ECSE324 : Computer Organization

Input/Output
Chapter 3, Chapter 7

Christophe Dubach
Fall 2021

Revision history: Warren Gross – 2017, Christophe Dubach – W2020&F2020, Brett H. Meyer – W2021, Christophe Dubach – F2021

Some material from Hamacher, Vranesic, Zaky, and Manjikian, *Computer Organization and Embedded Systems, 6 th ed*, 2012, McGraw Hill, and “Introduction to the ARM Processor using Altera Toolchain.”

Timestamp: 2021/10/20 10:26:00

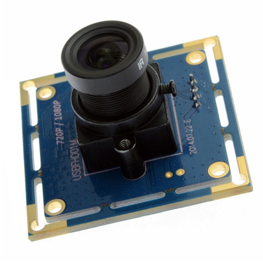
Disclaimer

Lectures are recorded live and posted unedited on *MyCourses* on the same day.

It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the textbook, or ask on the online forum for clarification.

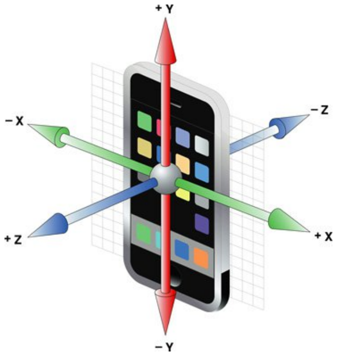
Introduction

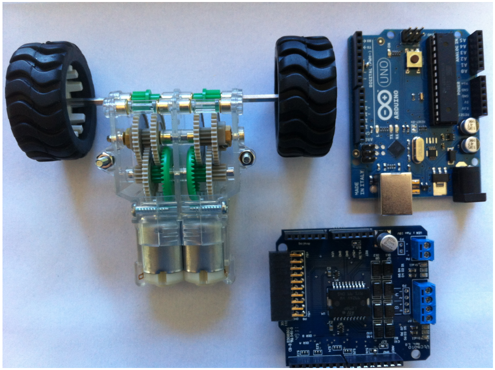
Input and Output: User Interfaces



Input and Output: Device Interfaces

Input and Output: Sensing and Actuating





Software Aspects of I/O

Software Aspects of I/O

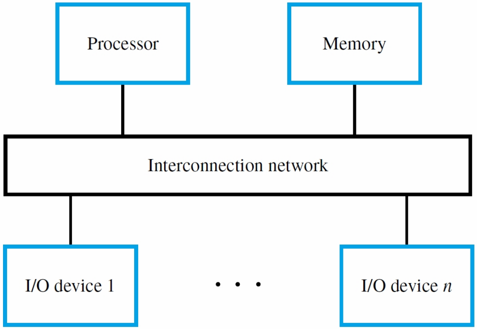
Memory-mapped Registers

Textbook§3.1, D.8.1

Accessing I/O Devices

From a programmers point of view, I/O is implemented as memory within same address space as code and data.

- HW view: computer system components communicate through an interconnection network
- SW view: to the CPU, the outside world is all memory

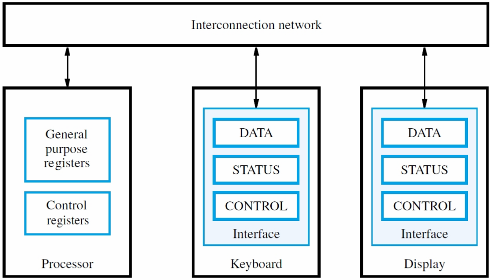


How does this work in practice?

I/O Device Interface

An I/O device interface is a circuit between a device and the interconnection network.

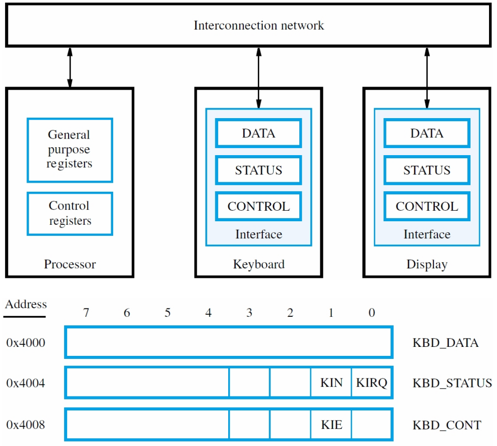
- Provides the means for data transfer and exchange of status and control information
- Includes data, status, and control registers accessible with load and store instructions



Memory-mapped I/O enables software to view these registers as locations in memory.

Memory-mapped I/O

I/O device registers are memory-mapped if they they are accessible with a load/store instruction.

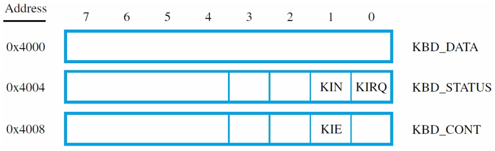


Address	Content
	...
0x4000	KBD_DATA
0x4004	KBD_STATUS
0x4008	KBD_CONT
	...
0x4010	DISP_DATA
0x4014	DISP_STATUS
0x4018	DISP_CONT
	...

Memory-mapped I/O

Locations associated with I/O devices are accessed with load and store instructions; addresses are saved in the text region.

```
kbd: .word 0x00004000   // keyboard base address
   ...
   LDR   R0, kbd        // read the base address
   LDR   R1, [R0, #0]   // read data register
   ...
   STR   R2, [R0, #8]   // write control register
```



Note: bit manipulation is required to set or check individual bits.

I/O Synchronization

We need to synchronize the timing of I/O devices and the processor.

- When does an input device have data ready for the processor to read?
- When is an output device ready to receive data written by the processor?

I/O Synchronization

We need to synchronize the timing of I/O devices and the processor.

- When does an input device have data ready for the processor to read?
- When is an output device ready to receive data written by the processor?

E.g., read keyboard characters, store them in memory, and display them on screen.

- A keyboard's data input rate (keyboard to processor) is likely to be at most a few characters per second.
- The rate of character output (processor to display) is likely to be much faster, *e.g.*, thousands of characters per second.
- The processor can execute many millions of instructions per second, much faster than the display can accept data!

How do we coordinate actions across such disparate time scales?

Software Aspects of I/O

Polling

Textbook§3.1, D.8.1

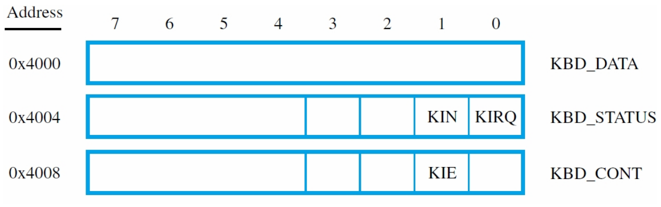
Program-controlled I/O

Assume that I/O devices have a way to send a `ready` signal to the processor.

- For the keyboard, this indicates that a character can be read; the processor responds with a load from the keyboard's data register.
- For the display, this indicates that a character can be sent; the processor responds with a store to the display's data register.

The `ready` signal is a status flag in the status register that is polled, or repeatedly checked, by the processor. This is referred to as *polling*.

Polling I/O: Reading

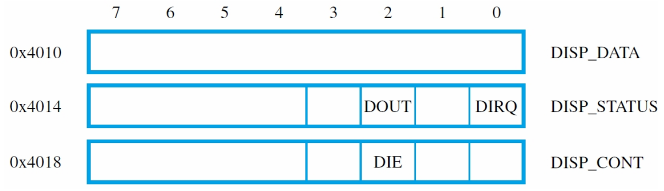


Before reading data, we need to check that data is ready.

- `KDB_STATUS` is accessible at 0x4004, and has a 1-bit flag `KIN` in bit 1
- The processor reads `KDB_STATUS`, and checks if `KIN` is 1 or 0
- If `KIN` is 1, the processor reads `KBD_DATA` at 0x4000

How would you implement this in assembly?

