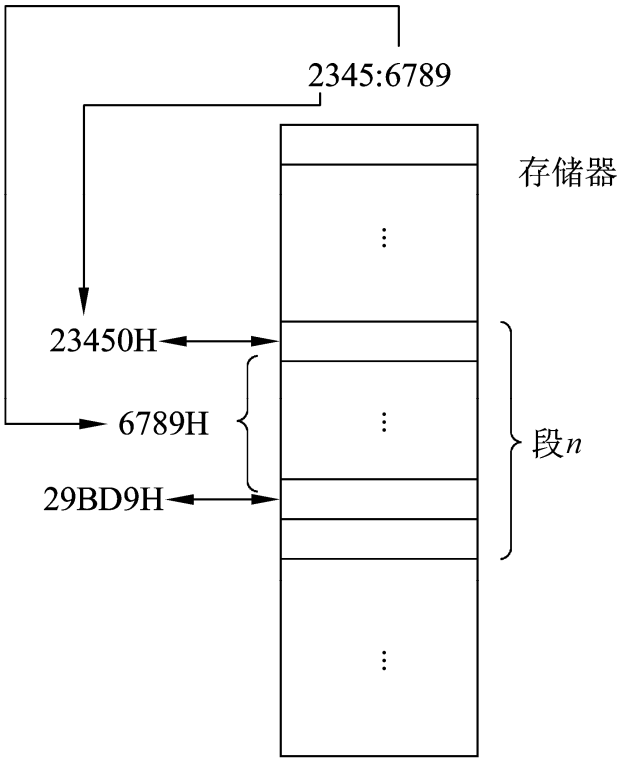
Debug：调试器，8086虚拟机

- 常用命令：a，d，e，g，r
- 存储模型：CS=DS=SS=ES
  - CS向下生长
  - SS向上生长
    - 空栈：sp＝FFEE





8086存储管理模式：段式

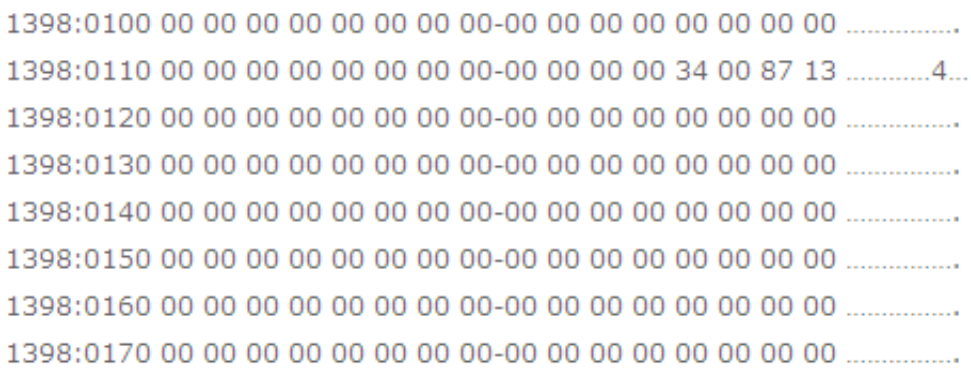



```
    2 3 4 5 0   （10H×段地址）
+     6 7 8 9   （偏移量）
-----------------
    2 9 B D 9   （物理地址）
```

```
1398:0100 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0110 00 00 00 00 00 00 00 00-00 00 00 00 34 00 87 13 ............4...
1398:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0130 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0140 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0150 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0160 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1398:0170 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

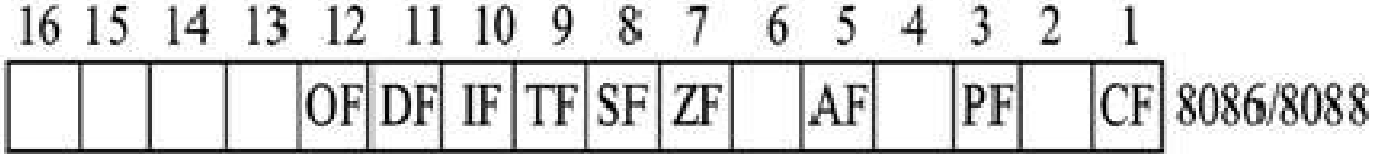
8086存储与I/O管理

- 内存空间：实地址模式
  - 中断向量区
  - 系统数据区(含STACK)
  - 操作系统区(含ISR)
  - 用户程序区
  - 数据缓冲区（ROM、显示缓冲）



- 存储模型（基址寻址）
  - 段式：数据段（ds）、代码段（cs）、堆栈段（ss）
  - 实地址方式：段基址寄存器16位（64K）
- I/O接口：独立地址空间
  - 端口寄存器（数据、命令、状态）

8086标志寄存器：系统当前状态



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）

```
C:\Users\Administrator>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B2D  ES=0B2D  SS=0B2D  CS=0B2D  IP=0100   NV UP EI PL NZ NA PO NC
0B2D:0100 7438          JZ      013A
-_
```

x86汇编语言

- 指令：机器指令助记符
- 伪指令
  - DW 定义字(2字节)
  - PROC 定义过程
  - ENDP 过程结束
  - SEGMENT 定义段
  - ASSUME 建立段寄存器寻址
  - ENDS 段结束
  - END 程序结束
- 系统调用（system calls）：BIOS，DOS
  - 显示、键盘、磁盘、文件、打印机、时间

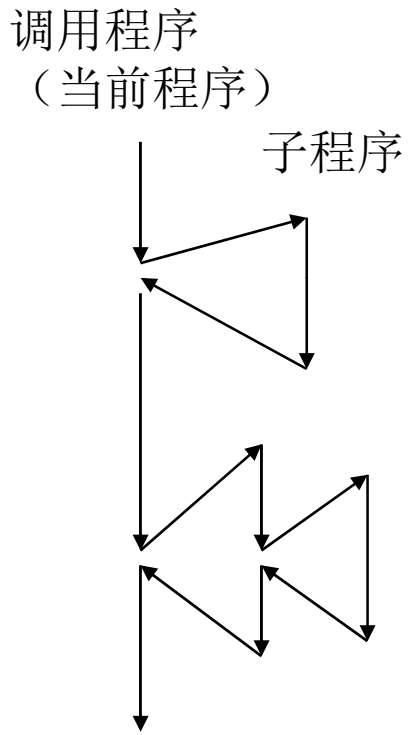
```
data SEGMENT '数据段，编程者可以把数据都放到这个段里
….数据部分
'数据格式是： 标识符 db/dw 数据。
data ENDS'数据段结束处。

edata SEGMENT '附加数据段，编程者可以把数据都放到这个段里
….附加数据部分
edata ENDS'附加数据段结束处。

code SEGMENT'代码段，实际的程序都是放这个段里。
ASSUME CS:code,DS:data,ES:edata '告诉编译程序，data段是数据段DS，code段是代码段CS
start:MOV AX,data '前面的start表示一个标识位，后面用到该位，如果用不到，就可以不加
MOV DS,AX '这一句与上一行共同组成把data赋值给DS。段寄存器.
