Resumen Organización del Computador 1

Logica Digital

Conceptos clave

- Logica combinatoria
- Logica secuencial
- Álgebra de bool

Cuando trabajamos con modelos computacionales, debemos entender en qué nivel de abstracción se encuentra nuestro proceso. La *lógica digital se encuentra* en el nivel 0 (tenemos 6 niveles).
Las computadoras que utilizamos actualmente se basan en el procesamiento de información en formato binario.
Como trabajamos a partir del modelo de Von Neumann - Turing, los programas que cargamos se encuentran alocados en memoria.

Algebra de bool

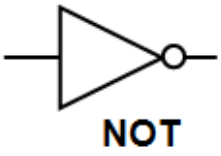
Como mencionamos anteriormente, las computadoras al trabajar en formato binario, cada una de sus operaciones está formalizado a partir de valores de verdad. 1 = True, 0 = False. Esta rama de la matemática recibe el nombre de *lógica simbólica*.

A partir de una serie de compuertas lógicas básicas, podemos construir universalmente cualquier expresión lógica.
-> AND ($\wedge$)
-> OR ($\vee$)
-> NOR ($\neg$)

La composicion de estos operadores logicos da lugar a otras operaciones como XOR, NAND, NOR, etc.

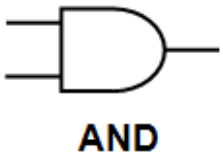
Circuitos logicos

Electricamente, estas compuertas lógicas se implementan a partir de la composición de transistores con un voltaje en particular, donde señales de alrededor de los 5V (True) y 0V (False). Para cada una de nuestras compuertas lógicas tenemos una *tabla de verdad* asociada para los valores de entrada que cada una puede recibir.



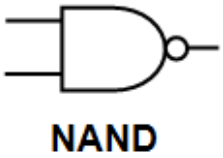
NOT

Input	Output
I	F
0	1
1	0



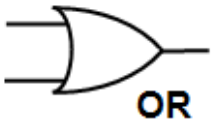
AND

Inputs		Output
A	B	F
0	0	0
1	0	0
0	1	0
1	1	1



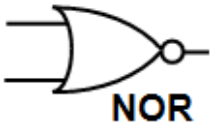
NAND

Inputs		Output
A	B	F
0	0	1
1	0	1
0	1	1
1	1	0



OR

Inputs		Output
A	B	F
0	0	0
1	0	1
0	1	1
1	1	1



NOR

Inputs		Output
A	B	F
0	0	1
1	0	0
0	1	0
1	1	0

EXCLUSIVE OR

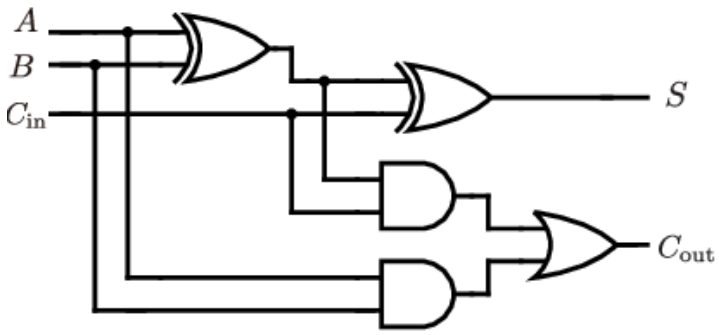
Inputs		Output
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

EXCLUSIVE NOR

Inputs		Output
A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

Un ejemplo de *circuito combinacional*, es decir, obtener un resultado a partir de la composición de compuertas lógicas, es un *full adder*.

- El full adder es una herramienta utilizada para sumar de a dos bits, recibimos dos valores booleanos y devolvemos el resultado de realizar la suma junto al *carry-out* respectivo.
- En la implementación del full adder consideramos la posibilidad de haber recibido C_{in} de otra operación.
- Si componemos este *full adder* n veces estaríamos haciendo un n-bit adder o *ripple carry adder*.



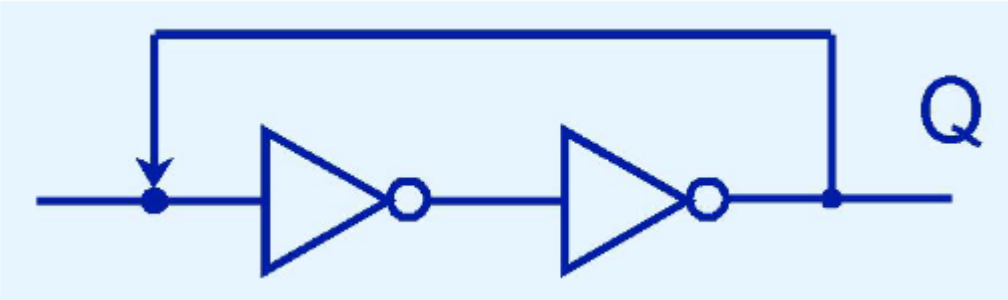
Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Al complejizar nuestra composición de circuitos lógicos, empezamos a crear componentes fundamentales en el funcionamiento de nuestras computadoras como:
-> ALU.
-> Multiplexores y demultiplexores.
-> Memoria ROM

Circuitos secuenciales

Como nuestras computadoras trabajan a partir de *estados*, es decir, momentos de tiempo t tal que la computadora se encuentra en un conjunto de valores X, tenemos que poder crear circuitos que nos permitan **almacenar** y **acceder** a la informacion en cualquier momento que necesitemos.
-> *Clock* (nos permite fijar una señal para sincronizar las operaciones que hagamos en nuestra computadora).

Para poder almacenar esa información a partir de los *tick* que nos da el clock, necesitamos un dispositivo que se **retroalimente** y mantenga esos valores guardados mientras necesitemos.



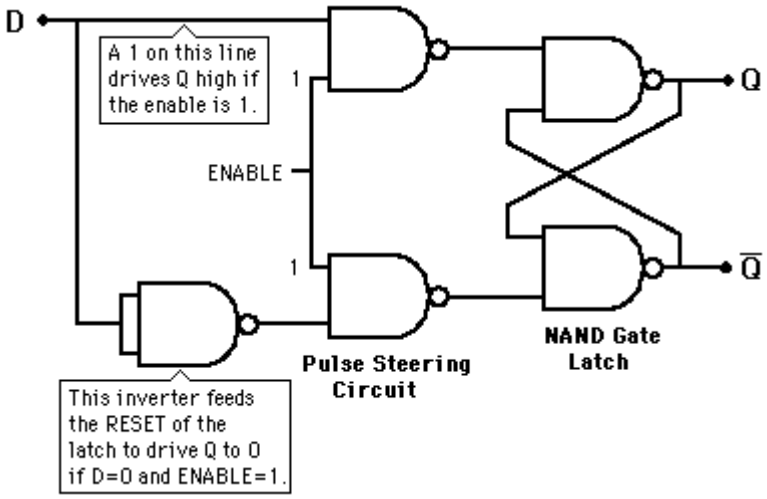
Flip-Flop
-> RS (set-reset)

- Si S = 1, Q se setea en 1 (set)
- Si R = 1, Q se setea en 0 (reset)

- $Q(t)$ es el valor de salida en un momento t => $Q(t+1)$ es el valor de Q en el siguiente clock
- *problema*: se indefine si S $\wedge$ R = 1

-> D

- Como el problema de indefinición de RS genera problemas en el almacenamiento, solucionamos este inconveniente simplificando nuestra implementación.
- Si tenemos una única entrada *D*, solamente guardamos el último valor que pasa por D.



