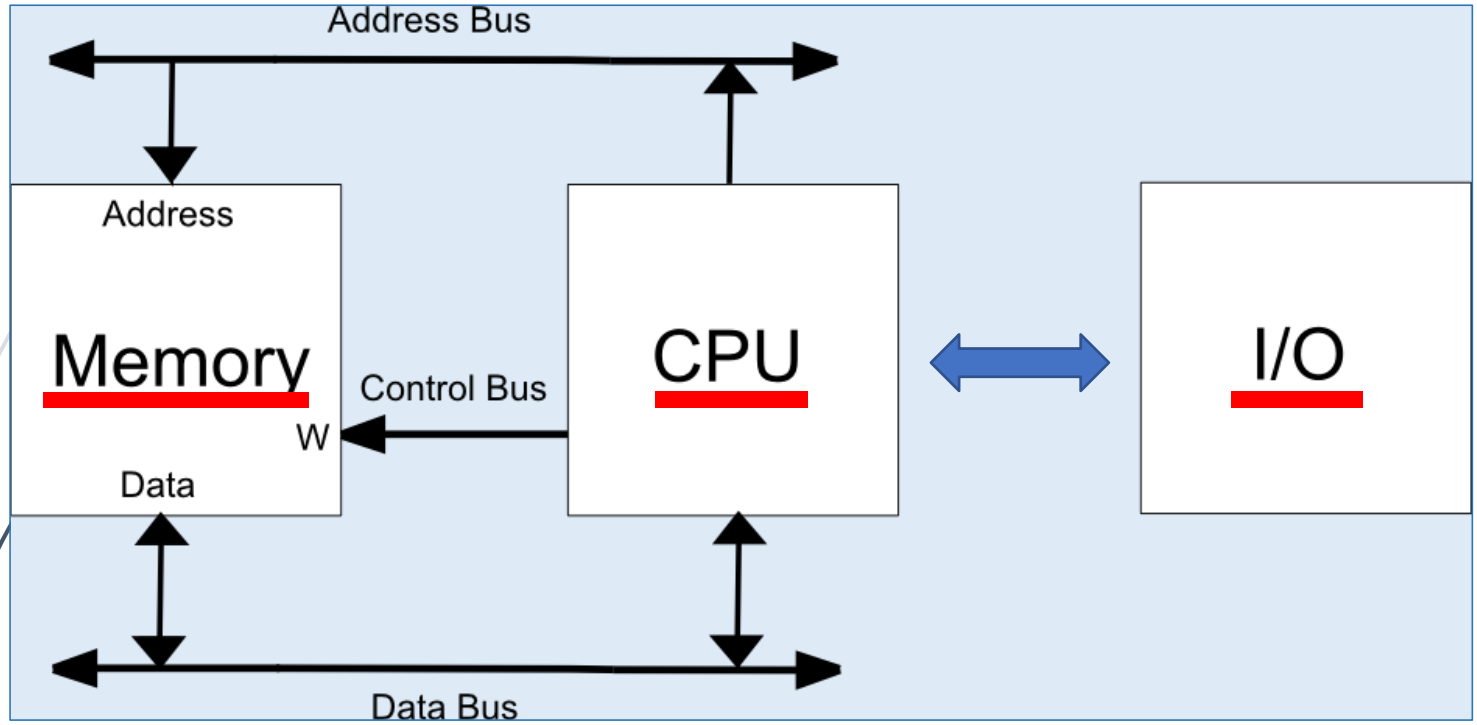
Input/output

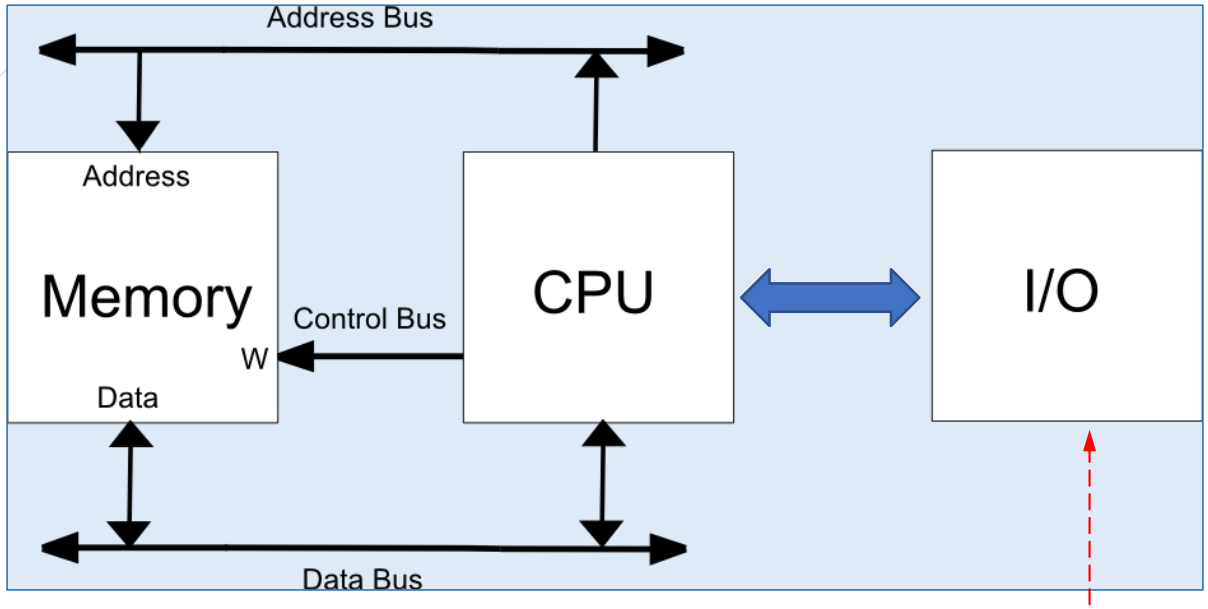
Arquitectura de Computadores – IIC2343

Yadran Eterovic S. (yadran@uc.cl)

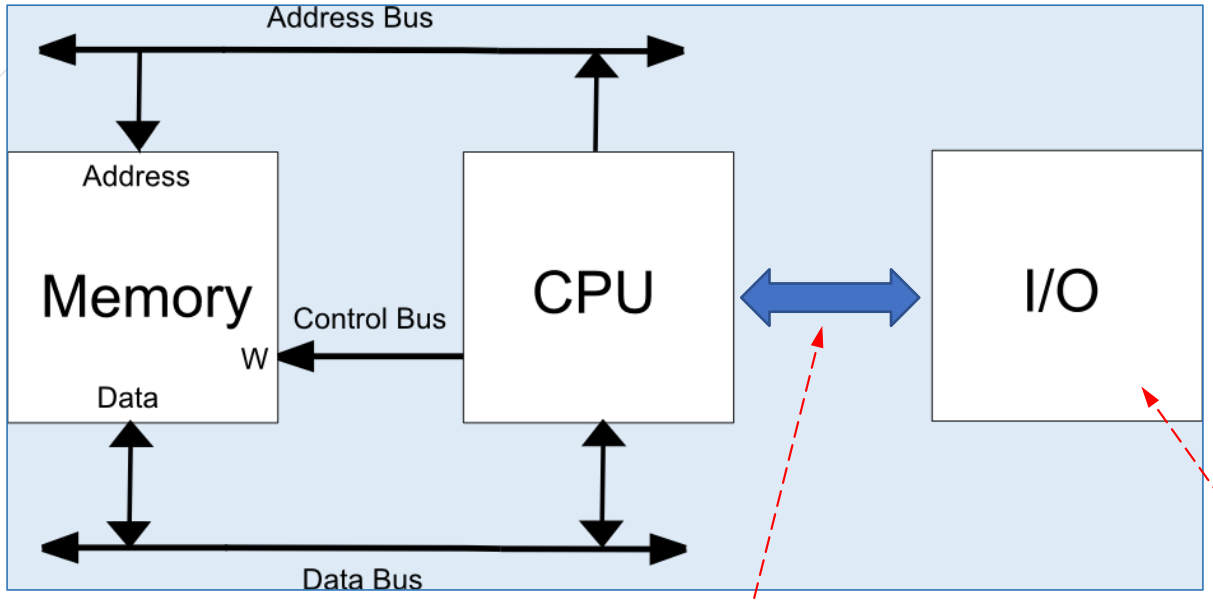
2025-2

Un sistema computacional tiene tres componentes principales





P.ej., teclado, monitor, impresora, disco duro, *modem*, mouse, cámara, audífonos, micrófono, sensores varios, manejador de DVDs, linterna, ...

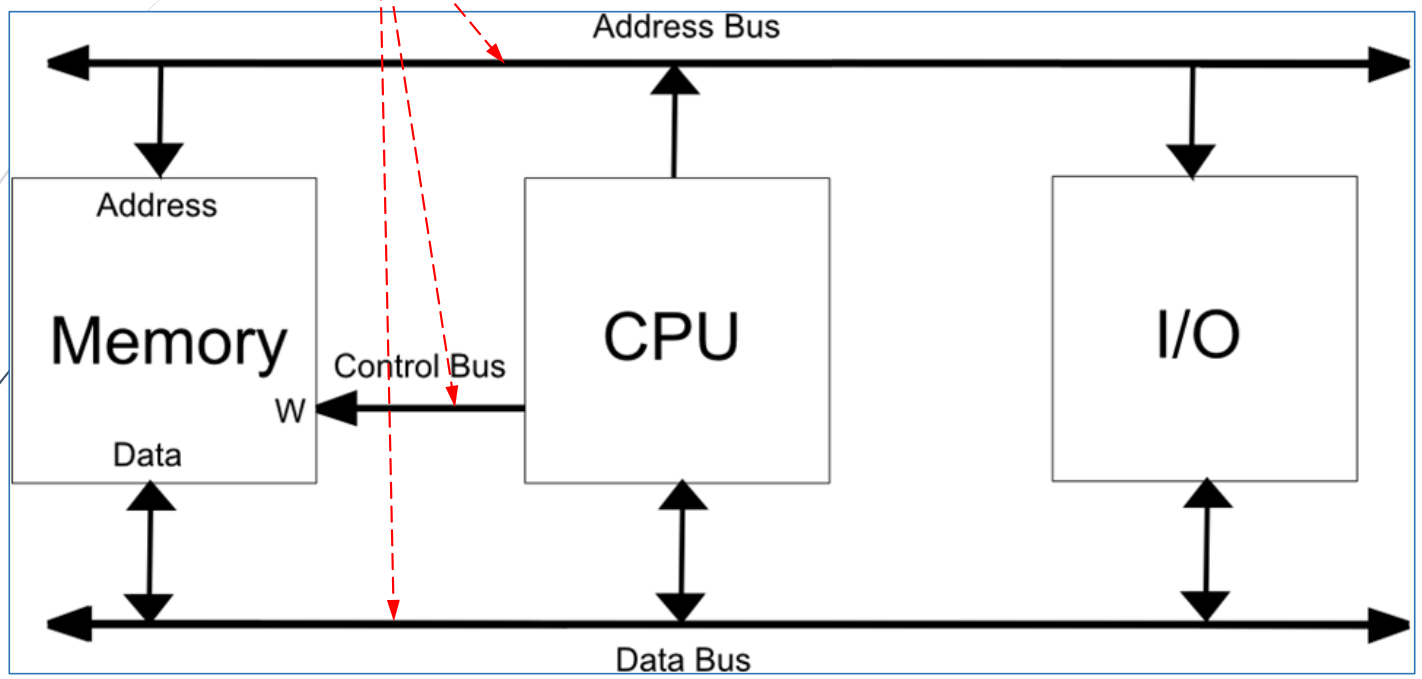


La CPU interactúa con los dispositivos de I/O por dos razones:

- controlar el dispositivo
- intercambiar datos con el dispositivo

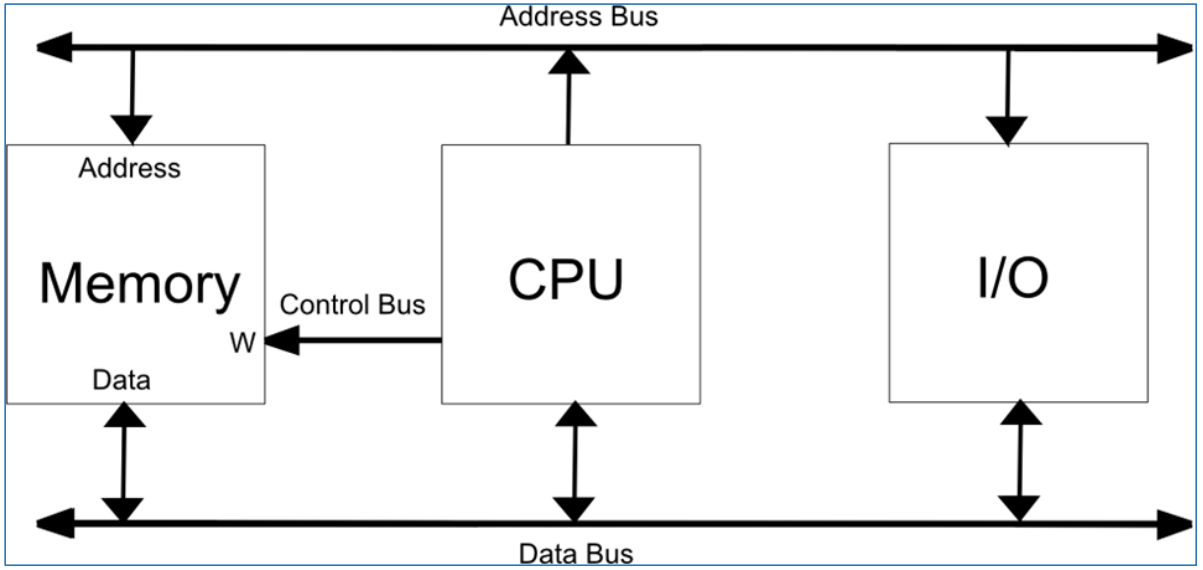
P.ej., teclado, monitor, impresora, disco duro, *modem*, mouse, cámara, audífonos, micrófono, sensores varios, manejador de DVDs, linterna, …

Todos los dispositivos de I/O se comunican con la CPU (y la memoria) a través de **buses**



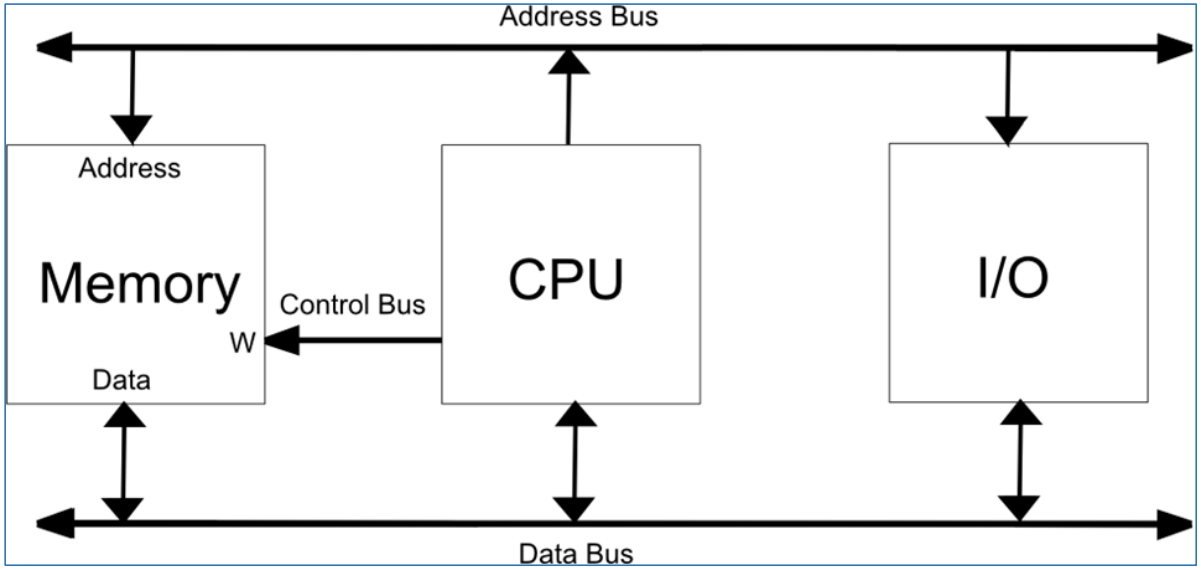
Un **bus** es una colección de alambres paralelos:

- transmiten **datos**, **señales de control**, y **direcciones de memoria**
- los buses de la figura son externos a la CPU (la CPU también tiene buses internos)
- en la práctica, un mismo bus tiene líneas de control, de datos y de dirección
- un computador tiene varios buses (CPU-memoria, CPU-I/O alta velocidad, CPU-I/O baja velocidad)



La comunicación entre la CPU y un dispositivo de I/O usa el **mismo modelo que la comunicación entre CPU y memoria**:

- la CPU "ve" al dispositivo de I/O **como si fuera una extensión de la memoria**
- en una operación de I/O, la CPU envía primero una dirección, y luego envía o recibe datos



En una **operación de I/O**, la CPU primero envía una dirección, y luego envía o recibe datos

Fetch (*load*, *read*):

1. Usa las **líneas de control** para obtener acceso al bus
2. Coloca una dirección en las **líneas de dirección**
3. Usa las **líneas de control** para especificar que se trata de una operación ***fetch***
4. Examina las **líneas de control** para esperar a que la operación se complete
5. Lee los valores de las **líneas de datos**
6. "Setea" las **líneas de control** para permitir que alguien más use el bus

En una **operación de I/O**, la CPU primero envía una dirección, y luego envía o recibe datos

Fetch (*load*, *read*):

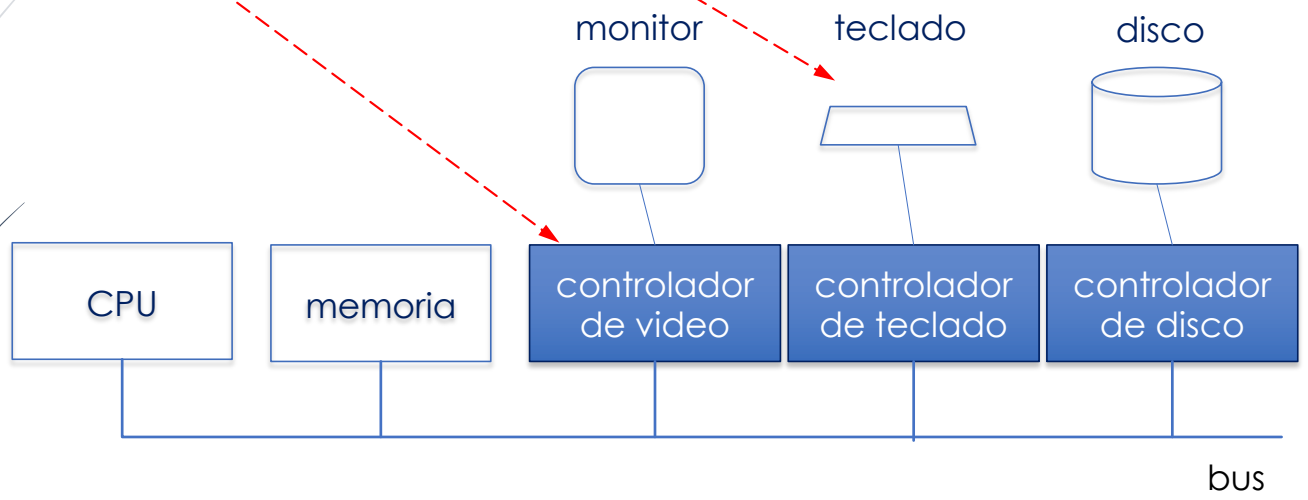
1. Usa las **líneas de control** para obtener acceso al bus
2. Coloca una dirección en las **líneas de dirección**
3. Usa las **líneas de control** para especificar que se trata de una operación ***fetch***
4. Examina las **líneas de control** para esperar a que la operación se complete
5. Lee los valores de las **líneas de datos**
6. "Setea" las **líneas de control** para permitir que alguien más use el bus

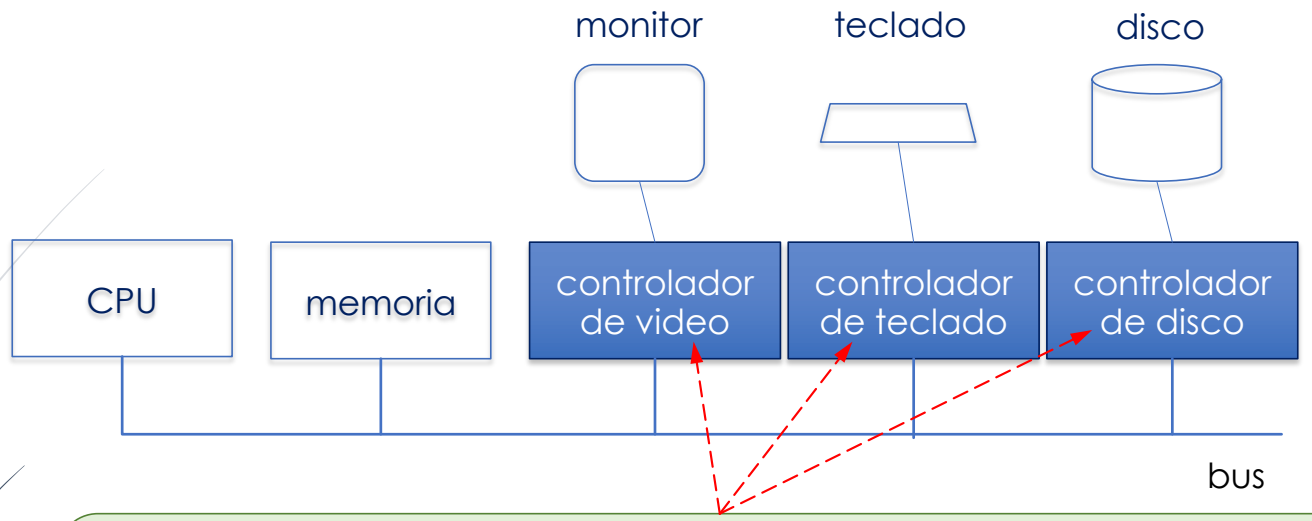
Store (*write*):

1. Usa las **líneas de control** para obtener acceso al bus
2. Coloca una dirección en las **líneas de dirección**
3. Coloca un valor en las **líneas de datos**
4. Usa las **líneas de control** para especificar que se trata de una operación ***store***
5. Examina las **líneas de control** mientras espera que la operación se complete
6. "Setea" las **líneas de control** para permitir que alguien más use el bus

Todo dispositivo de I/O tiene dos partes:

- el dispositivo propiamente dicho, formado por elementos electromecánicos que realizan las operaciones de interacción
- un **controlador** o **adaptador**





