

汇编语言程序设计

MIPS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Hennessy领导团队，设计出第一个MIPS架构的处理器。

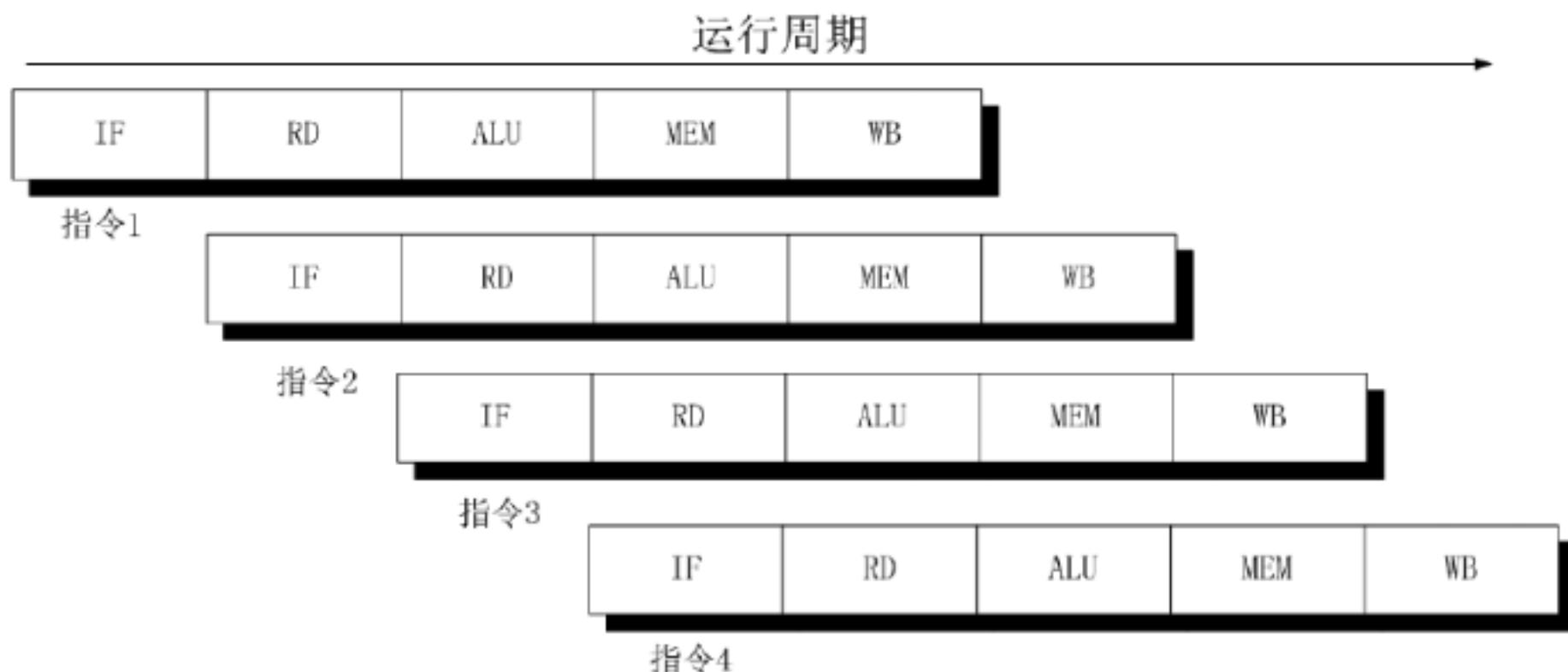
- 1984年， Hennessy教授离开斯坦福大学， 创立MIPS科技公司。
 - 于1985年， 设计出R2000芯片， 1988年， 将其改进为R3000芯片， 1991年： R4000。
 - 陆续推出R8000（于1994年）、R10000（于1996年）和R12000（于1997年）等型号。
 - 后重点转向嵌入式领域——2000年， MIPS公司发布了针对MIPS32 4Kc的版本以及64位MIPS 64 20Kc处理器内核。
- MIPS处理器是八十年代中期RISC CPU设计的一大热点：
 - 在许多领域， 如Sony， Nintendo的游戏机， Cisco的路由器和SGI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，已经十分成熟。
- 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指令架构。

MIPS32体系结构 (系统程序员可见的部分)

▶ 经典的五级流水

- Instruction Fetch (IF)
- Read Registers (RD)
- Arithmetic Operation (ALU)
- Memory Access (MEM)
- Write Back (WB)



▶ 指令集特点（与X86指令集对比） *MIPS32结构中的字长度为32bit

	MIPS32	X86-32
指令长度	固定（32位）	变长（1-15字节）
	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	

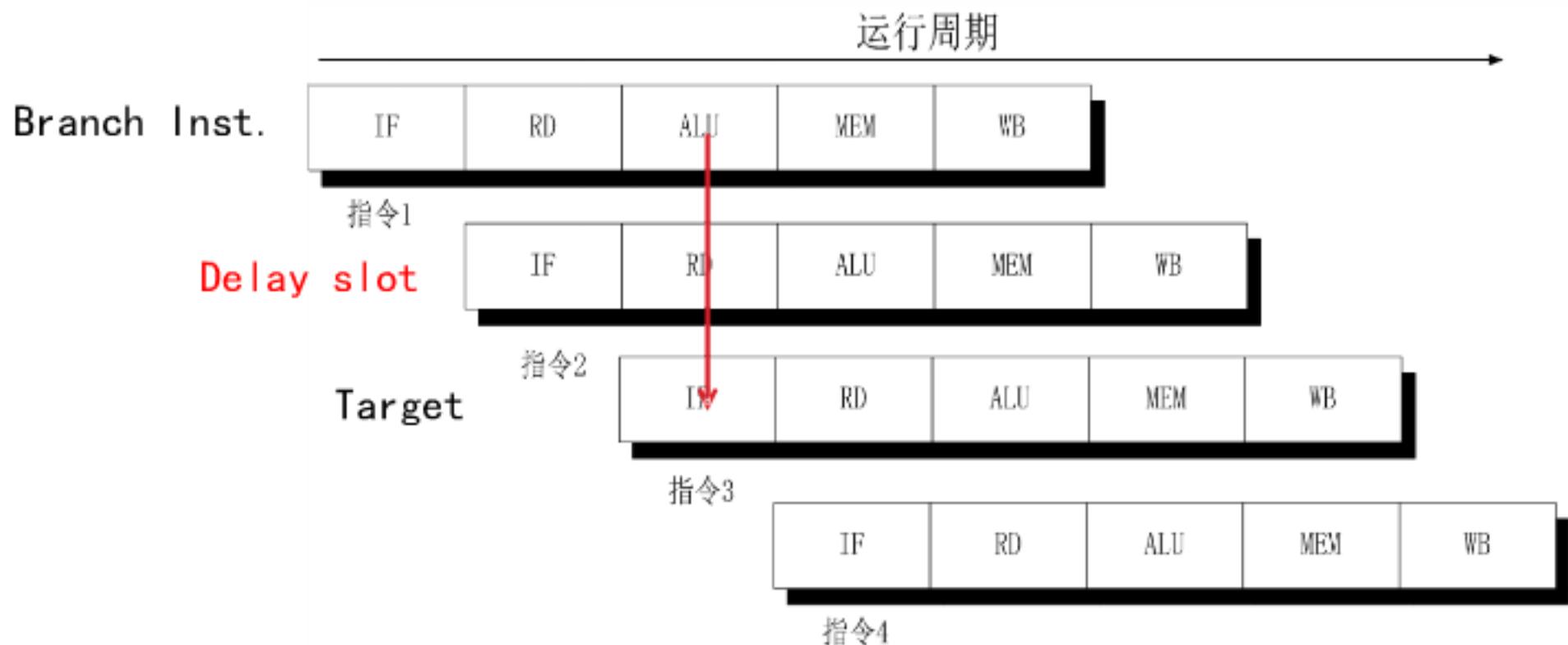
	MIPS32	X86-32
通用寄存器个数	32	8
	0号寄存器的值永远是0。0是最常用的常数，有利于节省指令编码。	
条件码	无	有
	所有信息存于通用寄存器，条件判断通过检测通用寄存器来进行。	
访存操作	只能通过load/Store指令	多数指令支持
	支持双字、字*、字节、半字等；一般需要地址对齐（有专门的不对齐load / store指令）	无需地址对齐
	一种寻址方式：基址+常数偏移量	多种

	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	对于中断与异常的处理主要依赖软件完成	

▶ 程序员可见的流水线效果

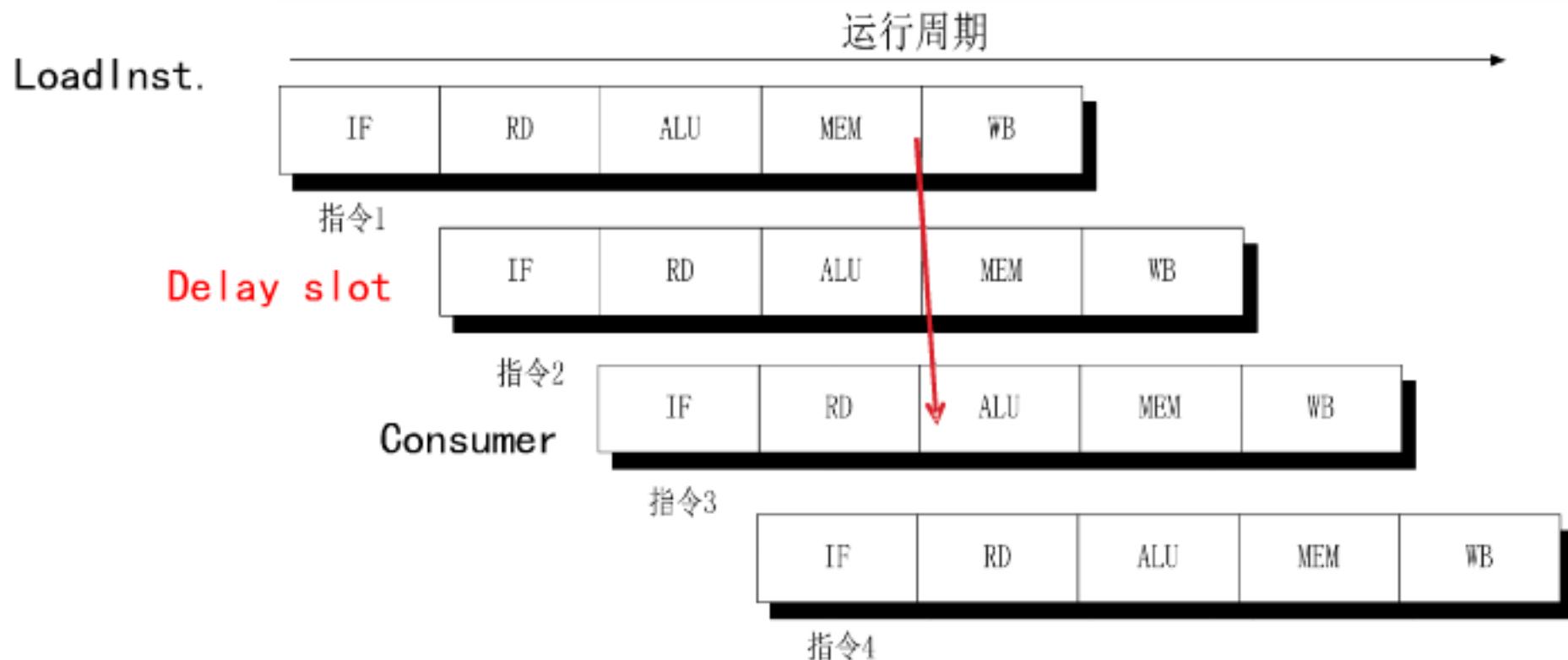
◦ Branch Delay Slot（跳转延迟）

- 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



- 加载延迟 (Load Delay Slot)

- Load指令在mem端才能获得数据，使得第二条指令无法立即使用加载数据，存在一个加载延迟槽
- 现代的MIPS处理器已经硬件支持这个延迟槽的正确处理



▶ 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）实现优化及完成复杂功能。

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           #调用过程（jal指令类似于X86的call）  
move $4, $6            #这条指令位于delay slot内，与前一条指令一起执行  
xxx # return here after call #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	支持无符号与带符号扩展
long long	dword	8	ld 中的”d”	
int	word	4	lw 中的”w”	
long ²				
short	halfword	2	lh 中”h”	
char	byte	1	lb 中的”b”	

lb \$1, offset(\$2) #mem[offset(\$2)] = 0xfe; \$1 = 0xfffffffffe

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x86-32中的eip）
- 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	保留给中断/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

寄存器编号	名字	习惯用途
29	sp	栈顶指针
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);

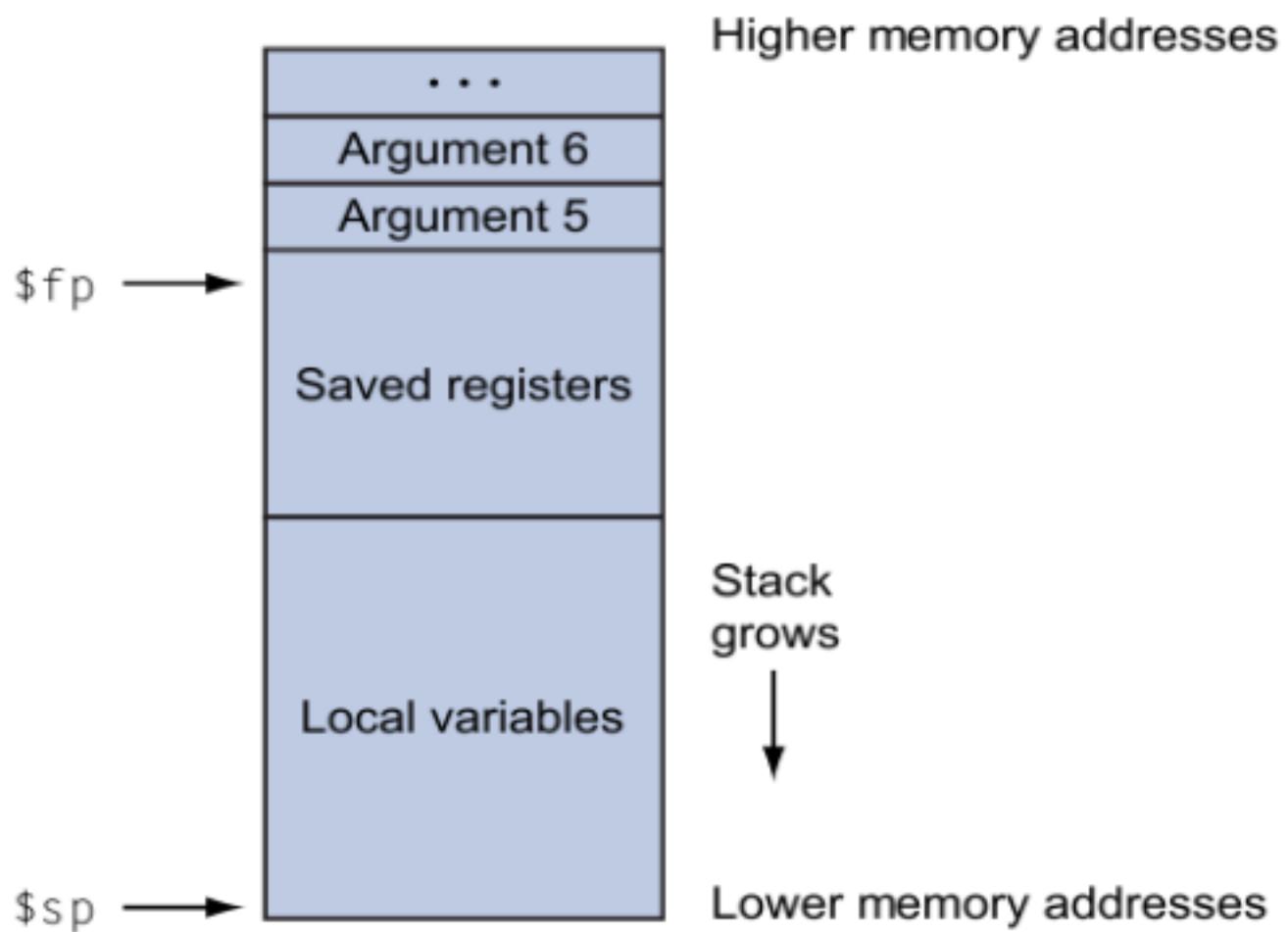
Register	Contents
a0	address of "bear"
a1	address of "bearer"
a2	4
a3	undefined

示例二：

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

Register	Contents
a0	"z" 46
a1	100000

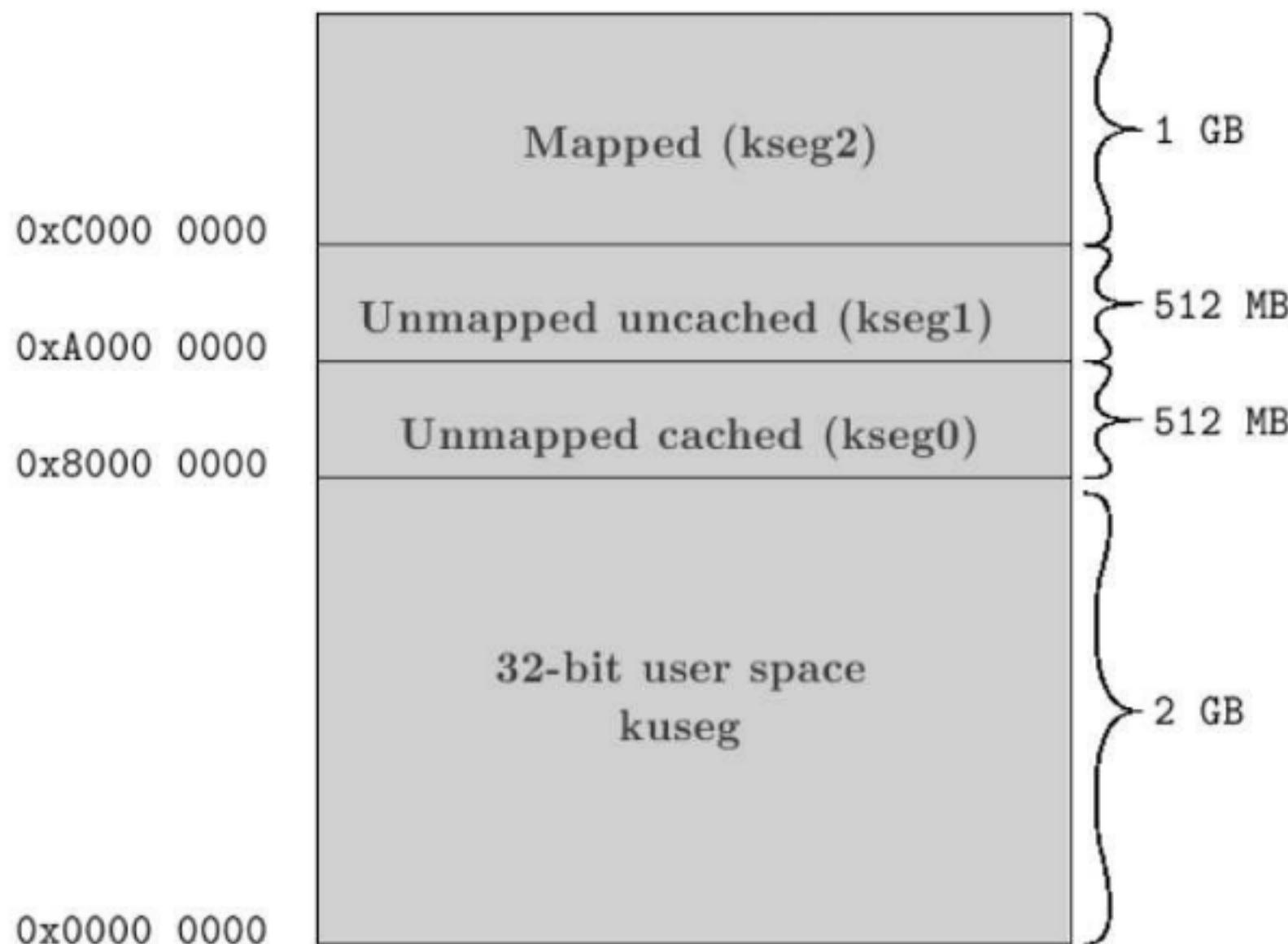
•MIPS32体系结构下C过程的栈帧Layout示意图



▶ 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
 - 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）都需要经过存储管理单元（MMU）的地址转换
 - 即“程序地址（或称虚拟地址）”转换为“物理地址”
 - 也有一些嵌入式处理器没有MMU，但是一般也经过地址转换
- 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（最高位为0）
 - 核心态下可以做任何事，访问任意空间
 - 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
 - 经过缓存
 - kseg1 最高三位清零后就是物理地址
 - 不经过缓存
 - 启动后期可使用，系统的启动地址就在这段
 - 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>
mtc	SR, t0

CPO寄存器部分汇总

寄存器名称	编号	功能描述
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	
EntryLo 0-1	2-3	
Index	0	
PageMask	5	
Random	1	
Wired	6	