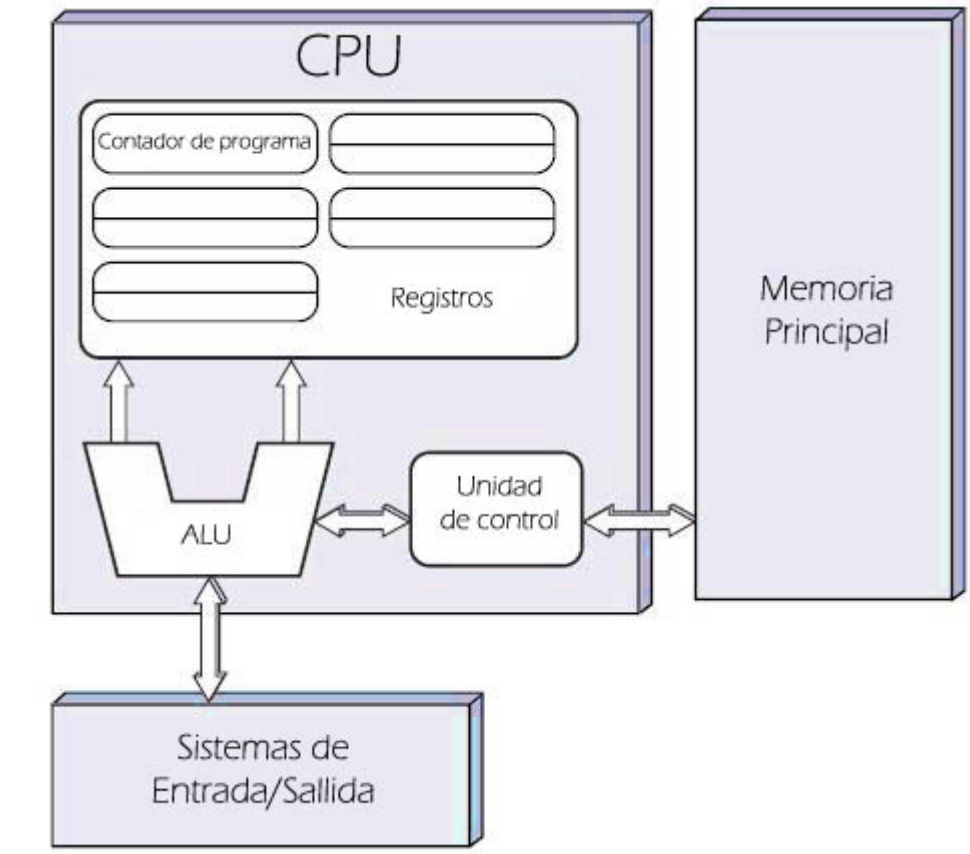
Cuando tratamos con *registros* en una arquitectura, simplemente hacemos referencia a un conjunto de flip-flops para guardar mis valores. La cantidad de bits que se quiere guardar dependerá de la función del registro.

Arquitectura del CPU - ISA - Modos de direccionamiento

Conceptos clave

- Ciclo de Instrucción
- ISA
- Tipos de arquitectura

Recordamos que nos encontramos bajo el modelo de arquitectura de Von Neumann-Turing.



-> Nuestra computadora se compone de:

- CPU
  -> ALU (Arithmetic-Logic Unit)
  -> Registros (incluyendo el Program Counter)
  -> Unidad de Control
- Memoria Principal
- Sistema de Entrada/Salida

Propiedades de una ISA

Cuando se crea una arquitectura, hay que tener ciertas consideraciones sobre:
-> Complejidad de el conjunto de instrucciones

- RISC (reduced instruction set computer)
- CISC (complex instruction set computer)

-> Longitud de las instrucciones

- 16 bits
- 32 bits
- 64 bits

-> Formato de representación

- Big endian vs Little endian

-> Tipo de arquitectura

- Arquitectura de Stack
- Arquitectura de Acumulador
- Arquitectura con registros de propósito general

Dentro de este modelo de cómputo, la idea general es *traer instrucciones que tenemos cargadas en la memoria de la CPU* o bien *ejecutar esas instrucciones que tenemos guardadas*.

- En cada uno de estos pasos, nos encontramos en una instancia del ciclo de *Fetch - Decode - Execute* o Ciclo de Instrucción.

Etapas del ciclo de instrucción(IC)

Fetch

- Una vez que tenemos un conjunto de instrucciones cargadas, el Program Counter apunta a la dirección de la siguiente instrucción la cual, en este caso, nos interesa traer de la memoria principal para luego ejecutar.
- La ALU incrementa el PC.