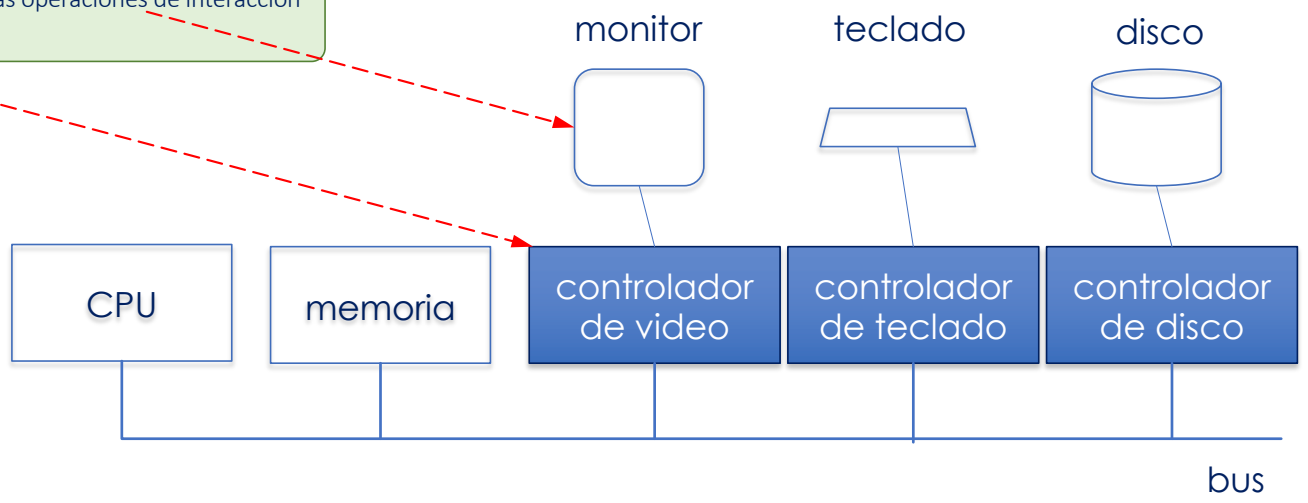
Chip incluido en la placa madre (*motherboard*) del procesador, o tarjeta con circuito impreso que puede insertarse en un *slot* del bus:

- puede incluir un procesador programado —el trabajo que hace el controlador puede ser muy complejo
- tiene **registros**: la CPU escribe en los **registros de comando** para pedir al dispositivo que envíe o acepte datos, que se encienda o se apague, etc.

  … la CPU lee los **registros de estado** para conocer 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

Todo dispositivo de I/O tiene dos partes:

- el dispositivo propiamente dicho, formado por elementos electromecánicos que realizan las operaciones de interacción
- un **controlador** o **adaptador**



Chip incluido en la placa madre (*motherboard*) del procesador, o tarjeta con circuito impreso que puede insertarse en un *slot* del bus:

- puede incluir un procesador embebido programado —el trabajo que hace el controlador es complejo
- tiene **registros**: la CPU escribe en los **registros de comando** para pedir al dispositivo que envíe o acepte datos, que se encienda o se apague, etc.; la CPU lee los **registros de estado** para conocer 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 poder darle instrucciones al controlador de un dispositivo de I/O particular se llama el **driver del dispositivo**

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

- la CPU —mediante la ejecución de instrucciones del *driver* correspondiente en el sistema operativo— le da una instrucción al controlador del disco

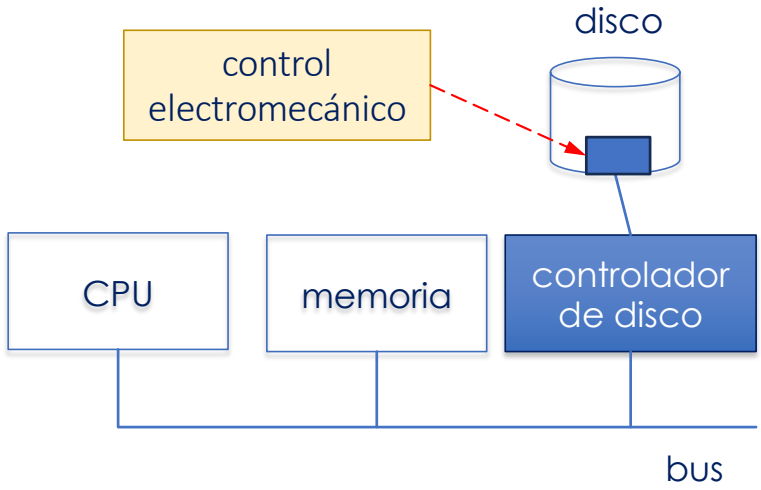
  … y el controlador le da instrucciones a su vez al (control electromecánico del) disco

- cuando el disco ha encontrado la pista y sector apropiados, envía los (p.ej.) 512 bytes al contro-lador, en la forma de un *stream* de bits

- el controlador (re)agrupa los bits en bytes, usando su buffer interno

  … verifica el *checksum* del bloque de bytes

  … y finalmente escribe el bloque en la memoria



¿Cómo asignamos direcciones a los controladores para que la CPU los vea como (extensiones de la) memoria?

Asignamos direcciones (únicas/distintas) a cada registro de cada controlador:

- *I/O ports*
- *Memory-mapped I/O*

I/O ports

Una forma de identificación de los registros de los controladores

A cada registro de cada dispositivo de I/O se le asigna un **número de puerto de I/O**

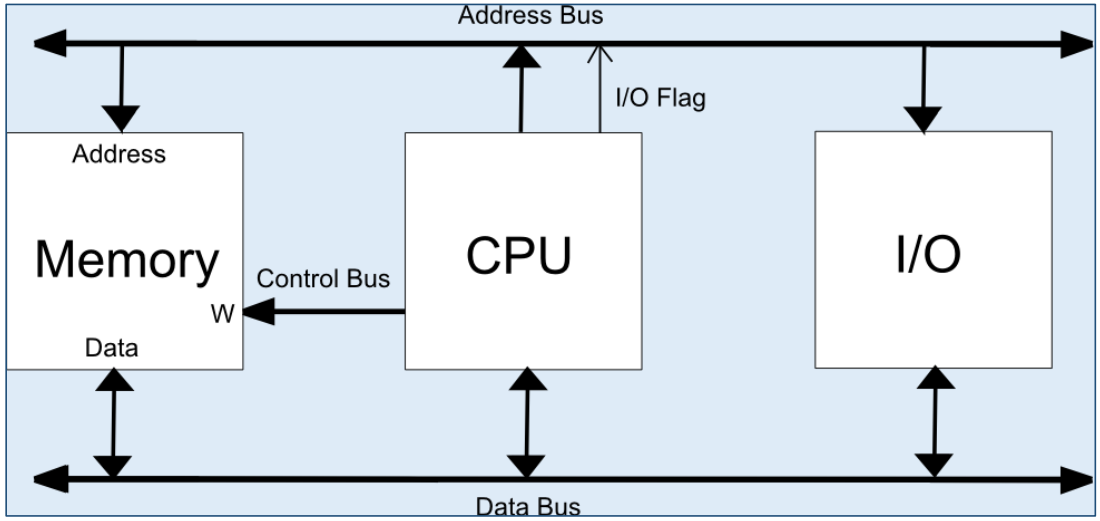
El conjunto de todos los puertos forman el **espacio de puertos de I/O**

… al que solo tiene acceso el sistema operativo

Los espacios de direcciones para la memoria y para (los puertos de) I/O son diferentes

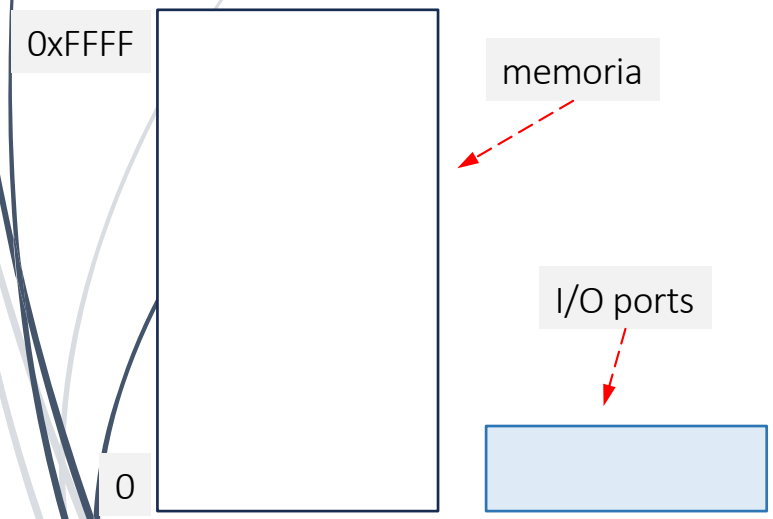
La CPU usa instrucciones especiales, p.ej.:

- **`IN reg,port`** : lee el registro **`port`** del controlador y guarda el valor en el registro **`reg`** de la CPU
- **`OUT port,reg`** : escribe el valor del registro **`reg`** de la CPU en el registro **`port`** del controlador

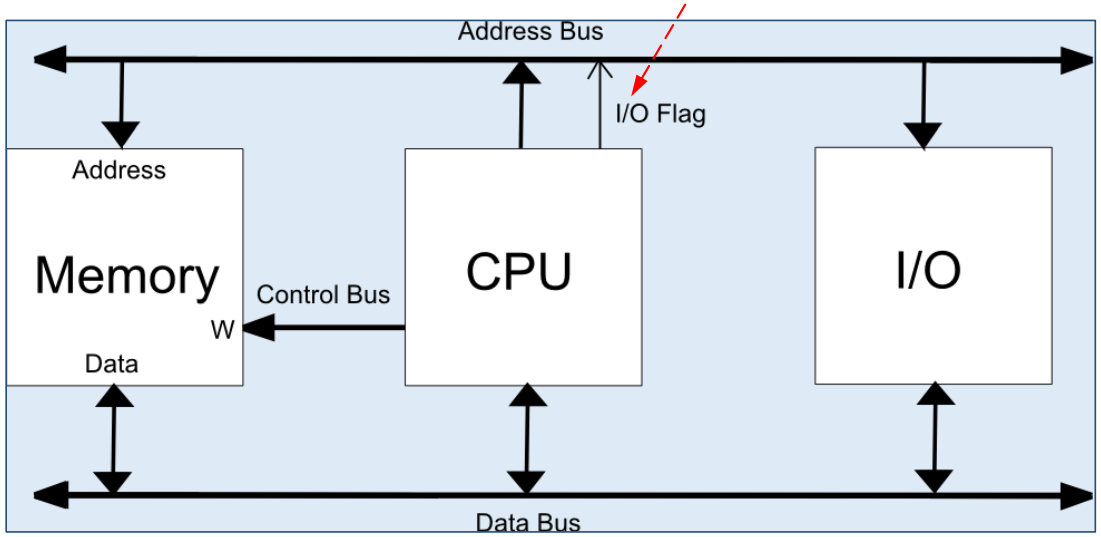


I/O ports

Una forma de identificación de los registros de los controladores



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



Memory-mapped I/O

Otra forma de identificación de los registros de los controladores

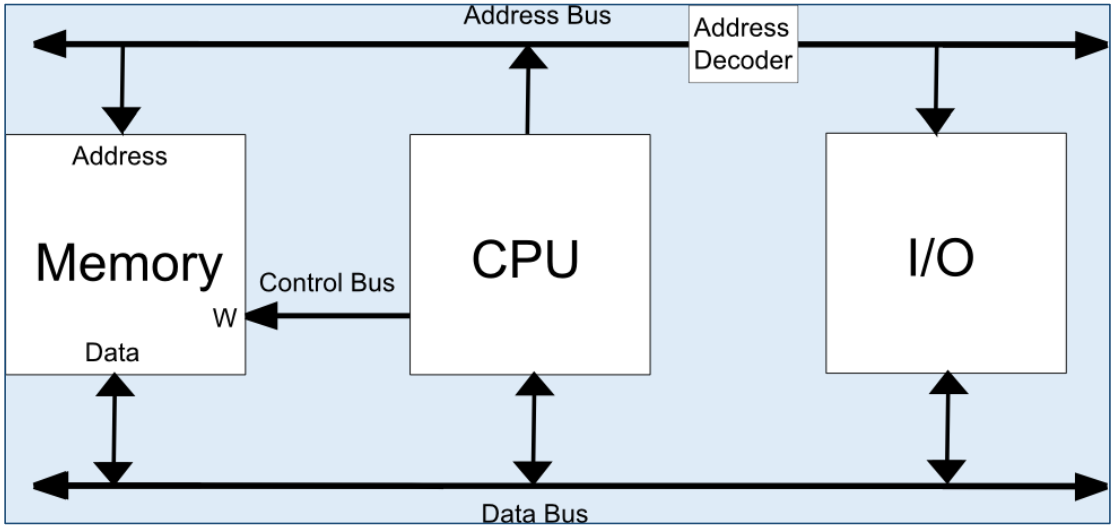
Las direcciones de los registros de los controladores son parte del mismo espacio de direcciones de la memoria

