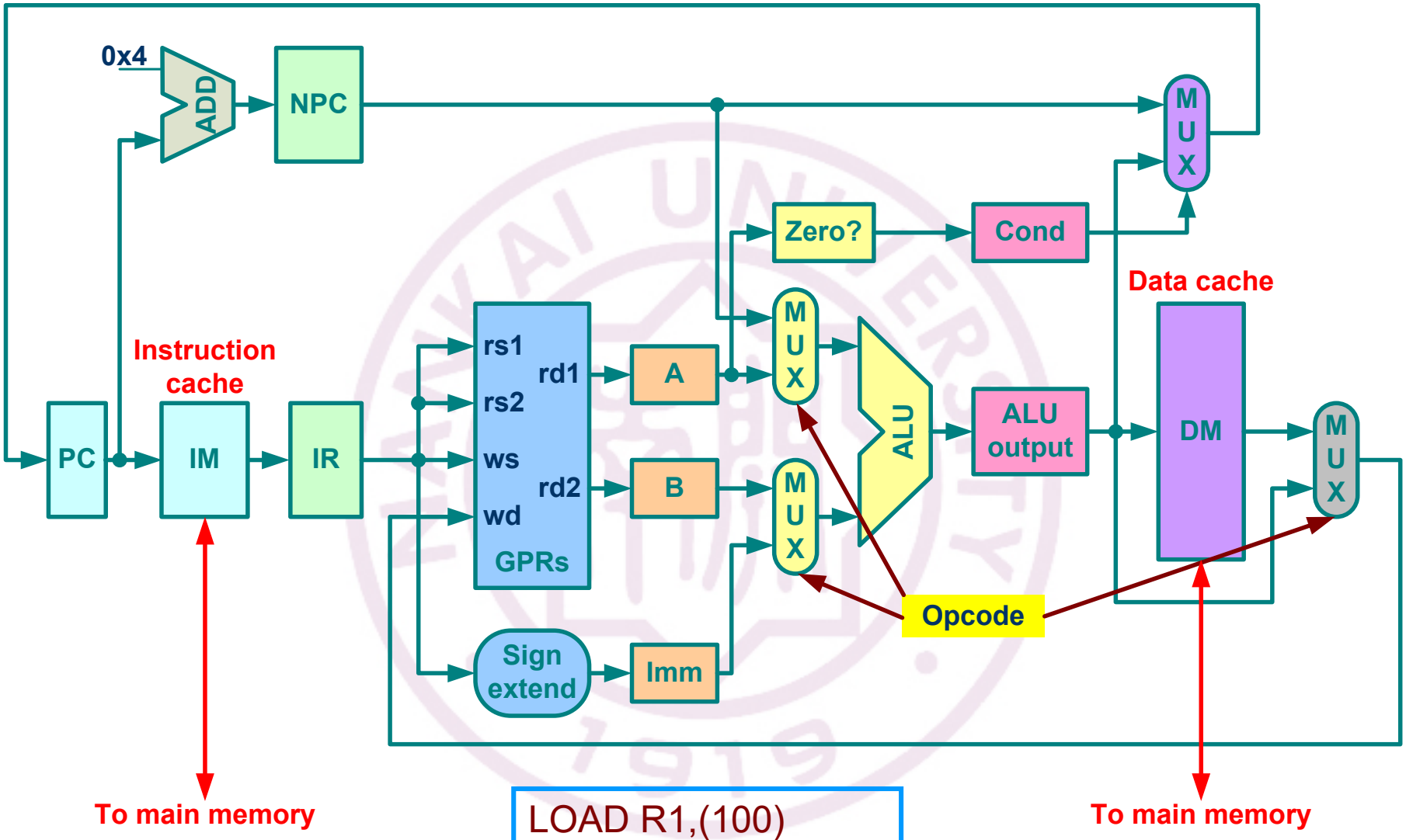
指令集体系结构

机器指令

- 操作码(opcode)
  - 操作内容(ADD, MULT)
  - 操作数类型(INT, FP)
- 操作数(operand)
  - 位置（寄存器或存储器）

```
LOAD R1,(100)
ADD R5,R1,R4
BZ R5,540
```

指令运行



LOAD R1,(100)
ADD R5,R1,R4
BZ R5,540

定义

- 定义：**Instruction Set Architecture** is a structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (time independent) program for that machine. [IBM, Introducing the IBM 360, 1964]
- 是软件和硬件之间的边界（接口）
- 是程序员或编译器所“见到”的计算机
- 在某种意义上，是由一组所使用的汇编语言及它们所完成的工作来定义的

基本内容

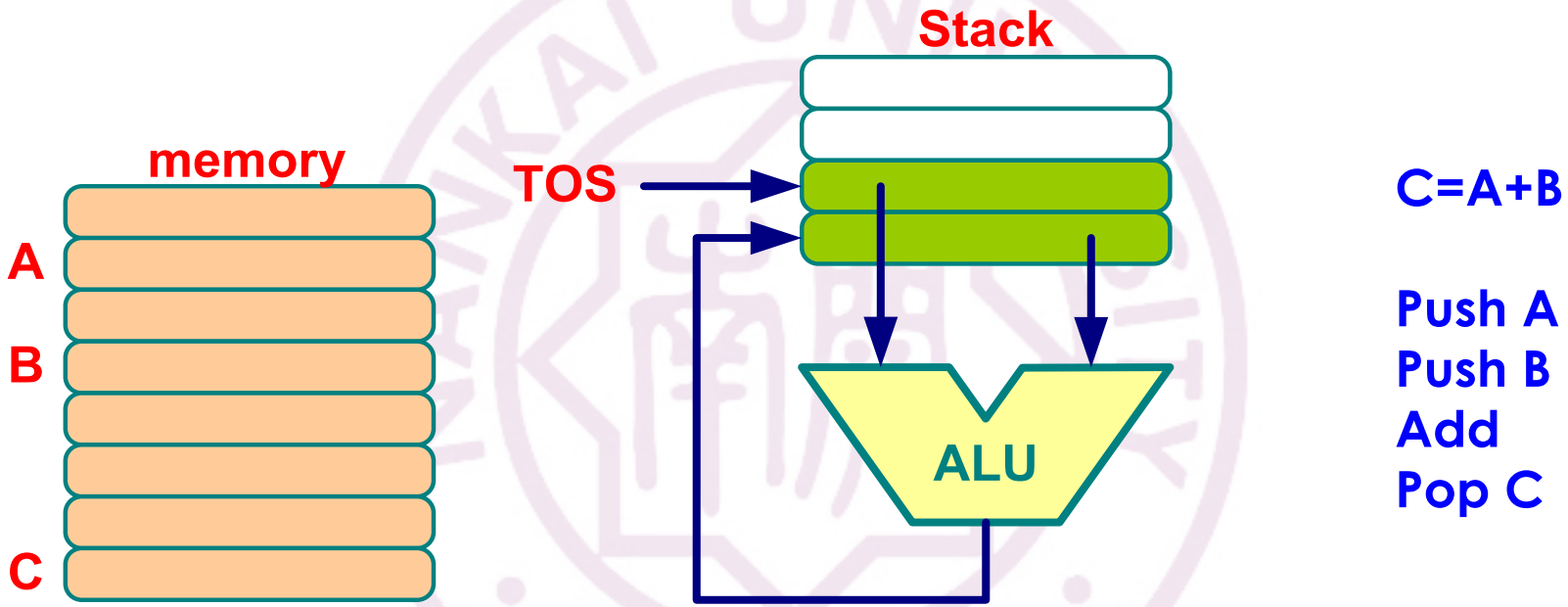
- 寄存器
- 存储器寻址和寻址方式
- 指令操作数
- 有效操作
- 控制流指令
- 指令编码
- ……

