Organización de computadoras - Teóricos

Resumen 2

Conceptos básicos

La palabra "informática" viene de "información automática". Y se eligió esa idea para definirla, básicamente porque la informática es —entre otras cosas— la ciencia que realiza un tratamiento racional y automático de la información.

Otra definición muy conocida dice que la informática es la ciencia que analiza y resuelve problemas utilizando la computadora.

¿Y qué es una computadora? Básicamente un aparato compuesto de varios componentes electrónicos (esto viene a ser el *hardware*), capaz de ejecutar instrucciones, las cuales pueden combinarse para formar programas (y eso sería el *software*).

Dicho de otra manera, **una computadora es una máquina digital sincrónica**[1], capaz de realizar cálculos numéricos y lógicos, que se controla mediante programas y que tiene una comunicación con el mundo exterior.

Ahora bien, cuando se describe una computadora, hay una distinción clave. Una distinción que de hecho dió lugar a las dos materias del primer año. Se trata de la distinción entre arquitectura y organización de un computador.

La **arquitectura de computadores** se refiere a los atributos de un sistema que son visibles para un programador, es decir, a aquellos atributos que tienen un impacto directo en la ejecución lógica de un programa. La **organización de computadores** se refiere a las unidades funcionales y sus interconexiones, que dan lugar a especificaciones arquitectónicas.

- Entre los ejemplos de atributos arquitectónicos se encuentran el conjunto de instrucciones, el número de bits usados para representar varios tipos de datos (por ejemplo, números y caracteres), los mecanismos de E/S y las técnicas para direccionamiento de memoria.
- Entre los atributos de organización se incluyen aquellos detalles de hardware transparentes al programador, tales como señales de control, interfaces entre el computador y los periféricos y la tecnología de memoria usada.

[1] En el contexto de computadoras, una **máquina** es un dispositivo que procesa información de manera automática según instrucciones predefinidas. **Digital**, significa que la computadora trabaja con valores discretos y finitos, específicamente con el sistema binario (0 y 1), o sea, con bits. Y **sincrónica** significa que todas las operaciones de la computadora están coordinadas por una señal de reloj común (el famoso *clock*).

Para poner un ejemplo, una cuestión de diseño arquitectónico es si el computador tendrá la instrucción de multiplicar. Una cuestión de organización es si esa instrucción será implementada por una unidad especializada en multiplicar, o por un mecanismo que haga un uso iterativo de la unidad de suma del sistema.

La distinción no es sólo teórica, también podemos verla en el mercado, ya que muchos fabricantes de computadores ofrecen una familia de modelos, todos con la misma arquitectura, pero con diferencias en cuanto a la organización. Consecuentemente, los diferentes modelos de la familia tienen precios y prestaciones distintas. Más aún, una arquitectura puede sobrevivir muchos años, pero su organización cambia con la evolución de la tecnología. Un ejemplo clásico de ambos fenómenos es la arquitectura IBM Sistema/370, y otro un poco más actual y conocido es la arquitectura x86 de Intel.

La ventaja de mantener la misma arquitectura, es mantener la compatibilidad del software. Como antes los programas podían demandar mucho dinero o tiempo, era bueno saber que uno podía comprarse una computadora más mejor (de la misma familia) sin perder los programas que ya había conseguido (solamente tenía que volver a instalarlos en la nueva compu).

PARÉNTESIS: las opiniones de Claude

Pregunta: si tuvieras que dar ejemplos de cosas que determinan la ORGANIZACIÓN y cosas que determinan la ARQUITECTURA de una pc, cómo lo explicarías?

Respuesta: Excelente pregunta, porque es una distinción fundamental que a menudo se confunde.

ARQUITECTURA (lo que ve el programador):

- **ISA**: conjunto de instrucciones disponibles (x86, ARM, RISC-V)
- **Registros**: cuántos hay y de qué tamaño (16 registros de 64 bits)
- **Modos de direccionamiento**: cómo se especifican las direcciones de memoria
- **Formato de instrucciones**: cómo se codifican las instrucciones
- **Tipos de datos**: qué tipos soporta (enteros, flotantes, etc.)
- **Manejo de interrupciones**: cómo se gestionan a nivel de programación

ORGANIZACIÓN (cómo se implementa internamente):

- **Pipeline**: cuántas etapas tiene y cómo está organizado
- **Unidades funcionales**: cuántas ALUs, FPUs, etc.
- **Jerarquía de caché**: tamaños de L1, L2, L3 y sus políticas
- **Predicción de saltos**: algoritmos usados para predecir branches
- **Ejecución fuera de orden**: si tiene y cómo funciona
- **Frecuencia de reloj**: a qué velocidad opera

- **Tecnología de fabricación**: 7nm, 5nm, etc.

Un ejemplo claro: Dos procesadores Intel Core i3 e i9 de la misma generación tienen la **misma arquitectura** (mismo ISA, mismos registros) pero **diferente organización** (diferentes cachés, más cores, diferentes frecuencias).

La arquitectura es el "contrato" con el software; la organización es cómo se cumple ese contrato internamente.

Volvamos ahora a una perspectiva más general.

Un computador es un sistema complejo de naturaleza jerárquica. Un sistema jerárquico es un conjunto de subsistemas interrelacionados, cada uno de los cuales, a su vez, se organiza en una estructura jerárquica, hasta que se alcanza el nivel más bajo del subsistema elemental.

En cada nivel, el sistema consta de un conjunto de componentes y sus interrelaciones. El comportamiento en cada nivel depende solo de una caracterización abstracta y simplificada del sistema que hay en el siguiente nivel más bajo. De cada nivel, al diseñador le importa la estructura y el funcionamiento:

- **Estructura**: el modo en que los componentes están interrelacionados.
- **Funcionamiento**: la operación de cada componente individual como parte de la estructura.

FUNCIONAMIENTO

Las funciones básicas de todas las computadoras son:

- procesamiento de datos
- almacenamiento de datos
- transferencia de datos
- control

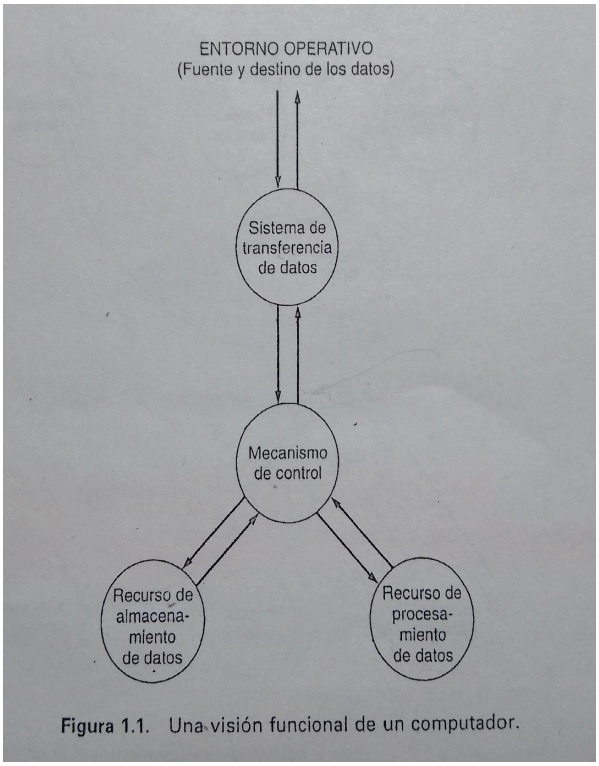


Figura 1.1. Una visión funcional de un computador.

El computador tiene que ser capaz de *transferir datos* entre él mismo y el mundo exterior. El entorno de operación del computador se componen de dispositivos que sirven bien como fuente o como destino de datos. Cuando se reciben o se llevan datos a un dispositivo que está directamente conectado con el computador, el proceso se conoce como *entrada-salida* (E/S), y este dispositivo recibe el nombre de *periférico*. El proceso de transferir datos a largas distancias, desde o hacia un dispositivo remoto, recibe el nombre de *comunicación de datos*.

Finalmente, debe haber un *control* de estas tres funciones. Este control es ejercido por el/los ente(s) que proporciona(n) instrucciones al computador. Dentro del computador, una unidad de control gestiona los recursos del computador y dirige las prestaciones de sus partes funcionales en respuesta a estas instrucciones.

A este nivel general de discusión, el número de operaciones posibles que pueden ser realizadas es pequeño. Hay cuatro posibles tipos de operaciones. La computadora, entonces, puede funcionar como un dispositivo de

- (a) movimiento de datos
- (b) almacenamiento de datos
- (c) procesamiento de datos en datos almacenados
- (d) procesamiento de datos en tránsito entre el almacén y el entorno externo (es decir, procesamiento desde almacenamiento a E/S, o desde E/S hacia almacenamiento).

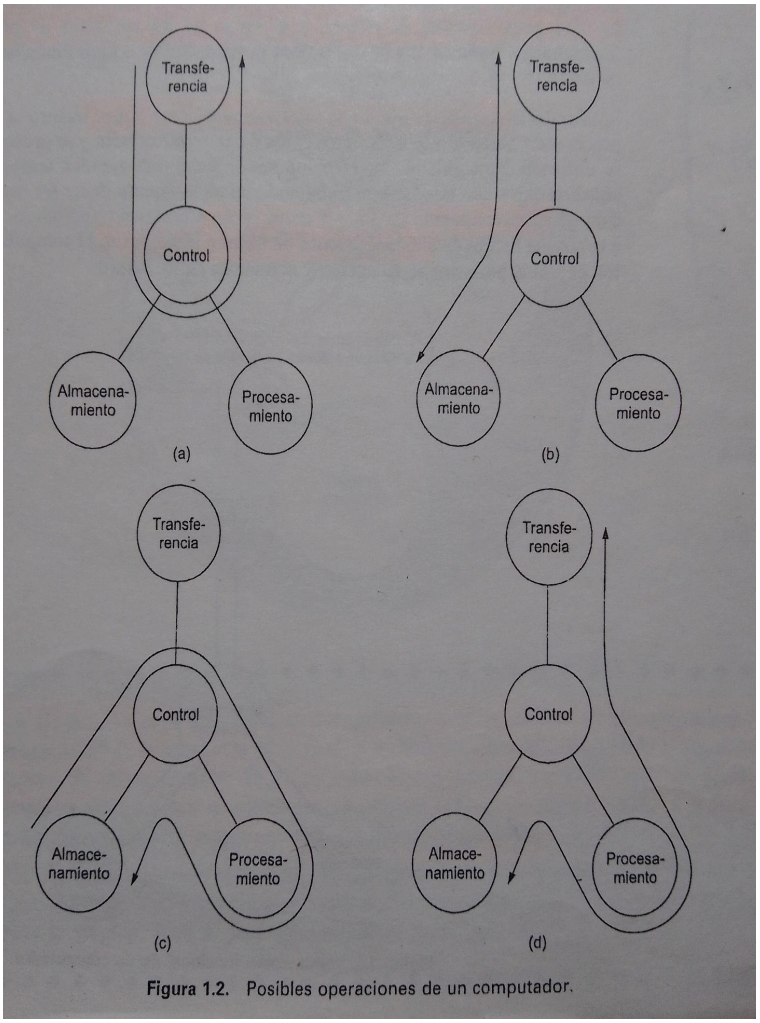


Figura 1.2. Posibles operaciones de un computador.

