Instruction Set Architecture

Consider x := y+z. (x, y, z are memory variables)

1-address instructions

LOAD	y	(r :=y)
ADD	z	(r:=r+z)
STORE	x	(x:=r)

2-address instructions

ADD	y,z	(y := y+z)
MOVE	x,y	(x := y)

3-address instructions

ADD x, y, z (x:= y+z)

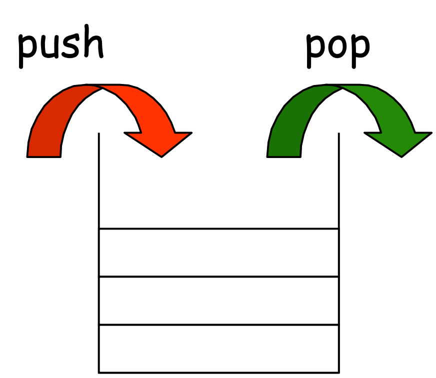
0-address instructions (for stack machines)

PUSH y (on a stack)

PUSH z (on a stack)

ADD

POP x



Points to Consider

- **Special-purpose or general purpose?**
- **Word size and instruction size?**

  [Now most instructions have 32-bits, and machines allow operation on 64-bit data operands]
- **Data types?**

  [Whatever the application demands]
- **0/1/2/3 address instructions, or a mix of them?**

  [Most modern designs allow 3-address instructions, and pack them in a 32-bit frame]
- **How many addressing modes, and which ones?**

  [Whatever the application demands]
- **Register or memory operands?**

  [Register operands can be accessed faster, but you cannot have too many registers]
- **Instruction formats and instruction encoding.**

  [Modern designs have fewer formats and they are less clumsy]

Instruction Types

BASIC INSTRUCTIONS

Data Movement	LOAD, STORE, MOVE
Arithmetic & Logical	ADD, SUB, AND, XOR, SHIFT
Branch	JUMP (unconditional)
	JZ, JNZ (conditional)
Procedure Call	CALL, RETURN
Input Output	Memory-mapped I/O*
Miscellaneous	NOP, EI (enable interrupt)

SPECIAL INSTRUCTIONS

Multimedia instructions (MMX)

Many SIMD or vector instructions operate simultaneously on 8 bytes | 4 half-words | 2 words

Digital Signal Processors include multiply-and-accumulate (MAC) to efficiently compute the dot-product of vectors.

Load Store Architecture

Only LOAD and STORE instructions access the memory. All other instructions use register operands. Used in all RISC machines.

If X,Y,Z are memory operands, then X:= Y+Z will be implemented as

```
LOAD   r1, Y
LOAD   r2, Z
ADD    r1, r2, r3
STORE  r3, X
```

Performance improves if the operand(s) can be kept in registers for most of the time. Registers are faster than memory.

Register allocation problem.

Common Addressing Modes

Op	data type	mode	reg	addr/data/offset

opcode(O) reg (R) address (D)

Mode	meaning
immediate	Operand = D
direct	Operand = M[D]
Register indirect Memory indirect	Operand = M[R] Operand = M[M[D]]
Auto-increment	Operand = M[R] R = R + n (n=1|2|4|8)
Auto-decrement	R = R - n (n = 1|2|4|8) Operand = M[R]
Indexed Scale-index-base (SIB)	Operand = M[R+D] Operand = M[s * R+D]
PC-relative	Operand = M[PC+D]
SP-relative	Operand = M[SP+D]

(Note: R = content of register R)

Question: Why so many addressing modes? Do we need all?

RISC or CISC?

Reduced **I**nstruction **S**et **C**omputers have a small number of simple, frequently used instructions.

Complex **I**nstruction **S**et **C**omputers include as many instructions as users might need to write efficient programs.