设计目标

- 较短的指令宽度
  - 最小化指令宽度，从而最小化程序长度
- 较好的指令密度
  - 最小化指令数目，从而最小化程序长度
- 快速的指令操作
  - 如：ADD (100),(200),(300) 的执行时间比 ADD R1,R2,R3 要长
- 简单的电路实现
- 优化的编译器

类型

- 堆栈型(Stack)
  - 所有操作数都来自堆栈

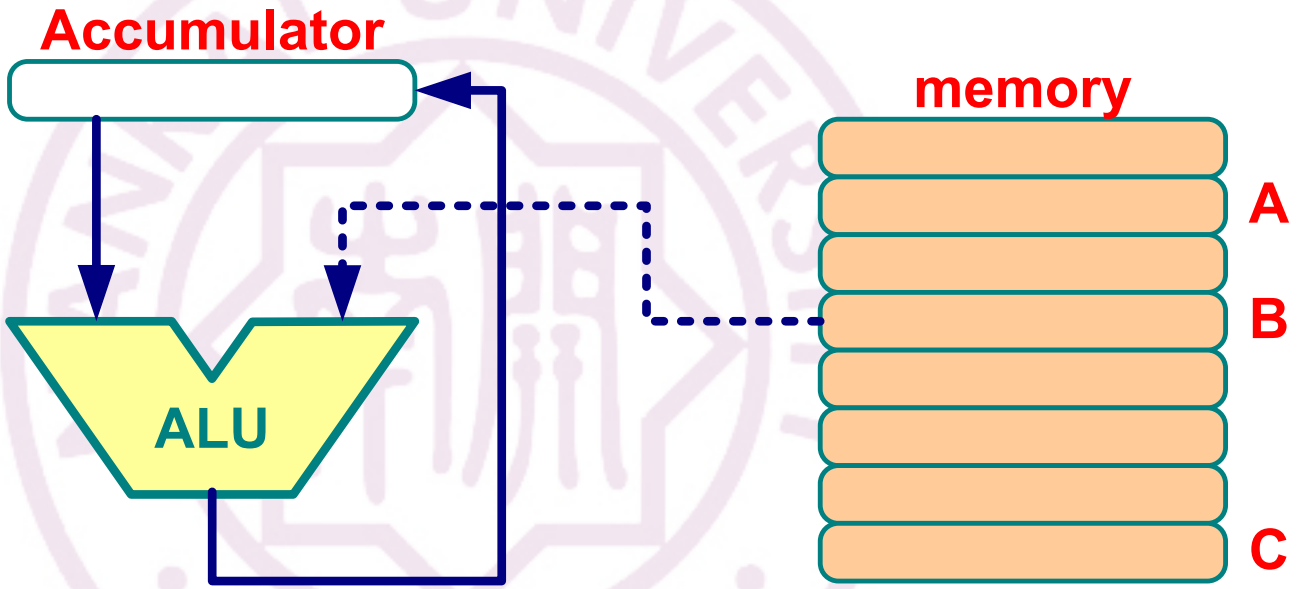


类型

- 累加器型(Accumulator)
  - 一个操作数在累加器中，另一个操作数必须显式地引用

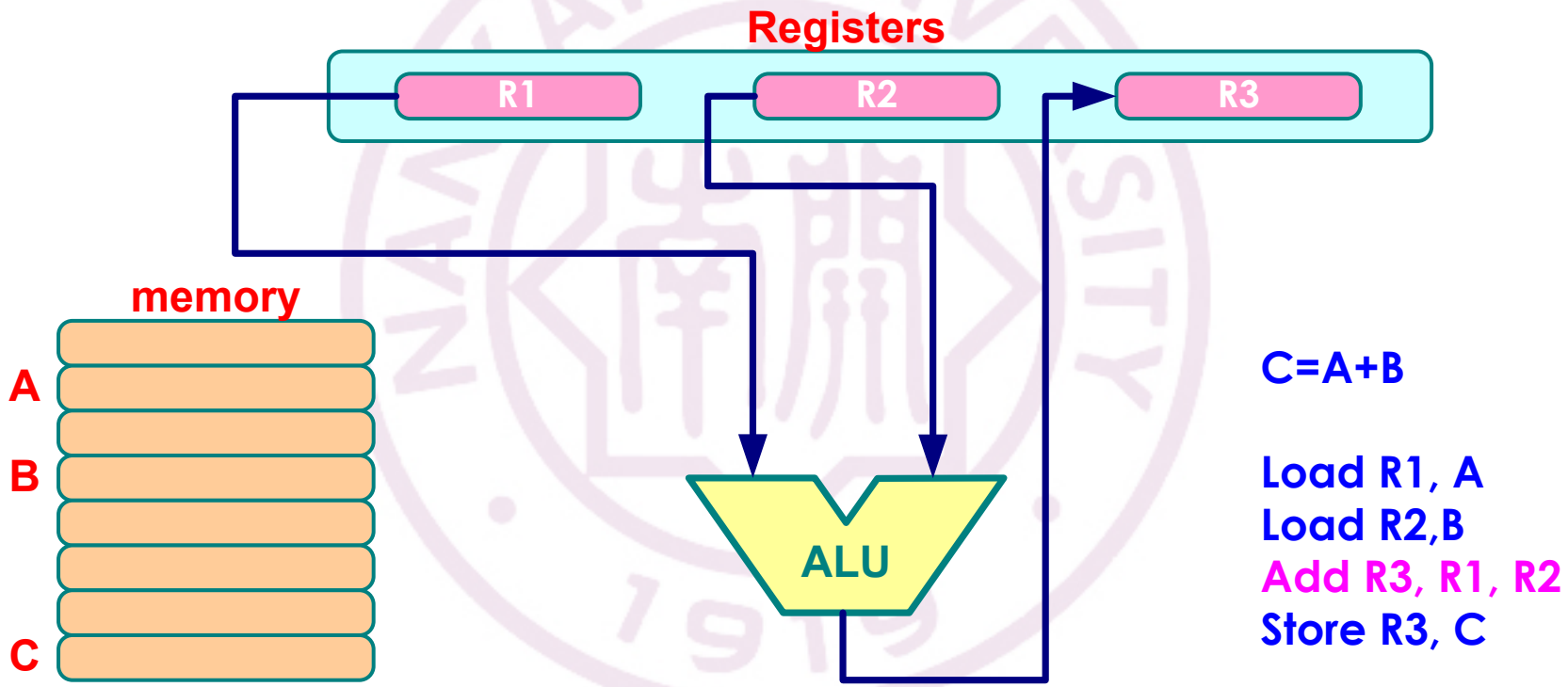
C=A+B

Load A
Add B
Store C



类型

- 寄存器型(Register)
  - 所有操作数必须显式地引用
  - 操作数可以存储在寄存器或存储器中



寄存器型指令集体系结构

- 操作具有明显的操作数形式
  - 寄存器组在CPU内部
- 指令集体系结构提供
  - LOAD：从存储器向寄存器装入数据
  - STORE：从寄存器向存储器存储数据
- 这种体系结构也称为通用寄存器(General-Purpose Register, GPR)指令集体系结构
- 可分为三种类型
  - 寄存器-存储器 ISA
  - 寄存器-寄存器 ISA（装入-存储 ISA）——现代CPU常用的架构
  - 存储器-存储器 ISA —— 废弃

