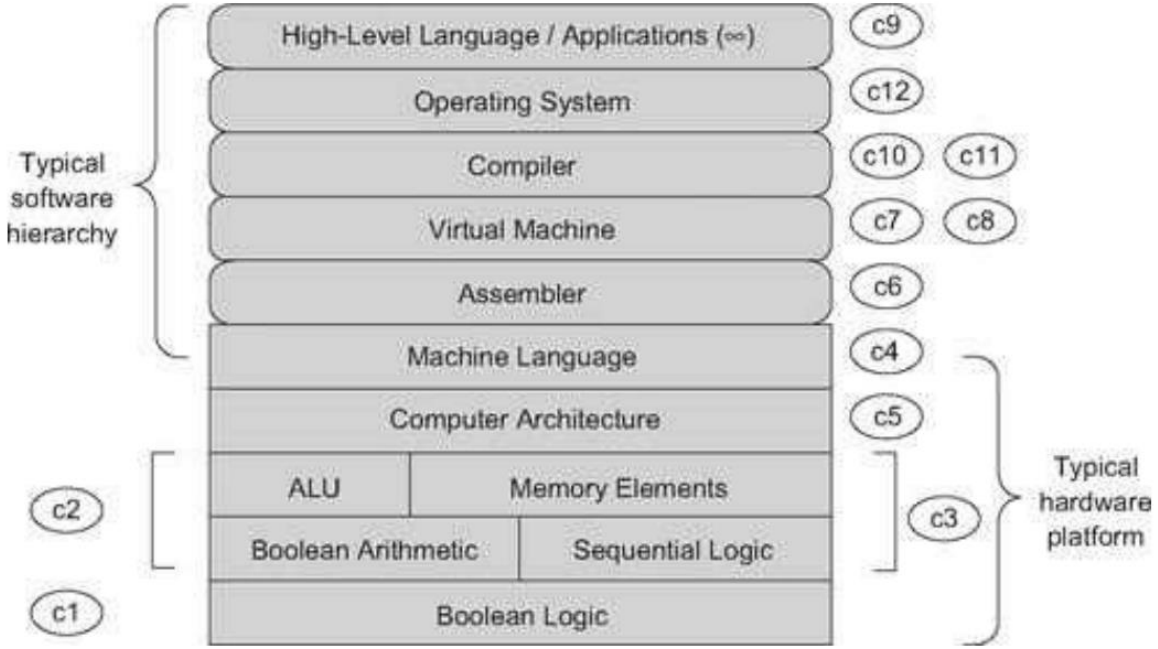
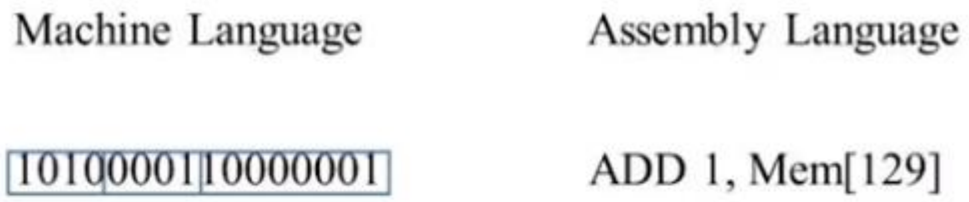
Arquitectura del Computador II

Repaso.
Parte II


High-Level Language / Applications (∞)
c9
Operating System
c12
Compiler
c10
c11
Virtual Machine
c7
c8
Assembler
c6
Machine Language
c4
Computer Architecture
c5
ALU
Memory Elements
Boolean Arithmetic
Sequential Logic
Boolean Logic
c2
c1
c3
Typical software hierarchy
Typical hardware platform

Símbolos



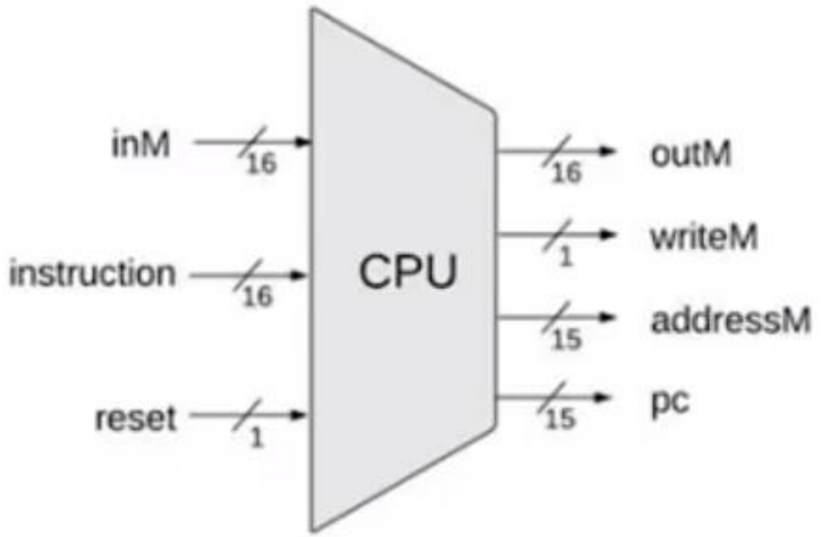
Machine Language	Assembly Language
1010 0001 10000001	ADD 1, Mem[129]

Sample Hack instructions:

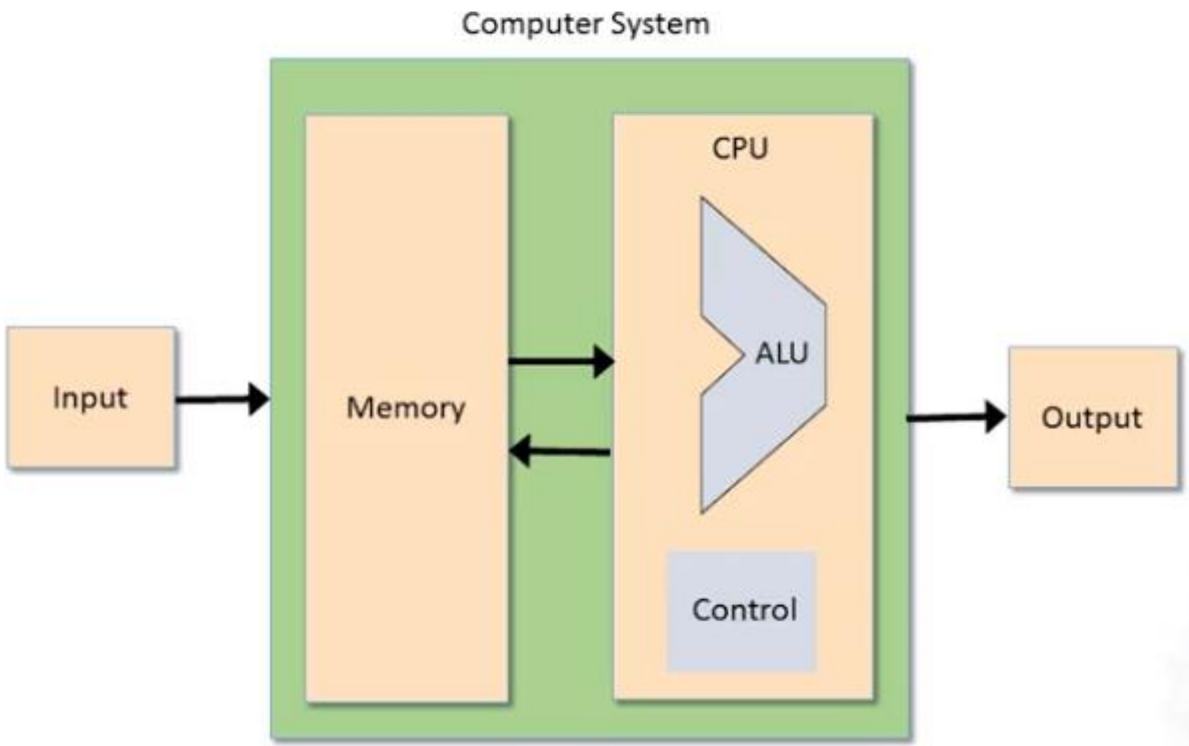
D = D-A

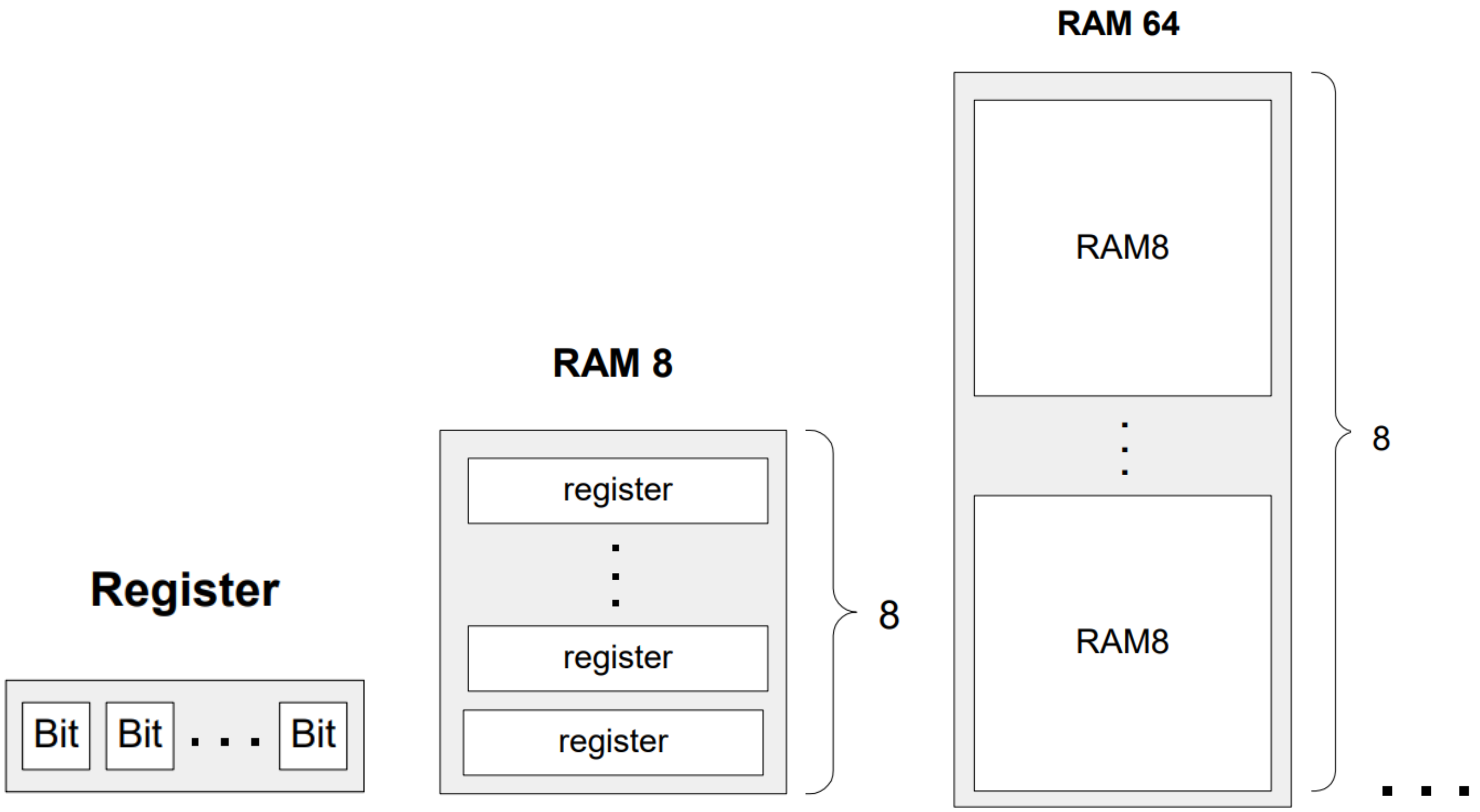
@17

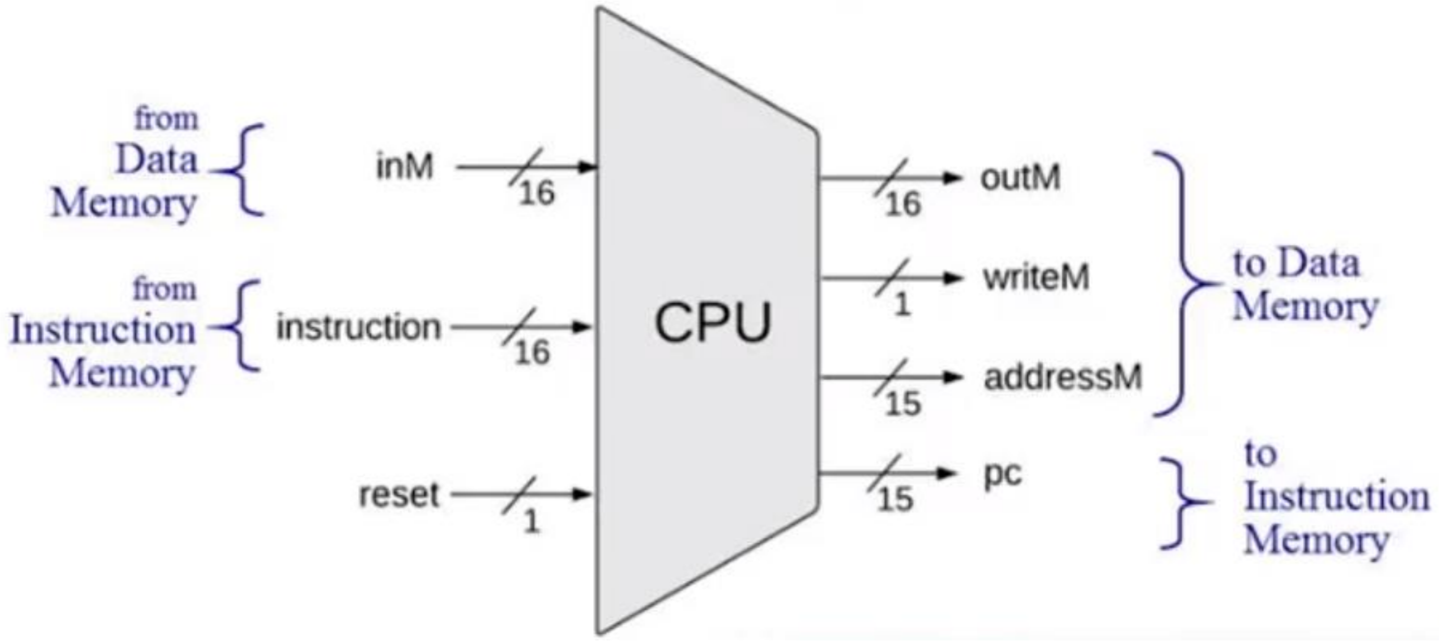
M = M+1

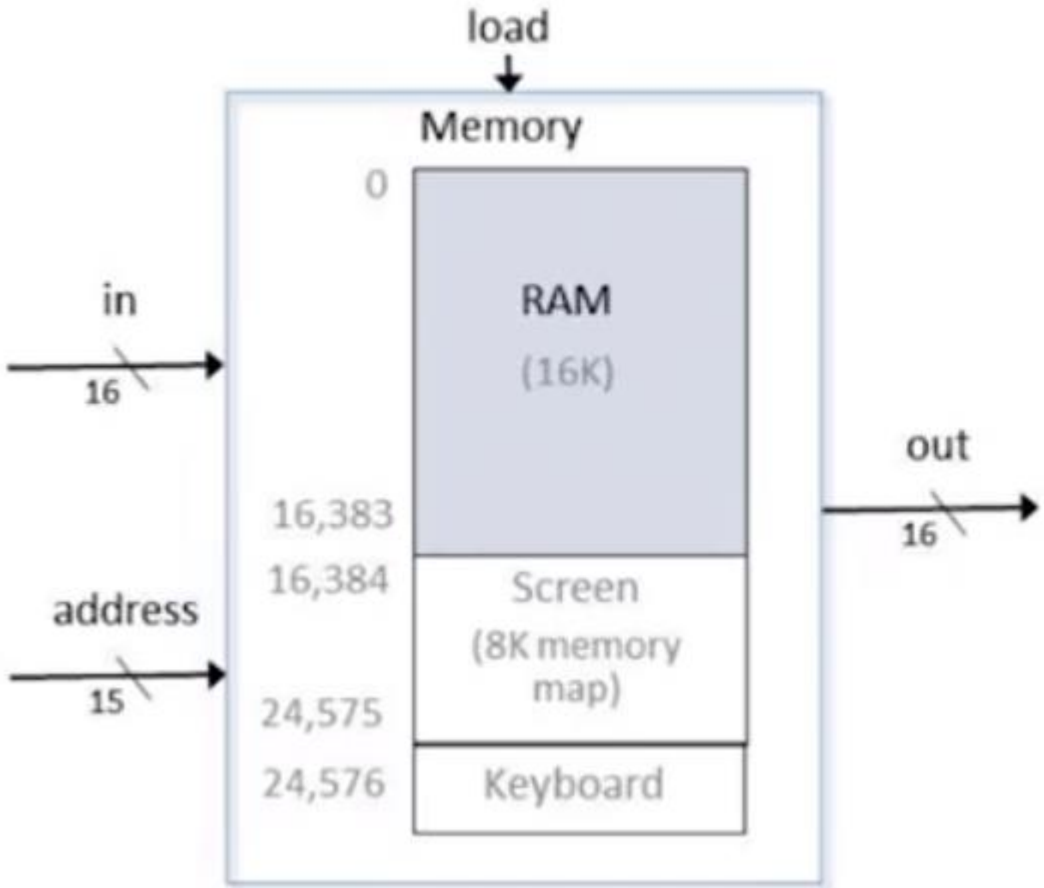
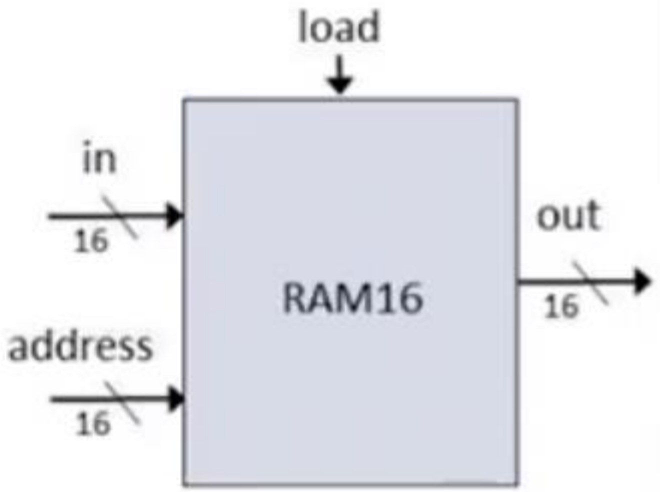


Von Neumann Architecture:



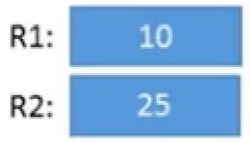

RAM 64
RAM8
RAM8
8
RAM 8
register
register