Features	CISC	RISC
Semantic Gap	Low	High
Code Size	Small	Large, but RAMs are cheap!
Cost	High	Low
Speed	Fast **only if the compiler generates appropriate code**	Slower, **but the problem is overcome using more registers and pipelining.**

MIPS Architecture

MIPS follows the RISC architecture. It has 32 registers r0-r31. Each register has 32-bits. The conventional use of these registers is as follows:

register	assembly name	Comment
r0	$zero	Always 0
r1	$at	Reserved for assembler
r2-r3	$v0-$v1	Stores results
r4-r7	$a0-$a3	Stores arguments
r8-r15	$t0-$t7	Temporaries, not saved
r16-r23	$s0-$s7	Contents saved for use later
r24-r25	$t8-$t9	More temporaries, not saved
r26-r27	$k0-$k1	Reserved by operating system
r28	$gp	Global pointer
r29	$sp	Stack pointer
r30	$fp	Frame pointer
r31	$ra	Return address

Example assembly language programs

Example 1 $f = g + h - i$

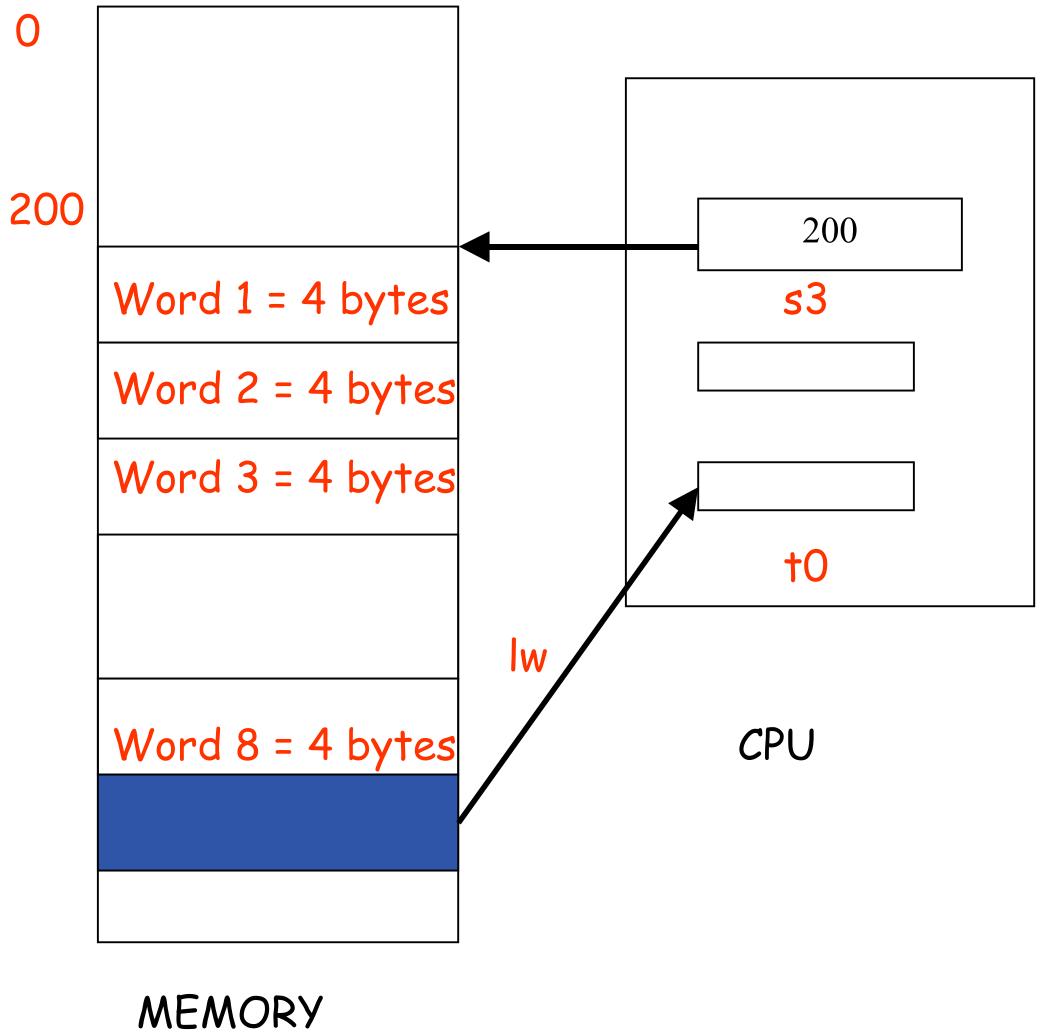
Assume that f, g, h, i are assigned to $s0, $s1, $s2, $s3

```
add $t0, $s1, $s2        # register $t0 contains g + h
sub $s0, $t0, $s3        # f = g + h - i
```