MOV AX,edata
MOV ES,AX '与前一句共同组成edata->ES
…….程序部分
MOV AX,4C00h'程序退出，该句内存由下一行决定。退出时，要求ah必须是4c。
INT 21h
code ENDS'代码段结束。
END start'整个程序结束，并且程序执行时由start那个位置开始执行。
```

汇编程序例

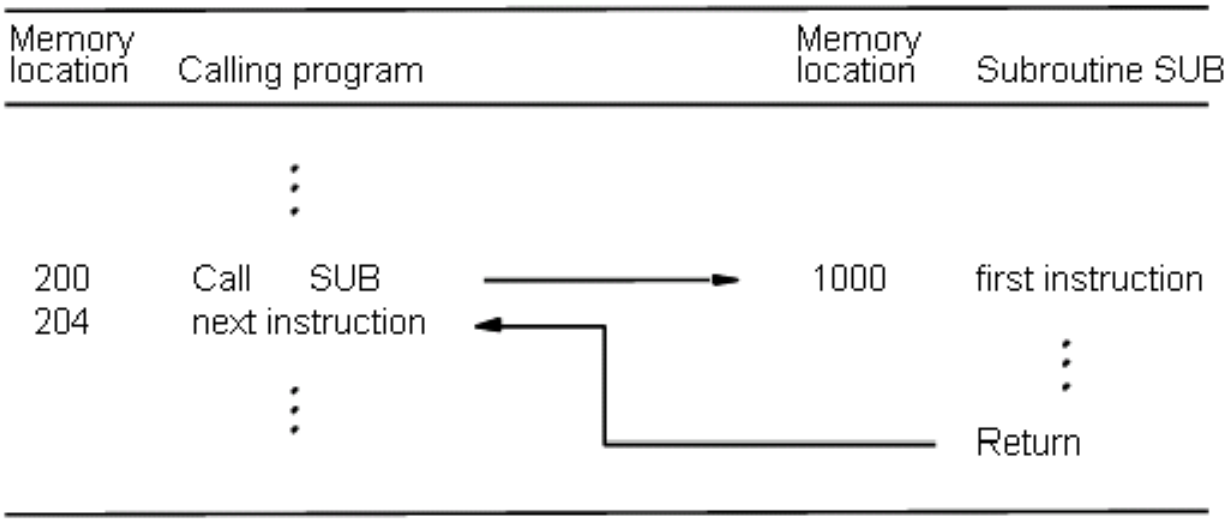
	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler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

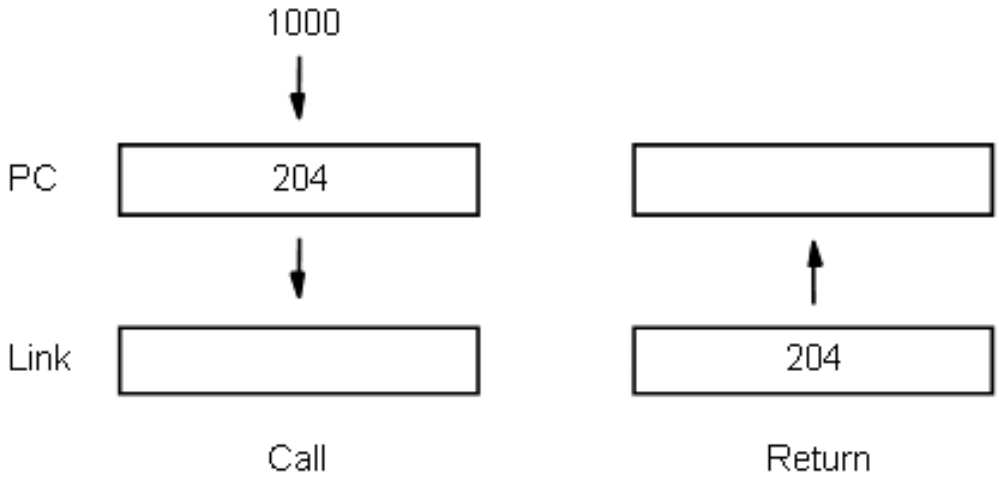
过程调用procedure

- 两个问题：参数传递，控制转移
- 步骤
  - 主将参数放在子过程可以访问的位置
    - 内存、寄存器、栈
  - Call子过程
    - 保存断点（nPC）
    - 将控制交给子过程（使PC指向子过程）
  - 子过程执行
    - 将结果放在调用程序可以访问的位置
  - Return
    - 将控制交回调用程序（PC = nPC）
- 控制转移方式
  - 方式一：call/return
  - 方式二：jmp

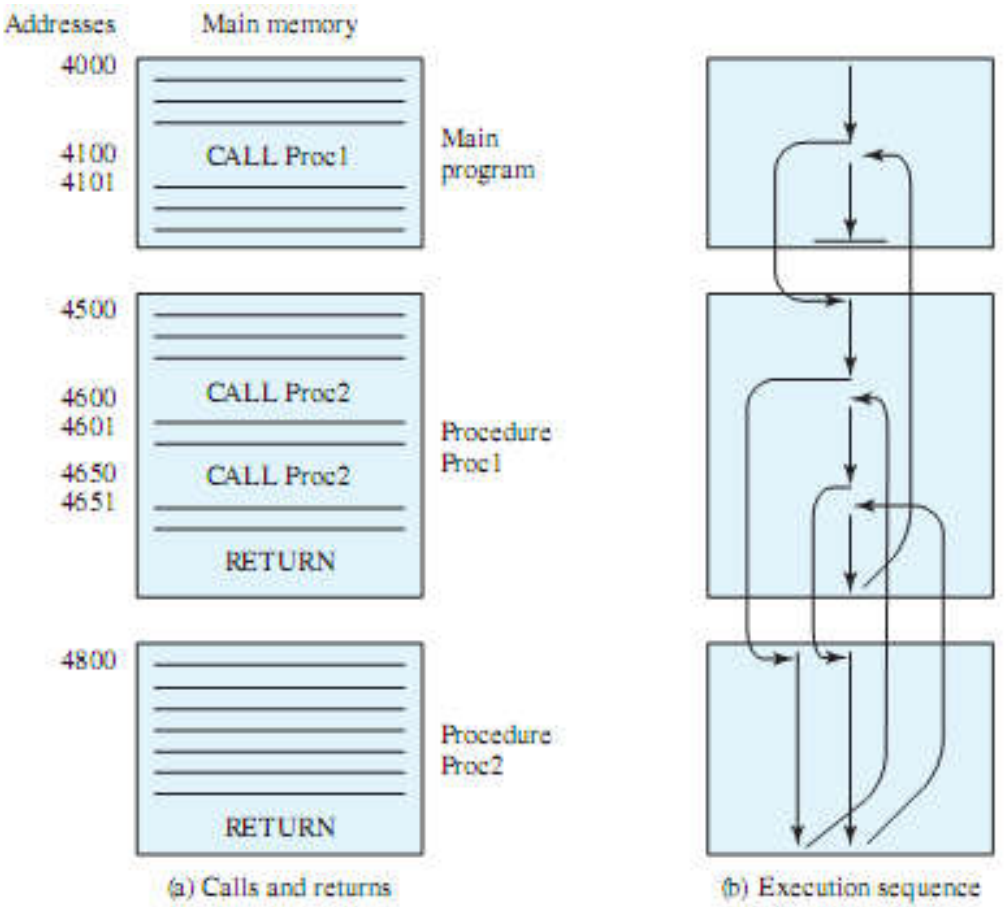


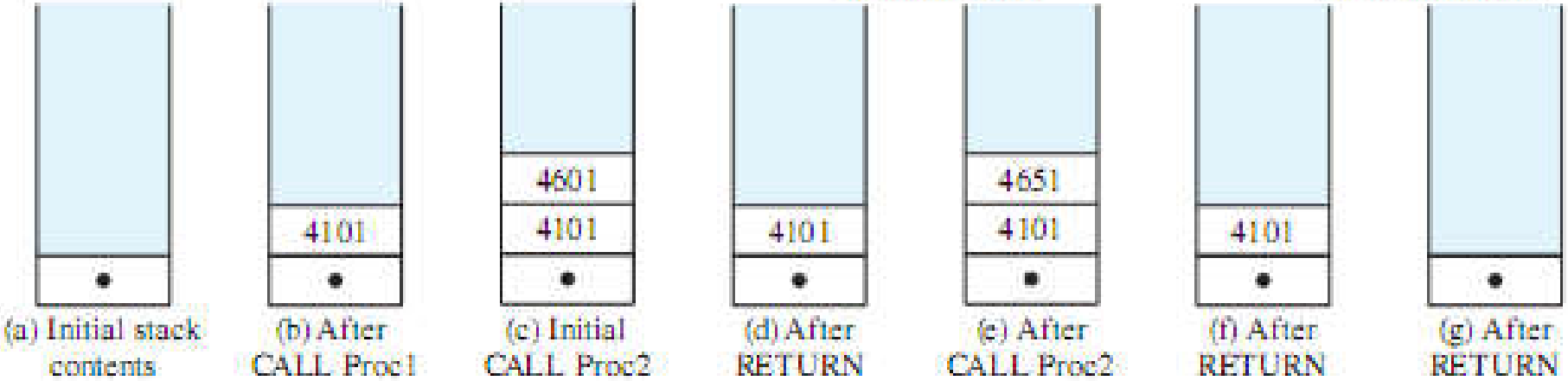
单级call & return：链接寄存器LR





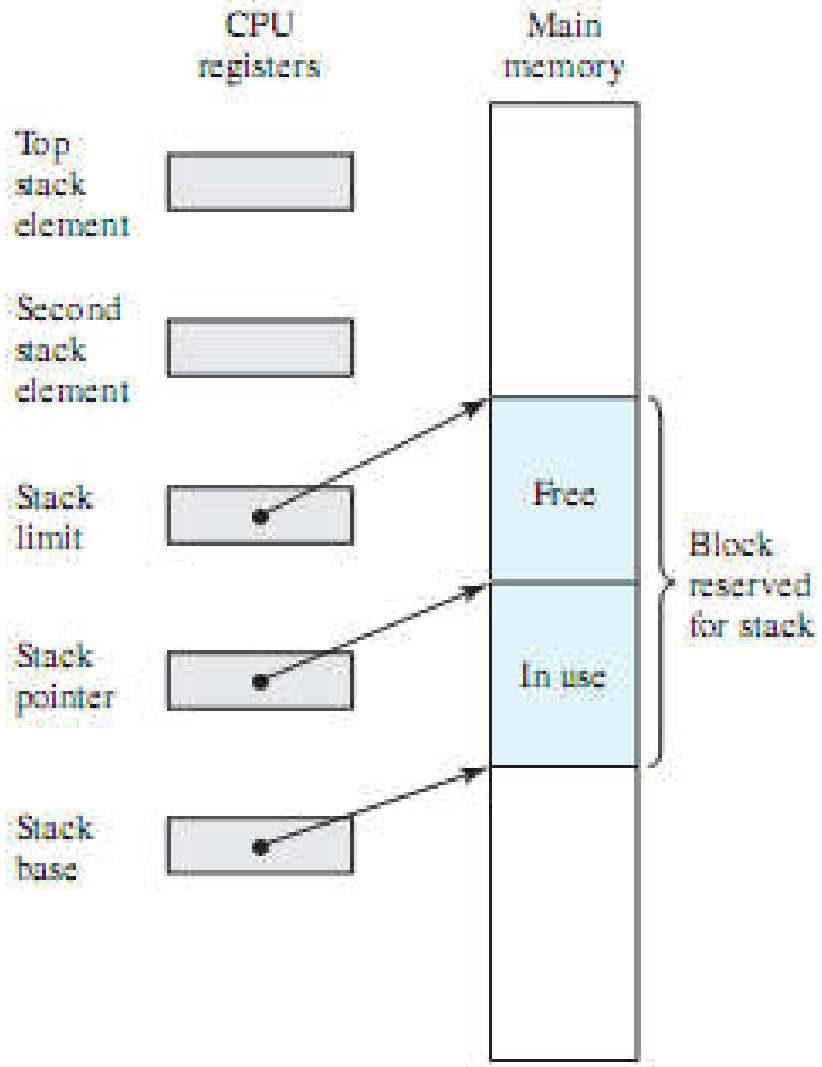
Use of Stack to Implement Nested Procedures



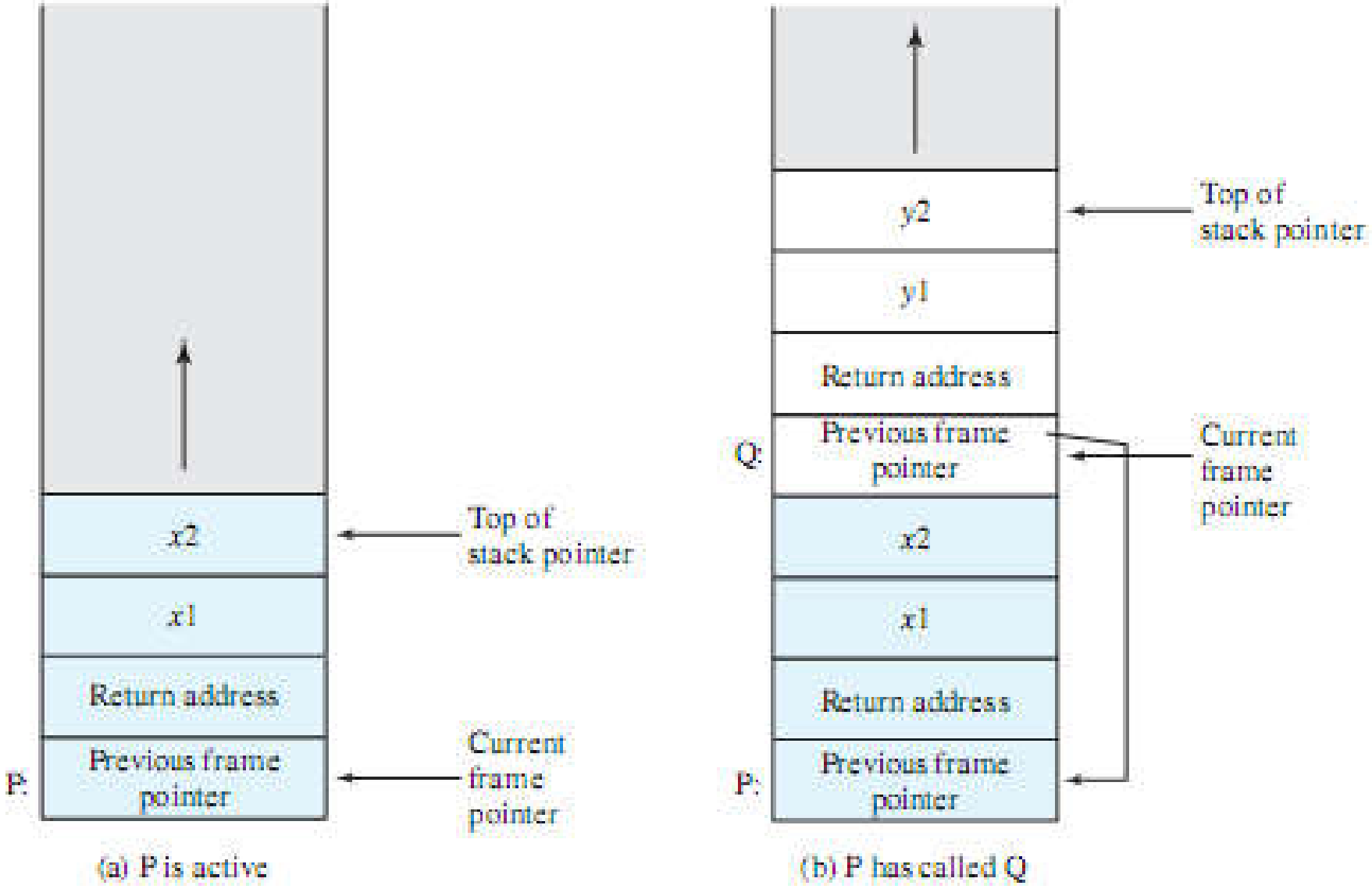


嵌套子程序：使用堆栈

- Stack组织
  - 处理器使用SP指向一个processor stack
