第二章　指令系统原理与实例

（教材附录A）

第二章 指令系统原理与实例

- 指令集系统结构（Instruction Set Architecture ISA），即计算机硬件对系统程序员和编译器开发者可见的部分
- **Different computers have different instruction sets**
  - But with many aspects in common
- 从四个方面讨论指令集系统结构：

（1）指令系统分类的方法

（2）指令系统的分析评价方法

（3）编译器与指令系统结构的相互影响

（4）典型的RISC系统结构---MIPS 64位ISA

第二章　指令系统原理与实例

- 2.1 简介
- 2.2 指令集系统结构的分类
- 2.3 存储器寻址
- 2.4 操作数的类型
- 2.5 指令系统的操作
- 2.6 控制流指令
- 2.7 指令系统的编码
- 2.8 编译器的角色
- 2.9 MIPS系统结构
- 2.10 谬误和易犯的错误
- 2.11 结论

2.1 简介

- 第一章提到的应用领域：
- 桌面计算：注重程序的定点和浮点运算性能，不注重程序的大小以及处理器的功耗。
- 服务器及集群：主要应用于数据库、文件服务器、Web应用等。注重定点和字符串方面的性能，有浮点指令但不注重。
- 嵌入式应用和个人移动设备：注重功耗和成本，代码量大小很重要。一些指令（如浮点指令）作为可定制的选项。

- 以上应用领域的指令系统仍然是非常相似的
- 本章的MIPS的ISA在桌面、服务器及嵌入式应用中均有广泛应用

2.2指令集系统结构的分类

- 指令集系统结构（Instruction Set Architecture ISA）的分类

  不同指令集系统结构最根本的区别：

  在于处理器内部数据的存储结构不同。

- 存储结构：栈、累加器或一组寄存器。操作数可以显式指定或者隐含指定。

  （1）栈系统结构中操作数隐含地位于栈顶

  （2）累加器系统结构中的一个隐含操作数就是累加器。

  （3）通用寄存器结构系统中显式地指定操作数，不是寄存器就是存储器地址。

2.2指令集系统结构的分类

- 堆栈结构
- 累加器结构
- 通用寄存器结构

根据操作数的来源不同，又可进一步分为:

