

Input / output

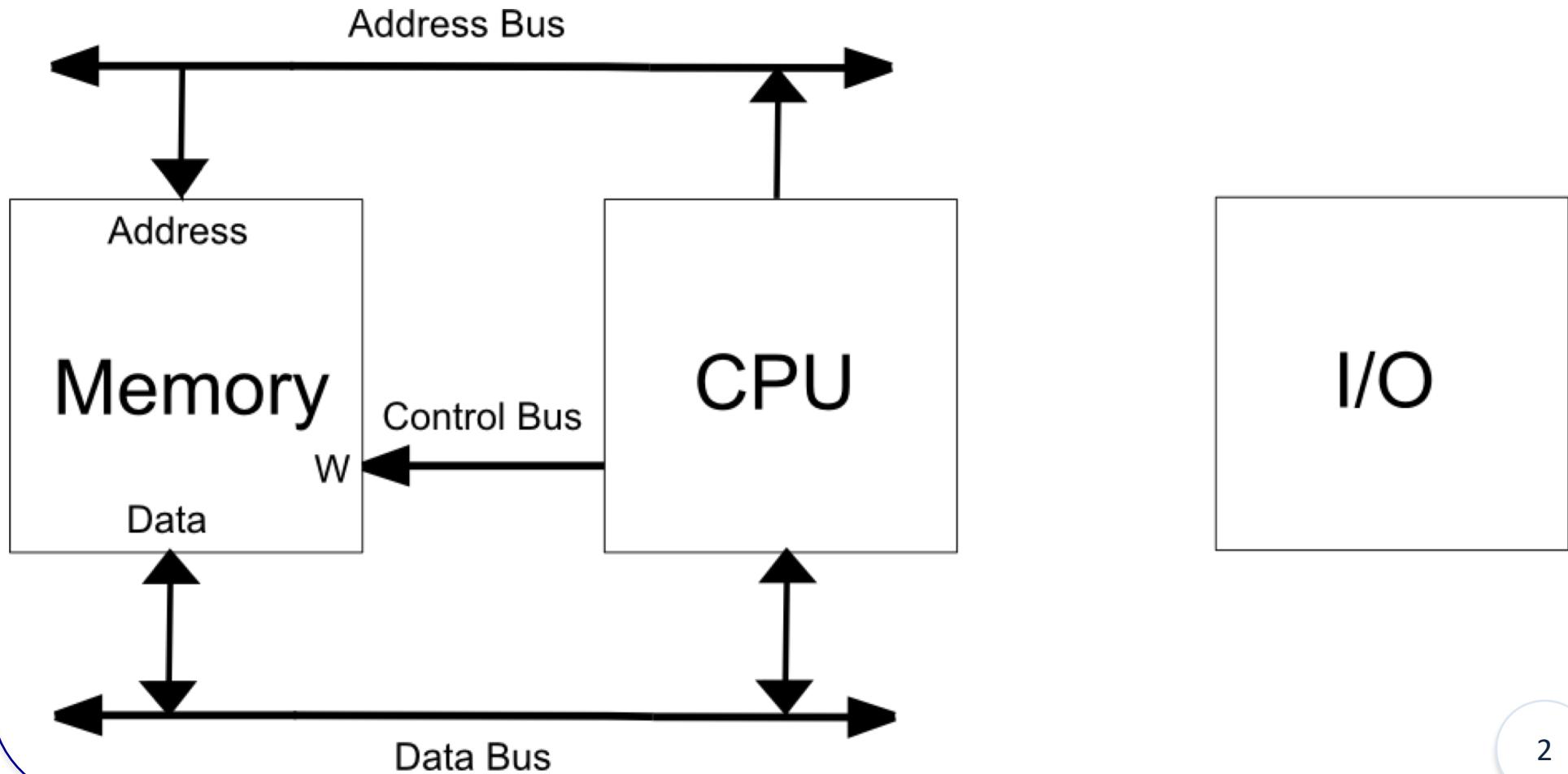
Arquitectura de Computadores – IIC2343

Un sistema computacional tiene tres componentes principales:

- CPU, memoria principal y secundaria, y **dispositivos de input/output (I/O)**

Dispositivos de I/O típicos:

- teclado, monitor, impresora, disco duro, *modem*, *mouse*, cámara, audífonos, micrófono, sensores, manejador de DVDs, linterna



Un procesador interactúa con un dispositivo de I/O de dos maneras:

1) El procesador controla el dispositivo:

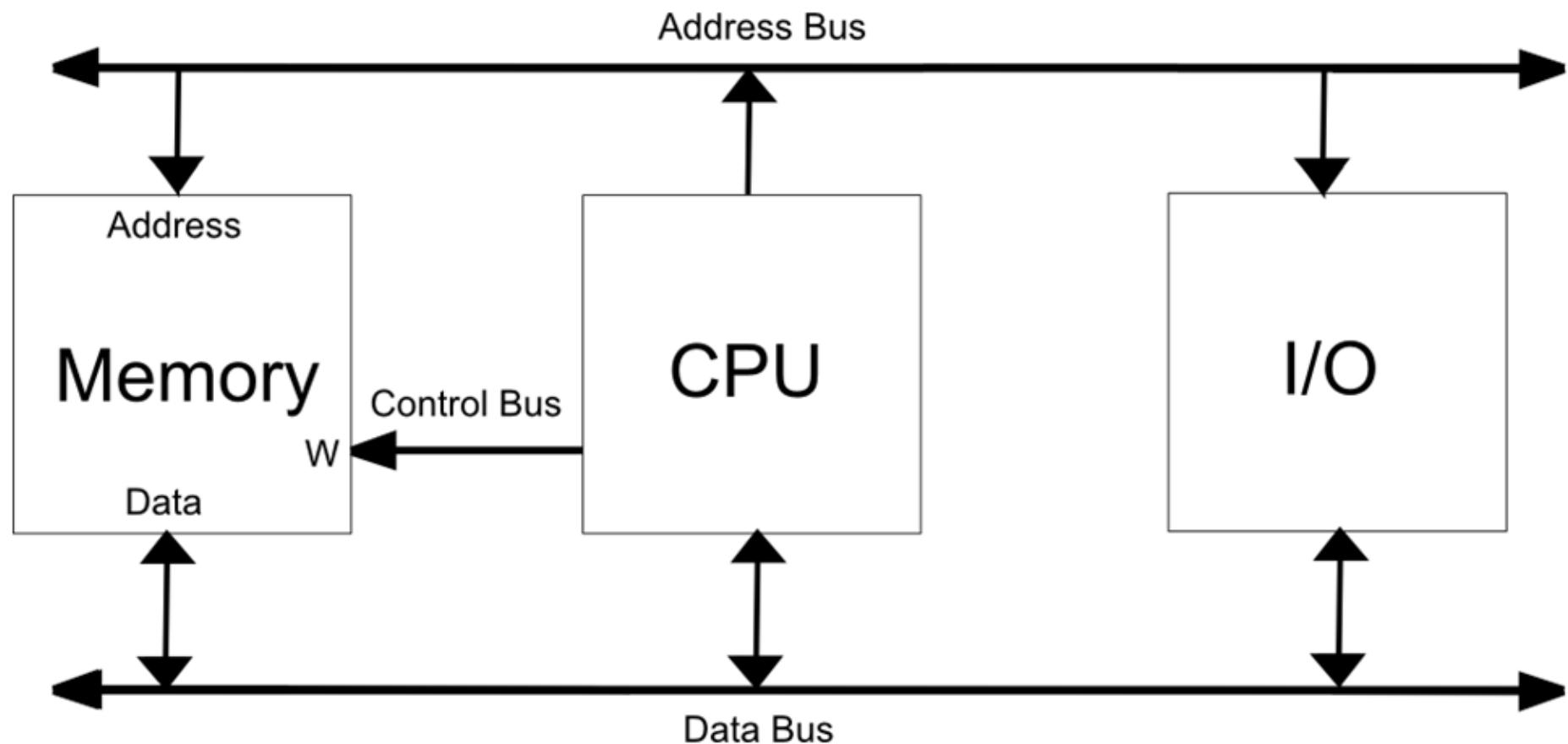
- p.ej., hace que el disco comience a girar, controla el volumen de un parlante, le dice a la cámara que tome una foto, apaga una impresora