    - call：将PC的内容压栈，将子程序地址取入PC
    - return：从堆栈中弹出返回地址，写入PC
- Stack Frame
  - 活动记录：栈中包含备份区（保存了当前过程调用传递的参数）和返回地址的一段单元
    - 备份区（bookkeeping）用于备份通用寄存器



stack frame：运行时栈



C调用约定（x86）

```
int add（int a, int b）
    { /* int c;
      char d; */

    return a＋b;
     }
```

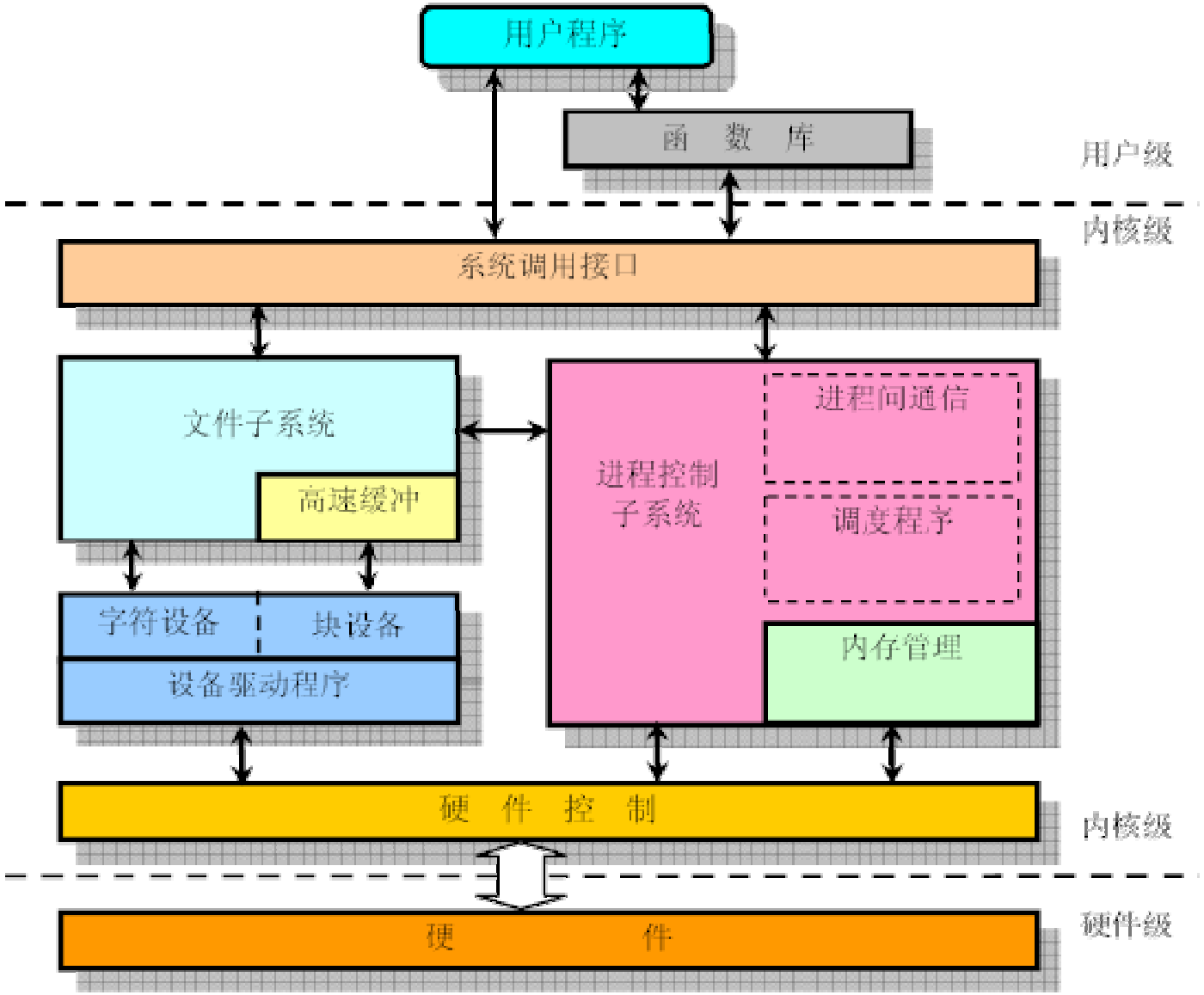
```
0C1C:0218 55          PUSH   BP
0C1C:0219 8BEC        MOV    BP,SP
0C1C:021B 8B4604      MOV    AX,[BP+04]
0C1C:021E 034606      ADD    AX,[BP+06]
0C1C:0221 EB00        JMP    0223
0C1C:0223 5D          POP    BP
0C1C:0224 C3          RET
```

1. 按从右到左的顺序将参数入栈：先b后a
2. 返回地址入栈：2字节（16位系统）
3. 进入被调函数
4. BP入栈；将SP拷贝到BP
5. 局部变量（自动变量）从后到前入栈：先d后c
6. 返回值占2字节的通过寄存器AX返回；返回值占4字节的通过DX:AX返回。
7. 将BP拷贝到SP
8. 弹出入口时保留的BP值到BP
9. 返回调用者
10. 清调用时压栈的参数

System calls

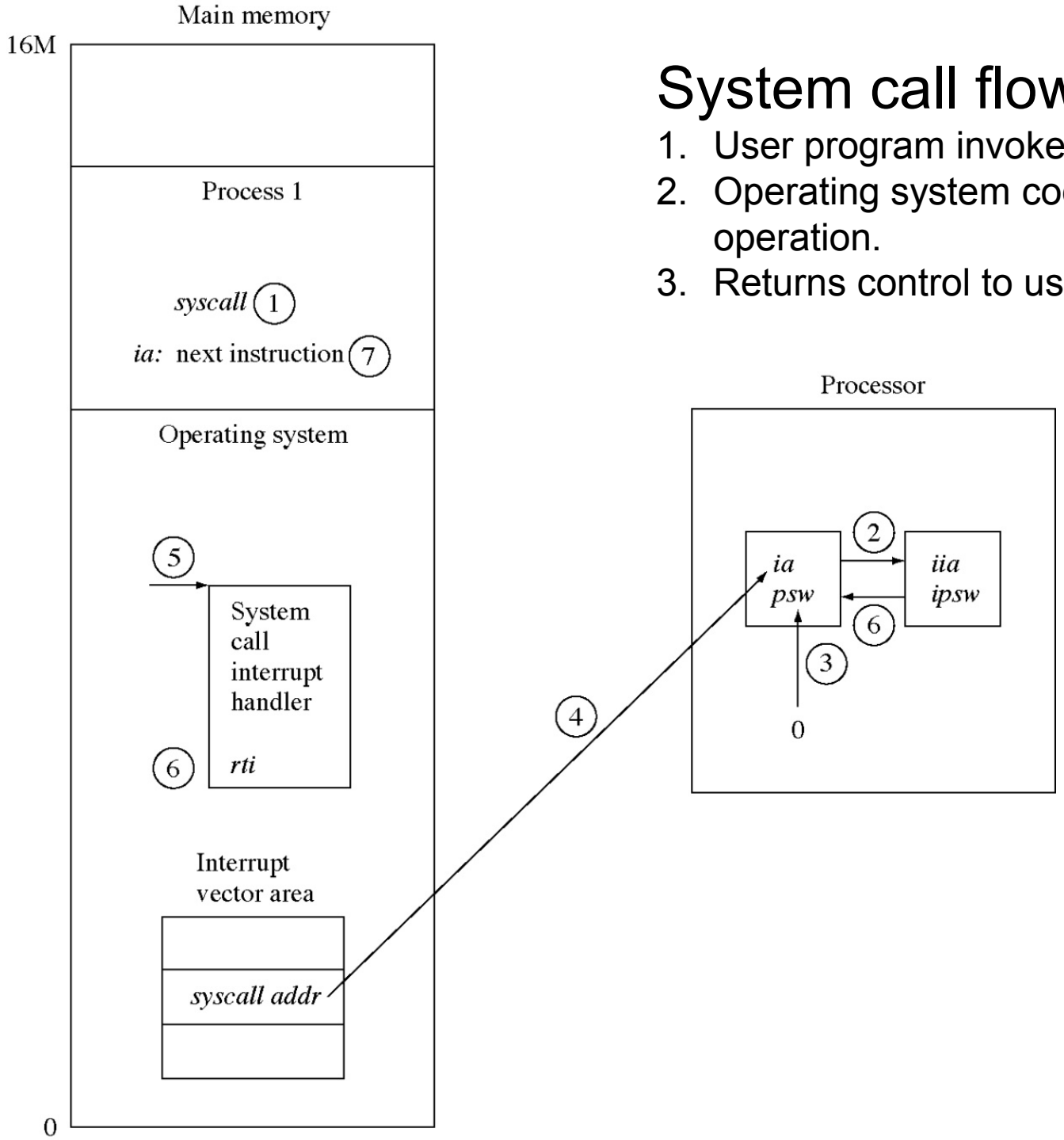
- Why
  - Certain operations require specialized knowledge and protection
