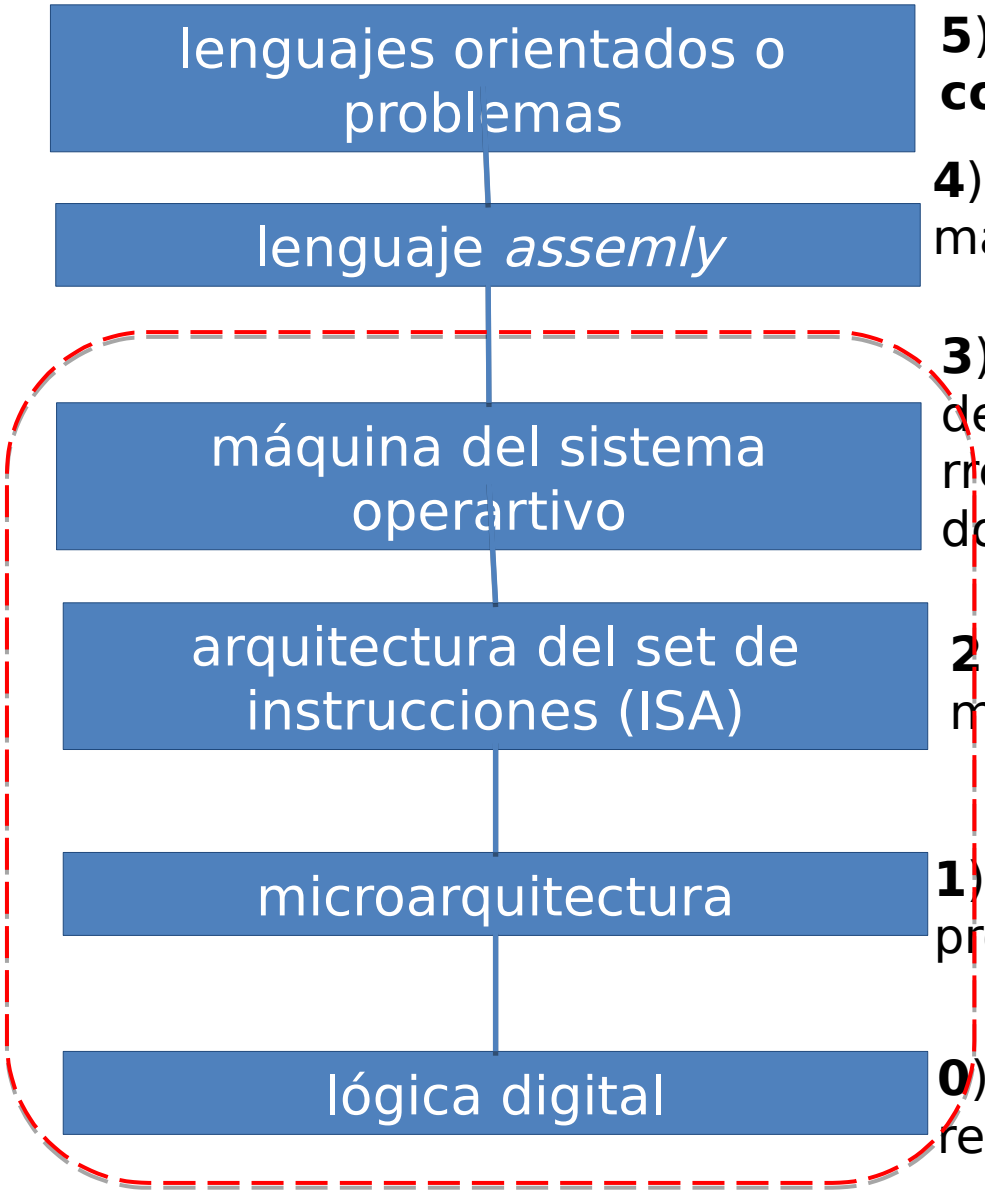
El nivel ISA
(Instruction Set Architecture)

Arquitectura de Computadores
(IIC2343)



5) lenguajes de alto nivel, traducidos por **compiladores**, o interpretados

4) forma simbólica de los lenguajes de más abajo, traducida por el **assembler**

3) instrucciones adicionales, otra organizació de memoria, capacidad de ejecución concu- rrente interpretado por un programa llama- do el sistema operativo

2) instrucciones ejecutadas por el microprograma o circuitos del hardware

1) 32 registros + **ALU** **datapath**, micro- programado o controlado por hardware

