Chap.2 Instructions：Language of the Computer

Chap.2 指令系统：计算机的语言

- 2.1 Introduction
- 2.2 Operations of the Computer Hardware
- 2.3 Operands of the Computer Hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People

Chap.2： Instructions: Language of the Computer

Chap.2 指令系统：计算机的语言

- 2.10 MIPS Addressing for 32-Bit Immediates and More Complex Addresses
- 2.11 Parallelism and instructions: Synchronization
- 2.12 Translating and Starting a Program
- 2.13 A C Sort Example to Put it All Together
- 2.14 Arrays versus Pointers
- 2.15 Advanced Material: Compiling C and Interpreting Java
- 2.16 Real Stuff：ARM Instructions
- 2.17 Real Stuff：x86 Instructions

2.1 Introduction

设计目标

- computer designers have a common goal：

  To find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and power.

- Instruction / Instruction Set：指令 / 指令集

  the words of a computer's language are called instructions, and its vocabulary is called an instruction set.

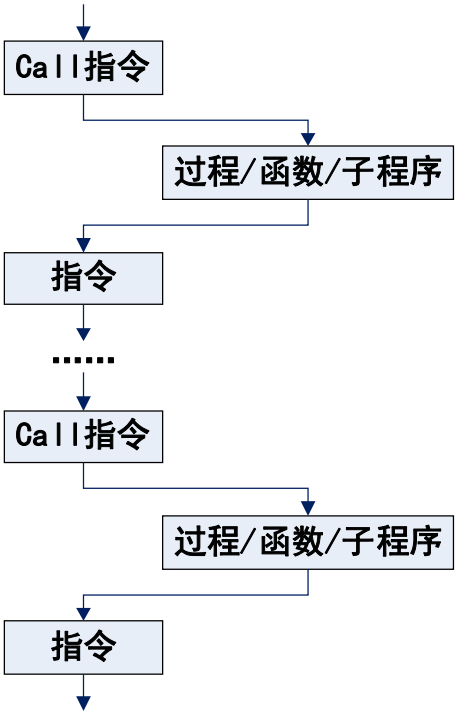
2.8 Supporting Procedures in Computer Hardware

关于过程调用的实现机制

procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.

caller The program that instigates a procedure and provides the necessary parameter values.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