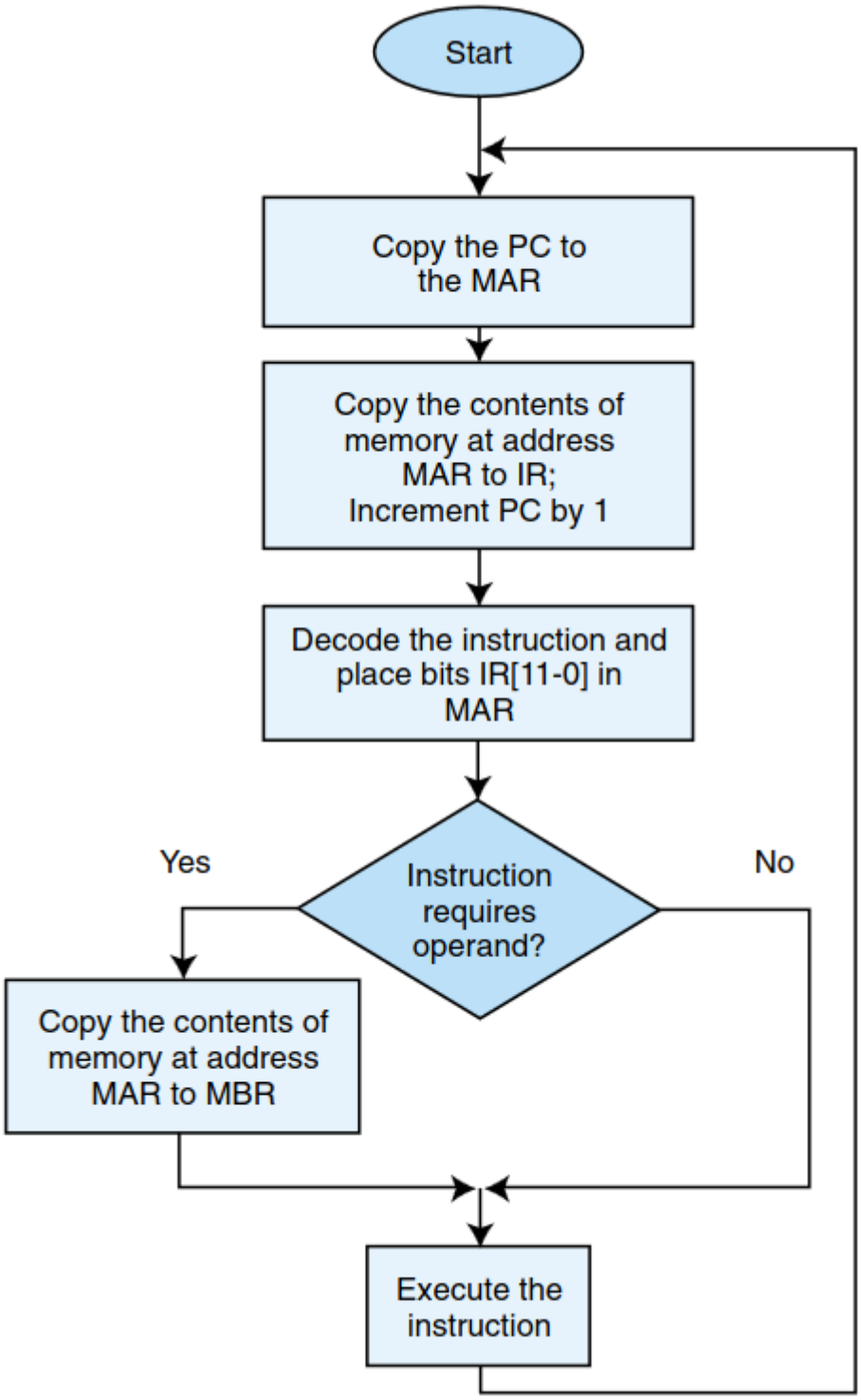
Decode

- Habiendo traido la "tira" con las instrucciones en la etapa del fetch, es necesario decodificarla para entender esa instruccion.
- Cada instrucción posee ciertas operaciones y operandos (datos con los que trabajamos). Una vez que tenemos estos operandos nos interesa que la UC los guarde en los registros para luego operar con ellos (o no).

Execute

- Con toda la información guardada en los registros que queremos, la ALU se encarga de realizar las operaciones con esos valores.
- La información luego se guarda en memoria.

Observamos que todo este *ciclo* se repite continuamente.



En cada etapa del ciclo de instrucción, es importante entender que el encargado de *orquestar* todos estos movimientos es la **Unidad de Control**.
-> Funciona como nuestro puente entre el nivel 0 y nivel 2.

Pregunta clave: ¿Qué registros se utilizan en cada ciclo?

-> *Memory Address Register (MAR):* Almacena la posición de memoria de la instrucción que se quiere ejecutar.

-> *Instruction Register (IR)*: Busca en la memoria principal la instrucción que se encuentra almacenada en el *MAR* (la cual fue guardada del Program Counter).

-> *Memory Buffer Register (MBR)*: Almaena la posición de memoria de la siguiente instrucción a ejecutar para luego guardarla en el PC.

Por otro lado también es necesario entender de qué forma se vinculará cada uno de nuestros componentes entre sí.

Register Transfer Language (RTL) y Datapath

- Cuando conectamos todos nuestros componentes entre sí, necesitamos tener algún sistema de notación que nos facilite *especificar* qué operaciones queremos realizar de forma más legible.
- Si queremos sumar dos números, en vez de explicitar cada paso a realizar, utilizamos la instrucción *ADD* para sumar dos números.
- Cada instrucción tiene "mini-instrucciones" -> *Microinstrucciones*

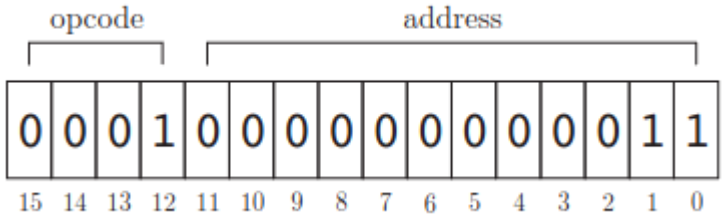
Etapa	Tipo de instrucción	acciones
IF	todas	IR <- mem[PC] PC <- PC + 4
ID	todas	A <- R[Rs] B <- R[Rt] ALUout <- PC + (inm16 < <2)
EX	tipo R Load/Store Branch Jump	ALUout <- A op B ALUout <- A + signextend(inm16) if(A==B) then PC <- ALUout PC<31:2> <- PC<31:28>, IR<25:0> < < 2

Acceder a nuestras instrucciones requiere un formato de direccionar a cada una de las mismas.
Ej: la notación para devolver lo que se encuentra *dentro* de un registro no es igual a la notación para devolver *la posición* en memoria de aquel registro

Modos de direccionamiento

-> ***Instruccion***: consiste en una tira de numeros binarios con

- Codigo de operacion
- Valores/Operandos -> Los operandos pueden ser *inmediatos*, *registros*, *arreglos*.

