

High-level vs. Assembly language

Consider the following statements

```
1.  a = x + y - z
2.  if x > y
      then x := x + y
      else x := x - y
```

How does a processor execute these?

HLL (High Level Language) programs are machine independent. They are easy to learn, easy to use, and convenient for managing complex tasks.

Assembly language programs are machine specific. It is the language that the processor directly understands.

Understanding Assembly Language

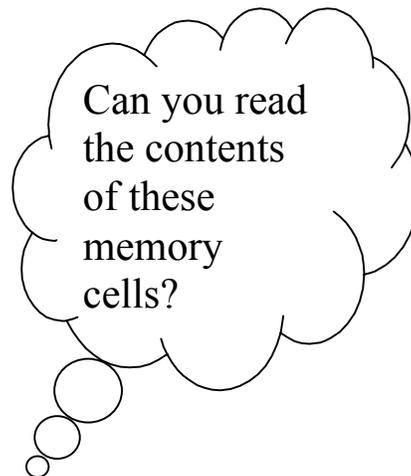
Let us begin with data representation. How to represent

- Signed integers
- Fractions
- Alphanumeric characters
- Floating point numbers
- Pictures?

Review

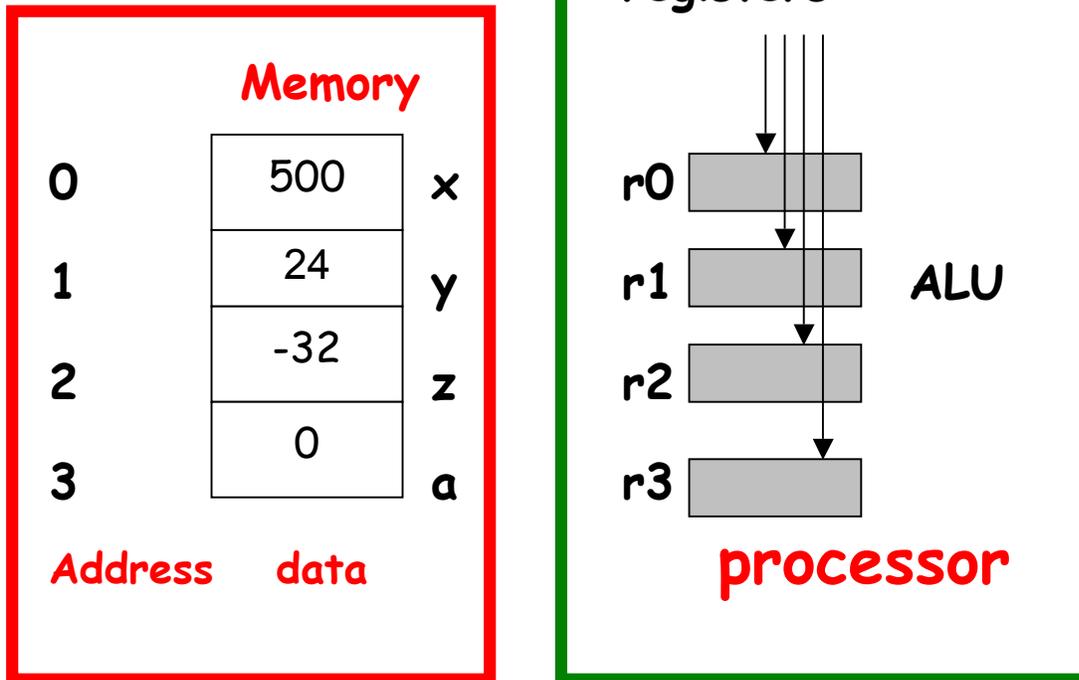
Memory

0 1 0 0 1 0 1 1