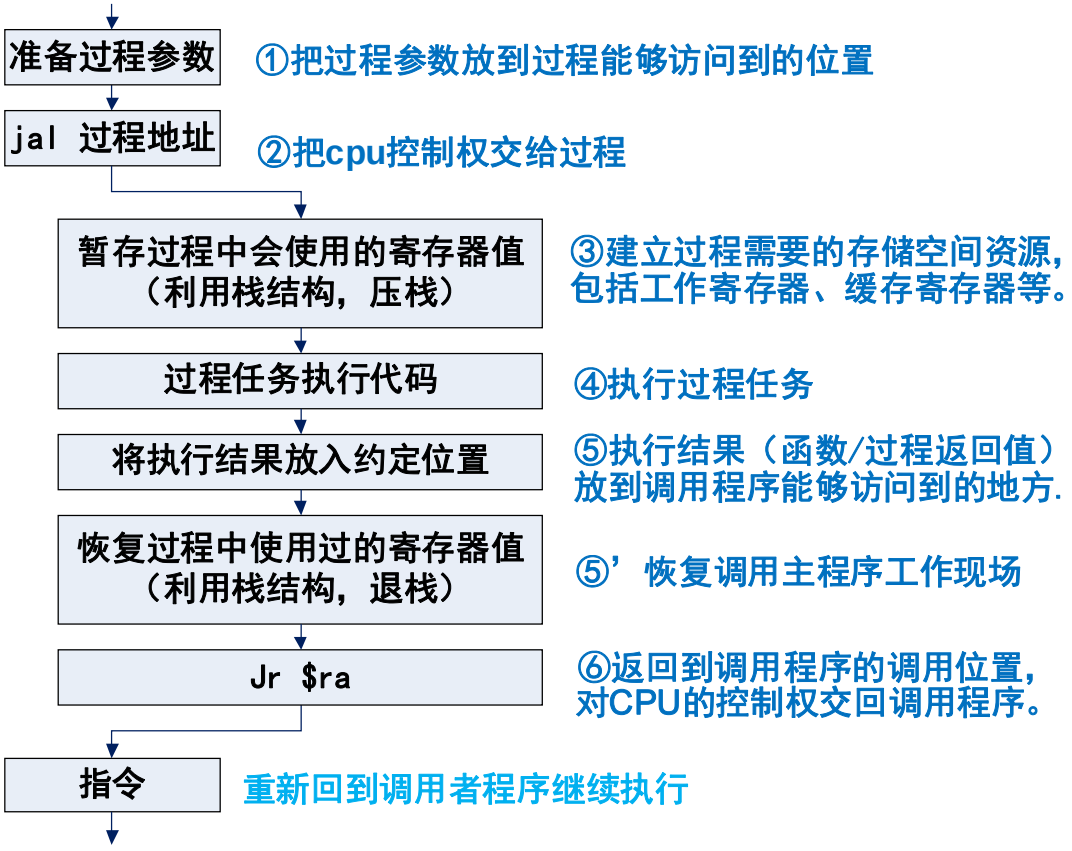


2.8 Supporting Procedures in Computer Hardware

过程/函数/子程序的实现步骤

1. Put parameters in a place where the procedure can access them.
   ①传参：把过程参数放到过程能够访问到的地方。
2. Transfer control to the procedure.
   ②把CPU控制权交给过程。
3. Acquire the storage resources needed for the procedure.
   ③建立过程需要的存储空间资源。
4. Perform the desired task.
   ④执行过程任务。
5. Put the result value in a place where the calling program can access it.
   ⑤执行结果放到调用程序能够访问到的地方。
6. Return control to the point of origin, since a procedure can be called from several points in a program.
   ⑥返回到调用程序的调用位置，对CPU的控制权交回调用程序。

过程/函数/子程序的机器语言级实现过程



MIPS对过程调用实现的设计支持

关于参数传递和对32个寄存器的使用约定

- MIPS软件系统在实现过程调用时，对32个寄存器的使用，遵循以下约定（非绝对规定）：
  - $a0 - $a3：采用这4个变量寄存器用于传递给过程的参数。
    four argument registers in which to pass parameters
  - $v0 - $v1：采用这2个寄存器用于存储和传递过程的返回值。
    two value registers in which to return values
  - $ra：采用该寄存器保存过程执行完毕后的返回地址。
    one return address register to return to the point of origin

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- $a0–$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

MIPS对过程调用实现的设计支持

jal 指令和 jr 指令

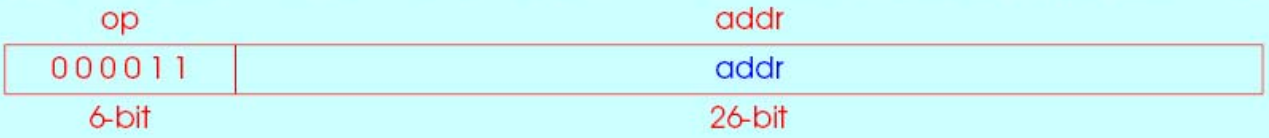
- jal 指令： jump-and-link instruction
  功　能： 保存返回地址（即紧接的下一条指令的地址）；跳转。
- jr 指令： jump register instruction
  功　能： 跳转到寄存器所指定的地址。

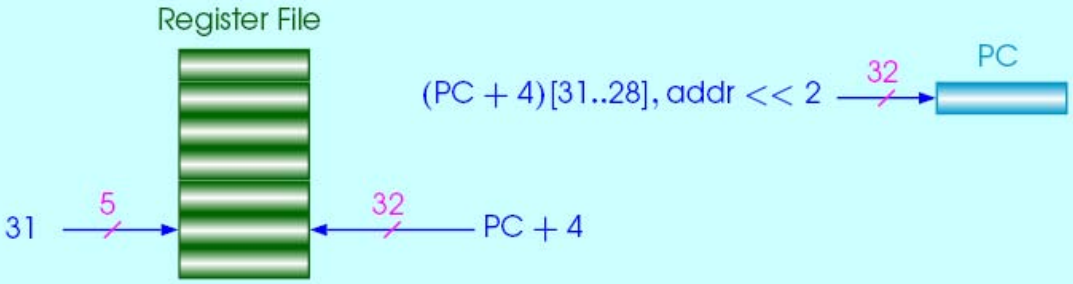
- 其它保存和恢复工作空间的工作步骤，据情由其它指令组合实现。

J	Unconditional Jump	PC = PC[31:28] || offset<<2
JAL	Jump and Link	GPR[31] = PC + 8 PC = PC[31:28] || offset<<2
JALR	Jump and Link Register	Rd = PC + 8 PC = Rs
JR	Jump Register	PC = Rs