Polling I/O: Writing



Writing follows a similar process to reading.

- `DISP_STATUS` is accessible at 0x4014, and has a 1-bit flag **DOUT** in bit 2
- The processor reads `DISP_STATUS`, and checks if **DOUT** is 1 or 0
- IF **DOUT** is 1, the processor writes `DISP_DATA` at 0x4010

What happens if you don't check a device's status register before reading or writing?

Waiting for your bit to come in

Program-controlled I/O is implemented by polling in a loop, referred to as busy-waiting or spin-waiting. *E.g.*,

- Assume the keyboard circuit places a character in `KBD_DATA` and sets `KIN` in `KBD_STATUS`
- The circuit clears the `KIN` flag when `KBD_DATA` is read*

```
kbd: .word 0x00004000   // keyboard base address
   ...
   LDR   R0, kbd        // read the base address
READWAIT:
   LDRB  R1, [R0, #4]   // read KBD_STATUS
   TST   R1, #2         // do R1 & 0b00000010, set CPSR
   BEQ   READWAIT       // spin while Z=1
   LDRB  R3, [R0]       // done spinning, read data
```

Waiting for your bit to come in

Program-controlled I/O is implemented by polling in a loop, referred to as busy-waiting or spin-waiting. *E.g.*,

- Assume the keyboard circuit places a character in `KBD_DATA` and sets `KIN` in `KBD_STATUS`
- The circuit clears the `KIN` flag when `KBD_DATA` is read*

```
kbd: .word 0x00004000   // keyboard base address
   ...
   LDR   R0, kbd        // read the base address
READWAIT:
   LDRB  R1, [R0, #4]   // read KBD_STATUS
   TST   R1, #2         // do R1 & 0b00000010, set CPSR
   BEQ   READWAIT       // spin while Z=1
   LDRB  R3, [R0]       // done spinning, read data
```

* How does it know?

Waiting for your bit to come in

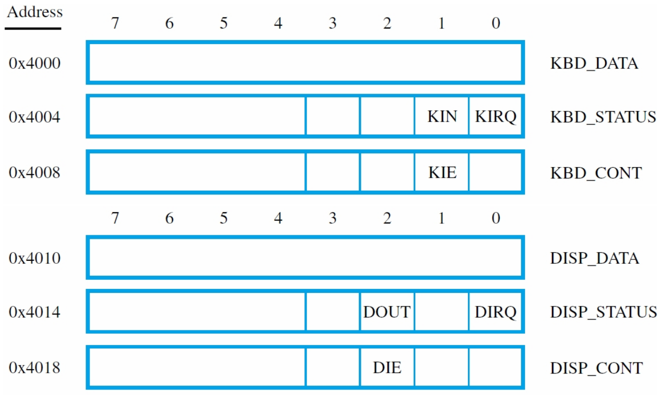
Once again, waiting to write is similar to waiting to read. *E.g.*,

- Assume the display circuit sets DOUT in DISP_STATUS after the previous character has been displayed
- The circuit clears DOUT when DISP_DATA is written

```
disp: .word 0x00004010   // display base address
   ...
   LDR   R0, disp        // read the base address
WRITEWAIT:
   LDRB  R1, [R0, #4]    // read DISP_STATUS
   TST   R1, #4          // do R1 & 0b00000100, set CPSR
   BEQ   WRITEWAIT       // spin while Z=1
   STRB  R3, [R0]        // done spinning, write data
```

Example: Echo

Let’s look at a program that uses polling to read, store, and display a line of characters (“echo” to the display). The program finishes when the carriage return (CR) character is entered on the keyboard.



Example: Echo

```
kbd:   .word 0x00004000  // keyboard base address
disp: .word 0x00004010   // display base address
buf: .space 80           // a buffer of 80 characters
equ CR,     13
   ...
   LDR   R0, kbd         // keyboard base address
   LDR   R1, disp        // display base address
   LDR   R2, =buf        // save characters in buf
READ:
   LDRB  V1, [R0, #4]    // read KBD_STATUS
   TST   V1, #2          // do R1 & 0b00000010, set CPSR
   BEQ   READ            // spin while Z=1
   LDRB  V2, [R0]        // done spinning, read character
   STRB  V2, [R2], #1    // save the character
ECHO:
   LDRB  V1, [R1, #4]    // read DISP_STATUS
   TST   V1, #4          // do R1 & 0b00000100, set CPSR
   BEQ   ECHO            // spin while Z=1
   STRB  V2, [R1]        // done spinning, write character
   TEQ   V2, #CR         // if not CR ...
   BNE   READ            // ... read more characters
```

Software Aspects of I/O

Interrupts

Textbook§3.2, D.7, D.8.2

Polling is terribly inefficient

What do you do if you are waiting for a pizza to arrive while studying?

Polling?

No, you wait for the delivery to *interrupt* you

Interrupts

Polling has several major drawbacks. One is that the processor is kept busy while it waits.

- The CPU executes continuously until the flag is changed
- The CPU cannot be used for other tasks in the meantime
- This wastes time, it wastes energy, …

Using interrupts, I/O hardware asserts an interrupt request signal (`IRQ`) when it is ready.

- The CPU only interacts with the device when it is useful to do so
- Otherwise, the CPU works on other tasks
  $\Rightarrow$ *improving performance*
- Or, the CPU can go to sleep
  $\Rightarrow$ *improving energy efficiency*

Interrupts are essential when timing is important

Consider a task that records and compresses video and audio.

- Video, audio are sampled at different rates (*e.g.*, 60 Hz, 44.1 kHz)
- Compression complexity is different for each, too
- Coordinating multiple polling tasks requires significant programmer effort: we cannot miss any samples
  - Time spent computing must be measured and managed
  - How often the CPU waits for each data source must be balanced
  - Everything is harder when computational complexity is data dependent (as is the case in compression)

Interrupts are essential when timing is important

Managing multi-rate data and tasks with different and variable complexity is much simpler with interrupts.

- Set a timer for each task: *e.g.*, 167 ms for one, 23 us for the other
- When a timer goes off, an interrupt service routine (ISR) is executed in response
- When the ISR finishes, the processor returns to whatever it was computing before

Using interrupts improve performance, energy efficiency, robustness, and portability.

Interrupt Service Routines (ISRs)

Each interrupt has associated with it an ISR. Interrupts may occur at any time: unlike subroutines, ISRs may be executed at any time.

If an interrupt occurs when the processor is executing instruction i:

- This instruction finishes
- The processor saves its state and executes the ISR
- When the ISR returns, state is restored, and PC is set to instruction $i + 1$

Enabling and Disabling Interrupts

There may be times when we do not want to respond to interrupts.

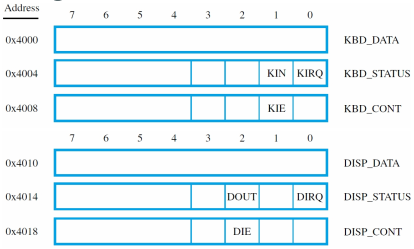
- *E.g.*, in different modes, or different use cases: there's no need to compress audio if audio is muted!
- When handling an interrupt
- When working with shared resources or data

We need a way to disable interrupts.

- A bit in the processor status register can globally disable interrupts (*i.e.*, ignore IRQ signals)
- A bit in an I/O device's control register can disable interrupts from that device
- ARM provides support for disabling groups of interrupts

Device Registers

The interrupt behavior of devices is controlled through their registers.



