MIPS registers

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 later use
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

MIPS insruction formats

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 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.**

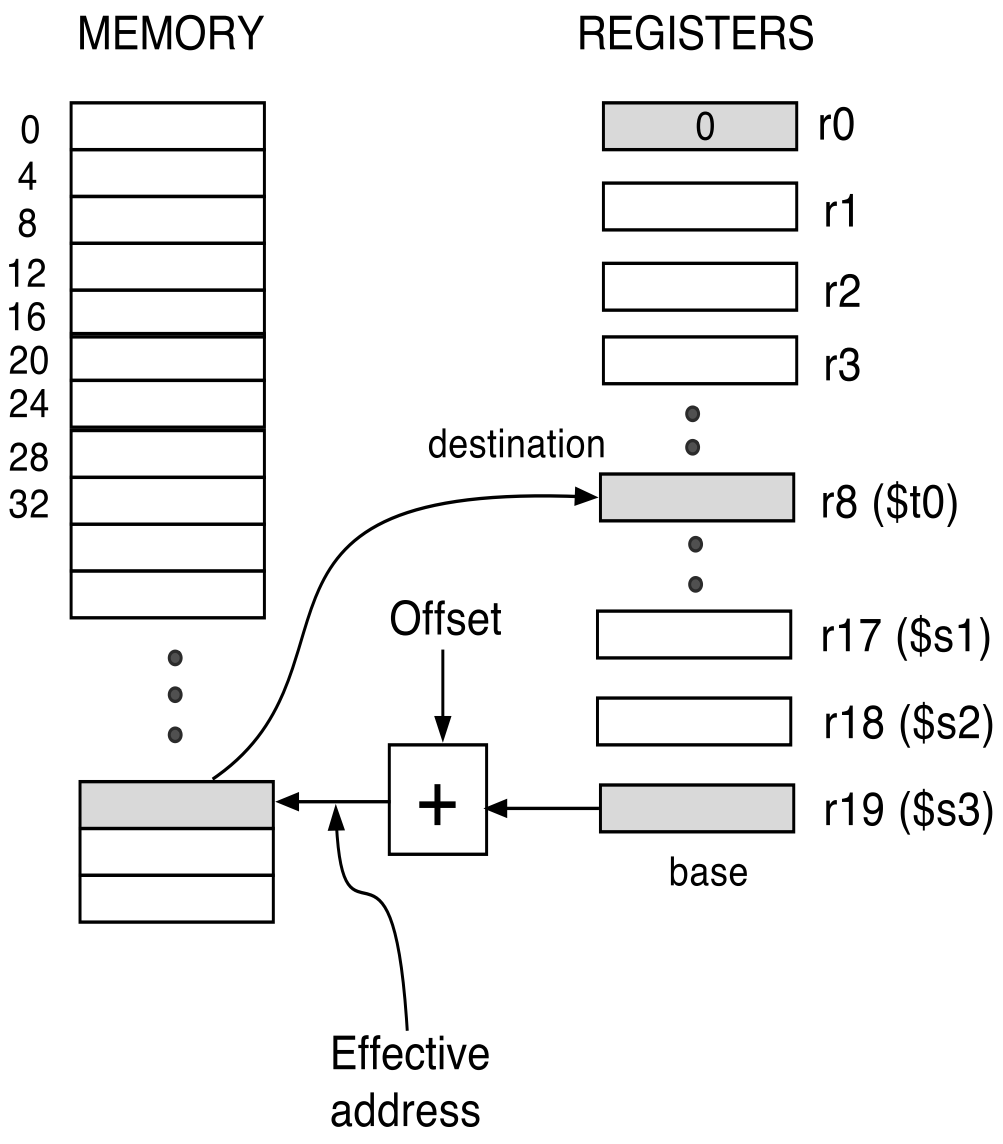
opcode	rs	rt	address
6	5	5	16
	↑	↑	↑
	base	dst	offset