2) El procesador intercambia datos con el dispositivo:

- es la función principal del dispositivo
- las decisiones arquitectónicas relativas a dispositivos de I/O apuntan a mecanismos que permitan el intercambio de datos con el procesador
 - ... p.ej., cómo se comunican los datos, quién inicia la comunicación (procesador o dispositivo), cómo se consiguen altas velocidades de transferencia

Todos los dispositivos que no sean la CPU o la memoria y que se comuniquen con ellos a través de los **buses** son llamados **dispositivos de I/O**

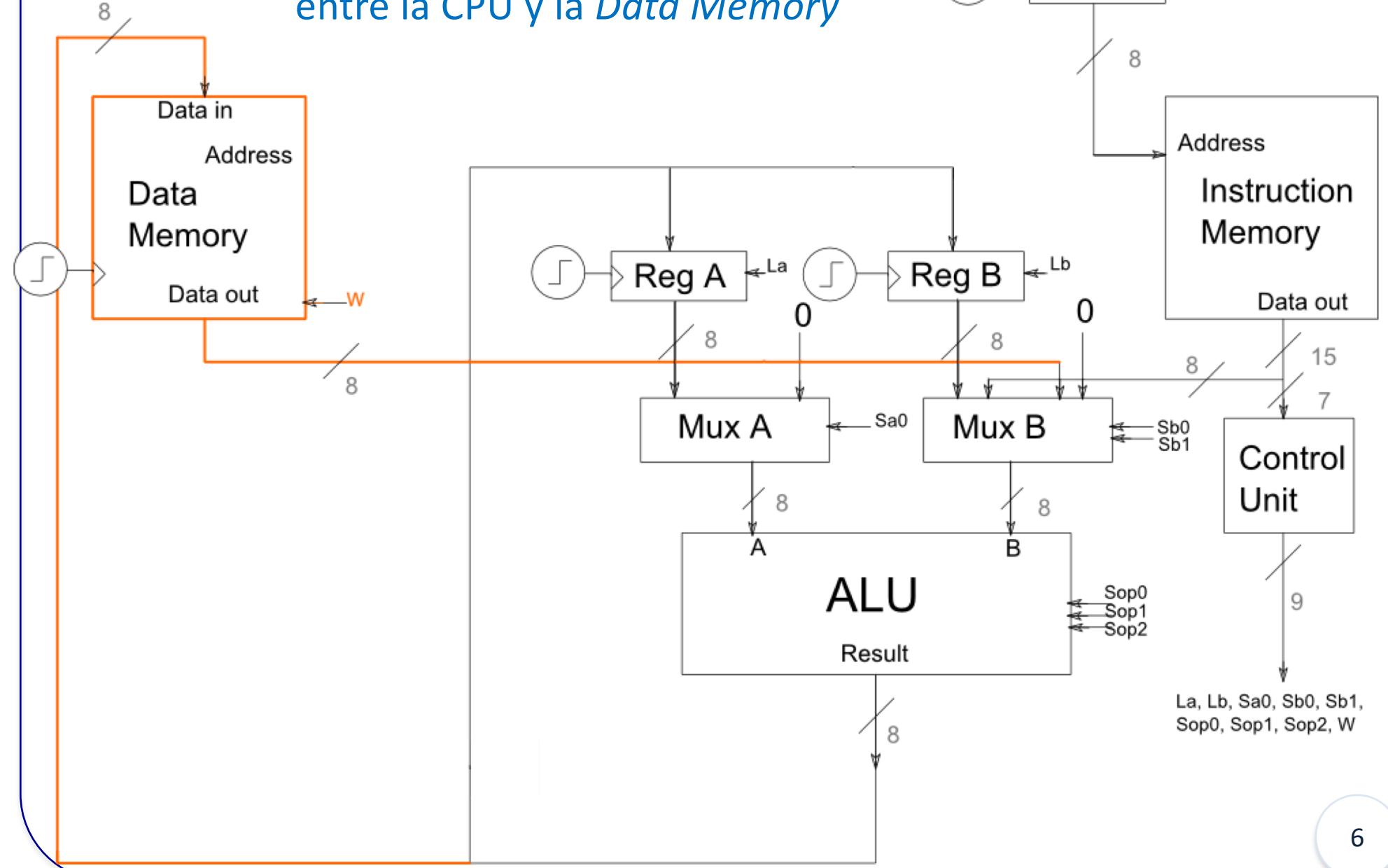
La conexión entre la CPU y un dispositivo de I/O usa el mismo paradigma básico que la conexión entre la CPU y la memoria



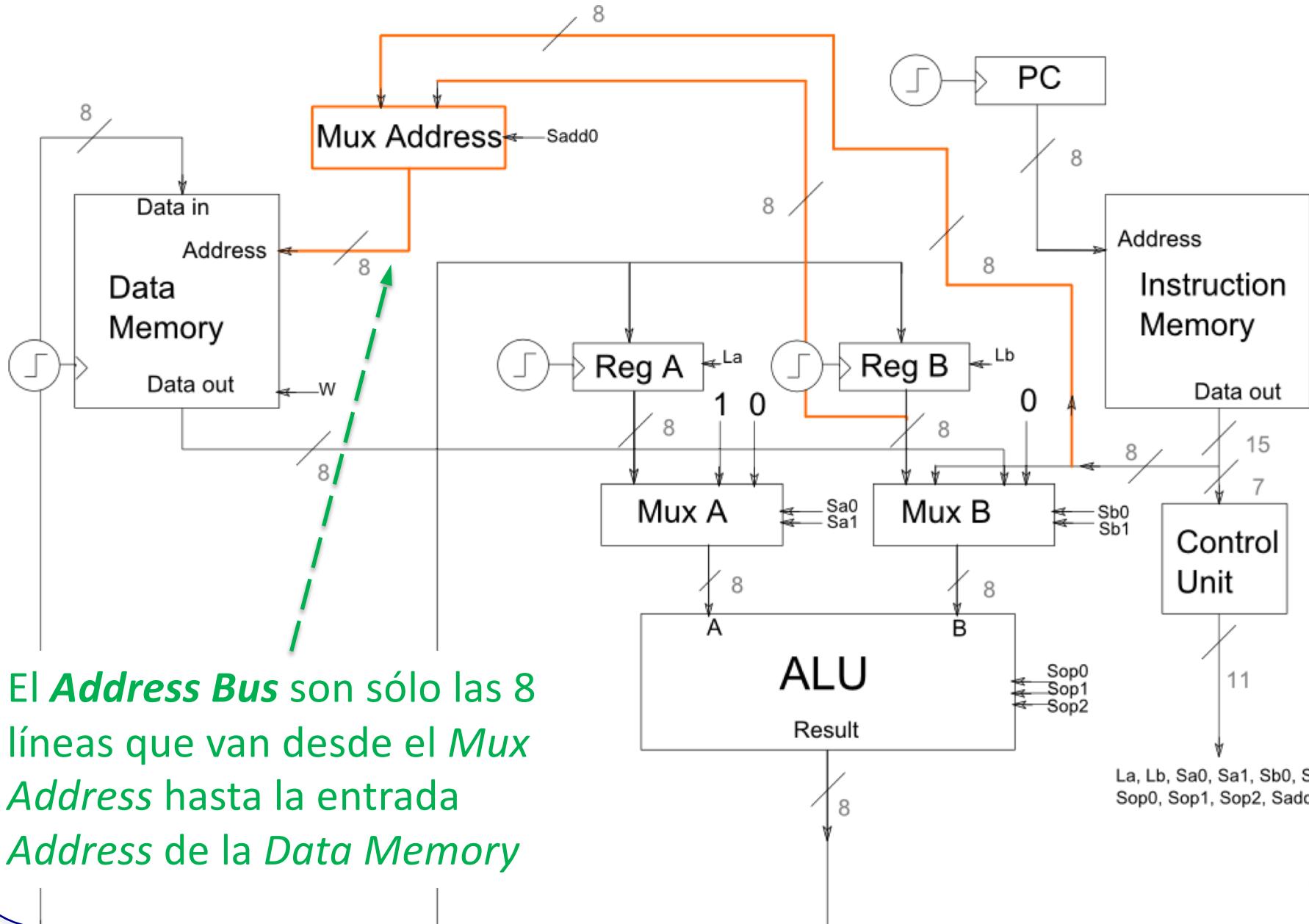
Estas componentes están conectadas mediante **buses**:

- un **bus** es una colección de alambres paralelos
- transmiten datos, señales de control, y direcciones
- estos buses son externos a la CPU —conectando la CPU a la memoria (como más o menos vimos) y a los dispositivos de I/O (como vamos a ver)
... también hay buses internos, que conectan los registros con la ALU y con la cache

P.ej., en el computador básico, el
Data Bus y el **Control Bus (W)**
entre la CPU y la **Data Memory**



P.ej., en el computador básico, el *Address Bus* entre la CPU y la *Data Memory*



El *Address Bus* son sólo las 8 líneas que van desde el *Mux Address* hasta la entrada *Address* de la *Data Memory*

En la práctica, un mismo bus externo tiene líneas de control, líneas de datos y líneas de dirección

Más aún, típicamente las líneas de datos y las líneas de dirección son las mismas:

- durante una *operación de I/O* —es decir, una interacción entre la CPU y un dispositivo de I/O— primero se envía una dirección y después se envían o reciben los datos

Un computador moderno puede tener varios buses (externos); p.ej.:

- un bus para conectar la CPU con la memoria
- un bus para conectar la CPU con los dispositivos de I/O de alta velocidad
- un bus para conectar la CPU con los dispositivos de I/O de menor velocidad

Acciones típicas de la CPU sobre un bus: *fetch* y *store*

Fetch:

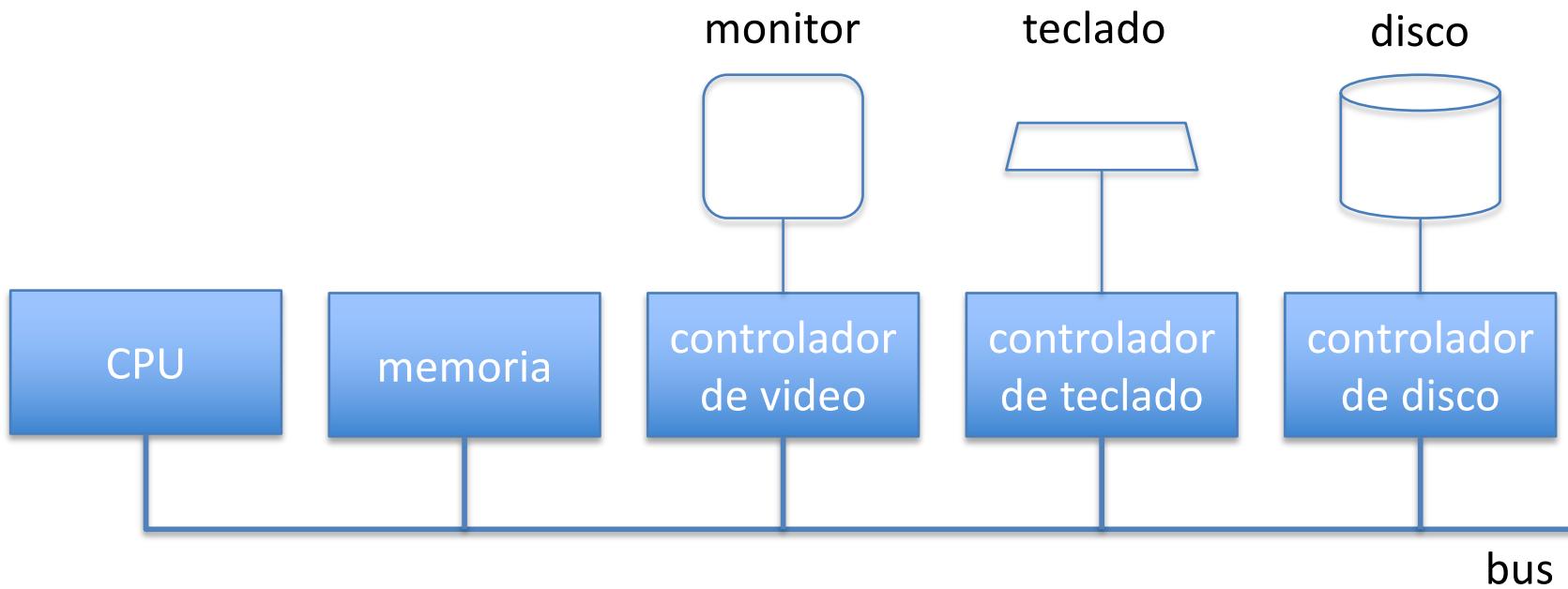
1. Usar las líneas de control para obtener acceso al bus
2. Colocar una dirección en las líneas de dirección
3. Usar las líneas de control para especificar que se trata de una operación *fetch*
4. Examinar las líneas de control para esperar a que la operación se complete
5. Leer los valores de las líneas de datos
6. “Setear” las líneas de control para permitir que alguien más use el bus

Store:

1. Usar las líneas de control para obtener acceso al bus
2. Colocar una dirección en las líneas de dirección
3. Colocar un valor en las líneas de datos
4. Usar las líneas de control para especificar que se trata de una operación *store*
5. Examinar las líneas de control para esperar a que la operación se complete
6. “Setear” las líneas de control para permitir que alguien más use el bus

Todo dispositivo de I/O tiene dos partes:

- un **controlador** (o adaptador)
 - ... un chip o conjunto de chips que físicamente controla el dispositivo
 - ... acepta comandos desde el sistema operativo y los lleva a cabo
- el dispositivo propiamente dicho
 - ... formado por elementos electromecánicos
 - ... que realizan las operaciones de interacción



El **controlador** (o *adaptador*) de un dispositivo de I/O:

- es un chip incluido en la placa madre (*motherboard*) del procesador
 - ... o una tarjeta con circuito impreso que puede insertarse en un slot del bus
- puede incluir un pequeño procesador embebido programado —el trabajo que hace el controlador es complejo
- tiene algunos registros, para comunicarse con la CPU
 - ... la CPU escribe en los registros para pedir al dispositivo que envíe datos, o que acepte datos, que se encienda o se apague, etc.
 - ... la CPU lee estos registros para saber cuál es el estado del dispositivo, si está listo para recibir una solicitud, etc.
- tiene un *buffer* de datos, que puede ser leído o escrito por la CPU

El software que hay que instalar en el sistema operativo para darle instrucciones al controlador de un dispositivo particular se conoce como el **driver del dispositivo**

P.ej., si un programa necesita datos desde el disco:

- la CPU —ejecutando instrucciones del sistema operativo— le da una instrucción al controlador del disco
 - ... y el controlador le da instrucciones a su vez al control electromecánico del disco —el *drive*
- cuando el *drive* ha encontrado la pista y sector del disco apropiados, le envía los 512 bytes del sector al controlador, en la forma de un *stream* de bits
- el controlador (re)agrupa los bits en bytes, usando el *buffer* interno
 - ... verifica el *checksum* del bloque de bytes
 - ... y finalmente escribe el bloque en la memoria