- 寄存器-存储器结构（RM结构）

  （操作数可以来自存储器 ）
- 寄存器-寄存器结构（RR结构）

  （所有操作数都是来自通用寄存器组）

  也称为load-store结构，这个名称强调：只有load指令和store指令能够访问存储器

2.2指令集系统结构的分类

❖不同的指令集系统结构的示意图

箭头指示操作数是输入还是ALU运算的结果，或者既是输入也是结果。

橙色表示输入，紫色表示输出，红色表示同时作为输入和输出。

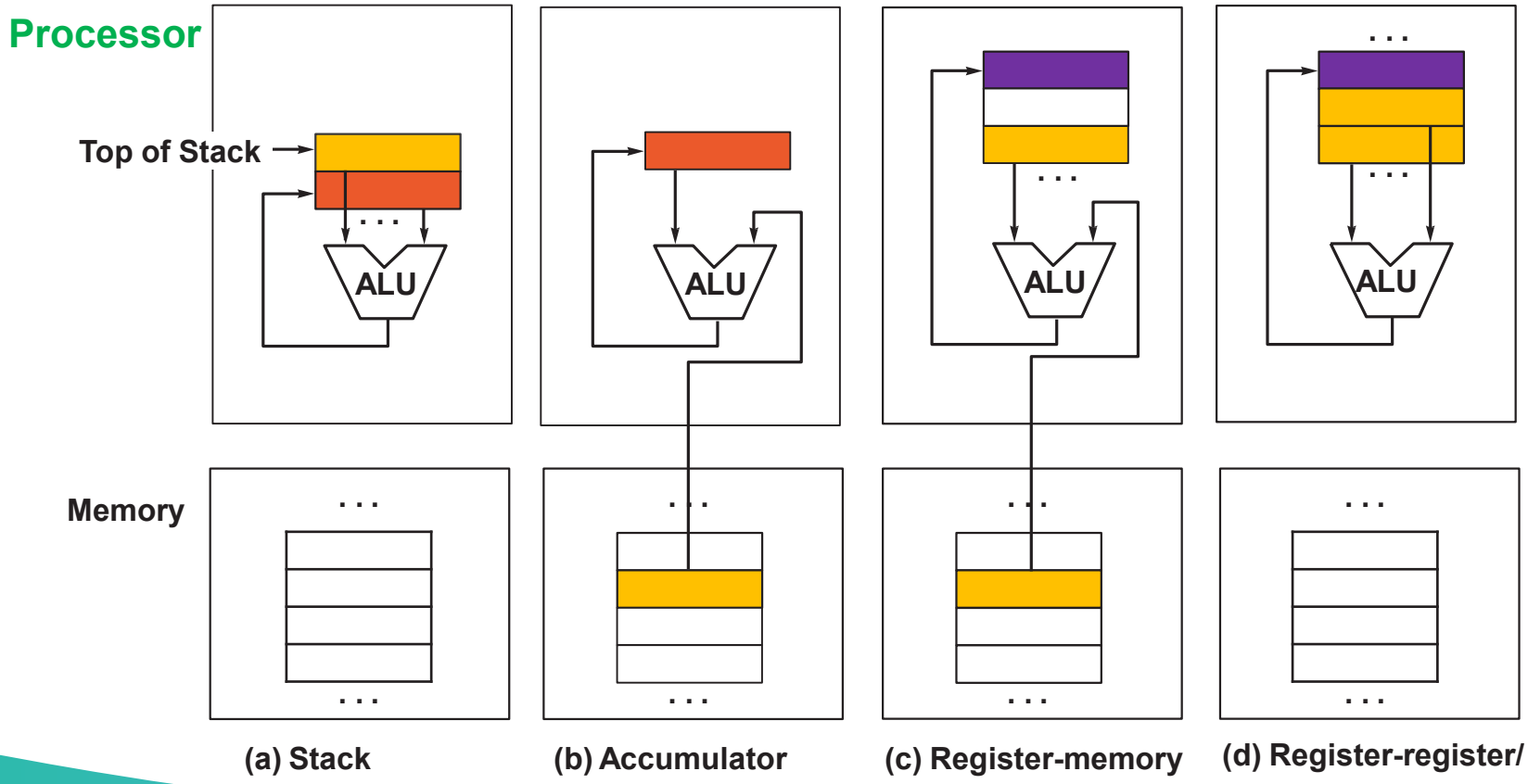


Figure A.1

2.2指令集系统结构的分类

- 不同的系统结构方块图

  (a)中，栈顶寄存器（TOS:Top of Stack）指向栈顶部的输入操作数，次栈顶存放的是第二个操作数。第一个操作数被从栈中移除，运算结果存放在第二个操作数的位置，同时栈顶寄存器指向运算结果。所有的操作数都是隐含的。

- (b)中，累加器既是隐含的输入操作数也运算结果。
- (c)中，其中一个操作数在寄存器中，另一个在存储器中，运算结果存放在寄存器中。
- (d)中，所有的操作数都是寄存器，与栈结构类似，也只能通过一些单独的指令传输到存储器中：（a）中是push或pop，(d)中是load或store。

2.2指令集系统结构的分类 Figure A.2

- 下表说明代码C=A+B在这三类系统结构中分别是如何表示：（设A,B和C都在存储器中且不破坏A和B的值）

栈	累加器	寄存器（Reg-mem）	寄存器（load-store）
Push A	Load A	Load R1，A	Load R1，A
Push B	Add B	Add R3，R1，B	Load R2，B
Add	Store C	Store R3，C	Add R3，R1，R2
Pop C			Store R3，C

说明：在栈结构和累加器结构中加法指令的操作数是隐含的，而在寄存器结构中操作数必须明确指定。

- Add, three operands
  - Two sources and one destination

```
Add R3, R1, R2  // R3 gets R1 + R2
```

C=A+B（设A,B和C都在存储器中且不破坏A和B的值）

栈	累加器	寄存器（Reg-mem）	寄存器（load-store）
Push A	Load A	Load R1，A	Load R1，A
Push B	Add B	Add R3，R1，B	Load R2，B
Add	Store C	Store R3，C	Add R3，R1，R2
Pop C			Store R3，C