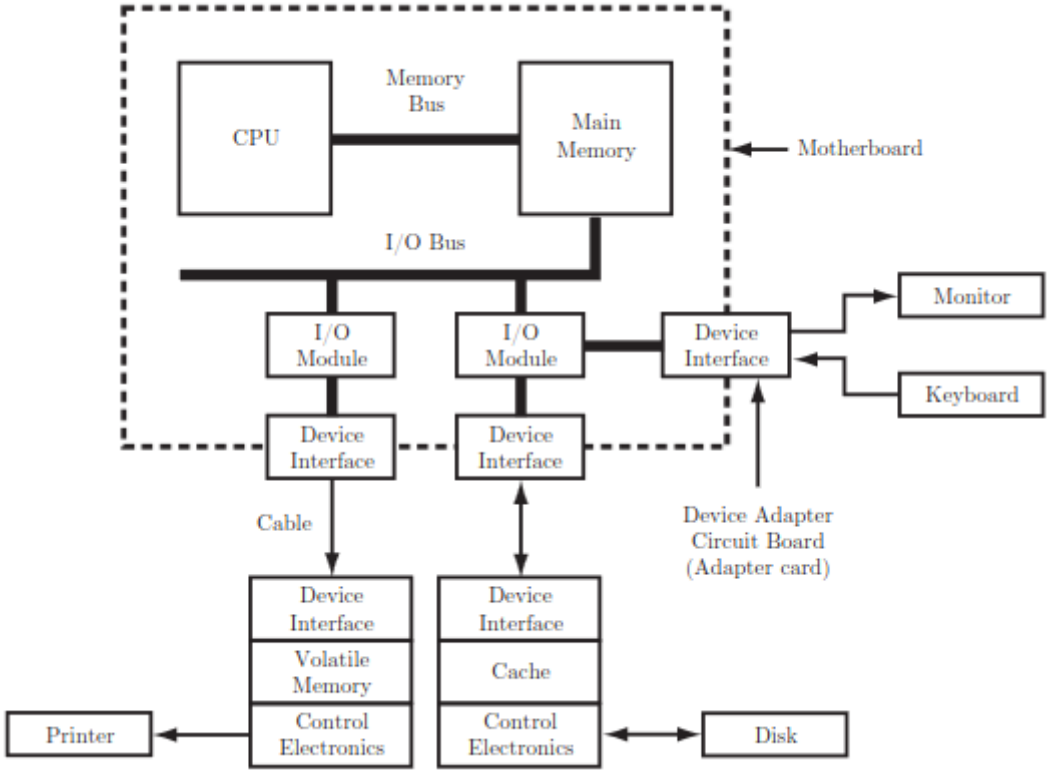


-> ***Inmediato***: constante
-> ***Directo***: contenido de una posicion de memoria especifica -> [0xFF] = 0xAB
-> ***Indirecto***: dado un puntero A, [[A]] busca primero el valor en una determinada posicion y luego accede al contenido de una registro con el primer valor obtenido.
-> ***Registro***: el operando es un registro dentro de la CPU.
-> ***Registro indirecto***: se accede al contenido del registro -> [Rn]
-> ***Desplazamiento***: se accede al contenido de un registro con un desplazamiento D *relativo a la posicion del registro* -> [Rn + D].

Entrada y Salida (I/O)

Conceptos clave

- *Polling*
- *Interrupciones*
- *Interruption Controller*
- *Mapeo a memoria*
- *Mapeo a registros*
- En nuestra arquitectura de Orga1 tenemos un módulo que se encarga del manejo y comportamiento de la información de entrada y salida.



Dentro de la memoria tenemos tres tipos de registro:

-> ***Registros de lectura**:* dedicados a almacenar información de los dispositivos de entrada.

-> ***Registros de escritura***: dedicados a almacenar información de los dispositivos de salida.

-> ***Registros de lectura/escritura***: almacenan la información de ambas señales.

- En la arquitectura Orga1 dedicamos posiciones en particular para dispositivos de entrada/salida (desde 0xFFF0 - 0xFFFF). => [0x0000, (...) , 0xFFF0, (...) , 0xFFFF]

¿Cómo sabemos si recibimos entrada o salida?

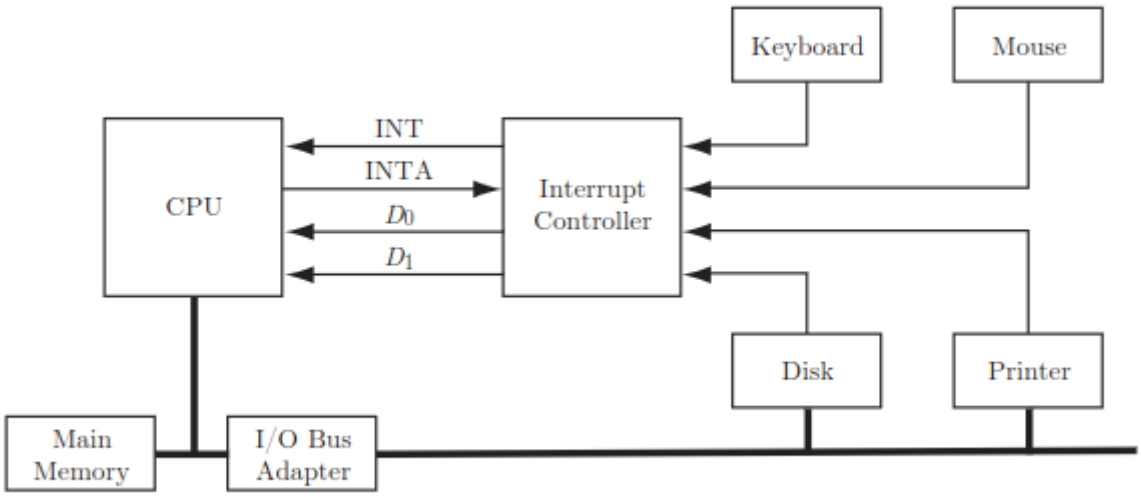
-> ***Polling***: me pregunto cada n segundos si tengo alguna interrupcion de entrada.

- Me cuesta más caro repetir siempre la misma operación periódicamente. (Contra)
- Solo me doy cuenta que llegó una interrupción cada n segundos, si necesitara una interrupción antes no me enteraría. (Contra)
- No necesito pausar todo lo que estoy haciendo para manejar la interrupción (Pro)

-> ***Interrupciones***: hago mis tareas normalmente y únicamente las interrumpo cuando recibo un aviso de que tengo una entrada (o salida) de información.

- Debo recordar el estado en el que dejó todo hasta el momento de la señal (guardo los flags, el valor de los registros y el estado de ejecución hasta la interrupción).
- Cuando termino de atender mi interrupción vuelvo a cargar mi estado anterior a la interrupción.

Subsistema de E/S con interrupciones



- *INT*: el Interrupt Controller envía una señal al CPU para interrumpir (porque tenemos una entrada) y leer los datos de entrada.
- *INTA*: el Interrupt Acknowledge devuelve el "reconocimiento" por la CPU de que hay una interrupción y está preparado para recibirla.
- D_0/D_1: envío los datos de información del controlador de interrupciones a la CPU.

-> El *INTA* sigue el protocolo similar al "handshake" dentro de una red.
-> Agregamos a nuestra unidad un nuevo flag (I,Z,N,C,V) => I = "interrupt".

Mapeo a Registros

- La información que recibimos de I/O tenemos que poder almacenarla en algún lugar dentro de la memoria.

-> *en espacio de memoria (general)*: no tengo restricción sobre donde guardar la información de entrada y salida.

- Puede ser problemático porque una instrucción puede llegar a pisarme alguna posición de memoria que contiene información de otra ejecución.