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

- CP0示例-1

- Cause寄存器

BD位： 中断或者异常是否发生在delay slot中（EPC保存的是处理完成后的返回地址）

IP7-0： 待处理的中断

ExcCode: 5位编码， 异常种类

31	30	29	28	27	26	25	24	23	22	21	16	15	10	9	8	7	6	2	1	0
BD	TI	CE	DC	PC I	0	IV	WP	0		IP7..IP2		IPI..IPO	0		Exc Code		0			

- CP0示例-2

- Context寄存器

- PTEBase: 内存页表地址

- BadVPN2: 当发生快表（TLB）缺失异常时，引起该异常的地址的部分（31-13 bit）会写入此字段。

31	23 22	4 3	0
PTEBase		BadVPN2	0

- CPO Hazard*

对于CPO的某一寄存器进行修改后，由于MIPS的流水线执行的特点，CPO的状态在真正改变之前后续指令就进入流水线开始操作，可能会对该指令的某些流水段的操作造成非预期的影响。

这个问题需要由软件或者编译器解决。

汇编语言程序设计

MIPS32指令集与编程

- MIPS32指令集
 - 以经典的嵌入式处理器MIPS 4kc系列为参照
- 编程实例

▶ 指令分类 (主要部分, 不包括浮点)

算术 (Arithmetic) 指令 (部分)

ADD	Add Word
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
ADDU	Add Unsigned Word
CLO	Count Leading Ones in Word
CLZ	Count Leading Zeros in Word
DIV	Divide Word
DIVU	Divide Unsigned Word
MADD	Multiply and Add Word to Hi, Lo
MADDU	Multiply and Add Unsigned Word to Hi, Lo
MSUB	Multiply and Subtract Word to Hi, Lo
MSUBU	Multiply and Subtract Unsigned Word to Hi, Lo
MUL	Multiply Word to GPR
MULT	Multiply Word
MULTU	Multiply Unsigned Word
SLT	Set on Less Than
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
SLTU	Set on Less Than Unsigned
SUB	Subtract Word
SUBU	Subtract Unsigned Word

◦ 举例

指令	Format	指令功能	其它
ADD	ADD rd, rs, rt	$rd \leftarrow rs + rt$	执行32位带符号整数加法；如果补码运算溢出则产生异常
ADDI	ADDI rt, rs, immediate	$rt \leftarrow rs + immediate$	16位带符号立即数符号扩展后执行加法；如果补码运算溢出则产生异常
ADDU	ADDU rd, rs, rt	$rd \leftarrow rs + rt$	不产生异常

指令	Format	指令功能	其它
CLO	CLO rd, rs	$rd \leftarrow rs$ 前导1的个数	X86指令集中有类似的BSF (Bit Scan Forward)、BSR指令
CLZ	CLZ rd, rs	$rd \leftarrow rs$ 前导0的个数	

补充*

lib_c库中有相应的函数

- ffs, ffsl, ffsll – find first bit set in a word

```
#include <strings.h>
int ffs(int i);
```

```
#include <string.h>
int ffsl(long int i);
int ffsll(long long int i);
```

指令	Format	指令功能	其它
MUL	MUL rd, rs, rt	$rd \leftarrow rs \times rt$	32位整数相乘，结果只保留低32位；Hi/Lo寄存器无定义
MULT	MULT rs, rt	$(HI, LO) \leftarrow rs \times rt$	32位带符号整数相乘，结果存于Hi/Lo寄存器
MULTU	MULTU rs, rt	$(HI, LO) \leftarrow rs \times rt$	32位无符号整数相乘，结果存于Hi/Lo寄存器
DIV	DIV rs, rt	$(HI, LO) \leftarrow rs / rt$	32位带符号数... 不会产生算术异常 (即便除以0)
DIVU	DIVU rs, rt	$(HI, LO) \leftarrow rs / rt$	32位无符号数... 不会产生算术异常 (即便除以0)

指令	Format	指令功能	其它
MADD	MADD rs, rt	$(HI,LO) \leftarrow (HI,LO) + (rs \times rt)$	32位带符号整数乘加
MADDU		?	?
MSUB		?	?
MSUBU		?	?
SLT	SLT rd, rs, rt	$rd \leftarrow (rs < rt)$	比较两个带符号32位整数，比较结果(1或者0)存入rd寄存器
SLTI		?	?
SLTIU		?	?
SLTU		?	?

分支 (Branch) 和跳转 (Jump) 指令 (部分)

BEQ	Branch on Equal
BGEZ	Branch on Greater Than or Equal to Zero
BGEZAL	Branch on Greater Than or Equal to Zero and Link
BGTZ	Branch on Greater Than Zero
BLEZ	Branch on Less Than or Equal to Zero
BLTZ	Branch on Less Than Zero
BLTZAL	Branch on Less Than Zero and Link
BNE	Branch on Not Equal
J	Jump
JAL	Jump and Link
JALR	Jump and Link Register
JR	Jump Register

指令控制 (Instruction Control) 指令

NOP (伪指令)	No Operation (SLL, r0, r0, 0) 伪指令
------------------	-----------------------------------

PC指的是下一条指令地址 (delay slot)

指令	Format	指令功能	其它
BEQ	BEQ rs, rt, offset	if rs = rt then branch	target_offset ← sign_extend(offset 00); if (rs = rt) then PC ← PC + target_offset offset的宽度为16位
BGEZ	BGEZ rs, offset	if rs ≥ 0 then branch	...
BGEZA L	BGEZAL rs, offset	if rs ≥ 0 then procedure_call	... GPR[31] ← PC + 4

PC指的是下一条指令地址 (delay slot)

指令	Format	指令功能	其它
J	J target	$PC \leftarrow PC_{(高四位)} (target 00)$ (target 26位)	在当前的256MB对齐的空间内跳转
JAL	JAL target	$GPR[31] \leftarrow PC + 4;$ $PC \leftarrow PC_{(高四位)} (target 00)$	在当前的256MB对齐的空间内执行过程调用
JALR	JALR rs (rd = 31 implied)	$rd \leftarrow PC + 4$ $PC \leftarrow rs$	执行过程调用，过程入口地址位于rs内
	JALR rd, rs		
JR	JR rs	$PC \leftarrow rs$	跳转至rs指定的地址

装载 (Load) 、存储 (Store) 指令 (部分)

LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LL	Load Linked Word
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
SB	Store Byte
SC	Store Conditional Word
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right

指令	Format	指令功能	其它
LW	LW rt, offset(base)	从内存中读取一个字存入目的寄存器	$rt \leftarrow memory[base+offset]$ (offset是16位带符号整数) 地址必须4字节对齐，否则产生异常
LB	LB rt, offset(base)	从内存中读取一个字节，符号扩展后存入目的寄存器	$rt \leftarrow sign_extend(memory[base+offsset])$ (offset是16位带符号整数)
LBU	LBU rt, offset(base)	无符号扩展，其他同上。	...
SW	SW rt, offset(base)	从源寄存器读取字存入内存	...
SB	SB rt, offset(base)	从源寄存器读取低8位存入内存	...

- 非对齐Load/Store指令

LWL / LWR / SWL / SWR

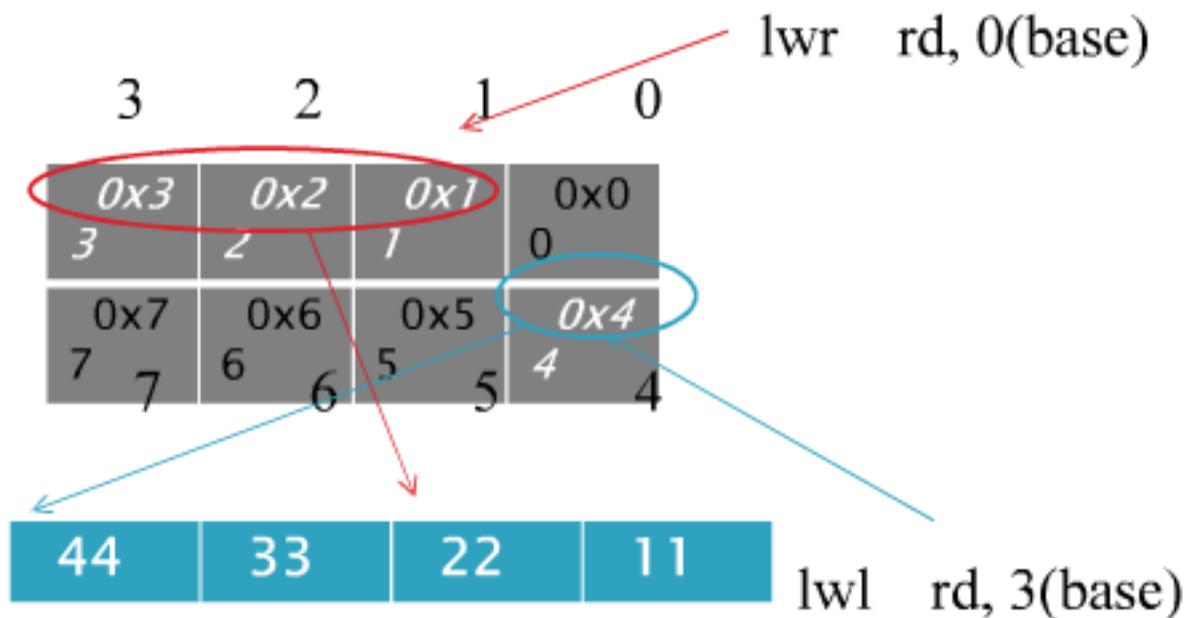
配对使用——从内存中的非4对齐地址取入WORD存于寄存器

- 指令中的L(左)、R(右)是针对寄存器而言的

举例（以小端为例）

lwr rd, 0 (base) #指令中的地址是目标word的最低字节地址

lwl rd, 3 (base) #指令中的地址是目标word的最高字节地址



LL / SC指令

- ▶ 在多线程程序中，为了实现对共享变量的互斥访问，一般需要一个TestAndSet的原子操作
 - 这种原子操作是需要专门的硬件支持才能完成的
- ▶ 在MIPS中，是通过特殊的Load/Store指令： LL (Load Linked，链接加载) 以及SC (Store Conditional，条件存储) 这一指令对完成的。

- 当使用 LL 指令从内存中读取一个字之后，处理器会记住 LL 指令的这次操作，同时 LL 指令读取的地址 也会保存在处理器的寄存器中。
- 接下来的 SC 指令，会检查上次 LL 指令执行后的 操作是否是原子操作（即不存在其它对这个地址的操作）
 - 如果是原子操作，则 V0（下一页的示例）的值将会被更新至内存中，同时 V0 的值也会变为1，表示操作成功；
 - 反之，如果不是原子操作（即存在其它对这个地址的访问冲突），则V0 的值不会被更新至内存中，且 V0 的值也会变为0，表示操作失败。

```
atomic_inc:

    ll      v0, 0(a0)                      # a0 has pointer to 'mycount'
    addu   v0, 1
    sc      v0, 0(a0)
    beq    v0, zero, atomic_inc          # retry if sc fails
    nop
    jr     ra
    nop
```

逻辑 (Logical) 指令

8条

AND	And
ANDI	And Immediate
LUI	Load Upper Immediate
NOR	Not Or
OR	Or
ORI	Or Immediate
XOR	Exclusive Or
XORI	Exclusive Or Immediate

转移 (Move) 指令

6条

MFHI	Move From HI Register
MFLO	Move From LO Register
MOVN	Move Conditional on Not Zero
MOVZ	Move Conditional on Zero
MTHI	Move To HI Register
MTLO	Move To LO Register

移位 (Shift) 指令

6条

SLL	Shift Word Left Logical
SLLV	Shift Word Left Logical Variable
SRA	Shift Word Right Arithmetic
SRAV	Shift Word Right Arithmetic Variable
SRL	Shift Word Right Logical
SRLV	Shift Word Right Logical Variable

指令	Format	指令功能	其它
AND	AND rd, rs, rt	针对32位寄存器执行逻辑与操作	$rd \leftarrow rs \text{ AND } rt$
ANDI	ANDI rt, rs, immediate	针对32位寄存器与立即数（0扩展后）执行逻辑与操作	$rt \leftarrow rs \text{ AND zero_extend(immediate)}$
LUI	LUI rt, immediate	将16位立即数装入目的寄存器的高16位（低16位清0）	$rt \leftarrow immediate 00\dots0(16)$
MOVZ	MOVZ rd, rs, rt	条件移动	$\text{if } rt = 0 \text{ then } rd \leftarrow rs$
SLL	SLL rd, rt, sa	左移操作	$rd \leftarrow rt << sa$ sa是一个5位立即数（无符号）
SLLV	SLLV rd, rt, rs	左移操作	寄存器rs的低5位表示左移的位数

陷阱 (Trap) 指令

BREAK	Breakpoint
SYSCALL	System Call
TEQ	Trap if Equal
TEQI	Trap if Equal Immediate
TGE	Trap if Greater or Equal
TGEI	Trap if Greater of Equal Immediate
TGEIU	Trap if Greater or Equal Immediate Unsigned
TGEU	Trap if Greater or Equal Unsigned
TLT	Trap if Less Than
TLTI	Trap if Less Than Immediate
TLTIU	Trap if Less Than Immediate Unsigned
TLTU	Trap if Less Than Unsigned
TNE	Trap if Not Equal
TNEI	Trap if Not Equal Immediate

分支 (Branch Likely) 指令 (不再建议使用)

BEQL	Branch on Equal Likely
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely
BGEZL	Branch on Greater Than or Equal to Zero Likely
BGTZL	Branch on Greater Than Zero Likely
BLEZL	Branch on Less Than or Equal to Zero Likely
BLTZALL	Branch on Less Than Zero and Link Likely
BLTZL	Branch on Less Than Zero Likely
BNEL	Branch on Not Equal Likely

EJTAG指令 (调试用)

DERET	Debug Exception Return
SDBBP	Software Debug Breakpoint

特权 (Privileged) 指令

CACHE	Perform Cache Operation
ERET	Exception Return
MFC0	Move from Coprocessor 0
MTC0	Move to Coprocessor 0
TLBP	Probe TLB for Matching Entry
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry
WAIT	Enter Standby Mode

某些操作数的限制（如立即数宽度）增加了汇编编程难度。
为降低难度，MIPS汇编器会做一些预处理。

addu \$2, \$4, 64 ⇒ addiu \$2, \$4, 64 #立即数加指令，不产生溢出异常

addu \$4, 0x12345 ⇒ li at, 0x12345
addu \$4, \$4, at #装载立即数（因为立即数超过了
16位二进制所能表示的范围）；
at则是保留给汇编器使用的寄存器；
注意li也是一条伪指令

li \$3, -5 ⇒ addiu \$3, \$0, -5 请解释下这些指令转

li \$4, 0x8000 ⇒ ori \$4, \$0, 0x8000 换？

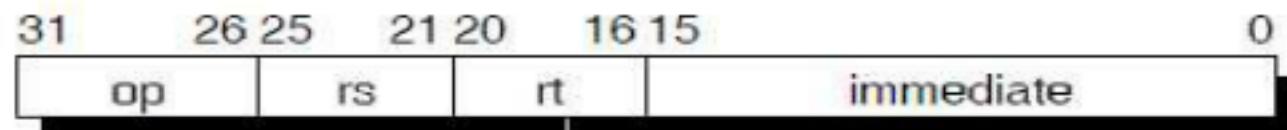
li \$5, 0x120000 ⇒ lui \$5, 0x12

li \$6, 0x12345 ⇒ lui \$6, 0x1
ori \$6, \$6, 0x2345

lw	\$2, (\$3)	\Rightarrow	lw	\$2, 0(\$3)
lw	\$2, 8+4(\$3)	\Rightarrow	lw	\$2, 12(\$3)
lw	\$2, addr	\Rightarrow	lui	at, %hi(addr)
			lw	\$2, %lo(addr) (at)
sw	\$2, addr(\$3)	\Rightarrow	lui	at, %hi(addr)
			addu	at, at, \$3
			sw	\$2, %lo(addr) (at)

▶ 指令编码

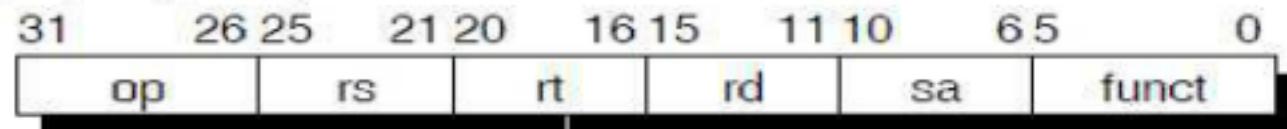
I-Type (Immediate)



J-Type (Jump)



R-Type (Register)



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

Opcode		Bits 28 .. 26							
		0	1	2	3	4	5	6	7
Bits 31..29		000	001	010	011	100	101	110	111
0	000	SPE CIAL	REG IMM	J	JAL	BEQ	BNE	BLEZ	BGT Z
1	001	ADD I	ADD IU	SLTI	SLTI U	AND I	ORI	XOIT	LUI
2	010	COP 0	保留	MIP S保留	MIPS 保留	BEQ L	BNEL	BLEZ L	BGT ZL
3	011	保留	保留	保留	保留	SPE CIAL 2	保留	保留	可扩 展
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	保留
5	101	SB	SH	SWL	SW	保留	保留	SWR	CAC HE
6	110	LL	保留	MIP S保留	PRE F	保留	保留	MIPS 保留	保留
7	111	SC	保留	MIP S保留	可扩 展	保留	保留	MIPS 保留	保留

Op 段编码*

function		Bits 2 .. 0							
		0	1	2	3	4	5	6	7
Bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	保留	SRL	SRA	SLLV	可扩展	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	可扩展	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	保留	可扩展	保留	保留
3	011	MULT	MULTU	DIV	DIVU	保留	保留	保留	保留
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	可扩展	可扩展	SLT	SLTU	保留	保留	保留	保留
6	110	Special操作码	TGE	码对应的Function	LT	可扩展	*TNE	可扩展	
7	111	保留	可扩展	保留	保留	保留	可扩展	保留	保留

Function		Bits 2 .. 0							
		0	1	2	3	4	5	6	7
Bits 5..3		000	001	010	011	100	101	110	111
0	000	MAD D	MAD DU	MUL	MIPS 保留	MSU B	MSUB U	MIPS 保留	MIPS 保留
1	001	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
2	010	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
3	011	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
4	100	CLZ	CLO	MIPS 保留	MIPS 保留	保留	保留	MIPS 保留	MIPS 保留
5	101	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
6	110	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留
7	111	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	MIPS 保留	SDBB P

Special2操作码对应的Function段编码*

REGIMM操作码对应rt段编码*

rt		Bits 18 .. 16								
		0	1	2	3	4	5	6	7	
Bits 20..19		000	001	010	011	100	101	110	111	
0	00	BLTZ	BGEZ	BLTZ L	BGEZ L	可扩 展	可扩展	可扩展	可扩 展	
1	01	TGEI	TGEI U	TLT	TLTI U	TEQI	可扩展	TNEI	可扩 展	
2	10	BLTZ AL	BGEZ AL	BLTZ ALL	BGEZ ALL	可扩 展	可扩展	可扩展	可扩 展	
3	11	可扩 展	可扩 展	可扩 展	可扩 展	可扩 展	可扩展	可扩展	可扩 展	

CP0-商Jrs-文編寫

CP0中的rs为CO时Function段编码*

- ▶ 编程实例（使用“SPIM” MIPS系统模拟器）

▶ MIPS汇编指示 (Directives)

- 段说明

.text .rdata .data .bss

```
.rdata
    msg:
        .asciiz "Hello world!\n"
.data
    table:
        .word 1
        .word 2
        .word 3
.text
    func:
        sub sp, 64
        ...
.bss
    .comm dbgflag, 4 # global common variable, 4 bytes
    .lcomm array, 300 # local common variable, 100 bytes
```

- 数据类型定义

- .byte .half .word .dword .float .double .ascii .asciiz

```
.byte 3                      # 1 byte: 3
.half 1, 2, 3                # 3 half-words: 1 2 3
.word 5 : 3, 6, 7            # 5 words: 5 5 5 6 7

.float 1.4142175             # 1 single-precision value
.double 1e+10, 3.1415         # 2 double-precision values

.ascii "Hello\0"
.ascii "Hello"
```

```
.align 4                    # 16-byte boundary( $2^4$ ) 对齐
var:
    .word 0
```

```
struc:
    .word 3
    .space 120                 # 120-byte 的空间
    .word -1
```

- 各类标识的属性

.globl .extern

```
.data
.globl status    # global variable
    status: .word 0
.text
.globs set_status # global function
set_status:
    subu sp, 24
...
.extern index, 4
.extern array, 100
lw $3, index          # load a 4-byte(1-word) external
lw $2, array($3)      # load part of a 100-byte external
```

A global symbol. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register \$gp.