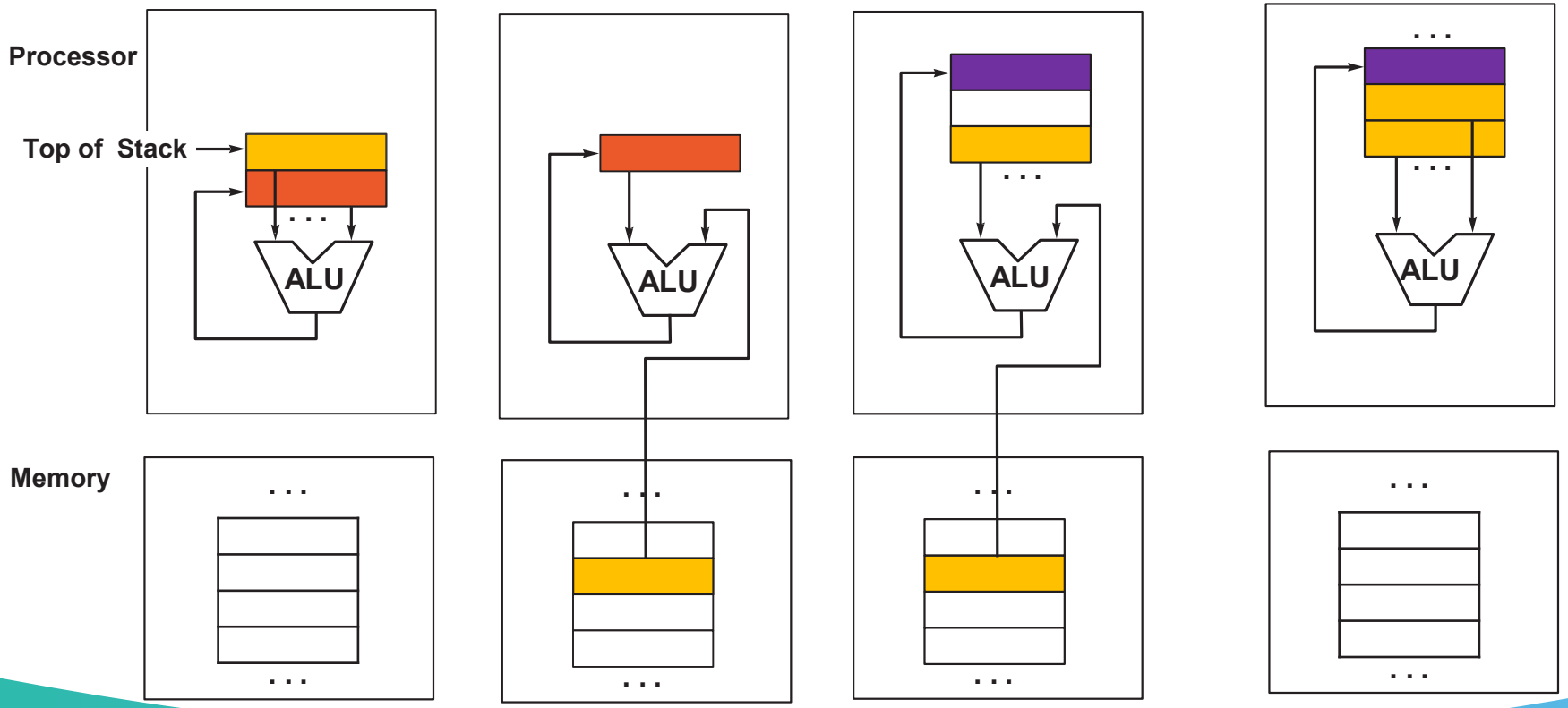


Figure A.2

Exercise A.9.a

Assume that values A, B, C, D, E, and F reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, and register addresses are 6 bits.

For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?

假定整数值A、B、C、D、E和F驻存在存储器中。 另外假定指令操作码以8位表示， 存储器地址为64位，寄存器地址为6位。

对于表A-1中的每个指令集体系结构， 对于计算C=A+B的代码，每条指令中出现多少个地址或名称？总代码大小为多少？

Exercise A.9.a

Assume that values A, B, C, D, E, and F reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, and register addresses are 6 bits.