- `KIE` and `DIE` enable interrupts from these devices
- `KIRQ` and `DIRQ` indicate an interrupt has been requested; these bits must be cleared by the ISR

Basic Interrupt Servicing Event Sequence

Assuming that interrupts are enabled (IE bit is set in the processor status register CPSR):

- A devices raises an interrupt request (IRQ)
- The processor completes the instruction it was executing
- The processor saves its state*, *e.g.*, PC and CPSR registers (on the stack)
- Interrupts are temporarily disabled by reseting (clearing) IE
- PC is set to the first instruction of the ISR
- ISR executes, performing requested processing and acknowledging the IRQ (device deasserts its IRQ)
- PC and CPSR are restored (and thus IE is set again)

Basic Interrupt Servicing Event Sequence

Assuming that interrupts are enabled (IE bit is set in the processor status register CPSR):

- A devices raises an interrupt request (IRQ)
- The processor completes the instruction it was executing
- The processor saves its state*, *e.g.*, PC and CPSR registers (on the stack)
- Interrupts are temporarily disabled by reseting (clearing) IE
- PC is set to the first instruction of the ISR
- ISR executes, performing requested processing and acknowledging the IRQ (device deasserts its IRQ)
- PC and CPSR are restored (and thus IE is set again)

If an ISR is going to use any general-purpose registers, what must it do before it does so?

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 1: How does the processor know which device is requesting an interrupt?

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 1: How does the processor know which device is requesting an interrupt?

Answer 1: One simplistic strategy is to poll the IRQ bit in each device's status register.

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 1: How does the processor know which device is requesting an interrupt?

Answer 1: One simplistic strategy is to poll the IRQ bit in each device's status register.

This makes sense if we have a single IRQ pin, and are targeting very, very low-cost hardware.

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 2: How is the starting address for the correct ISR determined?

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 2: How is the starting address for the correct ISR determined?

Answer 2: When polling devices to find a set IRQ bit, call the device-specific routine whenever a set IRQ bit is encountered. Service this interrupt (which resets the IRQ bit), and return to polling.

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 2: How is the starting address for the correct ISR determined?

Answer 2: When polling devices to find a set IRQ bit, call the device-specific routine whenever a set IRQ bit is encountered. Service this interrupt (which resets the IRQ bit), and return to polling.

But the whole point of interrupts is to avoid polling, right? Also, this approach isn't particularly portable.

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 2: How is the starting address for the correct ISR determined?

Answer 2: When polling devices to find a set IRQ bit, call the device-specific routine whenever a set IRQ bit is encountered. Service this interrupt (which resets the IRQ bit), and return to polling.

But the whole point of interrupts is to avoid polling, right? Also, this approach isn't particularly portable.

A much more efficient solution is vectored interrupts.

A Quick Aside: Interrupts? Or Exceptions?

You've encountered exceptions before: null pointer exceptions, index out of bounds exceptions, segmentation faults, etc.

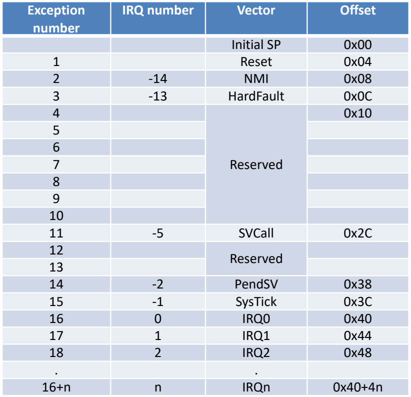
Exceptions are a special case of interrupt as they are caused by events in the processor itself (rather than an external device).

In the context of vectored interrupts, interrupts (from external devices) and exceptions (occurring within the processor) are handled in the *same way* ...

Vectored Interrupts

With vectored interrupts, each possible interrupt (exception) is assigned a number, and that number is used to index into a special table in memory, the interrupt vector table, which stores all ISR starting addresses.

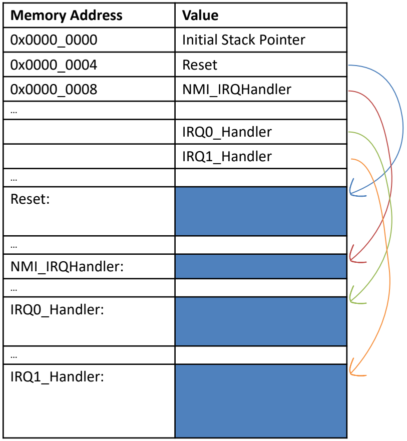
- When an interrupt occurs, the processor gets the identifying number for the interrupt.
- The vector table is located at a fixed location in memory (in ARMv7, starting at `0x0000 0004` by default).
- *E.g.*, if interrupt number *n* has occurred, `0x0000 0004 + n*4` has the PC value for the desired ISR.



Exception number	IRQ number	Vector	Offset
		Initial SP	0x00
1		Reset	0x04
2	-14	NMI	0x08
3	-13	HardFault	0x0C
4			0x10
5			
6			
7		Reserved	
8			
9			
10			
11	-5	SVCall	0x2C
12		Reserved	
13			
14	-2	PendSV	0x38
15	-1	SysTick	0x3C
16	0	IRQ0	0x40
17	1	IRQ1	0x44
18	2	IRQ2	0x48
.		.	
16+n	n	IRQn	0x40+4n

Vectored Interrupts

- The vector table simplifies ISR lookup: no polling is needed.
- The vector table also improves portability: different ISRs for different devices are easily combined; ISRs can be stored anywhere in memory.



Memory Address	Value
0x0000_0000	Initial Stack Pointer
0x0000_0004	Reset
0x0000_0008	NMI_IRQHandler
…	
	IRQ0_Handler
	IRQ1_Handler
…	
Reset:	🟦
…	
NMI_IRQHandler:	🟦
…	
IRQ0_Handler:	🟦
…	
IRQ1_Handler:	🟦

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 3: Should a device be allowed to interrupt the processor while another interrupt is being serviced?

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 3: Should a device be allowed to interrupt the processor while another interrupt is being serviced?

Answer 3: Some devices need to be serviced quickly, even if it means interrupting a currently executing ISR!

- Assign a priority to each interrupt
- The processor will switch to a higher priority interrupt (saving PC and CPSR on the stack, first) if it occurs when servicing a lower priority interrupt
- Lower priority interrupts are temporarily ignored

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 4: What if there are two simultaneous requests with the same priority?

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 4: What if there are two simultaneous requests with the same priority?

Answer 4: Arbitration.

- Polling? Polling order determines service order.
- Vectored interrupts? Arbitration hardware facilitates the selection of an interrupt to service.

Handling Multiple Devices

Things are more complicated if there are multiple devices that might request interrupt servicing.

Question 4: What if there are two simultaneous requests with the same priority?

Answer 4: Arbitration.

- Polling? Polling order determines service order.
- Vectored interrupts? Arbitration hardware facilitates the selection of an interrupt to service.

What about real devices? ARM processors are heavily optimized for interrupt processing, especially cases where multiple interrupt requests are outstanding.

Exception Handling

Exceptions generally indicate an event has occurred which requires special attention.

- Errors of various sorts: divide by zero, invalid instruction, etc
- Some other condition in software (*e.g.*, the throw-catch pattern in Java)
- Debugging: *e.g.*, upon reaching a breakpoint
- OS: exceptions are used to change between threads and processes

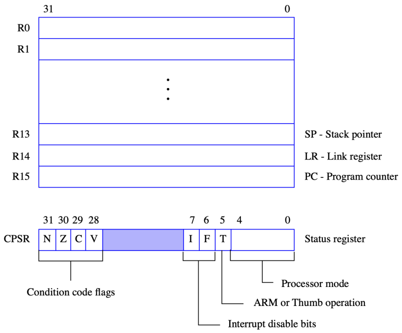
Exception Handling

Exceptions are handled with ISRs, too.