0) compuertas (gates), álgebra de Boole, registros, el motor de computación

Está ubicado entre la microarquitectura y el sistema operativo

A veces es llamado simplemente “la arquitectura” del computador

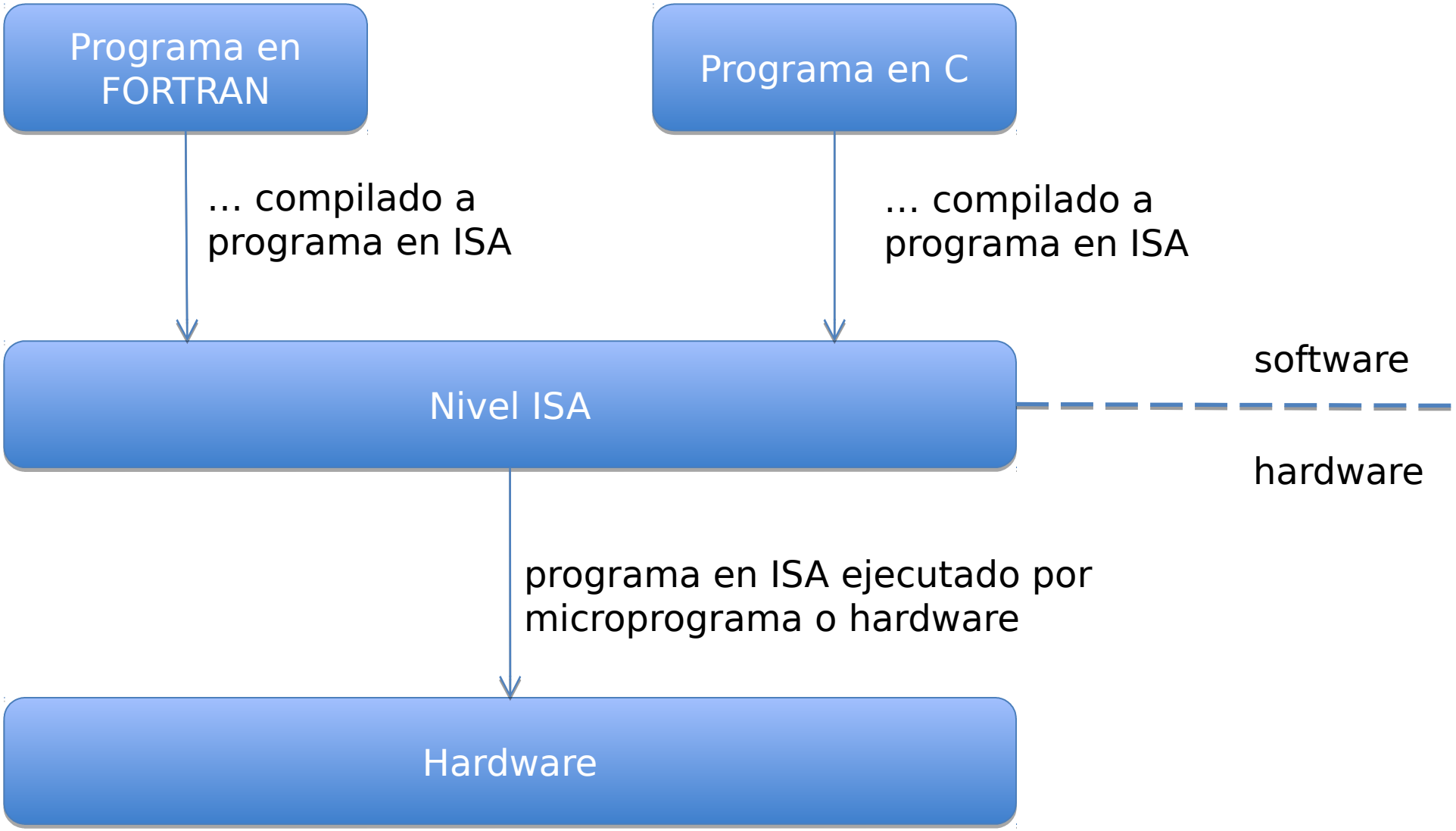
Es la interfaz entre el software (los compiladores) y el hardware:

- es el lenguaje que ambos tienen que entender

Se podría hacer que el hardware ejecutara directamente programas en C, Java o Python

… pero no sería una buena idea:

- se perdería la ventaja en cuanto a desempeño de la compilación sobre la interpretación
- para ser útiles, los computadores tienen que poder ejecutar programas escritos en múltiples lenguajes, y no solo uno


Programa en FORTRAN
Programa en C
... compilado a programa en ISA
... compilado a programa en ISA
Nivel ISA
software
hardware
programa en ISA ejecutado por microprograma o hardware
Hardware

Los programas, en diversos lenguajes de alto nivel, son traducidos al nivel ISA —una forma intermedia común

… y se diseña y construye el hardware que pueda ejecutar directa-mente programas en este nivel:

- al diseñar un nuevo computador, el arquitecto del ISA conversa tanto con los que escriben los compiladores como con los ingenieros de hardware

  … si unos quieren una instrucción que no se puede implementar eficientemente

  … o si los otros tienen un circuito con una funcionalidad para la cual no hay cómo generar código

  … entonces estas propiedades no quedan en la arquitectura

Negociaciones + simulaciones un ISA optimizado para diversos lenguajes de programación

Problema: Compatibilidad con versiones anteriores

Las dos características de un buen ISA:

- debe dejar contentos a los diseñadores de hardware

  … definiendo un conjunto de instrucciones que puede ser implementado eficientemente con las tecnologías vigentes y futuras

  … y resultando en diseños cost-effective (eficaces en relación a su costo) a lo largo de varias generaciones

- debe dejar contentos a los diseñadores de software

  … ofreciendo un lenguaje sin irregularidades para el código compilado

Propiedades generales

El nivel ISA se podría definir según cómo ve a la máquina un progra-mador de lenguaje de máquina:

- pero hoy no se programa mucho en lenguaje de máquina

Hoy lo definimos así:

el código del nivel ISA es lo que produce un compilador:

- el programador del compilador tiene que saber cuál es el modelo de memoria

  … qué registros hay

  … qué tipos de datos están disponibles

  … qué instrucciones están disponibles, etc.

Hay aspectos que no son parte del nivel ISA

… porque no son visibles para el programador del compilador:

- si la microarquitectura es microprogramada o no
- si es pipelined o no
- si es superescalar o no
- (… aunque algunas de estas propiedades afectan el desempeño, que sí es visible para el programador)

Modelos de memoria

Todos los computadores dividen la memoria en celdas, normalmente de 8 bits (byte), que tienen direcciones consecutivas:

- los caracteres ASCII son de 7 bits + bit de paridad (no muy usado)
- otros códigos usan múltiplos de 8 bits para representar caracteres

Los bytes son agrupados en palabras de 4 u 8 bytes

… y hay instrucciones para manejar palabras enteras

Muchas arquitecturas requieren que las palabras estén alineadas según su límite natural —**alineación**:

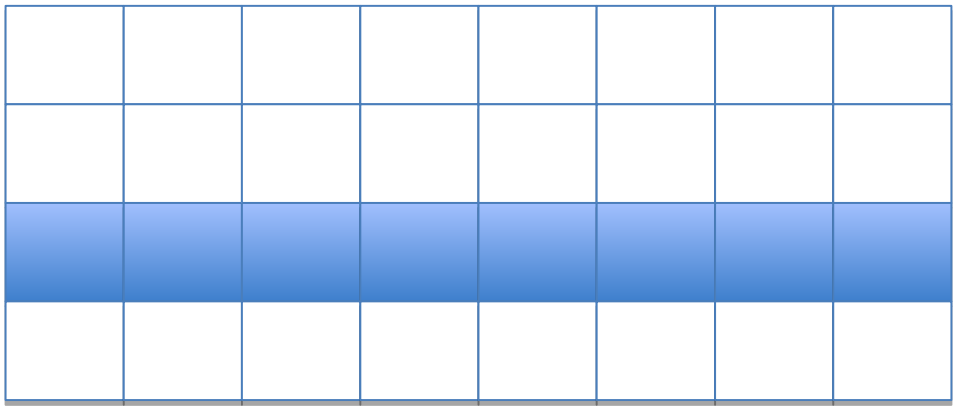
- p.ej., las de 4 bytes comienzan en las direcciones 0, 4, 8, etc.
- las memorias operan más eficientemente