  - Not every programmer knows (or wants to know) this level of detail
  - Usually not generated by HLLs，but in assembly language functions
- What
  - A special machine instruction that causes an interrupt
    - 产生状态切换，需保存PSW
  - various names: syscall, trap, svc
  - x86系统调用（system calls）：BIOS，DOS
    - 显示、键盘、磁盘、文件、打印机、时间

API 、 system calls

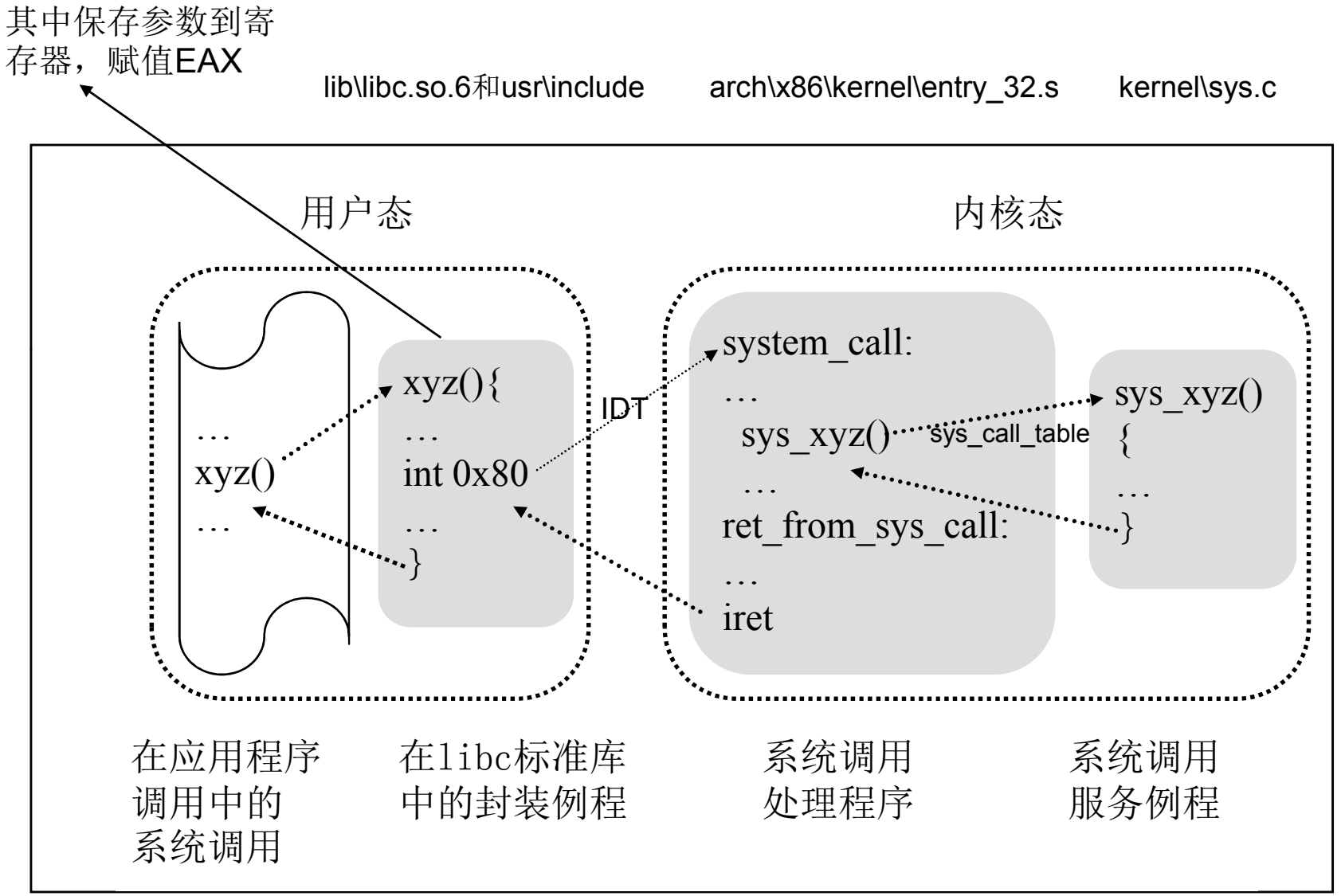


System call flow of control

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



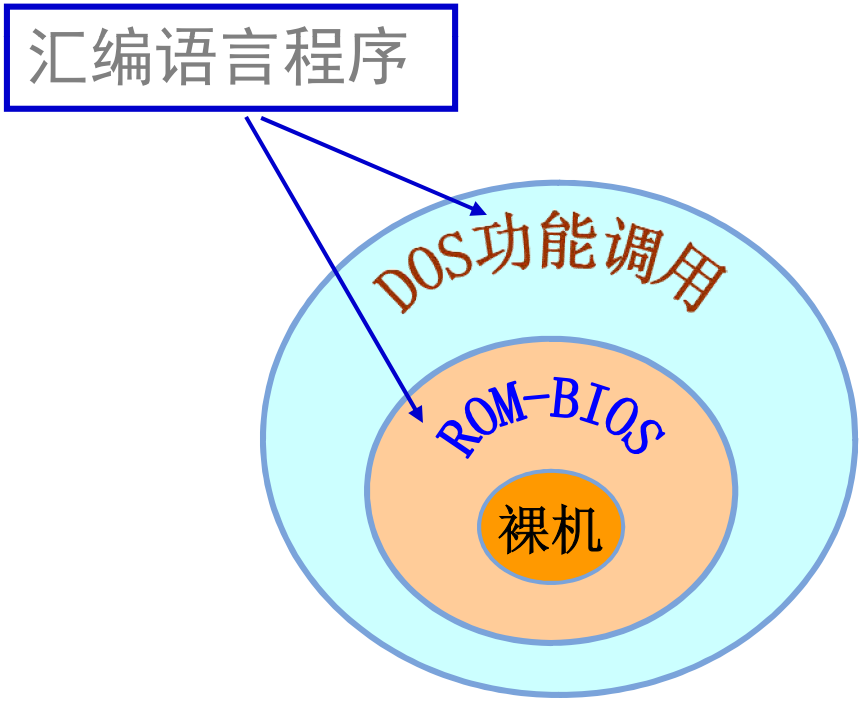
系统调用


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

IBM PC的BIOS调用及DOS功能调用

- 向显示器输出字符
  1. 字符的输出
  2. 字符串的输出
- 从键盘输入数据
  1. 字符的输入
  2. 字符串的输入
  3. 按键的判断
- 磁盘输入/输出数据
  1. 文件读
  2. 文件写

在内存的0FE000H开始的8KB ROM中存放着基本输入输出系统BIOS（Basic I/O System）21H号中断是DOS提供给用户的用于调用系统功能的中断，它有近百个功能供用户选择使用，主要包括设备管理、目录管理和文件管理三个方面的功能



例：字符输入输出

```
loop: mov ah,0          ; 键盘功能调用（int 16h）
