处理器体系结构

第二章 指令集体系结构C

—硬件对过程的支持

(Micro-processor Architecture)

硬件对过程的支持

“面向过程”：PO　(Procedure Oriented)　C语言

“面向对象”：OO　(Object Oriented)　C++、Java、C#...

C语言编写的程序就是PO，C++编写的程序就是OO ？

拿着青龙偃月刀的你能否打得过拿着木棍的关羽？-- 语言只是工具

面向过程：“步骤分解”->先怎么做、后怎么做->一个一个的步骤函数

五子棋程序：1、开始游戏；2、黑子先走；3、绘制画面；4、判断输赢；、5、轮到白子；6、绘制画面；7、判断输赢；8、返回步骤2；9、输出最后结果

简单、清晰　　不易维护

面向对象：“功能分解”->大功能->小功能->一个一个的功能函数

A、玩家对象接受输入；B、棋盘对象绘制画面；C、规则对象判别输赢；

硬件对过程的支持

过程：根据提供的参数执行一定任务的子程序　　—实现抽象的一种方法

使用过程的好处：使程序结构化、易懂、易重复使用

1)将参数放在过程可以访问的位置：参数传递

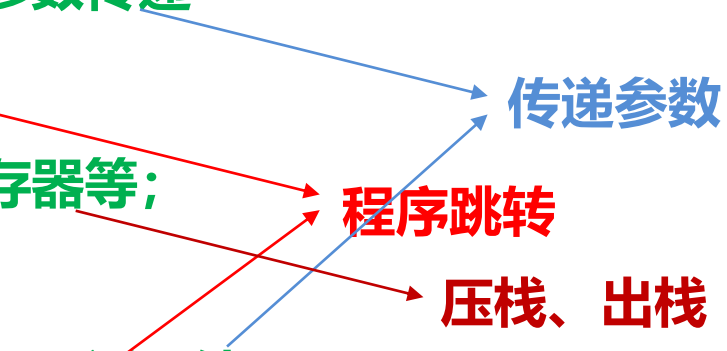
2)将控制转交给过程：调用子程序；

3)获得过程所需的存储资源：保护寄存器等；

4)执行过程任务：执行子程序体；

5)将结果放在调用程序可以访问的位置：返回结果；

6)将控制返回初始点：返回主程序。



Caller：过程调用者

Callee：过程被调用者

内容概述

硬件对过程的支持

1. 指令和寄存器
2. 叶过程(leaf procedures)
3. 嵌套过程(non-leaf procedures)
4. 二进制接口规范

硬件对过程的支持--1. 指令和寄存器

jal：jump and link，将PC+4的值存入$ra，并跳转到指定地址 (子程序入口)

jr：jump register，复制$ra到PC，也可用于其它跳转 (比如：转移表)

寄存器地址	寄存器名称	名称含义	用途
$0	$zero	Zero	常量0
$1	$at	Assembler temporary	留给汇编器作临时变量
$2-$3	$v0-$v1	Values	子函数调用返回值
$4-$7	$a0-$a3	Arguments	子函数调用参数
$8-$15	$t0-$t7	Temporaries	存放临时变量(随便用的)
$16-$23	$s0-$s7	Saved values	保存变量(子函数调用前后)
$24-$25	$t8-$t9	Temporaries	存放临时变量(随便用的)
$26-$27	$k0-$k1	Kernel	中断、异常处理保存的参数
$28	$gp	Global pointer	全局指针
$29	$sp	Stack pointer	堆栈指针
$30	$fp	Frame pointer	帧指针
$31	$ra	Return address	子函数返回地址

子函数调用/返回的参数

超过寄存器数怎么办?

->栈(stack)

压栈(push)、出栈(pop)

硬件对过程的支持--1. 指令和寄存器

jal：jump and link，将PC+4的值存入$ra，并跳转到指定地址 (子程序入口)　(J型指令)

jr：jump register，复制$ra到PC，也可用于其它跳转 (比如：转移表)　(R型指令)

指令格式：

jal　address

R型：	op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
I型：	op(6)	rs(5)	rt(5)	constant or address(16)		
J型：	op(6)	address(26)				

jr　$rs

000000	rs	00000	00000	00000	001000

->需要跳转到指定的32位立即数地址处怎么办?