寄存器-寄存器 ISA

- 特殊指令访问存储器
  - LOAD，STORE
- 基本运算步骤
  - 从存储器中装入第一个操作数到寄存器中
  - 从存储器中装入第二个操作数到寄存器中
  - 在寄存器中完成操作
  - 如果需要，将操作结果存储到存储器中
- 两个操作数和结果的位置都需要显式地引用

寄存器数目

- 现代指令集体系结构倾向于拥有更多的寄存器
- 拥有更多的寄存器使得编译器拥有更多的机会来提高效率
- 通常，编译器预留一些寄存器用于特殊目的
  - 保留函数调用的参数
  - 帮助表达式求值
- 剩余的寄存器用于保存程序变量

存储器寻址

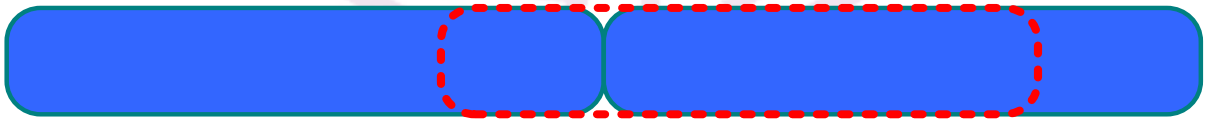
- ISA 必须定义存储器寻址方式
  - 如何解释
  - 如何在指令中说明
- 绝大多数存储器寻址采用字节地址
  - 能够寻址的最小存储器片段是一个字节
  - 意味着每个字节拥有一个使用二进制串表示的地址
- 通常，ISA 可以访问
  - 字节 （8 比特）
  - 半字 （16 比特）
  - 字 （32 比特）
  - 双字 （64 比特）

存储对齐

- 当大多数计算机访问大于一个字节的对象时，往往必须对齐。
- 定义：如果访问对象地址是对象长度 S 的倍数时，称为访问对象是对齐的。

$$(object's\ address)\ mod\ S = 0$$

- 例：
  - 在地址 000000 访问 1 个字节：对齐
  - 在地址 000001 访问 1 个半字：未对齐
  - 在地址 000010 访问 1 个半字：对齐
  - 在地址 011000 访问 1 个双字：对齐
- 目的在于简化硬件实现
  - 访问未对齐存储对象将增加存储器访问次数



字节顺序

- 在半字、字和双字内的字节存在着两种约定

  - 小端法(little endian)：位于地址 x…x000 的字节放在最低有效位置，如Pentium。

  - 大端法(big endian)：位于地址 x…x000 的字节放在最高有效位置，如SPARC。

- 某些体系结构可以通过设置来确定采用哪种字节顺序(如IA64)
- 在绝大多数情况下，不必关心字节顺序问题。但是，
  - 当以双字和字节方式访问相同存储位置时，可能出现问题；
  - 当在计算机之间交换数据时，可能出现问题。

寻址方式

- 寄存器寻址
- 立即数寻址
- 位移寻址
- 寄存器间接寻址
- 索引寻址
- 直接（绝对）寻址
- 存储器间接寻址
- 自动增量寻址
- 自动减量寻址
- 比例寻址

寄存器寻址

Addressing mode	**Register**
Example instruction	**Add R4,R3**
Meaning	**Regs[R4]←Regs[R4]+Regs[R3]**
When used	When a value is in a register.

立即数寻址

Addressing mode	**Immediate**
Example instruction	**Add R4,#3**
Meaning	**Regs[R4]←Regs[R4]+3**
When used	For constants.

位移寻址

Addressing mode	**Displacement**
Example instruction	**Add R4,100(R1)**
Meaning	**Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]**
When used	Accessing local variables (+ simulate register indirect, direct addressing modes).

寄存器间接寻址

Addressing mode	**Register indirect**
Example instruction	**Add R4,(R1)**
Meaning	**Regs[R4]←Regs[R4]+Mem[Regs[R1]]**
When used	Accessing using a pointer or a computed address.

索引寻址

Addressing mode	**Indexed**
Example instruction	**Add R3,(R1+R2)**
Meaning	**Regs[R3]←Regs[R3]** **+Mem[Regs[R1]+Regs[R2]]**
When used	Something useful in array addressing: R1= base of array R2= index amount

直接（绝对）寻址

Addressing mode	**Direct (absolute)**
Example instruction	**Add R1,(1001)**
Meaning	**Regs[R1]←Regs[R1]+Mem[1001]**
When used	Sometimes useful for accessing static data; address constant may need to be large.

存储器间接寻址

Addressing mode	**Memory indirect**
Example instruction	**Add R1,@(R3)**
Meaning	**Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]**
When used	If R3 is the address of a pointer *p*, then mode yields **p*.

自动增量寻址

Addressing mode	**Auto-increment**
Example instruction	**Add R1,(R2)+**
Meaning	**Regs[R1]←Regs[R1]+Mem[Regs[R2]]** **Regs[R2]←Regs[R2]+d**
When used	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.