- Errors are most often reported to the user (*e.g.*, segmentation fault); execution then ends
- If an instruction causes an exception (*e.g.*, divide by zero), that instruction is not allowed to complete like when interrupts occur
- Such exceptions require that the return address (PC) be modified accordingly (possibly re-executing the offending instruction)

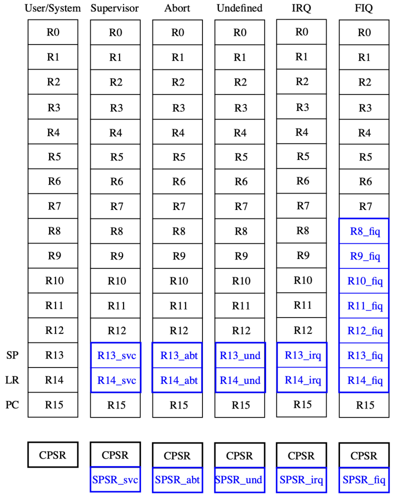
ARM Processor Modes

- The ARM processor has seven operating modes that determine what system resources a program has access to.
- When an interrupt occurs, the processor switches into one of two modes:
    - IRQ mode: entered when a normal interrupt is received
    - FIQ mode: entered in response to a fast interrupt request*



Banked Registers

- Some modes use shadow registers instead of the usual registers
- *E.g.*, `IRQ` mode: accesses to `R13` are to `R13_irq` instead
- This avoids saving/restoring some registers, reducing the time to enter an ISR
- *E.g.*, `FIQ` mode: no need to save `R8`–`R12`, reducing the time spent in an ISR



	User/System	Supervisor	Abort	Undefined	IRQ	FIQ
	R0	R0	R0	R0	R0	R0
	R1	R1	R1	R1	R1	R1
	R2	R2	R2	R2	R2	R2
	R3	R3	R3	R3	R3	R3
	R4	R4	R4	R4	R4	R4
	R5	R5	R5	R5	R5	R5
	R6	R6	R6	R6	R6	R6
	R7	R7	R7	R7	R7	R7
	R8	R8	R8	R8	R8	R8_fiq
	R9	R9	R9	R9	R9	R9_fiq
	R10	R10	R10	R10	R10	R10_fiq
	R11	R11	R11	R11	R11	R11_fiq
	R12	R12	R12	R12	R12	R12_fiq
SP	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
LR	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	R15	R15	R15	R15	R15	R15
	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Returning from Interrupts in ARM

Recall that in PC-relative addressing, PC doesn't point to the executing instruction:

`i`		(currently executing instruction)
`i+1`		(instruction in decode)
`i+2`	$\leftarrow$ PC	(instruction being fetched)

To return from an ISR, fetch instruction `i+1` by decrementing `LR`*:

```
SUBS  PC, LR, #4
```

Returning from Interrupts in ARM

Recall that in PC-relative addressing, PC doesn't point to the executing instruction:

`i`		(currently executing instruction)
`i+1`		(instruction in decode)
`i+2`	$\leftarrow$ PC	(instruction being fetched)

To return from an ISR, fetch instruction `i+1` by decrementing `LR`*:

```
SUBS  PC, LR, #4
```

*CPSR is restored automagically when PC is destination and S flag set

Hardware Aspects of I/O

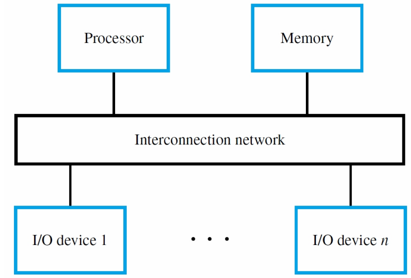
Hardware Aspects of I/O

Bus Protocols

Textbook§7.1, 7.2

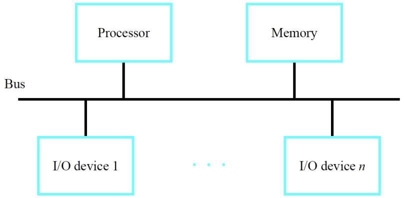
Interconnection Networks

- An interconnection network is used to transfer data among the processor, memory, and I/O devices
- A commonly-used interconnection network is called a bus



Single-bus System

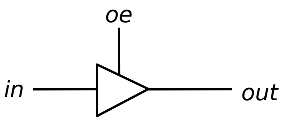
- A bus is a set of shared wires
- Only one source may drive the bus at any one time
- Hardware manages access to the bus to enforce this constraint

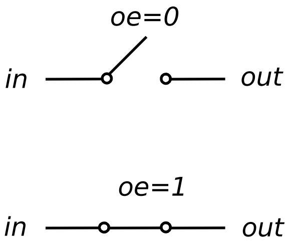


Tri-state Buffers

Buses are driven (pulled to Vdd or GND) by tri-state buffers. Tri-state buffers operate as follows:

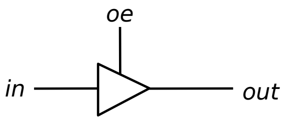
- When the control signal output enable `oe` is low, the buffer is disconnected from the output
- When `oe` is high, the buffer drives `in` onto `out`
- The disconnected state `Z` is high impedance

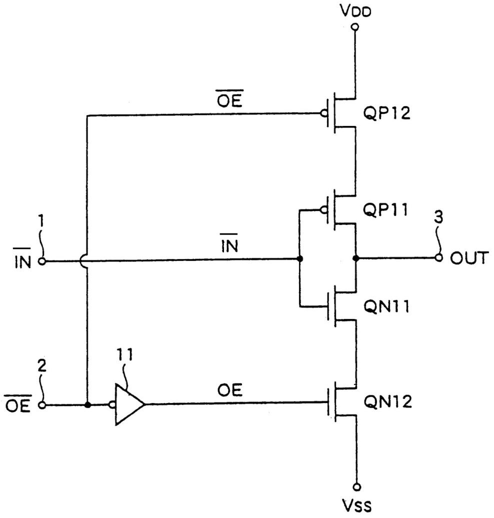




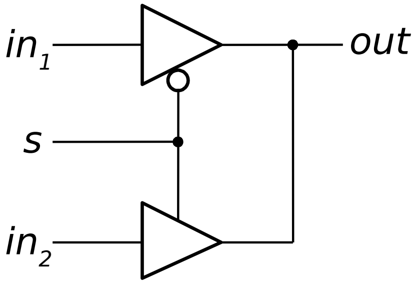
oe	in	out
0	0	Z
0	1	Z
1	0	0
1	1	1

Tri-state Buffers in CMOS



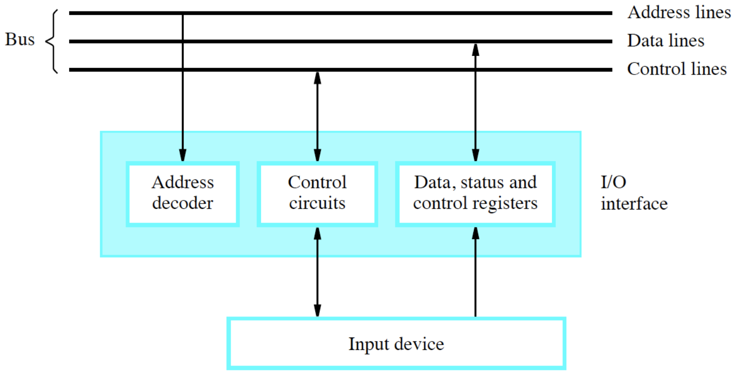


Tri-state buffers can also select among multiple inputs



Bus interface for an input device

Buses consist of address, data, and control wires. These wires are connected to different circuitry in the I/O interface for a device, and provide access to registers.



Bus Protocols

A bus protocol is a set of rules that govern exchanges of information between devices on a bus.