硬件对过程的支持--*32位立即数

lui：load upper immediate

指令格式：lui rt, immediate #将16位的immediate存入rt高位，低位补0

lui	001111	00000	rt	immediate	lui $1,100	$1=100*65536	rt <- immediate*65536；将16位立即数放到目标寄存器高16位，目标寄存器的低16位填0

例：将 0000 0000 0011 1101　　0000 1001 0000 0000存入$s0

lui $s0, 61　　0000 0000 0011 1101 0000 0000 0000 0000

ori $s0, $s0, 2304　　0000 0000 0011 1101 0000 1001 0000 0000

硬件对过程的支持--2.叶过程(leaf procedures)

叶（子）过程：leaf procedure，不调用其他过程的过程

C code:

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

f存于$s0

参数g, …, j存于$a0, …, $a3

结果保存在$v0

硬件对过程的支持--2.叶过程(leaf procedures)

叶（子）过程：leaf procedure，不调用其他过程的过程

C code:

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

f存于$s0

参数g, …, j存于$a0, …, $a3

结果保存在$v0

子函数内如果使用$s0~s7，则必须对其进行保存/恢复

MIPS code:

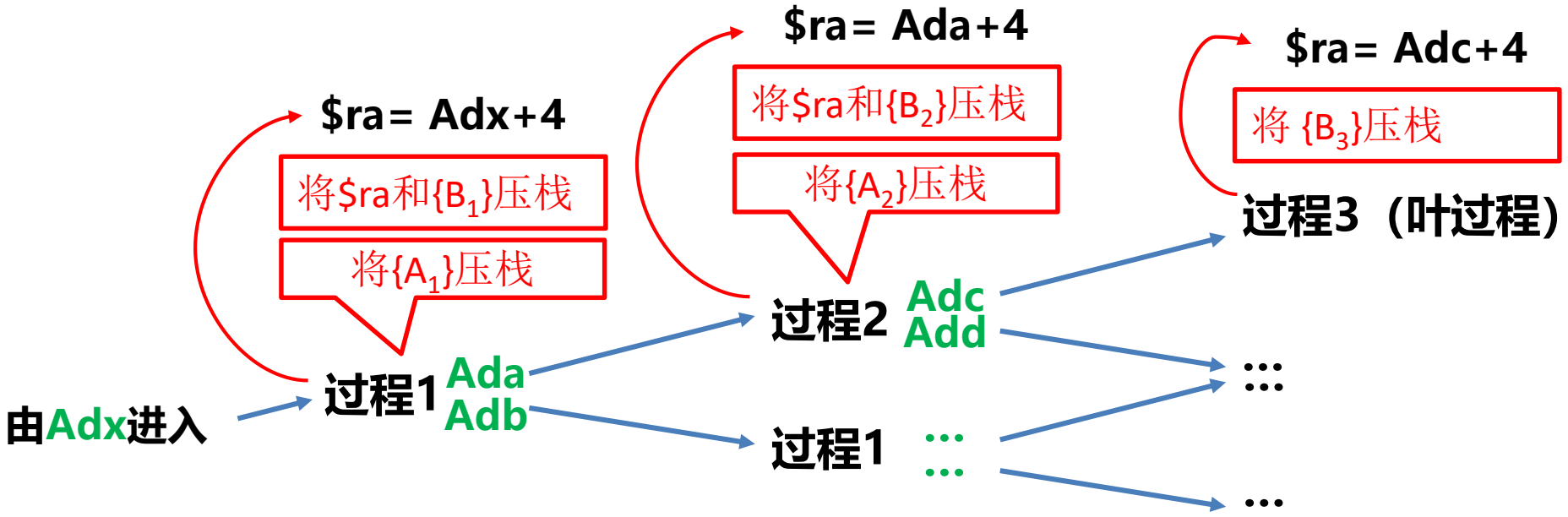
```
leaf_example:  addi $sp, $sp, -4   #$s0入栈
               sw   $s0, 0($sp)
               add  $t0, $a0, $a1
               add  $t1, $a2, $a3
               sub  $s0, $t0, $t1
               add  $v0, $s0, $zero #保存结果
               lw   $s0, 0($sp)     #$s0出栈
               addi $sp, $sp, 4
               jr   $ra             #返回
```