1 1 0 1 1 0 1 0
1 0 0 1 1 0 0 0

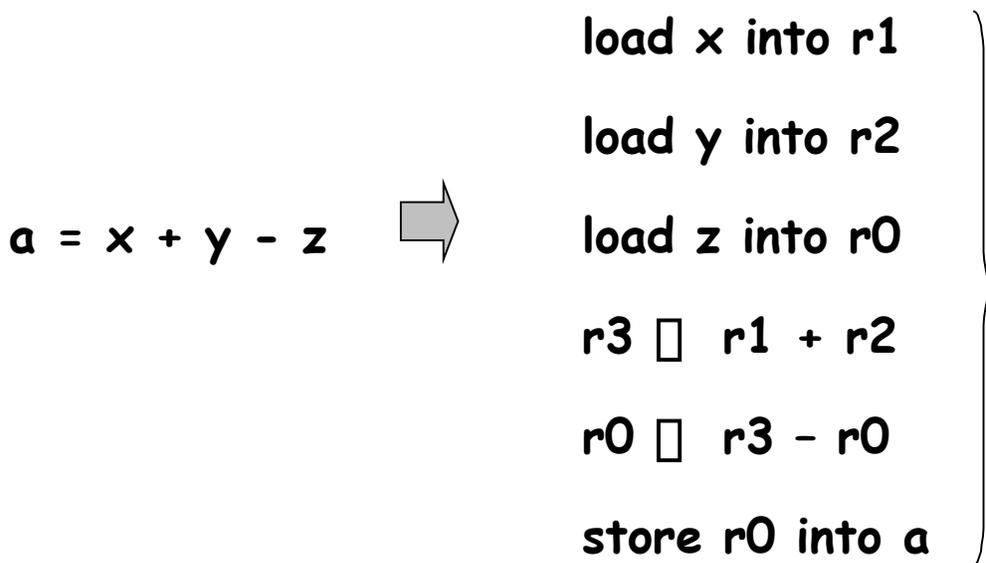


Visualizing instruction execution

(The main concept is register-transfer operation.)



A register is a fast storage within the CPU



Assembly language instructions for a hypothetical machine (not MIPS)

```
Load x, r1
Load y, r2
Load z, r0
Add r3, r1, r2
Sub r0, r3, r0
Store r0, a
```

Each processor has a different set of registers, and different assembly language instructions. The assembly language instructions of Intel Pentium and MIPS are completely different.

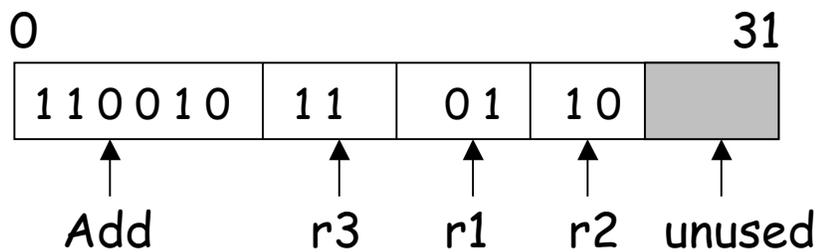
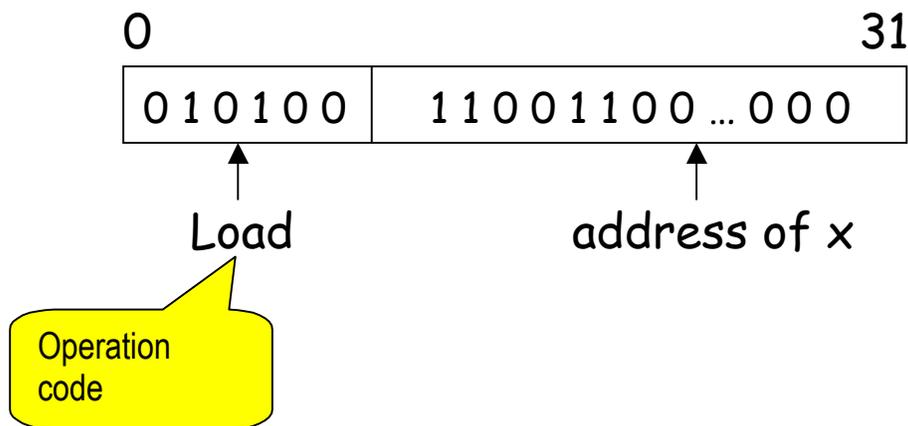
Motorola 68000 has 16 registers r0-r15

MIPS has 32 registers r0-r31

Pentium has 8 general purpose & 6 segment registers.