- Bus masters initiate communication on the bus; bus slaves respond accordingly
    - The processor is often, but not always, the master in an exchange
- Control signals indicate what and when actions are to be taken
- Address signals indicate which bus-connected resources are requested to participate in an exchange
- Data signals are used to for the exchange itself

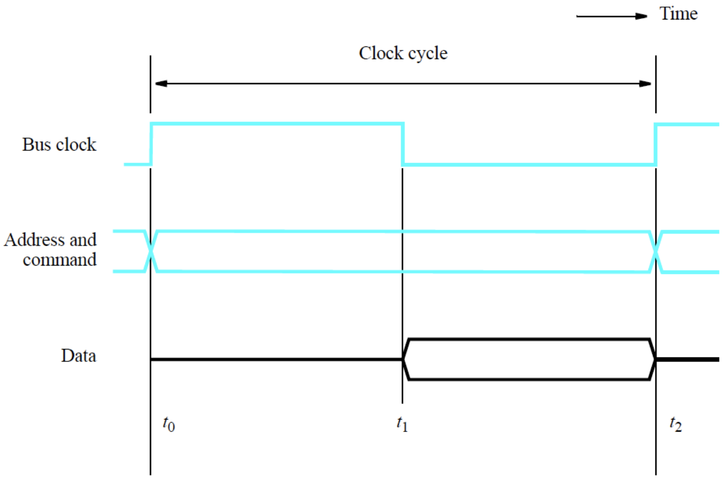
Bus Control Signals

Control signals set the parameters for an exchange of information on the bus.

- The $R/\overline{W}$ control signal specifies whether a read (1) or write (0) is being performed
- Data exchange size (byte, halfword, word, etc) is indicated with other signals
- Yet others specify timing information, depending on the type of bus
    - Synchronous bus: a clock signal synchronizes all devices
    - Asynchronous bus: devices synchronize using changes in special control signals

Synchronous Read Access Timing

1. Master sets address and control signals at t_0 (rising clock)
2. Slave responds on data signals at t_1 (falling clock)
3. Master reads data signals at t_2 (rising clock)



Synchronous Write Access Timing

Write functions similarly:

1. Master sets address, control and data signals at t_0 (rising clock)
2. At t_2, the device captures the data in a register (rising clock)

Synchronous Write Access Timing

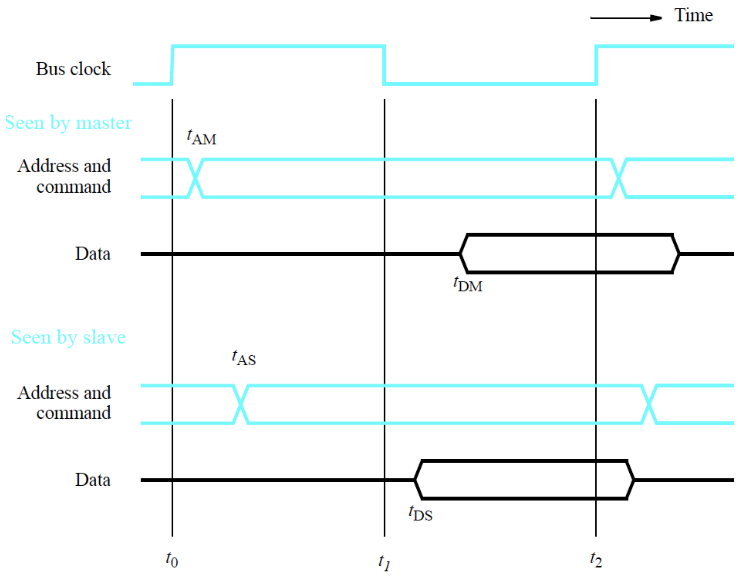
Write functions similarly:

1. Master sets address, control and data signals at t_0 (rising clock)
2. At t_2, the device captures the data in a register (rising clock)

Note that signals propagate to different devices at different times, depending on their location on the bus.

- We generally assume that changes in the bus clock are seen at all devices at the same time
- System designers spend a lot of time and energy making sure this is true

Detailed Synchronous Read Access Timing



Bus Timing Caveats

A few important notes about bus timing:

- Not all devices operate at the same speed
- $t_2 - t_0$, the bus cycle, must be long enough to accommodate the longest possible delay on the bus and slowest device interface
- All devices operate at the speed of the slowest device

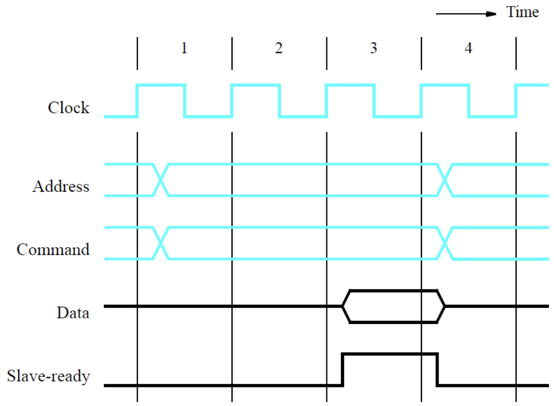
Bus masters assume that data is valid at t_2; what happens if there has been a malfunction?

Multi-cycle Data Transfers

To address both of these issues, most bus protocols include a device ready, or acknowledgment, signal.

- An acknowledgment signal indicates that the address was successfully decoded, and that the device is ready to participate in the requested data transfer
- This signal can also be used to adjust the delay of the transfer operation
- *E.g.*, a transfer can be allowed to span multiple bus cycles

Multi-cycle Read Access Timing



CC1: Master initiates read access, slave decodes address and control signals

CC2: Slave accesses its data

CC3: Data is ready, driven onto bus, and `ready` is asserted

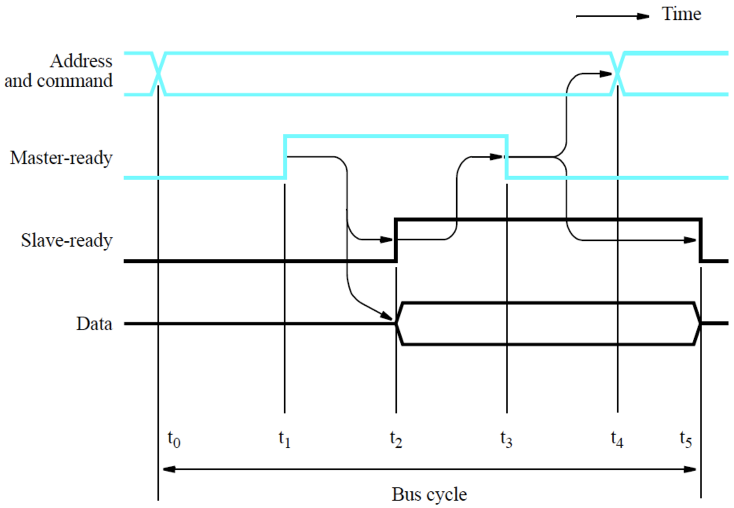
CC4: Slave deasserts `ready`, and master may initiate a new transfer

Asynchronous Bus Protocol

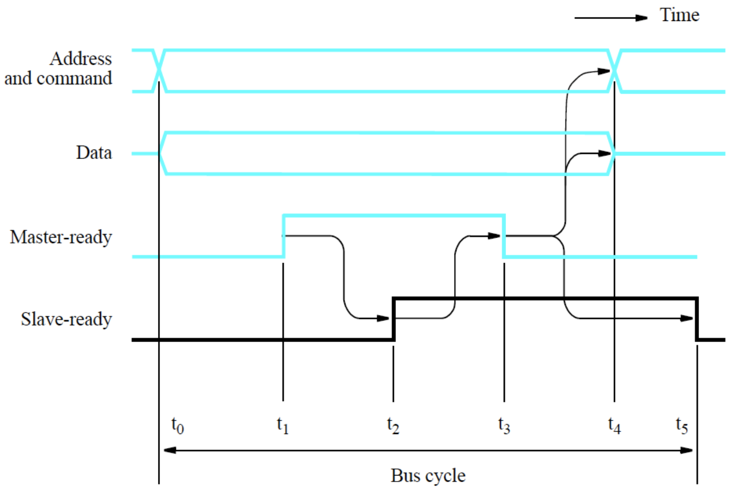
Asynchronous buses do not use a clock signal to synchronize devices.