A cada registro de cada dispositivo de I/O 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 son las que están en el extremo 0xFFFF

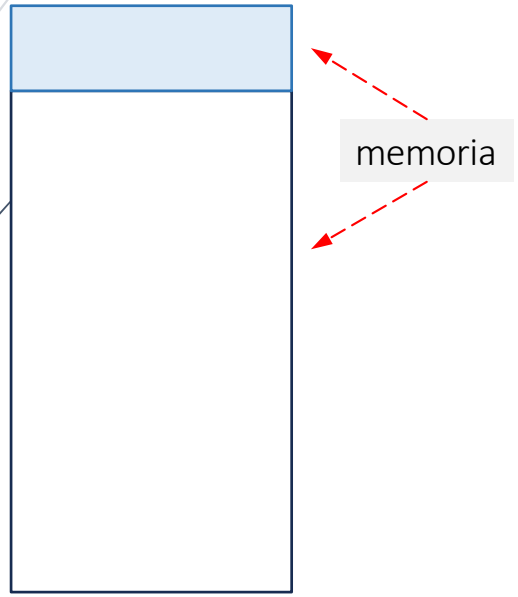
La CPU usa **instrucciones normales de acceso a memoria**: *load* y *store*

(Un “detalle” para más adelante: hay que evitar que el contenido de un registro de un controlador sea traído a la cache y que todos los accesos subsecuentes sean al valor que está en la cache)

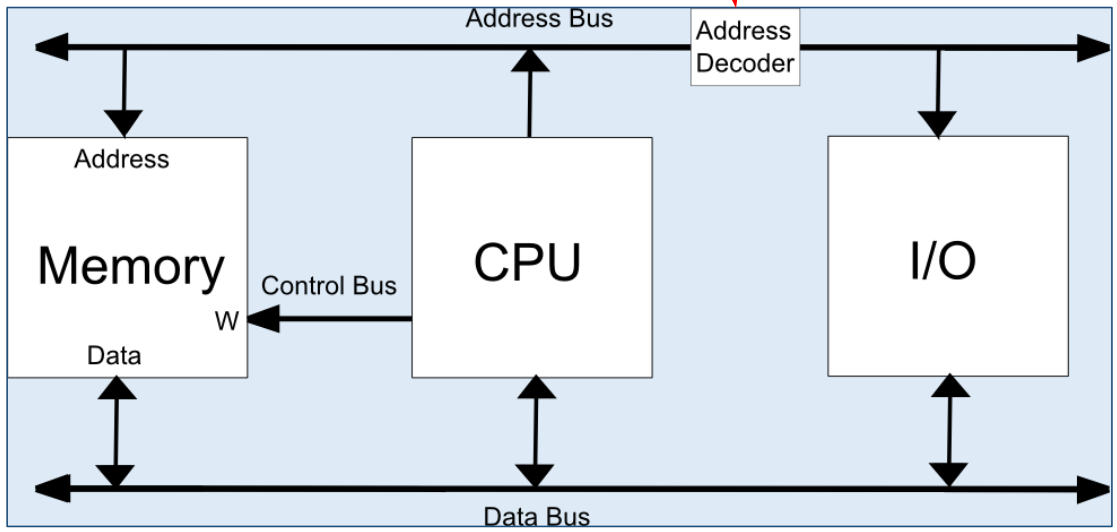


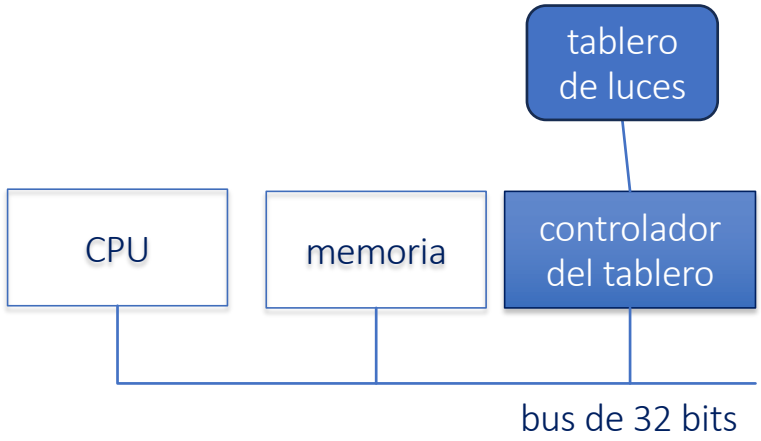
Memory-mapped I/O

Otra forma de identificación de los registros de los controladores



Es necesario usar un ***Address Decoder***: 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., una CPU debe controlar un tablero con 16 luces —el dispositivo de I/O— mediante cinco comandos:

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

Bajo *memory-mapped* I/O, se eligen las siguientes direcciones de memoria, no usadas por otros dispositivos

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

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

```
if (address = 1024 && op = store && data ≠ 0):
    encender_display
elseif (address = 1024 && op = store && data = 0):
    apagar_display
```

Aunque la operación es *store*, el dispositivo —en este caso, el tablero— no es una memoria y no almacena datos para usarlos más adelante

En cambio, la interfaz del dispositivo contiene un circuito que compara los bits de la dirección especificada en la operación con las direcciones asignadas al dispositivo

… y, si coinciden, entonces activa otro circuito que implementa el *store* (o *fetch*) correspondiente

… p.ej. arriba vemos la versión en software (para facilitar su comprensión) del primer *store* de la tabla

Supongamos que conectamos el computador básico a una impresora

El computador usa **memory-mapped I/O**, para los registros de estado y comando de la impresora, e **I/O ports**, para los registros de color, papel, y orientación:

dirección de memoria	registro (valor) de la impresora
0x70	estado: *off* (0x00), *on* (0x01), *printing* (0x02)
0x71	comando: *turn off* (0x00), *turn on* (0x01), *print* (0x02)

puerto	registro (valor) de la impresora
0x00	modo: blanco y negro (0x00) / color (0x01)
0x01	tamaño de papel: carta (0x00) / oficio (0x01)
0x02	orientación: vertical (0x00) / horizontal (0x01)

El siguiente programa, en *assembly* del computador básico

… primero enciende la impresora, si está apagada;

… y luego termina la ejecución del programa, si está imprimiendo,

… o bien imprime una hoja tamaño oficio a color y en orientación vertical

```
DATA:                   ;Se podrían guardar las direcciones como variables,
CODE:                   ;pero usaremos directamente los literales.
  main:
    MOV B,112           ;B = 0x70 = Registro de estado
    MOV A,(B)
    CMP A,2
    JEQ end             ;Está imprimiendo, no hacemos nada.
    CMP A,1
    JEQ print           ;Está prendida, imprimimos.
  turnOn:               ;Está apagada, la prendemos e imprimimos.
    MOV B,113           ;B = 0x71 = Registro de comando.
    MOV A,1
    MOV (B),A           ;Prender impresora, luego imprimir.
  print:
    MOV A,0
    OUT 2,A             ;Orientación vertical
    MOV A,1
    OUT 0,A             ;Impresión a color
    OUT 1,A             ;Tamaño oficio
    MOV B,113           ;B = 0x71 = Registro de comando
    MOV A,2
    MOV (B),A           ;Imprimir
  end:
```

El siguiente programa, en *assembly* del RISC-V, hace lo mismo:

… enciende la impresora, si está apagada;

… y termina la ejecución del programa, si está imprimiendo,

… o imprime una hoja oficio a color y en orientación vertical

Sólo que RISC-V no tiene soporte para I/O ports ⟹ hay que convertir todo a memory-mapped I/O

… y todos los registros de la impresora se consideran de 4 bytes en las direcciones consecutivas 0x70 a 0x80

```
.text                       # Se podrían guardar las direcciones como variables,
  main:                     # pero usaremos directamente los literales.
    addi s0, zero, 112      # s0 = 0x70 = Registro de estado
    addi s1, zero, 2        # s1 = Estado “imprimiendo”
    addi s2, zero, 1        # s2 = Estado “prendida”
    addi s3, zero, 1        # s3 = Comando “prender”
    addi s4, zero, 1        # s4 = Impresión a color
    addi s5, zero, 1        # s5 = Tamaño oficio
    addi s6, zero, 0        # s6 = Orientación vertical
    addi s7, zero, 2        # s7 = Comando “imprimir”
    lw t0, 0(s0)            # t0 = Valor de registro de estado
    beq t0, s1, end         # Está imprimiendo, no hacemos nada.
    beq t0, s2, print       # Está prendida, imprimimos.
  # Está apagada, la prendemos e imprimimos.
  turnOn:                   # Prender impresora, luego imprimir.
    sw s3, 4(s0)            # s0 + 4 = 0x74 = Registro de comando.
  print:
    sw s4, 8(s0)            # s0 + 8  = 0x78 = Registro de modo de color.
    sw s5, 12(s0)           # s0 + 12 = 0x7C = Registro de tamaño de papel.
    sw s6, 16(s0)           # s0 + 16 = 0x80 = Registro de orientación impresión.
    sw s7, 4(s0)            # Imprimir.
  end:
```

Control de la comunicación CPU – dispositivo de I/O

I/O programado, con *busy waiting*

I/O controlado por interrupciones

DMA I/O

I/O programado
(usando *busy waiting*)