硬件对过程的支持--3.嵌套过程(non-leaf procedures)

过程中调用新的过程

Caller需要保存调用后仍需使用(且callee需要写入)的$a和$t系列寄存器 $\{A_i\}$

Callee需要保存返回地址寄存器(叶过程除外)$ra和需要使用的$s系列寄存器$\{B_i\}$



别忘了出栈^-^：过程执行结束后，将先前压栈的数据恢复

硬件对过程的支持--3.嵌套过程(non-leaf precedures)

例：

C 代码：

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

n保存于$a0

结果存于$v0

MIPS 代码：

```
fact:  addi $sp, $sp, -8      # 空出栈位置
       sw   $ra, 4($sp)       # 返回地址保存入栈
       sw   $a0, 0($sp)       # 参数n保存入栈
       slti $t0, $a0, 1       # 检验 n < 1
       beq  $t0, $zero, L1
       addi $v0, $zero, 1     # n<1成立，返回结果1
       addi $sp, $sp, 8       # 恢复栈位置
       jr   $ra               #  跳转至返回地址
L1:    addi $a0, $a0, -1      # n=n-1
       jal  fact              # 调用fact，并将PC+4存入$ra
       lw   $a0, 0($sp)       # 参数n出栈
       lw   $ra, 4($sp)       #  返回地址出栈
       addi $sp, $sp, 8       # 恢复栈位置
       mul  $v0, $a0, $v0     #返回结果n*fact(n-1)
       jr   $ra               # 跳转到返回地址
```

当n<1时，为叶过程！

重点：fact过程即是caller也是callee

->需要保存调用后仍需使用的$a0、返回地址$ra；若使用了$s系列寄存器，则同样需要保存

硬件对过程的支持--4. 二进制接口规范

ABI：二进制接口（Application Binary Interface，ABI)

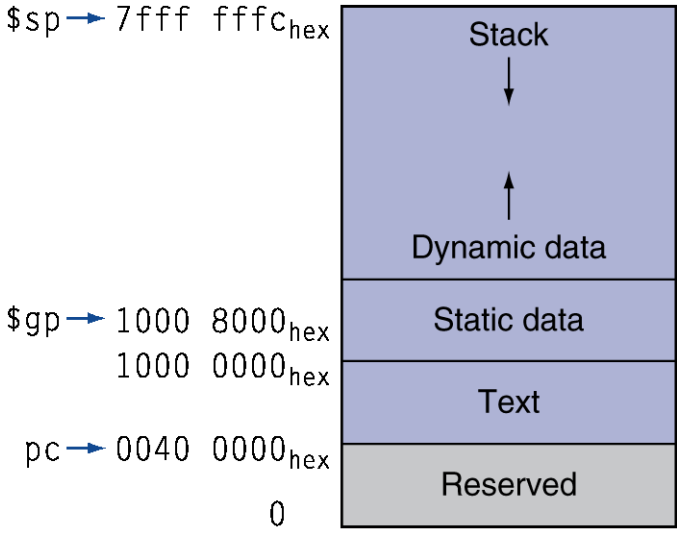
MIPS的ABI：o32、n32、n64等

包含各种规范(convention)：内存空间布局、寄存器的使用、栈帧构建等

1. 用于约束软件，而不是硬件；
2. 编译器或程序员需要遵守，因为违反这些规范会导致诡异错误；

硬件对过程的支持--4.1内存空间布局

MIPS分配内存的典型约定：(一种软件规范(convention))