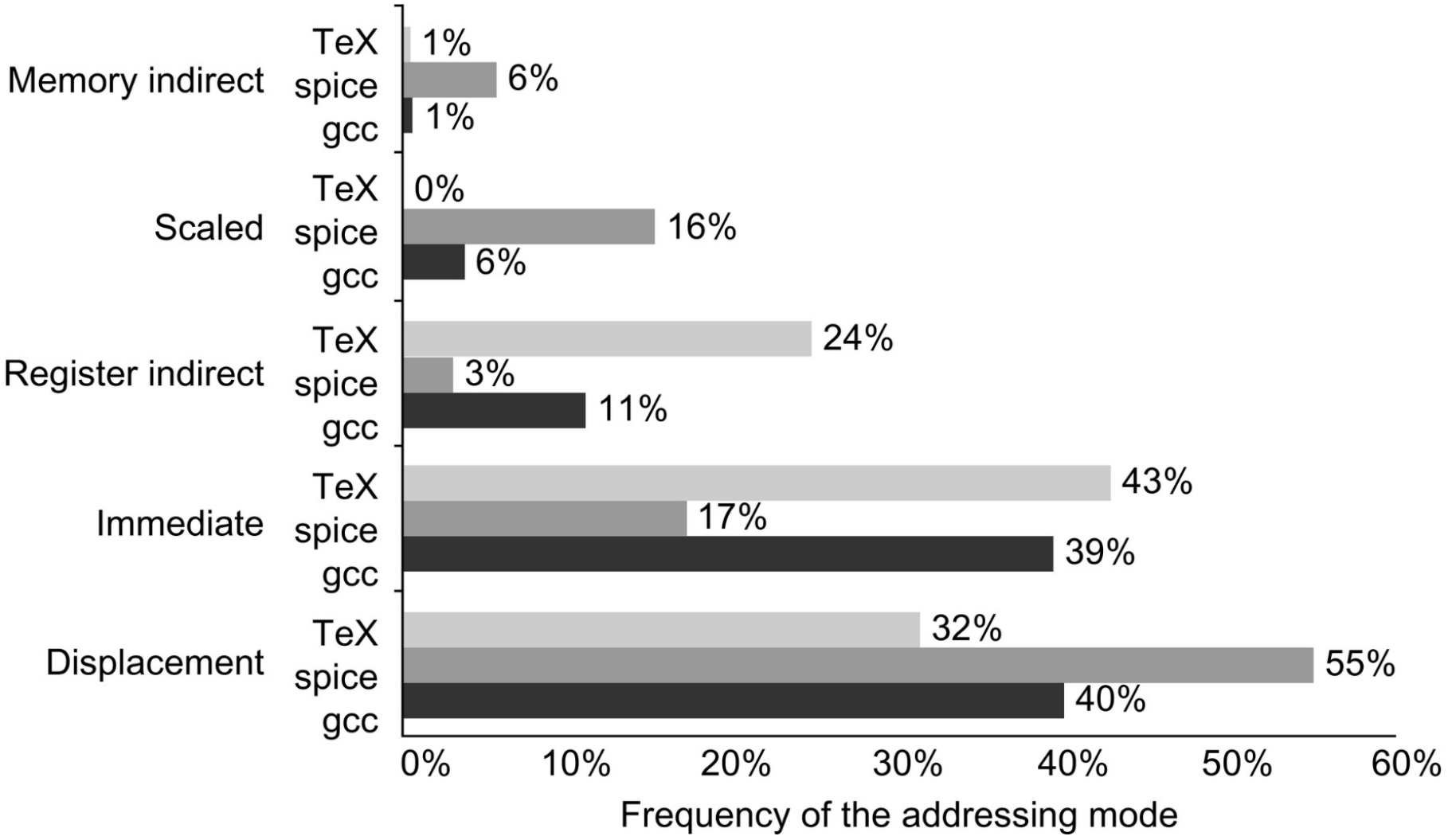
自动减量寻址

Addressing mode	**Auto-decrement**
Example instruction	**Add R1,-(R2)**
Meaning	**Regs[R2]←Regs[R2]-d** **Regs[R1]←Regs[R1]+Mem[Regs[R2]]**
When used	Same use as auto-increment. Auto-decrement/-increment can also act as push/pop to implement a stack.

比例寻址

Addressing mode	**Scaled**
Example instruction	**Add R1,100(R2)[R3]**
Meaning	**Regs[R1]←Regs[R1]** **+Mem[100+Regs[R2]+Regs[R3]*d]**
When used	Used to index arrays. May be applied to any indexed addressing mode in some computers.

基准程序中各种寻址方式的频度

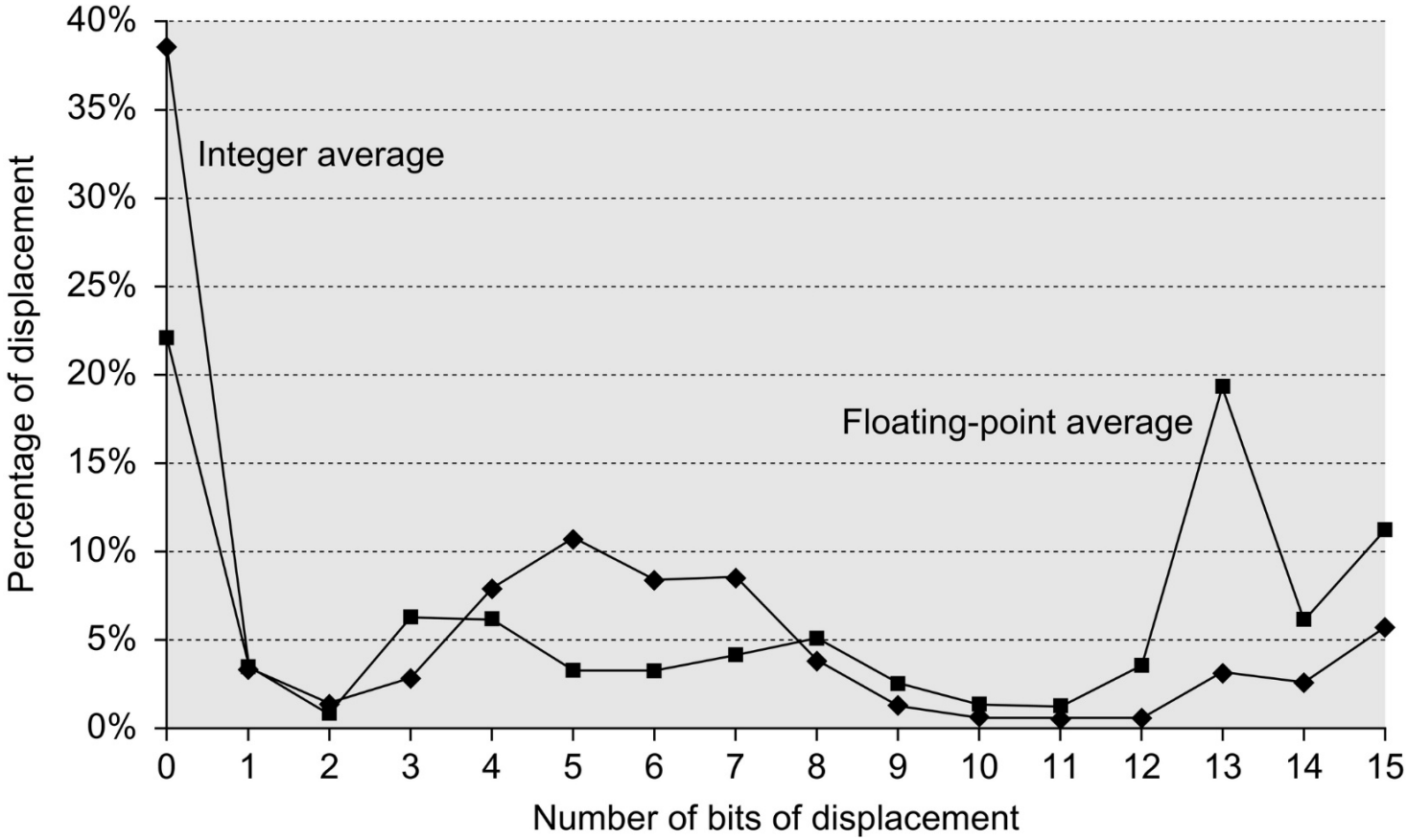


选择寻址方式的思考：平衡

- 寻址方式的选择
  - 一些 ISA 选择使用所有寻址方式，如 VAX
  - 另一些 ISA 只使用少数几种寻址方式，如 MIPS。
- 平衡因素
  - 寻址方式越多，IC 越低；
  - 寻址方式越多，硬件越复杂；
  - 寻址方式越多，CPI 越高。
- 目标：平衡
  - 简化硬件
  - 拥有足够的寻址方式，以至于书写代码不会太“痛苦”。