Example 2. $g = h + A[8]$

Assume that g, h are in $s1, $s2. A is an array of words the elements are stored in consecutive locations of the memory. The base address is stored in $s3.

```
lw t0, 32($s3)           # t0 gets A[8], 32= 4x8
add $s1, $s2, $t0        # g = h + A[8]
```


0
200
Word 1 = 4 bytes
Word 2 = 4 bytes
Word 3 = 4 bytes
Word 8 = 4 bytes
MEMORY
200
s3
t0
lw
CPU

Machine language representations

Instruction "add" belongs to the **R-type format.**

opcode	rs	rt	rd	shift amt	function
6	5	5	5	5	6
	↑ src	↑ src	↑ dst		

add \$s1, \$s2, \$t0 (s1 := s2 + t0) will be coded as

0	18	8	17	0	32
6	5	5	5	5	6

The function field is an extension of the opcode, and they together determine the operation.

Note that "sub" has a similar format.

Instruction “lw” (load word) belongs to **I-type format**.

opcode	rs	rt	address
6	5	5	16
	↑ base	↑ dst	↑ offset

lw $t0, 32($s3)   will be coded as

35	19	8	32
6	5	5	16

Both “lw” and “sw” (store word) belong to I-format.

Making decisions

if (i ==j)   f = g + h;   else   f = g - h

Use **bne** = branch-nor-equal, **beq** = branch-equal, and **j** = jump

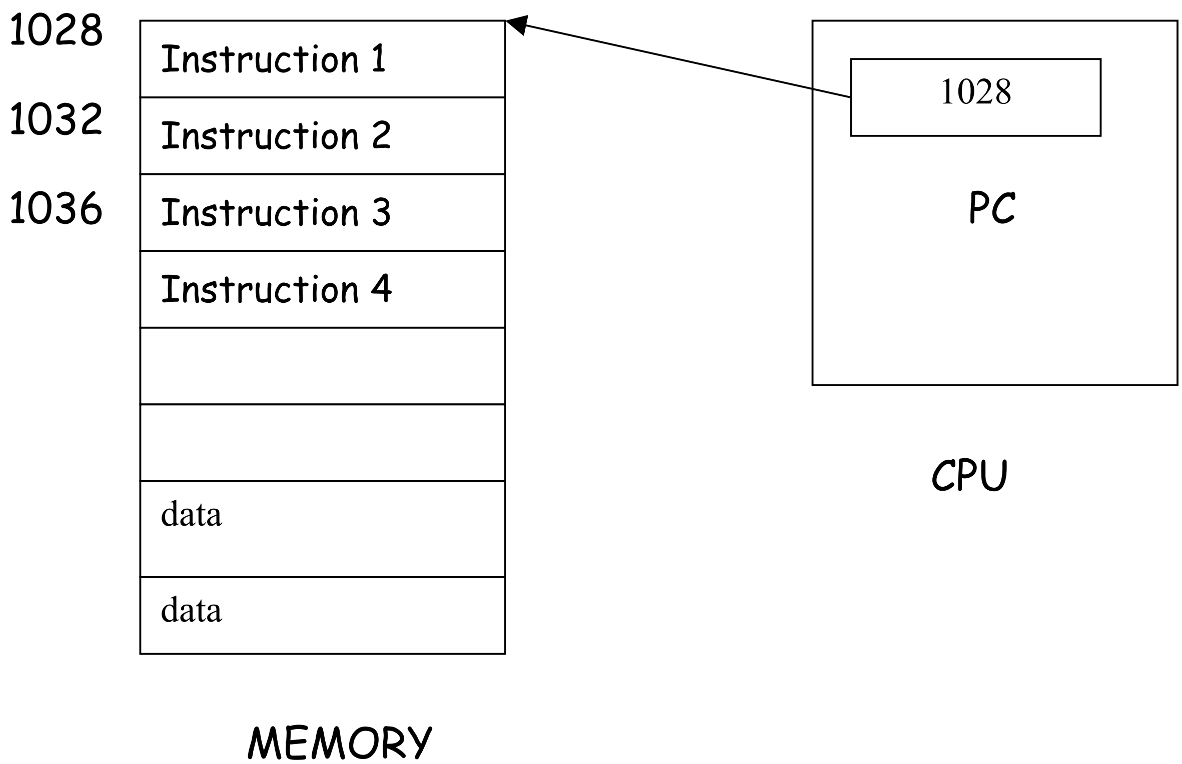
Assume, f, g, h, are mapped into $s0, $s1, $s2, and