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

35	19	8	32
6	5	5	16

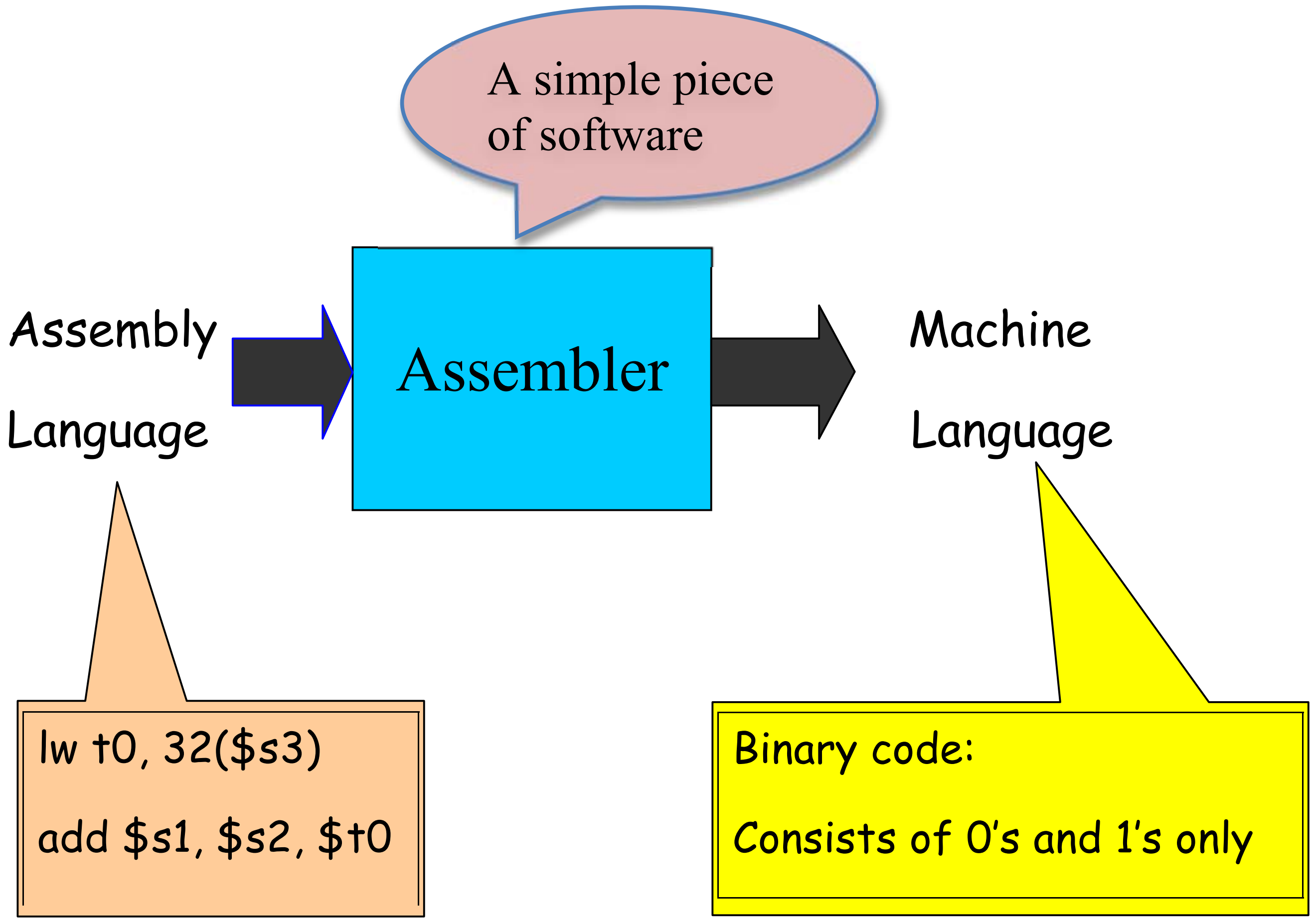
Both “lw” and “sw” (store word) belong to I-format.