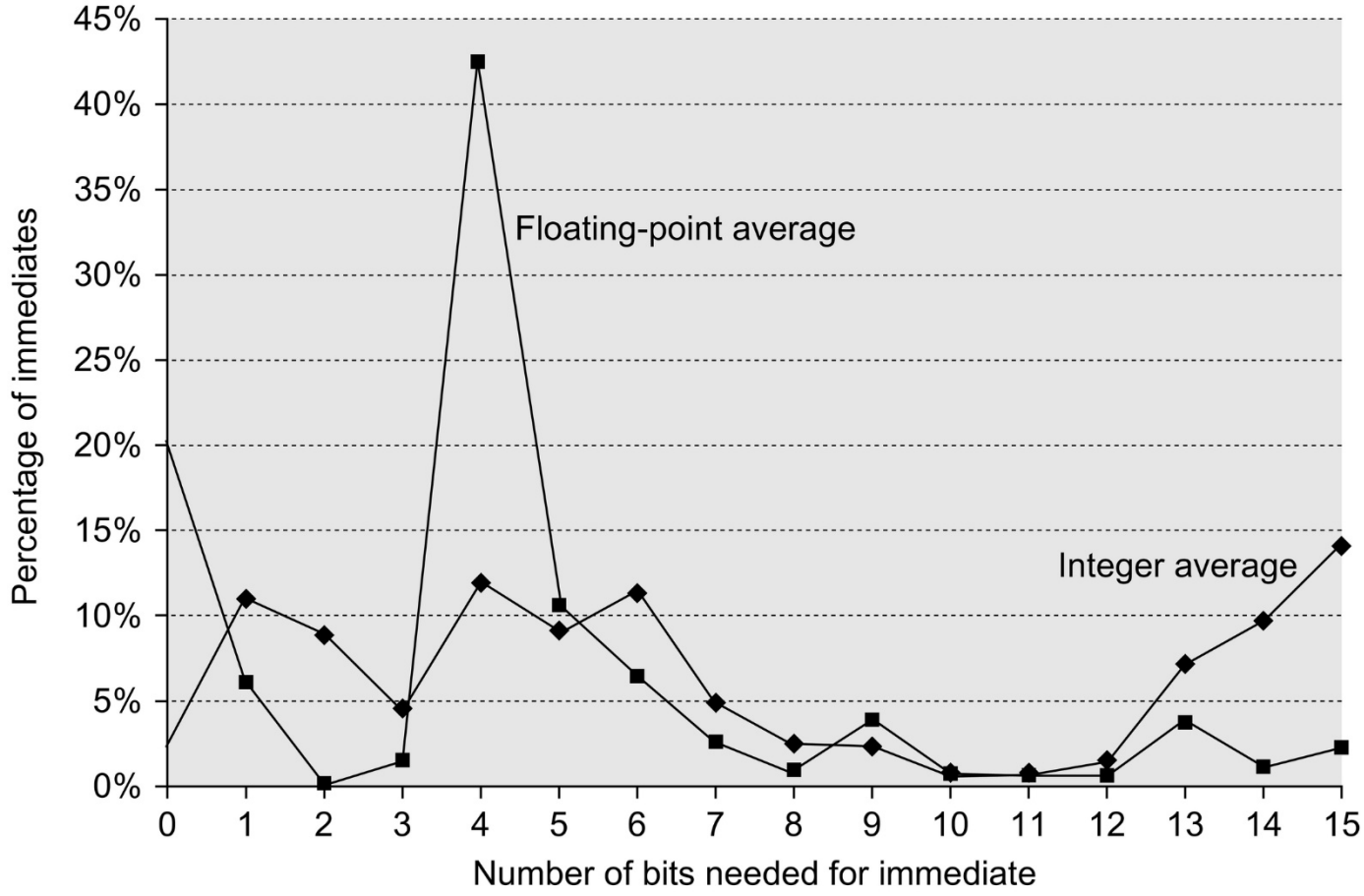
选择位移范围

- 权衡考虑
  - 位移最大值越大，指令使用范围越广；
  - 位移最大值越小，指令编码越短。



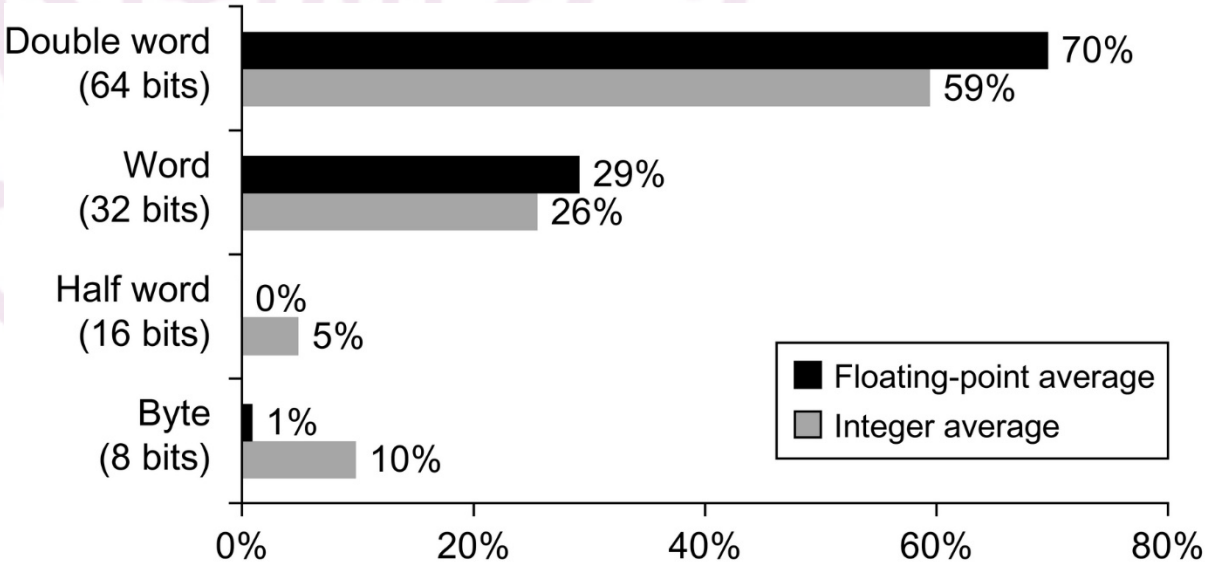
选择立即数范围

- 权衡考虑
  - 立即数最大值越大，指令使用范围越宽；
  - 立即数最大值越小，指令编码越短。



操作数类型和大小

- 字符：1 个字节
- 半字（短整数）：2 个字节
- 字（整数）：4 个字节
- 单精度浮点数和长整数：4 个字节
- 双精度浮点数：8 个字节
- 压缩十进制数(BCD)（部分支持）：每个字节两位数字