register
8
Register
Bit
Bit
Bit
. . .


from Data Memory
inM
16
from Instruction Memory
instruction
16
reset
1
CPU
outM
16
writeM
1
addressM
15
pc
15
to Data Memory
to Instruction Memory

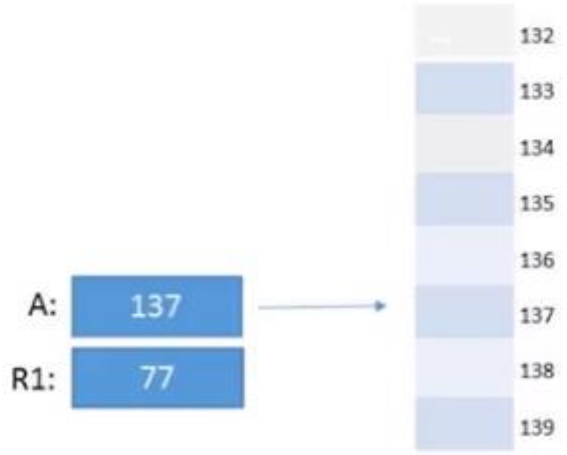

load
Memory
0
RAM
(16K)
16,383
16,384
Screen
(8K memory map)
24,575
24,576
Keyboard
in
16
address
15
out
16