jal 指令 (Jump And Link)

```
jal addr; $31<--PC+4;PC<--(PC+4)[31..28],addr<<2
```

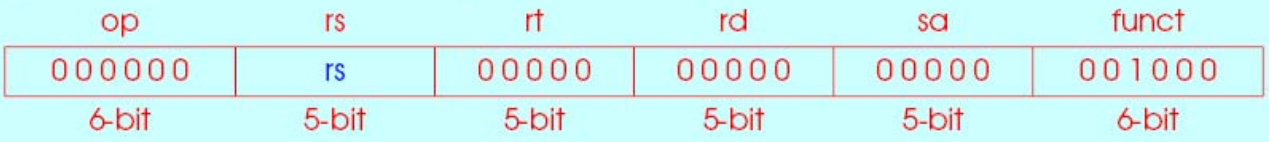


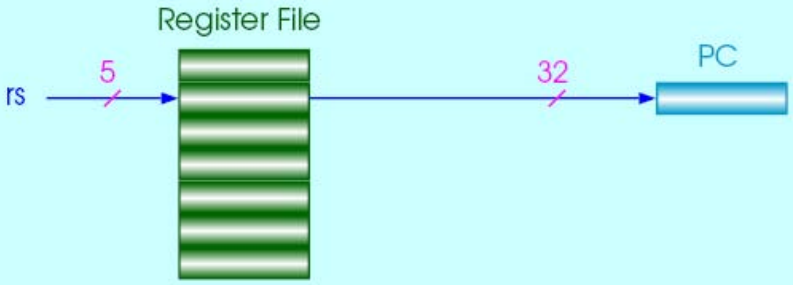


指令的意義: 調用(跳轉并保存返回地址)。目標地址是: 把26位的立即數addr左移兩位, 再與PC+4的高4位拼接。返回地址保存到寄存器31中。返回地址是PC+8, 我們暫時假定是PC+4

jr 指令 (Jump Register)

```
jr    rs              ; PC <-- rs
```





指令的意義: 把寄存器rs中的數據寫入PC中

MIPS对过程调用实现的支持

jal 指令 (Jump And Link)

In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register `$ra`. The **jump-and-link instruction** (`jal`) is simply written

```
jal ProcedureAddress
```

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This "link," stored in register `$ra` (register 31), is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

关于过程调用

部分名词

jump-and-link instruction An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (`$ra` in MIPS).

return address A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register `$ra`.

caller The program that instigates a procedure and provides the necessary parameter values.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

program counter (PC) The register containing the address of the instruction in the program being executed.

MIPS对过程调用实现的支持

jr 指令 (Jump Register)

To support such situations, computers like MIPS use *jump register* instruction (`jr`), introduced above to help with case statements, meaning an unconditional jump to the address specified in a register:

```
jr   $ra
```

Jump register instruction jumps to the address stored in register `$ra`—which is just what we want. Thus, the calling program, or **caller**, puts the parameter values in `$a0`–`$a3` and uses `jal X` to jump to procedure `X` (sometimes named the **callee**). The callee then performs the calculations, places the results in `$v0` and `$v1`, and returns control to the caller using `jr $ra`.

Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the **program counter**, abbreviated *PC* in the MIPS architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction actually saves PC + 4 in register `$ra` to link to the following instruction to set up the procedure return.

2.8 Supporting Procedures in Computer Hardware

Using More Registers

- 过程（函数/子程序）可能需要用到更多的寄存器，即多于4个传递过程参数的寄存器，或多于2个用于传递返回参数的寄存器。
- 在32个寄存器中，除部分几个用作特殊用途外，其它的通用寄存器可以被过程使用。
- 由于过程在执行完毕返回后，除返回预期的正常结果外，不能影响原调用代码的工作环境（否则...），因此，需要有工作环境的保存和恢复机制。
- 利用栈结构组织的内存，是一种可行、并且资源丰富的暂存方法。
- 什么是栈（stack）？

stack A data structure for spilling registers organized as a last-in-first-out queue.

