汇编语言程序设计

第12讲 MIPS 32处理器结构与指令集初步

MIPS架构——最经典的RISC

MIPS的由来与发展

Microprocessor without Interlocked Pipeline Stages
(Millions of Instructions Per Second的双关语)

尽量利用软件办法避免流水线中的数据相关问题

1981年，斯坦福大学教授John Hennessy（“RISC之父”）领导团队，设计出第一个MIPS架构的处理器

2017图灵奖授予John Hennessy与David Patterson，表彰他们对RISC微处理器的贡献

1984年， Hennessy教授离开斯坦福大学，创立MIPS科技公司

- 于1985年设计出R2000芯片；于1988年将其改进为R3000芯片；于1991年设计出R4000
- 陆续推出R8000（于1994年）、R10000（于1996年）和R12000（于1997年）等型号
- 后重点转向嵌入式领域——2000年，MIPS公司发布了针对MIPS32 4Kc的版本以及64位MIPS 64 20Kc处理器内核

MIPS处理器是八十年代中期RISC CPU设计的一大热点：

- 在许多领域，如Sony、Nintendo的游戏机，Cisco的路由器和SGI超级计算机中使用

通用处理器指令体系历经MIPS I、MIPS II、MIPS III、MIPS IV到MIPS V的发展；嵌入式指令体系历经MIPS16、MIPS32到MIPS64的发展，十分成熟

- 2012年，MIPS公司被Imagination与AST拆分收购
- MIPS架构具有非常重要的基础专利，应用广泛

在设计理念上MIPS强调软硬件协同提高性能，同时简化硬件设计

中国的龙芯采用的是MIPS架构

- 清华大学于2002-2004年期间也开发了基于MIPS32的自主微处理器芯片THUMP107 / 105系列

以下5张关于RISC的PPT摘自2018年图灵演讲*——

A New Golden Age for Computer Architecture：

Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development

John Hennessy and David Patterson

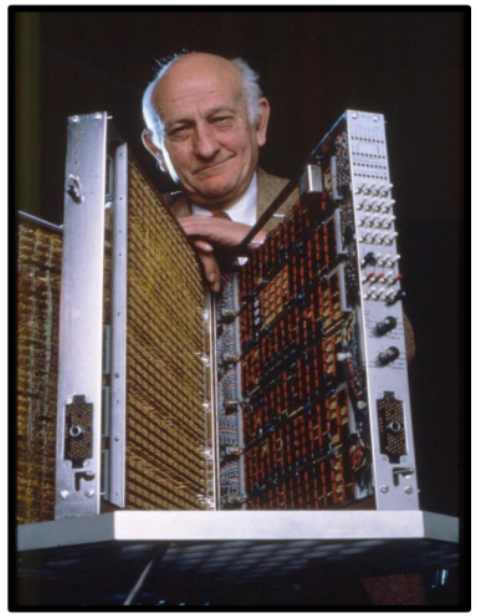
Stanford and UC Berkeley

June 4, 2018

*https://iscaconf.org/isca2018/docs/HennessyPattersonTuringLectureISCA4June2018.pdf

Analyzing Microcoded Machines 1980s

- World changed to HLL programming from assembly
  - Compilers now source of measurements
- John Cocke group at IBM
  - Worked on a simple pipelined processor, 801 minicomputer (ECL server), and advanced compilers inside IBM
  - Ported their compiler to IBM 370, only used simple register-register and load/store instructions (similar to 801)
  - Up to 3X faster than existing compilers that used full 370 ISA!
- Emer and Clark at DEC in early 1980s*
  - Found VAX 11/180 average clock cycles per instruction (CPI) = 10!
  - Found 20% of VAX ISA ⇒ 60% of microcode, but only 0.2% of execution time!
- Patterson after '79 DEC sabbatical: repair microcode bugs in microprocessors?**
  - What's magic about ISA interpreter in Writable Control Store? Why not other programs?