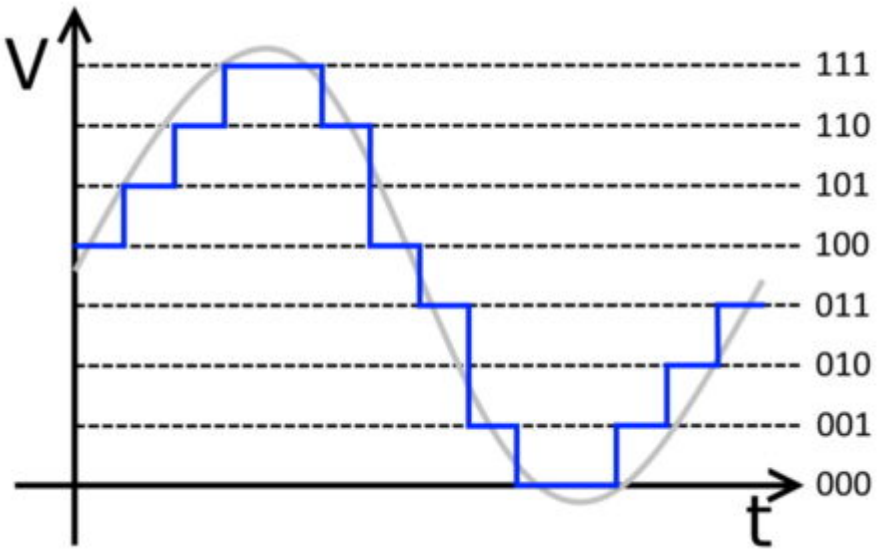
-> *en espacio de I/O*: como tenemos registros dedicados a entrada y salida, podemos guardar el contenido en esas posiciones de memoria asegurándome menor probabilidad de pisado de información incorrecto. (OxFFF0 -> 0xFFFF).

Conversión A/D D/A

Cuando recibimos señales a partir de nuestros dispositivos de *entrada* o *salida*, en muchos casos no recibimos la información en el formato que la computadora puede operar.

-> ***Señal analógica***

- Es un tipo de señal que se genera a partir de algún fenómeno (la voz, un sensor de temperatura, la luz) que tiene ***amplitud*** y ***frecuencia*** las cuales pueden ser representadas por una función continua $f(t)$.
- Como la computadora opera en *binario* necesitamos poder convertir esta señal al formato digital.
- *Problema:* dado que una función continua es de valores infinitos, tenemos que encontrar algún modo de "discretizar" esos valores -> *valores azules*.



-> ***Señal digital***

- Tiene valores dentro de un rango numérico y en instantes discretos de tiempo t y amplitud.

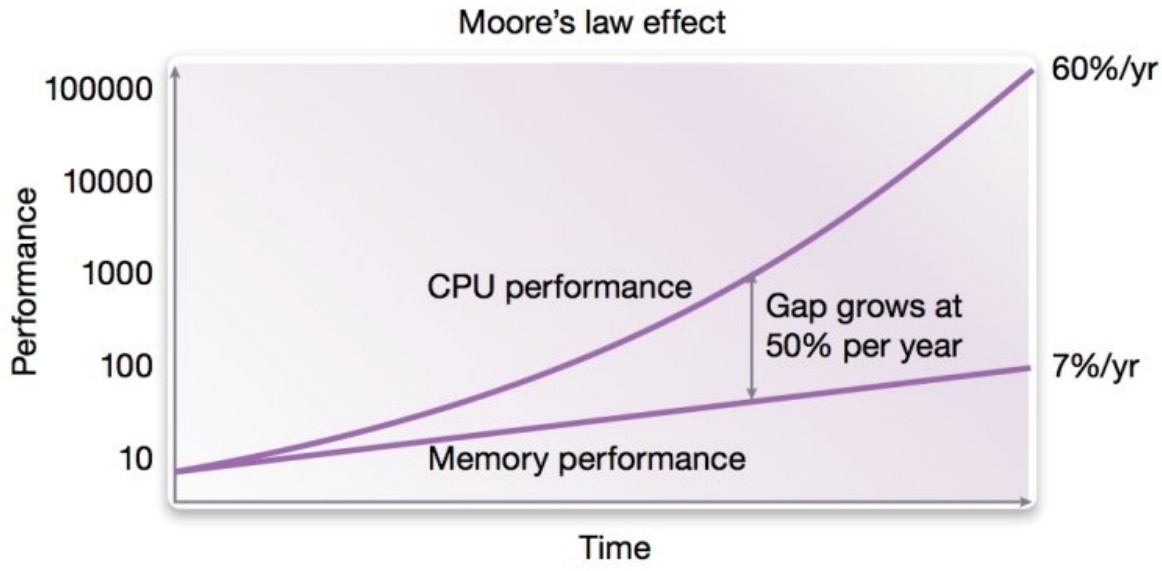
¿Cómo digitalizamos una señal analógica a digital?

Teorema del muestreo y muestreo en amplitud

- **Muestreo en el tiempo**: dada una *señal analógica* $x(t)$, con una frecuencia máxima $F_{\text{máxima}}$ y la *frecuencia de muestreo* $F_{\text{muestreo}} = \frac{1}{\Delta t}$
- **Teorema del muestreo**: $F_{\text{muestreo}} >= 2F_{\text{máxima}}$
- **Muestreo en amplitud**: dado un *rango dinámico* (x_{min}, x_{max}), entonces $\Delta x = \frac{x_{max} - x_{min}}{N}$
- **N**: se determina con la *cantidad de bits* del conversor analógico-digital $N = 2^{bits}$
- **Ejemplo**: un CD de música $F_{maxima} = 20\,\text{KHz}$ es muestreado en el tiempo a $F_{muestreo} = 44\,\text{KHz}$ y en amplitud con 16 bit, es decir $N = 65536$

Memoria y caché

- Con el paso del tiempo, el rendimiento de los procesadores fue avanzado en términos exponenciales mientras que, las memorias, por el contrario, avanzaban linealmente con el tiempo.
- Se tuvo que venir con una solución para poder acompañar el avance de la velocidad en los procesadores que sea eficiente en términos de costos y de rendimiento.



- Esta solución consistió en introducir un tipo de memoria especial llamada *caché* la cual se encarga de almacenar la información *frecuentemente utilizada* en términos temporales y espacialmente.

Memoria

Tenemos dos tipos de memoria principales.
-> *RAM (random access memory)*: se encarga del almacenamiento de programas e información necesaria para correr otros programas.

- Una vez que se apaga la computadora se pierde la información guardada en la misma.
- Tenemos dos tipos de RAM principales utilizadas para construir la RAM total:
  -> *SRAM* (Static RAM):

  - Se caracteriza por ser eficiente en lectura y escritura.
  - Se construye a partir de biestables (6 transistores) que utilizan 3 transistores a máxima potencia y otros 3 de reserva.
  - Lectura directa y no destructiva, una vez que se lee la instrucción no se borra la información.
  - ***Desventaja**:* necesita muchos transistores y por ende genera altas temperaturas por el consumo de electricidad.