- 过程指示(不是必需的)

.ent .end

```
.text
.ent localfunc
localfunc:
    addu v0, a1, a2      # return (arg1 + arg2)
    j ra
.end localfunc
```

```
.data  
array:  
    .word -4, 5, 8, -1  
msg1:  
    .asciiiz "\n The sum of the positive values = "  
msg2:  
    .asciiiz "\n The sum of the negative values = "  
  
.globl main  
.text  
main:  
    li    $v0, 4                  # system call code for print_str  
    la    $a0, msg1              # load address of msg1. into $a0  
    syscall                     # print the string  
    la    $a0, array              # Initialize address Parameter  
    li    $a1, 4                  # Initialize length Parameter  
    jal   sum                    # Call sum  
    nop      # delay slot, 可以通过设置SPIM的参数来关闭delay slot  
    move  $a0, $v0                # move value to be printed to $a0  
    li    $v0, 1                  # system call code for print_int  
    syscall                     # print sum of Pos:
```

```
li    $v0, 4          # system call code for print_str
la    $a0, msg2       # load address of msg2. into $a0
syscall             # print the string
li    $v0, 1          # system call code for print_int
move   $a0, $v1        # move value to be printed to $a0
syscall             # print sum of neg

li    $v0, 10         # terminate program run and
syscall # return control to system
```

```
sum:  
    li    $v0, 0  
    li    $v1, 0                                # Initialize v0 and v1 to zero  
  
loop:  
    blez $a1, retzz                            # If (a1 <= 0) Branch to Return  
    nop  
    addi $a1, $a1, -1                          # Decrement loop count  
    lw    $t0, 0($a0)                           # Get a value from the array  
    addi $a0, $a0, 4                            # Increment array pointer to next  
    bltz $t0, negg                            # If value is negative Branch to negg  
    nop  
    add  $v0, $v0, $t0                          # Add to the positive sum  
    b     loop                                 # Branch around the next two  
                                              # instructions  
    nop  
  
negg:  
    add  $v1, $v1, $t0                          # Add to the negative sum  
    b     loop                                 # Branch to loop  
    nop  
  
retzz:  
    jr   $ra # Return
```

◦ SPIM模拟的系统调用

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$a0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

QtSpim

File Simulator Registers Text Segment Data Segment Window Help

Regs | Int Regs [16] | Data | Text

Int Regs [16]

PC = 400000c EPC = 0 Cause = 0 BadVAddr = 0 Status = 3000ff10

EI = 0 LO = 0

R0 [r0] = 0 R1 [at] = 10010000 R2 [v0] = a R3 [v1] = ffffffff R4 [a0] = ffffffff R5 [a1] = 0 R6 [a2] = 7fffff3ac R7 [a3] = 0 R8 [t0] = ffffffff R9 [t1] = 0 R10 [t2] = 0 R11 [t3] = 0 R12 [t4] = 0 R13 [t5] = 0 R14 [t6] = 0 R15 [t7] = 0 R16 [s0] = 0 R17 [s1] = 0 R18 [s2] = 0 R19 [s3] = 0 R20 [s4] = 0 R21 [s5] = 0 R22 [s6] = 0 R23 [s7] = 0 R24 [t8] = 0 R25 [t9] = 0 R26 [k0] = 0 R27 [k1] = 0

Text

```
[00400024] 34020004 ori $2, $0, 4 ; 8: li $v0, 4 # system call code for print_str
[00400028] 3c011001 lui $1, 4097 [msg1] ; 9: la $a0, msg1 # load address of msg1. into
$a0
[0040002c] 34240010 ori $4, $1, 16 [msg1]
[00400030] 0000000c syscall ; 10: syscall # print the string
[00400034] 3c041001 lui $4, 4097 [array] ; 11: la $a0, array # Initialize address
Parameter
[00400038] 34050004 ori $5, $0, 4 ; 12: li $a1, 4 # Initialize length Parameter
[0040003c] 0c10001c jal 0x00400070 [sum] ; 13: jal sum # Call sum
[00400040] 00022021 addu $4, $0, $2 ; 14: move $a0, $v0 # move value to be printed
to $a0
[00400044] 34020001 ori $2, $0, 1 ; 15: li $v0, 1 # system call code for
print_int
[00400048] 0000000c syscall ; 16: syscall # print sum of Pos:
[0040004c] 34020004 ori $2, $0, 4 ; 17: li $v0, 4 # system call code for
print_str
[00400050] 3c011001 lui $1, 4097 [msg2] ; 18: la $a0, msg2 # load address of msg2. into
$a0
[00400054] 34240034 ori $4, $1, 52 [msg2]
[00400058] 0000000c syscall ; 19: syscall # print the string
[0040005c] 34020001 ori $2, $0, 1 ; 20: li $v0, 1 # system call code for
print_int
[00400060] 00032021 addu $4, $0, $3 ; 21: move $a0, $v1 # move value to be printed
to $a0
[00400064] 0000000c syscall ; 22: syscall # print sum of neg
```

Memory and registers cleared
 Loaded: C:/Users/Zhang/AppData/Local/Temp/qt_spim.qs024
 SPIM Version 0.1.7 rE February 12, 2012
 Copyright 1990-2012, Jason R. Larue.
 All Rights Reserved
 SPIM is distributed under a BSD license.
 See the file README for a full copyright notice.

示例二：阶乘 (delay slot关闭)

```
.text  
.globl main  
main:
```

```
subu    $sp, $sp, 32    # Stack frame is 32 bytes long  
sw      $ra, 20($sp)   # Save return address  
sw      $fp, 16($sp)   # Save old frame pointer  
addiu   $fp, $sp, 28   # Set up frame pointer
```

```
li      $a0, 10         # Put argument (10) in $a0  
jal    fact            # Call factorial function
```

```
move   $a0, $v0          #  
li     $v0, 1             # Print the result
```

Syscall

```
lw      $ra, 20($sp)   # Restore return address  
lw      $fp, 16($sp)   # Restore frame pointer  
addiu  $sp, $sp, 32   # Pop stack frame
```

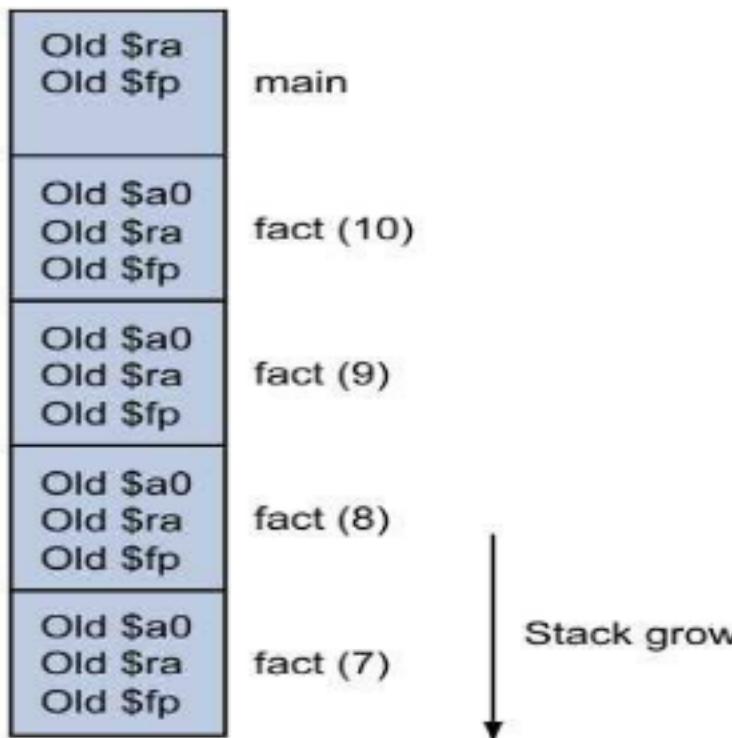
```
li      $v0, 10          # terminate program run and  
syscall                      # return control to system
```

```
.text
fact:
    subu    $sp, $sp, 32      # Stack frame is 32 bytes long
    sw      $ra, 20($sp)     # Save return address
    sw      $fp, 16($sp)     # Save frame pointer
    addiu   $fp, $sp, 28      # Set up frame pointer
    sw      $a0, 0($fp)      # Save argument (n)

    lw      $v0, 0($fp)      # Load n
    bgtz   $v0, $L2          # Branch if n > 0
    li      $v0, 1            # Return 1
    jr      $L1              #Jump to code to return
```

\$L2:

```
    lw      $v1, 0($fp)      # Load n
    subu   $v0, $v1, 1        # Compute n - 1
    move   $a0, $v0           # Move value to $a0
```

Stack

```
jal      fact          # Call factorial function
lw       $v1, 0($fp)    # Load n
mul     $v0, $v0, $v1   # Compute fact(n-1) * n
```

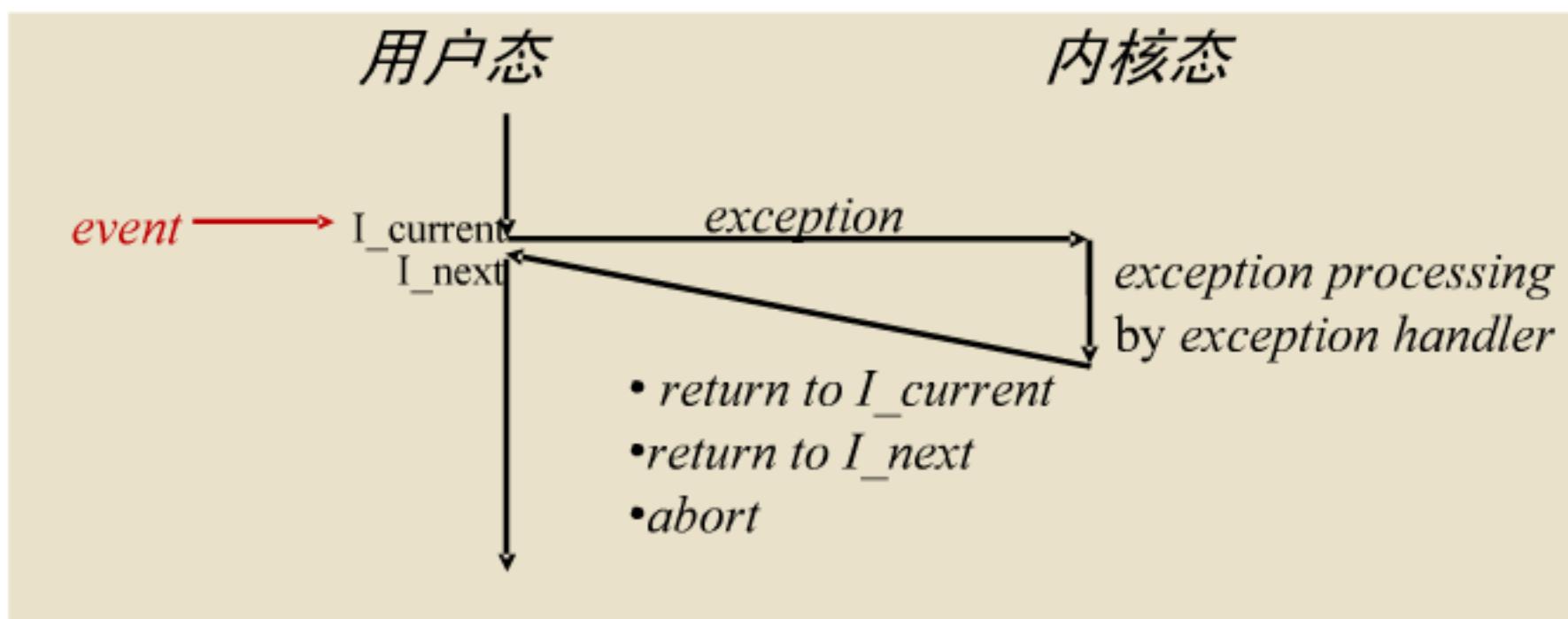
```
$L1:                                # Result is in $v0
    lw      $ra, 20($sp)  # Restore $ra
    lw      $fp, 16($sp)  # Restore $fp
    addiu $sp, $sp, 32    # Pop stack
    jr      $ra            # Return to caller
```

汇编语言程序设计

MIPS32异常处理

▶ 异常(Exception)

- 在程序运行过程中，某些打断程序正常运行流程的、且会引起运行态改变（从用户态到核心态）的事件。
 - 分为两类
 - 同步异常
 - 异步异常



▶ 同步异常

- 由指令执行引起的
- 陷入 (Traps)
 - 程序“故意”引起的
如：系统调用 (system call)，断点
(breakpoint)，Trap指令
 - 返回到下一条指令

- **Faults**

- 程序“无意”引起的、但是可恢复
如: 页缺失(recoverable)
- 重新执行当前指令

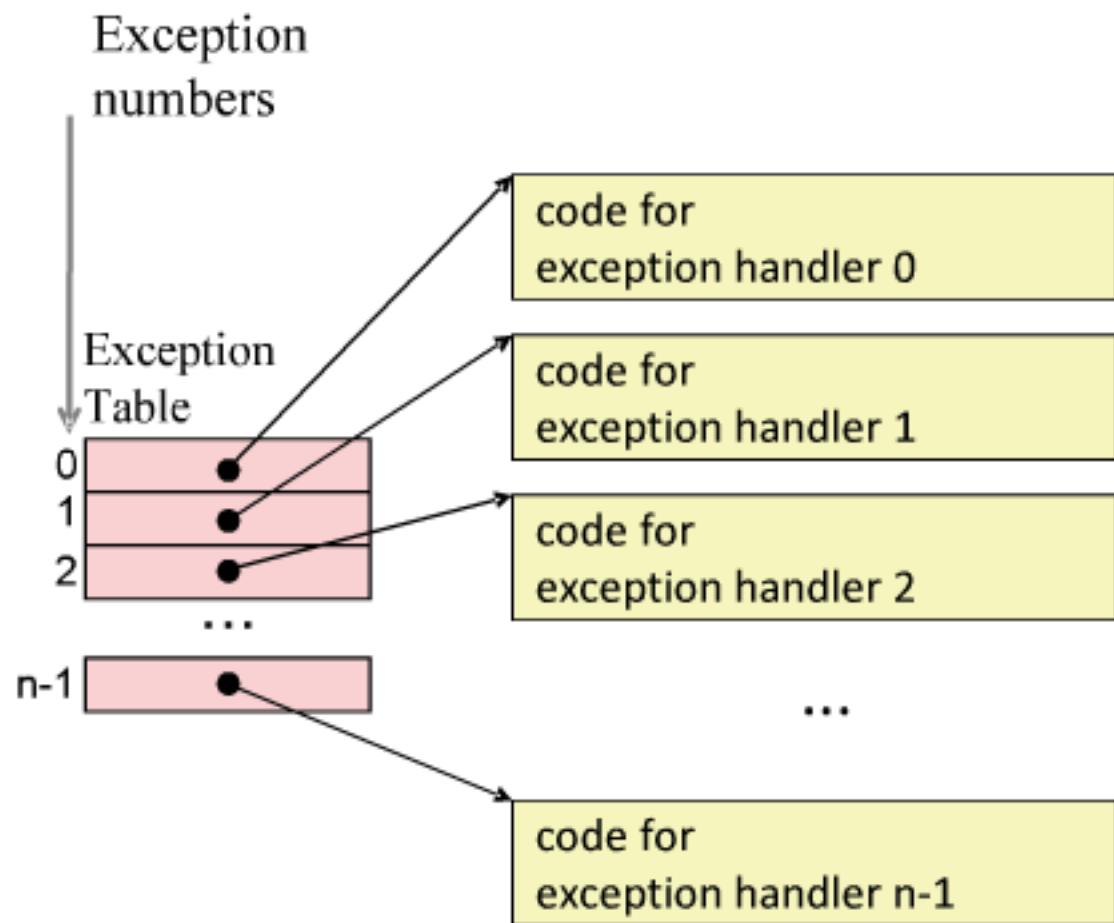
- **Aborts**

- 程序“无意”引起的、不可恢复
如: parity error, machine check
- 程序退出

▶ 异步异常（中断）

- 由“外部事件”引起，往往是外设触发。
如：IO中断、Hard Reset、Soft Reset
- 重新执行当前指令或者下一条指令

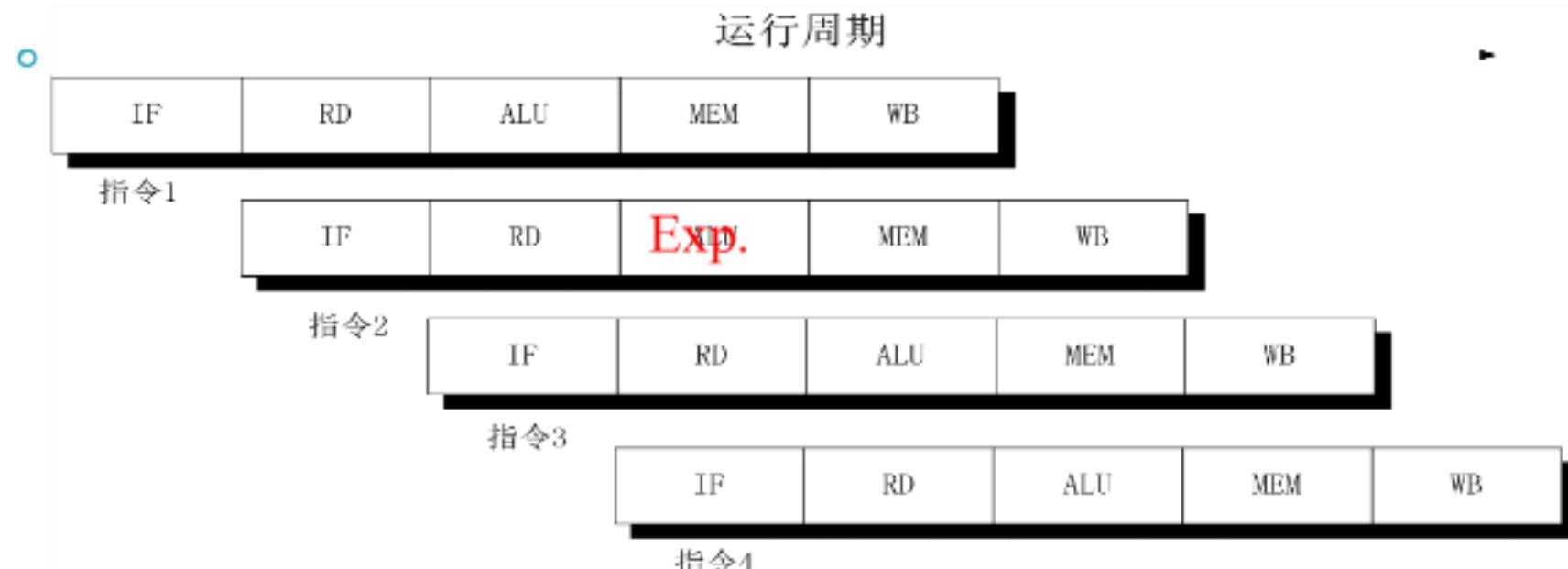
► 异常处理向量 (Vectors)



- 每类异常都有其编号以及异常处理入口地址，往往在内存中构成一张地址表。
- 所以称之为向量。

► 精确异常处理 (MIPS支持)

- 精确异常指的就是——在处理异常时，产生异常指令之前的指令都应执行完毕；该指令之后（包括该指令本身）的指令则都不处理。
 - 需要精确记录异常的位置（指令）
 - Branch Delay Slot
 - 需要取消后续指令
 - 需要正确恢复执行



▶ MIPS32下的异常种类

异常	描述
Reset	由SI_ColdReset信号引起
Soft Reset	由SI_Reset信号引起
DSS	EJTAG调试单步异常
DINT	EJTAG调试中断异常，由外部的EJ_DINT输入引起，或由设置ECR寄存器中的EjtagBrk位引起
NMI	由SI_NMI信号引起(不可屏蔽中断)
Machine Check	TLB写操作与一个存在的表项冲突
Interrupt	由未被屏蔽的硬件或软件中断信号引起
DIB	EJTAG调试硬件指令断点匹配

异常	描述
WATCH	访问地址与观察寄存器中的地址匹配 (取指时)
Deferred Watch	延迟的观察
AdEL	指令地址对齐错误，或用户模式下访问核心地址空间
TLBL	指令TLB缺失，指令TLB 无效（有效位为0）
IBE	取指时总线错误
DBp	EJTAG断点（执行SDBBP指令）
Sys	执行SYSCALL指令
Bp	执行BREAK指令
CpU	对一个未使能的协处理器执行协处理器指令

异常	描述
RI	执行保留指令
Ov	算术指令溢出
Tr	执行陷入指令（陷入条件为真时）
DDBL / DDBS	EJTAG数据地址断点（只对地址有效），或Store指令的EJTAG数据值断点（对地址和值有效）
WATCH	访问地址与观察寄存器中的地址匹配（访问数据时）
AdEL	读数据地址对齐错误，或用户模式下读核心地址空间数据
AdES	写数据地址对齐错误，或用户模式下写核心地址空间数据
TLBL	读数据时TLB缺失，或TLB无效（有效位为0）
TLBS	写数据时TLB缺失，或TLB无效（有效位为0）
TLB Mod	写TLB错误（写使能位为0）
DBE	读/写数据时总线错误
DDBL	EJTAG数据硬件断点与读指令读出的数据匹配

▶ MIPS异常处理基本过程

- 保存现场——在异常程序入口，硬件只记录了被打断程序的很少量的信息，因此需要保留相关的控制寄存器等值使得异常处理程序能够执行（k0、k1寄存器保留给异常处理使用）；
- 判断不同的异常——查询Cause寄存器，根据其不同的异常原因来进行不同的处理流程；
- 构造异常处理内存空间——异常处理程序可能由高级语言书写，需要保留通用寄存器，构造堆\栈存储区；
- 处理异常——.....
- 返回——恢复寄存器，清零Cause寄存器，将Status寄存器的相关位置1以开中断。