John Cocke

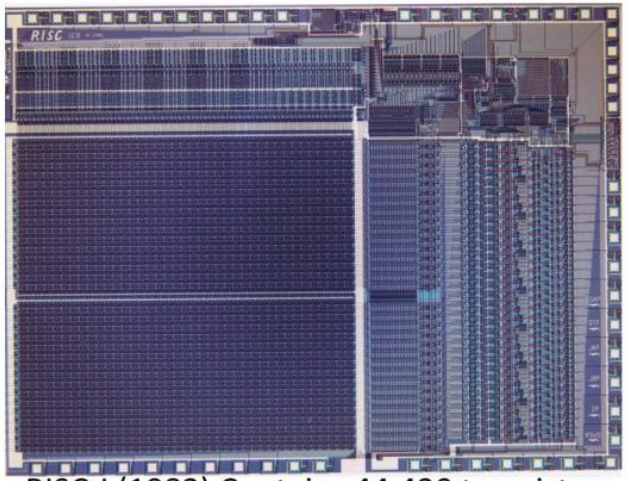
* "A Characterization of Processor Performance in the VAX-11/780," J. Emer and D.Clark, *ISCA*, 1984.
** "RISCy History," David Patterson, May 30, 2018, Computer Architecture Today Blog

From CISC to RISC

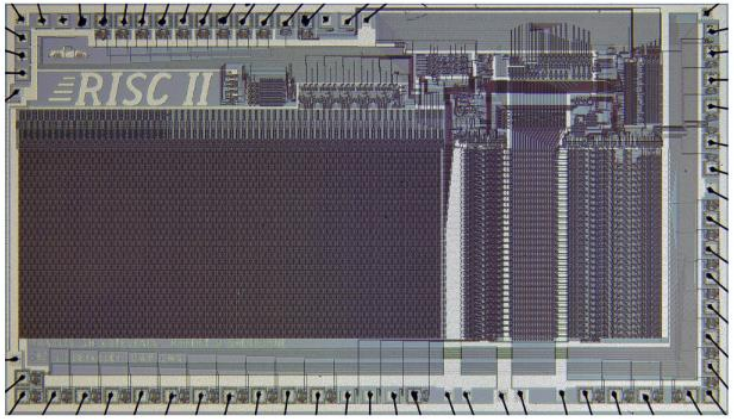
- Use SRAM for instruction *cache* of user-visible instructions
  - Contents of fast instruction memory change to what application needs now vs. ISA interpreter
- Use simple ISA
  - Instructions as simple as microinstructions, but not as wide
  - Compiled code only used a few CISC instructions anyways
  - Enable pipelined implementations
- Further benefit with chip integration
  - In early ‘80s, could finally fit 32-bit datapath + small caches on a single chip
- Chaitin’s register allocation scheme* benefits load-store ISAs

*Chaitin, Gregory J., et al. "Register allocation via coloring." *Computer languages* 6.1 (1981), 47-57.

Berkeley & Stanford RISC Chips



RISC-I (1982) Contains 44,420 transistors, fabbed in 5 µm NMOS, with a die area of 77 mm^2, ran at 1 MHz



RISC-II (1983) contains 40,760 transistors, was fabbed in 3 µm NMOS, ran at 3 MHz, and the size is 60 mm^2

Stanford MIPS (1983) contains 25,000 transistors, was fabbed in 3 µm & 4 µm NMOS, ran at 4 MHz (3 µm), and size is 50 mm^2 (4 µm)
(Microprocessor without Interlocked Pipeline Stages)

Fitzpatrick, Daniel, John Foderaro, Manolis Katevenis, Howard Landman, David Patterson, James Peek, Zvi Peshkess, Carlo Séquin, Robert Sherburne, and Korbin Van Dyke. "A RISCy approach to VLSI." *ACM SIGARCH Computer Architecture News* 10, no. 1 (1982):

Hennessy, John, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. "MIPS: A microprocessor architecture." In *ACM SIGMICRO Newsletter*, vol. 13, no. 4, (1982).

"Iron Law" of Processor Performance: How RISC can win

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Clock cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Clock cycle}}$$

- CISC executes fewer instructions per program

(≈ 3/4X instructions),
but many more clock cycles per instruction
(≈ 6X CPI)

⇒ RISC ≈ 4X faster than CISC

"Performance from architecture: comparing a RISC and a CISC with similar hardware organization," Dileep Bhandarkar and Douglas Clark, *Proc. Symposium, ASPLOS*, 1991.