ESTRUCTURA

En la siguientes figuras, tenemos la representación más sencilla posible de un computador. El computador es una entidad que interactúa de alguna manera con su entorno externo.

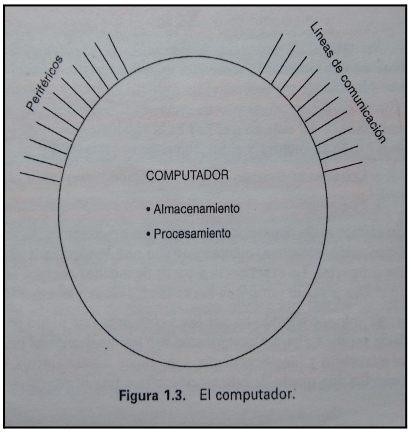
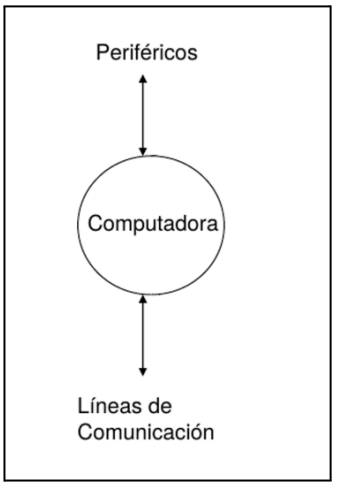


Figura 1.3. El computador.

Los periféricos (o dispositivos de entrada/salida locales) podrían ser, por ejemplo, el teclado, el monitor, etc., mientas que las líneas de comunicación (o interfaces de red/comunicación) podrían ser el ethernet, el WiFi, etc.



Pero lo que más nos interesa acá es la estructura interna del computador mismo, la cual tiene cuatro componentes principales:

- La **CPU** (*central processing unit*) controla el funcionamiento del computador y lleva a cabo sus funciones de procesamiento de datos. Frecuentemente se le llama simplemente *procesador*.
- La **memoria principal** (*main memory*) almacena datos.
- El **E/S** (en inglés, *I/O*) transfiere datos entre el computador y el entorno externo.
- El **sistema de interconexión** (*system bus*) es un mecanismo que proporciona la comunicación entre la CPU, la memoria principal y la E/S.

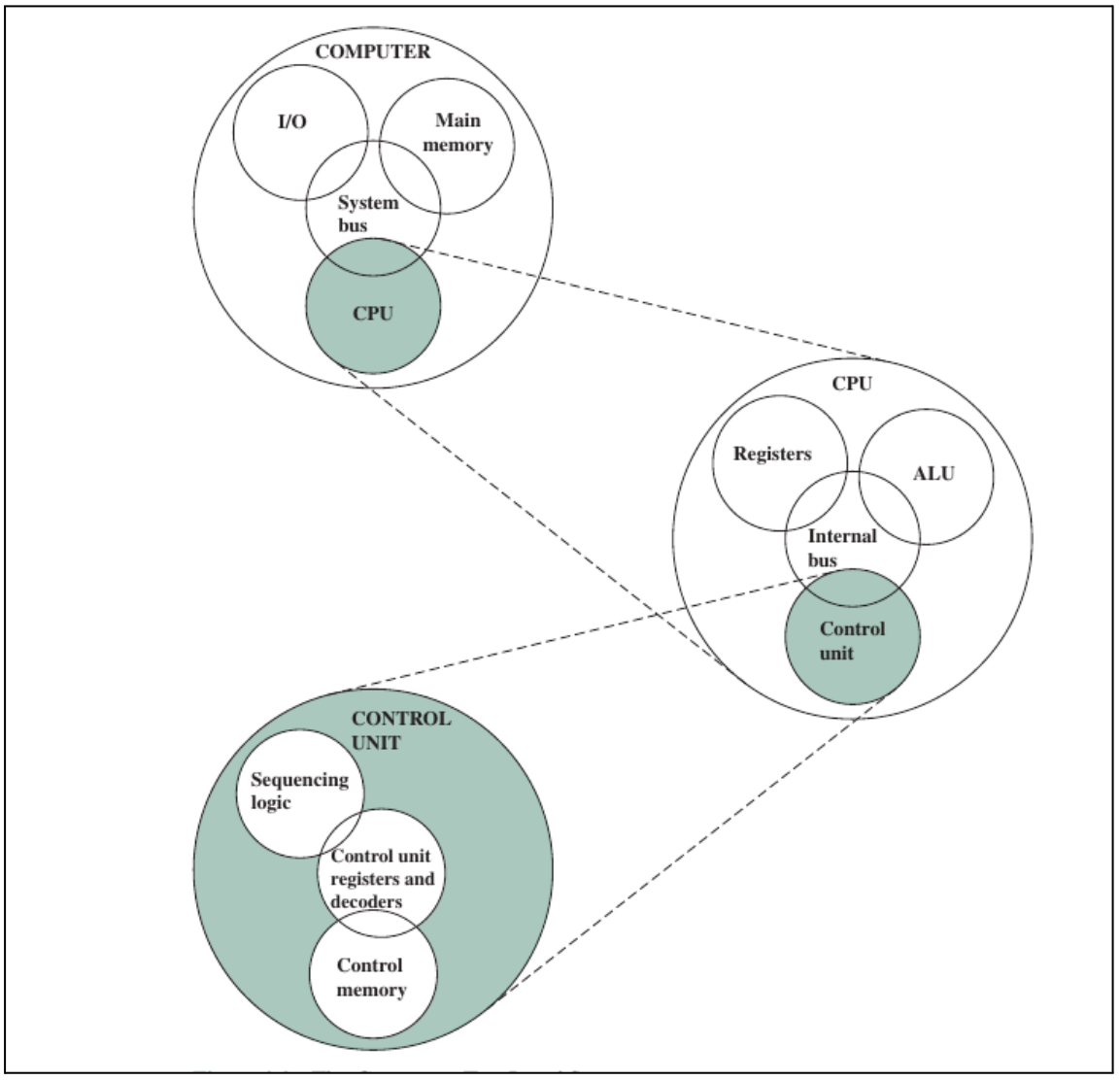
Puede que haya uno o más de cada uno de estos componentes. Tradicionalmente, por ejemplo, ha habido una sola CPU, pero hace ya algunas décadas existen computadores con más de una.

Yendo más en profundidad, haciendo zoom en nuestra PC, lo que más nos va a interesar ahora es la CPU. Sus principales componentes estructurales son:

- La **unidad de control** (UC o CU, *control unit*) controla el funcionamiento de la CPU y, por tanto, del computador.
- La unidad aritmético-lógica o **ALU** (*arithmetic and logic unit*) lleva a cabo las funciones de procesamiento de datos del computador.
- Los **registros** (*registers*) proporcionan almacenamiento interno a la CPU.
- La **interconexión interna de la CPU** (*CPU interconnection* o también *Internal bus*) es un mecanismo que permite la comunicación entre la UC, la ALU y los registros.

Y yendo más en detalle, dentro de la UC tenemos más componentes, pero no los estudiaremos ahora.

Todo lo anterior se resume en la siguiente imagen:



Primeras computadoras y la llegada de Von Neumann

Es difícil establecer a ciencia cierta cuál fue la primera computadora. Lo cierto es que la ENIAC (de 1946) fue una de las primeras. Básicamente, era una calculadora gigante. No utilizaba el sistema binario: era una máquina decimal. Funcionaba. Pero la tarea de cargar y modificar programas para el ENIAC era algo extremadamente tedioso: tenía que ser programado manualmente mediante conmutadores y conectando y desconectando cables.

El proceso de programación, pensaron algunos, podría ser más fácil si el programa se representara en una forma adecuada para ser guardado en la memoria junto con los datos.

Entonces, un ordenador podría conseguir sus instrucciones leyéndolas de la memoria, y se podría hacer o modificar un programa colocando los valores en una zona de memoria.

Esta idea, conocida como **concepto del programa-almacenado** [the *stored-program concept*], se le atribuye sobre todo a John Von Neumann.

Con este sueño en mente, John y sus amigos lanzaron la IAS en 1952, una computadora que tenía todas las características esenciales de lo que hoy llamamos "la arquitectura Von Neumann" o "modelo Von Neumann". Dicha computadora constaba de

- una **memoria principal**, que almacena tanto datos como instrucciones
- una **ALU**, capaz de hacer operaciones con datos binarios
- una **UC**, que interpreta las instrucciones en memoria y provoca su ejecución
- y un equipo de E/S dirigido por la unidad de control. (A veces, en vez de equipo de E/S, se suele hablar de una **unidad de entrada** (la que provee las instrucciones y los datos) y de **una unidad de salida** (la que envía los resultados)).