▶ 异常返回

- 一般而言，异常处理代码工作在核心态，而被中断的程序是在用户态，所以异常返回意味着状态转换。
- 这个转换与指令返回必须“同时”完成，即用一条指令完成。
 - 为什么？
- ERET指令
 - 返回EPC指向的地址
 - Status寄存器修改

▶ 中断

- 中断是异步发生的，是来自处理器外部的I/O设备的信号引发的。
- 硬件中断不是由任何一条专门的指令造成的
 - I/O设备，比如网络适配器，磁盘控制器等通过处理器芯片上的一个引脚发送信号，并将中断号放到系统总线上，用来触发中断，这个异常号标识了引起中断的设备。

- MIPS处理器的中断控制设计——
在中断发生时，如果指令已经完成了MEM阶段的操作，
则需保证该指令执行完毕。
反之，则丢弃这条指令的工作。

除NMI外，所有的内部或外部硬件中断(Hardware Interrupt)均共用这一个异常向量(Exception Vector)。

▶ 嵌套异常*

- 很多情况下，需要（或者是不可避免）允许在异常处理时发生另一个异常，这叫做嵌套异常。
 - 如果只是简单的处理，会产生混乱，因为被中断的程序的关键状态存放在EPC和STATUS寄存器中，并且另一个异常会立即修改这两个寄存器。所以在允许下一级嵌套异常之前，必须保存这些值。另外，一旦重新使能了异常，就不能再依赖k0和k1的值了。

- 嵌套异常的服务程序必须用一些内存空间来保存寄存器的值，使用的数据结构叫异常帧，多个嵌套异常的异常帧通常保存在栈中。
 - 每一个异常都会消耗栈资源，所以任意深度的嵌套异常是不可容忍的。
 - 大多数系统赋予每一类异常一个优先级，并且规定：当一个异常正在被服务时，只有高优先级的异常才被允许。这样的系统只需要和优先级数一样多的异常帧存储空间就够了。

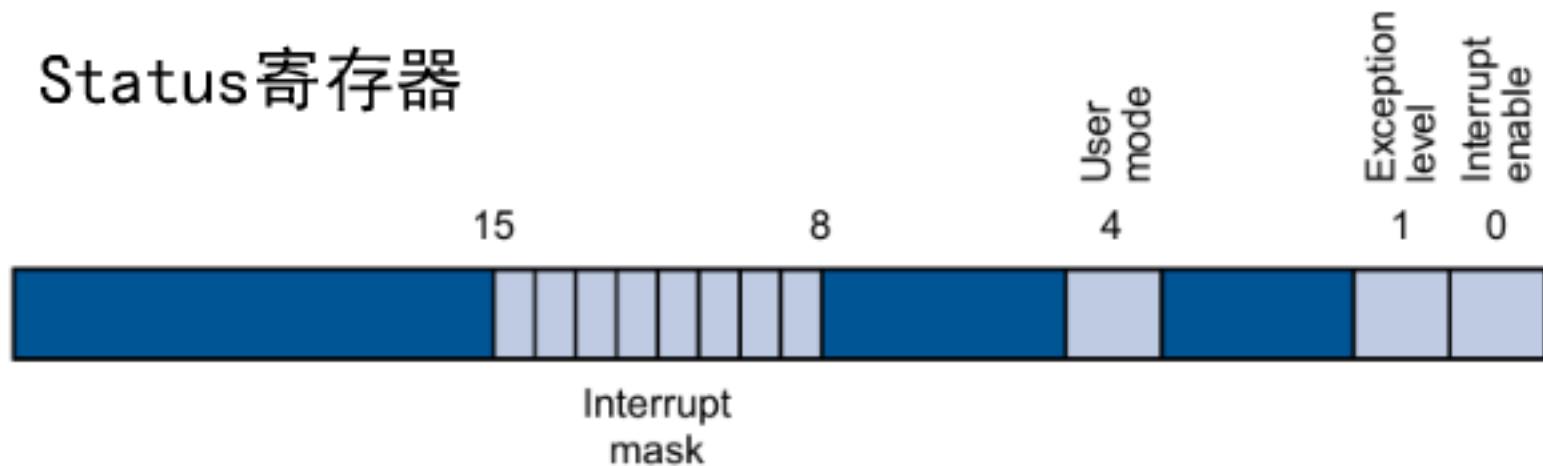
▶ SPIM模拟器支持的异常处理流程

SPIM实现了部分CP0寄存器

Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

mfc0 与mtc0 可以访问这些寄存器

Status寄存器



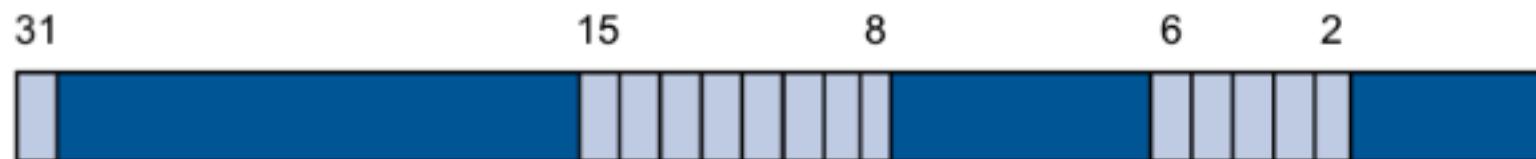
mask位为0: disable相应的中断（6个硬件中断、2个软件中断）

user mode位为0表示运行于内核态；否则为用户态（模拟器中固定为0）

exception level位：平时为0；当异常发生后被置为1（此时屏蔽了中断处理，即阻止在一个正在被处理的异常被打断，并表示运行在内核态）

interrupt enable位：当异常发生后被置为0，表示中断处理被禁止。

Cause寄存器



Branch
delay

Pending
interrupts

Exception
code

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

▶ 异常处理实例

- 触发读数据地址不对齐异常 (AdEL)
- 定位导致异常的指令 (EPC 或者 EPC+4, 取决于 Cause 中的相关位)
- 解码该指令, 取得异常数据地址, 并处理 (?)

- 如何返回?
如果位于branch Delay Slot中…

MIPS处理器的异常处理入口地址为0x80000180（kernel space）*。处理代码首先检查cause寄存器然后跳转至适当的具体处理代码。

处理代码示例：

```
.ktext 0x80000180
mov $k1, $at # 存储$at $v0 $a0; 后两者被异常处理过程使用
              # 前者则是汇编器使用
sw $v0, save0
sw $a0, save1 #为什么不存在栈上?
```

```
mfc0 $k0, $13      # Move Cause into $k0
srl $a0, $k0, 2     # Extract ExcCode field
andi $a0, $a0, 0xf
bne $k0, 0x18, ok_pc # Bad PC exception requires special checks

mfc0 $a0, $14      # EPC
andi $a0, $a0, 0x3   # Is EPC word-aligned?
beq $a0, 0, ok_pc
li $v0, 10          # Exit on really bad PC
syscall
```

*实际上的流程比这个复杂，入口地址也不止一个。这儿是简化说明

```
ok_pc:  
    ...                                # 异常处理代码  
  
ret:  
mfc0 $k0, $14          # Bump EPC register  
addiu $k0, $k0 4       # Skip faulting instruction  
# (Need to handle delayed branch case here)  
  
mtc0 $k0, $14          # Set EPC  
# Restore  
move $at, $k1          # Restore $at  
lw    $v0, s1           # Restore other registers  
lw    $a0, s2  
mtc0 $0, $13          # Clear Cause register  
mfc0 $k0, $12          # Set Status register  
ori  $k0, 0x1           # Interrupts enabled  
mtc0 $k0, $12  
eret                  # Return to EPC  
  
.kdata  
save0: .word 0  
save1: .word 0
```

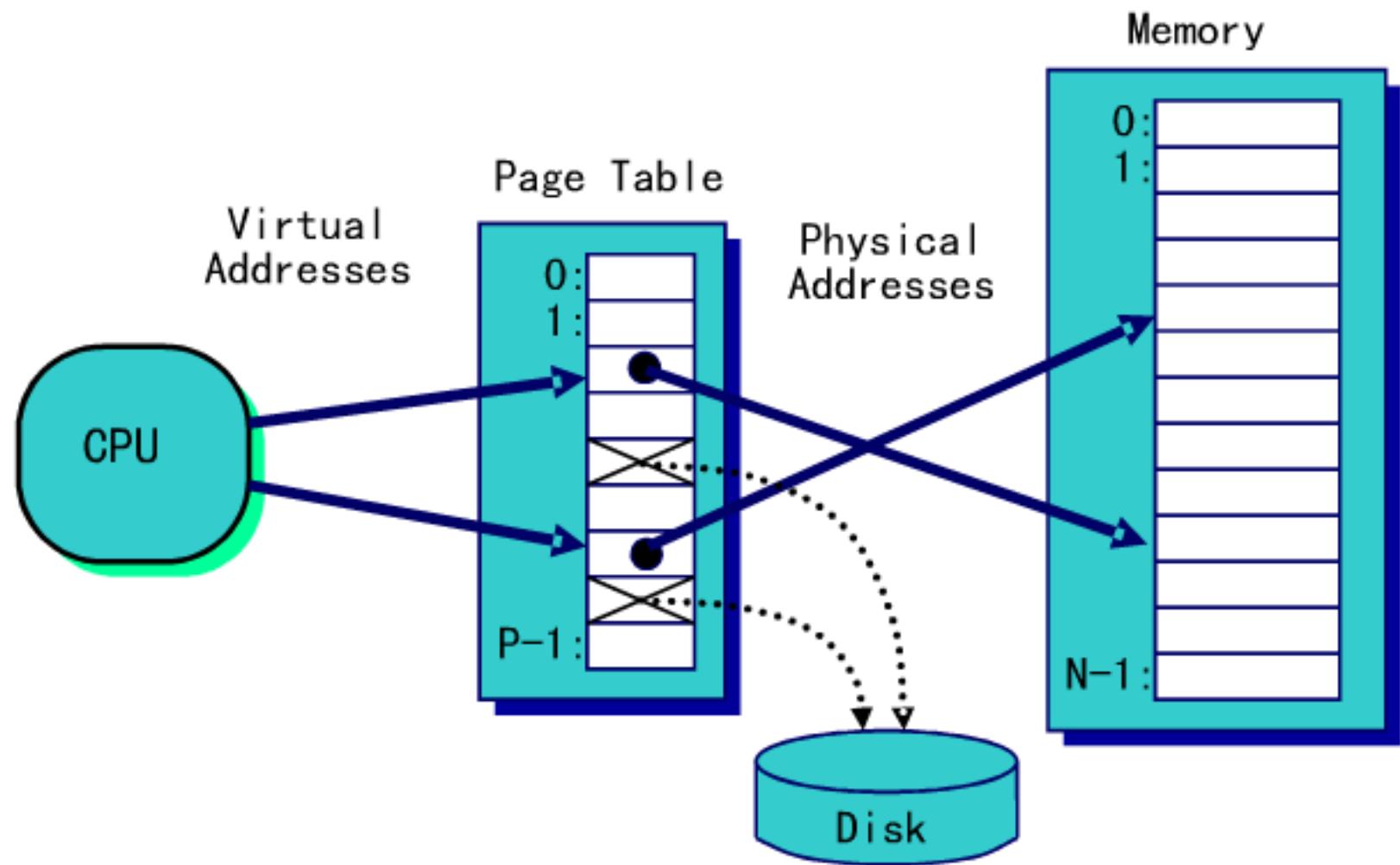
汇编语言程序设计

虚存概念初步

虚存设计背景

- ▶ 将物理内存作为磁盘数据的“缓存”
 - 因为进程的地址空间可能会超过物理内存的大小
- ▶ 简化程序的内存管理
 - 多个进程同时运行
 - 每个都有自己的地址空间
- ▶ 存储保护功能
 - 某进程不能访问其它进程的空间
 - 一个进程的不同空间区域具有不同的访问权限

▶ 将物理内存作为磁盘数据的快速“缓存”

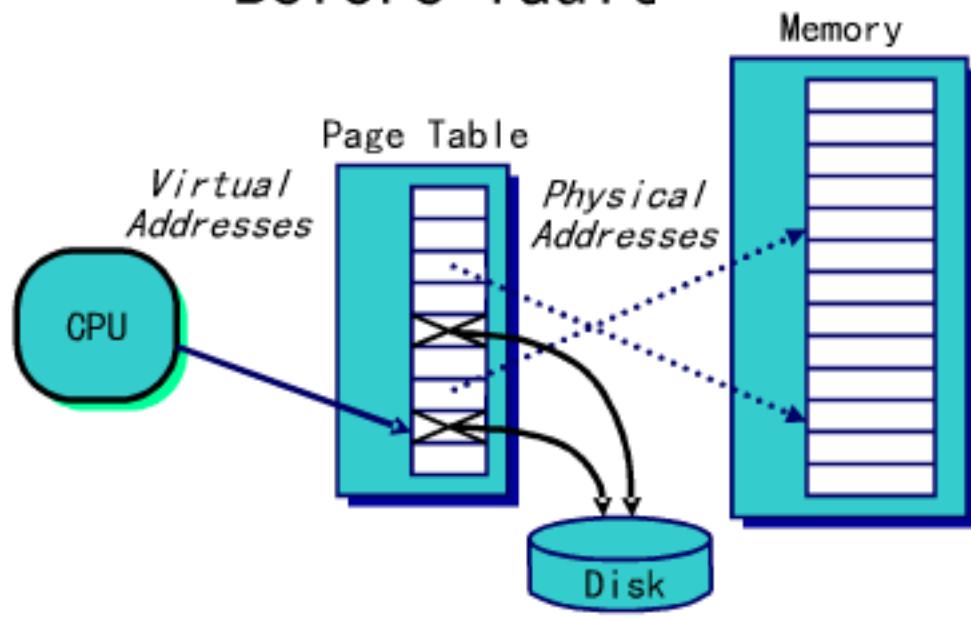


- 地址转换 (Address Translation)：一般由处理器硬件通过页表 (page table) 将虚拟地址 (程序地址) 转换为物理地址。

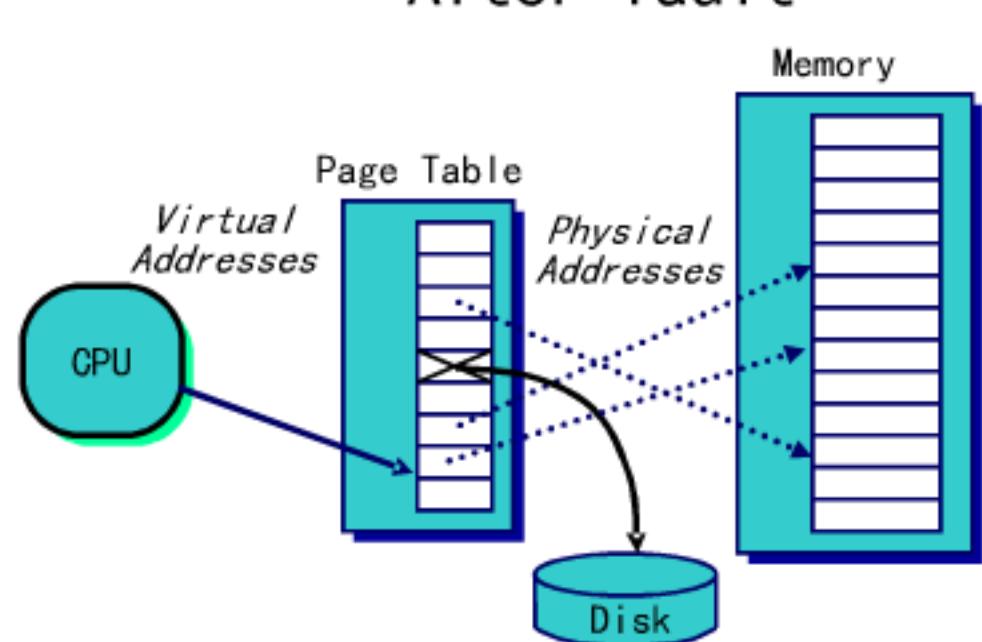
页缺失 (Page fault)