CISC vs. RISC Today

PC Era

- Hardware translates x86 instructions into internal RISC instructions
- Then use any RISC technique inside MPU
- > 350M / year !
- x86 ISA eventually dominates servers as well as desktops

PostPC Era: Client/Cloud

- IP in SoC vs. MPU
- Value die area, energy as much as performance
- > 20B total / year in 2017
  - x86 in PCs peaks in 2011, now decline ~8% / year (2016 < 2007)
  - x86 servers ⇒ Cloud ~10M servers total* (0.05% of 20B)
- 99% Processors today are RISC

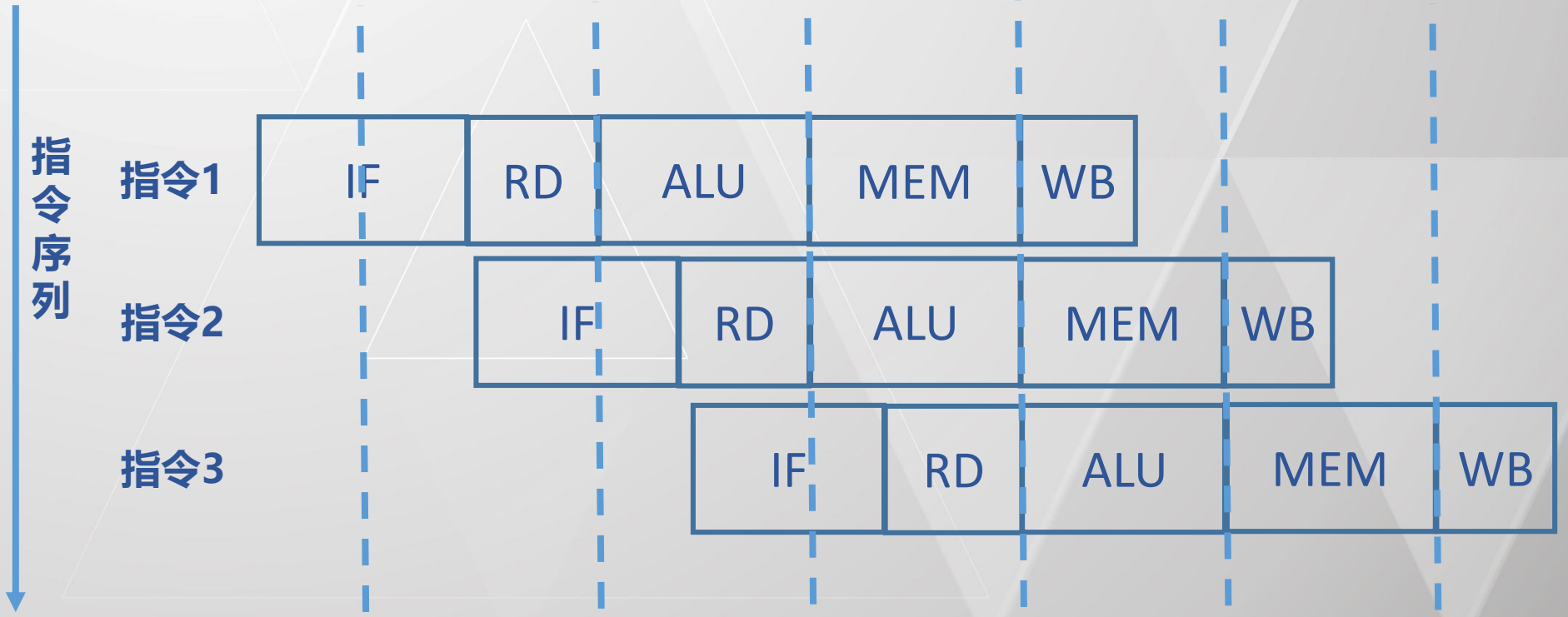
*"A Decade of Mobile Computing", Vijay Reddi, 7/21/17, *Computer Architecture Today*

- **MIPS的流水线结构**
- **指令集特点**
- **汇编指令初步**
  - 包括CP0（Coprocessor 0）与程序地址空间布局(layout)