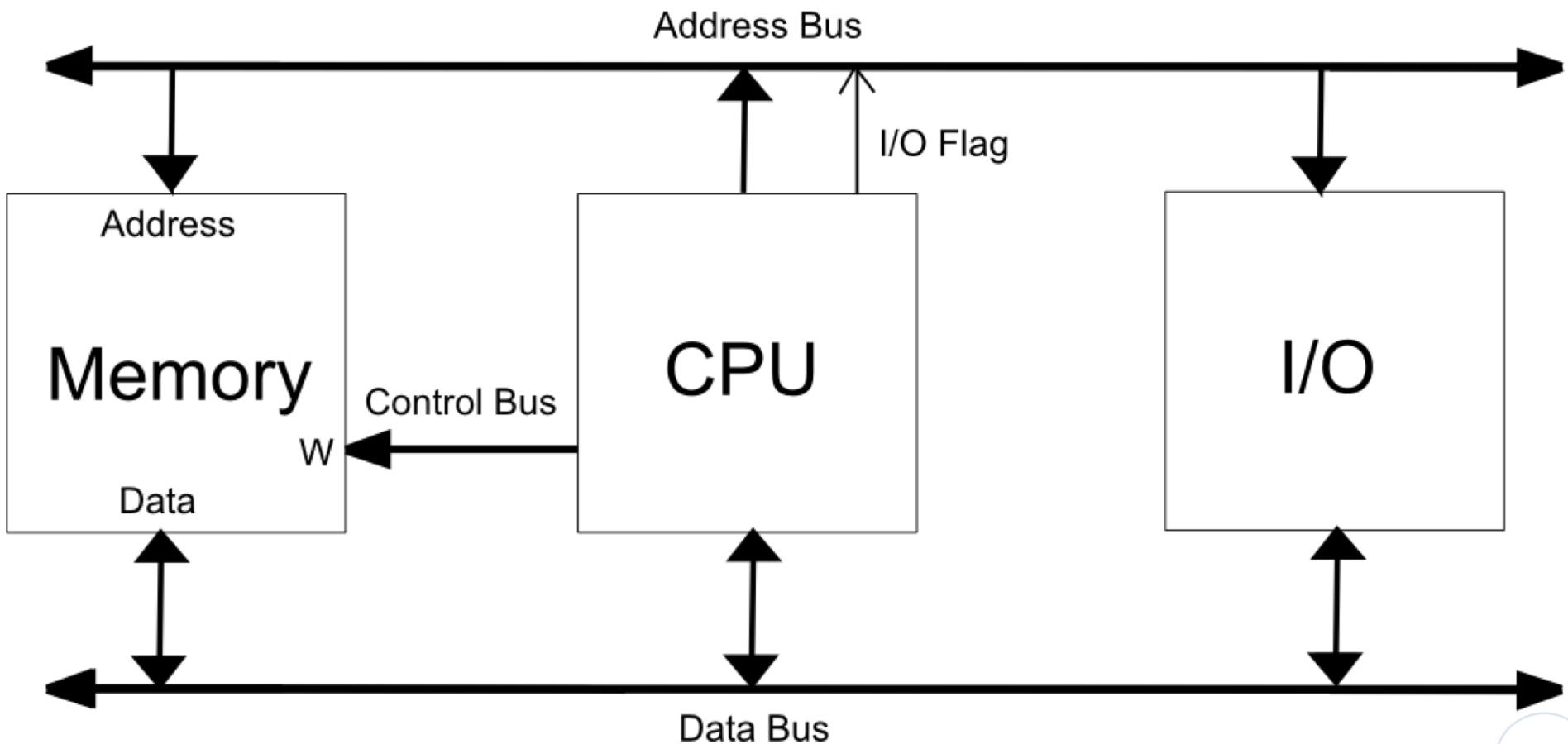
¿Cómo se comunica la CPU con —cómo tiene acceso la CPU a— los registros y el *buffer* de datos del controlador?

- *I/O ports*
- *memory-mapped I/O*

I/O ports:

- a cada registro se le asigna un número de **puerto de I/O**
- el conjunto de todos los puertos de I/O forman el **espacio de puertos de I/O**, al que solo tiene acceso el sistema operativo (está protegido)
- los espacios de direcciones para la memoria y para I/O son diferentes
- la CPU usa instrucciones especiales, p.ej.,
 - ... **IN reg, port** (lee el registro **port** del controlador y guarda el valor en el registro **reg** de la CPU) y
 - ... **OUT port, reg** (escribe el valor del registro **reg** de la CPU en el registro **port** del controlador)

En I/O ports, para indicar que una dirección es la dirección de un registro de un controlador (y no de una celda de memoria), se usa una señal de control: *I/O Flag*, una línea del bus



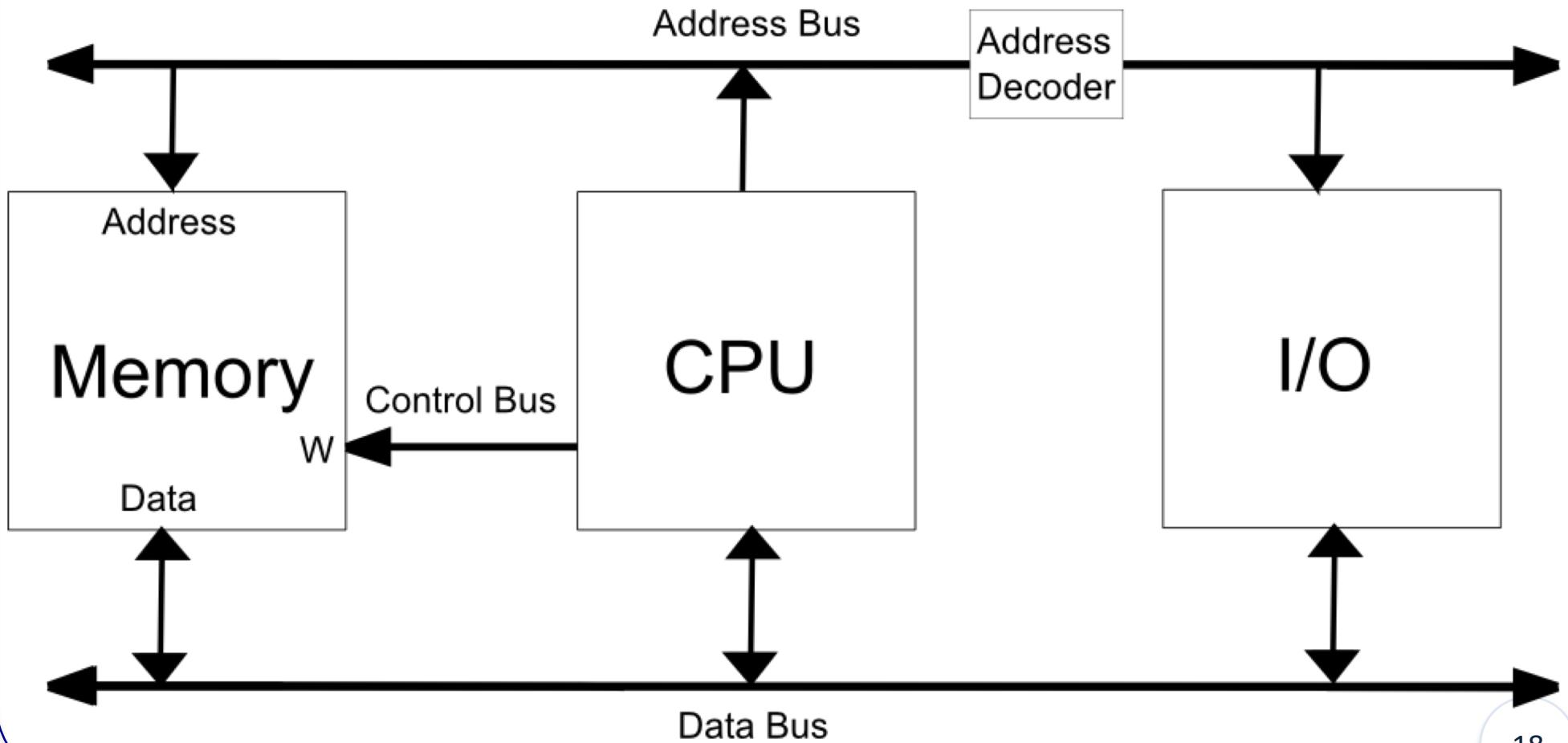
Memory-mapped I/O:

- todos los registros de los controladores son parte del mismo espacio de direcciones de la memoria del computador
- a cada registro se le asigna una dirección de memoria única, que no está asignada a ninguna celda de memoria
- p.ej., las direcciones asignadas a los registros están en la parte alta de la numeración (el extremo 0xFFFF)
- la CPU usa instrucciones normales de acceso a memoria

P.ej., la arquitectura Intel x86 usa un esquema híbrido:

- memory-mapped I/O para los *buffers* de datos de los controladores
- puertos I/O para los registros de los controladores

En memory-mapped I/O, es necesario ocupar un *Address Decoder*, una pieza de hardware que “vigila” las líneas de dirección del bus para determinar si la dirección corresponde a una celda de memoria o a un registro de un controlador



P.ej., un dispositivo con 16 luces de *status* conectado a un bus de 32 bits debe responder a los siguientes cinco comandos:

- encender el display
- apagar el display
- definir el brillo del display
- encender la *i*-ésima luz
- apagar la *i*-ésima luz

Se eligen las siguientes direcciones, no usadas por otros dispositivos

dirección	operación	significado
1024 (a 1027)	<i>store</i>	valor $\neq 0$ enciende el display; valor 0, lo apaga
1024 (a 1027)	<i>fetch</i>	devuelve 0 si el display está apagado, $\neq 0$ si está encendido
1028 (a 1031)	<i>store</i>	cambia brillo según los 4 bits de datos menos significativos: brillo de 0 a 15
1032 (a 1035)	<i>store</i>	los 16 bits menos significativos controlan c/u de las luces: 0 apaga, 1 enciende

Aunque las operaciones son conceptualmente *fetch* y *store*, la interfaz del dispositivo no actúa como una memoria:

- no almacena los datos para usarlos más adelante
- los bits en la solicitud que recibe del bus son simplemente bits
- contiene circuitos lógicos que comparan los bits de dirección con las direcciones asignadas al dispositivo
- p.ej., la primera operación de la diap. anterior podría implementarse así (obviamente, esta es la versión en software, para facilitar su comprensión)

if (address == 1024 && op == store && data != 0):

encender_display

elseif (address == 1024 && op == store && data == 0):

apagar_display

¿Cómo se lleva a cabo la comunicación entre la CPU y los dispositivos de I/O?