- 页表项记录某虚存地址（以页为单位）是否在物理内存中。
- 如果不是，那么出发“页缺失”异常，由异常处理代码将所需数据从外部存储（硬盘）读入内存。

Before fault

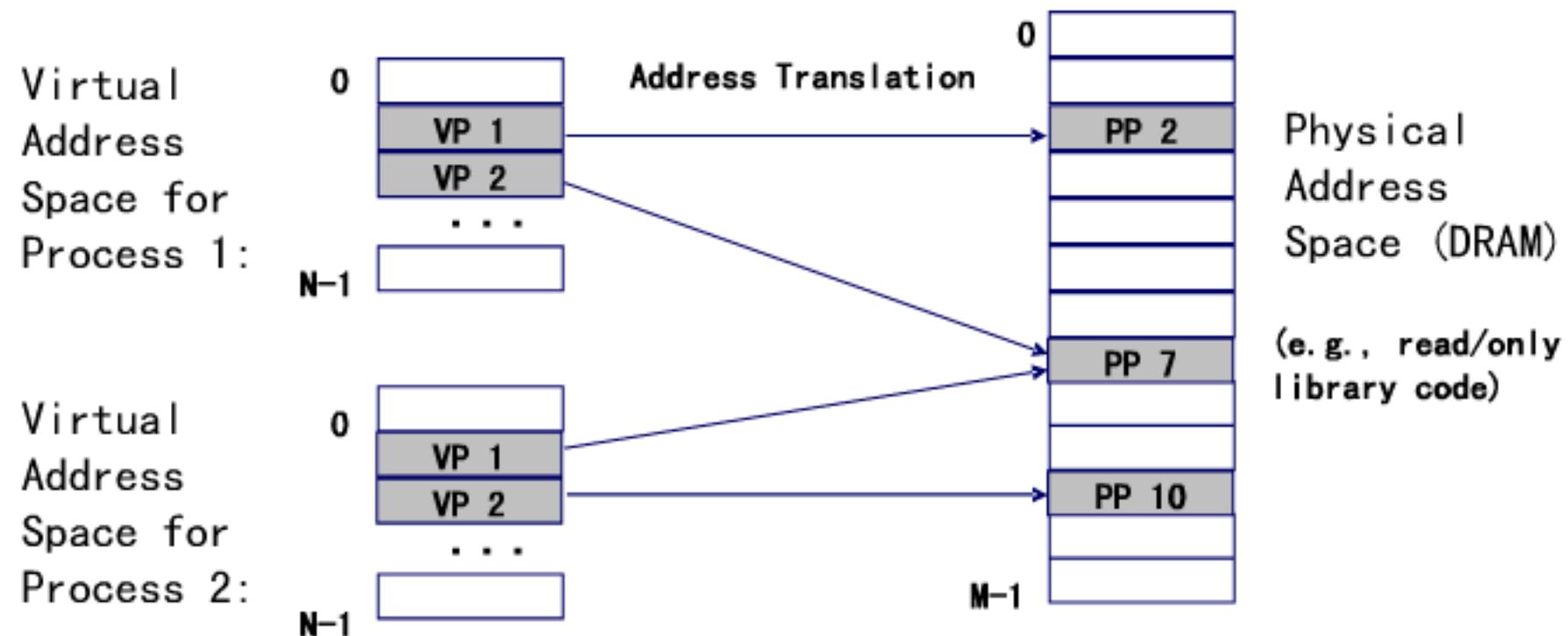


After fault



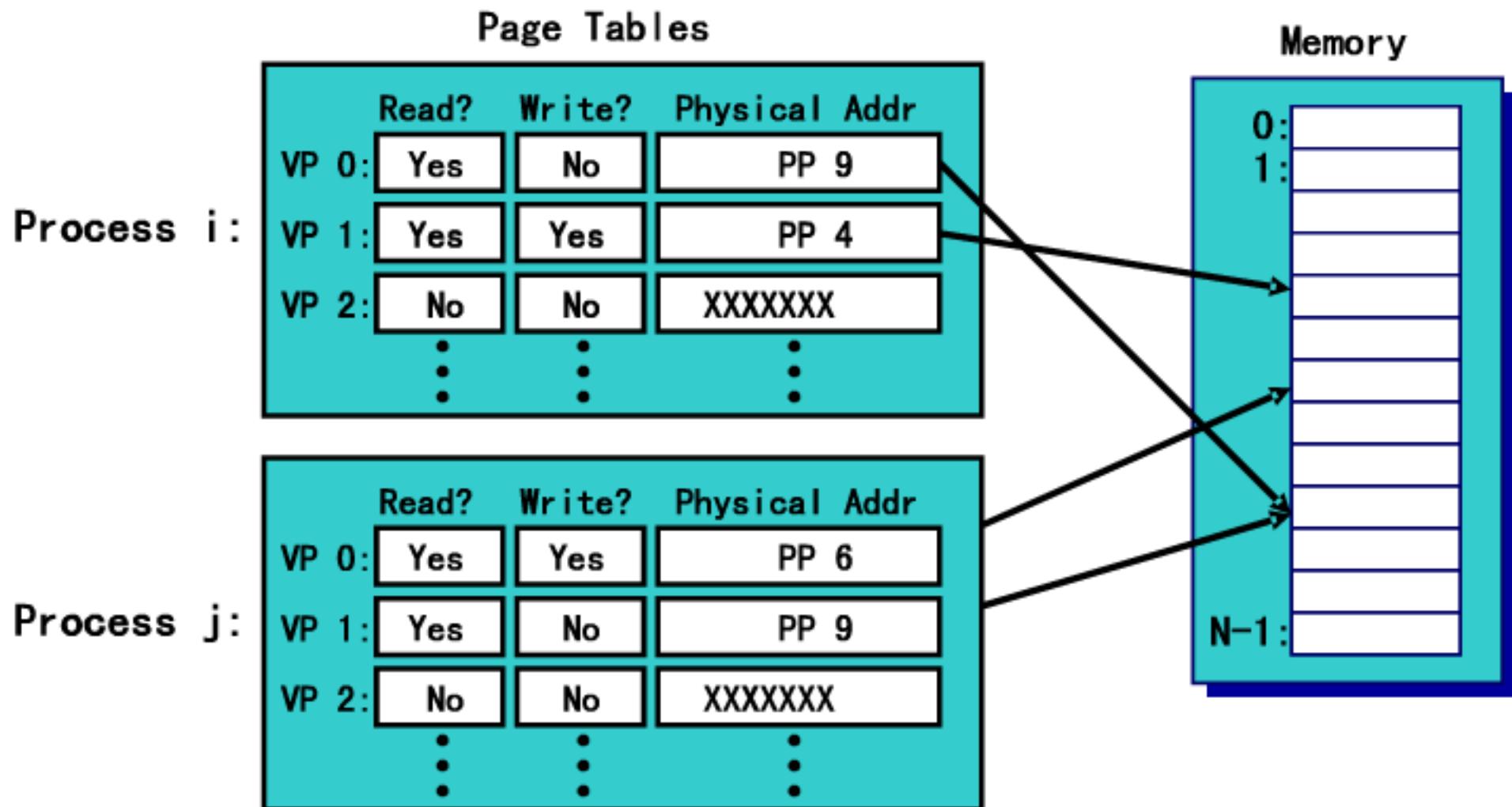
▶ 简化程序的内存管理

- 虚拟空间与物理空间都划分成相同大小的内存块（称为页）
- 每个进程都有其私有的虚拟地址空间
- 进程虚拟空间映射到物理空间

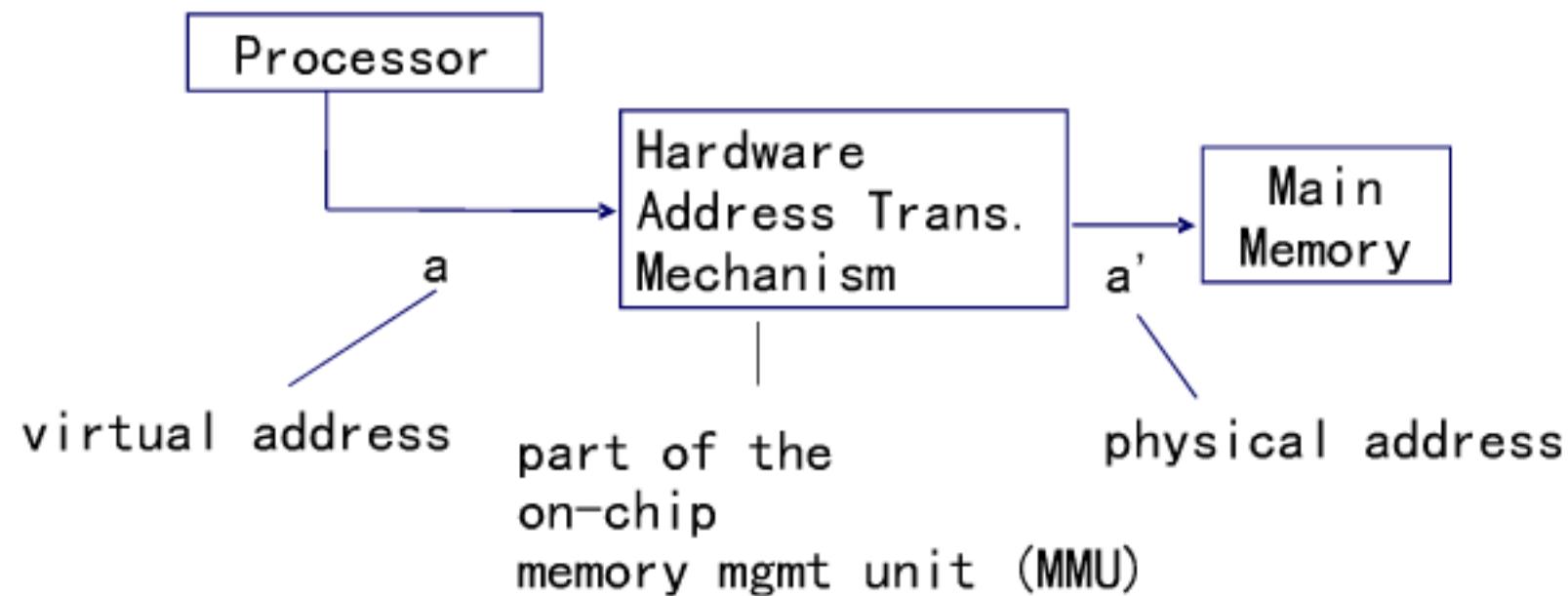


▶ 存储保护功能

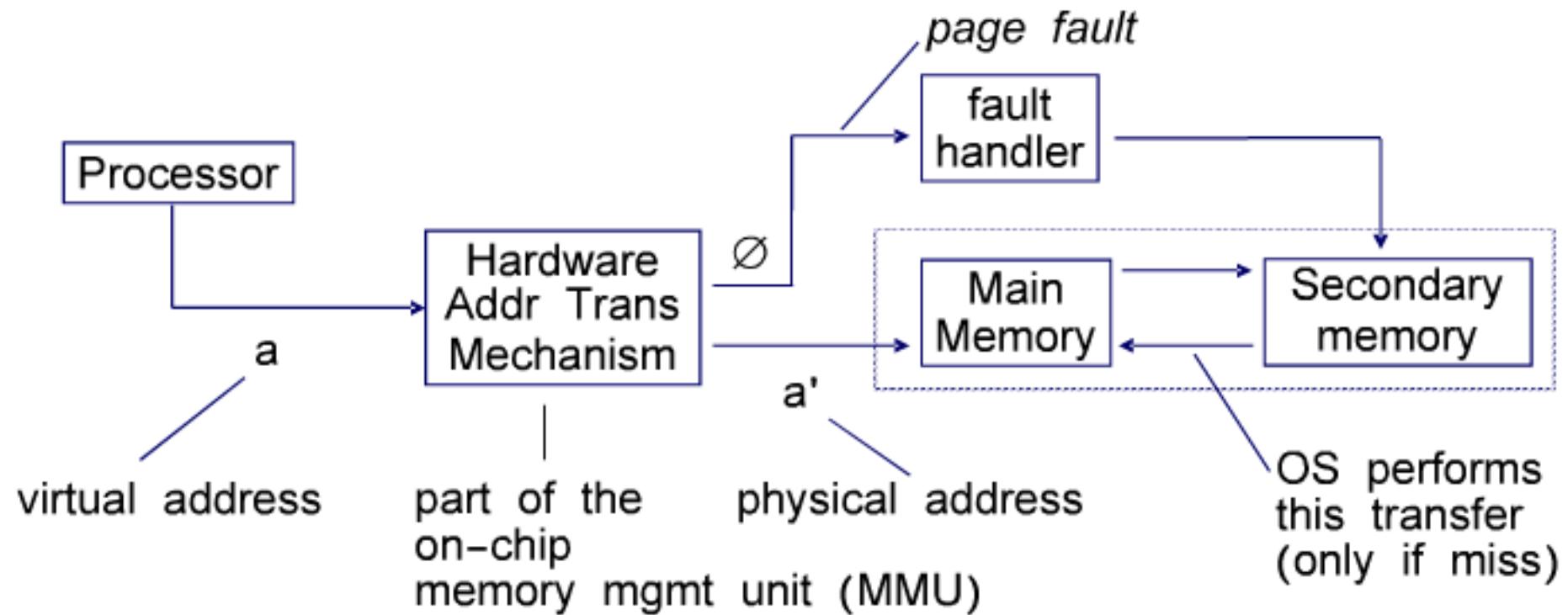
- 页表项也记录该页的访问控制信息
- 一般由硬件来完成访问控制。



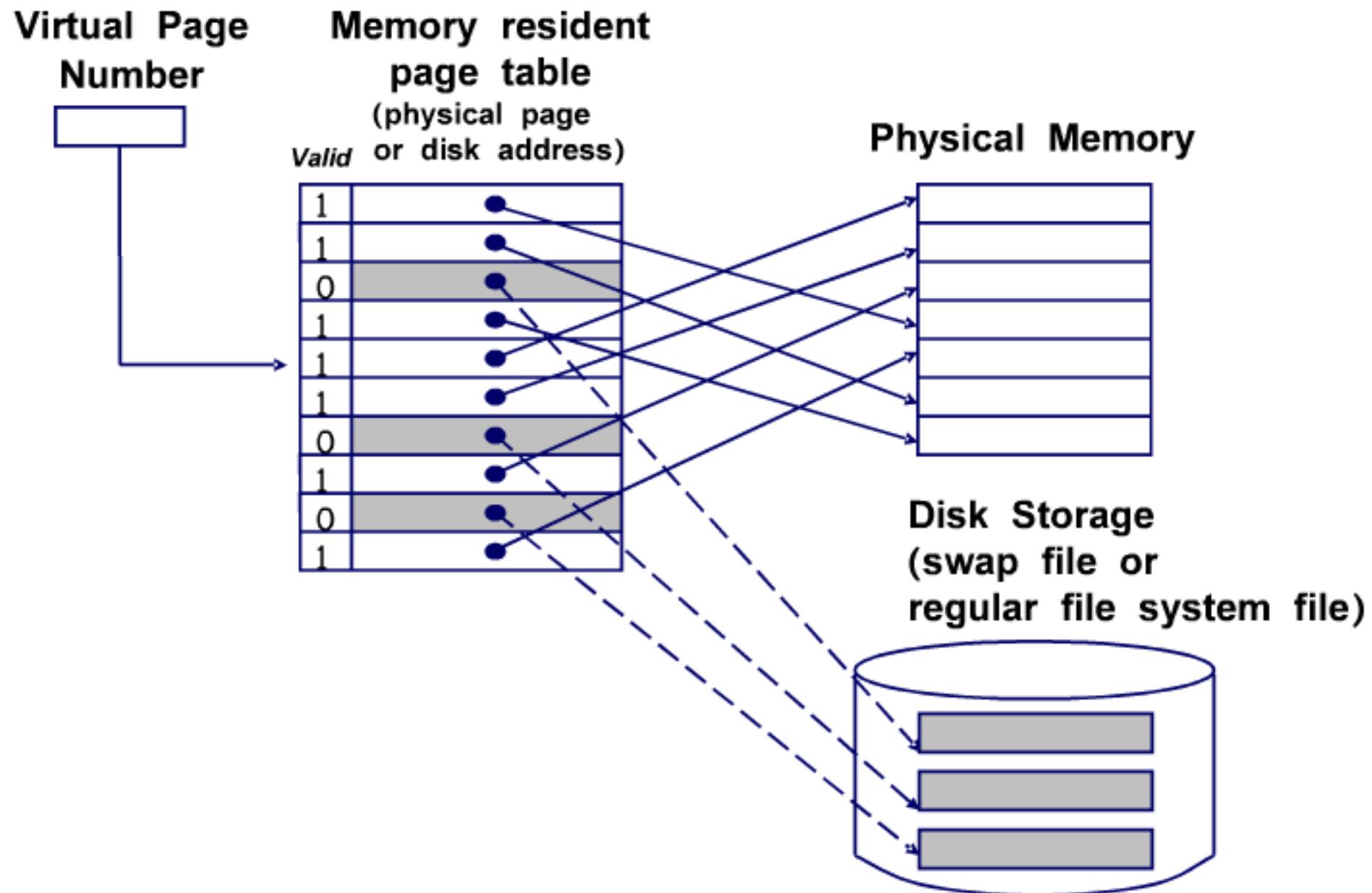
虚存地址转换：命中



虚存地址转换：页缺失



页表结构



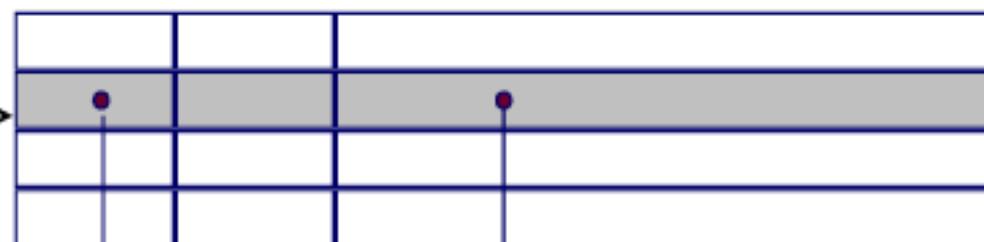
page table base register

virtual address

VPN acts
as
table index

n-1 p p-1 0
virtual page number (VPN) page offset

valid access physical page number (PPN)



if valid=0
then page
not in memory

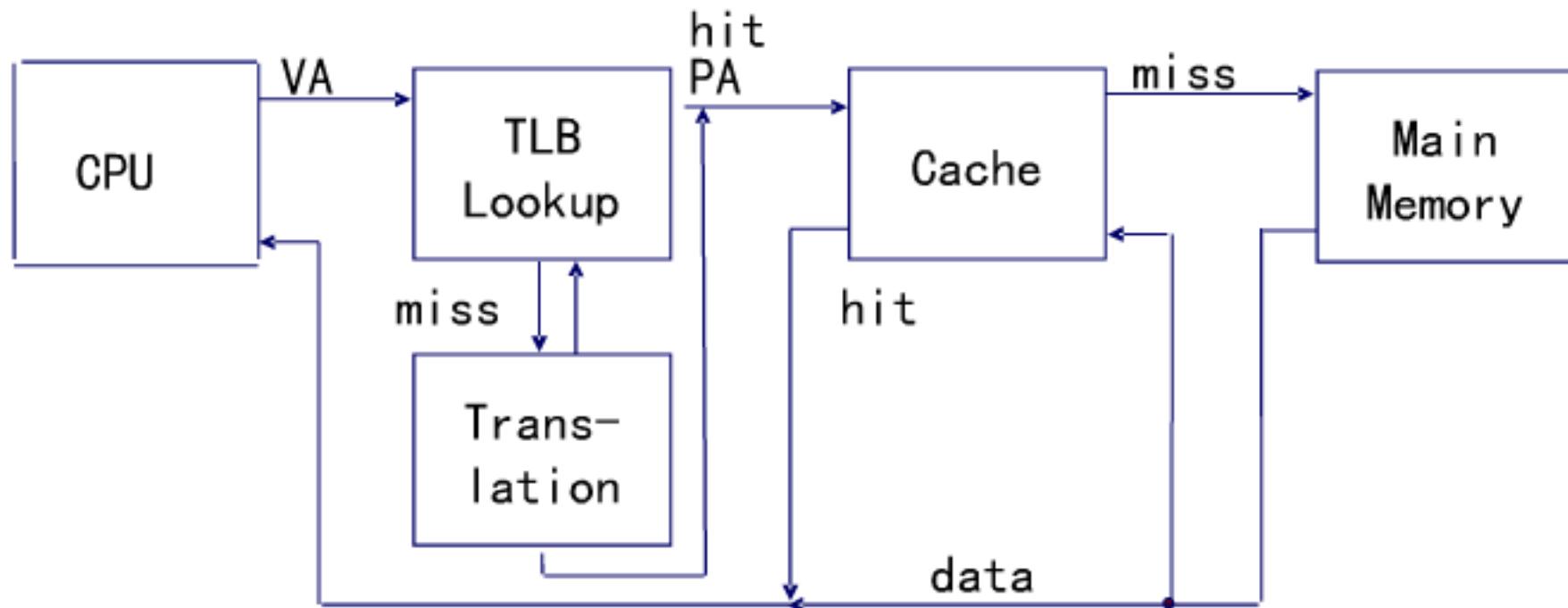
m-1 p p-1 0
physical page number (PPN) page offset

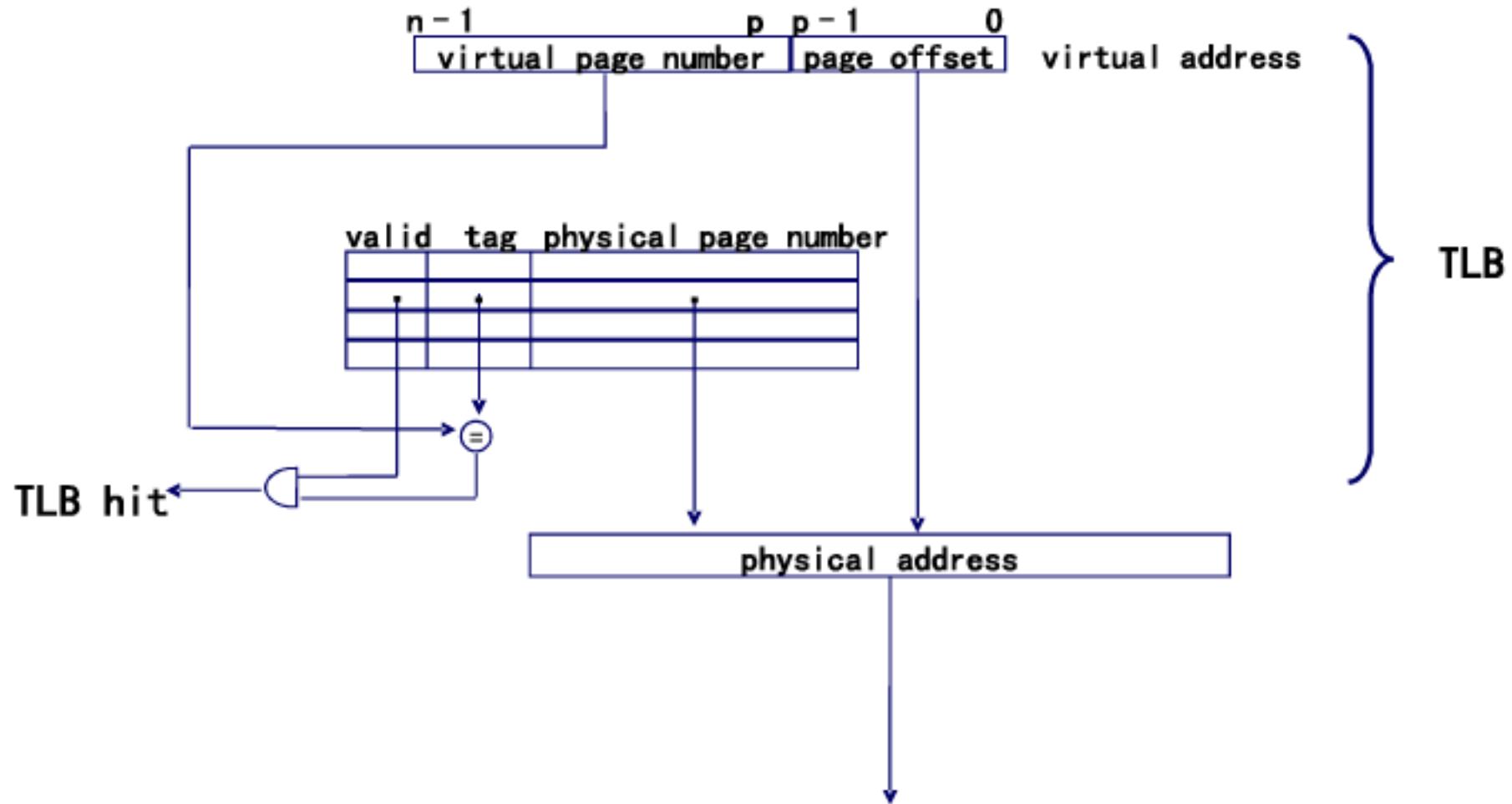
physical address

使用快表（TLB）来加速

“Translation Lookaside Buffer” (TLB)