MIPS的经典流水线结构

五级流水（四个周期）

- Fetch instruction
- Read registers
- Arithmetic operation
- Memory access
- Write back



指令集特点（与X86指令集对比）

*MIPS32结构中的字长度为32bit

	MIPS32	X86
指令长度	固定（32位）	变长
	流水线的“取指”段（IF）时间固定	
	常数字段小于32位（对于无条件跳转指令，立即数方式的目的地址为26位；其它指令的一般为16位）	
指令中的内存操作数	内存操作数无法直接参与运算	内存操作数可参与运算
	指令操作须适应流水线（第四段才是访存）	
计算指令的操作数个数	多为3个	多为2个
	对编译优化有利	

指令集特点（与X86指令集对比）

	MIPS32	X86
通用寄存器个数	32	8 / 16
	0号寄存器的值永远是0。0是最常用的常数，有利于节省指令编码。	64位结构才有16个寄存器
条件码	无	有
	所有信息存于通用寄存器，条件判断通过检测通用寄存器来进行。	
访存操作	只能通过load/Store指令	多数指令支持
	支持双字、字、字节、半字等； 需要地址对齐	无需地址对齐
	一种寻址方式：基址+常数偏移量	多种

指令集特点（与X86指令集对比）

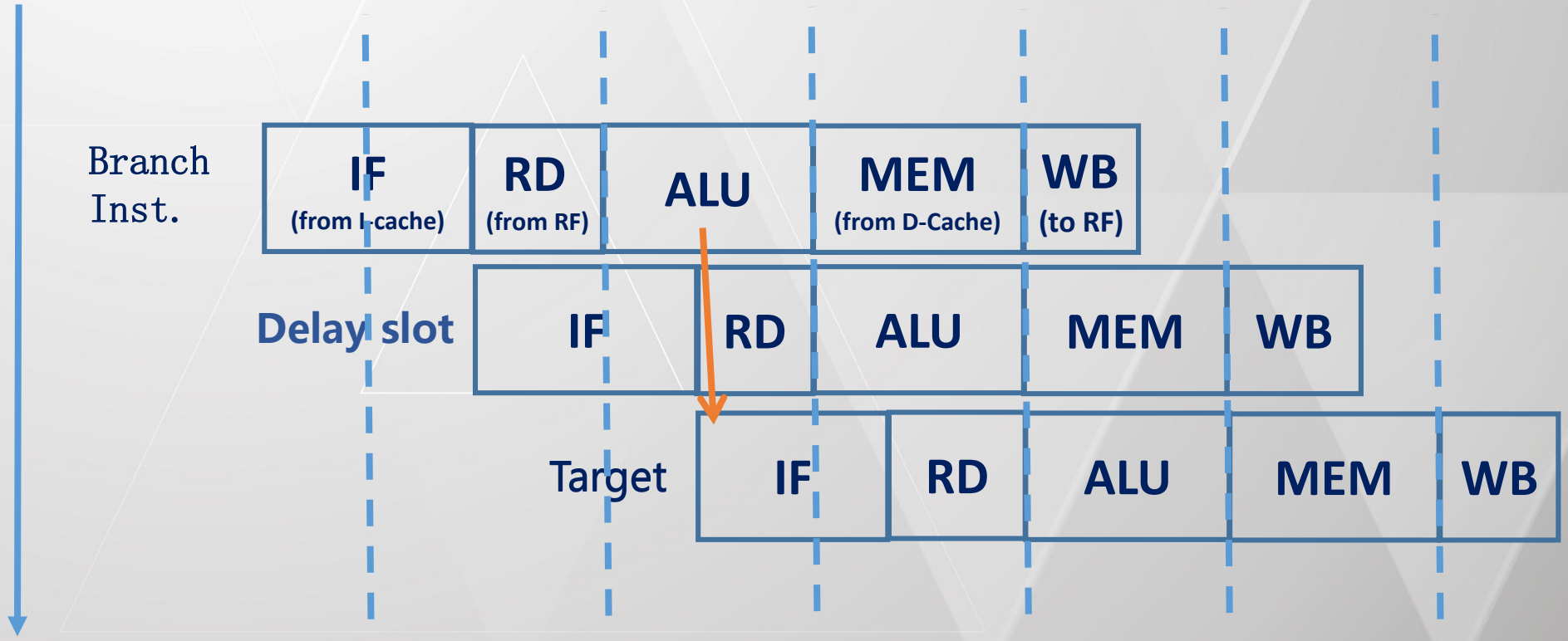
	MIPS	X86
半字或者字节运算	无（软件解决）	有
针对栈操作的支持	无	有专门指令
	一个通用寄存器习惯上作为栈顶地址寄存器	
过程调用指令	返回地址保存到31号寄存器	保存到栈
其它	对于异常的处理主要依赖软件完成	