MIPS has (fortunately) only three different instruction formats. The operation codes determine the format. This is how the control unit interprets the instructions.



(LW) LoadWord destination, offset($base register)

What is an Assembler?



If you know the instruction formats, then you can translate it. The machine language consists of 0's and 1's

Pseudo-instructions

(Makes your life a bit simpler)

These are simple assembly language instructions that do not have a direct machine language equivalent. During assembly, the assembler translates each pseudo-instruction into one or more machine language instructions.

Example

move $t0, $t1   # $t0 ← $t1

The assembler will translate it to

add $t0, $zer0, $t1

We will see more of these soon.

Think about these

Q1. How will you load a constant into a memory location (i.e. consider implementing x :=3)?

(Need some immediate mode instructions, like li which is a pseudo-instruction)

Q2. How will you implement x:= x+1 in assembly language?

What do you think?

Q3. Why is the load (and store too) instruction so "crooked?"

Used for its flexibility, let us discuss it.

Q4. How will you load a constant (say 5) into a register?

(Need the immediate mode instructions, like addi)

Loading a 32-bit constant into a register

The pseudo-instruction "load immediate"

```
li $s0, 0x 003A0012
```

hexadecimal

means "load the 32-bit constant into register $s0."

Internally it is translated into

```
lui $s0, 42          # load upper-half immediate
ori $s0, $s0, 18     # (one can also use andi)
```

Logical Operations

Shift left (logical)	sll
Shift right (logical)	srl
Bit-by-bit AND	and, andi (and immediate)

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

sll $t2, $s0, 4 means $t2 = $s0 << 4 bit position

(s0 = $16, t2 = $10)

0	0	16	10	4	0
6	5	5	5	5	6

s0 = 0000 0000 0000 0000 0000 0000 0000 1001

t2 = 0000 0000 0000 0000 0000 0000 1001 0000

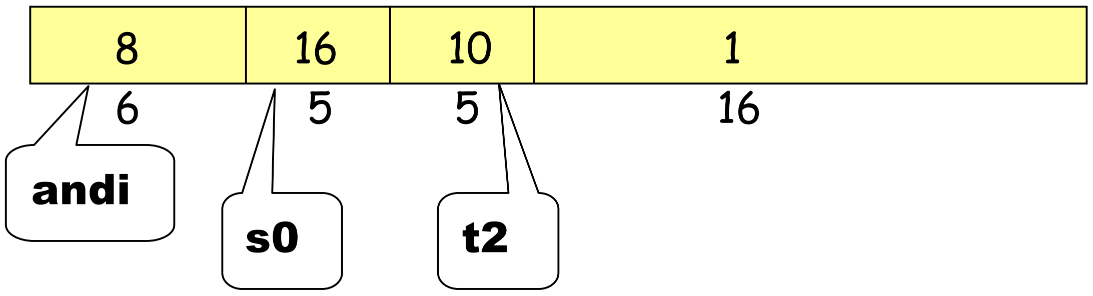
Why are these instructions useful?

Using AND for bit manipulation

To check if a register $s0 contains an odd number, AND it with a mask that contains all 0's except a 1 in the LSB position, and check if the result is zero (we will discuss decision making later)

```
andi $t2, $s0, 1
```

This uses **I-type format** (why?):