Es el más simple

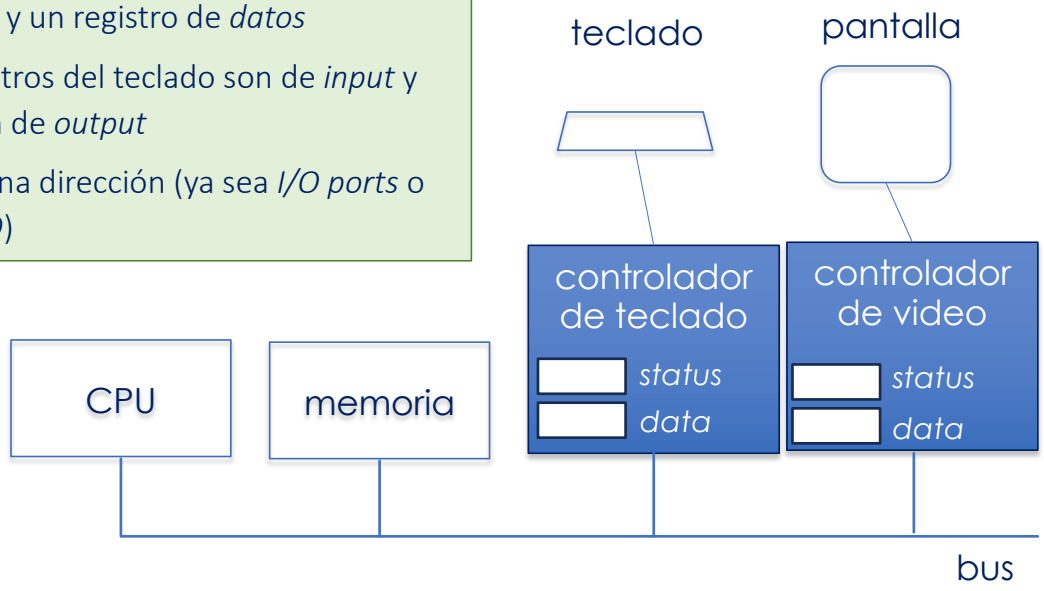
La CPU tiene una instrucción de input (o *fetch*) y una instrucción de output (o *store*):

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

Usado en microprocesadores en sistemas embebidos o sistemas de tiempo real, en que la CPU hace todo el trabajo

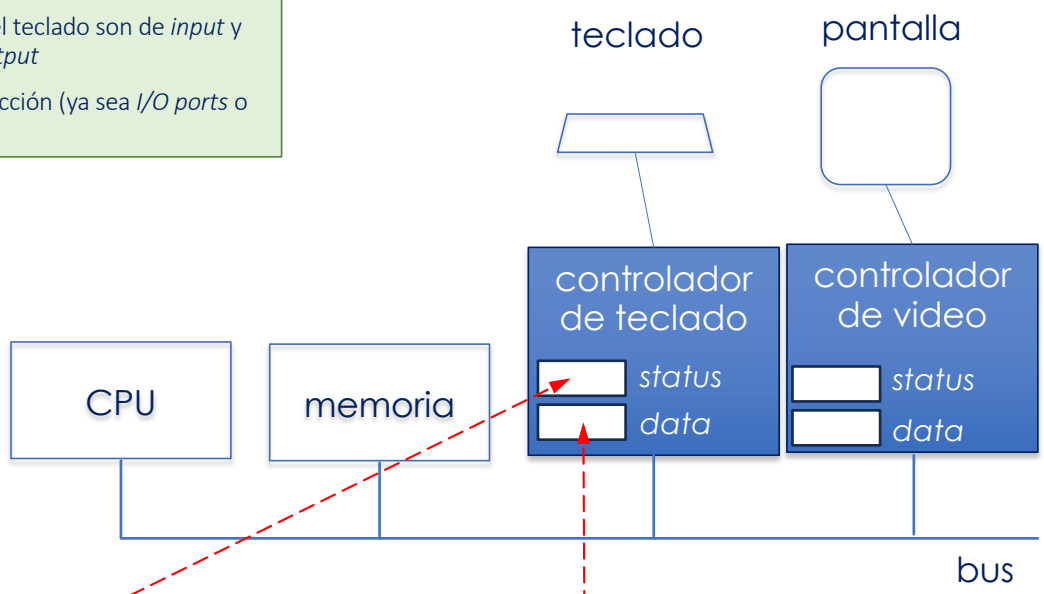
P.ej., supongamos un computador con pantalla y teclado, cuyos controladores tienen dos registros c/u:

- un registro de *status* y un registro de *datos*
- para la CPU, los registros del teclado son de *input* y los de la pantalla son de *output*
- cada registro tiene una dirección (ya sea *I/O ports* o *memory-mapped I/O*)



P.ej., supongamos un computador con pantalla y teclado, cuyos controladores tienen dos registros c/u:

- un registro de *status* y un registro de *datos*
- para la CPU, los registros del teclado son de *input* y los de la pantalla son de *output*
- cada registro tiene una dirección (ya sea *I/O ports* o *memory-mapped I/O*)



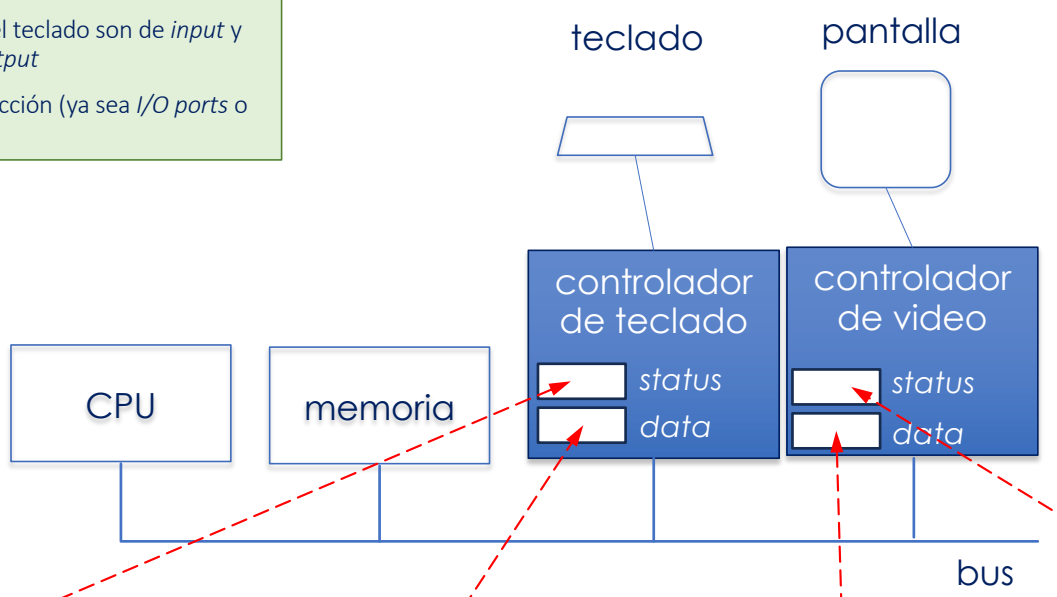
El bit 7 se pone en 1 cada vez que "llega" un nuevo carácter (el usuario apretó una tecla)

La CPU **lee repetidamente este registro** hasta que el bit 7 esté en 1

… entonces la CPU lee este registro, que almacena el carácter

P.ej., supongamos un computador con pantalla y teclado, cuyos controladores tienen dos registros c/u:

- un registro de *status* y un registro de *datos*
- para la CPU, los registros del teclado son de *input* y los de la pantalla son de *output*
- cada registro tiene una dirección (ya sea *I/O ports* o *memory-mapped I/O*)



El bit 7 se pone en 1 cada vez que "llega" un nuevo carácter (el usuario apretó una tecla)

La CPU **lee repetidamente este registro** hasta que el bit 7 esté en 1

... entonces la CPU lee este registro, que almacena el carácter

El bit 7 se pone en 1 cada vez que la pantalla está lista para aceptar un carácter

La CPU **lee repetidamente este registro** hasta que el bit 7 esté en 1

... entonces la CPU escribe el carácter en este registro

El problema de I/O programado es que la CPU pasa la mayor parte del tiempo en *loops* esperando a que el dispositivo haga algo —***busy waiting*** o ***polling***

... p.ej., a que se ponga en 1 el bit 7 de los registros de *status*

Busy waiting podría ser aceptable si la CPU no tuviera nada más que hacer

... pero es ineficiente cuando la CPU tiene otros trabajos que hacer —p.ej., ejecutar otros programas

Forma típica en la que se implementa *busy waiting* o *polling* :

- la CPU pregunta repetidamente al dispositivo si la operación ha terminado, antes de iniciar la próxima operación
- ver ej. en las próxs. diaps.

P.ej., *polling* en el caso de una CPU que da instrucciones a una impresora, cuyo controlador tiene 7 registros (dos de *status* y 5 de control), con direcciones (relativas*) 0, 4, … 24:

1) ver si la impresora está encendida (*on*)
2) ordenar a la impresora cargar una hoja de papel
3) "poll" para saber cuándo el papel ha sido cargado
4) especificar los datos de la memoria que dicen qué imprimir
5) "poll" para esperar a que la impresora cargue los datos
6) ordenar a la impresora empezar a imprimir una línea
7) "poll" para saber cuándo la impresión de la línea termine
8) ordenar a la impresora avanzar el papel hasta la próxima línea
9) "poll" para saber cuándo el papel ha avanzado
10) repetir los seis pasos anteriores para cada línea
11) ordenar a la impresora eyectar la página
12) "poll" para saber cuándo la página ha sido eyectada

*Cuando la impresora es instalada se asignan las direcciones reales

P.ej., *polling* en el caso de una CPU que da instrucciones a una impresora, cuyo controlador tiene 7 registros (dos de *status* y 5 de control), con direcciones (relativas*) 0, 4, … 24:

1) ver si la impresora está encendida (*on*)
2) ordenar a la impresora cargar una hoja de papel
3) "poll" para saber cuándo el papel ha sido cargado
4) especificar los datos de la memoria que dicen qué imprimir
5) "poll" para esperar a que la impresora cargue los datos
6) ordenar a la impresora empezar a imprimir una línea
7) "poll" para saber cuándo la impresión de la línea termine
8) ordenar a la impresora avanzar el papel hasta la próxima línea
9) "poll" para saber cuándo el papel ha avanzado
10) repetir los seis pasos anteriores para cada línea
11) ordenar a la impresora eyectar la página
12) "poll" para saber cuándo la página ha sido eyectada

fetch: registro de *status* en dirección 0 contiene ≠ 0 ⇒ impresora está *on*

store: registro de control en dirección 4 contiene ≠ 0 ⇒ impresora carga papel

store: registro de control en dirección 8 contiene dirección de string a imprimir

store: registro de control en dirección 12 contiene ≠ 0 ⇒ impresora lee la dirección

fetch: registro de *status* en dirección 24 contiene ≠ 0 ⇒ impresora ocupada

store: registro de control en dirección 16 contiene ≠ 0 ⇒ impresora inicia impresión

store: registro de control en dirección 20 contiene ≠ 0 ⇒ impresora avanza papel

*Cuando la impresora es instalada se asignan las direcciones reales

Si a la impresora de la diap. 21 le agregamos un cuarto estado:

turning-on (prendiendo), con valor 0x03

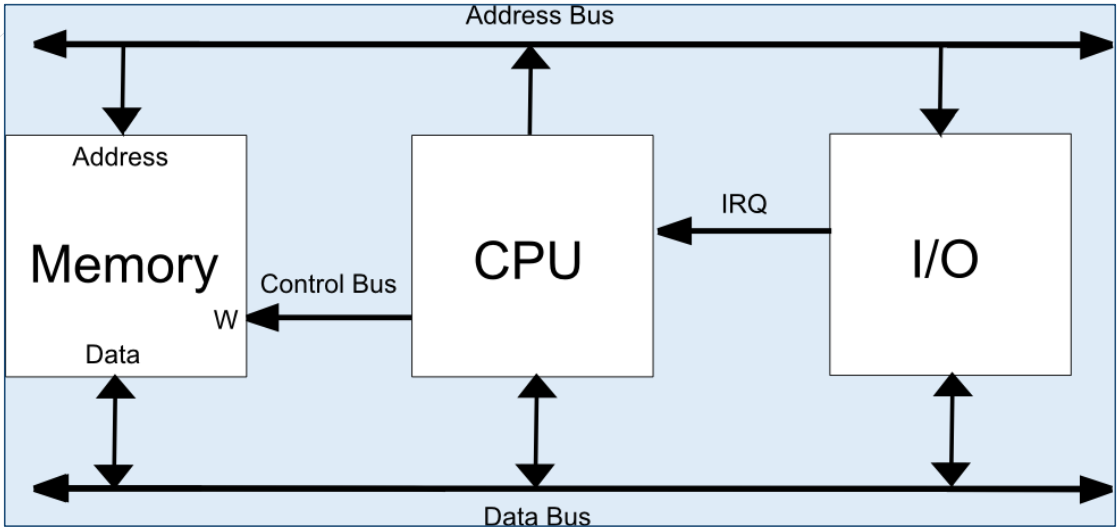
... el programa en *assembly* del computador básico hace ***polling*** a la impresora para que la impresión ocurra sólo cuando la impresora ya esté prendida (*on*)

```
DATA:                       ;Se podrían guardar las direcciones como variables,
CODE:                       ;pero usaremos directamente los literales.
  main:
    MOV B,112               ;B = 0x70 = Registro de estado
    MOV A,(B)
    CMP A,2
    JEQ end                 ;Está imprimiendo, no hacemos nada.
    CMP A,1
    JEQ print               ;Está prendida, imprimimos.
    CMP A,3
    JEQ waitTurningOn       ;Está prendiendo, esperamos a que prenda.
  turnOn:                   ;Está apagada, la prendemos e imprimimos.
    MOV B,113               ;B = 0x71 = Registro de comando.
    MOV A,1
    MOV (B),A               ;Prender impresora, luego imprimir.
  waitTurningOn:
    MOV B,112               ;B = 0x70 = Registro de estado
    MOV A,(B)
    CMP A,1
    JNE waitTurningOn       ;Si todavía no está prendida, repetimos la pregunta.
  print:
    MOV A,0
    OUT 2,A                 ;Orientación vertical
    MOV A,1
    OUT 0,A                 ;Impresión a color
    OUT 1,A                 ;Orientación horizontal
    MOV B,113               ;B = 0x71 = Registro de comando
    MOV A,2
    MOV (B),A               ;Imprimir
  end:
```

Idem, para el assembly RISC-V

```
.text                          # Se podrían guardar las direcciones como variables,
  main:                        # pero usaremos directamente los literales.
    addi s0, zero, 112         # s0 = 0x70 = Registro de estado
    addi s1, zero, 2           # s1 = Estado “imprimiendo”
    addi s2, zero, 1           # s2 = Estado “prendida”
    addi s3, zero, 3           # s3 = Estado “prendiendo”
    addi s4, zero, 1           # s4 = Comando “prender”
    addi s5, zero, 1           # s5 = Impresión a color
    addi s6, zero, 1           # s6 = Tamaño oficio
    addi s7, zero, 0           # s7 = Orientación vertical
    addi s8, zero, 2           # s8 = Comando “imprimir”
    lw t0, 0(s0)               # t0 = Valor de registro de estado
    beq t0, s1, end            # Está imprimiendo, no hacemos nada.
    beq t0, s2, print          # Está prendida, imprimimos.
    beq t0, s3, waitTurningOn  # Está prendida, imprimimos.
  # Está apagada, la prendemos y esperamos a que prenda para imprimir.
  turnOn:                      # Prender impresora, luego esperar a que prenda.
    sw s4, 4(s0)               # s0 + 4 = 0x74 = Registro de comando.
  waitTurningOn:
    lw t0, 0(s0)
    beq t0, s3, waitTurningOn # Si todavía no está prendida, repetimos la pregunta.
  print:
    sw s5, 8(s0)               # s0 + 8  = 0x78 = Registro de modo de color.
    sw s6, 12(s0)              # s0 + 12 = 0x7C = Registro de tamaño de papel.
    sw s7, 16(s0)              # s0 + 16 = 0x80 = Registro de orientación impresión.
    sw s8, 4(s0)               # Imprimir.
  end:
```

I/O controlado por interrupciones



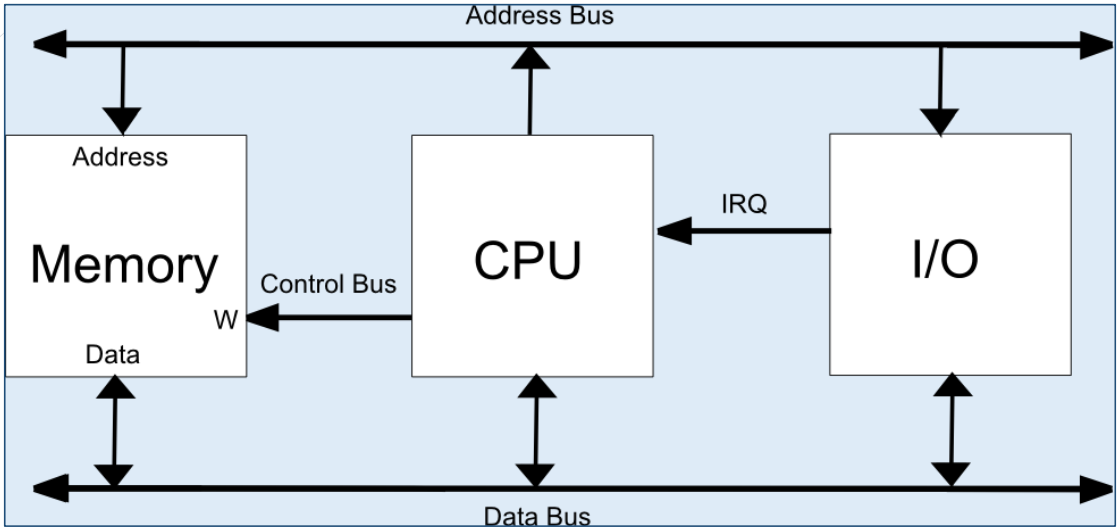
La CPU hace partir al dispositivo

… y le dice que la interrumpa 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*)

… y luego se pone a ejecutar algún programa

I/O controlado por interrupciones



La CPU hace partir al dispositivo

… y le dice que la interrumpa 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*

… y luego se pone a ejecutar algún programa

Cuando llega la interrupción, la CPU detiene la ejecución del programa

… y transfiere el control a un **manejador de la interrupción** (software parte de *driver* del dispositivo):

- el manejador, primero, ejecuta las acciones apropiadas a la interrupción
- luego, devuelve el control al programa, que debe ser reanudado en 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, así, ha iniciado la operación de I/O y ahora puede hacer otra cosa —ejecutar otro programa— mientras la pantalla efectivamente despliega el carácter

- a su debido tiempo el carácter va a ser desplegado en la pantalla

  … y en ese momento empieza la interrupción —la pantalla interrumpe a la CPU (para avisarle que ya desplegó el carácter y está lista para recibir y desplegar el próximo carácter)

  ⟶ los 6 pasos de la próx. diap., **ejecutados directamente por el hardware**

1. El controlador del dispositivo pone en 1 la *línea de interrupción del bus* IRQ

2. Cuando la CPU está lista para manejar la interrupción, pone en 1 la *línea de acknowledge* del bus: la CPU *acknowledges* la interrupción

3. Cuando el controlador ve esta señal, coloca su *id* (un número entero) en las líneas de datos del bus

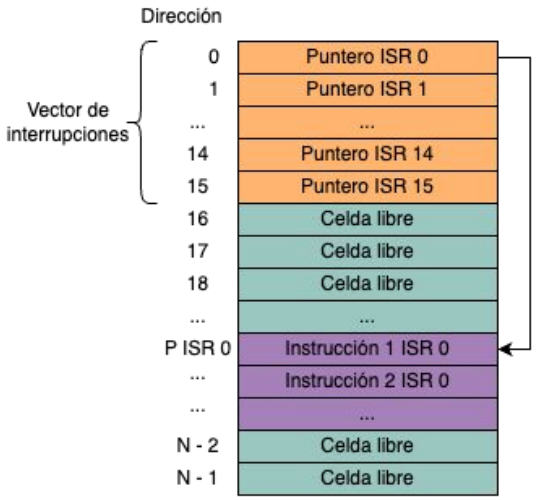
4. La CPU guarda el *id* del controlador

5. La CPU pone (los contenidos de) los registros *PC* y *PSW* en el stack

6. La CPU usa el *id* del controlador como índice en una tabla —*vector de interrupción*; el valor almacenado allí es asignado al registro *PC*

Este nuevo valor del registro *PC* apunta al comienzo de la función que maneja la interrupción específica (la *interrupt service routine* o *ISR*)

… ⟶ los 6 pasos de la próxima diap., **ejecutados por software**



7. La *ISR* guarda todos los registros, en el stack

8. Se identifica cuál dispositivo produjo la interrupción (puede haber más de uno con el mismo vector de interrupción); la CPU lee información adicional sobre la interrupción; y si ocurre un error de I/O, se maneja aquí

9. Las variables 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 (recordemos: esta *ISR* es específica para la operación de mostrar una línea de caracteres en la pantalla)

10. Si el protocolo lo requiere, se le avisa al controlador que la interrupción fue procesada

11. Se restaura el valor de todos los registros guardados (en el paso 7)

12. Se ejecuta la instrucción *return from interrupt*: la CPU vuelve al modo y estado que tenía justo antes de la interrupción (ver paso 5)

Cinco notas

1) Desde el punto de vista de un programador de aplicaciones (en Python, Java, C++, etc.), 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 un programa es el mismo si no ocurren interrupciones, si ocurre una sola interrupción, o si ocurren varias interrupciones durante la ejecución de las instrucciones del programa
- cuando ocurre una interrupción, se toman algunas acciones y se ejecuta algún código, pero cuando la interrupción termina, el computador vuelve al mismo estado que tenía antes de la interrupción (como acabamos de ver)

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

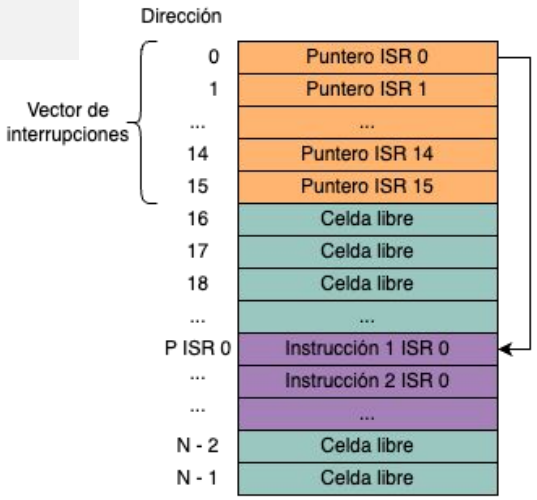
- se modifica el ciclo de ejecución de las instrucciones —*fetch*, *decode*, etc.