程序员可见的流水线效果

Branch Delay Slot （跳转延迟）

条件跳转指令的目标地址计算需要在ALU段获得，而此时第二条指令已经进入流水线，存在一个延迟槽（delay slot）

需要程序员或编译器优化来填充这个slot



简单总结

RISC的设计思想在于简化计算机指令功能、规格化指令设计，使得各个指令的流水线分段较为均匀，且操作相对简单规整，从而提高主频

采用Load/Store结构：其它指令均在寄存器之间对数据进行处理，提高处理速度

依赖软件（编译器）实现优化及完成复杂功能

多选题 0.5分

设置

对于MIPS32指令集，以下描述中正确的有：

A 访存指令中的数据地址必须对齐

B 只有load/store指令才能访存

C 有半字或者字节运算指令

D 指令字长度不固定

提交

MIPS汇编指令初步

第一条MIPS指令

```
entrypoint:                    # that's a label
    addu $1, $2, $3            # (registers) $1  $2 + $3
```

三操作数指令形式，目标寄存器在左侧（与AT&T风格相反！）

示例2

```
    ...
    jal printf
    move $4, $6
    xxx # return here after call
```

#调用过程（jal指令类似于X86的call）
#这条指令位于delay slot内，与前一条指令一起执行
#返回地址

访存指令

lw $1, offset($2)　　任何寄存器都可以作为地址或者目标寄存器；
offset　　为16位的带符号整数

C名字	MIPS名字	大小（字节）	汇编助记符
long long	dword	8	ld中的“d”
int long	word	4	lw中的“w”
short	halfword	2	lh中的“h”
char	byte	1	lb中的“b”

支持无符号与带符号扩展

lb　$1, offset($2)　　#mem[offset($2)] = 0xfe；　$1 = 0xfffffffe
lbu $1, offset($2)　　#$1 = ???

寄存器

使用32个通用寄存器

- 0号寄存器的值永远是0
- 31号寄存器存放函数调用的返回地址（JAL指令）
- 其它寄存器都是“一样”的
- 没有指令寄存器（如x86中的eip或rip）

整数乘除法的专用寄存器

- Hi / Lo

32个浮点寄存器（如果有浮点协处理器的话）

MIPS32寄存器命名与使用惯例

寄存器编号	名字	习惯用途
0	zero	值永远是0
1	at	保留给汇编器使用（编程时避免使用）
2-3	v0,v1	过程调用返回值
4-7	a0-a3	传参用寄存器
8-15	t0-t7	调用者保存寄存器
24, 25	t8,t9	
16-23	s0-s7	被调用者保存寄存器
26, 27	k0,k1	保留给异常处理使用
28	gp	全局指针(Global Pointer)——因为MIPS指令的立即数域宽度有限，gp寄存器可以作为基址寄存器进行load/store寻址