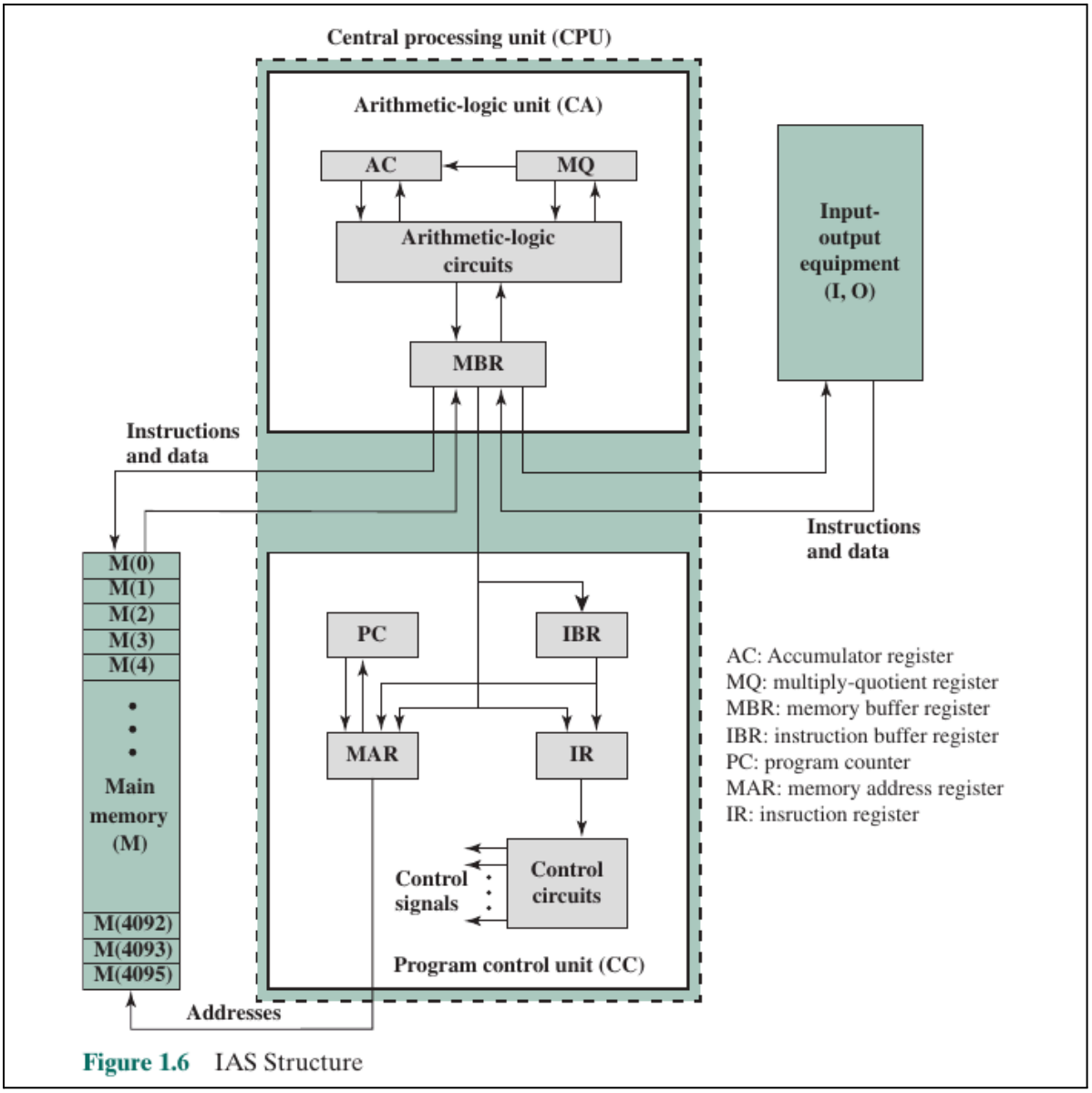


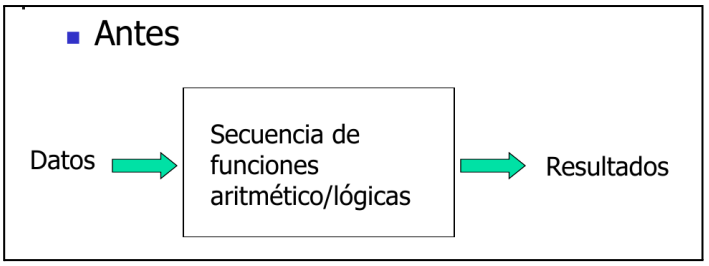
Figure 1.6 IAS Structure

Salvo raras excepciones, todas las computadoras de hoy en día tienen básicamente la misma estructura general y funcionamiento que la indicada en las máquinas de Von Neumann. Por ello merece la pena detenerse en este ejemplo.

Los aspectos más importantes del modelo Von Neumann son los siguientes:

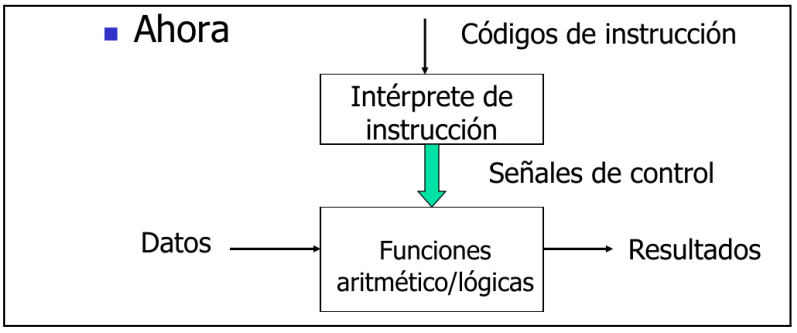
- **Se utiliza el sistema binario,** lo cual simplifica la implementación de funciones y disminuye la probabilidad de fallos.
- **Las instrucciones y datos residen en memoria**. Esto se implementa en conjunto con la idea de ejecutar el programa en forma secuencial. Por todo ello, se aumenta la velocidad.
- **La memoria es direccionable por localidad** sin importar el dato almacenado.

Y como ya mencionamos anteriormente, la clave del éxito estaba en el concepto de programa. Porque antes era así: se ingresaban datos → se les aplicaba una secuencia de funciones aritmético-lógicas →y obteníamos los resultados.



Todo lo cual, se realizaba con programación en hardware: cuando cambiábamos las tareas, debíamos modificar directamente el hardware (es decir, conectar y desconectar cables y esas cosas).

Ahora, desde la llegada de Von Neumann, la máquina es más compleja, pero la vida es más sencilla: ya no hay que tocar el hardware. Ahora a la máquina le ingresan tanto datos como códigos de instrucción (o *codops*), que deben ser interpretados, y luego la propia máquina entiende qué debe hacer, y luego tira los resultados:



Y así es como nació la programación en software, donde

- En cada paso se efectúa alguna operación sobre los datos.
- Para cada paso se necesita un nuevo conjunto de señales de control.
- Las instrucciones proporcionan esas señales de control.
- Aparece el nuevo concepto de programación.
- No hay que cambiar el hardware.

Ahora bien, ¿qué es un programa? Antes de decir lo que todos sabemos, revisemos la etimología: la palabra "programa" se deriva del griego "*prógramma*". Este término griego se forma a partir de "*pró-*" (antes) y "*gráphein*" (escribir), significando originalmente "lo escrito antes" o "lo publicado por adelantado". Por ello, en la antigua Grecia, "*prógramma*" se refería a un aviso público, un edicto o un anuncio. Así que podríamos decir que, en esencia, no perdió ese significado: ahora un programa es lo escrito por adelantado, para la computadora; o sea, son las instrucciones que le damos escritas, para que luego las ejecute.

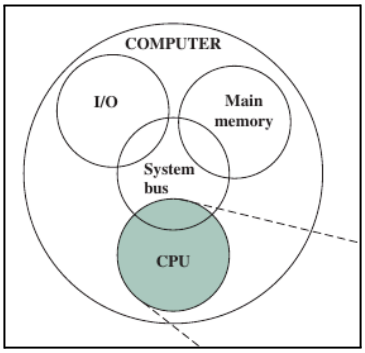
En términos más actuales, un programa es una secuencia de pasos, donde se hace una operación aritmético/lógica por cada paso. Y para cada operación se necesitan diferentes señales de control. Y, a su vez, para saber cuáles señales emitir, la UC interpreta el codop (código de operación) de cada instrucción, y luego envía las señales necesarias. (Esto lo veremos en detalle más adelante).

Interconexión de un sistema de cómputo

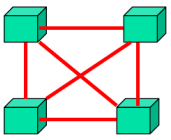
Como ya vimos, una compu está constituida por tres subsistemas: CPU, memoria y E/S.

Pero estos componentes, naturalmente, deben poder comunicarse entre si.

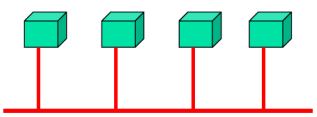
Para ello se inventaron los buses.



Ahora bien, a la hora de conectar componentes, tenemos dos opciones básicas: conectar todos con todos, o usar algo que conecte a todos mediante un canal de comunicación común.



Conexiones independientes entre los distintos dispositivos



Conexiones a través de un medio compartido

➢Pensar: ¿cómo conectar un nuevo dispositivo en cada sistema?

Sobre las dos opciones:

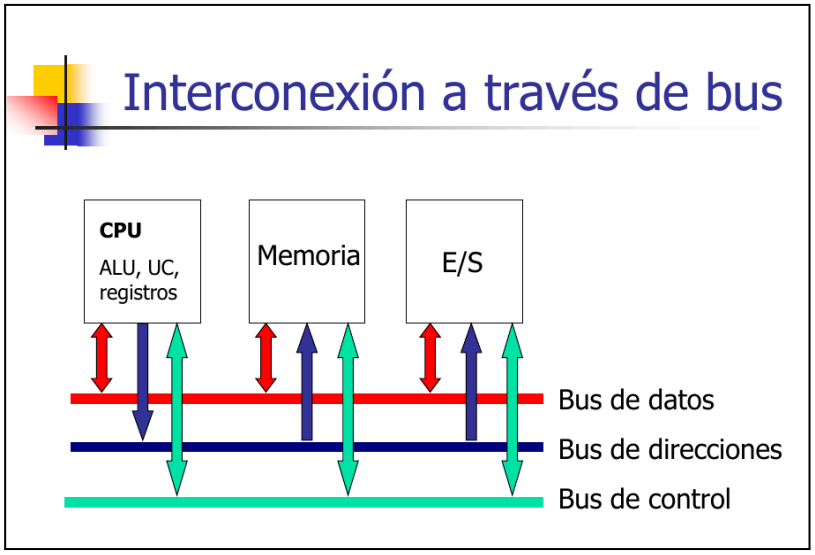
"Todos con todos" = topología de malla completa (muy costosa, pero rápida)

"Canal común" = topología de bus común (más económica, pero puede ser un cuello de botella)

En la realidad, no es una cosa o la otra, sino una combinación de ambas[2]. Pero tradicionalmente se utilizó un esquema más parecido al que se ve a la derecha.