  … p.ej., el Core i7 lee 8 bytes a la vez,

  … y la interfaz de memoria requiere direcciones que sean múltiplos de 8

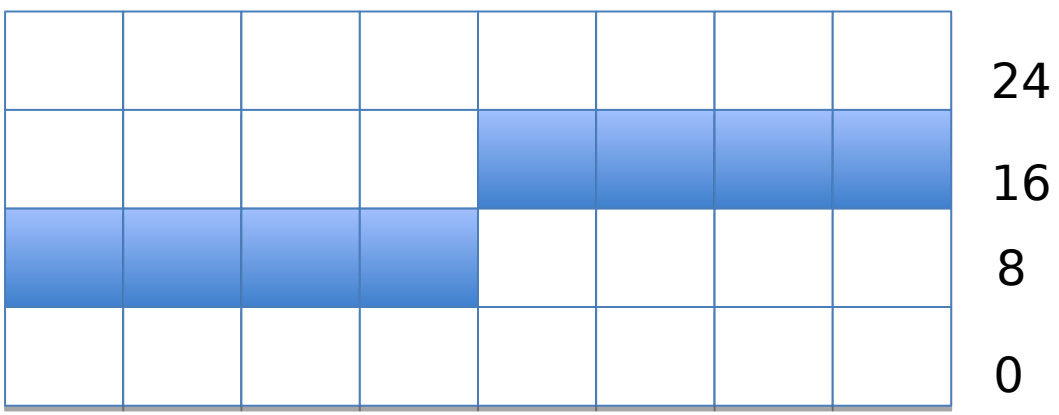
… aunque a veces esto causa problemas:

- en el Core i7, los programas pueden hacer referencia a palabras que em-piezan en cualquier dirección (en el 8088 el bus de datos era de un byte)



palabra de 8 bytes alineada
que empieza en la dirección 8:
requiere un acceso a memoria

palabra de 8 bytes no alineada
que empieza en la dirección 12:
requiere dos accesos a memoria
y luego armar la palabra



Espacio de direcciones único: 0 a 232 – 1, 0 a 264 – 1 bytes

… aunque algunas máquinas tienen espacios de direcciones separados para instrucciones y datos:

- más complejo
- permite tener 232 bytes de programa y otros 232 bytes de datos usando solo direcciones de 32 bits
- es imposible que un programa se modifique accidentalmente a sí mismo —todas las escrituras van automáticamente al espacio para datos
- malware no puede cambiar un programa, ni siquiera puede direccionarlo

(Espacios de direcciones separados para instrucciones y datos no es lo mismo que una cache de nivel 1 *split*)

Los registros

Registros visibles al nivel ISA:

- controlan la ejecución del programa
- almacenan resultados temporales

Los hay de propósito especial:

- *program counter*, *stack pointer*

... y de propósito general:

- variables locales, resultados intermedios

(Hay registros que no son visibles al nivel ISA)

Los registros proporcionan acceso rápido a datos muy usados (evitando accesos a memoria)

En algunos computadores, los registros de propósito general son totalmente intercambiables

… en otros, algunos de estos registros son un poco especiales:

- p.ej., EDX en el Core i7 es un registro general, pero que también recibe la mitad del producto en una multiplicación y almacena la mitad del dividendo en una división

Incluso cuando los registros generales son intercambiables,

… el sistema operativo o los compiladores pueden usarlos de manera especial:

- p.ej., algunos registros pueden almacenar parámetros para una función, otros pueden ser usados como almacenamiento temporal

Hay registros de propósito especial disponibles solo para el sistema operativo —ni los compiladores ni los usuarios tienen que conocerlos:

- controlan las caches, la memoria, los dispositivos de I/O, y otros aspectos del hardware

PSW (*program status word*) es un registro de control que almacena varios bits requeridos por la CPU

… p.ej., los códigos de condición, que son actualizados en cada ciclo de la ALU para reflejar el estado del resultado de la operación más reciente:

- **N** (*negative*), **Z** (*zero*), **V** (*overflow*), **C** (*carry*), **A** (*auxiliary carry*), **P** (*even parity*)
- son usados por las instrucciones de comparación y de salto condicional

Los 16 registros del Core i7

EAX, **EBX**, **ECX** y **EDX**: 32 bits, más o menos generales:

- EAX es el principal registro aritmético
- EBX almacena punteros (direcciones de memoria)
- ECX se usa en los *loops*
- EDX se usa en multiplicaciones y divisiones, para almacenar productos y dividendos de 64 bits junto a EAX
- cada uno contiene un registro de 16 bits (AX, BX, CX y DX) e incluso uno de 8 bits (AL, BL, CL y DL)

ESI, **EDI**, **EBP** y **ESP**: 32 bits, más o menos generales:

- ESI y EDI almacenan punteros para manejo de strings
- EBP típicamente apunta a la base del registro de activación vigente
- ESP es el *stack pointer*

CS, SS, DS, ES, FS y GS: 16 bits, pueden ser ignorados al usar un único espacio de direcciones de 32 bits (recuerdos del 8088)

EIP: 32 bits, *program counter*

EFLAGS: 32 bits, PSW

Las instrucciones

Es el componente principal del nivel ISA :

- controlan lo que el computador puede hacer
- siempre hay instrucciones del tipo *LOAD* y *STORE* para mover datos entre la memoria y los registros

  … e instrucciones del tipo *MOVE* para copiar datos entre registros
- siempre hay instrucciones arirméticas y lógicas

  … e instrucciones para comparar datos y “saltar” a una determinada instrucción dependiendo de los resultados

IA-32, de Intel, como está implementada en el procesador Core i7

Soporte completo para ejecutar programas escritos para los procesa-dores 8086 y 8088 (que tenían el mismo ISA):

- tenían el mismo ISA
- eran computadores de 16 bits (solo que el 8088 tenía un bus de 8 bits)

Su sucesor fue el 80286, también de 16 bits, pero con un espacio de direcciones más grande

El 80386 fue el primer computador de 32 bits de Intel

Todos los procesadores que siguieron (el 80486, la familia Pentium, Core 2 duo, y hasta el Core i7) tienen esencialmente la misma arquitectura de 32 bits —llamada **IA-32**:

- los cambios arquitectónicos posteriores principales han sido la adición de instrucciones especializadas para mejorar el desempeño en aplicaciones multimediales

  … y la versión de 64 bits (x86-64), en realidad introducida por AMD

El Core i7 tiene tres modos de operación:

- *real*: todas las propiedades agregadas después del 8088 son desabilitadas y se comporta como un 8088
- *virtual 8086*: el sistema operativo crea un ambiente aislado especial que se comporta como un 8088
- *protegido*: el Core i7 propiamente tal, con 4 niveles de privilegio contro-lados por bits en el PSW (nivel 0 = modo kernel, usado por el sistema operativo; nivel 3 = programas de usuarios)

El espacio de direcciones está organizado en 16,384 segmentos, cada uno con direcciones de 0 a $2^{32} - 1$

… aunque la mayoría de los sistemas operativos (p.ej., UNIX y Windows) manejan solo un segmento:

- las aplicaciones ven un espacio de direcciones lineal de 232 bytes

Cada byte tiene su propia dirección

… y las palabras tienen 32 bits en formato *little endian*:

- el byte de más a la derecha (*low-order*) tiene la dirección numéricamente menor

Los registros del Core i7 aparecen en la diap. #19

Address	Value
00	0x0F
01	0xA7
10	0x1D
11	0x23

int 32 bits *big endian*: 0x 0F A7 1D 23

int 32 bits ***little endian***: 0x 23 1D A7 0F

Tipos de datos

Un tema clave es si hay soporte de hardware para un tipo de datos en particular:

- si hay instrucciones que esperan datos en un formato particular

  … entonces el usuario no es libre de elegir otro formato

- p.ej., los números enteros con signo exigen que el signo sea el bit más significativo

Pueden ser divididos en dos categorías:

- numéricos
- no numéricos

El principal tipo de datos numérico son los números enteros:

- 8, 16, 32 y 64 bits
- sin signo, con signo
- todas las combinaciones están en el Core i7

… también se usan números de punto flotante:

- 32, 64 y 128 bits
- muchos computadores tienen registros separados para operandos enteros y para operandos de punto flotante
- el Core i7 implementa el estándar IEEE 754, en 32 y 64 bits

[Algunos lenguajes de programación —p.ej., COBOL— manejan núme-ros decimales:

- dos dígitos decimales por byte, de 4 bits cada uno (¿es eficiente?) **]**

Los computadores manejan mucha información no numérica:

- e-mail, Web, fotos digitales, multimedia

P.ej., los **caracteres**:

- ASCII, de 7 bits (más 1 bit de paridad)
- Unicode, de 16 bits —codificación universal de los alfabetos de la mayoría de los lenguajes humanos; usado por Java

El nivel ISA suele tener instrucciones especiales para strings:

- copia, búsqueda, edición, etc.

P.ej., los **valores Boolean**:

- 0 es falso; todo lo demás, verdadero
- bastaría con un bit, pero se usa un byte o una palabra (los bits no tienen dirección propia)
- … excepto cuando hay un arreglo de valores Boolean, o *bit-map* (p.ej., para seguirle la pista a los bloques del disco)

P.ej., los **punteros**, es decir, las direcciones de memoria:

- *stack pointer* y *program counter* (o *instruction pointer*) son punteros
- tener acceso a una variable a una distancia fija —un desplazamiento— de un puntero —una dirección base— es muy común en todos los computadores
- útiles, pero también son la fuente de muchos errores de programación con graves consecuencias —hay que usarlos con mucho cuidado

El formato de las instrucciones

Una instrucción consiste en un *opcode* :

- qué hace la instrucción

… e información adicional:

- p.ej., de dónde vienen los operandos o a dónde va el resultado (una, dos o tres direcciones —*direccionamiento*)

Todas las instrucciones podrían ser del mismo largo:

- más simple y facilita la decodificación, pero desperdicia espacio

En general, el largo de las instrucciones varía:

- pueden ser del mismo largo de una palabra

  … de la mitad o de un cuarto del largo de una palabra (caben dos o cuatro instrucciones en una palabra)

  … o pueden ocupar dos palabras

Es una decisión que debe tomarse al principio del diseño de un nuevo computador

¡Cuidado! Si el computador es exitoso, el set de instrucciones puede permanecer vigente por muchos años, en cuyo caso:

- puede ser necesario agregar instrucciones nuevas
- puede cambiar la tecnología, de la que depende la eficiencia del set de instrucciones

  … y después de un tiempo, algunas decisiones de diseño pueden verse como malas decisiones

Por supuesto, el corto plazo también importa:

- si este ISA tan bien diseñado es un poco más caro que sus competidores, la compañía puede no sobrevivir lo suficiente como para que el mundo aprecie tal elegancia

En general, instrucciones más cortas (menos bits) son mejores que instrucciones más largas:

- *n* instrucciones de 16 bits ocupan la mitad de espacio que *n* instrucciones de 32 bits
- el ancho de banda de la memoria no ha progresado lo mismo que la velocidad de los procesadores —las memorias (y las caches) no son capaces de entregar instrucciones y operandos tan rápidamente como el procesador los puede usar

Por otra parte, el formato de la instrucción debe tener suficiente espacio como para expresar todas las operaciones necesarias:

- *n* bits en el *opcode* significa un máximo de 2^n operaciones
- … y siempre hay que considerar la necesidad de no usar todos los *opcodes* inicialmente para poder agregar instrucciones en el futuro

Otro factor es el número de bits en las direcciones especificadas en la instrucción

Supongamos una memoria de 2^{32} bytes

... la unidad de direccionamiento podría ser el byte (de 8 bits):

- memoria de 2^{32} bytes, numerados 0 al 4,295 millones
- cada dirección necesita 32 bits
- para comparar dos caracteres, simplemente los lee

… o, en el otro extremo, la palabra (de 32 bits):

- memoria de 2^{30} palabras, numeradas 0 al 1,074 millones
- cada dirección necesita solo 30 bits instrucciones más cortas y más rápidas de leer

  … o bien puede direccionar 16 GB de memoria
- para comparar dos caracteres tiene que leer dos palabras y extraer los caracteres (de 8 bits) de cada una toma instrucciones adicionales