Now we have to test if $t2 = 1 or 0

Making decisions

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

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

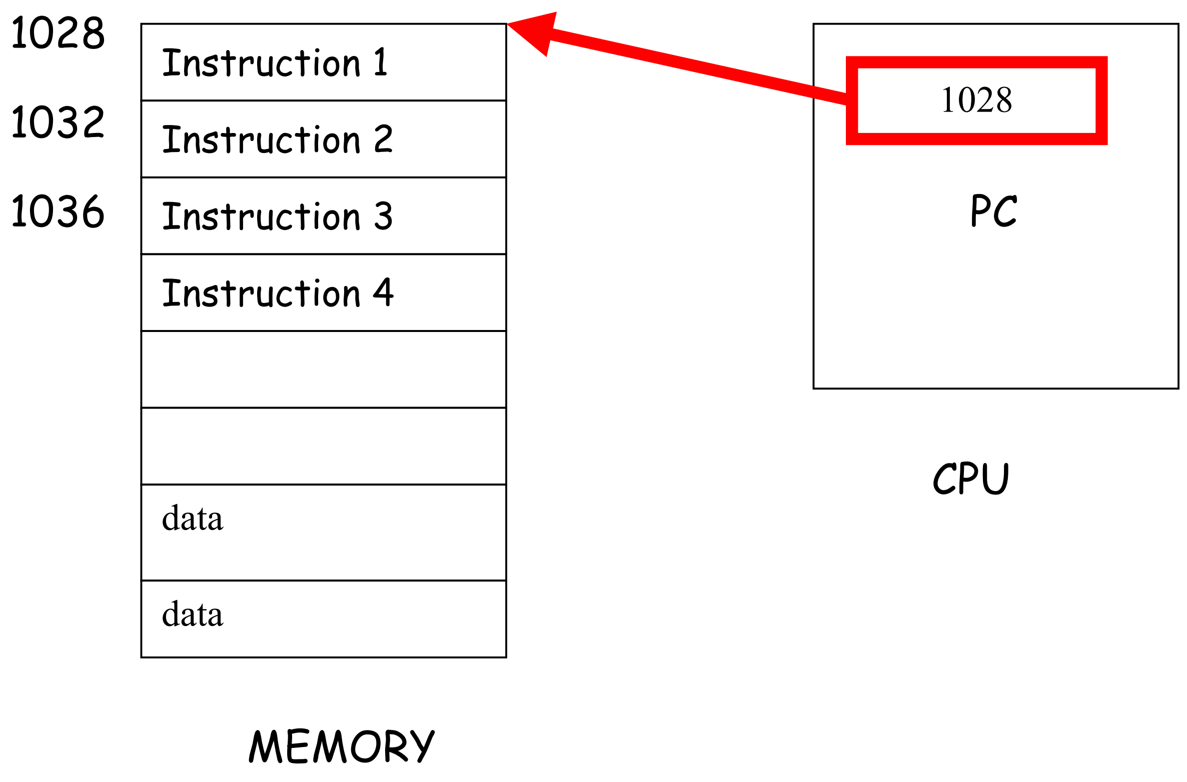
Assume that f, g, h, are mapped into $s0, $s1, $s2

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  (something new**)
Else:   sub $s0, $s1, $s2       # f = g – h
Exit:
```

The program counter and control flow

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;

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>
```

Anatomy of a MIPS assembly language program running on the MARS simulator

```
        .data
L1: .word 0x2345          # some arbitrary value
L2: .word 0x3366          # some arbitrary value
Res: .space 4
```

```
         .text
         .globl main
main:  lw $t0, L1($0)           #load the first value
         lw $t1, L2($0)          # load the second value
         and $t2, $t0, $t1       # compute bit-by-bit AND
         or $t3, $t0, $t1        # compute  bit-by-bit OR
         sw $t3, Res($0)         # store result in memory
```

```
        li $v0, 10              # code for program end
        syscall
```

Another example of input-output

```
        .data
str1:   .asciiz         "Enter the number:"
        .align 2        #move to a word boundary
res:    .space 4        # reserve space to store result

        .text
        .globl main
main:   li $v0, 4       # code to print string
        la $a0, str1
        syscall

        li $v0, 5       # code to read integer
        syscall

        move $t0, $v0       # move the value to $t0
        add $t1, $t0, $t0   # multiply by 2
        sw $t1, res($0)     # store result in memory

        li $v0, 1       # code to print integer
        move $a0, $t1   # move value to be printed to $a0
        syscall         # print to the screen

        li $v0, 10      # code for program end
        syscall
```