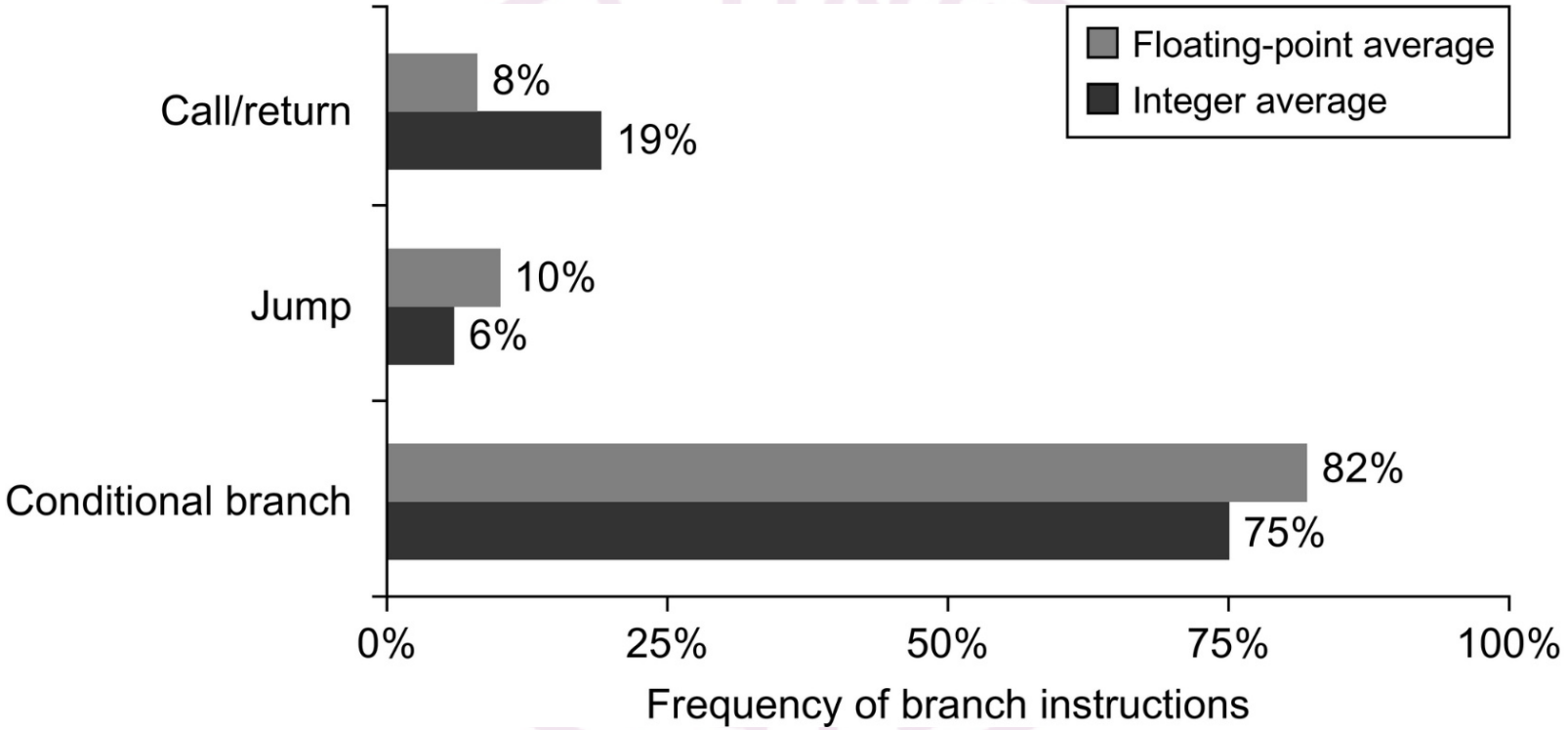
各种类型操作

- 算术和逻辑（通用）
  - 整数加法、减法、乘法和除法，与或运算
- 数据传输（通用）
  - 装入，存储
- 控制（通用）
  - 分支，跳转，过程调用和返回，陷阱
- 系统（可变）
  - 操作系统调用，管理虚拟内存指令
- 浮点数（可选）
  - 加法，乘法，除法，比较
- 字符串（可选）
  - 字符串比较，字符串搜索