Otra posibilidad son *opcodes* expandibles —supongamos instrucciones de 16 bits y 16 registros, en que se necesitan instrucciones con 0, 1, 2 o 3 direcciones (de registros)

opcode + 3 direcciones: 0000 xxxx yyyy zzzz 15 instrucciones
(4 bits) ... de 3 direcciones
1110 xxxx yyyy zzzz

opcode + 2 direcciones: 1111 0000 yyyy zzzz 14 instrucciones
(8 bits) ... de 2 direcciones
1111 1101 yyyy zzzz

opcode + dirección: 1111 1110 0000 zzzz 31 instrucciones
(12 bits) ... de una dirección
1111 1111 1110 zzzz

solo *opcode*: 1111 1111 1111 0000 16 instrucciones
(16 bits) ... sin dirección
1111 1111 1111 1111

Direccionamiento

La mayoría de las instrucciones tienen operandos

Direccionamiento (*addressing*) es cómo especificar dónde están los operandos

Cómo se interpretan los bits de una dirección para encontrar el operando —los **modos de dirección**

Direccionamiento **inmediato**

La parte de la instrucción en que (normalmente) se especifica una dirección en este caso contiene al operando propiamente tal (en lugar de su dirección):

- el operando es automáticamente traido de la memoria al mismo tiempo que la instrucción

  … por lo que está inmediatamente disponible para ser usado

  … y no requiere una referencia adicional a la memoria
- p.ej., **`MOV R1, #4`** carga el registro R1 con el valor 4
- ¿desventajas?

Direccionamiento **directo**

Se especifica explícitamente la dirección de memoria del operando:

- la instrucción siempre va a tener acceso a la misma dirección de memoria —puede cambiar el valor almacenado en la dirección, pero no la dirección
- solo para tener acceso a variables globales cuya dirección es conocida al momento de compilar
- ¿es útil?

Direccionamiento **por registro** (*modo registro*)

Conceptualmente igual a direccionamiento directo,

… pero especifica explícitamente un registro en lugar de una ubicación de memoria:

- modo de direccionamiento más común
- las variables que van a ser usadas más a menudo (p.ej., el índice de un *loop*) van en los registros

P.ej., en la arquitectura ARM, (casi) todas las instrucciones usan solo este modo:

- el “casi” es debido a las instrucciones que llevan un dato de la memoria a un registro (**`LDR`**), o viceversa (**`STR`**)

Direccionamiento **indirecto por registro**

El operando viene de o va a la memoria

… pero la dirección de memoria correspondiente está contenida en un registro —este registro es un **puntero**

… y es el registro el que se especifica en la instrucción

Ventajas:

- puede hacer referencia a la memoria sin tener que especificar explícita-mente una dirección de memoria —p.ej., de 32 bits— en la instrucción
- puede hacer referencia a diferentes direcciones de memoria en diferentes ejecuciones de la instrucción

P.ej., un programa que calcula en el registro R1 la suma de los 1024 elementos de un arreglo *A* de enteros de 4 bytes c/u, cuyo primer elemento es apuntado por el registro R2

El registro R3 apunta a la primera dirección después del arreglo, es decir, *A* + 4096

```
      MOV R1, #0
      MOV R2, #A
      MOV R3, #A+4096
LOOP: ADD R1, (R2)
      ADD R2, #4
      CMP R2, R3
      BLT LOOP
```

El programa emplea varios modos de direccionamiento:

- en las tres primeras instrucciones, modos registro (primer operando) e inmediato (el segundo operando es una constante, señalada por #)
- **MOV R2, #*A*** coloca en R2 la dirección de *A*, no el contenido

  … similarmente, **MOV R3, #*A*+4096**

El cuerpo del *loop* no contiene ninguna dirección de memoria:

ADD R1,(R2) usa modos registro e indirecto por registro

ADD R2,#4 usa modos registro e inmediato

CMP R2,R3 usa modo registro dos veces

¿Cómo se especifica en la instrucción **BLT** la dirección a la cual saltar (LOOP)?

- dirección de memoria, u …
- *offset* de 8 bits relativo a la propia instrucción

Direccionamiento **indexado**

Direccionamiento de la memoria especificando un registro (explícito o implícito) y un *offset* constante

P.ej., supongamos los arreglos A y B de 1024 palabras c/u

… queremos calcular A_i AND B_i para todos los pares y luego el OR de estos 1024 productos booleanos juntos:

- podríamos colocar la dirección de A en un registro, la de B en otro y luego avanzar a través de ellos (modo *indexado con base*: ya viene)

… pero también …

```
    MOV R1,#0     ; el OR acumulado
    MOV R2,#0     ; el índice i
    MOV R3,#4096          ; primer valor de i que no se usa
LOOP: MOV R4,A(R2)        ; R4 = A[i]
    AND R4,B(R2)          ; R4 = A[i] AND B[i]
    OR R1,R4      ; el OR de los productos boolean
    ADD R2,#4     ; i = i + 4 (4 bytes = una palabra)
    CMP R2,R3     ; ¿listo?
    BLT LOOP      ; si R2 < R3, repetimos
```

El cálculo de *A*(R2) y *B*(R2) usa modo indexado:

- un registro —R2— y un *offset* constante —la dirección de *A* o *B*— se suman entre ellos y el resultado se usa para hacer referencia a la memoria
- requiere poder almacenar en la instrucción una dirección de memoria
- la suma no es almacenada en ningún registro visible para el usuario

Direccionamiento **indexado con base**

La dirección de memoria es calculada sumando dos registros (más un *offset* opcional):

- uno de los registros es la base
- el otro es el índice

En el ej. anterior, si ponemos la dirección de *A* en R5 y la de *B* en R6, las instrucciones en LOOP y siguiente podrían ser

```
LOOP: MOV R4,(R2+R5)
      AND R4,(R2+R6)
```

Finalmente, …

Las instrucciones de tipo *jump* y las llamadas a funciones también necesitan especificar la dirección objetivo

Los modos vistos sirven para estos fines:

- modo directo: incluimos la dirección explícitamente en la instrucción
- indirecto por registro: el programa, en tiempo de ejecución, calcula la dirección objetivo, la pone en un registro, y luego salta hasta allá

  … más flexibilidad, pero también muchas oportunidades para producir *bugs* casi imposibles de encontrar
- indexado: usa un *offset* conocido desde un registro; similar al anterior
- indexado relativo al PC: el *offset*, con signo e incluido en la instrucción, se suma al PC

El stack se usa también para almacenar variables locales de la función que no caben en los registros, p.ej., arreglos