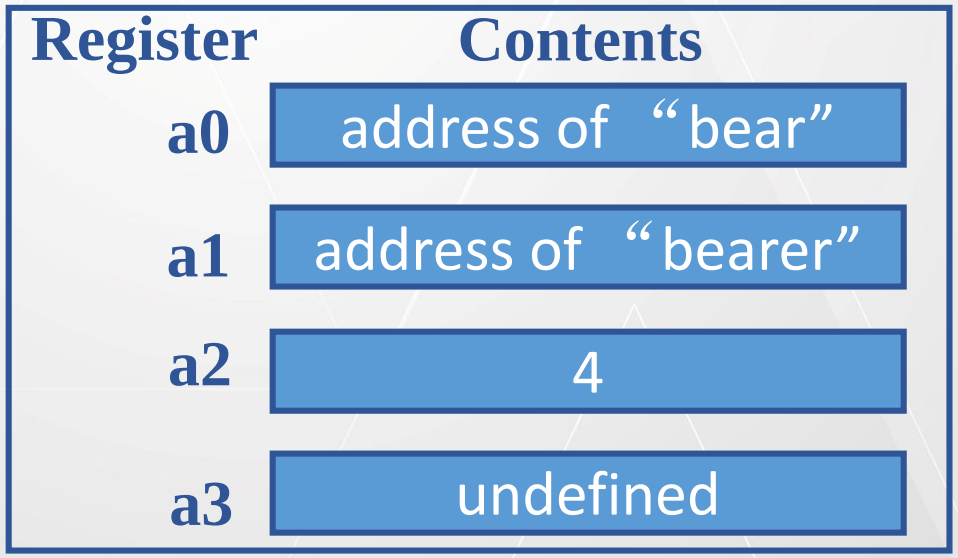
MIPS32寄存器命名与使用惯例

寄存器编号	名字	习惯用途
29	sp	栈顶寄存器
30	fp	栈帧寄存器（frame pointer）
31	ra	保存过程调用的返回地址

传统的MIPS32 传递过程参数方式（不包括浮点数）

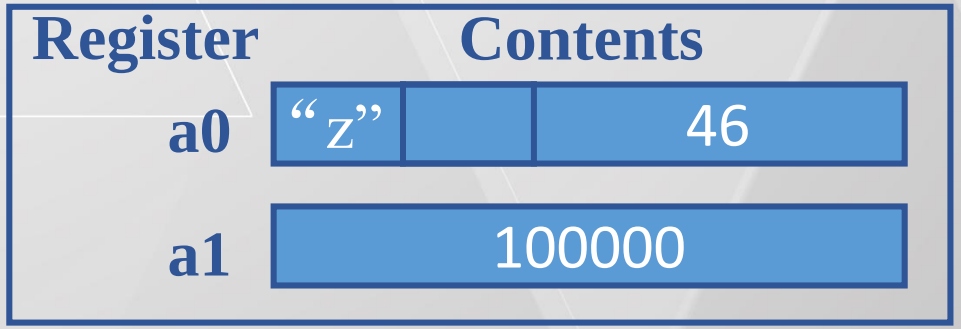
使用寄存器传参（前四个），剩余的使用栈

示例一：thesame = strncmp("bear", "bearer", 4);