栈（stack）

- 栈（stack）是一种先进后出的数据结构。数据就像弹夹中的子弹。
- 栈有两种操作：压栈（Push）和出栈（Pop）。
  - □Push：将数据放入栈顶；栈（顶）指针上（往小端地址）移。
  - □Pop：将数据从栈顶移出；栈（顶）指针下（往大端地址）移。

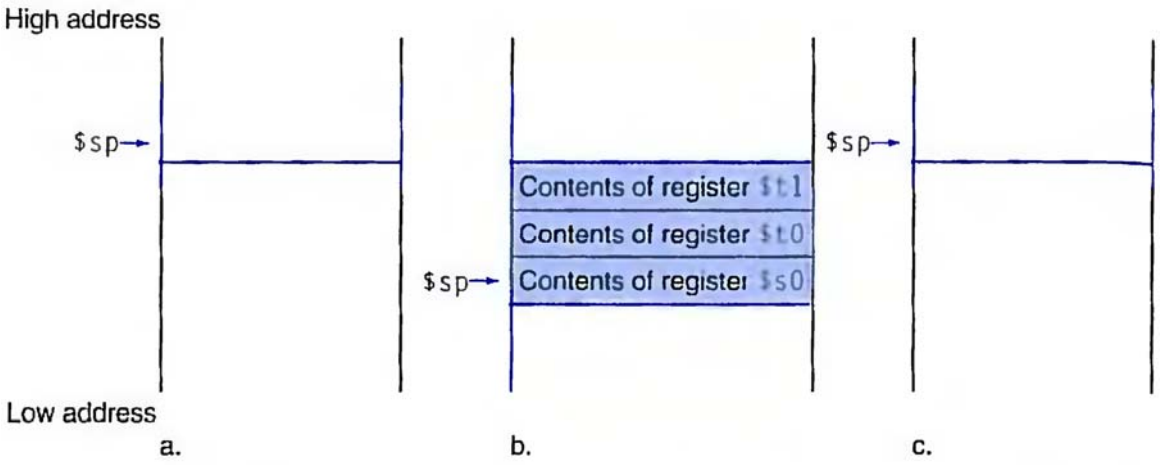


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

栈（stack）

The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found. The **stack pointer** is adjusted by one word for each register that is saved or restored. MIPS software reserves register 29 for the stack pointer, giving it the obvious name $sp. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a **push**, and removing data from the stack is called a **pop**.

By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

stack A data structure for spilling registers organized as a last-in-first-out queue.

stack pointer A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In MIPS, it is register $sp.

push Add element to stack.

pop Remove element from stack.

2.8 Supporting Procedures in Computer Hardware

【例】Compiling a C Procedure That Doesn't Call Another Procedure

EXAMPLE

■ C语言描述的过程/函数：

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

■ 假定g、h、i、j这4个变量已放入$a0、$a1、$a2、$a3这4个自变量寄存器，运算结果f放入$s0。

■ What is the compiled MIPS assembly code?

2.8 Supporting Procedures in Computer Hardware

【例】Compiling a C Procedure That Doesn't Call Another Procedure

■ C语言描述的过程/函数：

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

■ 假定g、h、i、j这4个变量已放入$a0、$a1、$a2、$a3这4个自变量寄存器，运算结果f放入$s0。

■ What is the compiled MIPS assembly code?