栈数据

此消彼长，高效利用

动态数据(堆)

静态数据(常量、数列等)

代码段：MIPS机器代码

$gp+偏移寻址范围：

1000 0000~1000 ffff

栈：箱子装“书”，先进后出（函数参数、变量）

堆：图书馆的“书架”，随意存取（动态数据）

硬件对过程的支持--4.2寄存器的使用(补充)

先前，我们已经赋予每个寄存器确定的含义：

寄存器地址	寄存器名称	名称含义	用途
$0	$zero	Zero	常量0
$1	$at	Assembler temporary	留给汇编器作临时变量
$2-$3	$v0-$v1	Values	子函数调用返回值
$4-$7	$a0-$a3	Arguments	子函数调用参数
$8-$15	$t0-$t7	Temporaries	存放临时变量(随便用的)
$16-$23	$s0-$s7	Saved values	保存变量(子函数调用前后)
$24-$25	$t8-$t9	Temporaries	存放临时变量(随便用的)
$26-$27	$k0-$k1	Kernel	中断、异常处理保存的参数
$28	$gp	Global pointer	全局指针
$29	$sp	Stack pointer	堆栈指针
$30	$fp	Frame pointer	帧指针
$31	$ra	Return address	子函数返回地址

但这不是一定的！

事实上，ABI规定了大部分寄存器的明确用途，另一些则可以有不同的用途(由编译器决定)

硬件对过程的支持--4.2寄存器的使用(补充)

MIPS o32中：

Table 1-1 General (Integer) Registers (–32)

Register Name	Software Name (from regdef.h)	Use and Linkage
$0		Always has the value 0.
$1 or **$at**		Reserved for the assembler.
$2..$3	v0-v1	Used for expression evaluations and to hold the integer type function results. Also used to pass the static link when calling nested procedures.
$4..$7	a0-a3	Pass the first 4 words of actual integer type arguments; their values are not preserved across procedure calls.
$8..$11 $11..$15	t0-t7 t4-t7 or ta0-ta3	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$16..$23	s0-s7	Saved registers. Their values must be preserved across procedure calls.
$25	t9 or jp	PIC jump register.
$26..27 or $kt0..$kt1	k0-k1	Reserved for the operating system kernel.
$28 or $gp	gp	Contains the global pointer.
$29 or $sp	sp	Contains the stack pointer.
$30 or $fp	fp or s8	Contains the frame pointer (if needed); otherwise a saved register (like **s0-s7**).
$31	ra	Contains the return address and is used for expression evaluation.

以$30为例：

GCC的MIPS C编译器：用作帧指针$fp

MIPS的C编译器：用作$s8

硬件对过程的支持--*关于“$fp”

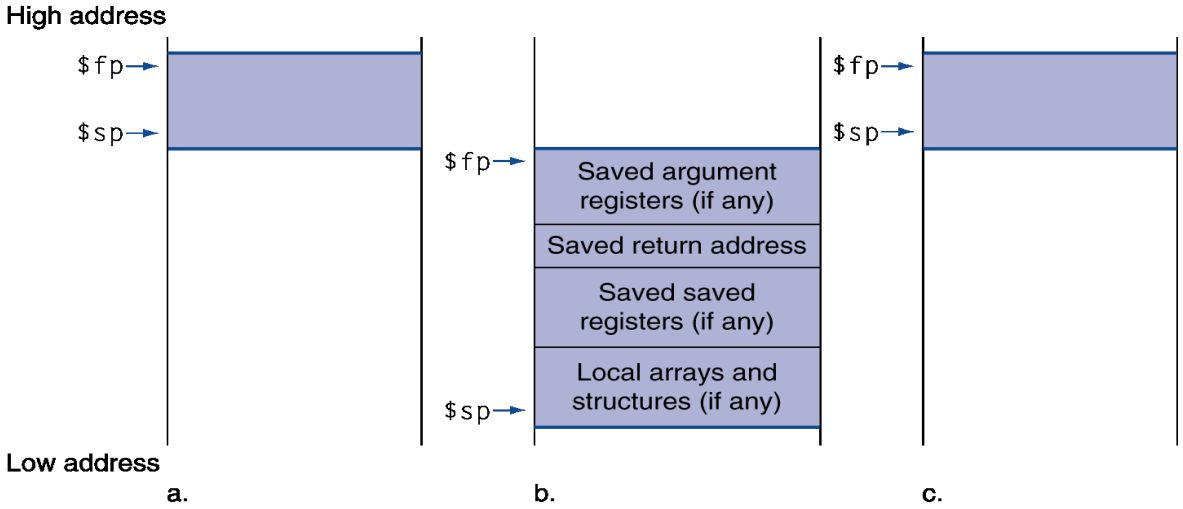
将$30当作$s8：不使用$fp，可以节省一些指令（每次调用函数不需要给$fp赋值，也不需要保存$fp）