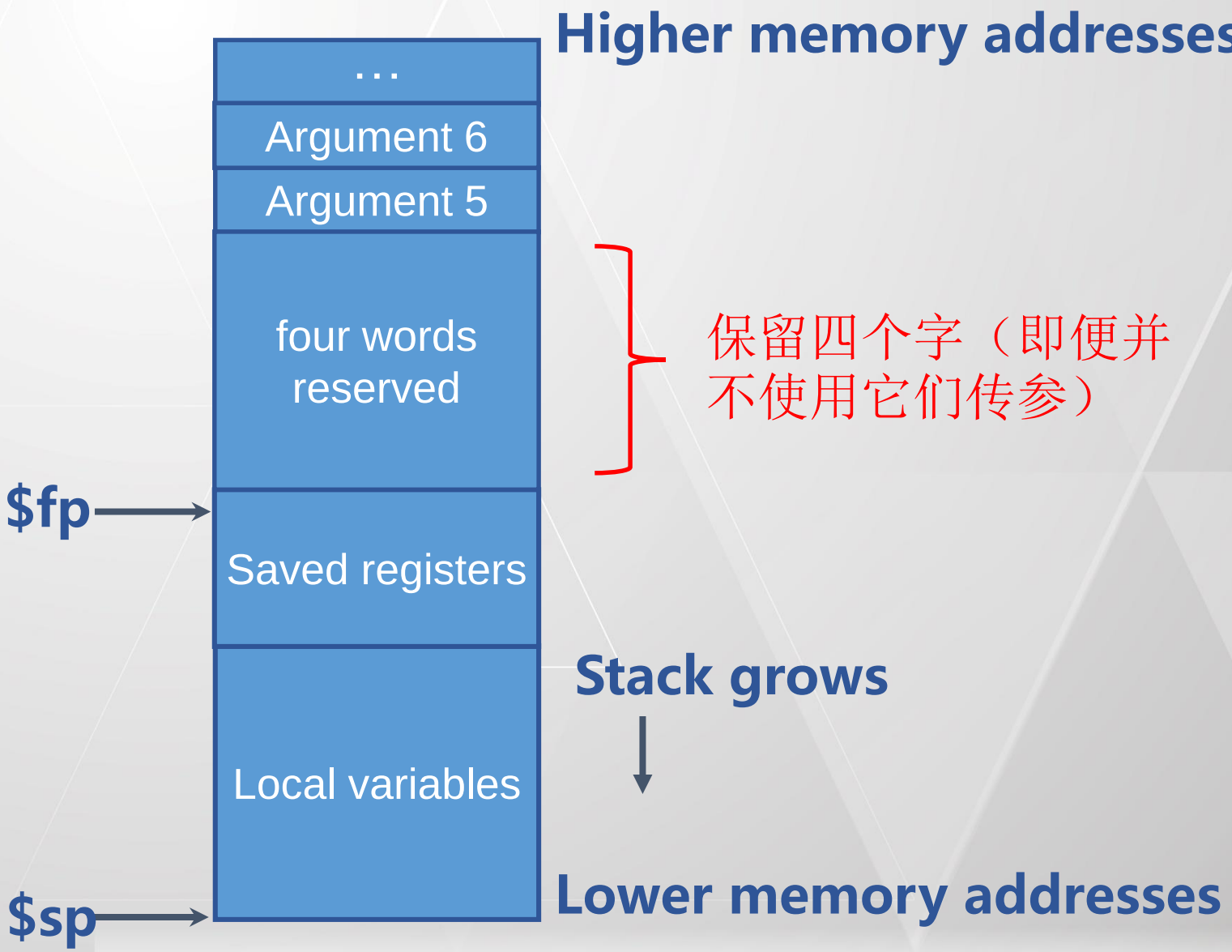
示例二:

```
struct thing {
char letter;
short count;
int value;
} = {"z", 46, 100000};
processing (thing) ;
```



a0~a3余下的内容通过栈传输

MIPS32体系结构下C过程的栈帧layout示意图



整数乘法与寄存器

- 使用了专用的乘法部件（不是主流水线的一部分）
- 两个32位数相乘得到64位结果:Hi / Lo寄存器
  - mfhi / mflo 指令

    mthi / mtlo 用途?
  - 也可存放除法结果：商（Lo）与余数（Hi）
- 乘除操作不产生异常，需要编译器判断
  - 除0不会产生异常!

程序地址空间布局

任何的处理器地址的访问（包括指令与数据）都需要经过MMU（内存管理单元）的地址转换，即程序（虚）地址→物理地址

有一些嵌入式处理器没有MMU，但是一般也经过地址转换(fixed mapping)