load
in
16
address
16
RAM16
out
16

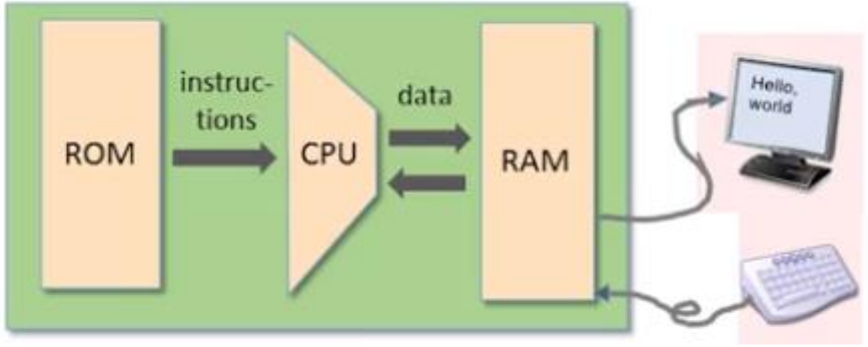
Registros

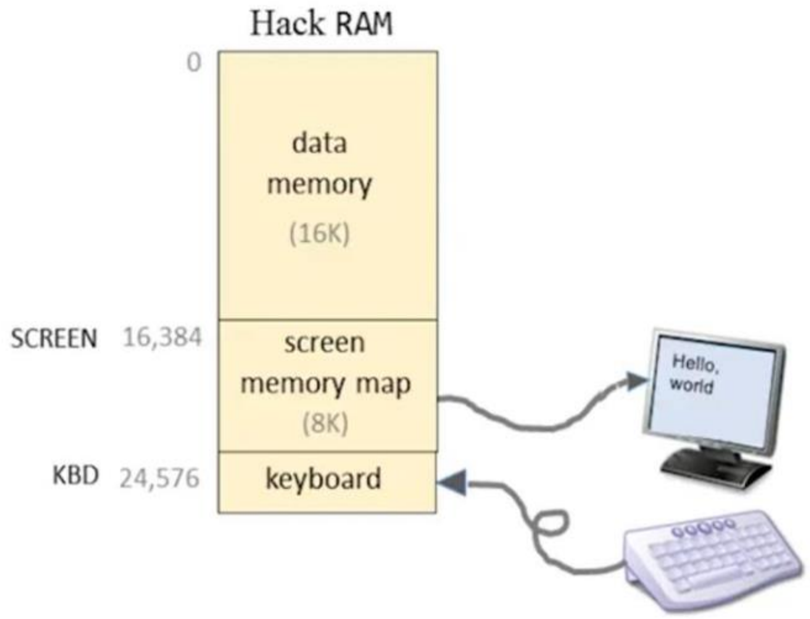
- Data Registers
  - Add R1, R2



- Address Registers
  - Store R1, @A

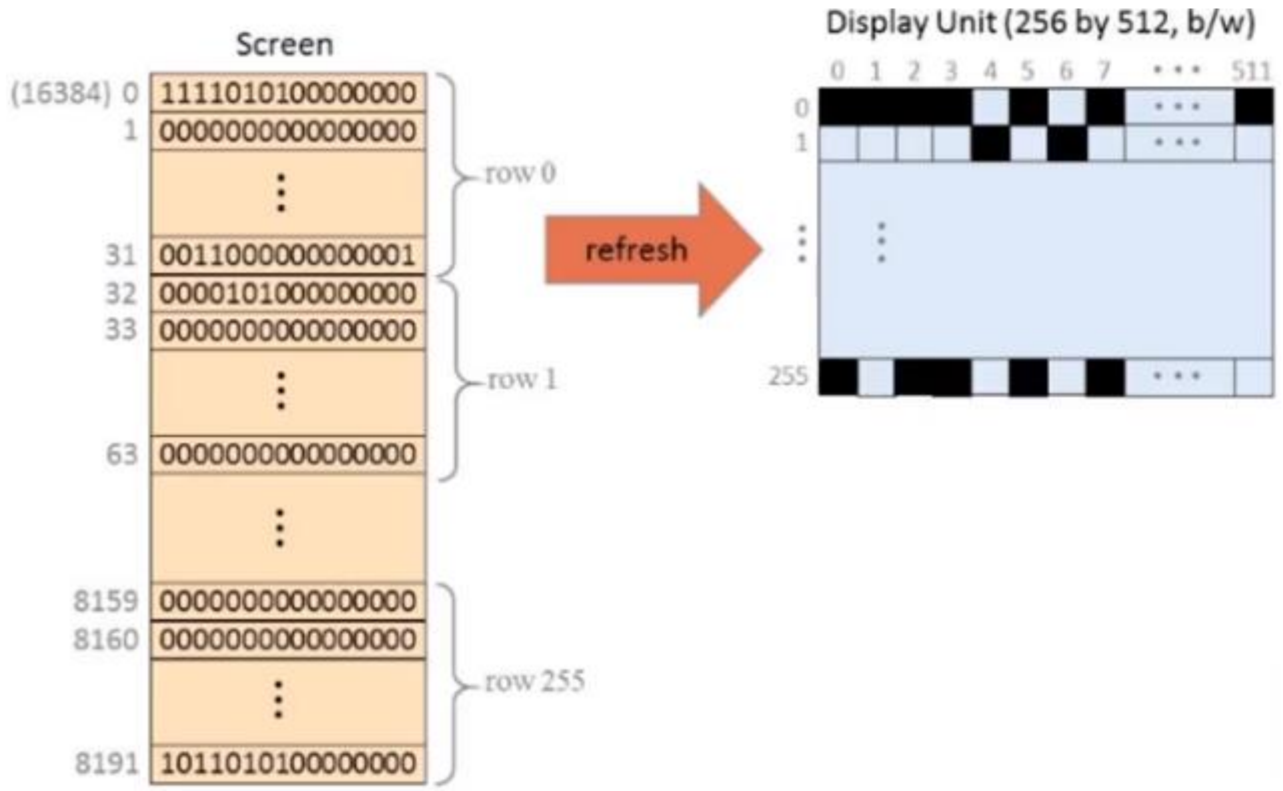


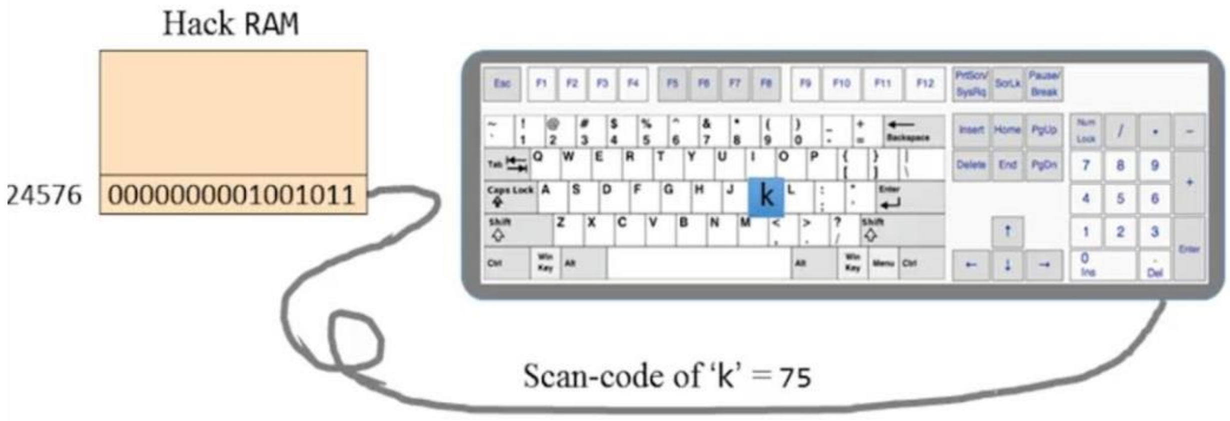

ROM
instruc-
tions
CPU
data
RAM
Hello,
world



<u>Hack language convention:</u>

- SCREEN: base address of the screen memory map
- KBD: address of the keyboard memory map


Screen
(16384) 0
1111010100000000
1
0000000000000000
31
0011000000000001
32
0000101000000000
33
0000000000000000
63
0000000000000000
8159
0000000000000000
8160
0000000000000000
8191
1011010100000000
row 0
row 1
row 255
refresh
Display Unit (256 by 512, b/w)
0 1 2 3 4 5 6 7 • • • 511
0
1
255