【例】Compiling a C Procedure That Doesn't Call Another Procedure

编译步骤

ANSWER

■ leaf_example：　　#过程代码首地址

① 将过程中要用到的寄存器压栈，即先暂存，以备后续恢复工作场景。

```
addi $sp, $sp, -12   # adjust stack to make room for 3 items
sw   $t1, 8($sp)     # save register $t1 for use afterwards
sw   $t0, 4($sp)     # save register $t0 for use afterwards
sw   $s0, 0($sp)     # save register $s0 for use afterwards
```

② 执行过程计算操作

```
add  $t0, $a0, $a1   # register $t0 contains g + h
add  $t1, $a2, $a3   # register $t1 contains i + j
sub  $s0, $t0, $t1   # f = $t0 - $t1, which is (g + h)-(i + j)
```

③ 将返回值放入$v0

```
add  $v0, $s0, $zero  # returns f ( $v0 = $s0 + 0)
```


【例】Compiling a C Procedure That Doesn't Call Another Procedure

编译步骤

ANSWER

④ 退出过程/函数/子程序前，从栈中恢复原工作场景。

```
lw   $s0, 0($sp)   # restore register $s0 for caller
lw   $t0, 4($sp)   # restore register $t0 for caller
lw   $t1, 8($sp)   # restore register $t1 for caller
addi $sp,$sp,12    # adjust stack to delete 3 items
```

⑤ 最后执行跳转返回到原调用位置

```
jr   $ra           # jump back to calling routine
```

⑥ 将返回值放入$v0

```
add  $v0, $s0, $zero  # returns f ( $v0 = $s0 + 0)
```

2.8 Supporting Procedures in Computer Hardware

MIPS关于过程调用中对寄存器的利用优化

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS software separates 18 of the registers into two groups:

- `$t0–$t9`: ten temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `$s0–$s7`: eight saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

关于嵌套过程（Nested Procedures）

【例】 Compiling a Recursive C Procedure, Showing Nested Procedure Linking

- C语言描述的计算阶乘的递归过程/函数：

EXAMPLE

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

- 假定变量 n 已放入$a0，运算结果 放入$s0。

- What is the compiled MIPS assembly code?

【例】Compiling a Recursive C Procedure, Showing Nested Procedure Linking

编译步骤

ANSWER

■ fact：　　# 过程代码首地址

① 将过程中要用到的寄存器压栈，即先暂存，以备后续恢复工作场景。

```
addi  $sp, $sp, -8    # adjust stack for 2 items
sw    $ra,  4($sp)    # save the return address
sw    $a0,  0($sp)    # save the argument n
```

② 确定递归退出条件

```
slti  $t0,  $a0, 1      # test for n < 1
beq   $t0,  $zero, L1   # if n >= 1, go to L1
```

③ 满足递归退出条件，将返回值放入$v0

```
addi  $v0, $zero, 1     # return 1
addi  $sp, $sp, 8       # pop 2 items off stack （注意 ？）
jr    $ra               # return to caller
```


【例】Compiling a Recursive C Procedure, Showing Nested Procedure Linking

编译步骤

ANSWER

④ 递归问题小规模化的递推：

```
L1:  addi  $a0, $a0, -1    # n >= 1: argument gets (n - 1)
     jal   fact            # call fact with (n - 1)，递归调用
```

⑤ 递归最后一级返回后，执行恢复原工作环境：

```
     lw    $a0, 0($sp)     # return from jal: restore argument n
     lw    $ra, 4($sp)     # restore Lhe return address
     addi  $sp, $sp, 8     # adjust stack pointer to pop 2 items
```

⑥ 将返回值放入$v0：

```
     mul   $v0, $a0, $v0   # return  n * fact (n - 1)
```

⑦ 转移返回至原调用处：

```
     jr    $ra             # return to the caller
```

注意：⑤和⑥两个步骤能换顺序吗？

2.8 Supporting Procedures in Computer Hardware

关于MIPS的寄存器

Name	Register number	usage	Preserved on call? 过程调用时是否自动保存？
$zero	0	The constant value 0	n.a.
$at	1	Assembler Temporary	no
$v0-$v1	2-3	Values for results and expression evaluation	no
$a0-$a3	4-7	Arguments	no
$t0-$t7	8-15	Temporaries	no
$s0-$s7	16-23	Saved temporaries	yes
$t8-$t9	24-25	More temporaries	no
$k0-$k1	26-27	Reserved for OS Kernel	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Acknowledgements

- These slides contain material developed and copyright by:
  - —Arvind (MIT)
  - —Krste Asanovic (MIT/UCB)
  - —John Kubiatowicz (UCB)
  - —David Patterson (UCB)
  - —Geng Wang (SJTU)
  - —Yanmin Zhu (SJTU)
  - —Li Yamin(Hosei Univ.)