小茗同学在写一本书，且每次写新内容的时候都要对上次写的内容进行修订，但每次都不记得上次写到哪里，必须一页一页地检查，好麻烦。。。

(如果有个书签就好了)

问题：程序运行时，栈指针是不断变化的！

找数据的时候，可以以栈指针为基址，但会降低程序可读性，增加汇编难度，并给调试带来困难



->$fp分配好后不会变，更便于数据访问和恢复栈指针

硬件对过程的支持--4.3栈帧的构建

过程调用帧：一个内存块，用于参数传递、保留callee可能改变而caller不希望改变的参数、为过程提供本地变量空间；
在多数编程语言中，过程调用和返回遵循严格的LIFO。因此过程调用帧也叫作**栈帧**

栈：相对整个系统而言
栈帧：相对某个函数而言，每个栈帧对应一个函数。栈帧反映了函数调用关系，每次调用一个函数，都要为这个函数实例分配栈空间，单个函数分配的空间就是栈帧

构建栈帧的重要性：
1.使调用者和被调用者达成某种约定（参数传递方式、函数返回方式、寄存器如何共享等）
2.定义了被调用者如何使用自己的栈帧来完成局部变量的存储和使用

硬件对过程的支持--4.3栈帧的构建

栈帧(stack frame)的构建有多种方式， 以MIPS o32的栈帧(stack frame)结构为例：

A simple non-leaf function that takes two arguments:

Memory Addresses ↑ Old sp →

Stack Grows ↓

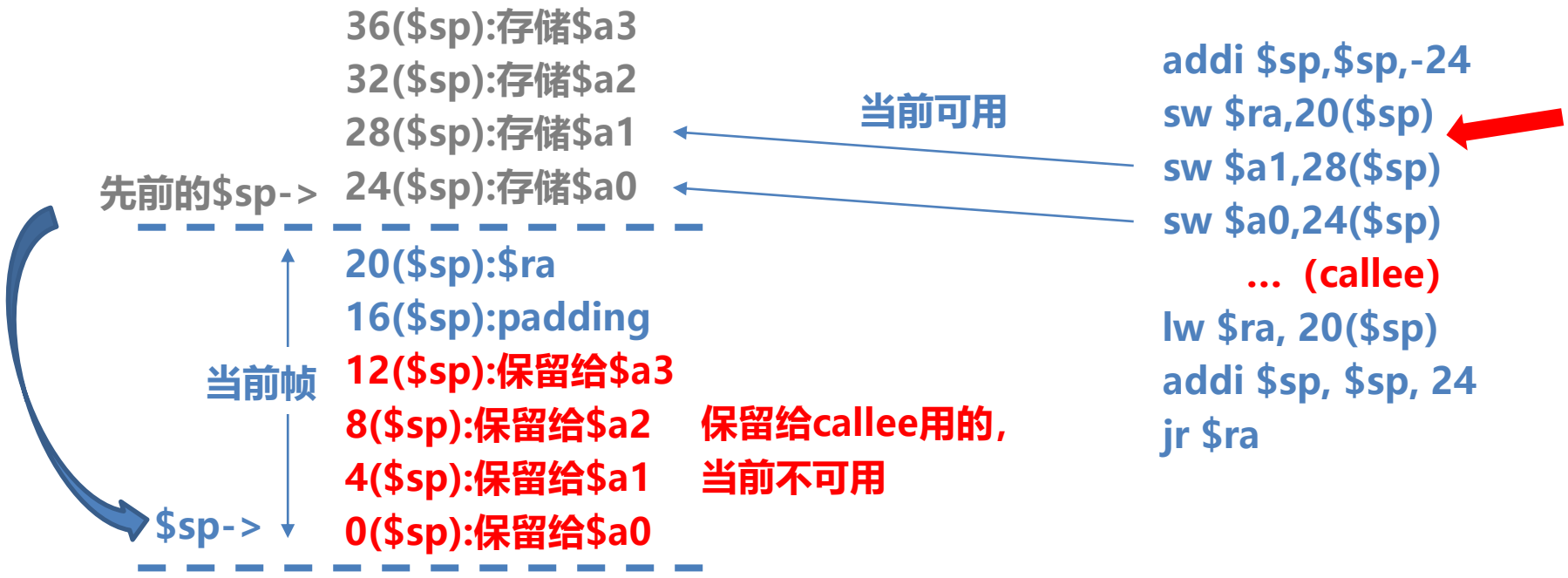
Slot for incoming a3	N/A	16 bytes	36(sp)
Slot for incoming a2	N/A		32(sp)
Slot for incoming a1	a1		28(sp)
Slot for incoming a0	a0		24(sp)
Register Save Area	ra ($31)	8 bytes	20(sp)
	Padding		16(sp)
Slot for outgoing a3	Reserved	16 bytes	Do not use
Slot for outgoing a2			
Slot for outgoing a1			
Slot for outgoing a0			

New sp →

This is intended to teach the MIPS o32 Calling Convention by example but should not be considered a complete substitute for reading the official MIPS ABI.

1. Caller在栈帧的底部保留四个字，这是给callee存储$a0~$a3用的。即使callee不使用，也保留。意味着：如果是嵌套过程，则不可使用0($sp),4($sp),8($sp),12($sp)(假如帧大小是32字节，则callee可以用32($sp),36($sp),40($sp),44($sp)来保存a0,a1,a2,a3)
2. 每个section都必须是双字对齐。这仅仅是为了能够在栈帧中存入双字数据。比如：如果saved regeisters仅有一个数据，那么就额外空出一个位置(padding)
3. 鉴于上述原因，嵌套过程的最小栈帧为24字节：16个字节用于存储$a0~$a3，8个字节用于saved registers(必须有$ra，而saved registers需要双字对齐)
4. 叶过程的最小栈帧为0字节