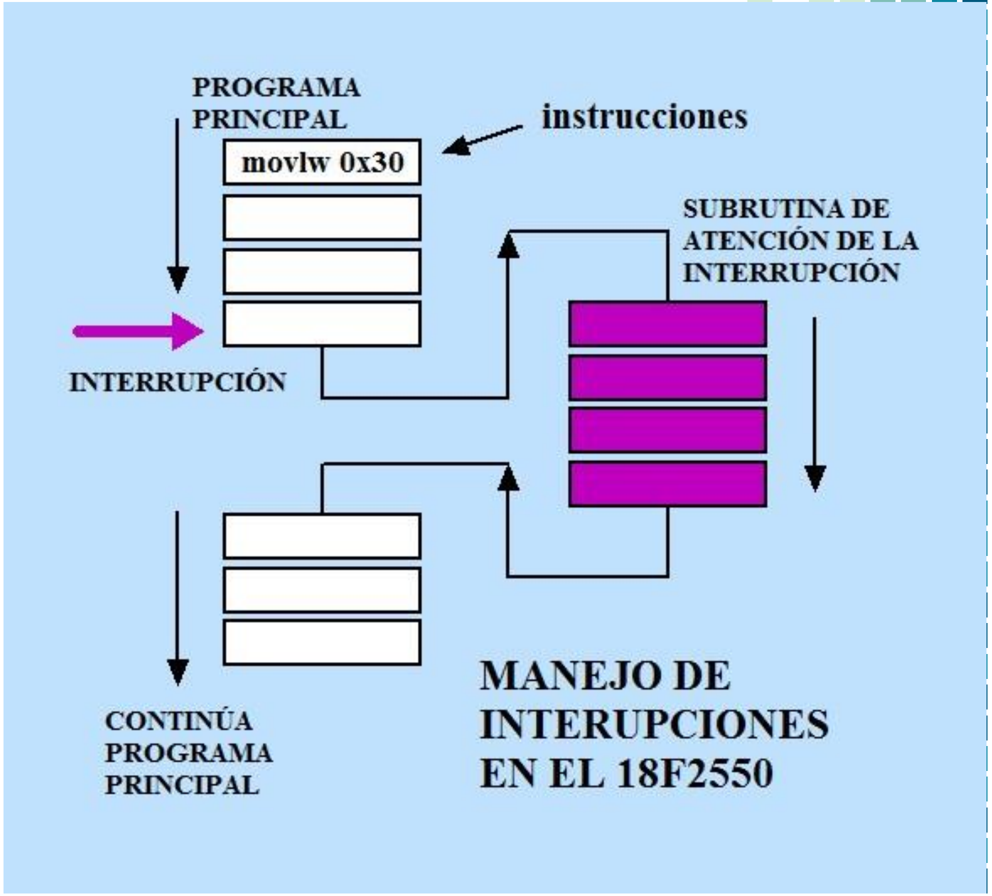
<u>To check which key is currently pressed:</u>

•Read the contents of `RAM[24576]` (address `KBD`)

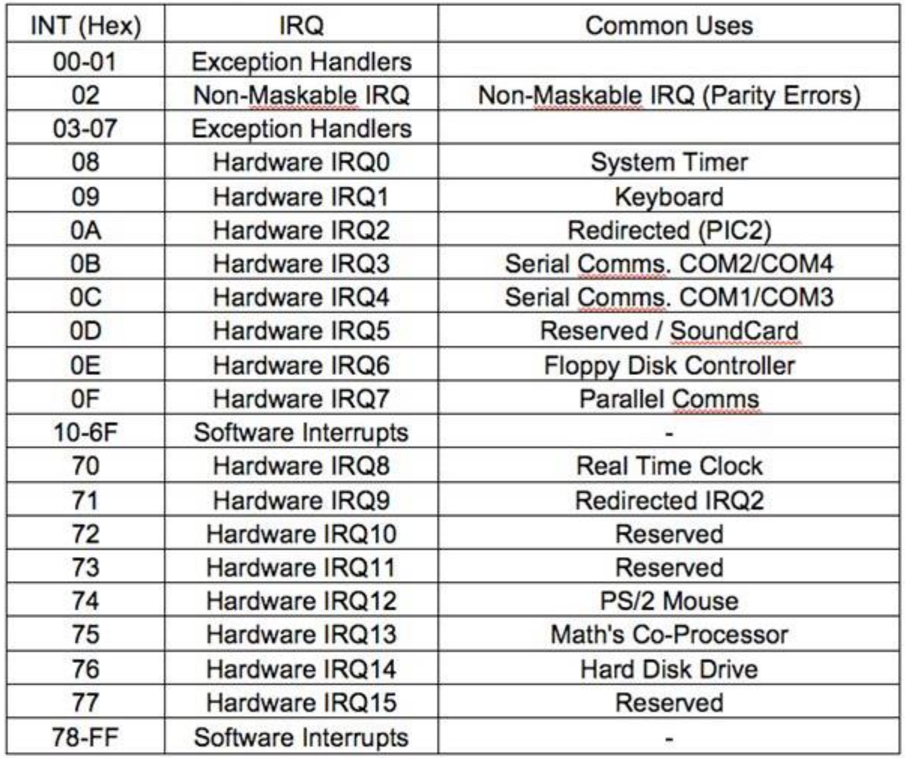
•If the register contains 0, no key is pressed

•Otherwise, the register contains the scan code of the currently pressed key.

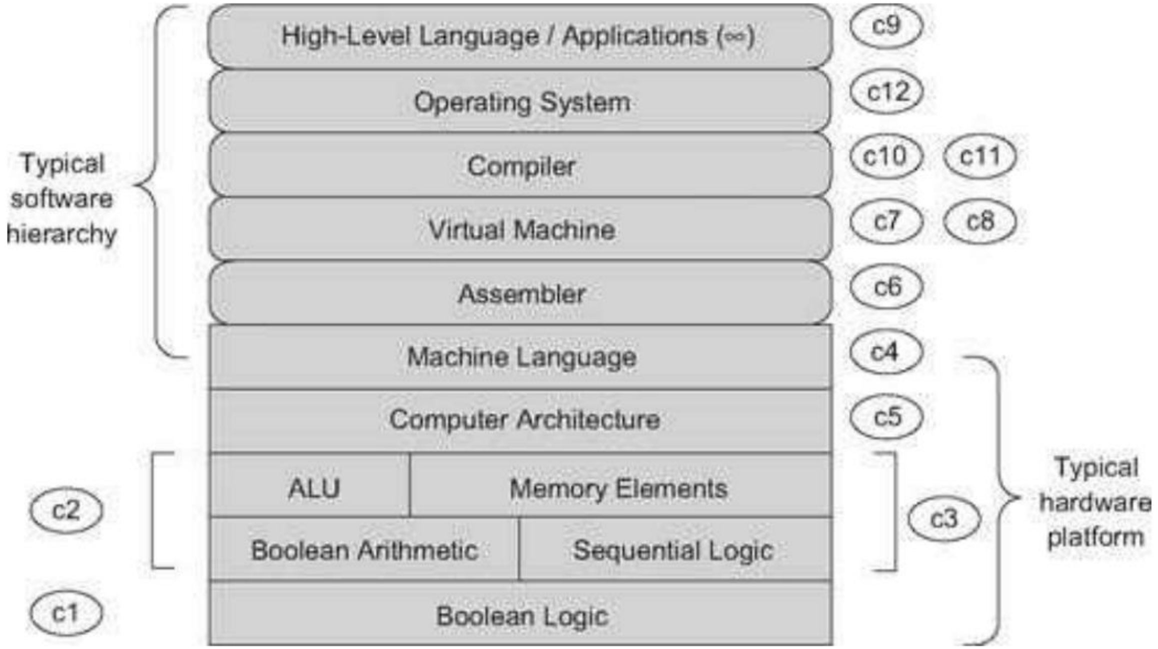
Vector de Interrupciones


PROGRAMA PRINCIPAL
instrucciones
movlw 0x30
SUBRUTINA DE ATENCIÓN DE LA INTERRUPCIÓN
INTERRUPCIÓN
CONTINÚA PROGRAMA PRINCIPAL
MANEJO DE INTERUPCIONES EN EL 18F2550