- sabemos que los dispositivos están conectados al bus
- ... y que la CPU puede interactuar con un dispositivo enviando operaciones (de tipo) *fetch* y *store* a través del bus a direcciones que han sido asignadas previamente al dispositivo

Tres esquemas de I/O:

- I/O programado, con *busy waiting* (espera ocupada)
- I/O controlado por interrupciones
- DMA I/O

I/O programado, con *busy waiting* (espera ocupada). Usado en microprocesadores en sistemas embebidos o sistemas de tiempo real, en que la CPU hace todo el trabajo

Es el más simple

La CPU tiene una sola instrucción de *input* y una sola instrucción de *output*:

- cada instrucción especifica un dispositivo
- se transfiere un solo carácter entre un registro fijo de la CPU y el dispositivo
- la CPU ejecuta una secuencia fija de instrucciones por cada carácter escrito o leído (enviado al o recibido desde el dispositivo, respectivamente)

P.ej., supongamos un terminal con cuatro registros, cada uno con una dirección única:

- dos registros de *input* (*status* y datos del teclado) y dos registros de *output* (*status* y datos de la pantalla)
- el bit 7 del registro de *status* del teclado se pone en 1 cada vez que llega un nuevo carácter (el usuario oprimió una tecla)
 - ... la CPU lee repetidamente este registro hasta que el bit 7 esté en 1
 - ... entonces lee el registro de datos del teclado (que almacena el carácter)
- similarmente, el bit 7 del registro de *status* de la pantalla se pone en 1 cada vez que ésta está lista para aceptar un carácter
 - ... la CPU lee repetidamente este registro hasta que el bit esté en 1
 - ... entonces escribe un carácter en el registro de datos de la pantalla

El problema con I/O programado es que la CPU pasa la mayor parte del tiempo en un *loop* esperando a que se ponga en 1 el bit 7 de los registros de *status* —***busy waiting***:

- si la CPU no tuviera nada más que hacer, entonces *busy waiting* podría ser aceptable
... pero cuando tiene más trabajo que hacer —p.ej., ejecutar otros programas— entonces *busy waiting* es ineficiente

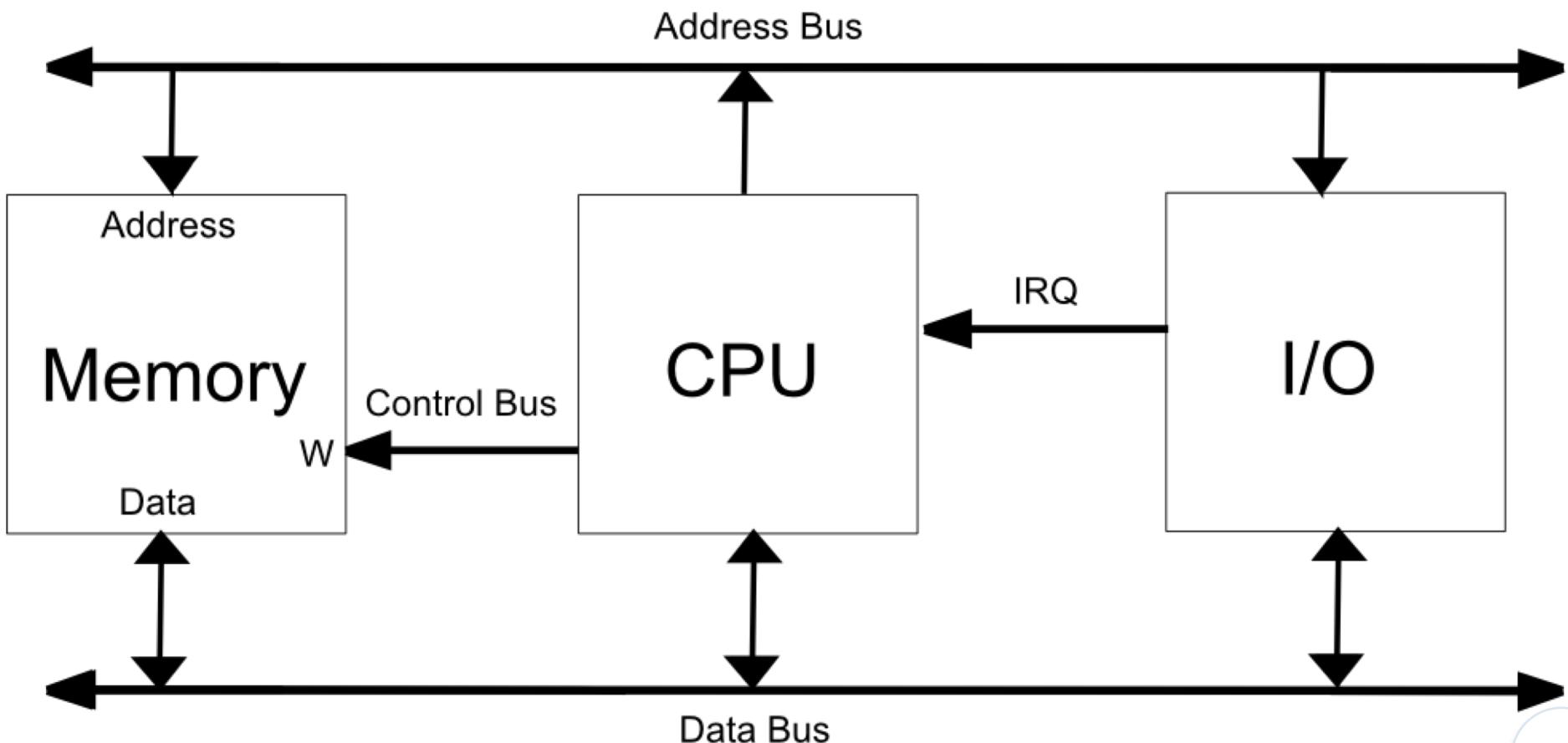
“Polling” — la CPU pregunta repetidamente al dispositivo si la operación ha terminado, antes de iniciar la próxima operación

P.ej., en el caso de una impresora:

- ver si la impresora está encendida
- ordenar a la impresora cargar una hoja de papel
- “poll” para saber cuando el papel ha sido cargado
- especificar los datos en la memoria que dicen qué imprimir
- “poll” para esperar a que la impresora cargue los datos
- ordenar a la impresora empezar a *spray* una línea de tinta
- “poll” para saber cuando el mecanismo de la tinta termine
- ordenar a la impresora avanzar el papel hasta la próxima línea
- “poll” para saber cuando el papel ha avanzado
- repetir los seis pasos anteriores para cada línea
- ordenar a la impresora ejectar la página
- “poll” para saber cuando la página ha sido ejectada

I/O controlado por interrupciones. La CPU hace partir al dispositivo y le dice que la interrumpa cuando esté listo (es decir, cuando el dispositivo haya completado la operación de I/O):

- la CPU “setea” el bit *interrupt enable* del controlador del dispositivo (p.ej., el bit 6 del registro de *status*)



En términos generales, una **interrupción** cambia el flujo de control:

- normalmente debido a un dispositivo de I/O
 - ... y no debido (al menos, no directamente) al programa que se está ejecutando