  ⟶ el primer paso, justo antes de hacer *fetch* de la (próxima) instrucción, es chequear si algún dispositivo ha solicitado una interrupción (en cuyo caso se maneja la interrupción)

- es decir, **una interrupción ocurre entre la ejecución de dos instrucciones**

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

- la asignación ocurre automáticamente al “bootear” el computador
- cada entrada del vector de interrupción apunta al (es la dirección del) software manejador de interrupciones (ISR) 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 (entre los pasos 3 y 4, diap. 37), es **deshabilitar interrupciones subsecuentes**, hasta que la interrupción vigente termine

- **solución sofisticada** (diaps. 43 a 47): a cada dispositivo se le asigna un nivel de prioridad de interrupción

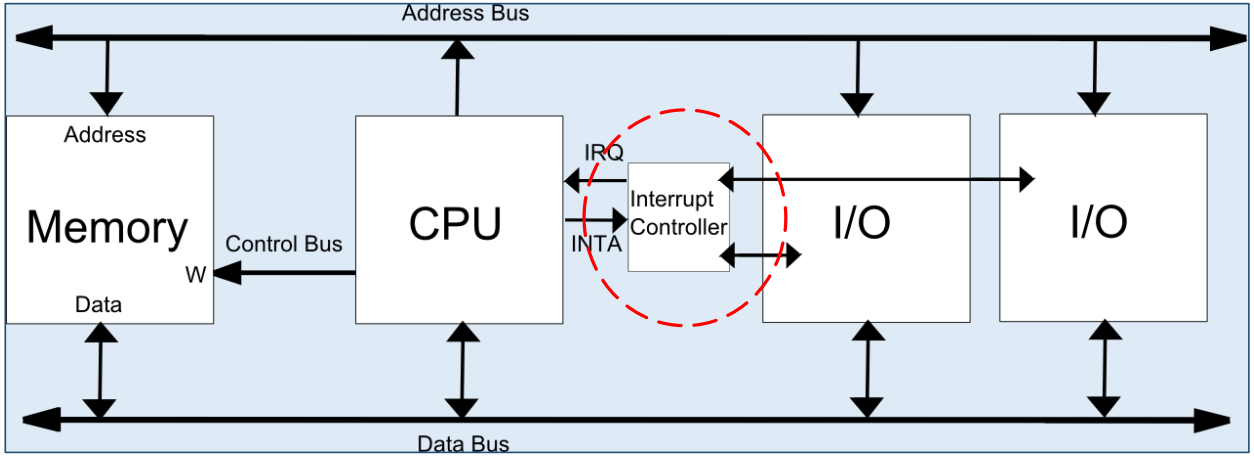
  … se usa un **controlador de interrupciones**

  … cuando la CPU está ejecutando una interrupción con nivel de prioridad *k*, el controlador no permite que ocurran otras interrupciones con nivel de prioridad ≤ *k*

Controlador de interrupciones

Dispositivo que maneja múltiples interrupciones desde varios dispositivos de I/O con diferentes prioridades:

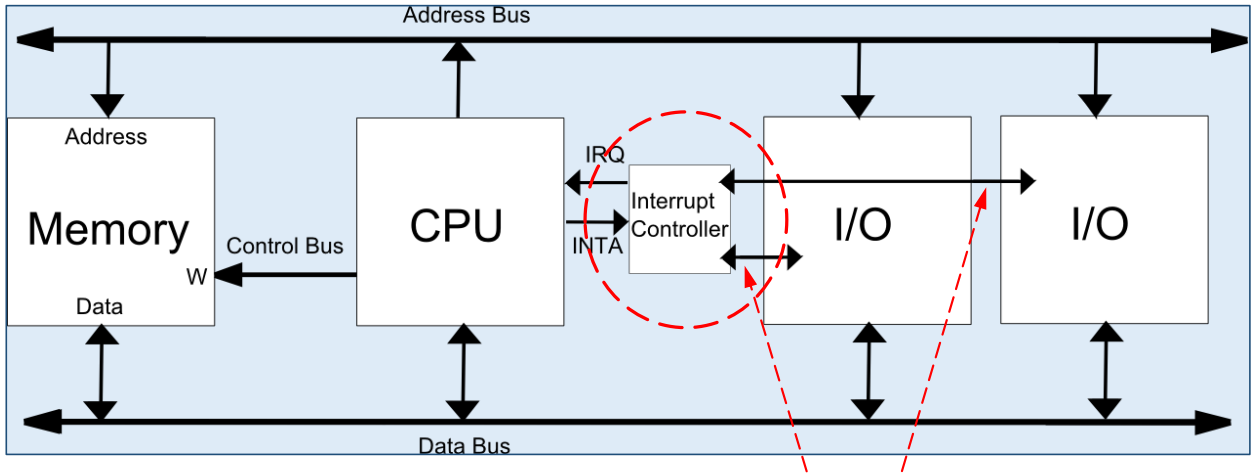
- es un árbitro que da prioridad a los dispositivos más críticos
- se conecta entre la CPU y los dispositivos de I/O



Controlador de interrupciones

Dispositivo que maneja múltiples interrupciones desde varios dispositivos de I/O con diferentes prioridades:

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

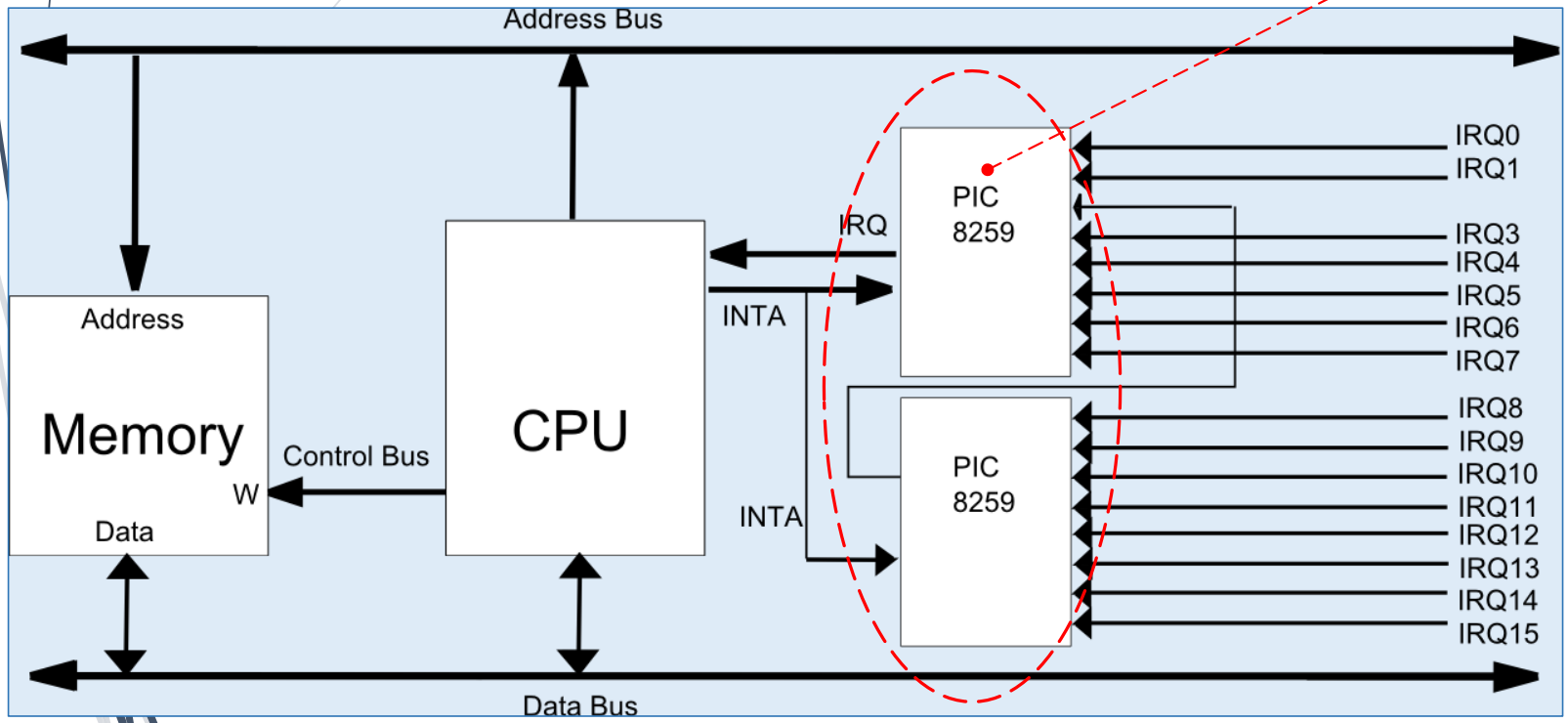


P.ej., el controlador de interrupciones 8259A:

- permite conectar hasta 8 controladores de dispositivos
- 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 8259A coloca en 1 su salida INT (en la figura, *IRQ*), 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 8259A especifica en el bus de datos cuál input causó la interrupción
- el hardware de la CPU usa este número como índice en su vectores de interrupción para encontrar la dirección de la función *ISR* que hay que ejecutar para manejar la interrupción

P.ej., la arquitectura x86 tradicional incluía un chip compuesto por dos controladores 8259A conectados en cascada, que permite conectar hasta 15 controladores de dispositivos de I/O

Programmable Interrupt Controller



IRQ0	timer
IRQ1	teclado
IRQ2	
IRQ3	puerto serial
IRQ4	puerto serial
IRQ5	puerto paralelo
IRQ6	floppy disk
IRQ7	puerto paralelo
IRQ8	Real Time Clock
IRQ9	uso libre
IRQ10	uso libre
IRQ11	uso libre
IRQ12	mouse
IRQ13	coproc. matemático
IRQ14	controlador de disco
IRQ15	conrrolador de disco

Ej. de varios dispositivos de I/O con distintas prioridades de interrupción
[A. Tanenbaum, T. Austin: "Structured Computer Organization", 6th ed., Pearson 2013, p. 416]