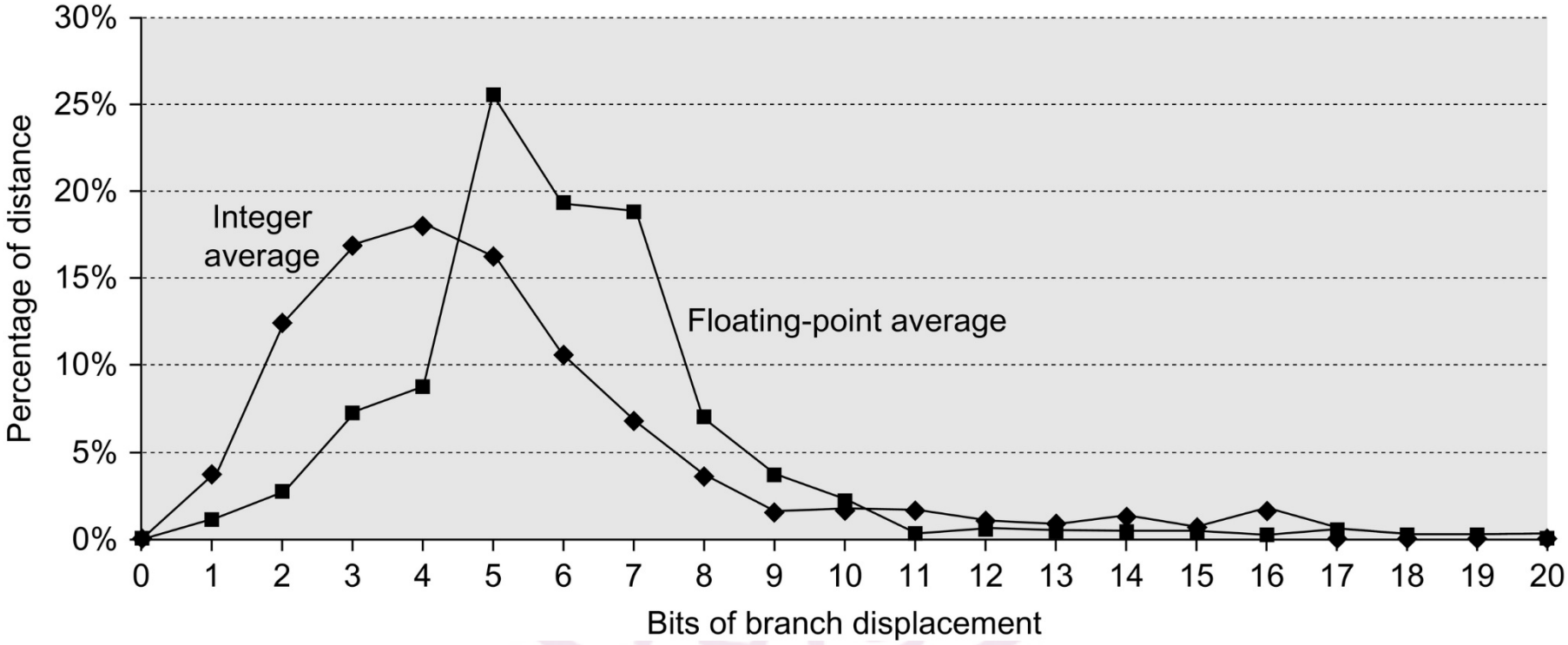
各种类型操作

- 图形（可选）
  - 像素点和顶点操作，压缩与解压操作

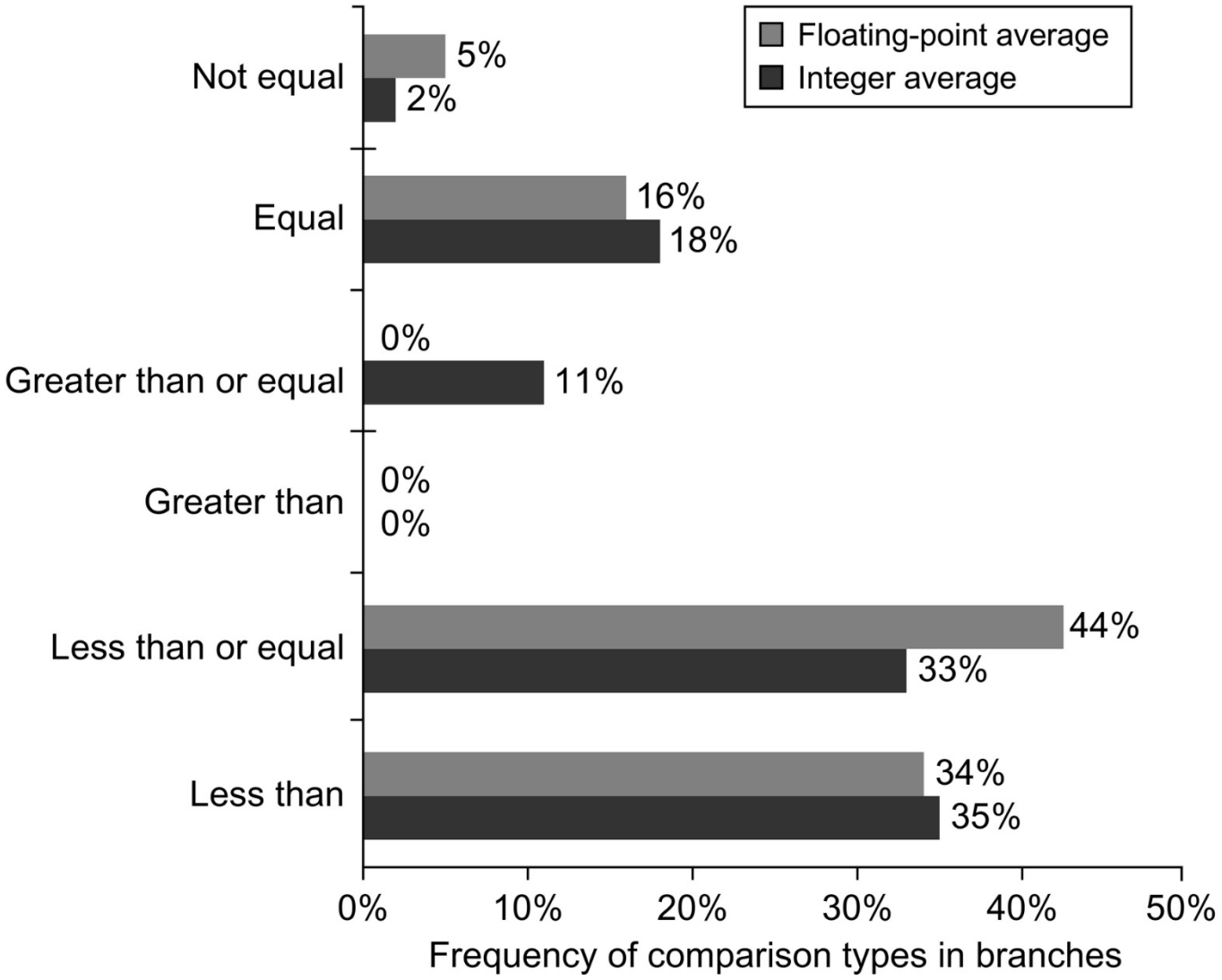
控制流指令的频度


Floating-point average
Integer average
Call/return
8%
19%
Jump
10%
6%
Conditional branch
82%
75%
0%
25%
50%
75%
100%
Frequency of branch instructions

相对转移的偏移量


30%
25%
20%
15%
10%
5%
0%
Percentage of distance
Integer average
Floating-point average
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Bits of branch displacement

各种比较操作的频度

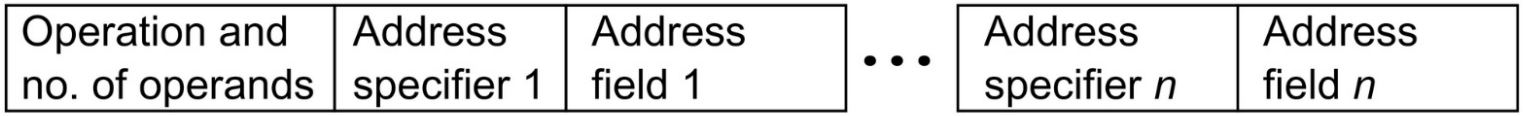

Floating-point average
Integer average
Not equal
5%
2%
Equal
16%
18%
Greater than or equal
0%
11%
Greater than
0%
0%
Less than or equal
44%
33%
Less than
34%
35%
0%
10%
20%
30%
40%
50%
Frequency of comparison types in branches

指令编码

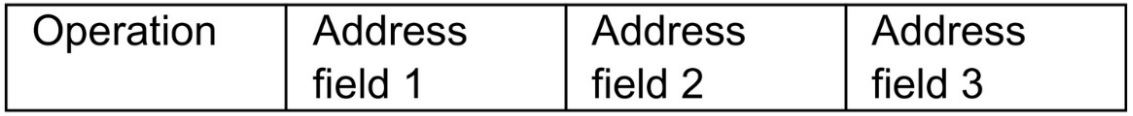
- 关键考虑因素
  - 期望拥有尽可能多的寄存器和寻址方式，会造成较长的指令代码
  - 期望降低平均指令代码长度和平均程序长度
  - 期望指令能够容易且快速译码

- 指令编码方法
  - 可变格式（满足前两个期望）
  - 固定格式（满足第三个期望）
  - 混合格式（尝试达到平衡）

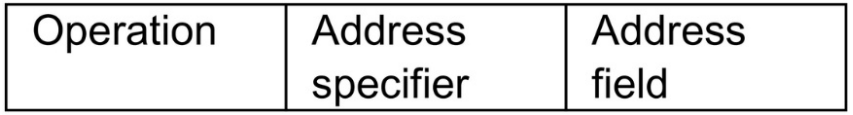
指令编码

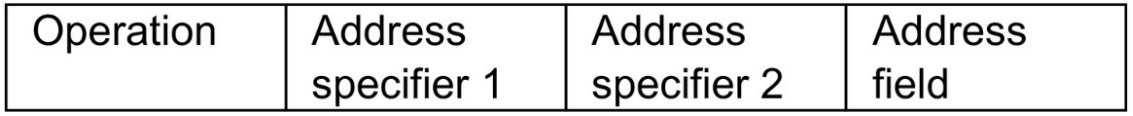


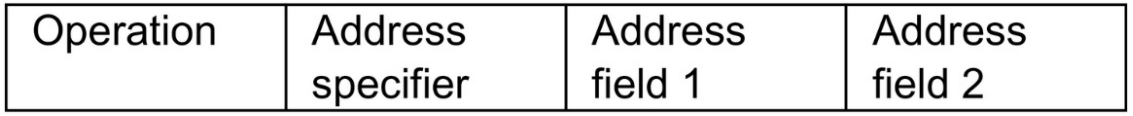
(A) Variable (e.g., Intel 80x86, VAX)



(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)







(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

编译优化

- 高层优化
  - 针对源代码进行，可能是源到源的转换；
  - 如：为了Cache有效性来映射数据，除去条件等
- 局部优化
  - 在较短的直线段源代码中优化代码
- 全局优化
  - 跨过分支指令进行优化，循环优化（循环展开）
- 寄存器分配
  - 寄存器赋初值

CISC vs. RISC

- 谬误：“简约指令集”并不意味着指令数目较少
- RISC 的关键
  - 在寄存器中完成操作
  - 使用 LOAD 和 STORE 指令完成寄存器和存储器之间的通信
  - 代码是由一些列这样简单操作实现的
- 注：当代码量是关键因素时，RISC 并不是一个好的选择。