- Timing automatically adjusts to delays
- A handshake protocol is used to coordinate between devices
- If each signal change results in a response, this is called full handshake or fully interlocked handshaking
- Data transfer is controlled by `ready` signals for the master and slave devices
- Whenever a master makes a request, it waits for a response before taking the next action

Asynchronous Read Access Timing



Asynchronous Write Access Timing



Synchronous vs. Asynchronous Buses

Synchronous buses	Asynchronous buses
Require careful design to ensure timing constraints are met	Flexibly adjust to the timing of each device automatically; this is especially useful for long buses
Transfers only require two end-to-end delays (one round trip)	Transfers require four end-to-end delays (two round trips)
Used in high-speed interconnect between devices that are close together (mm to m)	Used in cars, airplanes, and factories (where buses may extend up to 1 km)

Hardware Aspects of I/O

Arbitration

Textbook§7.3

Arbitration: granting access to a shared resource

Typical bus-based systems have multiple components that may act as master. *E.g.*,

- There are multiple processors on the same bus, or
- There are multiple components capable of writing to memory (*e.g.*, a processor and a DMA, or direct memory access module)

Only one component is allowed to initiate a bus request at a time; bus-based systems need policies for determining which components have permission to do so at what times.

Bus Arbitration

Each different type of bus will specify a different protocol for arbitration.

There are two basic types of arbitration:

- bus-based arbitration, and
- cooperative arbitration

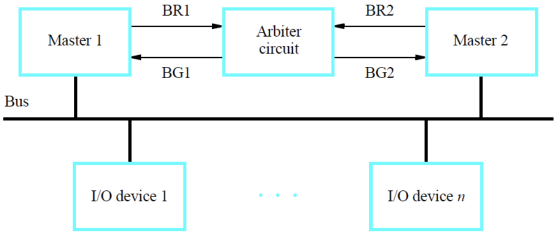
In bus-based arbitration, special circuitry determines which master device can next initiate a request.

In cooperative arbitration, master devices achieve consensus on which device can next initiate a request.

Bus Arbitration Using Arbitration Hardware

In bus-based arbitration:

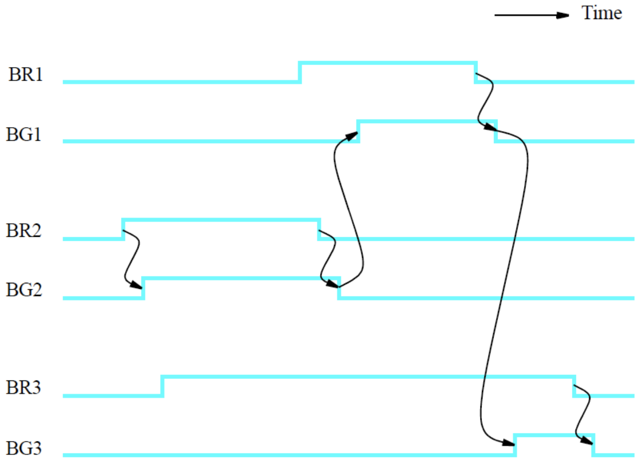
- Devices request permission to use the bus
- An arbiter circuit grants a access to device based on an arbitration policy
- The bus that is granted access carries out its request



Device priority is one common approach to determining which device should be granted access.

Priority-based Arbitration

In this example, the priorities are BR1 > BR2 > BR3.

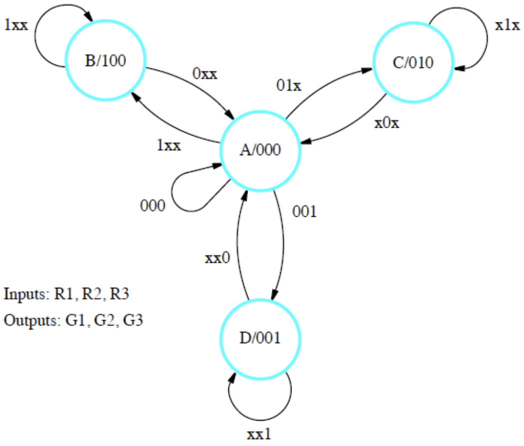


Textbook Example 7.2

An arbiter receives three request signals, R1, R2, and R3, and generates three grant signals, G1, G2, G3. R1 has the highest priority; R3 has the lowest.

Draw a state diagram that describes the behavior of this arbiter.

Solution



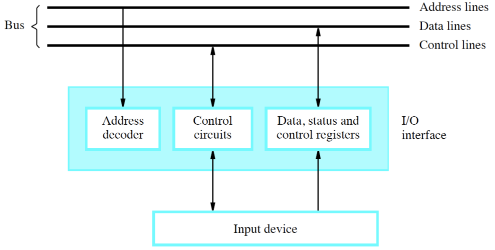
B/100
C/010
A/000
D/001
1xx
0xx
01x
x1x
x0x
1xx
000
001
xx0
xx1
Inputs: R1, R2, R3
Outputs: G1, G2, G3

Hardware Aspects of I/O

Parallel and Serial Interfaces

Textbook§7.4, 7.5

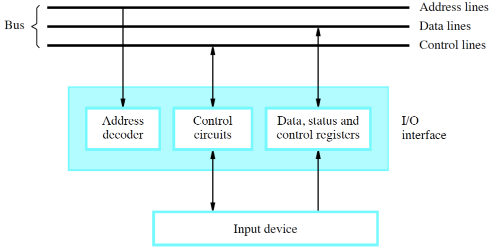
I/O Ports



Recall: an I/O port (interface) connects a device to a bus.

- Parallel ports transfer several bits simultaneously
- Serial ports transfer bits one* at a time

I/O Ports

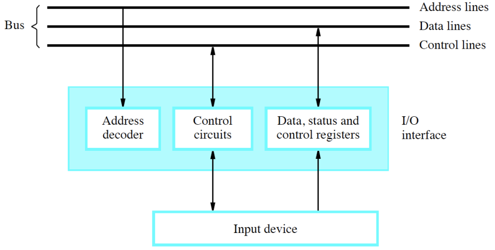


Recall: an I/O port (interface) connects a device to a bus.

- Parallel ports transfer several bits simultaneously
- Serial ports transfer bits one* at a time

* Communication with the processor is still parallel: conversion from parallel to serial happens inside the interface circuit.

I/O Ports



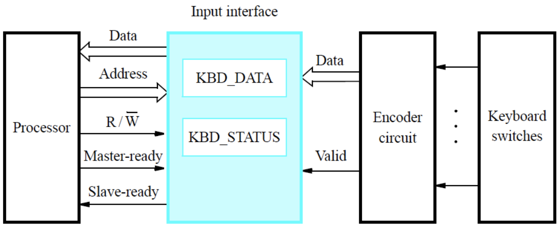
Recall: an I/O port (interface) connects a device to a bus.

- Parallel ports transfer several bits simultaneously
- Serial ports transfer bits one* at a time

* Communication with the processor is still parallel: conversion from parallel to serial happens inside the interface circuit.

* Some “serial” interfaces use multiple data lines, but still transfer data in multiple bus cycles.