用户态\核心态\调试态

用户态下某些指令是非法的，且访存地址有限（0x8000 0000以下）

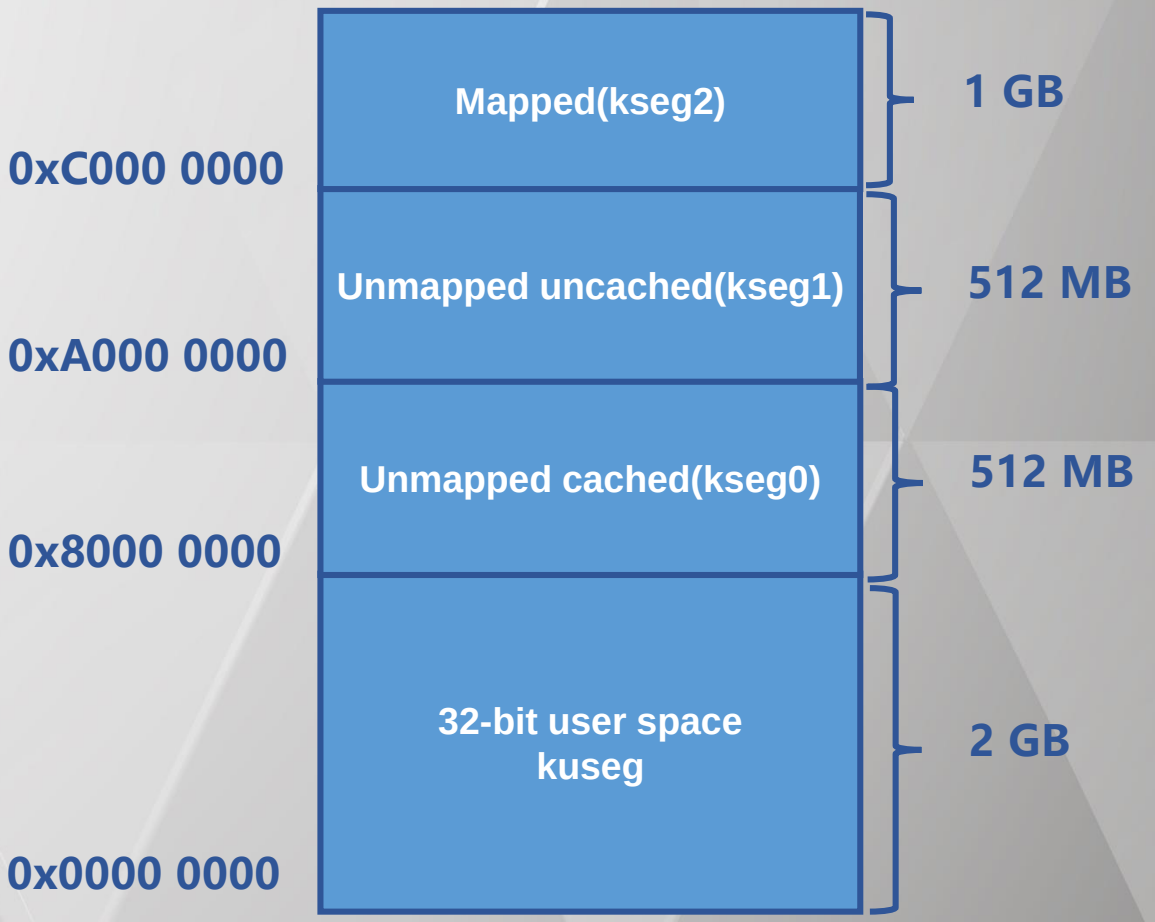
Kuseg——用户态可以使用的地址，需经过MMU转换

Kseg0——最高位清零后就是物理地址（cacheable）

Kseg1——最高三位清零后就是物理地址

Uncached；启动后可使用，系统启动地址就在这段

Kseg2——核心态使用，需经过MMU转换



协处理器0——CP0

支持虚拟存储、异常处理、运行状态切换等的系统控制协处理器

从程序员的角度来看，就是一系列寄存器

指令：MFC0　MTC0

不能在用户态下访问

与下列处理功能密切相关

处理器运行模式，如大小端模式、当前运行态等；

缓存控制

异常/中断处理

存储管理（MMU）

其它……

```
mfc0  t0, SR
and  t0, <要清零的位的反码>
or  t0, <要置1的位>
mtc0  SR, t0
```

CP0寄存器部分汇总

寄存器名称	编号	功能描述
SR (Status)	12	状态寄存器，包括处理器运行的状态、协处理器使能、某些中断使能以及一些处理器的配置信息
Cause	13	什么原因导致中断或者异常
EPC	14	中断/异常处理完成后从哪儿开始恢复执行
Count	9	这一组寄存器形成了一个高分辨率定制器，频率一般是处理器频率的50%。
Compare	11	
BadVaddr	8	引起地址相关异常的指令/数据地址
Context	4	对MMU编程的寄存器
EntryHi	10	
EntryLo0/1	2-3	
Index	0	
PageMask	5	
Random	1	
Wired	6	

CP0寄存器部分汇总

寄存器名称	编号	功能描述
PRId	15	CPU类型与版本号
Config	16.0	CPU参数设置
TagLo	28.0	用于对处理器缓存（cache）编程的寄存器
DataLo	28.1	
TagHi	29.0	
DataHi	29.1	
其它还有一系列用于硬件调试、性能计数的寄存器等等		