  -> *DRAM* (Dynamic RAM):
  - Mucho más eficiente el costo de almacenamiento bit a bit.
  - Requiere una lógica mucho más compleja y la lectura es destructiva.
  - Los capacitores necesitan ser recargados cada n nanosegundos.
  - ***Desventaja**:* tener que estar refrescando constantemente la carga hace la lectura sucesiva más lenta (acumulativamente). Alto tiempo de acceso.
  - ***Ventaja**:* Menor costo de fabricación. Pasamos de 6 transistores a 1 solo. Alta capacidad de almacenamiento.

-> *ROM (read only memory)*:

- Guarda información crítica para el funcionamiento del sistema operativo.
- Sirve únicamente para leer la memoria y no modificarla. Una vez apagada la computadora esa información se retiene.
- Una pequeña porción de la memoria general está dedicada para la memoria ROM.
- Tenemos distintos tipos de ROM:
  -> *PROM (programmable ROM)*
  -> *EPROM (electrically programmable ROM)*: permite el borrado de la información a partir de una iluminación ultravioleta.
  -> *EEPROM (electrically erasable ROM)*: no requiere de una herramienta especial para borrar la información, permite el borrado byte a byte.

Jerarquía de memorias

Cuando necesitamos acceder a un dato en memoria, la CPU deberá seguir un esquema de búsqueda basado en la "cercanía" al procesador. *(esta distancia se mide a partir de los ciclos que tenga que realizar para acceder a la información)*.

Esta aproximación recibe el nombre de *"memoria jerarquizada"*. Cuanto más rápida sea la memoria que accedamos, mayor será el costo de producir cada memoria.

Existe una determinada terminología para hablar de jerarquía de memoria.

-> *Hit:* La información buscada recide en el nivel que nos encontramos.

-> *Miss:* La información no está en el nivel que nos encontramos.

-> *Hit-rate*: #\AccesosAcertados / #\AccesosTotales. (por probabilidad de Laplace)

-> *Miss-rate:* 1 - Hit-rate (complemento)

-> *Hit-time*: tiempo que cuesta acceder a una posición del nivel.

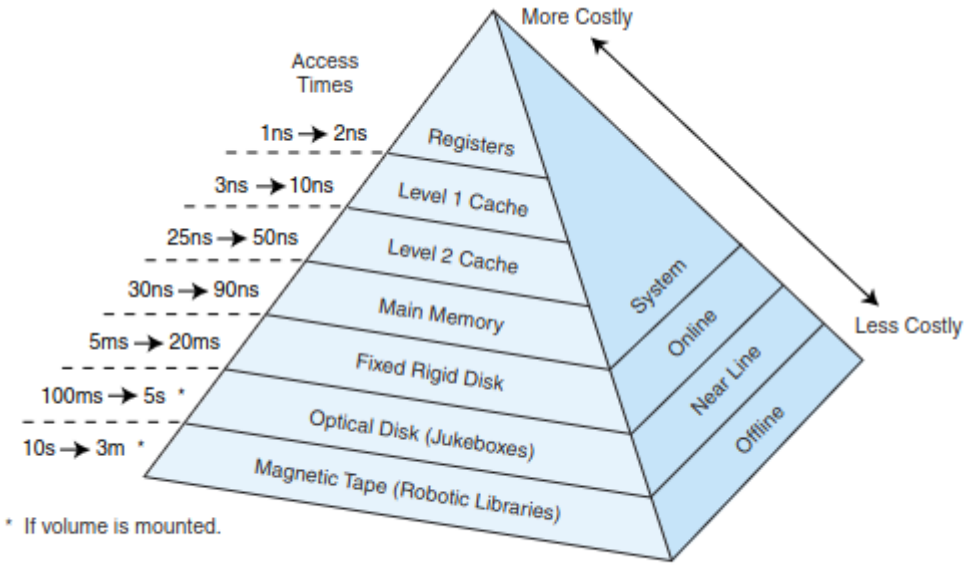


FIGURE 6.1 The Memory Hierarchy

Principio de vecindad temporal => Una forma *probabilistica* para mejorar el tiempo de acceso a un elemento es, asumiendo que un elemento X se accede en un tiempo t, entonces los elementos vecinos a X serán accedidos en un tiempo muy cercano a $t + t_0$ con $t_0 \to 0$.

Principio de vecindad espacial => Si una *posición* de memoria fue accedida en un momento t, entonces probablemente los elementos vecinos a esa posición serán accedidos en un momento cercano a $t + t_0$ con $t_0 \to 0$.

Memoria caché

- Ejemplo de motivación para una caché en página 237 a 238 *Linda Null computer-organization-and-architecture*. Un carpintero siempre querrá tener sus herramientas cerca para trabajar en vez de siempre ir a buscarlas al estante donde guarda todas sus herramientas cada vez que necesita usarlas.