- 处理器内存管理单元中的硬件部件
- 功能是将虚存页地址转换为物理内存地址
 - 存储了内存中页表的一个子集





汇编语言程序设计

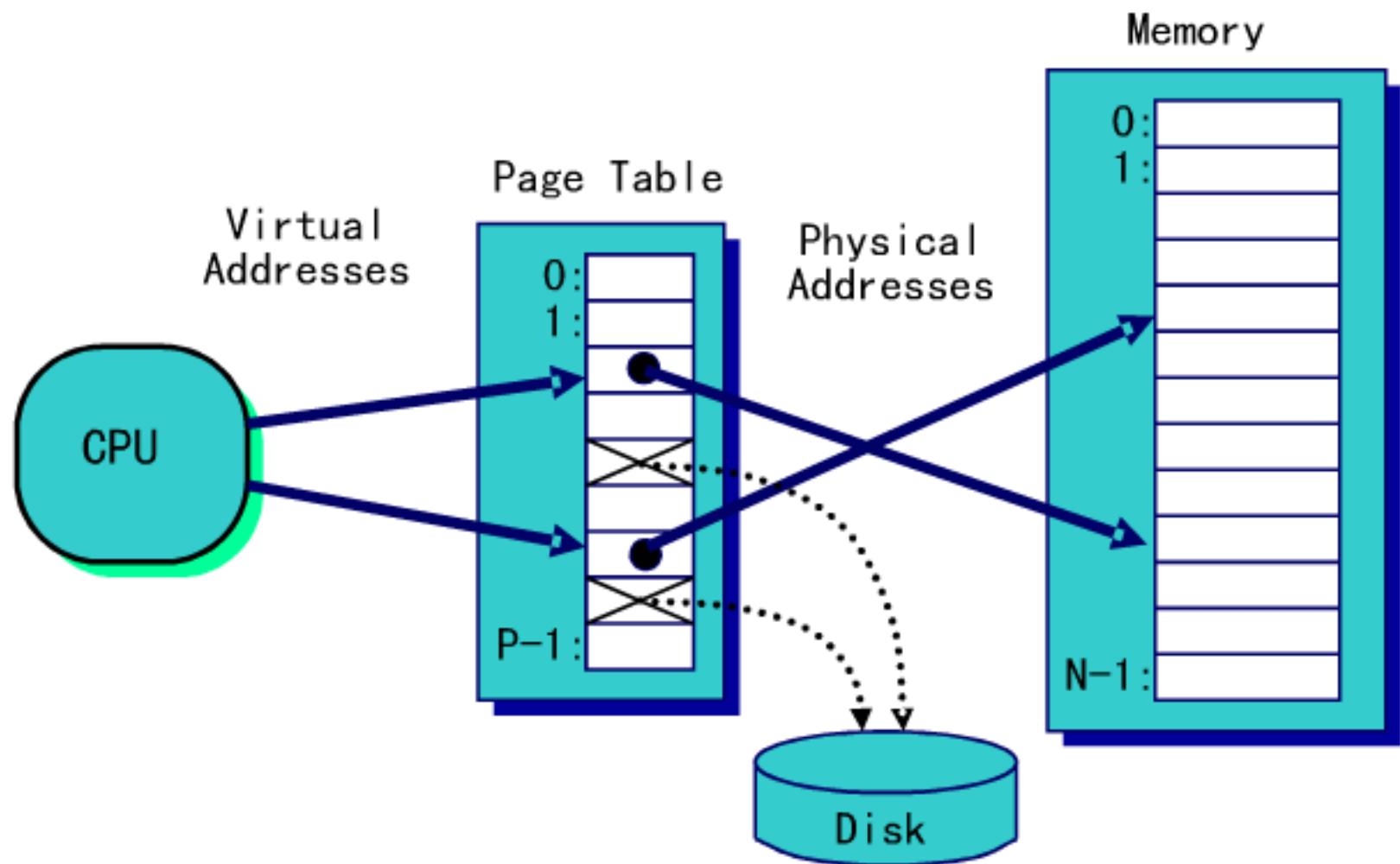
虚存与MIPS 32内存管理

- ▶ 虚存概念初步
 - 设计背景
 - 基本管理流程
- ▶ MIPS 32的内存管理

虚存设计背景

- ▶ 将物理内存作为磁盘数据的“缓存”
 - 因为进程的地址空间可能会超过物理内存的大小
- ▶ 简化程序的内存管理
 - 多个进程同时运行
 - 每个都有自己的地址空间
- ▶ 存储保护功能
 - 某进程不能访问其它进程的空间
 - 一个进程的不同空间区域具有不同的访问权限

▶ 将物理内存作为磁盘数据的快速“缓存”

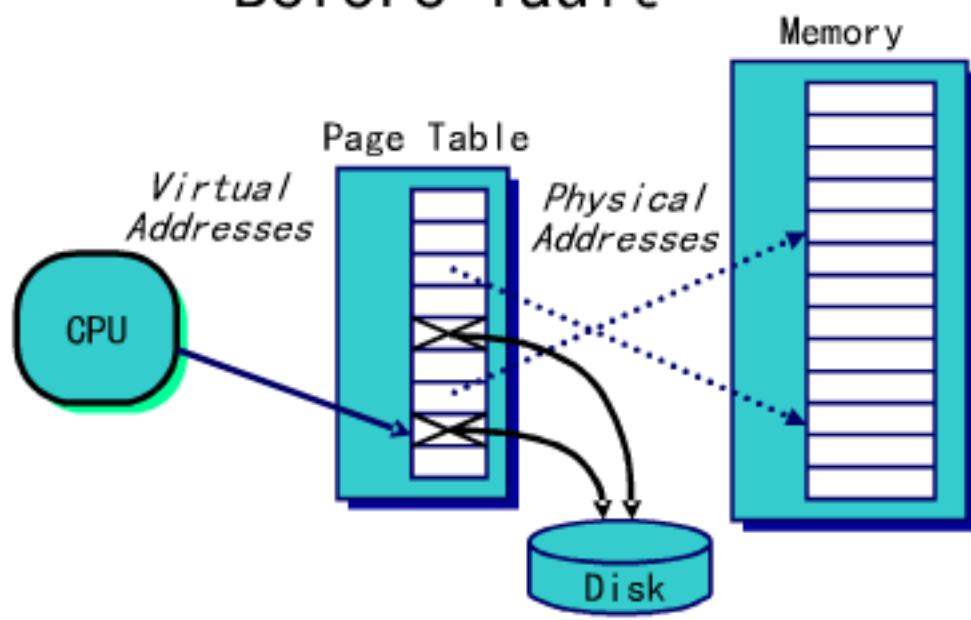


- 地址转换 (Address Translation)：一般由处理器硬件通过页表 (page table) 将虚拟地址 (程序地址) 转换为物理地址。

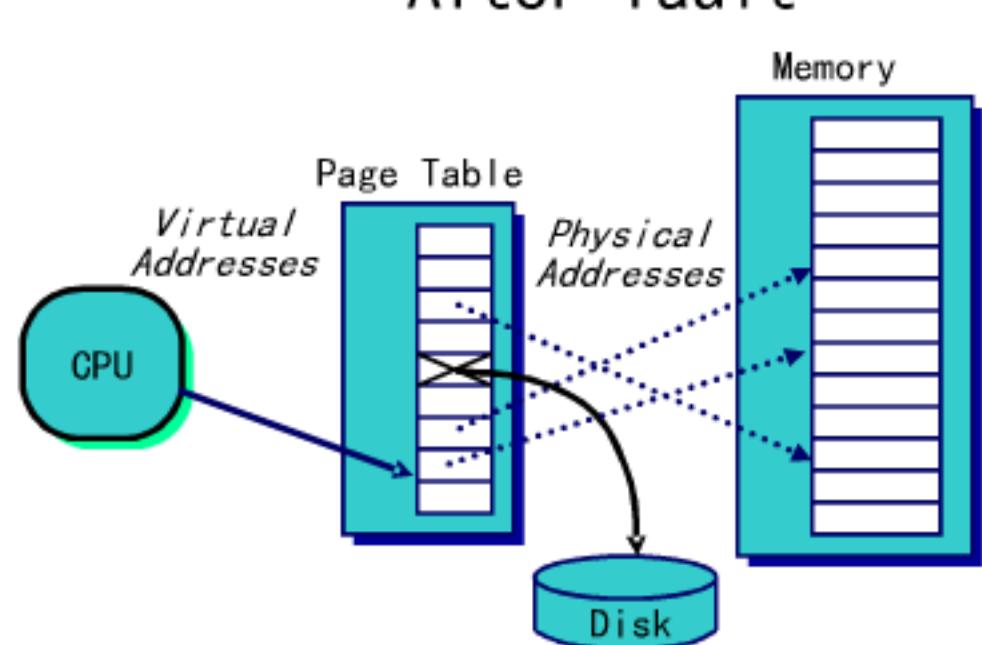
页缺失 (Page fault)