¿Y qué es un **bus**? Es un camino de comunicación que conecta dos o más dispositivos.

- Usualmente de tipo "broadcast"[3].
- A menudo agrupadas, en el sentido de que existe un número de canales en un mismo bus (por ejemplo: un bus de 32 bits son 32 canales separados de un solo bit cada uno).
- Las líneas de energía pueden no mostrarse.



DIRECCIONES

- Si el bus es compartido por diferentes elementos, éstos deben tener identidades distintivas: direcciones.
- La dirección de memoria identifica una celda de memoria en la que almacena información.
- Lectura y escritura se plantean respecto de la CPU.

[2] La realidad de hoy en día: se usa una modalidad híbrida que cuenta con buses principales junto con conexiones punto a punto para componentes críticos.

[3] En Informática, la difusión amplia o *broadcast* es una conexión multipunto que permite la transmisión de información a usuarios de una red sin tener necesariamente las direcciones de cada destinatario. En cuando al proceso de difusión, un emisor envía información a los respectivos destinatarios simultáneamente desde un solo nodo en lugar de varios.
Dicho de forma sencilla: cuando la comunicación es de tipo *broadcast* significa que el emisor le envía el mensaje a todos los receptores, sin importar para quién es el mensaje (sin importale, por ejemplo, que el mensaje es solamente para uno de los componentes).

BUS DE DATOS

- Transporta datos (no hay diferencia entre “dato” e “instrucción” en éste nivel).
- El ‘ancho’ es un valor determinante de las prestaciones (ej.: 8, 16, 32 o 64 bits).

BUS DE DIRECCIONES

- Identifica el origen o el destino de los datos. Esto le sirve a la CPU, porque la CPU necesita leer una instrucción (dato) de una ubicación en memoria dada.
- El ancho del Bus determina la máxima capacidad de memoria del sistema. Por ejemplo, el Intel 8080 tiene un bus de direcciones de 16 bits dando un espacio de direcciones de 64k.

BUS DE CONTROL

- Información de control y temporizado
- Señales de lectura/escritura de Memoria o E/S
- Señales de selección o habilitación
- Señales de Reloj (Clock)
- Señales de pedido de Interrupción

Hablando un poco de hardware, podríamos hacer una distinción entre componentes de hardware para procesamiento y componentes para almacenamiento. Los componentes para procesamiento serían la CPU y la memoria (principal), mientras que en la segunda entrarían la memoria (principal), los discos (sean rígidos, diskettes o SSD), cintas, CD, DVD.

Let’s dive into the Central Processing Unit!

A continuación, trataremos de entender un poco mejor cómo funciona la CPU: el cerebro de la computadora. Ya presentamos los componentes principales (la ALU, la CU, etc.). Ahora pasaremos a detallar algunos registros que habíamos pasado por alto. Para ello, seguiremos parcialmente el recorrido de una instrucción.

Supongamos que leímos una instrucción. Llegó al MBR. Luego la instrucción se almacena temporalmente en un registro de la CPU llamado IR. El bloque control puede “leer” IR y así saber qué hacer, dónde están los operandos y dónde poner el resultado.

Dicho de forma más precisa: la unidad de control recibe el *opcode*[4] (*operation code* o código de operación) de la instrucción, que es la parte de la instrucción que especifica la operación que debe realizar la CPU, y el resto de la instrucción va a donde tenga que ir (si era un operando, quizás termine en la ALU o algún registro; si era, en cambio, una dirección, podría ir al MAR).

Una vez leída y ejecutada la instrucción: ¿cómo sabe la CPU dónde encontrar la próxima instrucción? Hay un registro en la CPU llamado PC (*Program Counter*[5]). Cuando un programa va a ser ejecutado, el PC contiene la dirección de la primera instrucción. Alcanzada la primera instrucción, el PC es incrementado para apuntar a la siguiente instrucción.

Todas las CPU tienen registros internos de propósito general que pueden ser referenciados por el programador, como fuente o destino (o ambos) en una instrucción. "Como si" fuesen memoria, pero mucho más rápidos. Son lugares de almacenamiento temporario: D0, D1, D2, etc.

La CPU interactúa con la memoria a través de un par de registros que están "ocultos" al programador.

- MAR = registro de dirección de memoria →acá se colocan las direcciones que saldrán por el bus de direcciones.
- MBR = registro de dato de memoria →acá los datos que llegan o salen por el bus de datos.

Además la CPU tiene otros registros que permiten almacenar direcciones; para poder brindar flexibilidad (como A0, A1, A2, etc.).

Si no se entendió de verdad cómo funciona la CPU, es porque no recorrimos todo el ciclo. Eso es lo que veremos a continuación.

Ciclo de instrucción

Podemos descomponer el procesamiento de instrucciones en dos etapas:

1) Búsqueda: leer desde memoria. Esto es común a todas las instrucciones.
2) Ejecución: dependiendo de la instrucción puede implicar varias operaciones.

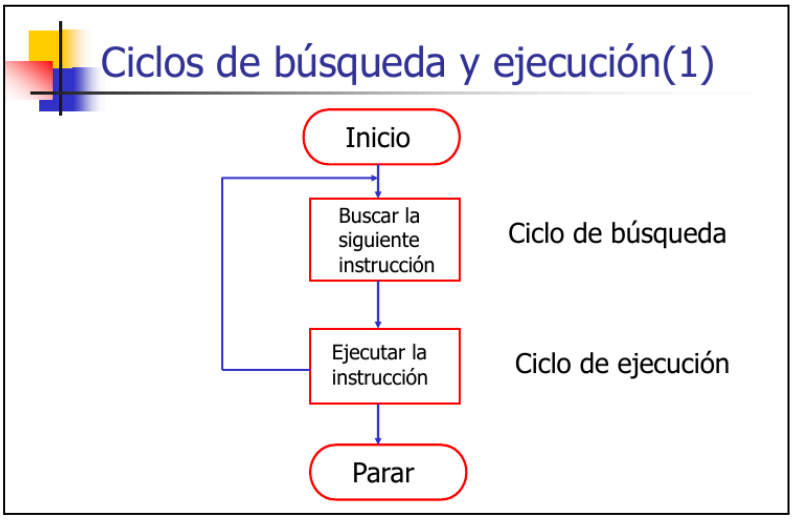
Dicho de otra manera: el procesamiento requerido para una sola instrucción se llama ciclo de instrucción, y a este podemos descomponerlo en dos pasos: el subciclo de búsqueda y el subciclo de ejecución.

---

[4] Por ejemplo, una instrucción del VonSim podría ser *ADD AX, BX*. En este caso, ADD sería el opcode, AX el primer operando y BX el segundo operando.
[5] BY THE WAY, en el VonSim es lo que aparece como IP, por *Instruction Pointer*.

Una vez que se procesa una instrucción en un programa, le sigue otra. La ejecución del programa se interrumpe sólo si la máquina se apaga, hay un error o aparece una instrucción que interrumpa a la computadora.

Gráficamente, lo que dijimos recién puede representarse así:



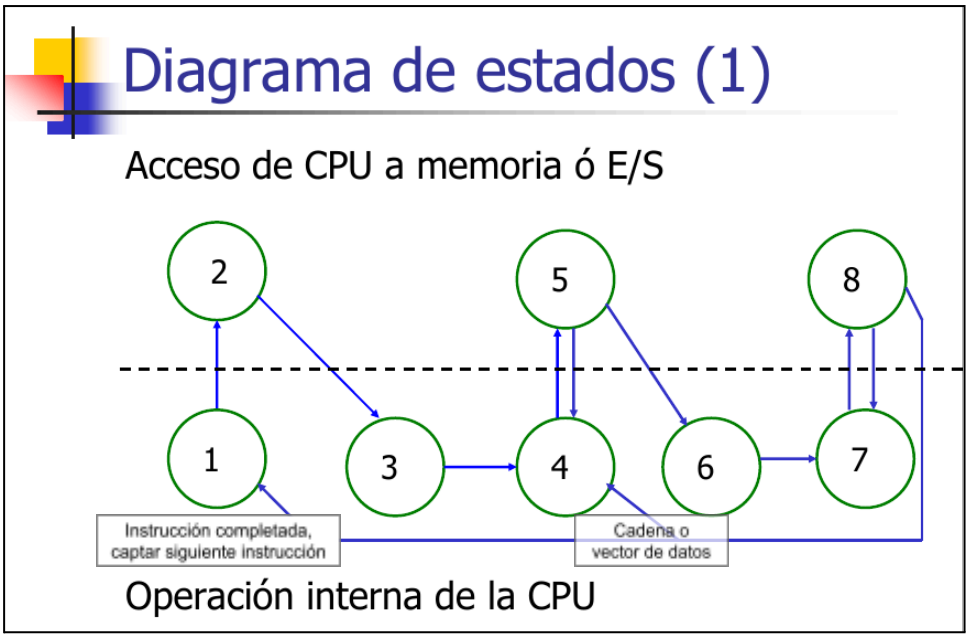
Pero, ¿cómo empieza todo? Supongamos un inicio sencillo. Supongamos que ya tenemos la dirección de la próxima instrucción a ejecutar en el PC. Lo que sucede a continuación, es que esa dirección se copia en el registro MAR, y el PC se incrementa en 1, para ya tener lista la dirección de la siguiente instrucción. Pero todavía estamos con la primera, que quedó en el registro MAR. De ahí, se envía por el bus de direcciones y eso sirve para saber qué celda de la memoria leer. Ese dato, que en realidad es —en este caso— la instrucción que vamos a ejecutar, se copia en el MBR y luego pasa al IR. La instrucción está en la forma de un código binario que especifica las acciones que tomará la CPU. La CPU interpreta cada instrucción y lleva a cabo las acciones requeridas.