      int 16h           ; al←按键的ASCII码
      mov bx,0          ; 显示功能调用（int 10h）
      mov ah,0eh
      int 10h           ; 显示
getk: mov ah,0bh        ; 按任意键继续
      int 21h
      or al,al          ; al＝0?
      jz getk           ; al＝0，没有按键，继续等待
      jmp loop
      int 3h
```

Debug命令：a，d，e，g，r

MIPS汇编示例

C code:
```
A = B + C + D;
E = F - A;
```

MIPS code:
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

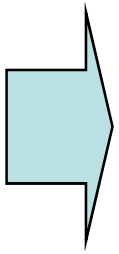
C code:
```
A[8] = h + A[8];
```

MIPS code:
```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

```
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1:sub $s3, $s4, $s5
Lab2:...
```

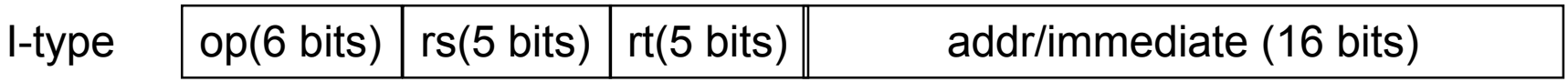
```
swap(int v[], int k);
{ int temp;
   temp = v[k]
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



```
swap:
   muli $2, $5, 4
   add  $2, $4, $2
   lw $15, 0($2)
   lw $16, 4($2)
   sw $16, 0($2)