La porción del stack que contiene los registros guardados y las variables locales de una función se conoce como **marco de activación**

Un (nuevo) registro —*frame pointer* (**`$fp`**, o el registro 30, en MIPS)— apunta a la primera palabra del marco de una función (normalmente, un registro con el valor de un parámetro):

- el *stack pointer* puede cambiar durante la ejecución de una función ⇒ las referencias a una variable local podrían tener diferentes *offsets*, depen-diendo de dónde están en el marco
- el *frame pointer* tiene un valor fijo, conveniente para las referencias a memoria locales
- al hacerse una llamada, el *frame pointer* recibe el contenido del *stack pointer*

  … y al volver de la llamada, el *stack pointer* es actualizado con el valor del *frame pointer*

Modos de direccionamiento del Core i7

Son irregulares

… y difieren dependiendo de si la instrucción ocupa 16, 32 o 64 bits

Modos en el formato de 32 bits:

- inmediato
- directo
- por registro
- indirecto por registro
- indexado
- modo especial para direccionar elementos de arreglos

No todos los modos aplican a todas las instrucciones

… ni todos los registros pueden usarse en todos los modos:

- escribir un compilador se hace difícil
- el compilador produce peor código

Las instrucciones tienen 0 a 5 bytes como prefijo, uno o dos para el *opcode*, 0 a 4 para el desplazamiento, y 0 a 4 para un operando de tipo *inmediate*

Adermás, tienen un byte MODE que controla los modos de direccion-amiento, con tres campos:

- MOD, 2 bits
- REG, 3 bits
- R/M, 3 bits

Un operando se especifica combinando MOD y R/M

… el otro es siempre un registro: REG

Las 32 (= 2^5) combinaciones de MOD y R/M:

- MOD = 00 es una dirección de memoria almacenada en un registro
- MOD = 01 es una dirección basada en un registro más un *offset* de 8 bits
- MOD = 10 es una dirección basada en un registro más un *offset* de 32 bits
- MOD = 11 permite elegir entre dos registros, dependiendo de si la instruc-ción es una palabra o un byte

R/M = 100 ⇒ modo SIB (*scale*, *index*, *base*)

A través del byte SIB de la instrucción, especifica un factor de escala y dos registros:

- la dirección del operando se calcula multiplicando el registro índice por 1, 2, 4 u 8 (dependiendo de la escala)

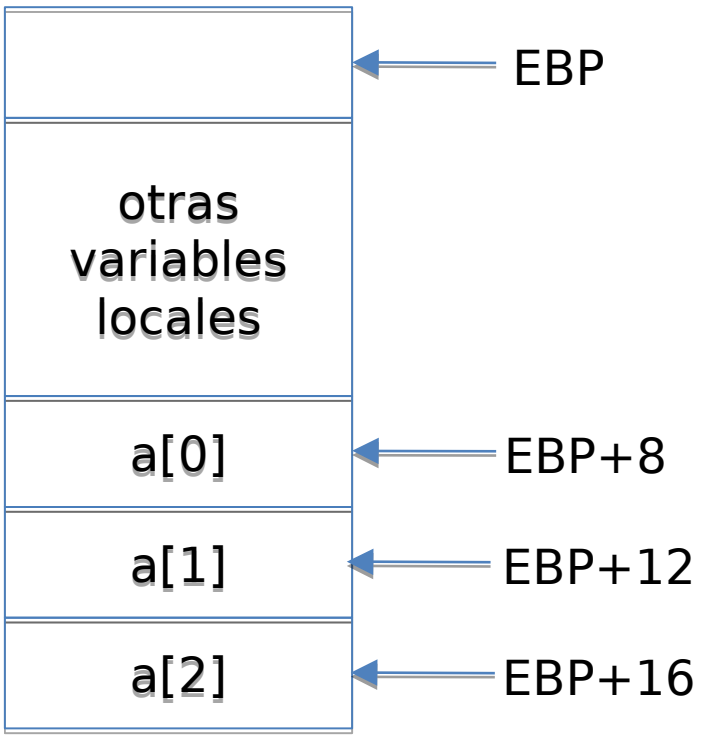
  … a lo que se suma el registro base

  … y un desplazamiento de 8 o 32 bits (MOD = 01 o 10)
- todos los registros pueden ser usados como índice o base

P.ej., supongamos la sentencia

```
for (i = 0; i < n; i++)
a[i] = 0
```

- **a** es un arreglo de enteros de 4 bytes local a la función vigente
- EBP apunta a la base del *stack frame* (o marco de activación)
- **i** podría estar en EAX
- para tener acceso a **a[i]** usa el modo SIB, en que la dirección del operando es la suma de 4 × EAX, EBP y 8
- esta instrucción puede hacer una asignación a **a[i]** en una instrucción



Para tener acceso a **a[0]**, la referencia en modo SIB es M[4 * EAX + EBP + 8]

Tipos de instrucciones

Movimiento de datos (mejor dicho, duplicación de datos):

copiar un dato de un lugar a otro —en realidad, crear un nuevo objeto con un patrón de bits idéntico al original— es la operación más fundamental

- debido a asignaciones en el programa del usuario
- debido a la necesidad de llevar datos de la memoria a los registros (*load*) o vice versa (*store*), o copiar entre registros (*move*), ya que muchas instrucciones de máquina pueden tener acceso a variables solo cuando estas están en los registros

Operaciones diádicas:

combinar dos operandos para producir un resultado

- suma, resta, multiplicación y división de números enteros
- (algunas de las 16) funciones booleanas de dos variables —*and*, *or*, *not*, *exclusive or*, *nor*, nand

  p.ej., para extraer el segundo carácter de una palabra *A* de 4 bytes y dejarlo como el único carácter en otra palabra —el de más a la derecha, con puros ceros en los otros bytes:

  10110111 10111100 11011011 10001011 A
  <u>00000000 11111111 00000000 00000000</u> B (*mask*)
  00000000 10111100 00000000 00000000 A AND B

  … y este resultado lo desplazamos (*shift*) 16 bits a la derecha

- operaciones de punto flotante

Operaciones monódicas:

tienen un operando y producen un resultado

- *shift* a la derecha o a la izquierda —*shift* a la derecha puede mantener el signo o no

  shift a la izquierda (*left shift*) de k bits ⇒ multiplicación por 2^k

  shift a la derecha (*right shift*) de k bits ⇒ división por 2^k (excepto en el caso de números negativos)