RISC-V 指令集体系结构

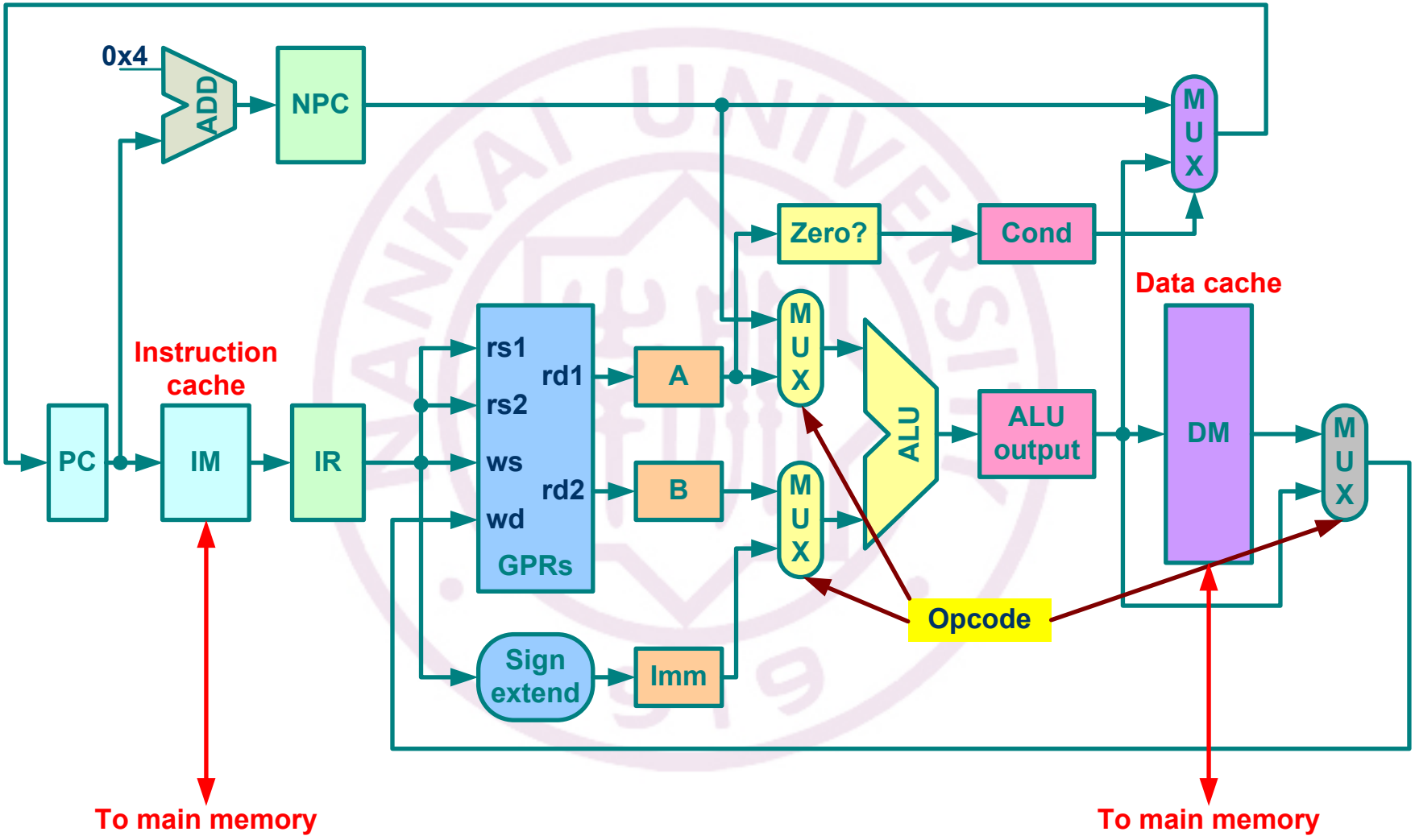
- RISC-V 指令集体系结构
  - 加州大学伯克利分校开发的最新开源 RISC 指令集
  - 特点：大量的寄存器，容易流水操作的指令，精干的操作等
- RISC-V 寄存器
  - 32 个通用寄存器，32 个浮点数寄存器

Reg.	Name	Use	Saver
x0	zero	constant 0	n/a
x1	ra	return addr.	caller
x2	sp	stack ptr.	callee
x3	gp	gbl. ptr.	
x4	tp	thread ptr.	
x5-x7	t0-t2	Temporaries	caller
x8	s0/fp	saved/frame ptr.	callee
x9	s1	saved	callee

Reg.	Name	Use	Saver
x10-x17	a0-a7	arguments	caller
x18-x27	s2-s11	saved	callee
x28-x31	t3-t6	temporaries	caller
f0-f7	ft0-ft7	FP temps	caller
f8-f9	fs0-fs1	FP saved	callee
f10-f17	fa0-fa7	FP arguments	callee
f18-f27	fs2-fs21	FP saved	callee
f28-f31	ft8-ft11	FP temps	caller

RISC-V ISA

- 通用寄存器体系结构(general-purpose register architecture)



RISC-V ISA

- 寻址方式
  - 寄存器
  - 立即数（常数）
  - 位移（常数偏移量 + 寄存器内容）

- 操作数类型和大小
  - ASCII 字符：8 比特
  - Unicode 字符或半字：16 比特
  - 整数或字：32 比特
  - 双字或长整数：64 比特
  - 单精度浮点数(IEEE 754)：32 比特
  - 双精度浮点数(IEEE 754)：64 比特

RISC-V ISA

- 操作类型
  - 算术和逻辑
  - 数据传输
  - 控制
  - 浮点数
- 控制流指令
  - 条件分支（条件判断使用寄存器内容而不是状态位）
  - 无条件转移
  - 过程调用和返回（返回地址在寄存器中而不是在堆栈中）
- 指令编码
  - 32 比特固定长度编码方式

诚信 创新 实践

讨论题

- 问题：为什么当前 CPU 均采用 load-store ISA 架构?

讨论题

- 问题：为什么当前 CPU 均使用 load-store ISA?
- 答案：
  - 寄存器访问较快
  - 相比存储器来说，寄存器可以使用较短比特来命名
  - 寄存器允许编辑优化（如乱序执行）
  - 寄存器能够用来保存与指定代码段有关的所有变量

讨论题

Architectures	Advantages	Disadvantages
Register-register	1) Simple 2) Fixed-length instruction 3) Similar CPI 4) Compiler optimizations	1) Higher instruction count 2) Longer programs
Register-memory	Better instruction density	1) Source operand is destroyed 2) Longer instructions 3) CPI vary by operand location
Memory-memory	Better instruction density	1) Longest instructions 2) CPI vary by operand location 3) Memory bottleneck

讨论题

Architectures	Advantages	Disadvantages
Stack	1) Simple effective address 2) Short instructions 3) Good code density 4) Simple I-decode	1) Lake of random access 2) Efficient code is difficult to generate 3) Stack is often a bottleneck
Accumulator	1) Minimal internal state 2) Fast context switch 3) Short instructions 4) Simple I-decode	Very high memory traffic
Register	1) Lots of code generation options. 2) Efficient code since compiler has numerous useful options.	1) Longer instructions 2) Possibly complex effective address generation 3) Size and structure of register set has many options.