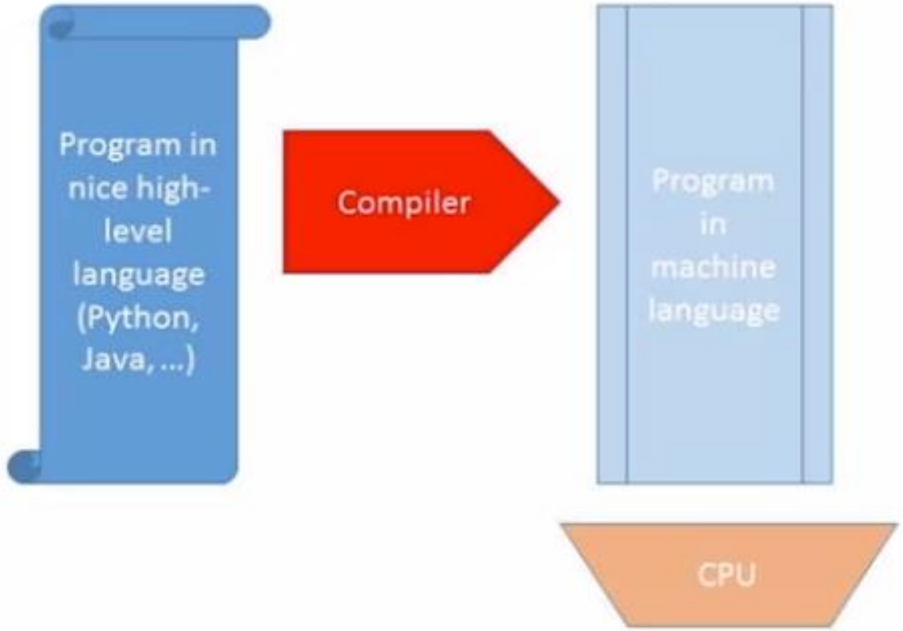
- Vector de interrupciones x86

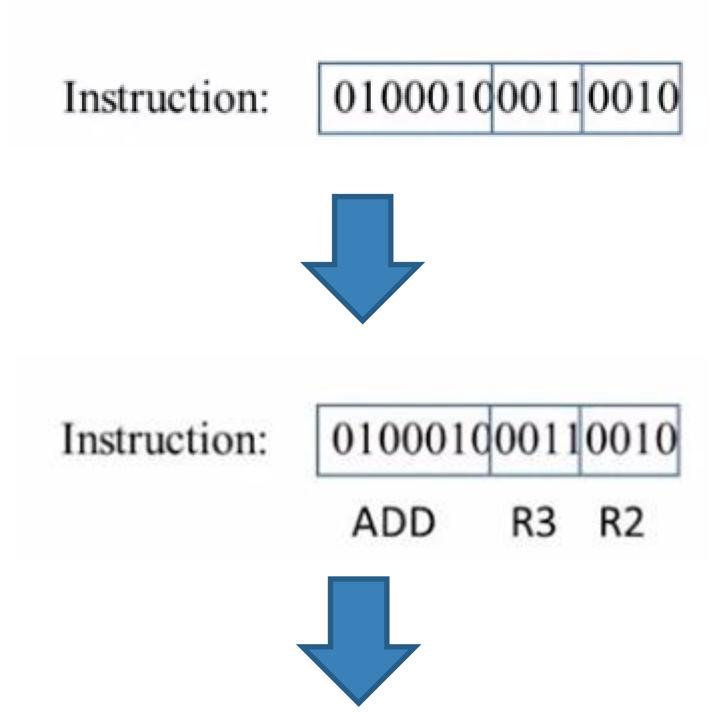


INT (Hex)	IRQ	Common Uses
00-01	Exception Handlers	
02	Non-Maskable IRQ	Non-Maskable IRQ (Parity Errors)
03-07	Exception Handlers	
08	Hardware IRQ0	System Timer
09	Hardware IRQ1	Keyboard
0A	Hardware IRQ2	Redirected (PIC2)
0B	Hardware IRQ3	Serial Comms. COM2/COM4
0C	Hardware IRQ4	Serial Comms. COM1/COM3
0D	Hardware IRQ5	Reserved / SoundCard
0E	Hardware IRQ6	Floppy Disk Controller
0F	Hardware IRQ7	Parallel Comms
10-6F	Software Interrupts	-
70	Hardware IRQ8	Real Time Clock
71	Hardware IRQ9	Redirected IRQ2
72	Hardware IRQ10	Reserved
73	Hardware IRQ11	Reserved
74	Hardware IRQ12	PS/2 Mouse
75	Hardware IRQ13	Math's Co-Processor
76	Hardware IRQ14	Hard Disk Drive
77	Hardware IRQ15	Reserved
78-FF	Software Interrupts	-



High-Level Language / Applications (∞)
Operating System
Compiler
Virtual Machine
Assembler
Machine Language
Computer Architecture
ALU
Memory Elements
Boolean Arithmetic
Sequential Logic
Boolean Logic
c9
c12
c10
c11
c7
c8
c6
c4
c5
c3
c2
c1
Typical software hierarchy
Typical hardware platform


Program in nice high-level language (Python, Java, ...)
Compiler
Program in machine language
CPU

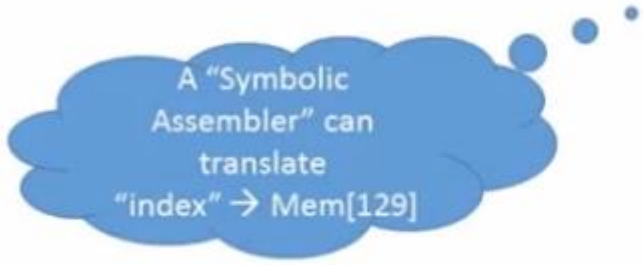


Assembly Language

```
ADD 1, Mem[129]
```

(location 129 in memory holds the “index”)

```
ADD 1, index
```



Alto nivel

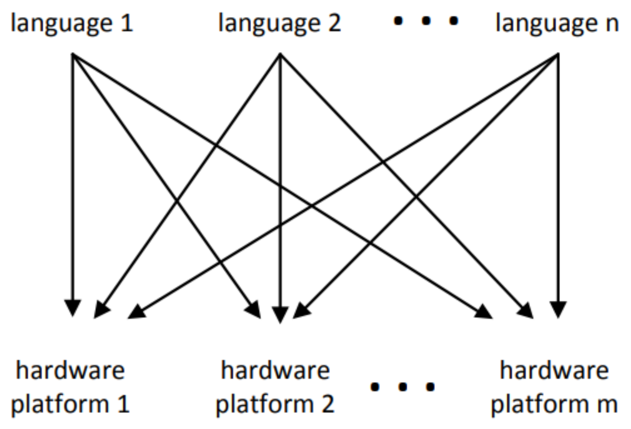
```
class Main {
  static int x;

  function void main() {
    // Inputs and multiplies two numbers
    var int a, b, x;
    let a = Keyboard.readInt("Enter a number");
    let b = Keyboard.readInt("Enter a number");
    let x = mult(a,b);
    return;
  }
}

  // Multiplies two numbers.
  function int mult(int x, int y) {
    var int result, j;
    let result = 0; let j = y;
    while ~(j = 0) {
      let result = result + x;
      let j = j - 1;
    }
    return result;
  }
}
```

Compilador

Código de maquina

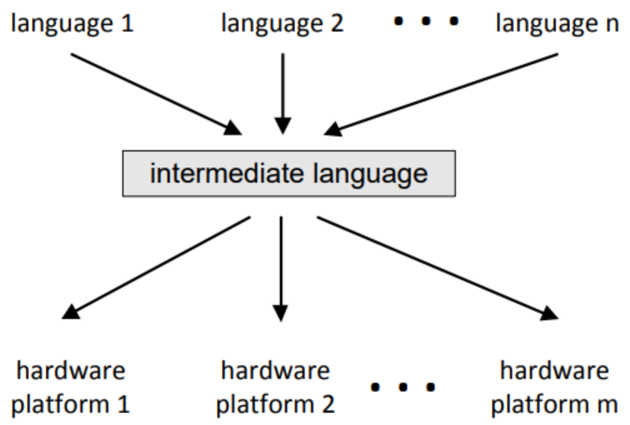
```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
```