- *rotate* (*shift* en que los bits que salen por un extremo reaparecen en el otro)

  p.ej., para chequear cada bit de una palabra, poniéndolos uno por uno en la posición del bit de signo, y restaurando el valor de la palabra al final

- operaciones diádicas en que un operando particular es muy común

  p.ej., asignar 0 (**CLR** *address*) o sumar 1 (**INC** *address*) a una variable

Ejemplo del lenguaje *assembly* del Core i7

Función para resolver el problema de las **Torres de Hanoi**

```
towers(int n, int i, int j):
   int k
   if (n == 1):
      print(“Mover disco de” i “a” j)
   else:
      k = 6-i-j
      towers(n-1, i, k)
      towers(1, i, j)
      towers(n-1, k, j)
```

Hacemos la llamada **towers(3, 1, 3)**

Las Torres de Hanoi en lenguaje (*assembly*) del Core i7:

- EBP □ *frame pointer*
- las dos primeras palabras se usan para vínculos
- primer parámetro *n* está en EBP+8
- … seguido de los parámetros *i* y *j*, en EBP+12 y EBP+16
- variable local *k* en EBP+20

Al hacerse una llamada, se arma un nuevo *frame* al final del antiguo:

- copia ESP a EBP
- compara *n* con 1
- si *n* > 1, va al *else*
- si *n* = 1, coloca en el stack: la dirección del *string format*, *i*, *j*

```
.CODE
towers:  PUSH EBP                    ; guarda EBP y decrementa ESP
         MOV EBP,ESP                 ; nuevo EBP encima de ESP
         CMP [EBP+8],1                   ; if (n == 1)
         JNE L1                          ; jump a L1 si n no es 1
         MOV EAX,[EBP+16]                ; EAX = j
         PUSH EAX                    ; push j
         MOV EAX,[EBP+12]                ; EAX = i
         PUSH EAX                    ; push i
         PUSH OFFSET FLAT:format ; push dirección de format
         CALL print
         ADD ESP,12
         JMP Done

L1:          ...
```

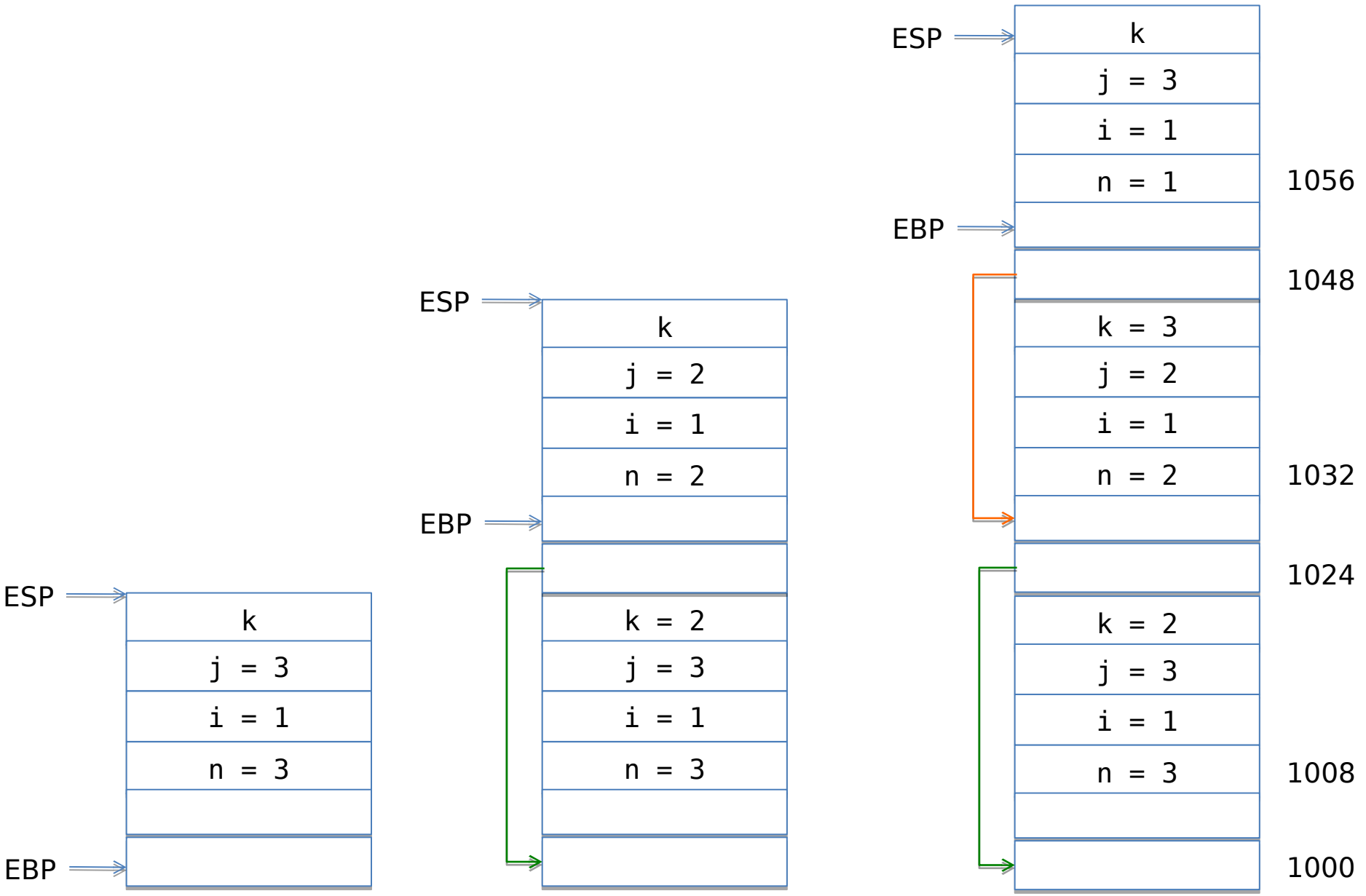
ver próximas diaps.