En general las acciones pueden clasificarse en alguno de estos 4 tipos:

- CPU — Memoria → datos pueden transferirse entre memoria y CPU.
- CPU — E/S → datos pueden transferirse entre CPU y entrada/salida.
- Procesamiento de datos → CPU efectúa operaciones aritméticas o lógicas en datos.
- Control → alterar la secuencia de ejecución de instrucciones.

Las filminas 11 a 24 de la clase 6 dan algunos ejemplos detallados de todo esto.

Ahora bien, siendo más realistas, el ciclo de instrucción no consta solo de dos etapas, sino que puede llegar a tener varios pasos. Además, como no todas las instrucciones son iguales, algunas pueden requerir mayor o menor cantidad de pasos. El objetivo de la siguiente figura es describir un ciclo de instrucción típico en forma de diagrama de estados, pero obviamente para cualquier ciclo de instrucción dado, algunos estados pueden no estar y otros pueden repetirse.



1) **Cálculo de la dirección de la instrucción**: determina la dirección de la siguiente instrucción a ejecutar. Normalmente, esto implica añadir un número fijo a la dirección de la instrucción previa (típicamente, esto sería incrementar el PC). Por ejemplo, si las instrucciones tienen un tamaño de 16 bits y la memoria se organiza en palabras de 16 bits, se suma 1 a la dirección previa. En cambio, si la memoria se organiza en bytes direccionables individualmente, entonces hay que sumar 2 a la dirección previa.
2) **Búsqueda de la instrucción (o Captación de instrucción)**: la CPU lee la instrucción desde su posición en memoria.
3) **Decodificación de la operación indicada en la instrucción**: analiza la instrucción para determinar el tipo de operación a realizar y los operandos que se usarán.
4) **Cálculo de la dirección del operando (fuente)**: si la operación implica la referencia a un operando en la memoria o disponible mediante E/S, entonces se determina la dirección del operando (fuente).
5) **Búsqueda del operando (o Captación de operando)**: capta el operando desde memoria o se lee desde el dispositivo de E/S.
6) **Operación sobre los datos**: realiza la operación indicada en la instrucción.
7) **Cálculo de la dirección del operando (destino)**: si es necesario (es decir, si el resultado va a ir a parar a memoria o a algún dispositivo de E/S), calcula dónde almacenar el resultado (es decir, el operando destino).
8) **Almacenamiento de operando**: escribe el resultado en memoria o lo saca a través de un dispositivo de E/S.

¿Por qué hay una flecha de 8 a 4? Porque en algunas máquinas, se puede especificar con una única instrucción una operación a realizar con un vector de números o con una cadena [*string*] de caracteres. Por ello, como lo indica la figura, esto indicaría una repetición de estados de captación y/o almacenamiento de operando (o sea, se repetiría el ciclo pero sin pasar por los estados 1, 2 y 3).

Formatos de instrucción[6]

No todas las instrucciones son iguales, en el sentido de que no todas tienen el mismo formato. Sin embargo, hay 4 componentes principales que se consideran parte de la instrucción, por más que no estén presentes en todos los formatos de forma explícita. Estos componentes son: 1), 2.1), 2.2) y 3).

1) Un **código de operación** (codop) o, en inglés, un *operation code* (opcode). Esta es la parte de la instrucción que especifica la operación a realizar (por ejemplo: ADD, SUB, AND, etc.). Esto es lo mínimo e indispensable para que haya una instrucción: sin codop no hay instrucción. Y de hecho, puden existir instrucciones de cero direcciones que siempre tengan sus operandos implícitos (ver Anexo 2).
2) Al menos una **referencia a un operando**. Esta referencia indica dónde se encuentra el operando con el cual vamos a trabajar (puede ser el operando fuente o el operando de destino). En casos *muy especiales*, puede que alguna instrucción no necesite usarlo, ya que hay instrucciones que no necesitan ningún operando (pensemos por ejemplo en la instrucción HLT).

Con estos dos elementos ya podríamos armarnos instrucciones para una máquina de 1 dirección, ya que las instrucciones de esta máquina solo deben contar con un codop y una referencia del operando fuente (o de destino). No necesita indicar explícitamente cuál será el segundo operando (ni el resultado) porque se sabe que será lo que esté cargado en el acumulador; ni tampoco la siguiente instrucción (ya que está en el PC).

De todas formas, si bien la máquina de 1 dirección existe, es una máquina un poco especial. Lo más normal es trabajar con máquinas de dos direcciones. Pero antes de pasar a eso, continuemos viendo los elementos de una instrucción máquina. Empezemos por descomponer la referencia a un operando en dos:

2.1) **Referencia del operando fuente**. Establece dónde se encuentra el operando. La operación puede involucrar uno o más operando fuente (o de entrada).

2.2) **Referencia del operando resultado**: establece dónde almacenar el resultado.

---

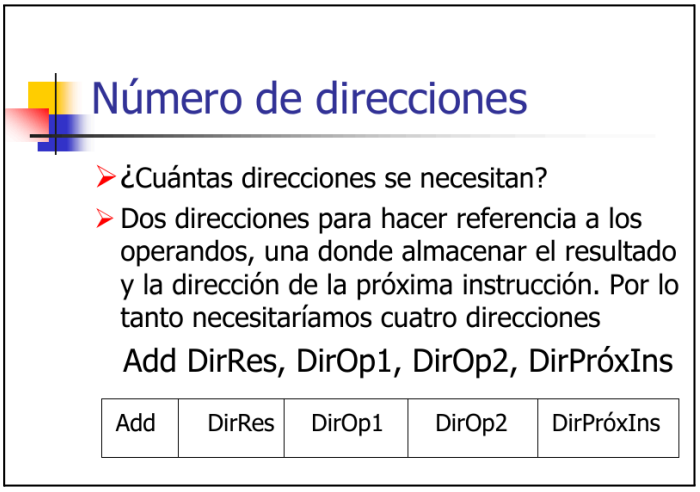
[6] No está de más aclarar que, en este contexto, instrucción siempre significa INSTRUCCIÓN-MÁQUINA. ¿Y qué es eso? Básicamente, una instrucción de máquina es una orden básica que el CPU puede ejecutar directamente, sin requerir ningún paso previo o traducción previa.
La colección o SET DE INSTRUCCIONES de instrucciones de máquina que puede interpretar y ejecutar un procesador, constituyen su ISA (del inglés *instruction set architecture*, «arquitectura del conjunto de instrucciones»). El ISA define la arquitectura del conjunto de instrucciones, que es una parte clave de la arquitectura general de la computadora.
Con todo eso, podemos llegar a entender el LENGUAJE DE MÁQUINA (o *machine language*): es el sistema de códigos directamente interpretable por la CPU. Este lenguaje está compuesto, justamente, por las instrucciones-máquina. El lenguaje de máquina es específico de la arquitectura de la máquina.
Machine language is the lowest level of computer programming language, consisting of binary code (sequences of 0s and 1s) that the computer's CPU directly executes. It's the language the hardware understands, and all other programming languages are ultimately translated into machine code for execution. (Esto se ilustra en la última diapositiva de la clase 8, donde se compara el código máquina con el código del lenguaje ensamble).

En el VonSim, por ejemplo, que es una máquina de dos direcciones, la primera dirección es la referencia del operando destino, y la segunda la del operando fuente.

Por otro lado, también es posible que la instrucción contenga otro elemento más, es posible que exista una

3) **Referencia de la siguiente instrucción**: le dice a la CPU dónde buscar la siguiente instrucción después de la ejecución de la instrucción anterior. En la mayoría de los casos se ubica a continuación de la instrucción actual.

Es raro que tenga eso, pero bueno, acá tenemos un ejemplo de cómo quedaría en una máquina de cuatro direcciones:

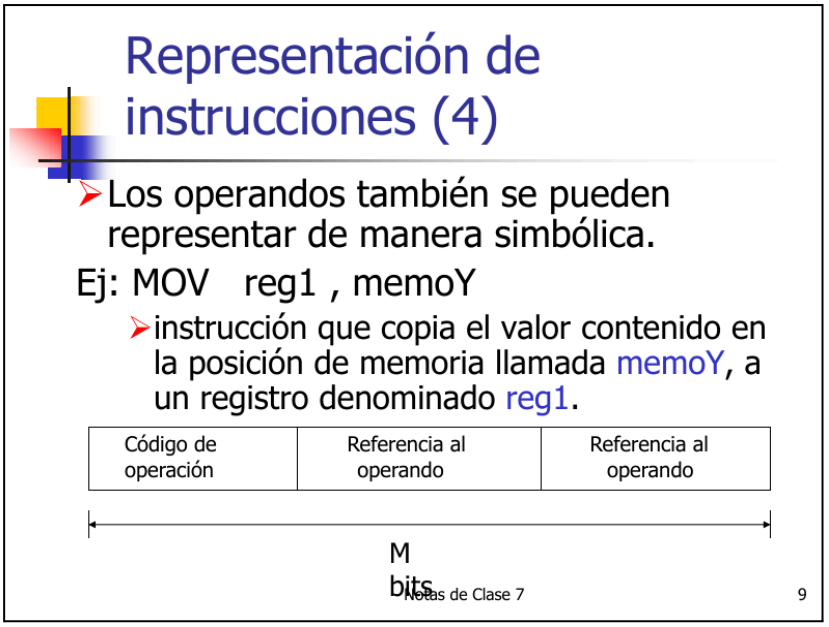


Los operandos fuente y resultado pueden estar en tres lugares:

- Memoria
- Registro de la CPU
- Dispositivo de E/S

Así como los números para la computadora no son más que unos y ceros, así también, las instrucciones no son más que unos y ceros. Pero como al programador le resulta difícil recordar y expresarse en esos términos, para referirse a los codops se utiliza una representación simbólica. Los códigos de operación se representan por medio de abreviaturas, llamadas mnemónicos que indican la operación. O sea, en vez de escribir 010101, ponemos ADD y ya está, después algún compilador (como el assembler, que es el programa que traduce assembly a código máquina) lo traducirá a binario.