   sw $15, 4($2)
   jr $31
```

位扩展

- 需求：字长32位
  - addi $s3,$s3,4；$s3 = $s3 + 4
  - lw $t1, offset($t2); M($t2+offset) -> $t1
  - beq：以npc为基准，指令中的target为16位，进行32位有符号扩展后左移两位（补00）
- 位扩展：从较小的数据类型转换成较大的类型
  - 无符号扩展（zero extension）：高位补0
  - 符号扩展（sign extension），补码：高位补1



Policy of Use Conventions for registers

$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluat
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

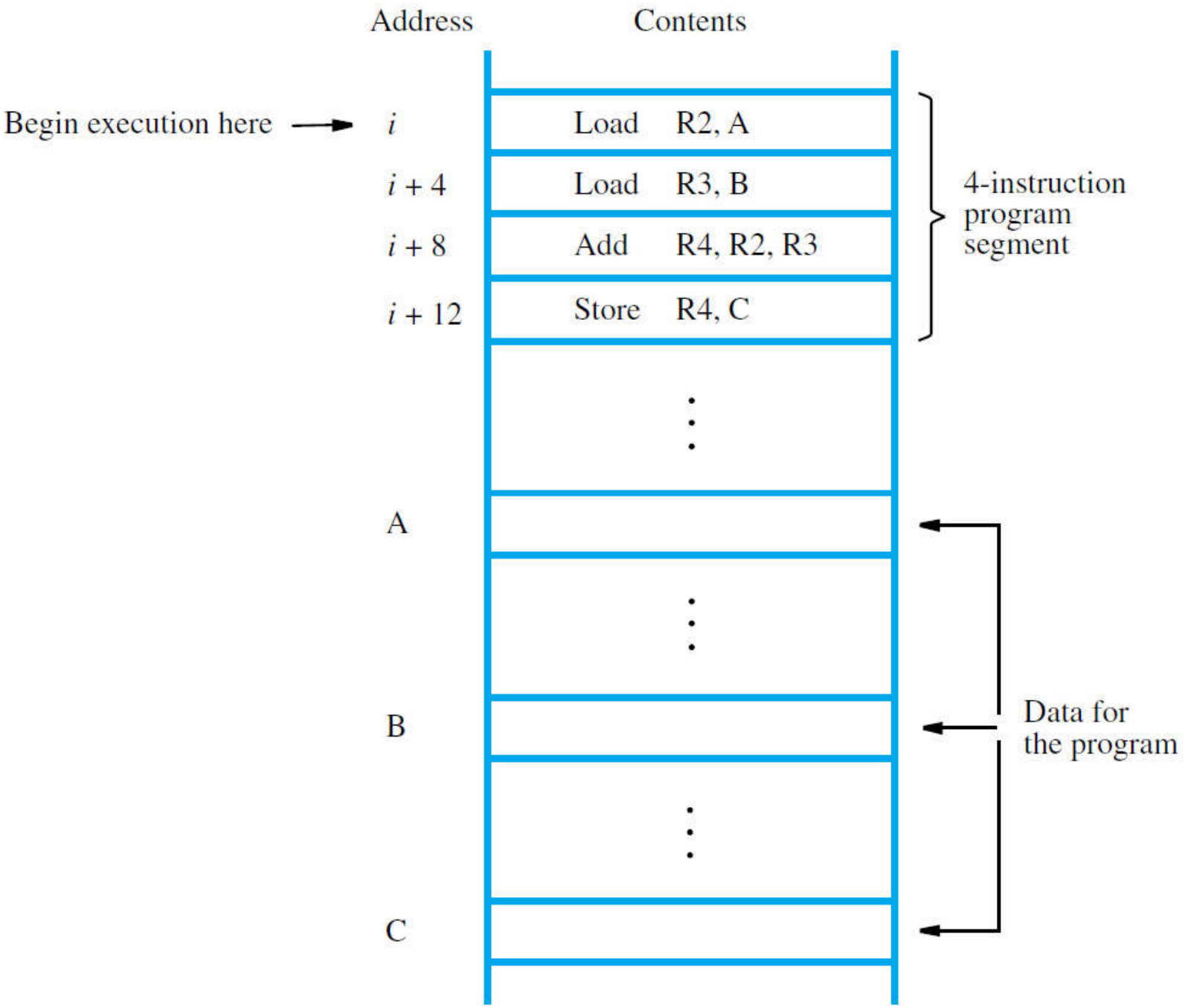
MIPS过程调用

- for procedure calling
  - $a0–$a3: four argument registers in which to pass parameters
  - $v0–$v1: two value registers in which to return values
  - $ra: one return address register to return to the point of origin
- jal： jump-and-link， “call”
- jr： jump register， “return”

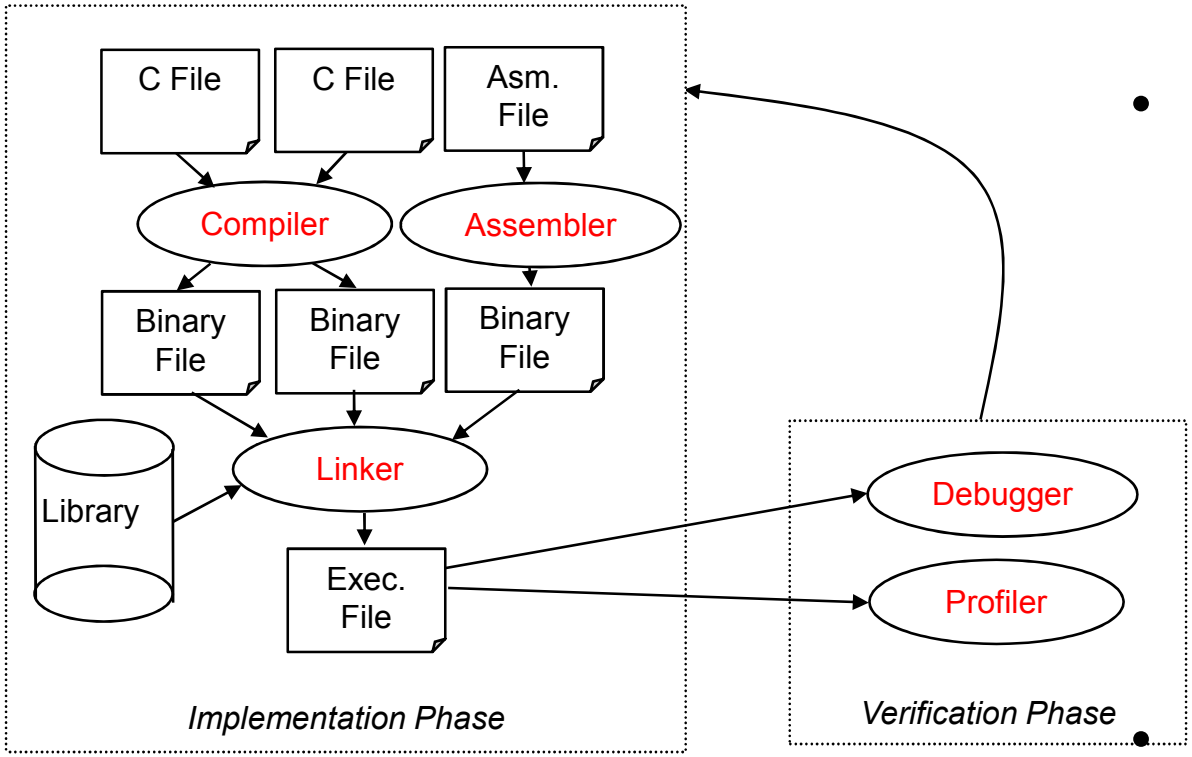
可执行程序生成与执行

汇编程序

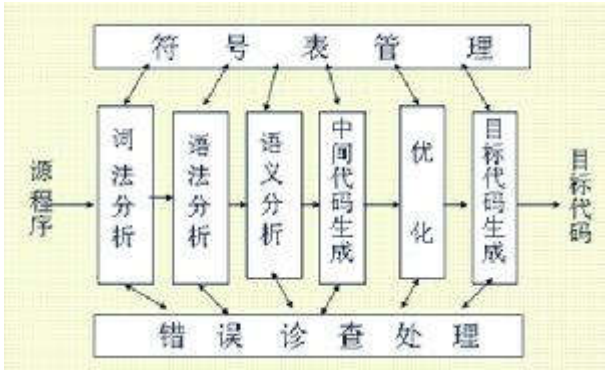