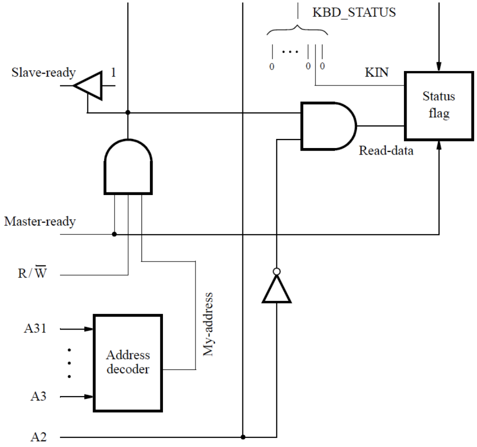
Input Interface: Keyboard to Processor



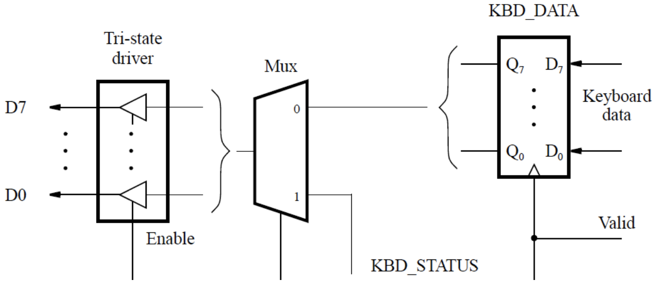
- A debouncing circuit ensures key presses are signaled just once
- When `Valid` rises, 8-bit `Data` is sampled by `KBD_DATA` and `KBD_STATUS.KIN` $\leftarrow$ 1.
- `Valid` later falls (*only to rise again*).
- When the processor reads `KBD_DATA` (asynchronously), `KBD_STATUS.KIN` $\leftarrow$ 0.

Input Interface: Keyboard to Processor

- `A1` and `A0` are not used: accesses are word-aligned.
- When R, `My-address`, `Master-ready` are asserted, the keyboard drives the bus.
- When `A2` is 0, `KIN` is cleared (because `KBD_DATA` is read).

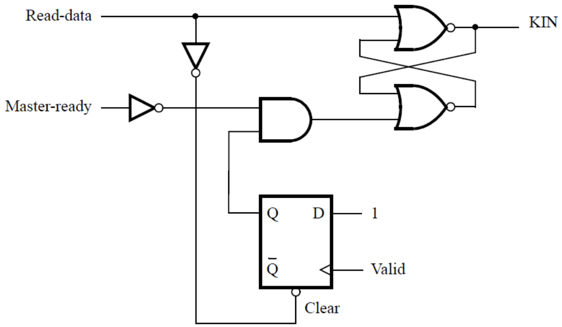


Input Interface: Keyboard to Processor



- A2 selects between KBD_DATA and KBD_STATUS.
- Valid triggers KBD_DATA capture, and sets KBD_STATUS.KIN.

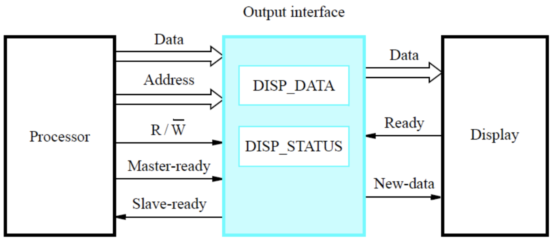
Status Flag Control Circuit



S	Read-data	KIN
0	0	hold
0	1	0
1	0	1
1	1	X

`KIN` is set by `Valid` and cleared by a read operation, but only when `Master-ready` is not asserted.

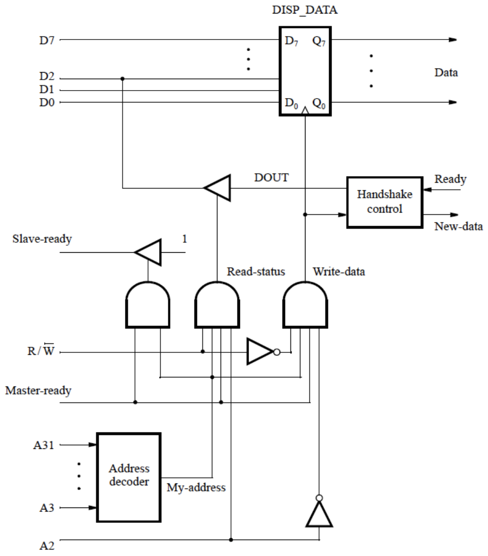
Output Interface: Processor to Display



- When the display asserts `Ready`, `DISP_STATUS.DOUT` $\leftarrow$ 1.
- When the processor observes this, a character is sent to `DISP_DATA`.
- Then `DISP_STATUS.DOUT` $\leftarrow$ 0, and `New-data` $\leftarrow$ 1.
- Finally, `Ready` $\leftarrow$ 0, and the character is displayed.

Output Interface: Processor to Display

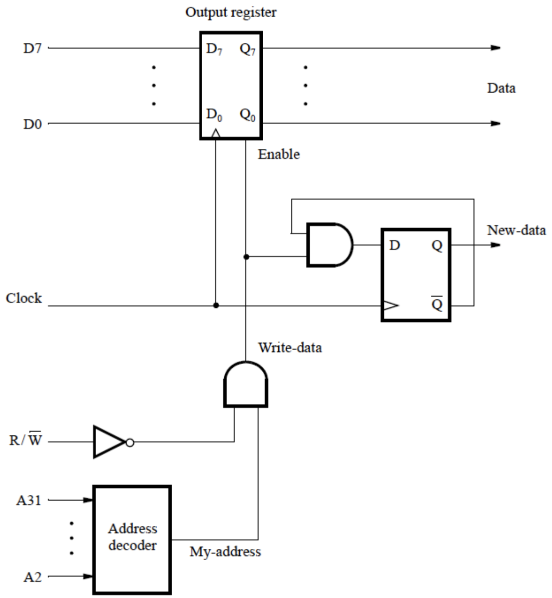
- When A2 and R are 0, bus data is sampled by DISP_DATA.
- When A2 and R are 1, bus is driven with DISP_STATUS.
- DOUT is b_2 of DISP_STATUS.



Textbook Example 7.3

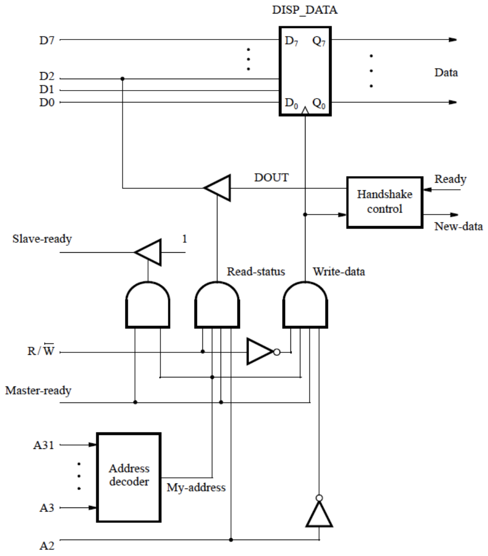
Design an output interface circuit for a synchronous bus. When data are written into the data register of this interface the interface sends a pulse with width of one clock cycle on a line called `New-data`. The pulse lets the output device connected to the interface know that new data are available.

Solution


Output register
D7
D0
D_7
Q_7
D_0
Q_0
Data
Enable
New-data
D
Q
$\overline{Q}$
Clock
Write-data
R/$\overline{W}$
A31
A2
Address decoder
My-address

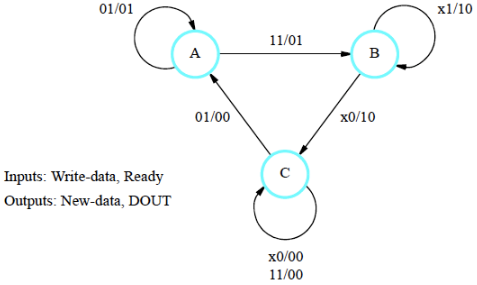
Textbook Example 7.4

Draw a state diagram for a finite-state-machine (FSM) that represents the behavior of the handshake control circuit.



Solution

- Start in A.
- Move to B on *Write-data*.
- Stay in B until $\overline{\textit{Ready}}$.
- Stay in C until $\overline{\textit{Write-data}}$ and *Ready*.



Serial Buses

Many modern I/O interconnection standards use serial data transmission. Serial buses has a number of advantages of parallel buses.