i, j are mapped into $s3, $s4

```
        bne $s3, $s4, Else     # goto Else when i=j
        add $s0, $s1, $s2      # f = g + h
        j     Exit             # goto Exit
Else:   sub $s0, $s1, $s2      # f = g - h
Exit:
```

The program counter

Every machine has a **program counter** (called PC) that points to the next instruction to be executed.

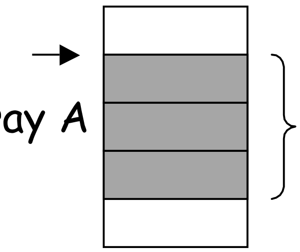


Ordinarily, PC is incremented by 4 after each instruction is executed. A branch instruction alters the flow of control by modifying the PC.

Compiling a while loop

while (A[i] == k)   i = i + j;

array A



Initially \$s3, \$s4, \$s5 contains i, j, k respectively.

Let \$s6 store the base of the array A. Each element of A is a 32-bit word.

```
Loop:   add $t1, $s3, $s3      # $t1 = 2*i
        add $t1, $t1, $t1      # $t1 = 4*i
        add $t1, $t1, $s6      # $t1 contains address
                               of A[i]
        lw $t0, 0($t1)         # $t0 contains $A[i]
        add $s3, $s3, $s4      # i = i + j
        bne $t0, $s5, Exit     # goto Exit if A[i] ≠ k
        j   Loop               # goto Loop
Exit:   <next instruction>
```

Note the use of pointers.