	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler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


Address
Contents
Begin execution here
i
i + 4
i + 8
i + 12
Load R2, A
Load R3, B
Add R4, R2, R3
Store R4, C
4-instruction program segment
A
B
C
Data for the program

Program Development Process



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



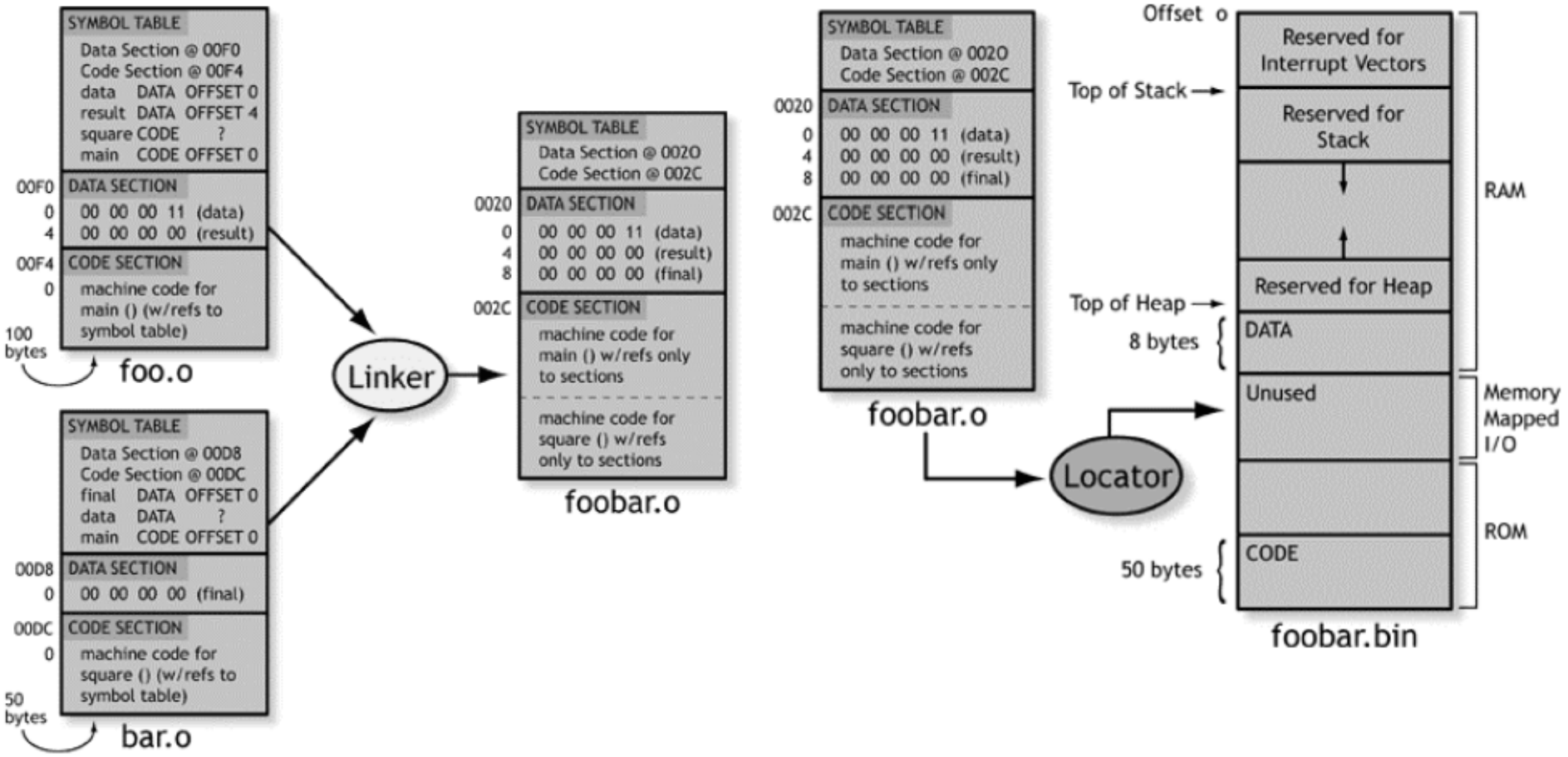
The Assembly Process

- 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

The Linker

- Combines object files into object program
  - 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

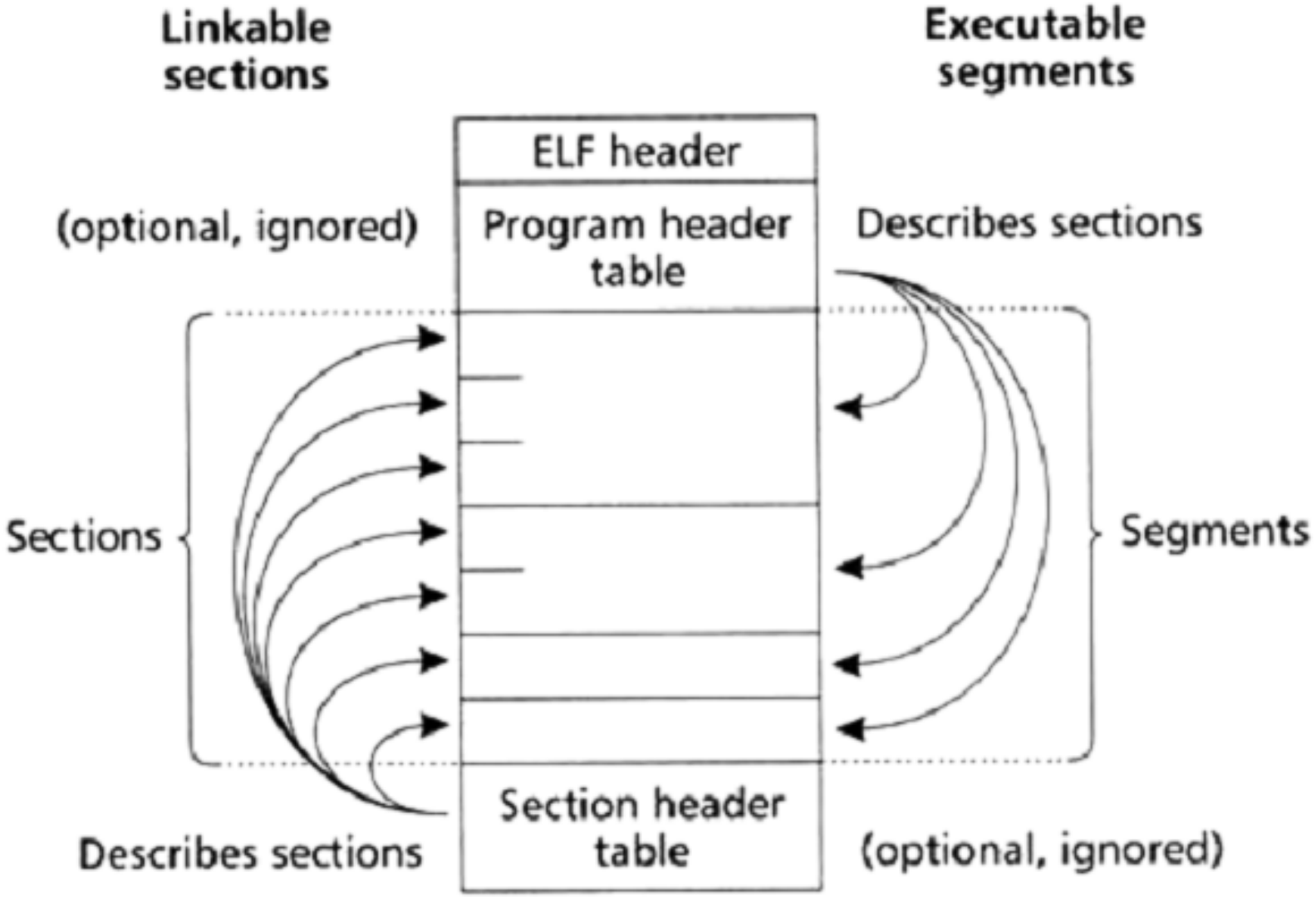
Linking and Locating



Loading/Executing Object Programs

- Loader invoked when user types command or clicks on icon in graphical user interface
  - *Object file* has information on starting location in memory and length of program
- Loader transfers object program from disk to memory and branches to starting address
- At program termination, loader recovers space in memory and awaits next command

ELF 格式目标文件结构图



#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!

ET：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”

小结

- 作业：
  - 2.10，2.13，2.21.1
- 思考（选一）
  - CPU的ISA要定义哪些内容？
    - 见Yale Patt附录A
  - 8086为什么要采用段式内存管理模式？
  - Windows系统中可执行程序的格式？
- 实验报告：
  - 基于x86或MIPS汇编，设计一个冒泡排序程序，并用Debug工具调试执行。
  - 测量冒泡排序程序的执行时间。

Bubble sort (trace)

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

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

Basic idea:

① $j \leftarrow n-1$ (index of last element in A)
② If $A[j] < A[j-1]$, swap both elements
③ $j \leftarrow j-1$, goto ② if $j > 0$
④ Goto ① if a swap occurred

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

④ Swap occured? (Yes, goto ①)

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