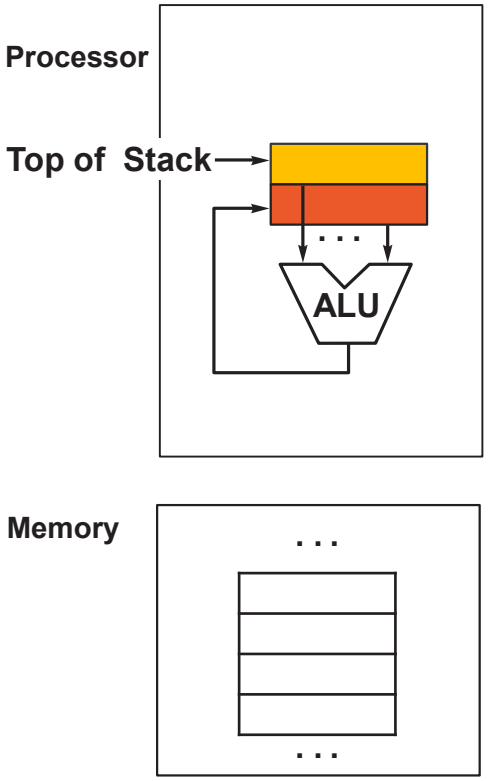
For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?

栈	累加器	寄存器（Reg-mem）	寄存器（load-store）
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R3, R1, B	Load R2, B
Add	Store C	Store R3, C	Add R3, R1, R2
Pop C			Store R3, C

Figure A.2

Exercise A.9.a

- For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?



a.Stack:

Push A **// one address appears in the instruction, code size = 8 bits (opcode) + 64 bits (memory address) = 72 bits;**

Push B **// one address appears in the instruction, code size = 72 bits;**

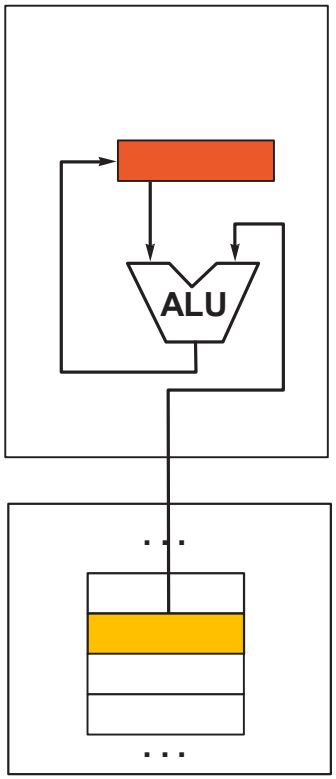
Add **// zero address appears in the instruction, code size = 8 bits;**

Pop C **// one address appears in the instruction, code size = 72 bits;**

Total code size = 72 + 72 + 8 + 72 = 224 bits.

Exercise A.9.a

- For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?



2)Accumulator

Load A **// one address appears in the instruction, code size = 8 bits (opcode) + 64 bits (memory address) = 72 bits;**

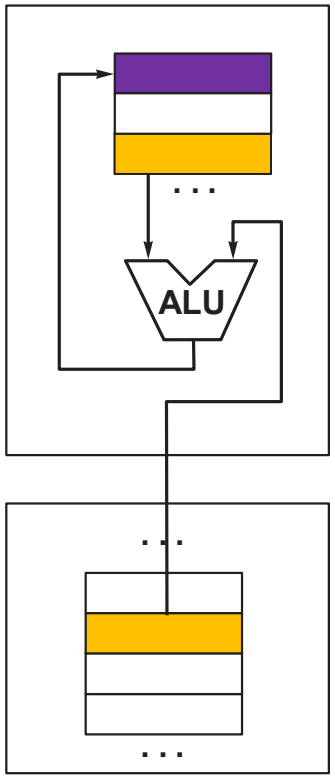
Add B **// one address appears in the instruction, code size = 72 bits;**

Store C **// one address appears in the instruction, code size = 72 bits;**

Total code size = 72 + 72 + 8 + 72 = 216 bits.

Exercise A.9.a

- For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?



3)Register-memory

Load R1, A **// two addresses appear in the instruction, code size = 8 bits (opcode) + 6 bits (register address) + 64 bits (memory address) = 78 bits;**

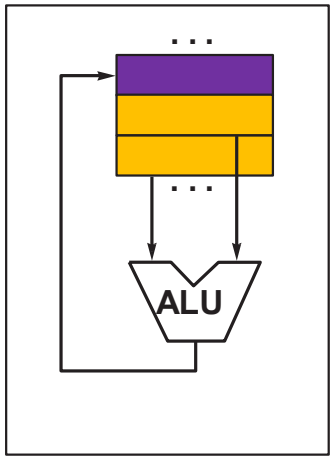
Add R3, R1, B **// three addresses appear in the instruction, code size = 8 bits (opcode) + 6 bits (register address) + 6 bits (register address) + 64 bits (memory address) = 84 bits;**

Store R3,C **// two addresses appear in the instruction, code size = 78 bits;**

Total code size = 78 + 84 + 78 = 240 bits.