Compiling a switch statement

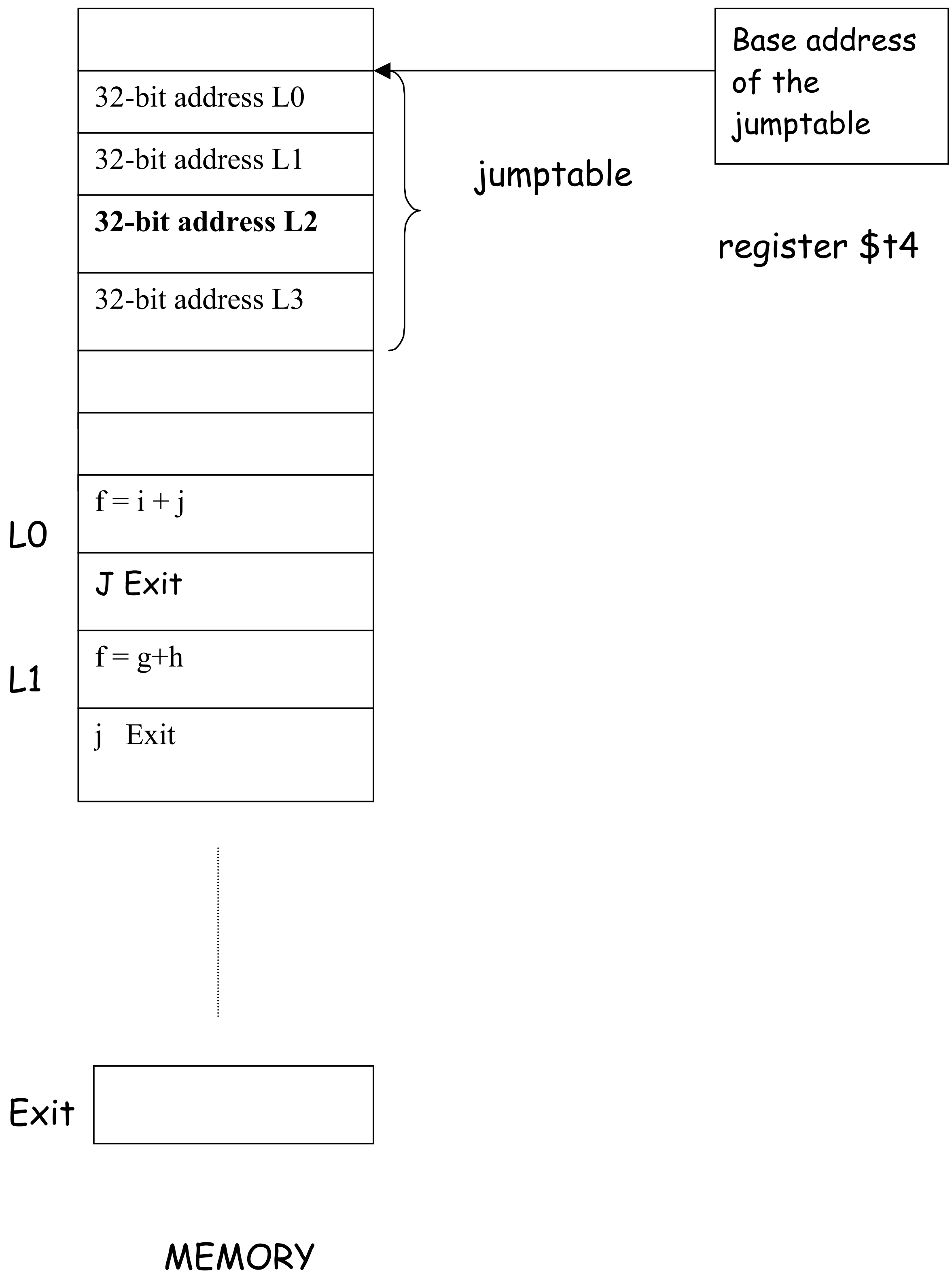
```
switch (k) {
    case 0:   f = i + j; break;
    case 1:   f = g + h; break;
    case 2:   f = g – h; break;
    case 3:   f = I – j; break;
}
```

Assume, \$s0-\$s5 contain f, g, h, i, j, k.

Assume \$t2 contains 4.

```
slt $t3, $s5, $zero       # if k<0 then $t3 = 1 else $t3=0
bne $t3, $zero, Exit      # if k<0 then Exit
slt $t3, $s5, $t2         # if k<4 then $t3 = 1 else $t3=0
beq $t3, $zero, Exit      # if k≥ 4 the Exit
```

What next? Jump to the right case!


32-bit address L0
32-bit address L1
32-bit address L2
32-bit address L3
jumptable
Base address of the jumptable
register $t4
L0
f = i + j
J Exit
L1
f = g+h
j Exit
Exit
MEMORY

Here is the remainder of the program;

```
      add $t1, $s5, $s5
      add $t1, $t1, $t1
      add $t1, $t1, $t4
      lw $t0, 0($t1)
      jr $t0
L0:   add $s0, $s3, $s4
      J Exit
L1:   add $s0, $s1, $s2
      J Exit
L2:   sub $s0, $s1, $s2
      J Exit
L3:   sub $s0, $s3, $s4
Exit: <next instruction>
```