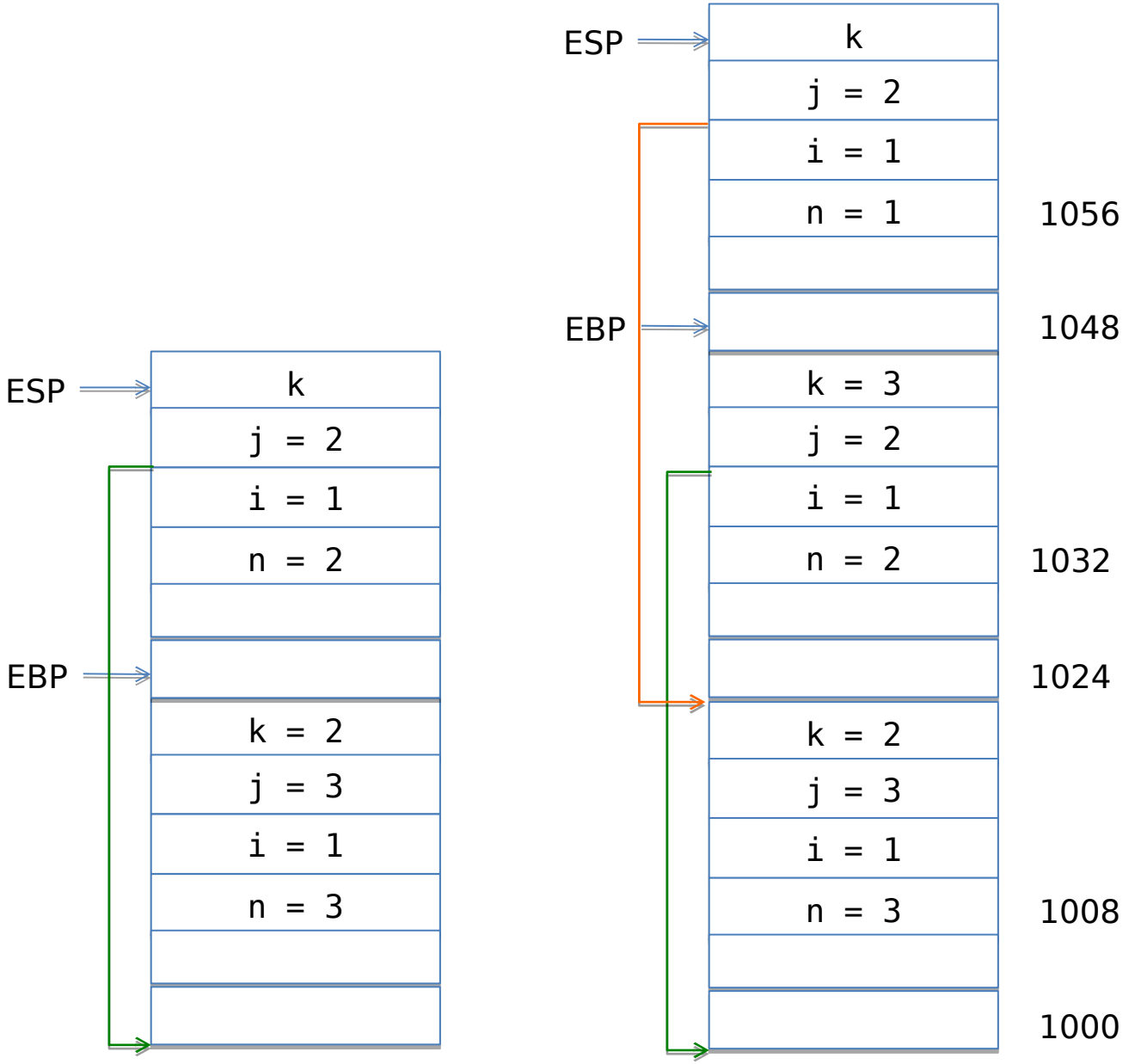
```
Done:    LEAVE                       ; prepararse para salir
         RET 0                       ; volver
```

```
L1:     MOV EAX,6                   ; empieza k = 6 – i – j
        SUB EAX,[EBP+12]            ; EAX = 6 – i
        SUB EAX,[EBP+16]            ; EAX = 6 – i – j
        MOV [EBP+20],EAX            ; k = EAX
        PUSH EAX                  ; empieza towers(n–1, i, k)
        MOV EAX,[EBP+12]            ; EAX = i
        PUSH EAX                  ; push i
        MOV EAX,[EBP+8]             ; EAX = n
        DEC EAX                   ; EAX = n – 1
        PUSH EAX                  ; push n–1
        CALL towers               ; towers(n–1, i, 6–i–j)
        ADD ESP,12                ; sacar parametros del stack
        MOV EAX,[EBP+16]            ; empieza towers(1, i, j)
        PUSH EAX
        MOV EAX,[EBP+12]
        PUSH EAX
        PUSH 1
        CALL towers               ; towers(1, i, j)
        . . .
```

```
        . . .
        ADD ESP, 12
        MOV EAX,[EBP+12]          ; empieza towers(n–1, 6–i–j, i)
        PUSH EAX                  ; push i
        MOV EAX,[EBP+20]          ; EAX = k
        PUSH EAX                  ; push k
        MOV EAX,[EBP+8]           ; EAX = n
        DEC EAX                   ; EAX = n–1
        PUSH EAX                  ; push n–1
        CALL towers               ; towers(n–1, 6–i–j, i)
        ADD ESP,12                ; ajustar stack pointer

DATA
format   DB “Mover disco de % a %”     ; string de formato
END
```


ESP
k
j = 3
i = 1
n = 3
EBP
ESP
k
j = 2
i = 1
n = 2
EBP
k = 2
j = 3
i = 1
n = 3
ESP
k
j = 3
i = 1
n = 1
1056
EBP
1048
k = 3
j = 2
i = 1
n = 2
1032
1024
k = 2
j = 3
i = 1
n = 3
1008
1000


ESP
k
j = 2
i = 1
n = 2
EBP
k = 2
j = 3
i = 1
n = 3
ESP
k
j = 2
i = 1
n = 1
1056
EBP
1048
k = 3
j = 2
i = 1
n = 2
1032
1024
k = 2
j = 3
i = 1
n = 3
1008
1000

Problemas con el ISA IA-32: Antiguo, con las propiedades equivocadas para la tecnología actual

instrucciones de largo variable y en muchísimos formatos —difíciles de decodificar rápidamente sobre la marcha

es de dos direcciones orientado a memoria —la mayoría de las instrucciones hacen referencia a la memoria

pocos registros e irregulares —muchos resultados intermedios deben ir a memoria, produciendo más referencias a memoria que las necesarias