Exercise A.9.a

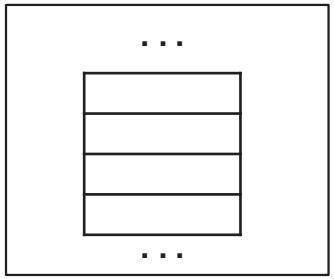
- For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?



4)Register-register

Load R1, A **// two addresses appear in the instruction, code size = 8 bits (opcode) + 6 bits (register address) + 64 bits (memory address) = 78 bits;**

Load R2, B **// two addresses appear in the instruction, code size = 78 bits;**



Add R3, R1, R2 **// three addresses appear in the instruction, code size = 8 bits (opcode) + 6 bits (register address) + 6 bits (register address) + 6 bits (register address) = 26 bits;**

Store R3, C **// two addresses appear in the instruction, code size = 78 bits;**

Total code size = 78 + 78 + 26 + 78 = 260 bits.

Exercise A.9.a

Assume that values A, B, C, D, E, and F reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, and register addresses are 6 bits.

For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?

栈	累加器	寄存器（Reg-mem）	寄存器（load-store）
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R3, R1, B	Load R2, B
Add	Store C	Store R3, C	Add R3, R1, R2
Pop C			Store R3, C
224 bits	**216 bits**	**240 bits**	**260 bits**

2.2指令集系统结构的分类

按照通用寄存器访问方式划分，有两种通用寄存器系统结构的计算机：

（1） register-memory系统结构，一般指令都可以访问存储器。

（2） register-register或 load-store系统结构，只能通过load和store指令来访问内存

教材提到的纯 memory-memory系统结构，目前不采用的结构，把所有的数据都保存在存储器中。

此外，有的ISA在累加器外扩展了其他寄存器，称为扩展累加器计算机。

2.2指令集系统结构的分类

❖通用寄存器（GPR）出现的原因：

- 寄存器比存储器快
- 编译器使用寄存器很方便，比使用其他存储形式效率更高。

例如：(A*B) – (C*D) –(E*F) 在寄存器系统结构的计算机上，可以按任意顺序来执行三个乘法，但是在栈计算机上则只有一种计算顺序，因为操作数隐含在栈中，且必须多次载入。

- 寄存器用来存放变量，减少了数据流量，加速程序运行（寄存器比存储器快）；改善代码密度（寄存器地址比存储器地址的位数少）。

2.2指令集系统结构的分类

- 通用寄存器ISA运算类指令的两个特性：

- ALU指令中包括两个还是三个操作数。在三个操作数格式中，指令包含一个结果（目的操作数）和两个源操作数。在两个操作数的格式中，有一个既是结果操作数也是源操作数。
- ALU指令中包括多少个存储器操作数。
  典型的ALU指令中所支持的存储器操作数的数量可能是从0~3个不等。

2.2指令集系统结构的分类

运算类指令特点

存储器地址个数	最多操作数个数	系统结构类型	举例
0	3	Load-store（Reg-Reg）	Alpha, ARM, MIPS, PowerPC, SPARC
1	2	Reg-Mem	IBM360/370, Intel 80x86, Motorola 6800, T1 TMS320C54x
2	2	Mem-Mem	VAX(也有2个操作数的格式)
3	3	Mem-Mem	VAX(也有3个操作数的格式)