Compilación directa



requires $n \cdot m$ translators

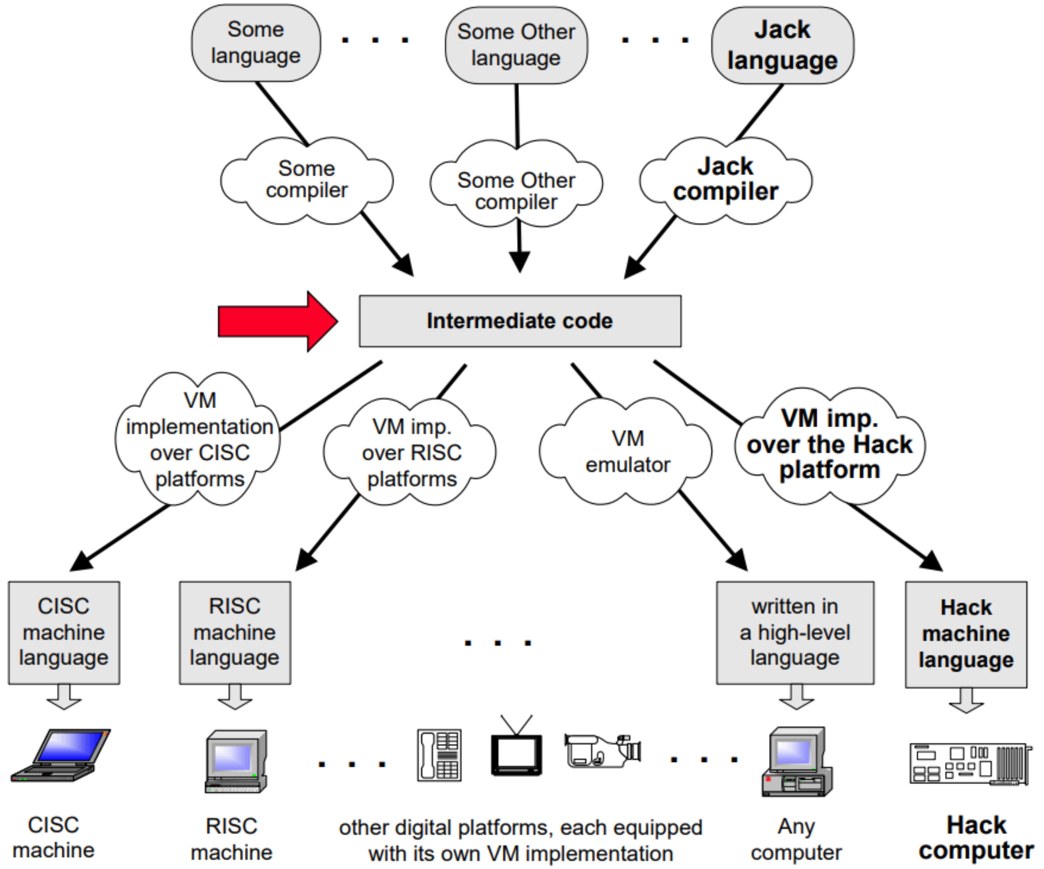
Compilación a 2 niveles



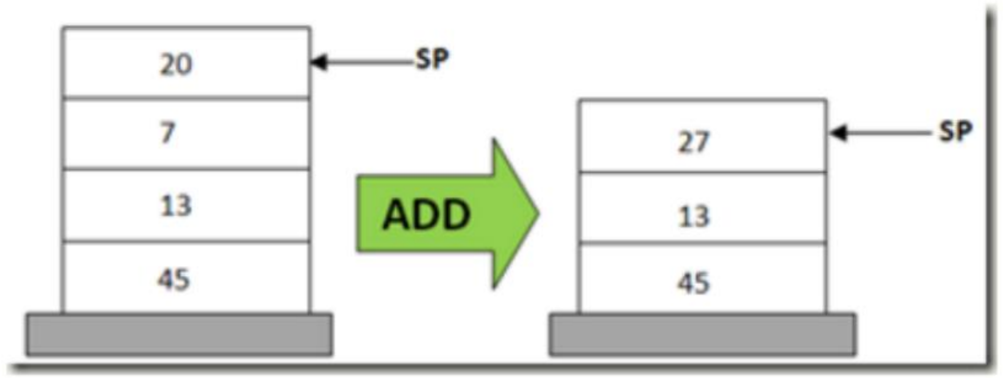
requires $n + m$ translators

Compilación a dos niveles:

- Primera etapa de compilación: depende únicamente de los detalles del lenguaje fuente
- Segunda etapa de compilación : depende únicamente en los detalles del lenguaje destino


Some language
. . .
Some Other language
. . .
Jack language
Some compiler
Some Other compiler
Jack compiler
Intermediate code
VM implementation over CISC platforms
VM imp. over RISC platforms
VM emulator
VM imp. over the Hack platform
CISC machine language
RISC machine language
. . .
written in a high-level language
Hack machine language
CISC machine
RISC machine
. . .
. . .
other digital platforms, each equipped with its own VM implementation
Any computer
Hack computer

Movimiento de data – Pila:

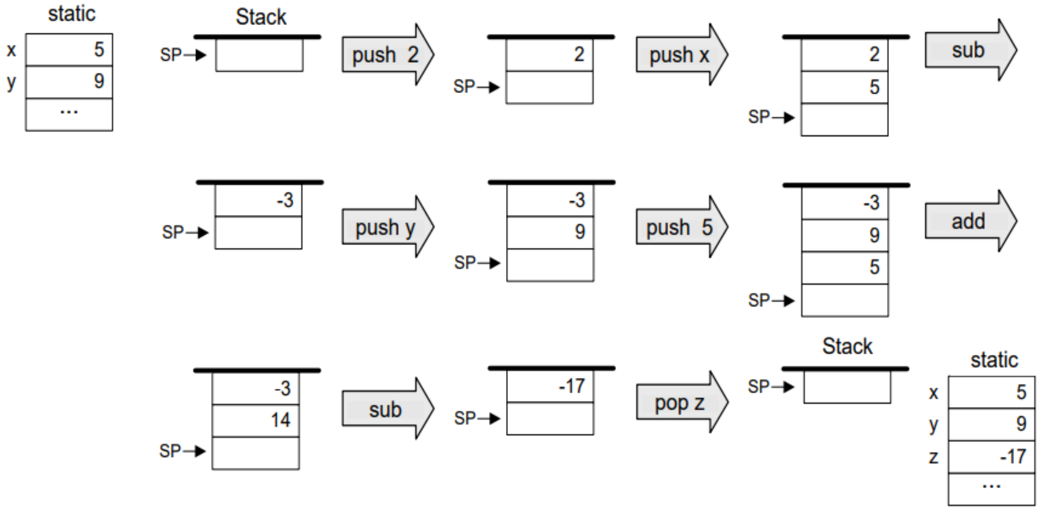


1. **POP 20**
2. **POP 7**
3. **ADD 20, 7, result**
4. **PUSH result**

Código VM

```
// z=(2-x)-(y+5)
push 2
push x
sub
push y
push 5
add
sub
pop z
```

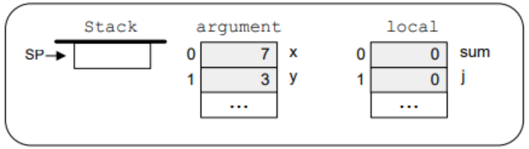
(suponer que
 x se refiere a static 0,
 y se refiere a static 1, y
 z se refiere a to static 2)



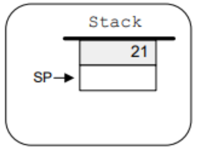
High-level code

```
function mult (x,y) {
  int result, j;
  result = 0;
  j = y;
  while ~(j = 0) {
    result = result + x;
    j = j -1;
  }
  return result;
}
```