La interrupción detiene el programa y transfiere el control a un (código) **manejador de la interrupción**:

- primero, ejecuta las acciones apropiadas a la interrupción
- luego, devuelve el control al programa, que debe ser reanudado en exactamente el mismo estado que tenía cuando ocurrió la interrupción

P.ej., supongamos que el computador quiere mostrar una línea de caracteres en la pantalla:

- el software primero recolecta todos los caracteres en un *buffer*
 - ... inicializa las variables **ptr** (dirección del *buffer*) y **count** (número de caracteres)
 - ... y verifica si la pantalla está lista, en cuyo caso envía el primer carácter
- la CPU ha iniciado la operación de I/O y puede hacer otra cosa
- a su debido tiempo el carácter va a ser desplegado en la pantalla
 - ... y empieza la interrupción → los 6 pasos de la próxima diap., ejecutados directamente por el hardware

1. El controlador del dispositivo pone en 1 la línea de interrupción del bus
2. Cuando la CPU está lista para manejar la interrupción, la CPU pone en 1 la línea de *acknowledge* (de la interrupción) en el bus
3. Cuando el controlador ve esta señal, coloca su *id*, un número entero, en las líneas de datos del bus
4. La CPU guarda este número (el *id* del controlador) temporalmente
5. La CPU pone los registros *PC* y *PSW* en el stack
6. La CPU usa el *id* del controlador como índice en una tabla; el valor almacenado allí —llamado *vector de interrupción*— es asignado al registro *PC*

Este nuevo valor del registro *PC* apunta al comienzo de la función que maneja la interrupción específica → los 6 pasos de la próxima diap., ejecutados por software

7. La función guarda todos los registros, en el stack o en una tabla
8. Se identifica cuál dispositivo produjo al interrupción (puede haber más de uno con el mismo vector de interrupción); la CPU lee alguna otra información sobre la interrupción; y si ocurre un error de I/O, se maneja aquí
9. Las variables globales son actualizadas: **ptr** = **ptr**+1, para que apunte al próximo byte, y **count** = **count**-1; si **count** > 0, entonces el byte apuntado por **ptr** se copia en el buffer de output
10. Si el protocolo lo requiere, se avisa al controlador que la interrupción fue procesada
11. Se restaura el valor de todos los registros guardados
12. Se ejecuta la instrucción *return from interrupt*: la CPU vuelve al modo y estado que tenía justo antes de la interrupción

Notas

1) Desde el punto de vista de un programador de aplicaciones, una interrupción es *transparente*:

- el programador escribe las aplicaciones como si las interrupciones no existieran
- el hardware es diseñado de modo que el resultado de una computación es el mismo si no ocurren interrupciones, si ocurre una sola interrupción, o si ocurren varias interrupciones durante la ejecución de las instrucciones
- cuando ocurre una interrupción, se toman algunas acciones y se ejecuta algún código, pero cuando la interrupción termina, el computador vuelve a exactamente el mismo estado que tenía antes de la interrupción

2) ¿Cómo se da cuenta la CPU de que un dispositivo ha solicitado una interrupción (es decir, ha puesto un 1 en la línea de interrupción del bus)?

- se modifica el ciclo de ejecución de las instrucciones
 - el primer paso, antes de hacer *fetch* de la instrucción, es chequear si algún dispositivo ha solicitado una interrupción (en cuyo caso se maneja la interrupción)
- así, una interrupción ocurre *entre* la ejecución de dos instrucciones

3) Los *ids* de los controladores (números enteros pequeños) se asignan de modo que puedan ser usados como índices a una tabla (arreglo) de punteros a localidades reservadas de memoria:

- la asignación ocurre automáticamente al “bootear” el computador
- cada uno de estos punteros es llamado un *vector de interrupción*
- cada *vector de interrupción* apunta al software *manejador de interrupciones* de un dispositivo particular, o de un tipo particular de dispositivos

4) ¿Qué pasa si durante la ejecución de una interrupción, otro dispositivo quiere generar su propia interrupción?

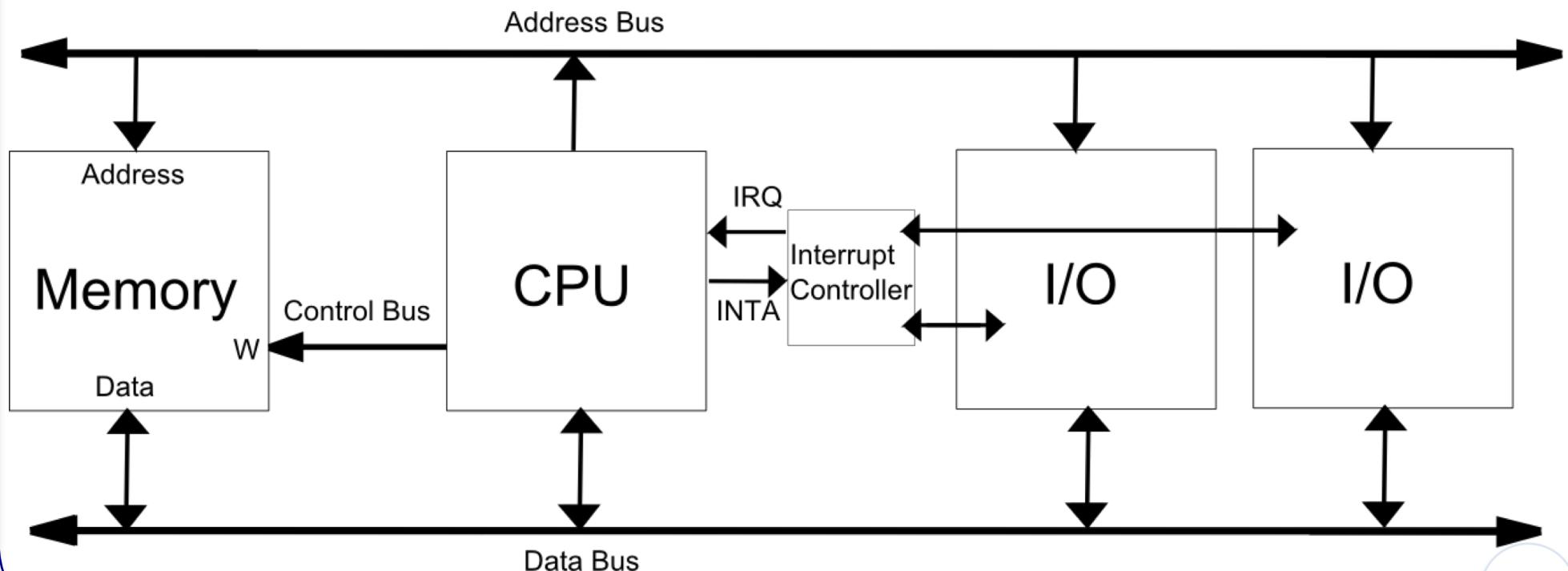
- solución simple: lo primero que hace un manejador de interrupciones, incluso antes de guardar los registros, es deshabilitar interrupciones subsiguientes, hasta que la interrupción vigente termine
- solución sofisticada: a cada dispositivo se le asigna un nivel de prioridad de interrupción; cuando la CPU está ejecutando una interrupción con nivel de prioridad k , el hardware no permite que ocurran otras interrupciones con nivel de prioridad k o menor (ver diaps. #36 a 39)

5) Al pasar de *polling* a interrupciones, muchos aspectos del computador deben ser diseñados apropiadamente:

- el hardware de los dispositivos de I/O —ya no opera solo bajo el control de la CPU, sino que independientemente; y al terminar, debe ser capaz de interrumpir a la CPU
- la arquitectura y funcionalidad del bus —debe permitir comunicación en ambos sentidos
- la arquitectura de la CPU —debe tener un mecanismo que haga que la CPU suspenda temporalmente la ejecución normal, maneje la solicitud de un dispositivo, y luego reanude la ejecución
- el paradigma de programación —en vez de un estilo secuencial y sincrónico, en que el programador especifica cada paso de la operación del dispositivo, se requiere un estilo asíncrono en que el programador escribe código para manejar eventos

Controlador de interrupciones. Cómo se manejan en la práctica múltiples interrupciones provenientes de dispositivos de I/O con diferentes prioridades:

- árbitro que da prioridad a los dispositivos más críticos
- se conecta entre la CPU y múltiples dispositivos de I/O

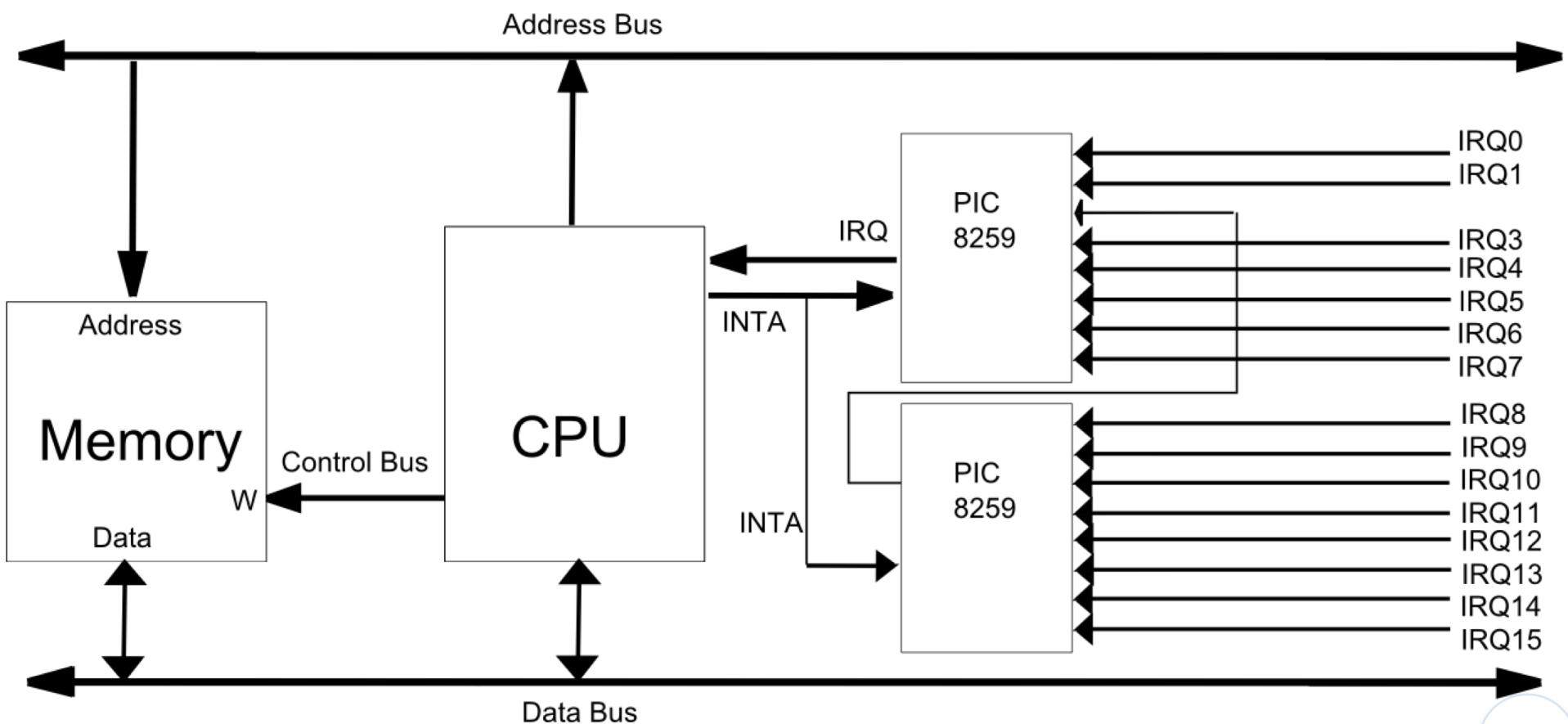


P.ej., el controlador de interrupciones 8259A:

- permite conectar hasta 8 controladores de dispositivos —p.ej., reloj, teclado, disco, impresora, etc.
- cuando cualquiera de éstos quiere producir una interrupción, pone en 1 su línea de input IRQx
- cuando una o más líneas de input están en 1, el controlador coloca en 1 su salida INT, que a su vez opera directamente el pin de interrupción de la CPU
- cuando la CPU está en condiciones de manejar la interrupción, envía una señal de vuelta al controlador por su entrada INTA
- el controlador especifica en el bus de datos cuál input causó la interrupción
- el hardware de la CPU usa este número como índice en su tabla de punteros —o **vectores de interrupción**— para encontrar la dirección de la función que hay que ejecutar para manejar la interrupción

La arquitectura x86 incluye el controlador Intel ICH10:

- chip compuesto por dos controladores 8259A conectados en cascada, que permite conectar hasta 15 controladores de dispositivos de I/O

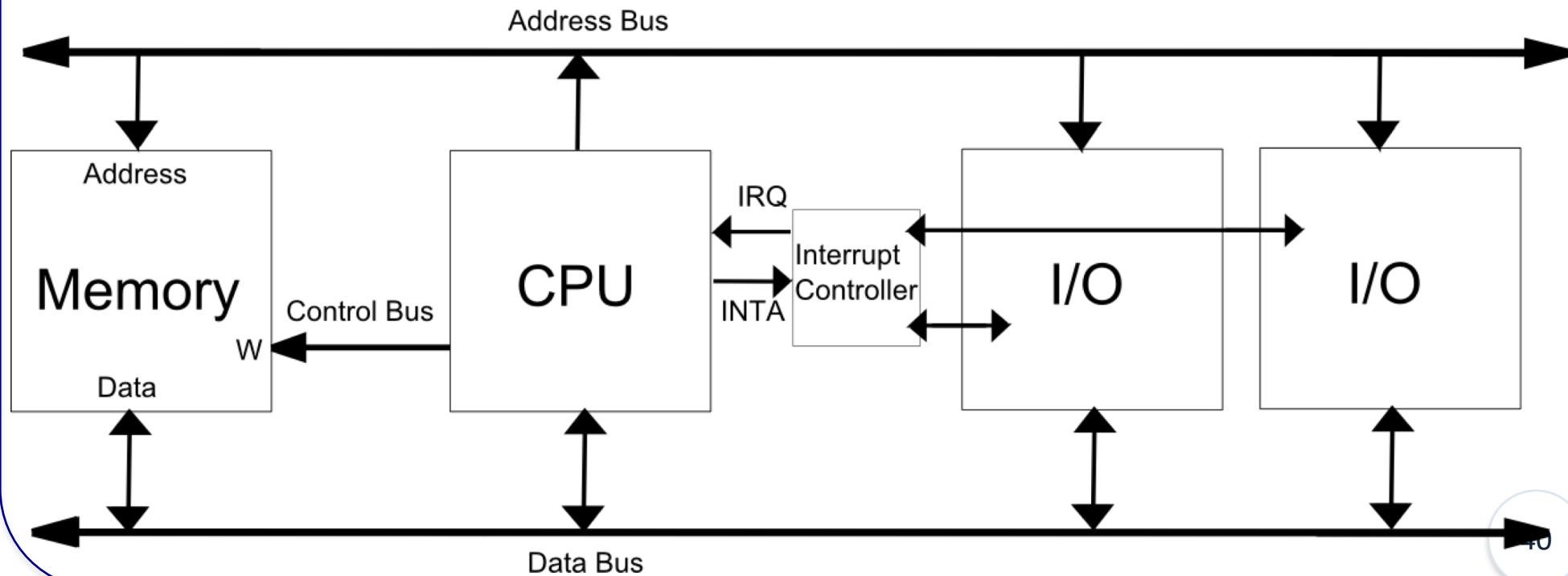


Cada input IRQx está asociado a un dispositivo y un vector de interrupción específicos

IRQ	Dispositivo	Vector de interrupción
IRQ0	Timer del sistema	08
IRQ1	Puerto PS/2: Teclado	09
IRQ2	Conectada al PIC esclavo	0A
IRQ3	Puerto serial	0B
IRQ4	Puerto serial	0C
IRQ5	Puerto paralelo	0D
IRQ6	Floppy disk	0E
IRQ7	Puerto paralelo	0F
IRQ8	Real time clock (RTC)	70
IRQ9-11	No tienen asociación estándar, libre uso.	71-73
IRQ12	Puerto PS/2: Mouse	74
IRQ13	Coprocesador matemático	75
IRQ14	Controlador de disco 1	76
IRQ15	Controlador de disco 2	77