Y, de alguna manera, los operandos también se pueden representar de manera simbólica. Eso es lo que hacemos en el VonSim cuando utilizamos variables en vez de utilizar direcciones de memoria de forma “bruta”. O sea, las variables nos sirven como etiquetas para referenciar de forma elegante direcciones de memoria como 1002h —por tirar un número—, y le aportan flexibilidad y claridad al código. Aquí podemos verlo en un ejemplo de una máquina de dos direcciones:



ACLARACIÓN PARA PREGUNTAS TRAMPOSAS

Dentro de la computadora cada instrucción está representada mediante una secuencia de bits. La secuencia se divide en *campos* en correspondencia a los elementos que la componen. ENTONCES, si me preguntan "¿cuántos campos tiene una instrucción de una máquina de dos direcciones?", la respuesta no será 2. La respuesta será 3, ya que además de los dos campos de las direcciones, está el campo —el insoslayable campo— del codop.

Modo de direccionamiento: la forma en que una instrucción dice dónde está un dato

Como ya vimos, normalmente al menos uno de los campos de la instrucción sirve para hacer referencia a algún operando. Ahora bien, esto puede hacerse de varias maneras. Y existen buenos motivos. Los modos de direccionamiento tienen como objetivo

- disminuir la cantidad de bits en la instrucción
- la dirección puede que no se conozca hasta el momento de ejecutar el programa
- manejo más eficiente de datos (arreglos)

Hay algunos MDD bastante sencillos: el inmediato, el directo, por registro, indirecto por memoria, indirecto por registro y del stack.

Ahhh, pero el MDD "por desplazamiento" puede ser confuso. Repasemos ese.

La idea básica es clara: realizamos una suma entre dos cosas, y el resultado es la dirección donde está lo que buscamos. ¿Qué dos cosas sumamos? Para decirlo en términos no-muy-técnicos, sumamos una *dirección base* y un *desplazamiento*. La dirección base suele ser parecida al resultado final, y el desplazamiento suele ser un número pequeño; así solíamos verlo en el VonSim, pero obviamente no tiene por qué ser siempre así. Ahora bien, la cosa es que la dirección base y el desplazamiento pueden ser cosas distintas. Normalmente, se suelen distinguir tres tipos de MDD por desplazamiento:

1) **Relativo** → *aquí se suma lo que haya en el PC, con la dirección*[7] *que ya tenemos en la instrucción*. (Lo primero sería la base, y lo segundo el desplazamiento).
2) **De registro base** →*aquí se suma lo que haya en algún registro (el típico es BX), con la dirección que ya tenemos en la instrucción.* (Lo primero sería la base, y lo segundo el desplazamiento).
3) **Indexado**[8] →*aquí se suma lo que haya en el registro índice, con la dirección que ya tenemos en la instrucción.* (En este caso, el que se encarga del desplazamiento sería el registro índice, y lo que tenemos en la instrucción sería la dirección base). La ventaja de este MDD es que, en algunos sistemas (no todos), el registro índice está preparado para incrementarse automáticamente (lo cual se denomina *autoindexado*).

En las clases prácticas solo vimos ejemplos del segundo tipo. Es cuando poníamos algo tipo:

MOV AL, **[BX + 2]**

Aquí la instrucción claramente nos avisa: *obtendrás la dirección cuando sumes BX con el valor que ya tengo dentro mío* (que es el número 2).

Lamentablemente, la historia no termina acá. Esta es la visión teórica y general. En la realidad, cada procesador combina distintos MDD con las herramientas que tenga a mano, es decir, con otros tipos de registros y posibilidades que aquí no mencionamos. Esto puede verse en la clase 8.

Organización de registros

Registros visibles al usuario: son utilizados por el programador. Ejemplos:

- Propósito general: AX, BX, CX, DX (en x86).

[7] Cuando decimos “con la dirección que ya tenemos en la instrucción”, en realidad conviene entenderse como “el contenido de un campo de dirección de la instrucción”. ¿Por qué? Porque en el MDD por desplazamiento en realidad la dirección no viene dada en la instrucción, sino que lo que viene en la instrucción es simplemente un valor que sumado a otro posteriormente dará por resultado la dirección efectiva.
[8] “Index” viene del inglés. Significa “índice”.

- Datos: registros para operaciones aritméticas (muchas veces son los mismos de propósito general).
- Dirección: Punteros: SP[9] (Stack Pointer), BP (Base Pointer); Índices: SI (Source Index), DI (Destination Index).
- Códigos de condición: FLAGS/EFLAGS (contiene Zero Flag, Carry Flag, etc.).

Registros de control y estado: son utilizados por la UC para controlar la operación de la CPU (en la mayoría de las máquinas no son visibles al programador). Los 4 esenciales para la ejecución de instrucciones:

- Contador de programa (PC).
- Registro de instrucción (IR).
- Registro de dirección de memoria (MAR).
- Registro buffer de memoria (MBR).

Formatos de Instrucción: Modos de Direccionamiento y Formatos (tomado de Stallings, pp. 469-470)

Un formato de instrucción define la disposición de los bits de una instrucción, en términos de sus campos constituyentes. Un formato de instrucción debe incluir un código de operación y, implícita o explícitamente, cero o más operandos. Cada operando explícito se referencia usando uno de los MDD vistos anteriormente. El formato debe, implícita o explícitamente, indicar el MDD para cada operando. Para la mayoría de conjuntos de instrucciones, se utiliza más de un formato de instrucción.

El diseño de un formato de instrucción es un arte complejo, y se ha implementado una variedad asombrosa de diseños. Examinamos las cuestiones clave de diseño, revisando brevemente algunos diseños para ilustrar puntos, y luego examinamos las soluciones de x86 y ARM en detalle.

Longitud de Instrucción

La cuestión de diseño más básica a enfrentar es la longitud del formato de instrucción. Esta decisión afecta, y es afectada por, el tamaño de memoria, la organización de memoria, la estructura del bus, la complejidad del procesador y la velocidad del procesador. Esta decisión determina la riqueza y flexibilidad de la máquina tal como la ve el programador de lenguaje ensamblador.

El compromiso más obvio aquí es entre el deseo de un repertorio de instrucciones poderoso y la necesidad de ahorrar espacio. Los programadores quieren más códigos de operación, más operandos, más modos de direccionamiento y mayor rango de direcciones. Más códigos de operación y más operandos facilitan la vida del programador, porque se pueden escribir

[9] El programador puede modificar el SP, pero normalmente conviene no hacerlo y dejar que se modifique automáticamente.

programas más cortos para realizar tareas dadas. Similarmente, más modos de direccionamiento dan al programador mayor flexibilidad en la implementación de ciertas funciones, como manipulaciones de tablas y ramificaciones múltiples. Y, por supuesto, con el aumento en el tamaño de la memoria principal y el uso creciente de memoria virtual, los programadores quieren poder direccionar rangos de memoria más grandes. Todas estas cosas (códigos de operación, operandos, modos de direccionamiento, rango de direcciones) requieren bits y empujan en la dirección de longitudes de instrucción más largas. Pero una longitud de instrucción mayor puede ser un desperdicio. Una instrucción de 64 bits ocupa el doble del espacio que una instrucción de 32 bits pero probablemente es menos del doble de útil.

Más allá de este compromiso básico, hay otras consideraciones. Ya sea que la longitud de instrucción debería ser igual a la longitud de transferencia de memoria (en un sistema de bus, longitud del bus de datos) o una debería ser múltiplo de la otra. De otro modo, no obtendremos un número integral de instrucciones durante un ciclo de búsqueda. Una consideración relacionada es la tasa de transferencia de memoria. Esta tasa no ha mantenido el ritmo con los aumentos en la velocidad del procesador. En consecuencia, la memoria puede convertirse en un cuello de botella si el procesador puede ejecutar instrucciones más rápido de lo que puede buscarlas. Una solución a este problema es usar memoria caché; otra es usar instrucciones más cortas. Así, las instrucciones de 16 bits pueden ser buscadas al doble de la tasa de las instrucciones de 32 bits pero probablemente pueden ser ejecutadas en menos del doble de rapidez.

Una característica aparentemente mundana pero no obstante importante es que la longitud de instrucción debería ser un múltiplo de la longitud del carácter, que usualmente es de 8 bits, y de la longitud de los números de punto fijo. Para ver esto, necesitamos hacer uso de esa palabra desafortunadamente mal definida, *palabra*. La longitud de palabra de la memoria es, en cierto sentido, la unidad "natural" de organización. El tamaño de una palabra usualmente determina el tamaño de los números de punto fijo (usualmente los dos son iguales). El tamaño de palabra también es típicamente igual a, o al menos integralmente relacionado con, el tamaño de transferencia de memoria. Debido a que una forma común de datos son los datos de caracteres, nos gustaría que una palabra almacene un número integral de caracteres. De otro modo, hay bits desperdiciados en cada palabra al almacenar múltiples caracteres, o un carácter tendrá que atravesar un límite de palabra. La importancia de este punto es tal que IBM, cuando introdujo el System/360 y quería emplear caracteres de 8 bits, tomó la decisión desgarradora de moverse de la arquitectura de 36 bits de los miembros científicos de la serie 700/7000 a una arquitectura de 32 bits.

Memoria

El problema principal es que la velocidad del procesador históricamente aumentó más rápido que la velocidad de la memoria. Si esta desigualdad aumentara sin parar, el rendimiento del sistema empeoraría (creando un cuello de botella donde la CPU esperaría constantemente por datos). Para intentar reducir este desfasaje, se desarrollaron distintos tipos de memoria con diferentes características de velocidad, capacidad y costo. Por ello, una computadora típica emplea distintos tipos de memoria, desde la más rápida y cara (registros) hasta la más barata y lenta (discos). La forma en que se organizan estos distintos tipos de memoria es lo que se conoce como jerarquía de memoria.