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



VM code (first approx.)

```
function mult(x,y)
   push 0
   pop result
   push y
   pop j
label loop
   push j
   push 0
   eq
   if-goto end
   push result
   push x
   add
   pop result
   push j
   push 1
   sub
   pop j
   goto loop
label end
   push result
   return
```

VM code

```
function mult 2
  push    constant 0
  pop     local 0
  push    argument 1
  pop     local 1
label     loop
  push    local 1
  push    constant 0
  eq
  if-goto end
  push    local 0
  push    argument 0
  add
  pop     local 0
  push    local 1
  push    constant 1
  sub
  pop     local 1
  goto    loop
label     end
  push    local 0
  return
```

Principios del diseño:

- Compatibilidad
- Implementabilidad
- Programabilidad
- Usabilidad
- Eficiencia en codificación

Tipos de Set de Instrucciones

- RISC – Reduced Instruction Set Computer
- CISC – Complex Instruction Set Computer
- MISC – Minimal Instruction Set Computer
- VLIW – Very Long Instruction Word
- EPIC – Explicitly Parallel Instruction Computing
- OISC – One Instruction Set Computer
- ZISC – Zero Instruction Set Computer

- RISC – Requiere alta cantidad de RAM, siempre ejecuta una única y simple instrucción por ciclo de reloj.
- CISC – Realiza un conjunto de instrucciones.
- MISC – Set simple y minimalista, usualmente lleva a necesidades sequenciales reduciendo nivel de paralelismo.
- VLIW – Diseñado para explotar el paralelismo a nivel de instrucción. Contiene las instrucciones para ejecutarlas paralelamente.
- EPIC – Permite la ejecución de instrucciones de forma paralela directamente en el compilador. Pensado en un proceso simple sin impacto en altas frecuencias de reloj.
- OISC – Set básico de una única instrucción (operación incluída).
- ZISC – Set basado en pattern matching y comparable con neural networks.

CISC	RISC
The original microprocessor ISA	Redesigned ISA that emerged in the early 1980s
Instructions can take several clock cycles	Single-cycle instructions
Hardware-centric design – the ISA does as much as possible using hardware circuitry	Software-centric design – High-level compilers take on most of the burden of coding many software steps from the programmer
More efficient use of RAM than RISC	Heavy use of RAM (can cause bottlenecks if RAM is limited)
Complex and variable length instructions	Simple, standardized instructions
May support microcode (micro-programming where instructions are treated like small programs)	Only one layer of instructions
Large number of instructions	Small number of fixed-length instructions
Compound addressing modes	Limited addressing modes

RISC	CISC
`LOAD A, eax`	`MULT B, A`
`LOAD B, ebx`	
`PROD eax, ebx`	
`STORE ebx, A`	

- High-level language program (in C)

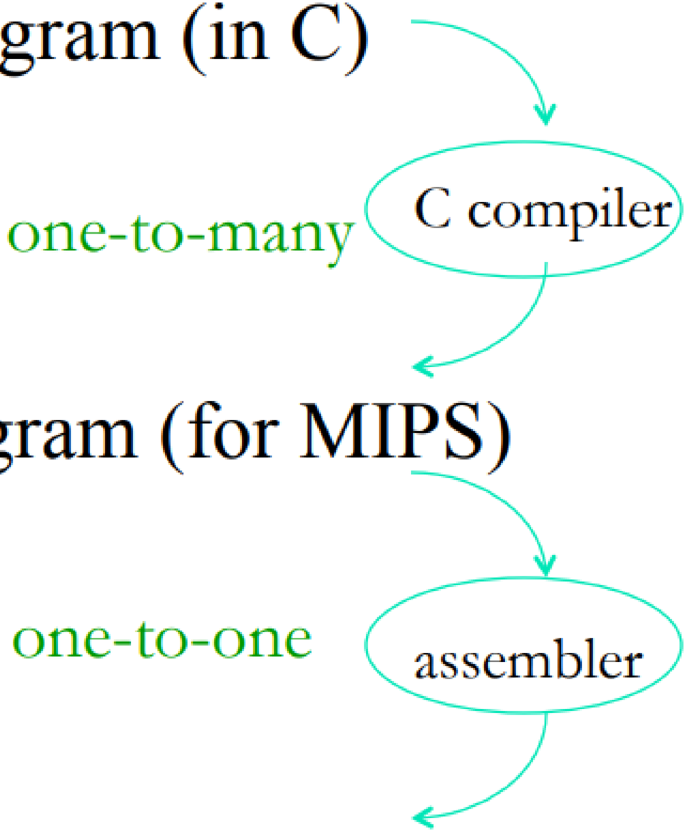
```
swap (int v[], int k)
{      int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

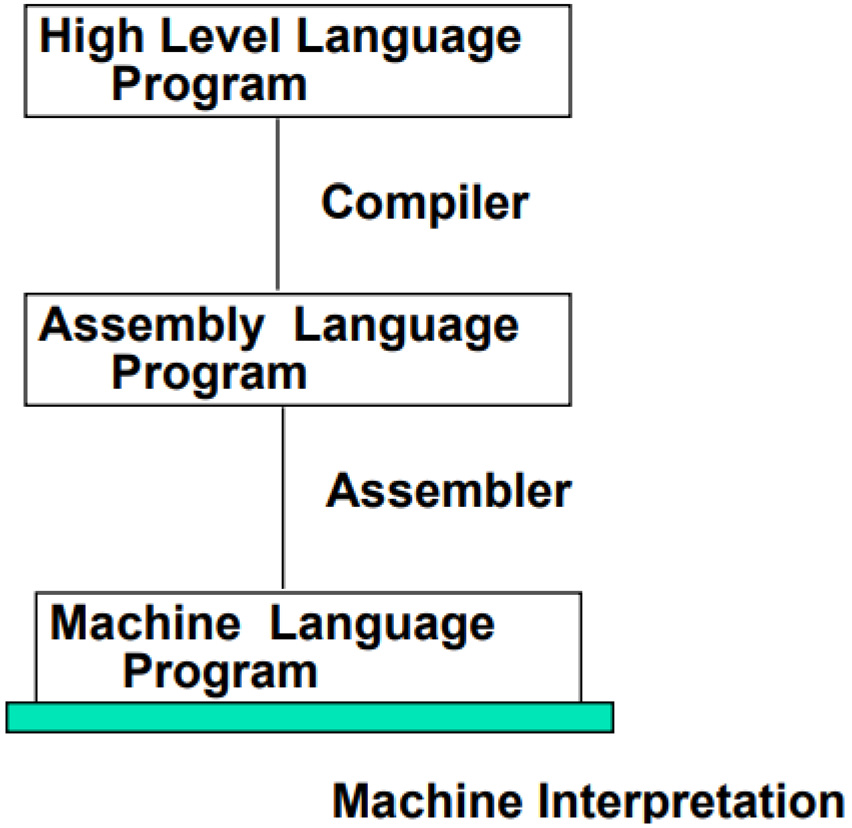
- Assembly language program (for MIPS)

```
swap:  sll    $2, $5, 2
       add    $2, $4, $2
       lw     $15, 0($2)
       lw     $16, 4($2)
       sw     $16, 0($2)
       sw     $15, 4($2)
       jr     $31
```

- Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
. . .
```





```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,   0($2)
lw  $16,   4($2)
sw  $16,   0($2)
sw  $15,   4($2)
```

```
10001100011000100000000000000000
10001100111110010000000000000100
10101100111110010000000000000000
10101100011000100000000000000100
```

*programadores que programan los programas *
programadores que programan los programas para programar

THANKS!

Any questions?