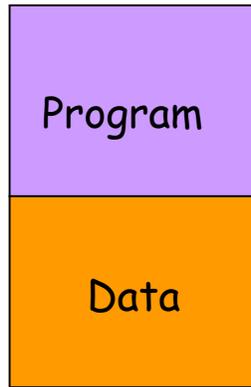
Binary or Machine Language program

Both program and data are represented **using only 0's and 1's** inside a computer. Here is a sample:



These are **instruction formats**. Each instruction has a specific format.

Can we distinguish program from data?



Both are bit strings.
Indistinguishable.

MEMORY

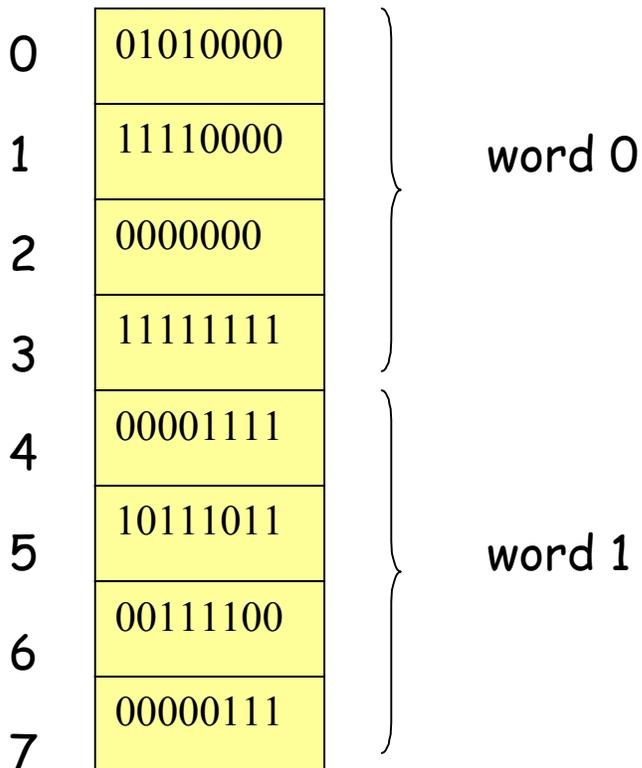
Normally, the programmer has to tell the machine (or use some convention) to specify the address of the first instruction. Incorrect specification will lead to errors, and the program is most likely to crash.

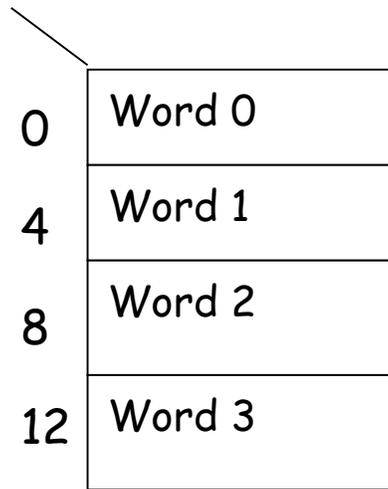
Bits, bytes, words

Bit: 0, 1

Byte: string of 8 bits. Each byte has an address.

Word: one or more bytes (usually 2 or 4 or 8).





Byte order in a word

Big Endian order [byte 0, byte 1, byte 2, byte 3]

Little Endian order [byte 3, byte 2, byte 1, byte 0]

Registers vs. memory

Data can be stored in **registers or memory locations**. **Memory access is slower (takes approximately 50 ns) than register access (takes approximately 1 ns or less)**.

To increase the speed of computation it pays to keep the variables in registers as long as possible. However, due to technology limitations, the number of registers is quite limited (typically 8-64).

MIPS registers

MIPS has 32 registers r0-r31. 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 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

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 $A[8]$, $32 = 4 \times 8$
add \$s1, \$s2, \$t0	# $g = h + A[8]$

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 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.