Hasta el momento, todos los datos deben pasar por la CPU para llegar a la memoria o a un dispositivo de I/O; p.ej., para leer un bloque desde el disco y escribirlo en memoria:

- si el disco no está girando, hacerlo girar
- calcular el cilindro que contiene al bloque y mover el brazo del disco hasta allí
- esperar a que el disco gire hasta el sector correcto
- leer los bytes desde el disco y colocarlos en un buffer
- transferir los bytes desde el buffer a la memoria



DMA I/O. Añadimos un nuevo chip al sistema, un **controlador DMA** (*direct memory access*) que permite que los dispositivos de I/O tengan acceso directo a la memoria:

- clave para tener I/O de alta velocidad

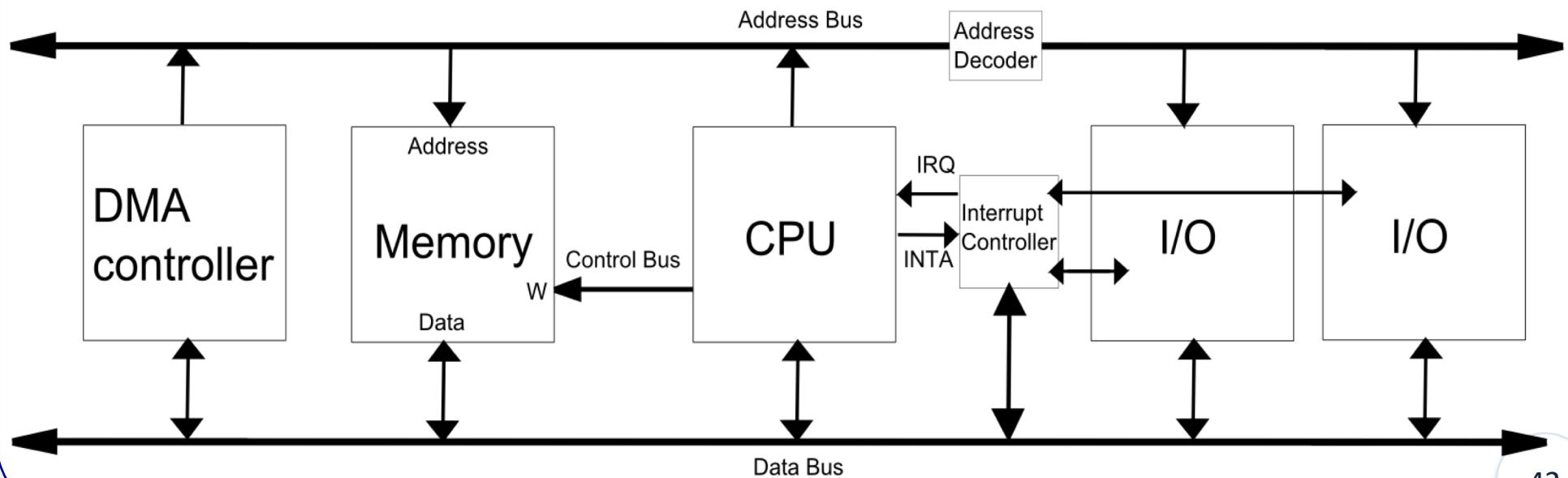
El controlador DMA tiene acceso directo al bus

... y tiene registros que pueden ser escritos por la CPU:

- la dirección de memoria que va a leer o escribir
- número (*count*) de bytes o palabras a transferir
- número (identificador) del dispositivo de I/O que se usará
- dirección (lectura o escritura) de transferencia

Una vez inicializados los registros, el controlador DMA

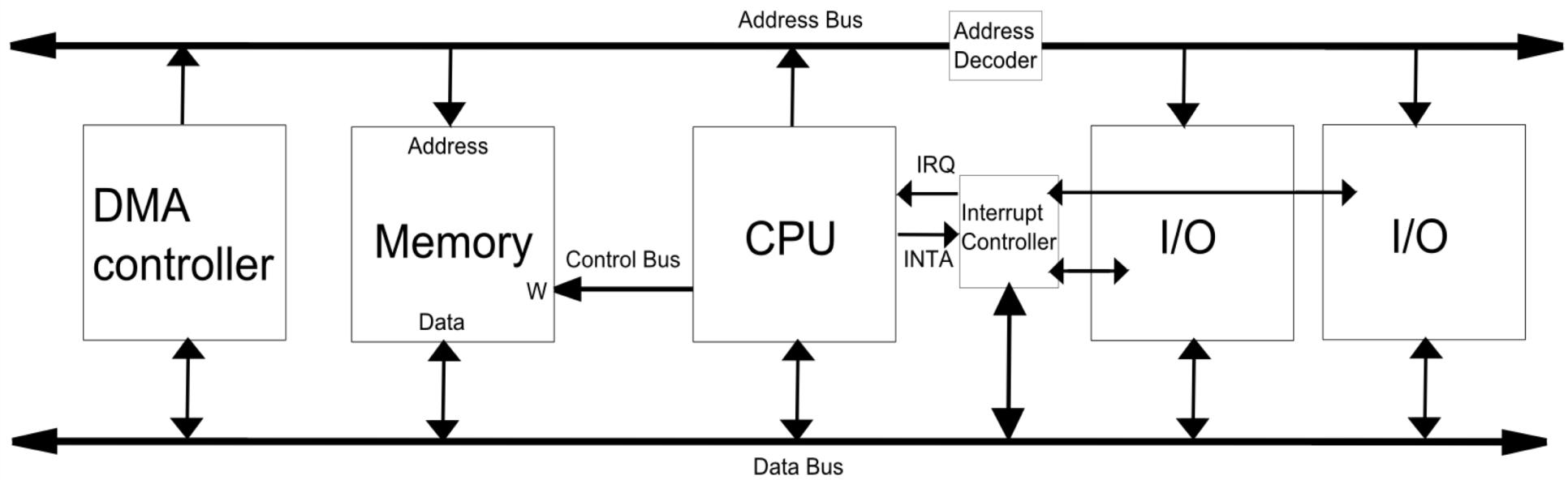
- hace una solicitud al bus, p.ej., para leer 32 bytes a partir de la dirección 100 de la memoria
- hace una solicitud de I/O al dispositivo correspondiente, p.ej., un terminal, para escribir allí el byte leído
- incrementa su registro de dirección y decrementa su registro *count*
- si *count* es > 0, repite las operaciones anteriores
- finalmente, cuando *count* = 0, interrumpe a la CPU



Un controlador que lee y escribe datos desde y en la memoria sin intervención de la CPU está haciendo *acceso directo a memoria* o **DMA**:

- cuando la transferencia está completa, el controlador produce una **interrupción**
 - ... obligando a la CPU a suspender de inmediato el programa vigente
 - ... y empezar a ejecutar el **manejador de interrupciones**
 - ... que, entre otros, informa al sistema operativo que el I/O ha terminado
- cuando el manejador de interrupciones termina, la CPU continúa con el programa que había suspendido

Arquitectura de un computador con memory-mapped I/O, interrupciones y DMA



Los dispositivos de I/O, las redes y los buses pueden tener tasas de transferencia de datos (ttd) muy diferentes

Además, algunos dispositivos funcionan con buses USB, otros con buses PCI, etc.

→ se usan componentes de hardware llamados **puentes** (*bridges*) para interconectar buses

... y hacer *mapping* de direcciones entre ellos

... *transparentemente* para la CPU y los dispositivos de I/O

dispositivo/red/bus	ttd
teclado	10 bytes/s
mouse	100 bytes/s
modem 56K	7 KB/s
scanner (300 dpi)	1 MB/s
camcorder	3.5 MB/s
disco blu-ray 4x	18 MB/s
802.11n wireless	37.5 MB/s
USB 2.0	60 MB/s
FireWire 800	100 MB/s
Ethernet (gigabit)	125 MB/s
disk drive SATA 3	600 MB/s
USB 3.0	625 MB/s
bus SCSI Ultra 5	640 MB/s
bus PCIe 3.0	985 MB/s
bus Thunderbolt 2	2.5 GB/s
red SONET OC0-768	5 GB/s