硬件对过程的支持--4.3栈帧的构建

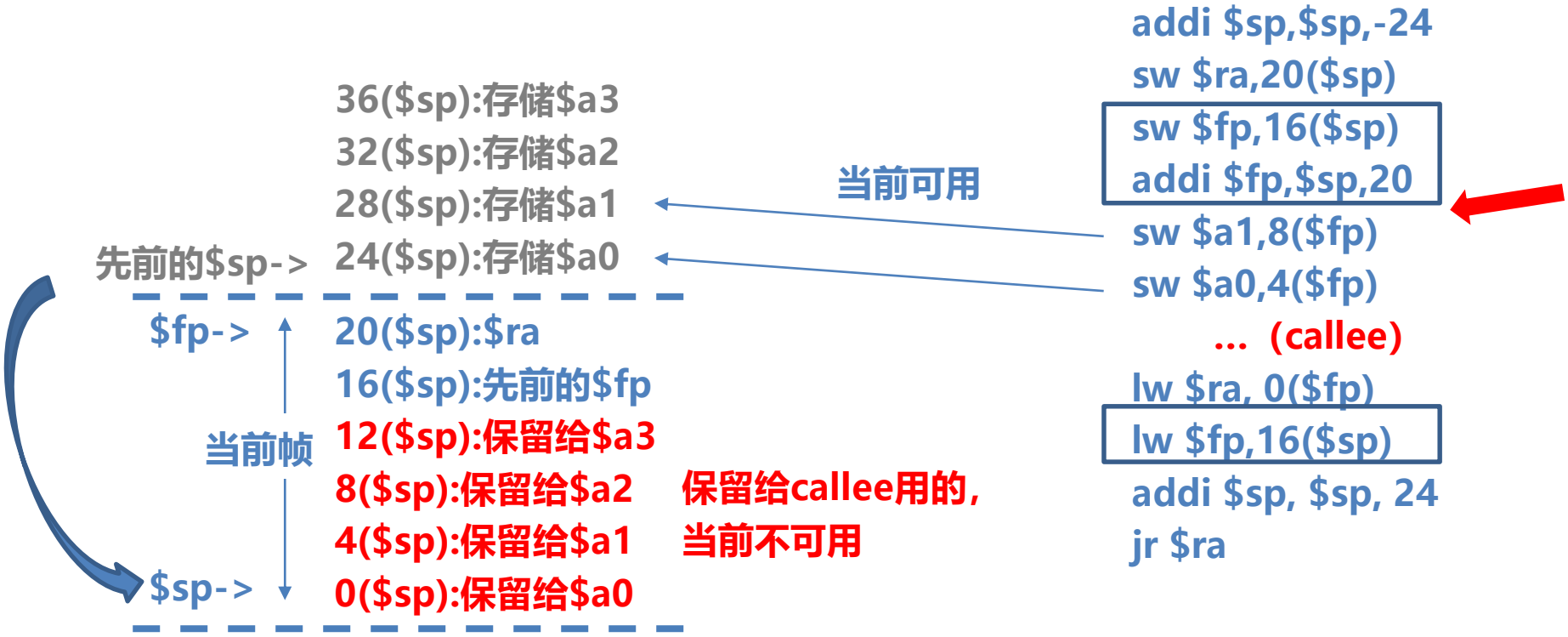


未使用$fp

好处：指令数量少

问题：$sp在在分配好后可能会继续改变，从而给栈帧数据的访问带来困难

硬件对过程的支持--4.3栈帧的构建



小结

硬件对过程的支持

1. 指令和寄存器
2. 叶过程(leaf procedures)
3. 嵌套过程(non-leaf procedures)
4. 接口规范

已掌握的汇编语言

算术运算

逻辑运算

程序跳转

存取数

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

已用到的寄存器

寄存器地址	寄存器名称	名称含义	用途
$0	$zero	Zero	常量0
$1	$at	Assembler temporary	留给汇编器作临时变量
$2-$3	$v0-$v1	Values	子函数调用返回值
$4-$7	$a0-$a3	Arguments	子函数调用参数
$8-$15	$t0-$t7	Temporaries	存放临时变量(随便用的)
$16-$23	$s0-$s7	Saved values	保存变量(子函数调用前后)
$24-$25	$t8-$t9	Temporaries	存放临时变量(随便用的)
$26-$27	$k0-$k1	Kernel	中断、异常处理保存的参数
$28	$gp	Global pointer	全局指针
$29	$sp	Stack pointer	堆栈指针
$30	$fp	Frame pointer	帧指针
$31	$ra	Return address	子函数返回地址

思考

作为嵌套过程，被调用的子函数为什么要保存$ra

课后作业

下列MIPS代码中，A、B和C位置分别要将那些寄存器压栈?

```
Fun1:  A
       jal Fun2
       addi $t0, $t1, $zero
       (出栈)
       jr $ra
Fun2:  B
       sub $s0, $t0, $s1
       jal Fun3
       (出栈)
       jr $ra
Fun3:  C
       addi $t1, $a0, $zero
       (出栈)
       jr $ra
```