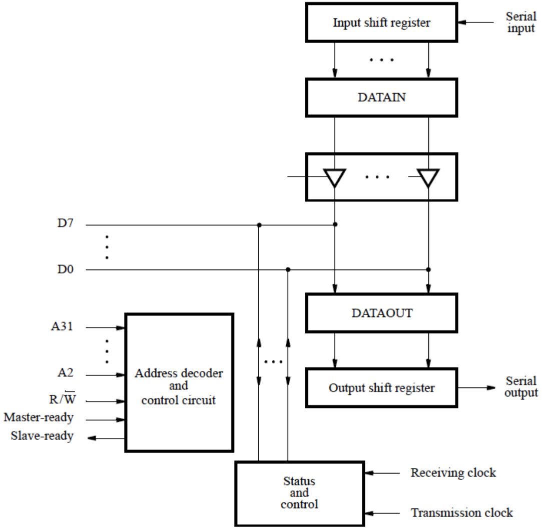
- Fewer connectors: more reliable.
- Fewer wires: better signal integrity, faster, lighter, cheaper.
- Smaller interface: lower complexity, power.

Data is transmitted one bit at a time.

- This requires a means for the receiver to recover timing information.
- One simple scheme for low-speed transmission is called start-stop, and is implemented in the Universal Asynchronous Receiver Transmitter (UART) protocol.

UART

Double buffering allows the interface to continue to receive new data while the processor handles the previous frame.

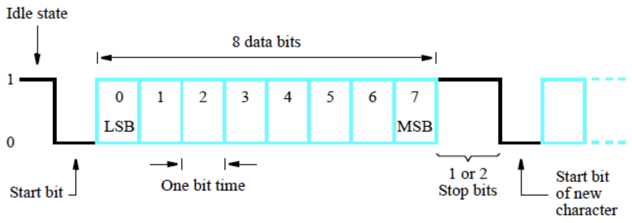


Start-Stop Transmission

In asynchronous serial communication, receiver and transmitter maintain their own clocks.

With UART, $f_R \sim 16 f_T$.

- On start bit's falling edge, reset modulo-16 counter.
- At a count of 8, check if the input is still 0; reset the counter.
- Sample each of the next eight bits at a count of 16.



I/O Interconnection Standards

Standards facilitate system integration by setting the constraints for interconnection. This decouples processor from I/O device development, making it possible for devices to work with a variety of processors, and vice versa.

In this context, everyone designs for conformance to the standard; compliant devices are then assumed to work interchangeably.

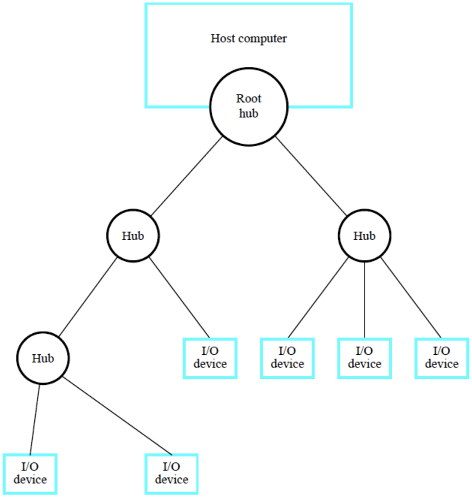
Universal Serial Bus (USB)

Perhaps the most commonly encountered I/O standard today is the Universal Serial Bus (USB): it is used for keyboards, mice, headphones, microphones, flash storage devices, printers, external disk drives, cameras, etc.

- USB 1: 12 Mbps; USB 2: 480 Mbps; USB 3: 5 Gbps
- Point-to-point connections using serial transmission and two twisted pairs (+5V, ground, two data wires)
- Low-speed transmission is single-ended: one data wire for 0, the other for 1
- High-speed transmission uses differential signaling
    - Data is encoded as the difference in voltage between the two lines
    - Noise is canceled, as it is common to both wires

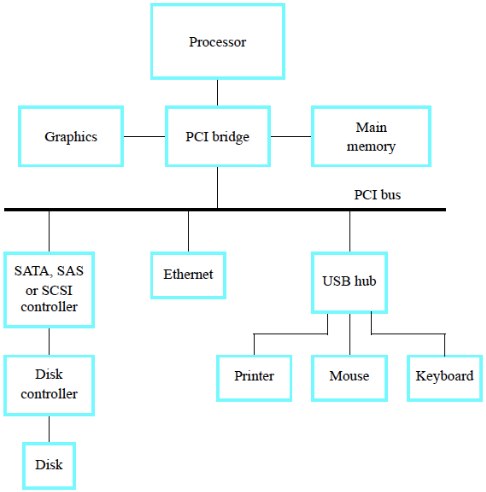
Universal Serial Bus (USB)

- USB can connect many devices using simple P2P links and hubs
- Plug-and-play: system detects new devices automatically
- USB hubs poll devices to initiate exchanges (avoiding issues with simultaneous communication)



Peripheral Component Interconnect (PCI) Bus

- PCI is a processor-independent motherboard bus.
- Devices on the PCI bus appear in the address space of the processor.

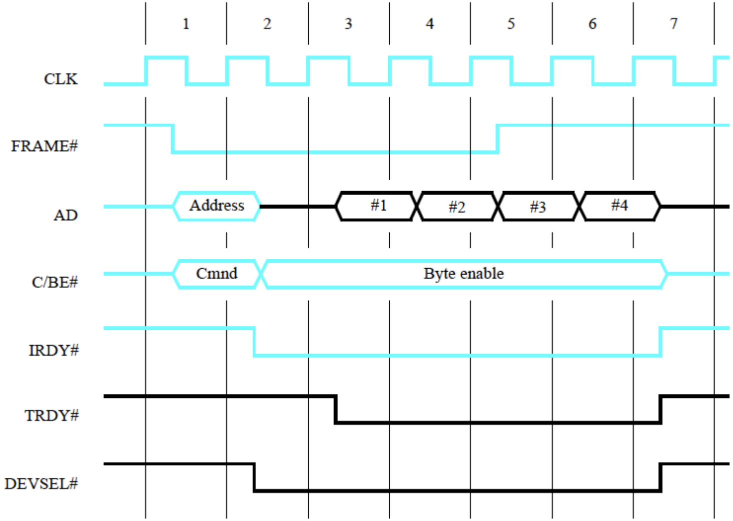


PCI Plug-and-Play

PCI pioneered plug-and-play, which was made possible by its initial connection protocol.

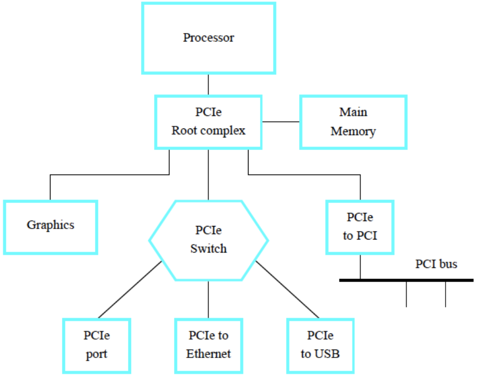
- There are up to 21 device connectors on the PCI bus
- Each PCI device has a small ROM that stores the device's characteristics
- The processor scans all connectors to determine where devices are connected
- It then assigns addresses to each device and reads the contents of each ROM
- With this information, the system selects the appropriate device driver, performs initialization, etc

PCI Read Timing



PCI Express (PCIe)

- P2P connections with one or more switches forming a tree.
- Root complex provides high-speed ports for memory and other devices.



PCIe Links

- The basic connection is called a lane
- A lane consists of two twisted-pair or optical lines for each direction of transmission
- The data rate is 2.5 Gbps in each direction
- A link may use up to 16 lanes
- The PCIe protocols are fully compatible with PCI, *e.g.*, using the same initial connection protocol

Conclusions

This set of lectures has introduce how computer systems receive input and send output. We've looked at:

- Software aspects of I/O: polling and interrupts
- Hardware aspects of I/O: buses, arbitration, synchronous and asynchronous communication protocols
- Standard interconnection networks

Next time: memory technology and efficient memory organization!