- 页表项记录某虚存地址（以页为单位）是否在物理内存中。
- 如果不是，那么出发“页缺失”异常，由异常处理代码将所需数据从外部存储（硬盘）读入内存。

Before fault

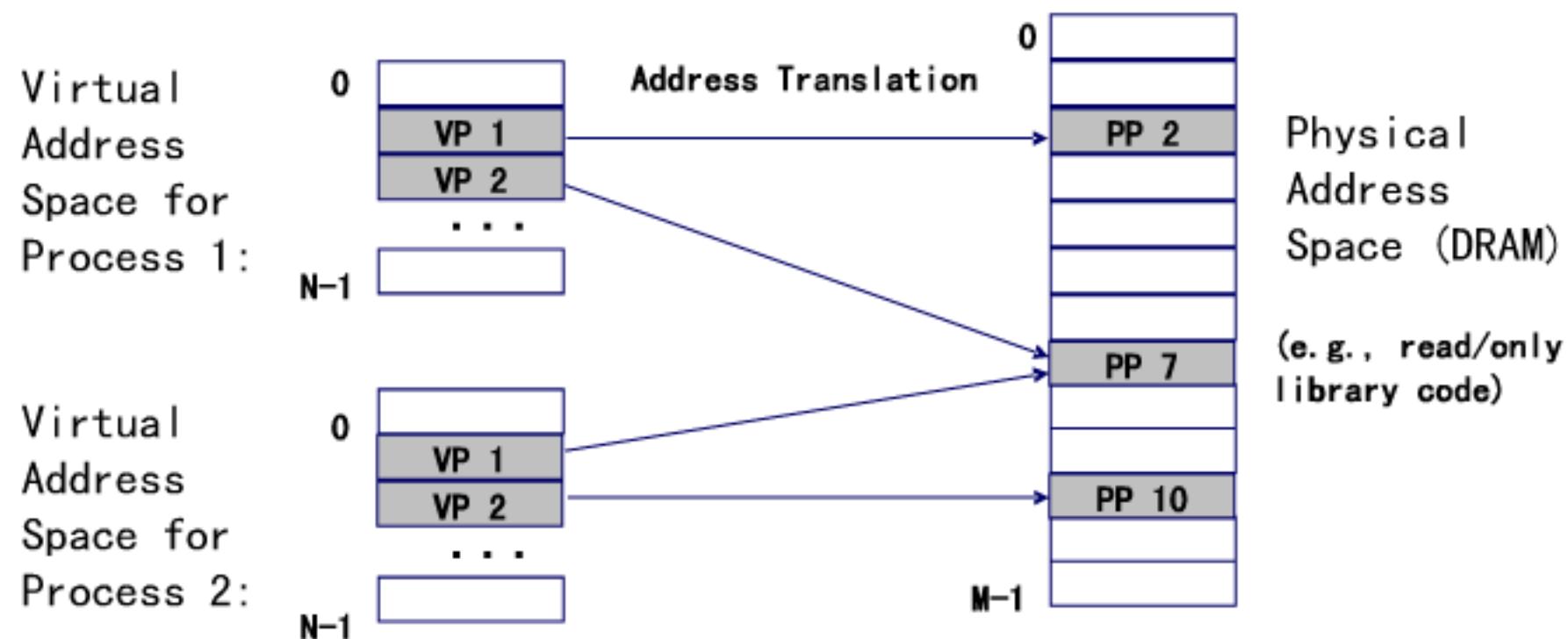


After fault



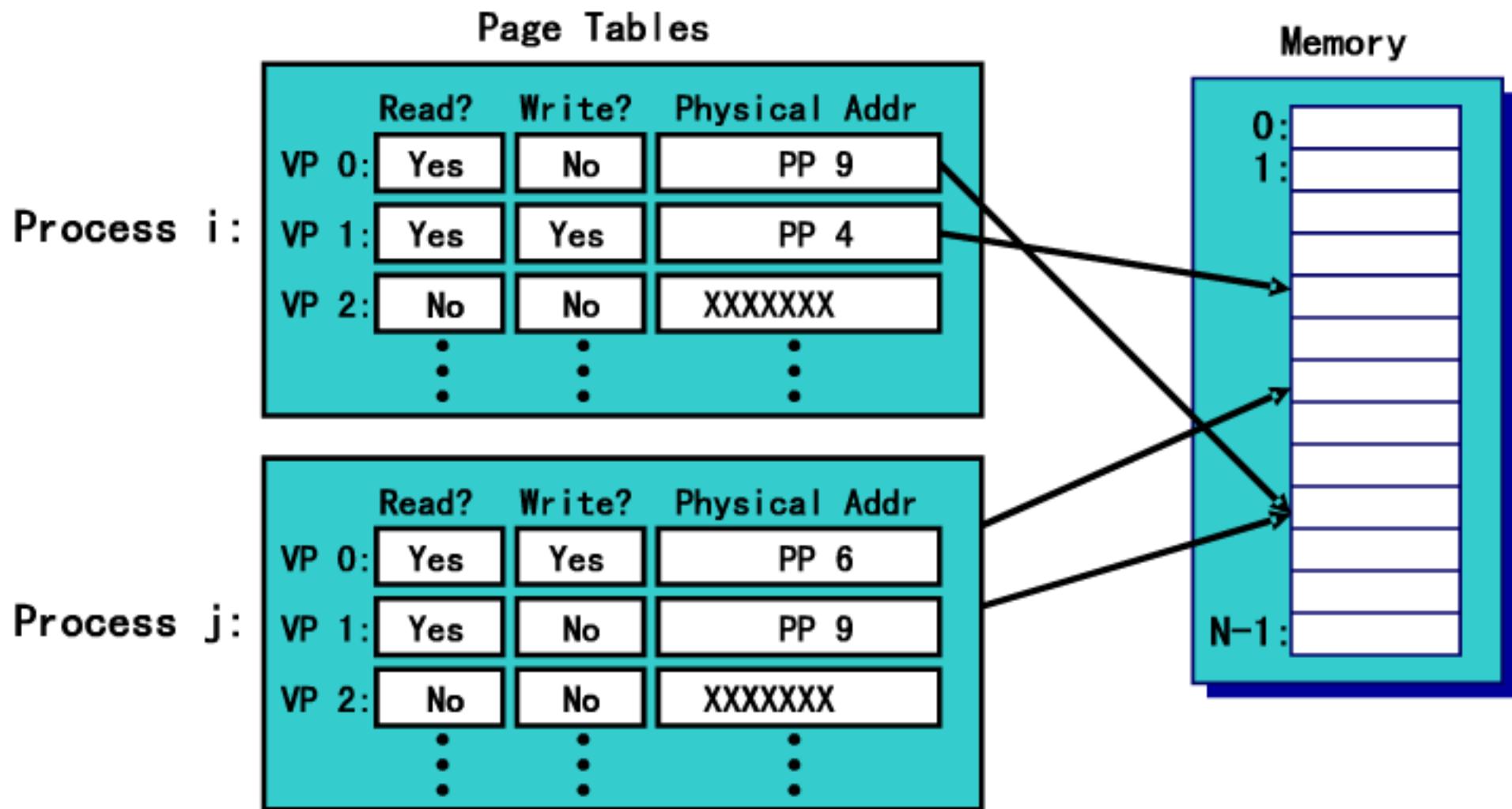
▶ 简化程序的内存管理

- 虚拟空间与物理空间都划分成相同大小的内存块（称为页）
- 每个进程都有其私有的虚拟地址空间
- 进程虚拟空间映射到物理空间

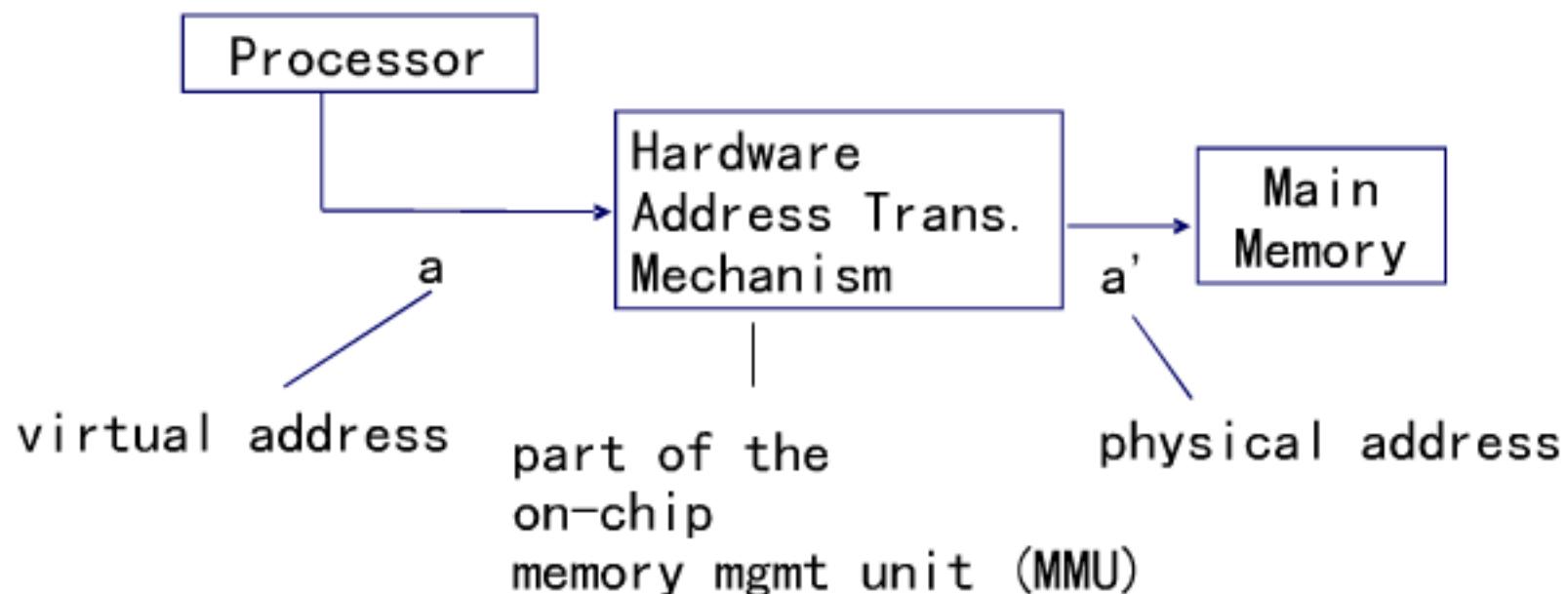


▶ 存储保护功能

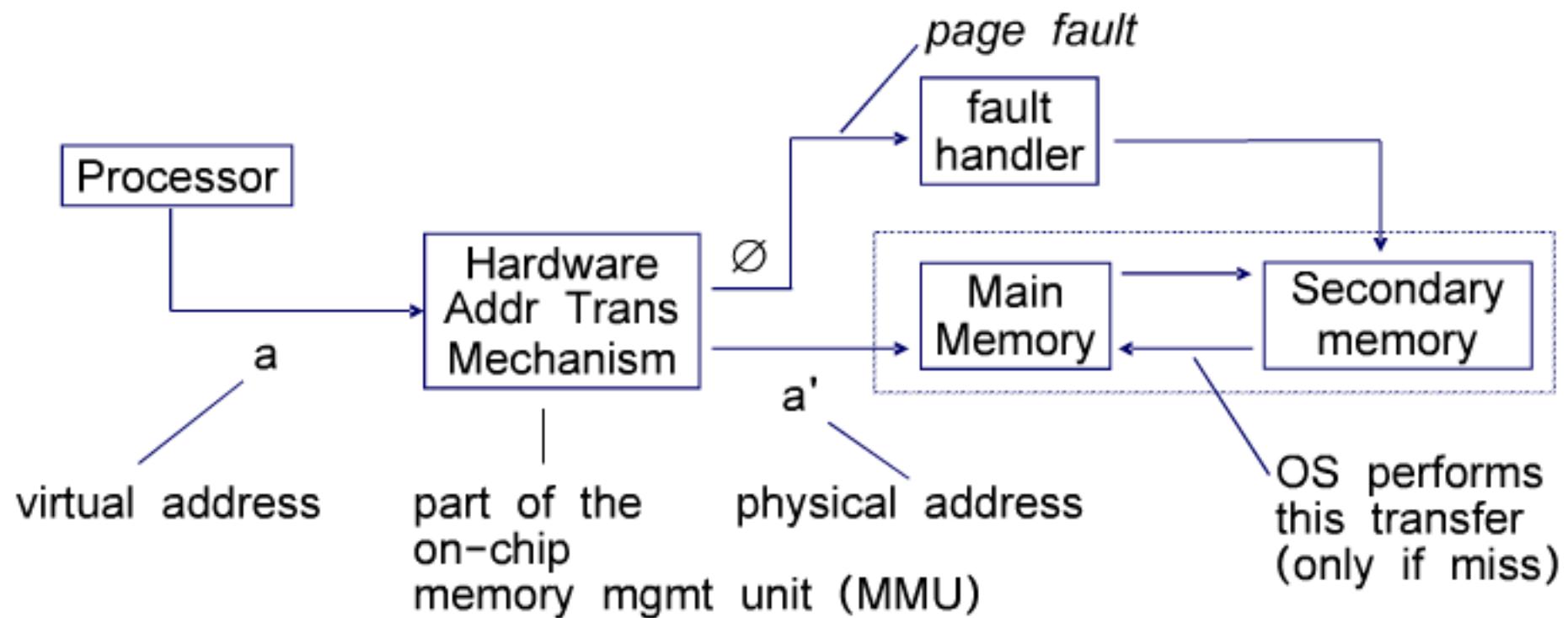
- 页表项也记录该页的访问控制信息
- 一般由硬件来完成访问控制。



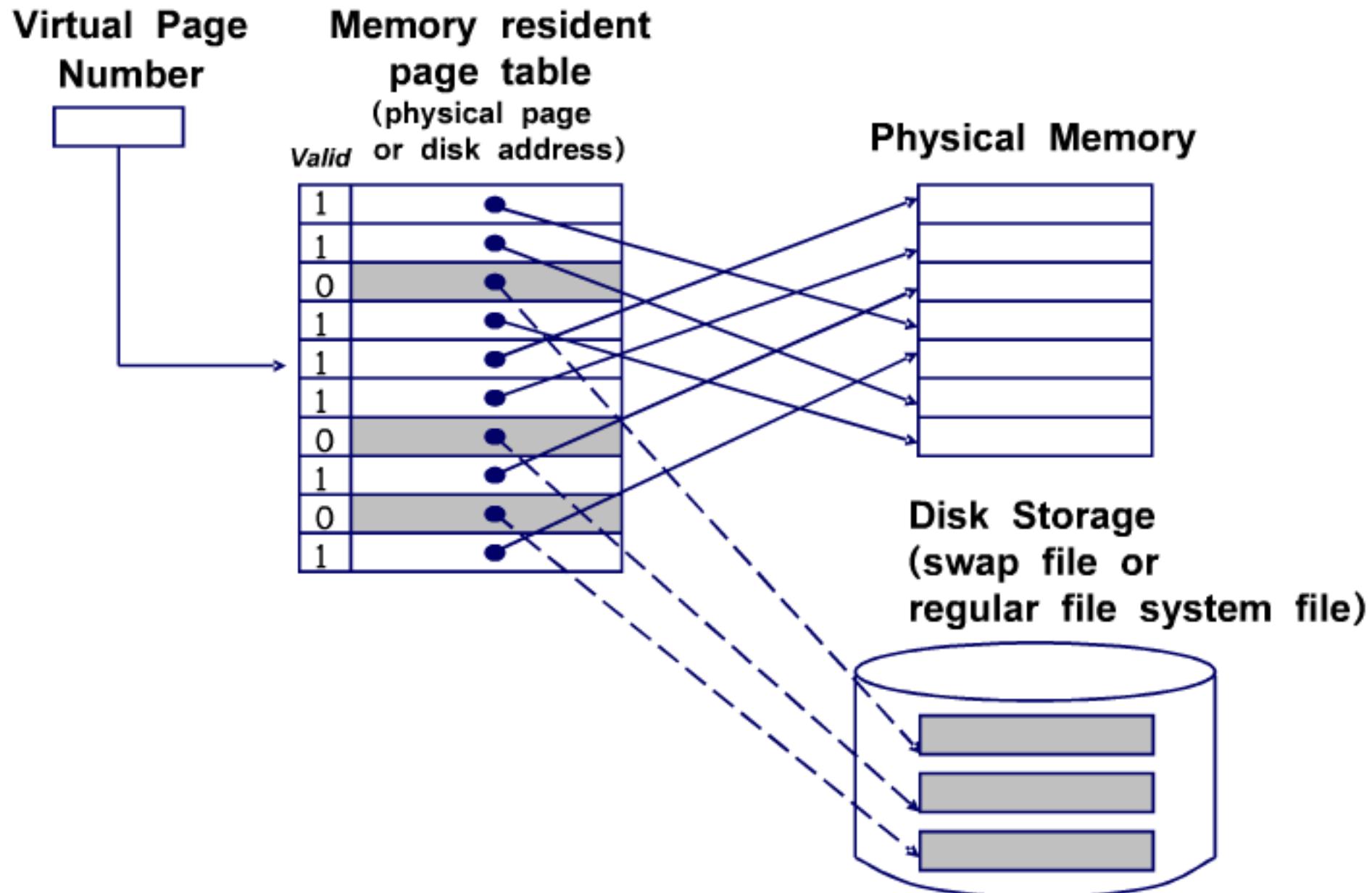
虛存地址轉換：命中



虚存地址转换：页缺失



页表结构



page table base register

virtual address

VPN acts
as
table index

n-1

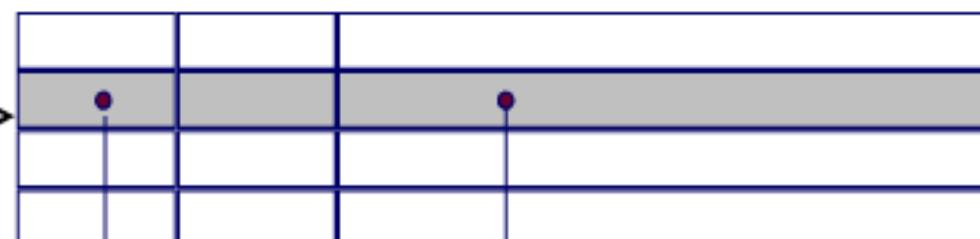
p p-1

0

virtual page number (VPN)

page offset

valid access physical page number (PPN)



if valid=0
then page
not in memory

physical page number (PPN)

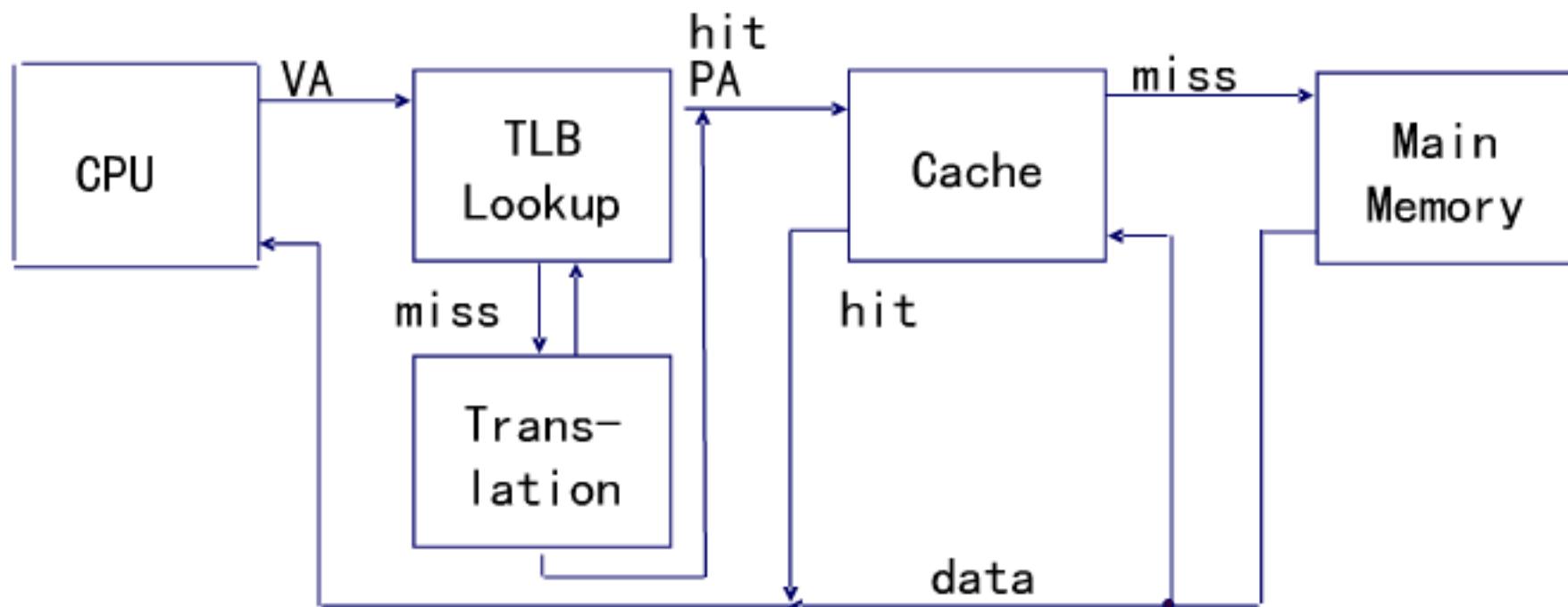
page offset

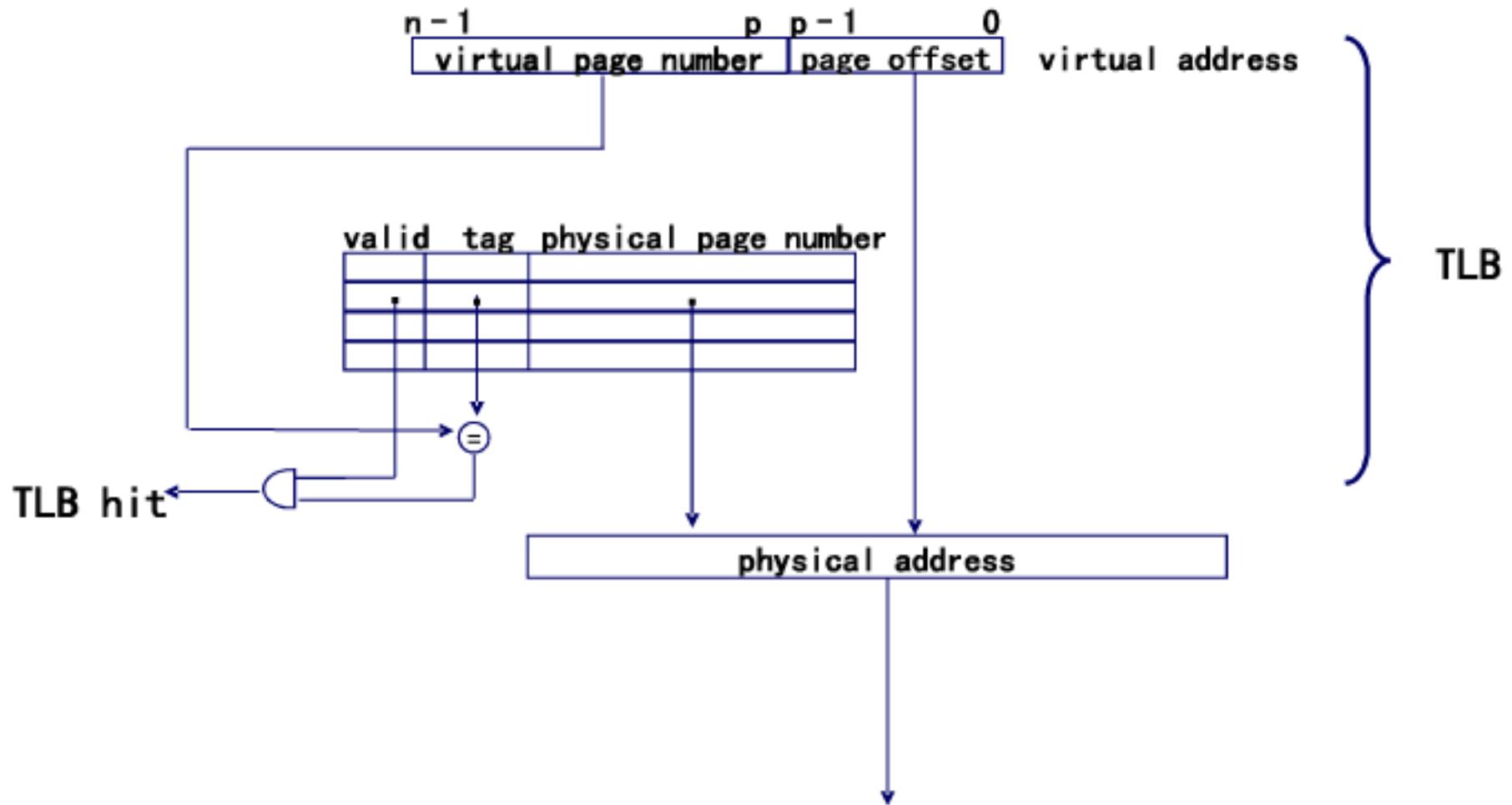
physical address

使用快表 (TLB) 来加速

“Translation Lookaside Buffer” (TLB)

- 处理器内存管理单元中的硬件部件
- 功能是将虚存页地址转换为物理内存地址
 - 存储了内存中页表的一个子集

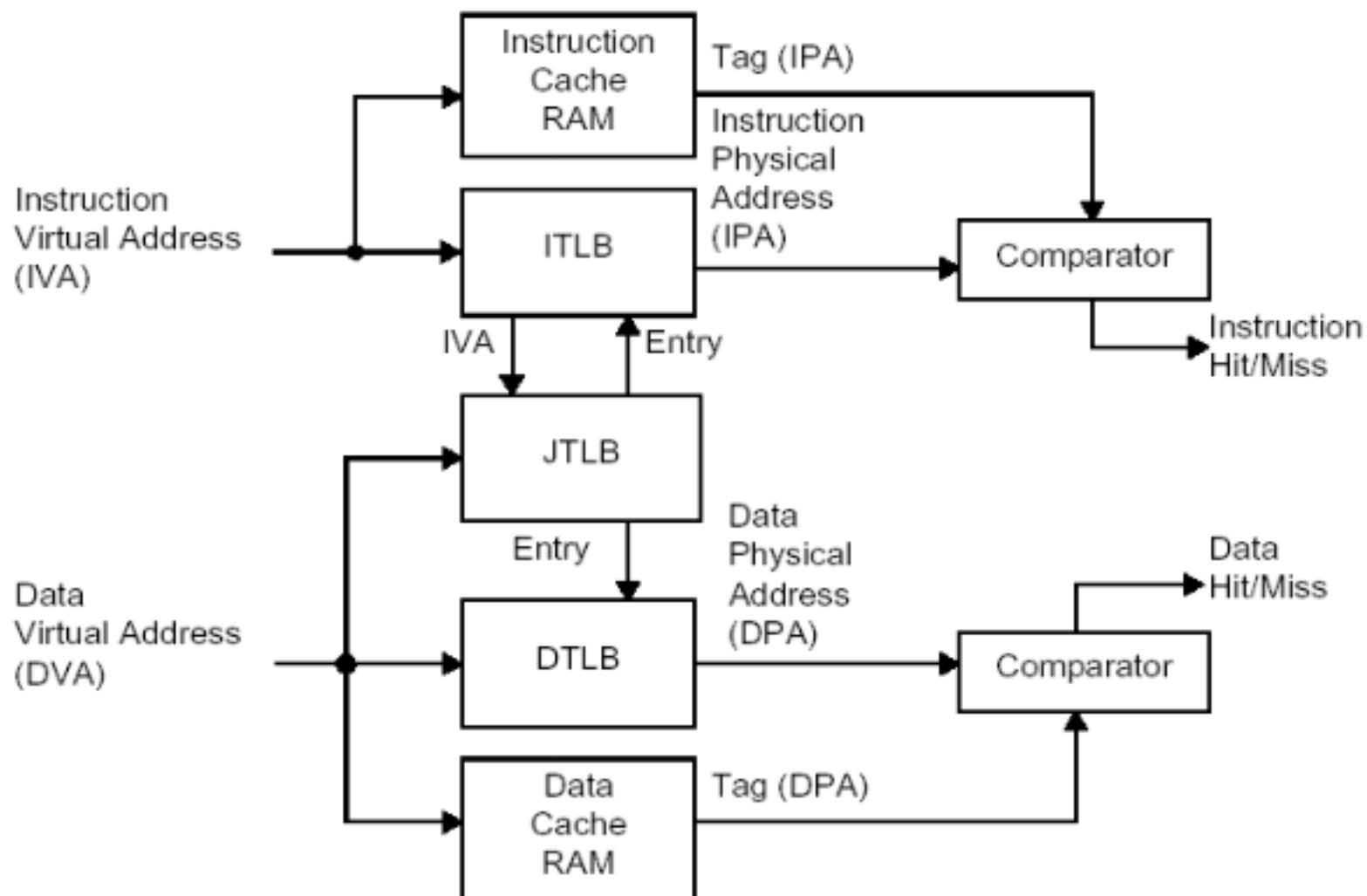




MIPS 32处理器的内存管理

MIPS处理器在执行单元和访存部件（包括缓存）间设计了存储管理单元（MMU）

- 是基于TLB实现的
- 基本功能是将数据/指令的虚拟地址送到TLB转换成物理地址



For valid address space, see the section describing Modes of operation in this chapter.



■ MMU结构

- 包括三个地址转换缓冲：一个全相联联合TLB（JTLB），一个指令TLB（ITLB），和一个数据TLB（DTLB）。

■ MMU根据所处流水部件的不同区分是指令虚拟地址还是数据虚拟地址

- 只有在用户空间（Kuseg）/ Kseg2的虚拟地址使用TLB转换，其它段的虚拟地址直接转换成物理地址

工作模式

■ 用户模式（User Mode）

- 用户模式经常用在应用程序

■ 核心模式（Kernel Mode）

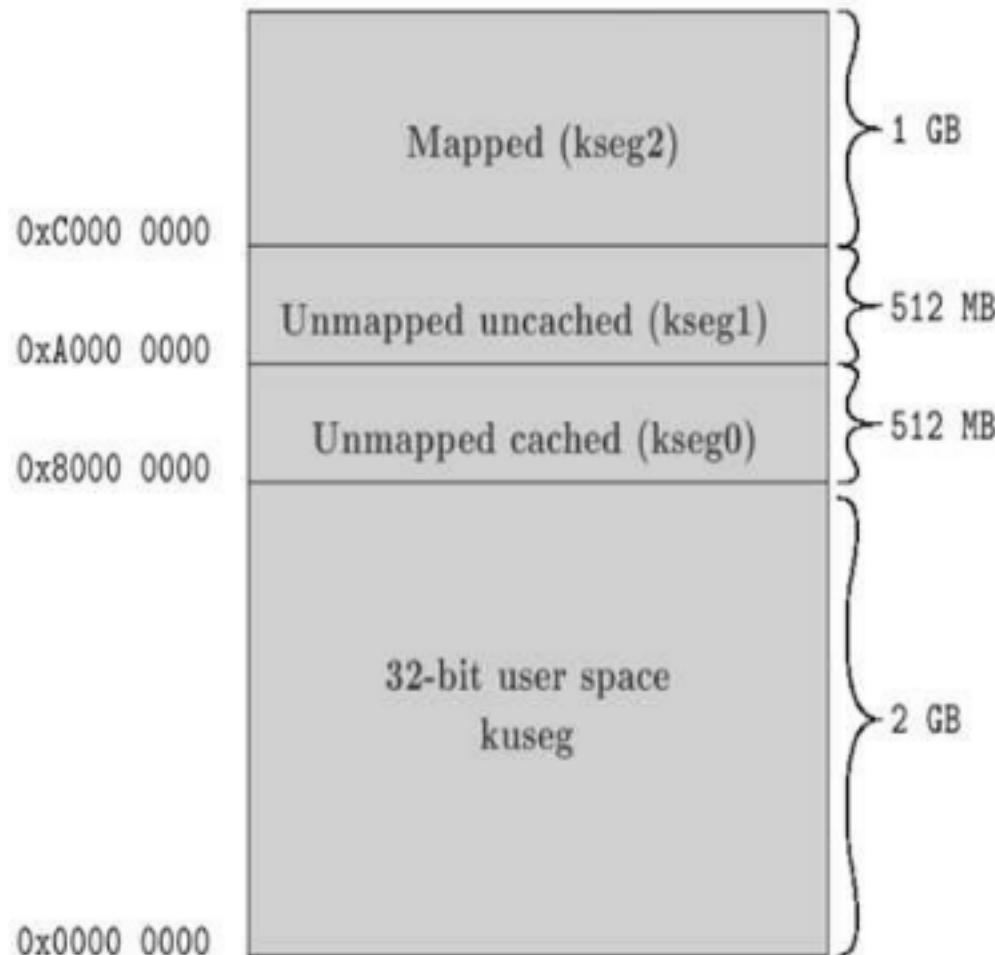
- 核心模式主要用于例外处理和特权操作系统功能（包括CPU管理和I/O设备访问）。

■ 调试模式（Debug Mode）*

- 调试模式主要用于软件调试，经常用在软件开发工具中。

■ 地址转换依赖于处理器的操作模式。在不同模式下，虚拟存储空间的划分是不同的，在将虚拟地址转换成物理地址时必须考虑到操作模式。

- 处于核心模式的软件可以访问整个地址空间，也可以访问所有的CPU寄存器。
- 用户模式仅能访问0x0000_0000到0x7FFF_FFFF的虚拟地址空间，不能进行CPU的操作。
 - 在用户模式，0x8000_0000到0xFFFF_FFFF的虚拟地址空间是无效的，对这个地址空间的访问会产生异常。