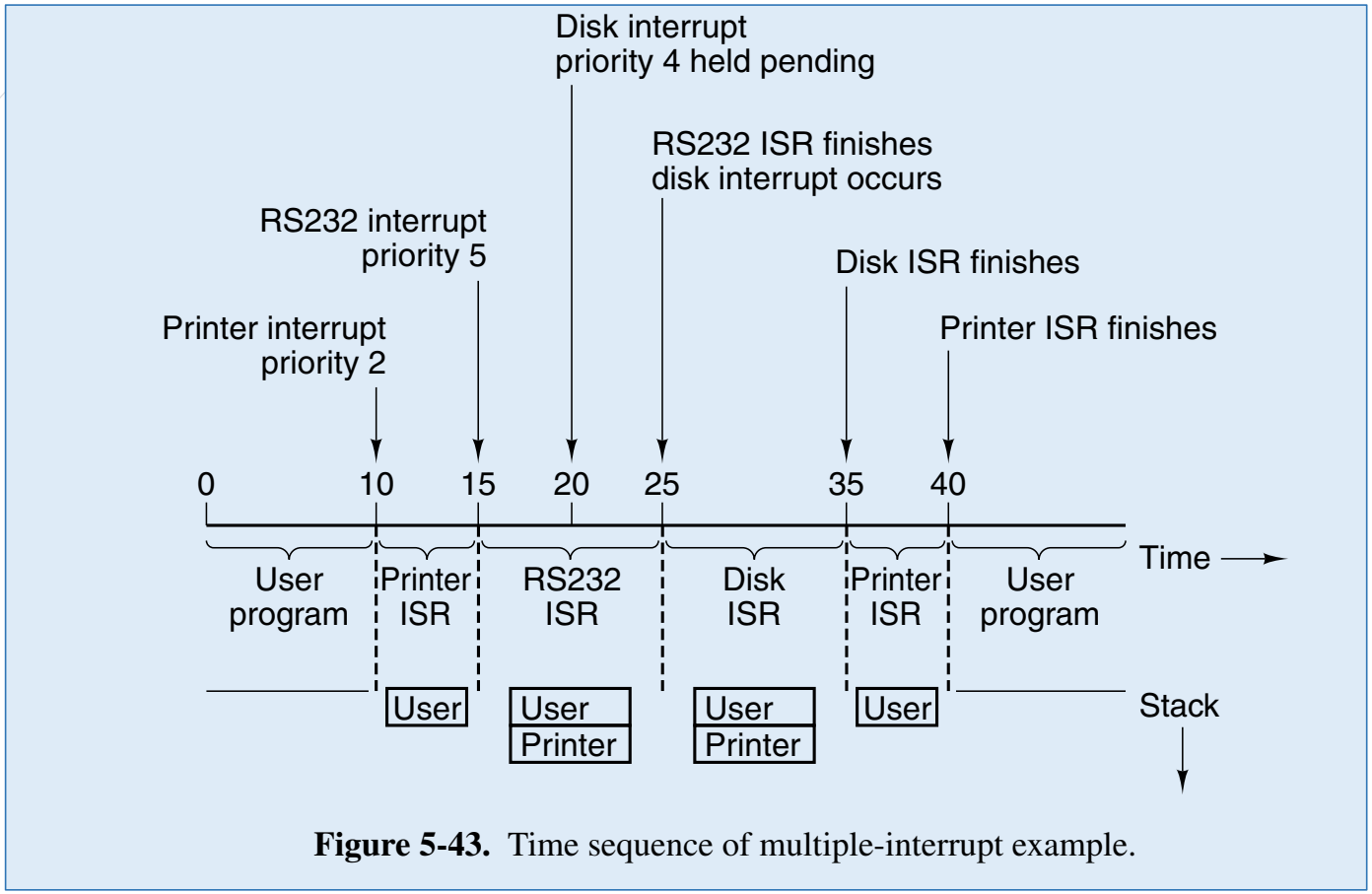


Figure 5-43. Time sequence of multiple-interrupt example.

Ej. de varios dispositivos de I/O con distintas prioridades de interrupción
[A. Tanenbaum, T. Austin: "Structured Computer Organization", 6th ed., Pearson 2013, p. 416]

CPU conectada a tres dispositivos: impresora, disco, y línea serial RS232, con prioridades 2, 4 y 5

t = 10: interrupción de la impresora, manejada por la ISR respectiva; el estado de la CPU, que está ejecutando un programa de usuario, va al stack

t = 15: interrupción de la RS232; tiene mayor prioridad que la impresora, por lo que es atendida, mientras que el estado de la CPU, que está ejecutando la ISR de la impresora, va al stack

t = 20: el disco interrumpe; su prioridad es menor que la de la RS232, por lo que queda pendiente hasta *t* = 25, cuando la ISR de la RS232 termina y la CPU vuelve al estado de ejecutar la ISR de la impresora; pero como la prioridad de esta última es 2, antes de que se ejecute una sola de sus instrucciones se acepta la interrupción del disco, con prioridad 4, y se ejecuta su ISR hasta su término en *t* = 35, cuando se reanuda la ISR de la impresora

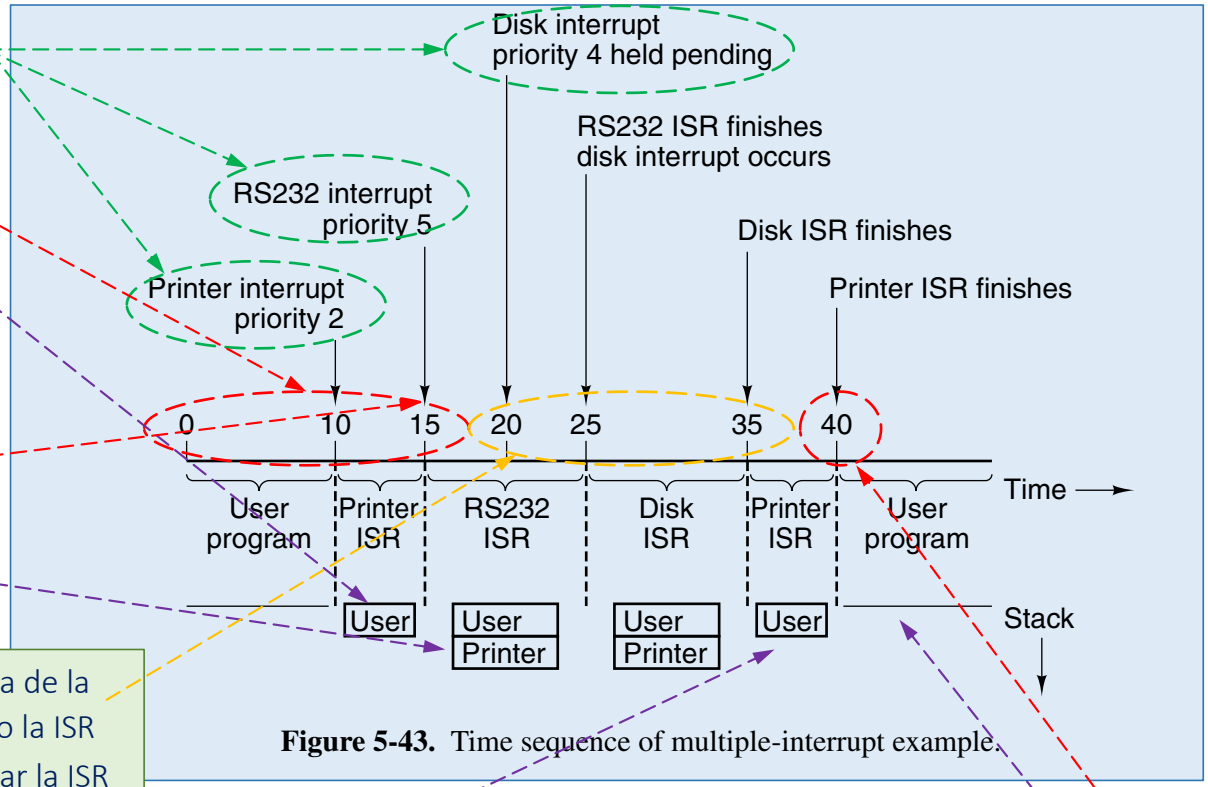


Figure 5-43. Time sequence of multiple-interrupt example.

t = 40: todas las ISRs han finalizado; la CPU reanuda el programa que estaba ejecutando cuando llegó la primera interrupción

5) Al emplear interrupciones, muchos aspectos del computador deben ser diseñados apropiadamente:

- el hardware de los dispositivos de I/O ya no opera bajo el control de la CPU, sino que independientemente; al terminar, debe poder interrumpir a la CPU
- el 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 cambia de un estilo secuencial y sincrónico en que el programador especifica cada paso de la operación del dispositivo, a un estilo asíncrono en que el programador escribe código para manejar las interrupciones (más generalmente, *eventos*)

El **controlador DMA** tiene registros que pueden ser escritos y leídos por la CPU:

- la dirección de memoria que se va a leer o escribir
- número (count) de bytes o palabras a transferir
- número (id) del dispositivo de I/O que se usará, o direcciones de los registros correspondientes
- tipo de la transferencia (lectura desde o escritura al dispositivo)
- la unidad de transferencia (byte o palabra)
- el número de bytes transferidos de una misma vez

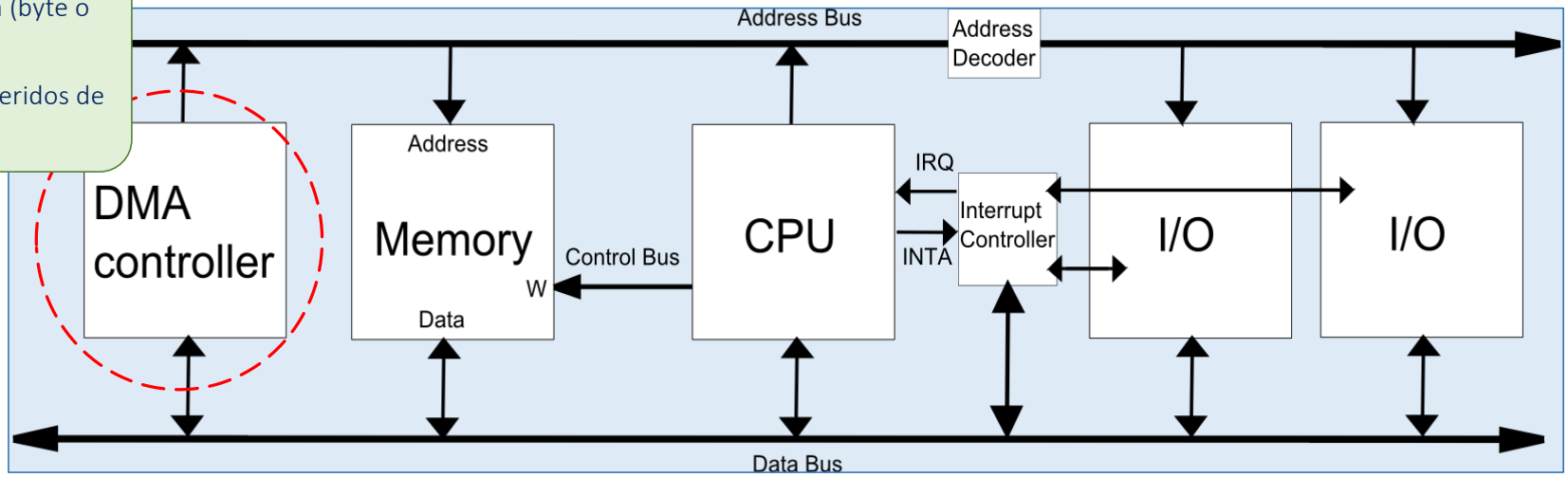
DMA I/O

Hasta aquí, todos las transferencias de datos entre la memoria y un dispositivo de I/O deben pasar por la CPU

En el esquema DMA I/O, añadimos un nuevo chip al sistema, un controlador DMA (*direct memory access*)

… que tiene acceso al bus independientemente de la CPU

… y permite que los dispositivos de I/O tengan acceso directo a la memoria —característica clave para tener I/O de alta velocidad



P.ej., para enviar (escribir) **32** bytes que parten en la dirección de memoria **100** al terminal cuyo *id* es **4**, la CPU sólo hace lo siguiente:

- escribe los valores **100**, **32** y **4** en los tres primeros registros
- escribe el código para ***write*** (p.ej., **1**) en el cuarto registro

… y queda libre para hacer otra cosa hasta que la transferencia completa esté terminada

… momento en el cual recibe una interrupción desde el controlador DMA

Una vez que la CPU ha inicializado los registros del controlador DMA, éste hace lo siguiente:

- hace una solicitud al bus para leer el byte 100 de la memoria
- hace una solicitud de I/O al dispositivo 4 para escribir allí el byte leído
- incrementa su registro address en 1 y decrementa su registro count en 1
- si count es > 0, repite las operaciones anteriores
- finalmente, cuando count = 0, deja de transferir bytes y pone un 1 en la línea de interrupción del bus (solicita interrumpir a la CPU)

La información fue transferida de la memoria al terminal sin usar la CPU

… y el terminal no interrumpió a la CPU en cada paso de la operación

P.ej., la operación de una transferencia DMA a memoria:

- la CPU, además de programar al controlador DMA (paso 1), ordena al controlador de disco que lea desde el disco a su *buffer* interno
- cuando los datos están en el *buffer*, comienza el **direct memory access** (pasos 2 a 4, que son repetidos hasta que *count* = 0)