Figure A.3

2.2 指令集系统结构的分类

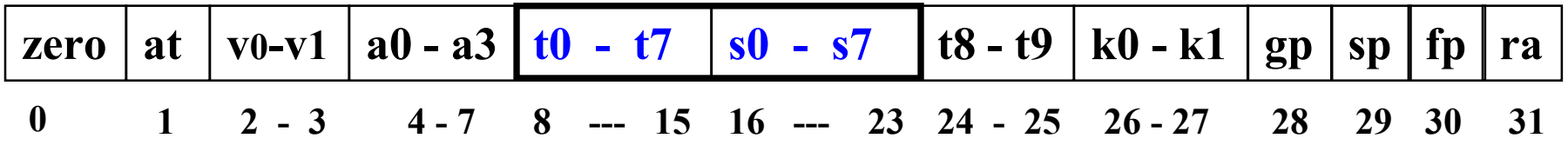
三种常见通用寄存器计算机的优缺点 Figure A.4

类型	优点	缺点
Reg-Reg(0，3)	简单、定长的指令编码；简单的代码生成模式；每条指令运行的时钟周期数相近	目标代码指令数（RISC）比直接访问存储器的系统结构（CISC）多；指令多和指令密度低使程序变得很大
Reg-Mem(1，2)	存储器数据不需要专门的载入指令就可以直接访问；指令格式更加易于编码，代码密度高	由于源操作数在二元操作中被破坏了，所以操作数不是等价的；在一条指令中同时对存储器地址和寄存器号码进行编码会限制寄存器的数量；操作数位置不同使得每条指令执行所需的时钟周期不同
Mem-Mem (2，2)或(3，3)	最紧凑。不浪费寄存器来做临时交换空间	指令长短不相同，特别是三操作数指令；同样，每条指令的操作各不相同；存储器访问带来了存储器瓶颈

(m，n)表示指令的n个操作数中有m个存储器操作数。

MIPS Registers

Name	number	Usage	Reserved on call?
zero	0	constant value =0	
at	1	reserved for assembler	
v0 ~ v1	2 ~ 3	values for results	no
a0 ~ a3	4 ~ 7	Arguments	yes
t0 ~ t7	8 ~ 15	Temporaries	no
s0 ~ s7	16 ~ 23	Saved	yes
t8 ~ t9	24 ~ 25	more temporaries	no
k0 ~ k1	26 ~ 27	reserved for kernel	
gp	28	global pointer	yes
sp	29	stack pointer	yes
fp	30	frame pointer	yes
ra	31	return address	yes



Registers are referenced either by number—$0, … $31,
or by name —$zero, $at… $ra.

MIPS Instruction Formats

- **I-format**: used for instructions with immediates, `lw` and `sw` (since offset counts as an immediate), and branches (`beq` and `bne`)
- **J-format**: used for `j` and `jal`
- **R-format**: used for many other instructions

R-format Example 1

r_format_1.s

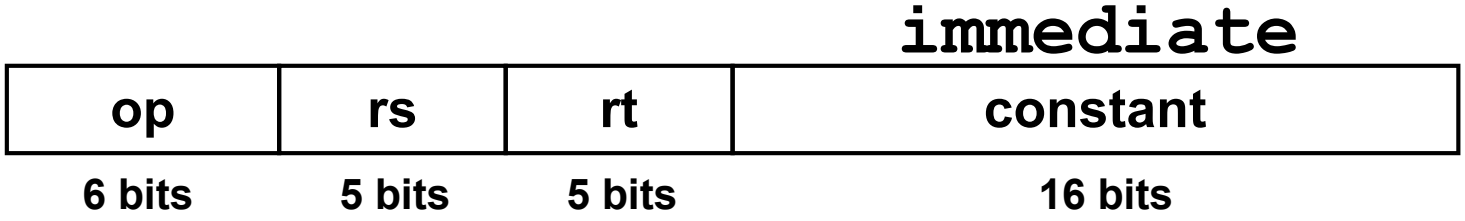
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17_{ten}	18_{ten}	8_{ten}	0	32_{ten}
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

MIPS I-format Instructions



- **Immediate arithmetic and load/store instructions**
  - rs: source register or base address for load/store word
  - rt: destination or source register number for store word
  - immediate: constant added to rs

Demo:add_5_7.s(MIPS)

- # add the numbers
- addi $t0, $zero, 5
- addi $t1, $zero, 7
- add $t2, $t0, $t1

QtSPIM:System Service(MIPS)

Service	System call code ($v0)	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	
allocate memory	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	

Demo: add_5_7.s(MIPS)

```
# Adds two numbers together, 5+7
# prints the result, and then exits

        .text # indicates the start of code

main:         # always start with main
        # add the numbers
        addi $t0, $zero, 5
        addi $t1, $zero, 7
        add  $t2, $t0, $t1

        # print out the result
        li $v0, 1
        move $a0, $t2
        syscall

        # exit the program
        li $v0, 10
        syscall
```

COD_5e Exercise 2.2

- For the following MIPS assembly instructions below, what is a corresponding C statement ?
- 下面的MIPS汇编语言程序段对应的C语言表达式是什么?

```
add     f, g, h
add     f, i, f
```

2.2 Answer

- For the following MIPS assembly instructions below, what is a corresponding C statement?

```
add     f, g, h
add     f, i, f
```

- `f = g + h + i`