La Jerarquía de Memoria

Las restricciones de diseño en la memoria de una computadora pueden resumirse en tres preguntas: ¿Cuánta? ¿Qué tan rápida? ¿Qué tan costosa?

La pregunta de cuánta es algo abierta. Si la capacidad está ahí, es probable que se desarrollen aplicaciones para usarla. La pregunta de qué tan rápida es, en cierto sentido, más fácil de responder. Para lograr el máximo rendimiento, la memoria debe ser capaz de mantenerse al día con el procesador. Es decir, mientras el procesador está ejecutando instrucciones, no quisiéramos que tuviera que pausarse esperando instrucciones u operandos. La pregunta final también debe considerarse. Para un sistema práctico, el costo de la memoria debe ser razonable en relación con otros componentes.

Como era de esperarse, hay un compromiso entre las tres características clave de la memoria: capacidad, tiempo de acceso y costo. Se usa una variedad de tecnologías para implementar sistemas de memoria, y a través de este espectro de tecnologías, se mantienen las siguientes relaciones:

- **Menor tiempo de acceso (= más rápida), mayor costo por bit;**
- **Mayor capacidad, menor costo por bit;**
- **Mayor capacidad, mayor tiempo de acceso (= más lenta).**

El dilema que enfrenta el diseñador es claro. Al diseñador le gustaría usar tecnologías de memoria que proporcionen memoria de gran capacidad, tanto porque la capacidad es necesaria como porque el costo por bit es bajo. Sin embargo, para cumplir con los requisitos de rendimiento, el diseñador necesita usar memorias costosas, de capacidad relativamente menor, con tiempos de acceso cortos.

La salida de este dilema no es depender de un solo componente o tecnología de memoria, sino emplear una jerarquía de memoria. Una jerarquía típica se ilustra en la Figura 4.1. A medida que uno desciende por la jerarquía, ocurre lo siguiente:

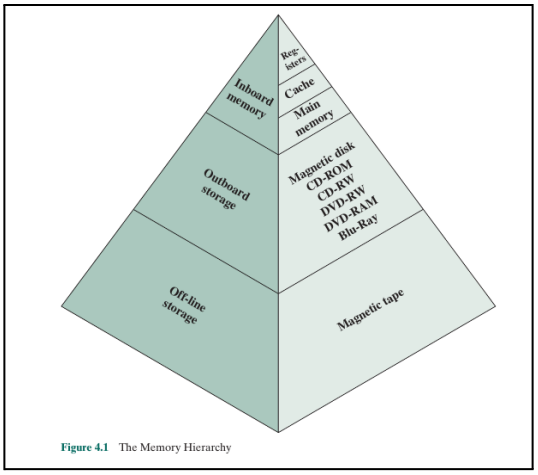


Figure 4.1 The Memory Hierarchy

a. **Costo decreciente por bit;**

b. **Capacidad creciente;**

c. **Tiempo de acceso creciente;**

d. **Frecuencia decreciente de acceso a la memoria por parte del procesador.**

Así, las memorias más pequeñas, más costosas y más rápidas son complementadas por memorias más grandes, más baratas y más lentas. La clave del éxito de esta organización es el punto (d): frecuencia decreciente de acceso. Examinamos este concepto en mayor detalle cuando discutimos la caché, más adelante en este capítulo.

El uso de dos niveles de memoria para reducir el tiempo promedio de acceso funciona en principio, pero solo si se aplican las condiciones (a) hasta (d). Al emplear una variedad de tecnologías, existe un espectro de sistemas de memoria que satisface las condiciones (a) hasta (c). Afortunadamente, la condición (d) también es generalmente válida.

La base para la validez de la condición (d) es un principio conocido como **localidad de referencia**. Durante el curso de ejecución de un programa, las referencias de memoria por parte del procesador, tanto para instrucciones como para datos, tienden a agruparse. Los programas típicamente contienen una cantidad de bucles iterativos y subrutinas. Una vez que se entra a un bucle o subrutina, hay referencias repetidas a un conjunto pequeño de instrucciones. Similarmente, las operaciones en tablas y arreglos involucran acceso a un conjunto agrupado de palabras de datos. Durante un período largo de tiempo, los grupos en uso cambian, pero durante un período corto de tiempo, el procesador está trabajando principalmente con grupos fijos de referencias de memoria.

Ejemplo 4.1: Jerarquía de Memoria de Dos Niveles

Supongamos que el procesador tiene acceso a dos niveles de memoria. El Nivel 1 contiene 1000 palabras y tiene un tiempo de acceso de 0.01 ms; el nivel 2 contiene 100,000 palabras y tiene un tiempo de acceso de 0.1 ms. Asumamos que si una palabra a ser accedida está en el nivel 1, entonces el procesador la accede directamente. Si está en el nivel 2, entonces la palabra es primero transferida al nivel 1 y luego accedida por el procesador. Por simplicidad, ignoramos el tiempo requerido para que el procesador determine si la palabra está en el nivel 1 o nivel 2.

La Figura 4.2 muestra la forma general de la curva que cubre esta situación. La figura muestra el tiempo promedio de acceso a una memoria de dos niveles como función de la tasa de aciertos H, donde H se define como la fracción de todos los accesos a memoria que se encuentran en la memoria más rápida (por ejemplo, la caché), T1 es el

tiempo de acceso al nivel 1, y T2 es el tiempo de acceso al nivel 2[10]. Como puede verse, para altos porcentajes de acceso al nivel 1, el tiempo total promedio de acceso está mucho más cerca del nivel 1 que del nivel 2.

En nuestro ejemplo, supongamos que el 95% de los accesos a memoria se encuentran en el nivel 1. Entonces el tiempo promedio para acceder a una palabra puede expresarse como:

(0.95)(0.01 ms) + (0.05)(0.01 ms + 0.1 ms) = 0.0095 + 0.0055 = 0.015 ms

El tiempo promedio de acceso está mucho más cerca de 0.01 ms que de 0.1 ms, como se deseaba.

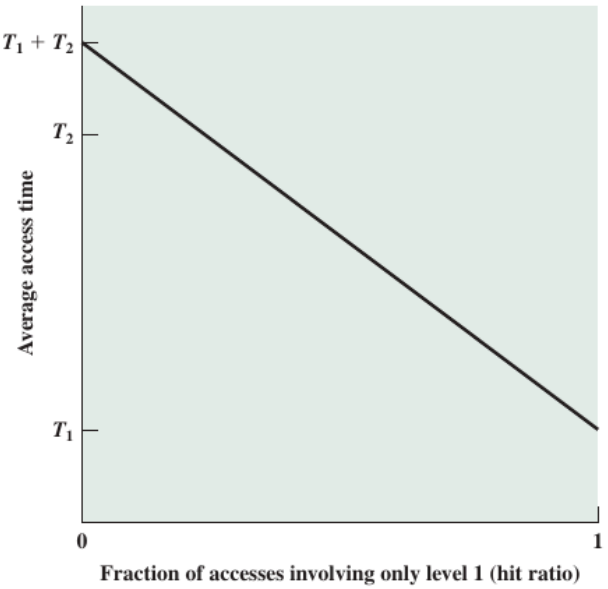


Figure 4.2 Performance of Accesses Involving only Level 1 (hit ratio)

En consecuencia, es posible organizar datos a través de la jerarquía de tal manera que el porcentaje de accesos a cada nivel sucesivamente más bajo sea sustancialmente menor que el del nivel superior. Consideremos el ejemplo de dos niveles ya presentado. Que la memoria del nivel 2 contenga todas las instrucciones y datos del programa. Los grupos actuales pueden ser temporalmente colocados en el nivel 1. De vez en cuando, uno de los grupos en el nivel 1 tendrá que ser intercambiado de vuelta al nivel 2 para hacer espacio para un nuevo grupo que entra al nivel 1. En promedio, sin embargo, la mayoría de las referencias serán a instrucciones y datos contenidos en el nivel 1.

[10] If the accessed word is found in the faster memory, that is defined as a **hit**. A **miss** occurs if the accessed word is not found in the faster memory.

Este principio puede aplicarse a través de más de dos niveles de memoria, como sugiere la jerarquía mostrada en la Figura 4.1. El tipo de memoria más rápido, más pequeño y más costoso consiste en los registros internos del procesador. Típicamente, un procesador contendrá unas pocas docenas de tales registros, aunque algunas máquinas contienen cientos de registros.

La memoria principal es el sistema de memoria interno principal de la computadora. Cada ubicación en la memoria principal tiene una dirección única. La memoria principal usualmente se extiende con una caché de mayor velocidad y menor tamaño. La caché usualmente no es visible al programador o, de hecho, al procesador. Es un dispositivo para organizar el movimiento de datos entre la memoria principal y los registros del procesador para mejorar el rendimiento.

Las tres formas de memoria recién descritas son, típicamente, volátiles y emplean tecnología de semiconductores. El uso de tres niveles explota el hecho de que la memoria de semiconductores viene en una variedad de tipos, que difieren en velocidad y costo.

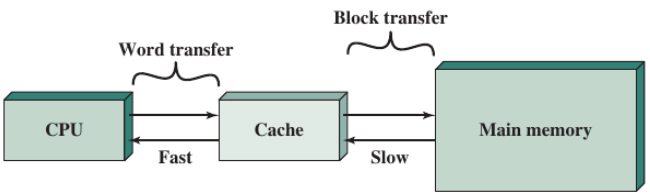
Los datos se almacenan más permanentemente en dispositivos de almacenamiento masivo externos, de los cuales los más comunes son el disco duro y medios removibles, como disco magnético removible, cinta y almacenamiento óptico. La memoria externa, no volátil, también se conoce como memoria secundaria o memoria auxiliar. Estas se usan para almacenar archivos de programas y datos y usualmente son visibles al programador solo en términos de archivos y registros, a diferencia de bytes o palabras individuales.