Unmapped段

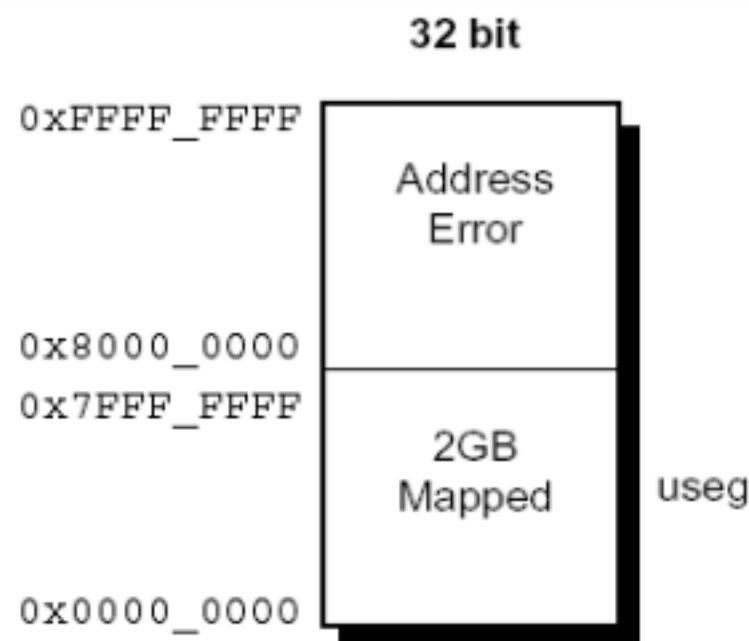
- 非映射段不使用TLB进行虚拟地址到物理地址的转换
 - 系统启动后首先进入非映射的存储段是非常重要的（why？）
 - ① 因为这时TLB还没有初始化
- Kseg1总是不缓存的（Uncached）
 - Kseg0是否缓存是由CPO的寄存器Config中的k0字段决定。

Mapped段

- 映射段使用TLB进行虚地址和物理地址的转换。
- 映射段转换是以页为基础的
 - 包含是否能缓存（Cacheable）以及页的保护特性。

用户态

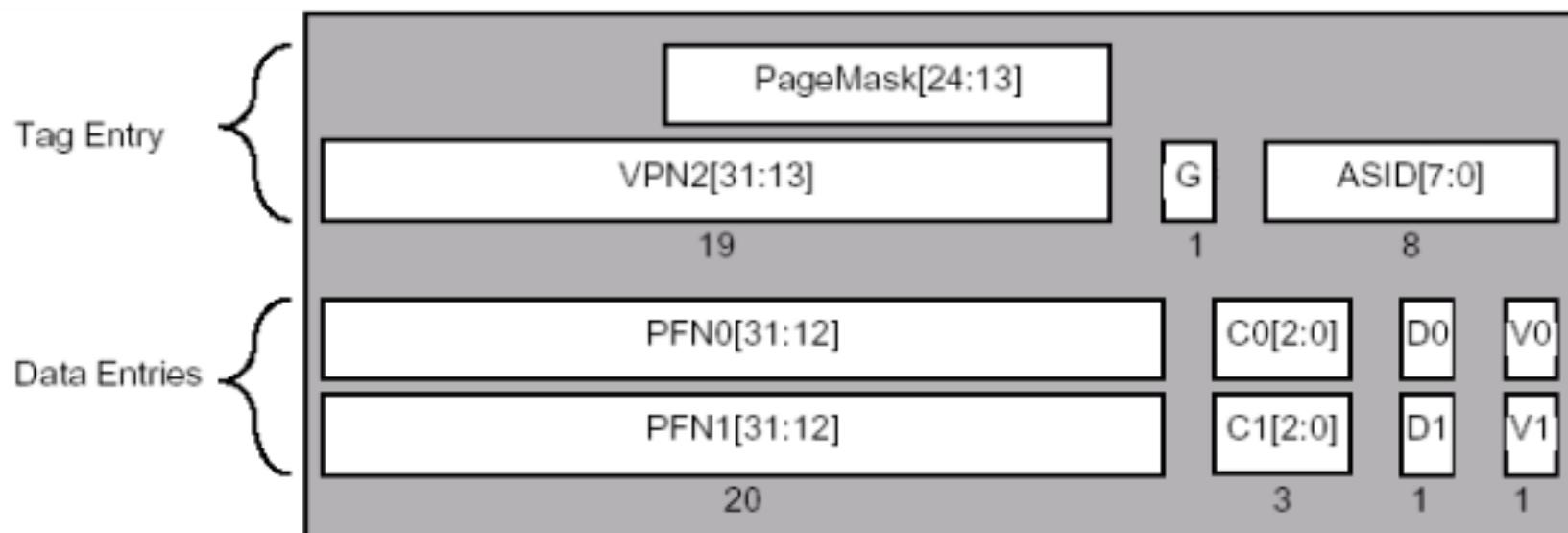
- 单一的连续2G bytes虚拟地址空间
- 系统通过TLB实现访问。在转换前，虚拟地址加上8位ASID（进程标识符）产生一个唯一的虚拟地址。



TLB结构（以MIPS 4KC处理器为例）

■ JTLB

- 全相联JTLB映射32位虚拟地址到相应的物理地址。
- JTLB包括16对奇偶项，每项包含的页的大小从4Kbytes到16Mbytes



Field Name	Description																								
PageMask[24:13]	<p>Page Mask Value. The Page Mask defines the page size by masking the appropriate VPN2 bits from being involved in a comparison. It is also used to determine which address bit is used to make the even-odd page (PFN0-PFN1) determination. See the table below.</p> <table border="1" data-bbox="763 332 1937 897"> <thead> <tr> <th data-bbox="827 340 1126 385">PageMask[11:0]</th><th data-bbox="1126 340 1532 385">Page Size</th><th data-bbox="1532 340 1937 433">Even/Odd Bank Select Bit</th></tr> </thead> <tbody> <tr> <td data-bbox="827 441 1126 486">0000_0000_0000</td><td data-bbox="1126 441 1532 486">4KB</td><td data-bbox="1532 441 1937 486">VAddr[12]</td></tr> <tr> <td data-bbox="827 505 1126 550">0000_0000_0011</td><td data-bbox="1126 505 1532 550">16KB</td><td data-bbox="1532 505 1937 550">VAddr[14]</td></tr> <tr> <td data-bbox="827 569 1126 614">0000_0000_1111</td><td data-bbox="1126 569 1532 614">64KB</td><td data-bbox="1532 569 1937 614">VAddr[16]</td></tr> <tr> <td data-bbox="827 633 1126 678">0000_0011_1111</td><td data-bbox="1126 633 1532 678">256KB</td><td data-bbox="1532 633 1937 678">VAddr[18]</td></tr> <tr> <td data-bbox="827 697 1126 742">0000_1111_1111</td><td data-bbox="1126 697 1532 742">1MB</td><td data-bbox="1532 697 1937 742">VAddr[20]</td></tr> <tr> <td data-bbox="827 761 1126 806">0011_1111_1111</td><td data-bbox="1126 761 1532 806">4MB</td><td data-bbox="1532 761 1937 806">VAddr[22]</td></tr> <tr> <td data-bbox="827 825 1126 870">1111_1111_1111</td><td data-bbox="1126 825 1532 870">16MB</td><td data-bbox="1532 825 1937 870">VAddr[24]</td></tr> </tbody> </table> <p>The PageMask column above show all the legal values for PageMask. Because each pair of bits can only have the same value, the physical entry in the JTLB will only save a compressed version of the PageMask using only 6 bits. This is however transparent to software, which will always work with a 12 bit field.</p>	PageMask[11:0]	Page Size	Even/Odd Bank Select Bit	0000_0000_0000	4KB	VAddr[12]	0000_0000_0011	16KB	VAddr[14]	0000_0000_1111	64KB	VAddr[16]	0000_0011_1111	256KB	VAddr[18]	0000_1111_1111	1MB	VAddr[20]	0011_1111_1111	4MB	VAddr[22]	1111_1111_1111	16MB	VAddr[24]
PageMask[11:0]	Page Size	Even/Odd Bank Select Bit																							
0000_0000_0000	4KB	VAddr[12]																							
0000_0000_0011	16KB	VAddr[14]																							
0000_0000_1111	64KB	VAddr[16]																							
0000_0011_1111	256KB	VAddr[18]																							
0000_1111_1111	1MB	VAddr[20]																							
0011_1111_1111	4MB	VAddr[22]																							
1111_1111_1111	16MB	VAddr[24]																							
VPN2[31:13]	Virtual Page Number divided by 2. This field contains the upper bits of the virtual page number. Because it represents a pair of TLB pages, it is divided by 2. Bits 31:25 are always included in the TLB lookup comparison. Bits 24:13 are included depending on the page size, defined by PageMask.																								
G	Global Bit. When set, indicates that this entry is global to all processes and/or threads and thus disables inclusion of the ASID in the comparison.																								
ASID[7:0]	Address Space Identifier. Identifies which process or thread this TLB entry is associated with.																								

Field Name	Description																				
PFN0[31:12], PFN1[31:12]	Physical Frame Number. Defines the upper bits of the physical address. For page sizes larger than 4 KBytes, only a subset of these bits is actually used.																				
C0[2:0], C1[2:0]	<p>Cacheability. Contains an encoded value of the cacheability attributes and determines whether the page should be placed in the cache or not. The field is encoded as follows:</p> <table border="1" data-bbox="968 454 1773 1180"> <thead> <tr> <th data-bbox="968 454 1148 524">C[2:0]</th><th data-bbox="1148 454 1773 524">Coherency Attribute</th></tr> </thead> <tbody> <tr> <td data-bbox="968 524 1148 595">000</td><td data-bbox="1148 524 1773 595">Maps to entry 011b*</td></tr> <tr> <td data-bbox="968 595 1148 665">001</td><td data-bbox="1148 595 1773 665">Maps to entry 011b*</td></tr> <tr> <td data-bbox="968 665 1148 736">010</td><td data-bbox="1148 665 1773 736">Uncached</td></tr> <tr> <td data-bbox="968 736 1148 822">011</td><td data-bbox="1148 736 1773 822">Cacheable, noncoherent, write-through, no write allocated</td></tr> <tr> <td data-bbox="968 822 1148 892">100</td><td data-bbox="1148 822 1773 892">Maps to entry 011b*</td></tr> <tr> <td data-bbox="968 892 1148 963">101</td><td data-bbox="1148 892 1773 963">Maps to entry 011b*</td></tr> <tr> <td data-bbox="968 963 1148 1033">110</td><td data-bbox="1148 963 1773 1033">Maps to entry 011b*</td></tr> <tr> <td data-bbox="968 1033 1148 1088">111</td><td data-bbox="1148 1033 1773 1088">Maps to entry 010b*</td></tr> <tr> <td colspan="2" data-bbox="968 1088 1773 1180">Note: * These mappings are not used on the 4K processor cores but do have meaning in other MIPS</td></tr> </tbody> </table>	C[2:0]	Coherency Attribute	000	Maps to entry 011b*	001	Maps to entry 011b*	010	Uncached	011	Cacheable, noncoherent, write-through, no write allocated	100	Maps to entry 011b*	101	Maps to entry 011b*	110	Maps to entry 011b*	111	Maps to entry 010b*	Note: * These mappings are not used on the 4K processor cores but do have meaning in other MIPS	
C[2:0]	Coherency Attribute																				
000	Maps to entry 011b*																				
001	Maps to entry 011b*																				
010	Uncached																				
011	Cacheable, noncoherent, write-through, no write allocated																				
100	Maps to entry 011b*																				
101	Maps to entry 011b*																				
110	Maps to entry 011b*																				
111	Maps to entry 010b*																				
Note: * These mappings are not used on the 4K processor cores but do have meaning in other MIPS																					
D0, D1	“Dirty” or Write-enable Bit. Indicates that the page has been written, and/or is writable. If this bit is set, stores to the page are permitted. If the bit is cleared, stores to the page cause a TLB Modified exception.																				
V0, V1	Valid Bit. Indicates that the TLB entry and, thus, the virtual page mapping are valid. If this bit is set, accesses to the page are permitted. If the bit is cleared, accesses to the page cause a TLB Invalid exception.																				

- 操作JTLB

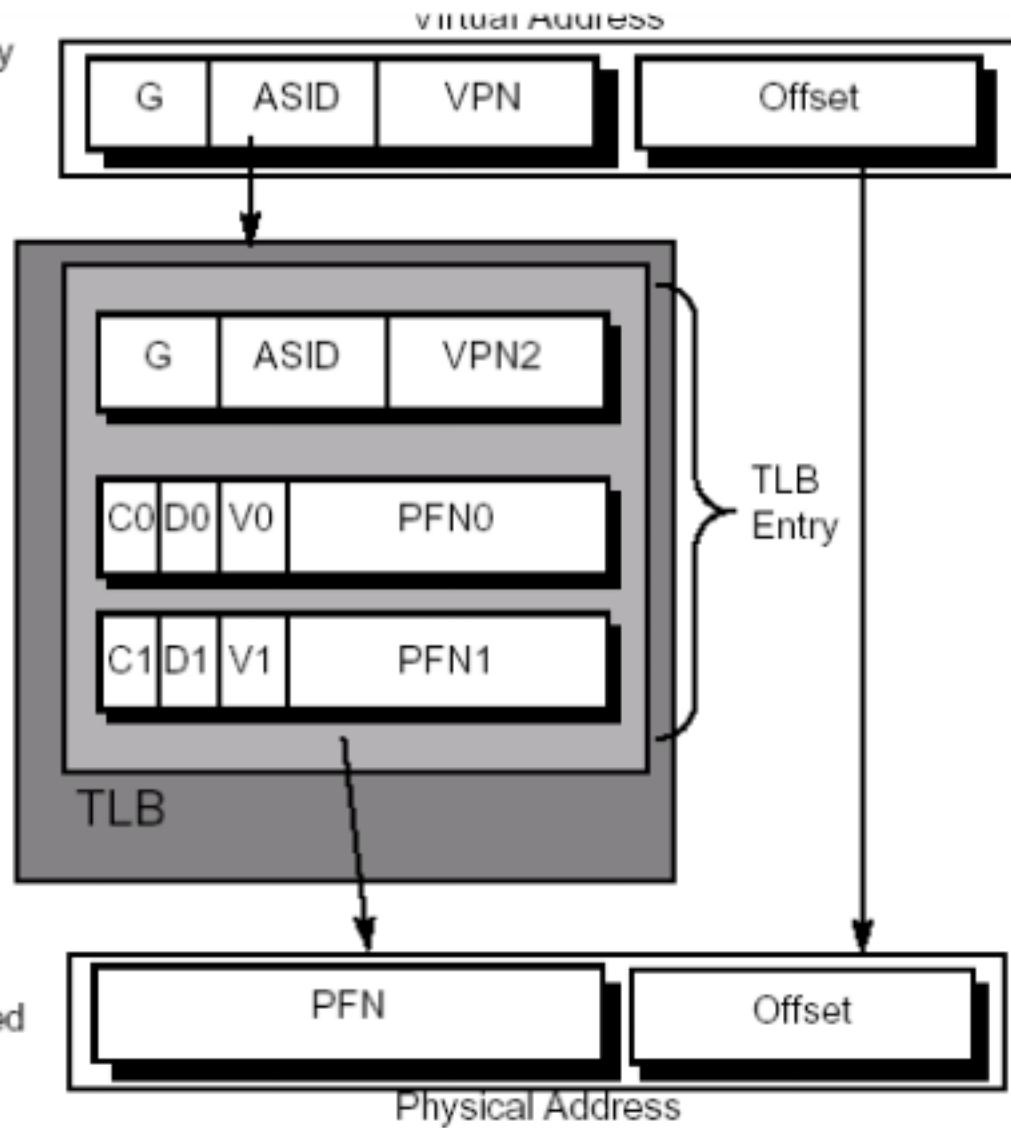
- ① 软件通过TLBWI和TLBWR指令填充JTLB的某个入口
- ② 在填充TLB之前，首先用CPO指令更新和TLB相关的CPO寄存器，然后用相关CPO寄存器的内容填充TLB入口的不同字段

- ITLB & DTLB

- 硬件管理，对软件透明。
- 如果虚拟地址在ITLB或者DTLB中缺失，则由JTLB在下一个周期处理；后者如果成功，则将相应表项复制到ITLB（或者DTLB）。

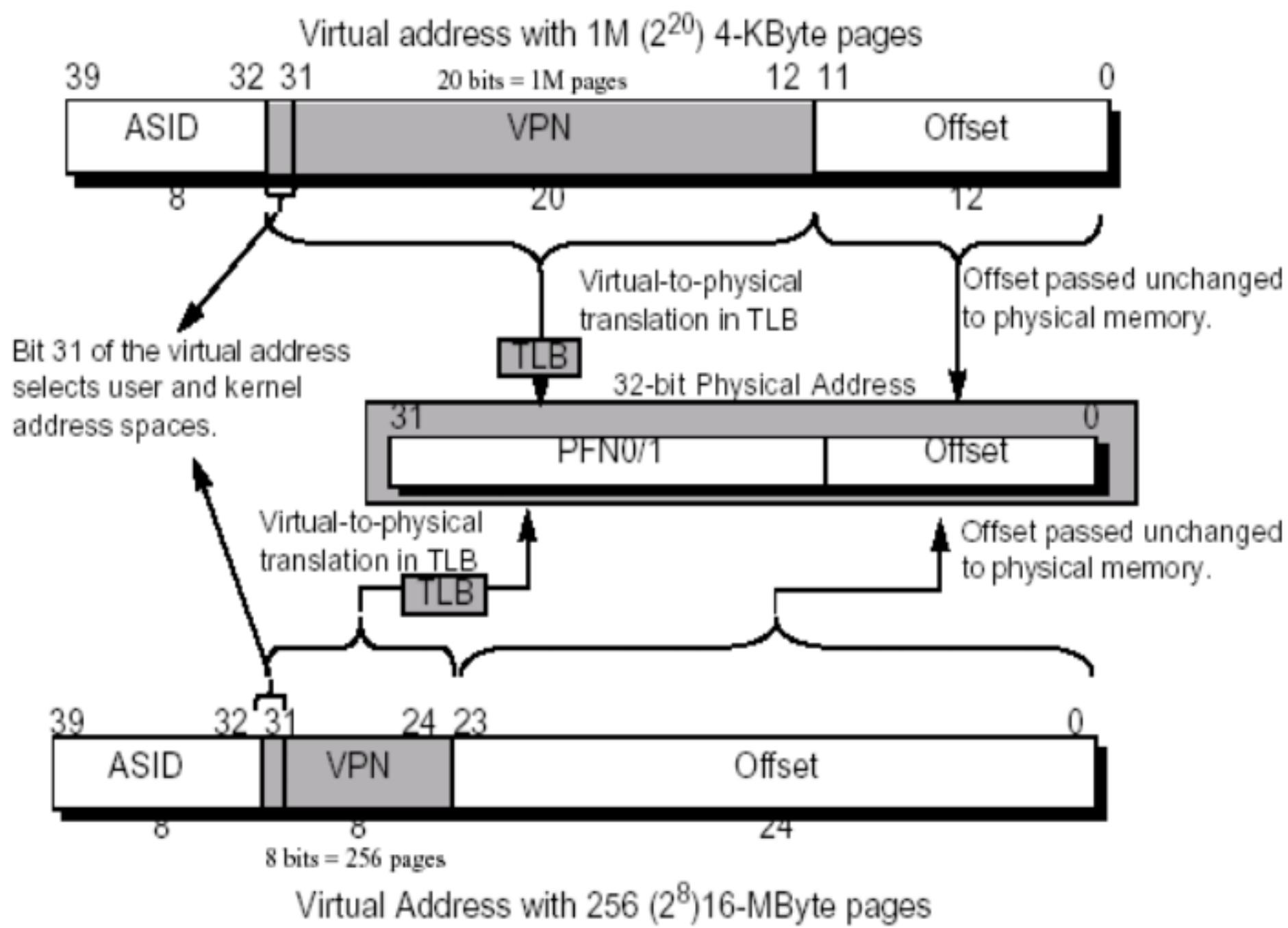
虚拟地址到物理地址的详细转换流程

1. Virtual address (VA) represented by the virtual page number (VPN) is compared with tag in TLB.



3. The Offset, which does not pass through the TLB, is then concatenated with the PFN.

如果发生TLB缺失异常，软件从内存页表中读取相关表项重填TLB。



■ TLB缺失和多匹配

- 如果TLB项没有匹配（TLB缺失），异常发生
 - ⌚ 软件从驻留在内存中的页表重填TLB。
 - ⌚ 软件可以写一个选定的TLB项或者利用硬件机制写到一个随机的项里（CPO的Random寄存器选择哪个TLB项用于TLBWR）
- 通过TLB写比较机制来保证多TLB项匹配不能发生
 - ⌚ 在TLB写操作中，要写的VPN2域与TLB中的所有其它项比较。如果有一个匹配发生，处理器引发一个机器检查例外（machine—check exception）

TLB操作指令*

名称	格式	功能描述	其它
TLBP	TLBP	To find a matching entry in the TLB	The <i>Index</i> register is loaded with the address of the TLB entry whose contents match the contents of the <i>EntryHi</i> register.
TLBR	TLBR	To read an entry from the TLB	The <i>EntryHi</i> , <i>EntryLo_0</i> , <i>EntryLo_1</i> , and <i>PageMask</i> registers are loaded with the contents of the TLB entry pointed to by the <i>Index</i> register.
TLBWI	TLBWI	To write a TLB entry indexed by the <i>Index</i> register	The TLB entry pointed to by the <i>Index</i> register is written from the contents of the <i>EntryHi</i> , <i>EntryLo_0</i> , <i>EntryLo_1</i> , and <i>PageMask</i> registers.
TLBWR	TLBWR	To write a TLB entry indexed by the <i>Random</i> register.	The TLB entry pointed to by the <i>Random</i> register is written from the contents of the <i>EntryHi</i> , <i>EntryLo_0</i> , <i>EntryLo_1</i> , and <i>PageMask</i> registers.