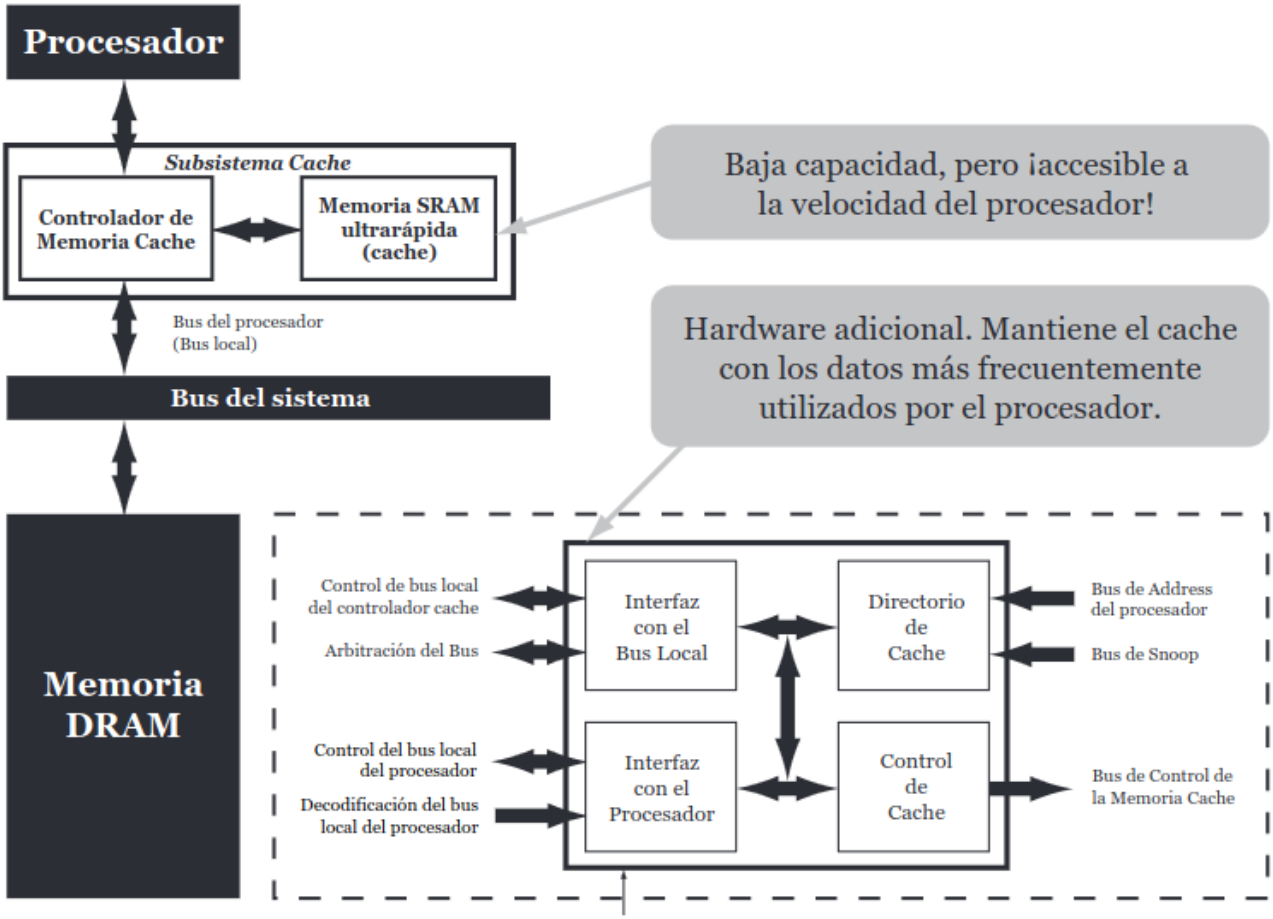
La memoria caché la encontramos dentro de la memoria principal. Esta memoria puede estar totalmente organizada o desorganizada, eso depende totalmente de quien programe el *mapping* de esta memoria.

Dentro de la memoria caché tenemos dos "niveles" o "tipos".

-> *L1's*: Es una memoria caché muy chica que reside dentro del procesador. Si bien en el modelo de Von Neumann se encuentra dentro de la memoria principal, en realidad su posición puede llegar a variar.

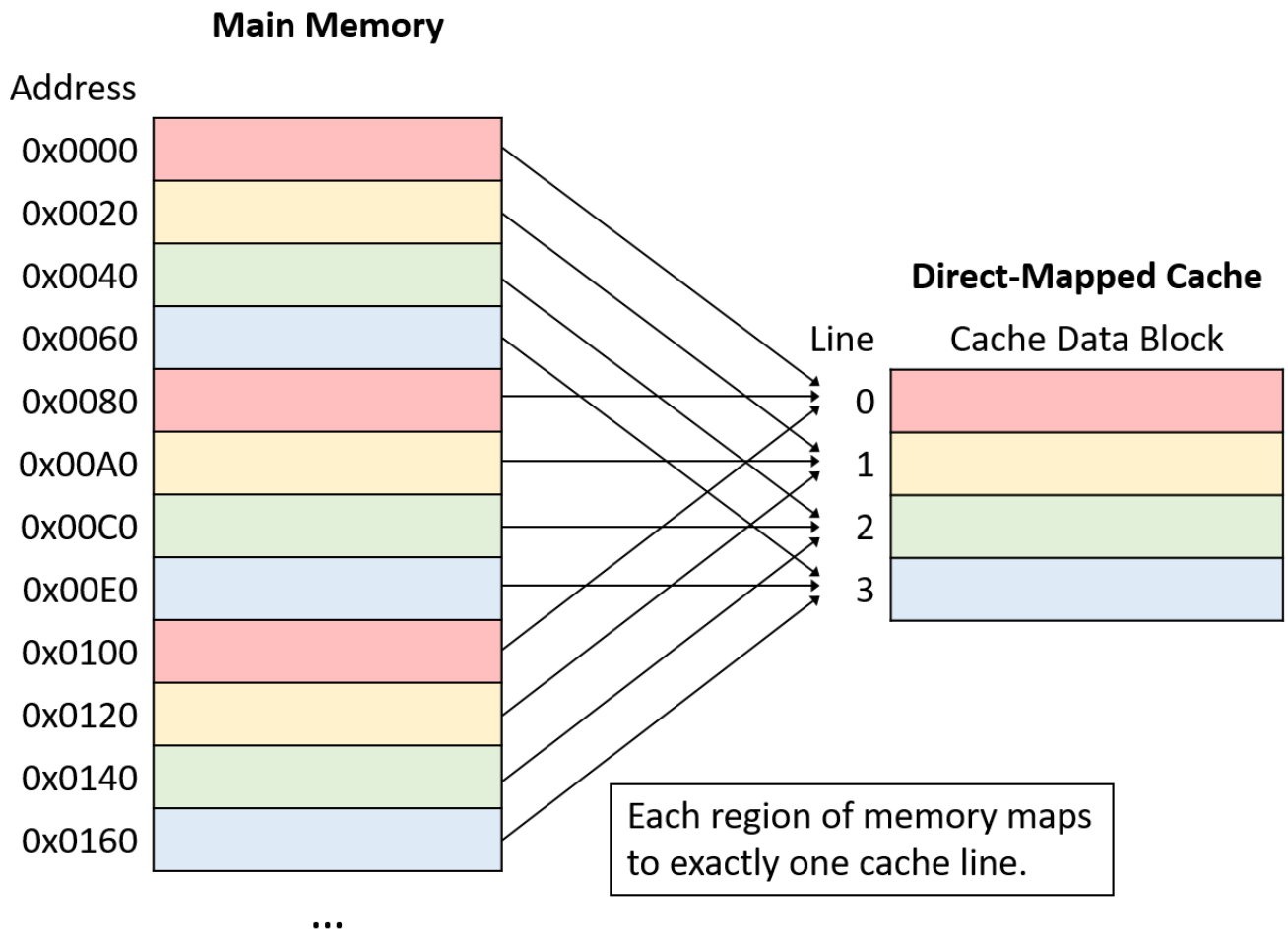
-> L2: Es una memoria que reside entre la CPU y la memoria principal.

- Una particularidad de la memoria caché es que no se accede por dirección sino por contenido. Si nos encontramos dentro de la caché, entonces aplicamos una *transformación* que lleva de memoria principal a *caché*.