Principios de la Memoria Caché

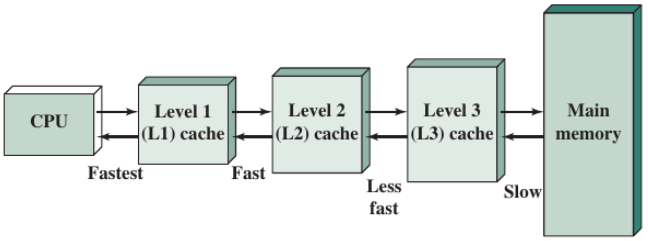
La memoria caché está diseñada para combinar el tiempo de acceso a memoria de memoria costosa y de alta velocidad con el gran tamaño de memoria de memoria menos costosa y de menor velocidad. El concepto se ilustra en la Figura 4.3a. Hay una memoria principal relativamente grande y lenta junto con una memoria caché más pequeña y más rápida. La caché contiene una copia de porciones de la memoria principal.

Cuando el procesador intenta leer una palabra de memoria, se realiza una verificación para determinar si la palabra está en la caché. Si es así, la palabra se entrega al procesador [esto sería un acierto de caché]. Si no [esto sería un fallo de caché], un bloque de memoria principal, consistente en un número fijo de palabras, se lee hacia la caché y luego la palabra se entrega al procesador. *Debido al fenómeno de localidad de referencia, cuando un bloque de datos se trae a la caché para satisfacer una sola referencia de memoria, es probable que haya futuras referencias a esa misma ubicación de memoria o a otras palabras en el bloque.*

La Figura 4.3b representa el uso de múltiples niveles de caché. La caché L2 es más lenta y típicamente más grande que la caché L1, y la caché L3 es más lenta y típicamente más grande que la caché L2.



(a) Single cache



(b) Three-level cache organization

Figure 4.3 Cache and Main Memory

Entonces, ¿por qué funciona la caché?

El uso de la memoria caché se sustenta en dos principios o propiedades que exhiben los programas:

1) **Principio de <u>localidad espacial</u> de referencia**: cuando se accede a una palabra de memoria, es 'muy probable' que el próximo acceso sea en la vecindad de la palabra anterior. La localidad espacial se sustenta en:
    a) Ejecución secuencial del código
    b) Tendencia de los programadores a hacer próximas entre sí variables relacionadas
    c) Acceso a estructuras tipo matriz ó pila

2) **Principio de <u>localidad temporal</u> de referencia**: cuando se accede a una posición de memoria, es 'muy probable' que un lapso de 'tiempo corto', dicha posición de memoria sea accedida nuevamente. La localidad temporal, se sustenta en:
    a) Formación de ciclos o bucles
    b) Subrutinas (Procedimientos o Funciones)
    c) Pilas

ANEXO: HABLAMOS DE CANTIDADES

A la hora de calcular la cantidad de direcciones y el tamaño de una memoria, es fundamental recordar qué tenemos que calcular, y cómo se llaman los "números redondos".

Si queremos saber **cuántas direcciones** puede llegar a referenciar un bus de direcciones, debemos preguntarnos por sus bits, ya que la cantidad la obtenemos así: 2^n (siendo n la cantidad de bits).

Por ello, si tenemos un bus de direcciones de 8 bits, podremos referenciar 2^8 (o sea, 256) celdas de memoria[11]. Con un bus de 14 bits, podremos referenciar 2^{14} celdas. Pero acá ya podemos hacer algo más. Recordemos que una potencia puede descomponerse así: $2^{14} = 2^{4+10} = 2^4 \cdot 2^{10} = 16 \cdot 2^{10}$. ¿Y para qué me sirve eso? Messirve porque 2^{10} tiene un nombre conocido: "k".

Entonces, cuando nos pregunten **cuántas direcciones** puede referenciar un bus de 14 bits, podemos responder: puede referenciar hasta 16k celdas de memoria.

Y si nos preguntan **de qué tamaño** sería esa memoria, les decimos: no lo sabemos. Si no nos dicen el tamaño de las celdas (o posiciones de memoria), no podremos saber qué tamaño total tendrá la memoria, pero sí la cantidad de celdas que puede llegar a referenciar.

Si en cambio nos dijeran: ¿qué tamaño de memoria puede referenciar un bus de 14 bits cuyas posiciones de memoria son de 1 byte? Ahí sí, les decimos: la memoria tendrá un tamaño de 16KB o 16Kbytes.

Los otros "números redondos" son:

$2^{10} = K$, de kilo.

$2^{20} = M$, de mega.

$2^{30} = G$, de giga.

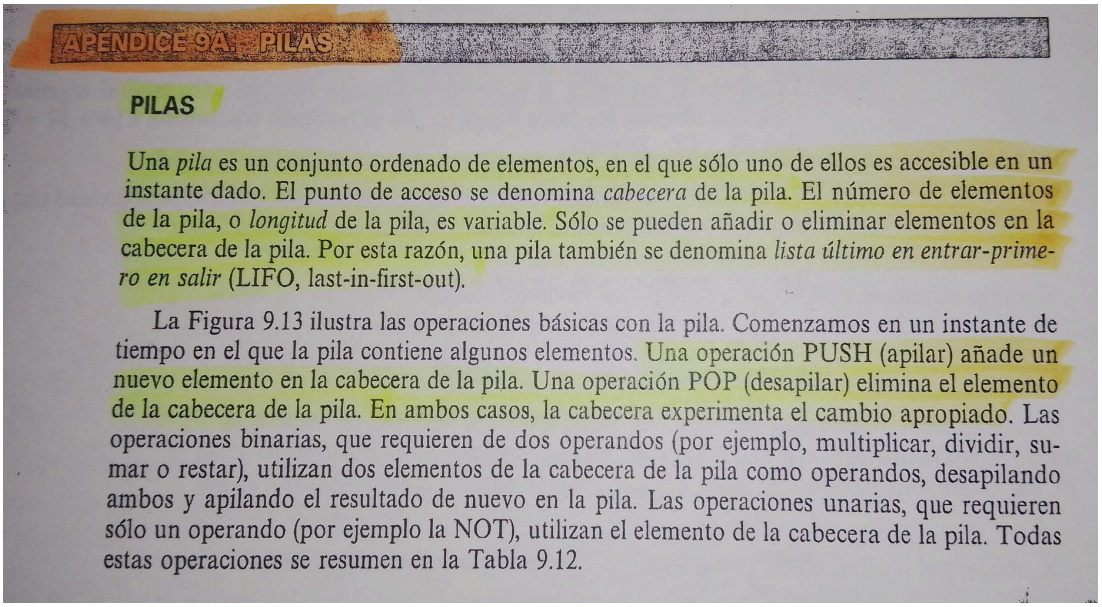
$2^{40} = T$, de tera.

[11] No estoy siendo muy preciso con la terminología. En este contexto celda de memoria, dirección de memoria y posición de memoria son prácticamente sinónimos. Pero quizás lo más correcto sería hablar de direcciones o posiciones de memoria.

ANEXO 2: PILA PILA

Hablando de los formatos de instrucción, y más precisamente sobre el número de direcciones, STALIN'S dice:

> Los formatos de instrucción de **tres direcciones** no son comunes porque requieren un formato de instrucción relativamente largo para contener las tres referencias de dirección. Con las instrucciones de **dos direcciones**, y para operaciones binarias, una dirección debe cumplir una doble función como operando y como resultado. Por tanto, la instrucción SUB Y, B lleva a cabo el cálculo Y - B y almacena el resultado en Y. El formato de dos direcciones reduce el requisito de espacio pero también introduce cierta incomodidad. Para evitar alterar el valor de un operando, se usa una instrucción MOVE para mover uno de los valores a una ubicación de resultado o temporal antes de realizar la operación. Nuestro programa de ejemplo se expande a seis instrucciones.
>
> Aún más simple es la instrucción de **una dirección**. Para que esto funcione, una segunda dirección debe ser implícita. Esto era común en máquinas anteriores, siendo la dirección implícita un registro del procesador conocido como acumulador (AC). El acumulador contiene uno de los operandos y se usa para almacenar el resultado. En nuestro ejemplo, se necesitan ocho instrucciones para realizar la tarea.
>
> De hecho, es posible arreglárselas con **cero direcciones** para algunas instrucciones. Las instrucciones de cero direcciones son aplicables a una organización especial de memoria llamada pila (*stack*). Una pila es un conjunto de ubicaciones que funciona como "último en entrar, primero en salir" (*last-in-first-out*). La pila está en una ubicación conocida y, a menudo, al menos los dos elementos superiores están en registros del procesador. Por lo tanto, las instrucciones de cero direcciones harían referencia a los dos elementos superiores de la pila.



APENDICE 9A PILAS

PILAS

Una *pila* es un conjunto ordenado de elementos, en el que sólo uno de ellos es accesible en un instante dado. El punto de acceso se denomina *cabecera* de la pila. El número de elementos de la pila, o *longitud* de la pila, es variable. Sólo se pueden añadir o eliminar elementos en la cabecera de la pila. Por esta razón, una pila también se denomina *lista último en entrar-primero en salir* (LIFO, last-in-first-out).

La Figura 9.13 ilustra las operaciones básicas con la pila. Comenzamos en un instante de tiempo en el que la pila contiene algunos elementos. Una operación PUSH (apilar) añade un nuevo elemento en la cabecera de la pila. Una operación POP (desapilar) elimina el elemento de la cabecera de la pila. En ambos casos, la cabecera experimenta el cambio apropiado. Las operaciones binarias, que requieren de dos operandos (por ejemplo, multiplicar, dividir, sumar o restar), utilizan dos elementos de la cabecera de la pila como operandos, desapilando ambos y apilando el resultado de nuevo en la pila. Las operaciones unarias, que requieren sólo un operando (por ejemplo la NOT), utilizan el elemento de la cabecera de la pila. Todas estas operaciones se resumen en la Tabla 9.12.

Tabla 9.12. Operaciones orientadas al uso de la pila

PUSH	Añade un nuevo elemento en la cabecera de la pila.
POP	Elimina el elemento de la cabecera de la pila.
Operación unaria	Realiza una operación con el elemento de la cabecera de la pila. Sustituye el elemento de la cabecera con el resultado.
Operación binaria	Realiza una operación con dos elementos de la cabecera de la pila. Elimina de la pila dichos elementos. Pone el resultado de la operación en la cabecera de la pila.