Formas de mapeo a memoria caché

Direccionamiento directo



- Observamos que para cada elemento de la memoria principal le asignamos una dirección en la memoria caché a los primeros elementos => [memoriaPrincipal] mod |cache|.

Politicas de reemplazo en caché

- Cuando queremos reemplazar en una caché, dependiendo de nuestra implementación manejamos los reemplazos de una forma u otra (si es *Asociativa de dos vías* o *Totalmente asociativa*)
  -> LRU (*Last Recently Used*): borra de la memoria caché el último elemento utilizado.
  -> FIFO (*First In First Out*): primero que entra es el primero que sale. Similar a una cola de prioridad.
  -> LIFO (*Last In First Out*).
  -> Random (no recomendado).

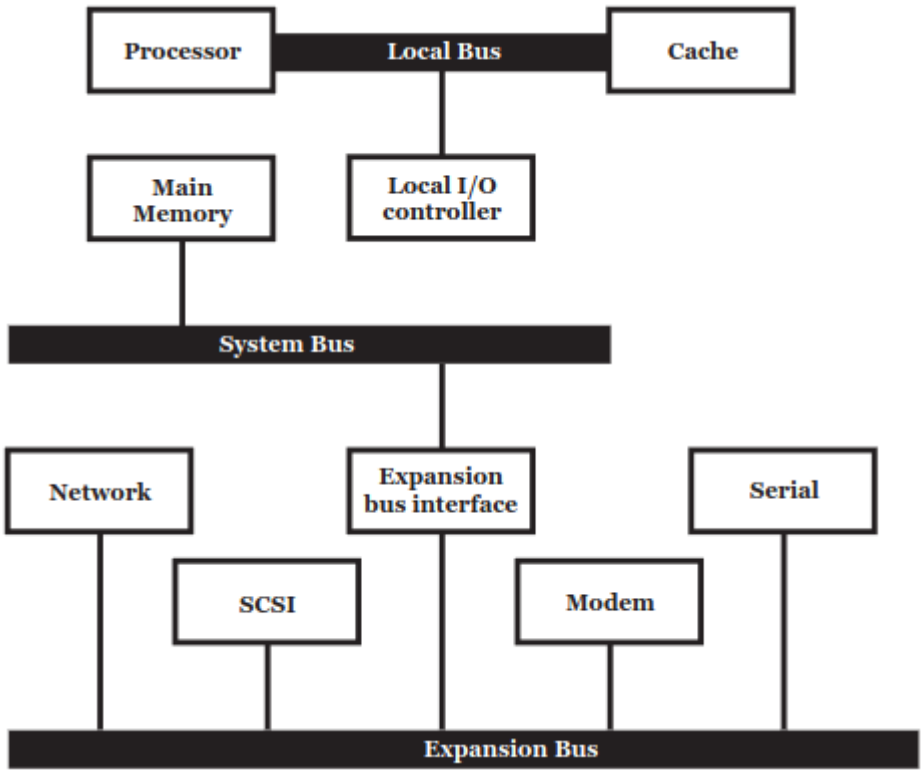
Buses y Almacenamiento

Queremos poder interconectar todos nuestros componentes que se encuentran en nuestro dispositivo. Introducimos a partir de esta necesidad el concepto de *buses*.

Buses

Un bus es un camino de comunicación entre uno o más dispositivos.

- Estos buses pueden portar distintos tipos de información y pueden tener distintas utilidades.



- Podemos hacer una analogía con una autopista. Tenemos carriles lentos, medios y rápidos.
- Cuanto más cerca estamos del procesador más rápido es el bus.
- El bus puede tener distintos tipos de señales:
  - *dirección* (permite intercambiar elementos puntuales de un subsistema)
  - *datos* (permite transferir datos de un lugar a otro)
  - *control* (permite el manejo efectivo de la información para no tener inconvenientes en la transferencia de datos o direcciones).

Local Bus -> System Bus -> Expansion Bus

Constantemente nos encontramos lidiando con el problema de la diferencia de velocidades. No solamente la velocidad es un factor limitante, sino el caudal de información que debemos pasar por un único bus de datos.
el modelo de Von Neumann plantea la existencia de un único bus de datos para enviar toda la información. --> ***cuello de botella de Von Neumann***

Si queremos que nuestro sistema sea escalable, debemos tener múltiples líneas donde cada una cumple una tarea en particular dentro de la transmisión de información.

Tipos de lineas

-> ***Lineas dedicadas***:

- *Dedicación física:* conectan siempre el mismo subconjunto de módulos (bus de E/S)
- *Dedicación funcional:* realizan la misma función (lineas de control del bus) menos disputadas por acceso al bus.

-> ***Lineas multiplexadas***:

- Propósitos diferentes en distrintos instantes del tiempo (bus de datos/direcciones según líneas de control)
- Menos líneas (más económico), circuitería más complejda, menos velocidad de la computadora.

Temporización

Tenemos que poder tener una coordinación entre los eventos del bús (una especie de *controlador de tráfico*). Si queremos realizar *lectura/escritura* tenemos que poder decidir cuando realizar cual.

-> ***Temporización sincrónica***

- Los eventos se coordinan a partir de los ticks del clock de la CPU.
- Todo evento determinado por el clock está determinada la cantidad en función de los ticks que ese clock realiza en total.

-> ***Temporización asincrónica***

- Eventos que siceden en el bus producen nuevos eventos, estos indican el flujo haciendo que el CPU se adecue a los dispositivos.
- Mejor rendimiento al haber dispositivos lentos y rápidos.
- Difícil de implementar.

Arbitraje

Dentro de la transferencia en el bus de datos, tenemos dispositivos que se encargan de iniciar las transferencias al bus (*activos/maestros*) y aquellos que la reciben (*pasivos/esclavos*).

Ejemplo:

Master	Slave	Ejemplo
CPU	Memoria	Fetch de instrucciones e información
CPU	I/O	Inicializar transferencia de datos
CPU	Coprocesador	Manejo CPU para envío de instrucción a coprocesador

Master	Slave	Ejemplo
I/O	Memoria	DMA (Direct Memory Access)

Este tipo de arbitraje se puede dividir en dos tipos:
-> ***Centralizado***: una única componente se encarga de determinar que dispositivo puede utilizar el bus.
-> ***Distribuido***: dentro de una computadora, varios dispositivos tienen salida al bus de datos. Si cada dispositivo tiene una prioridad distinta, cuando se hagan solicitudes para usar el bus, se deberán atender las mismas dependiendo la prioridad del *requester*. Esta prioridad se da a partir de que todos los dispositivos estén monitoreando las solicitudes del bus.
Una vez que se acepta la transferencia, se deberá esperar al siguiente clock para que otro dispositivo pueda utilizar el bus.