I/O Interfaces

Computer Organization

- From 2655 - computer = machine which:
- reads input data (+ instructions)
- executes instructions on data
- writes output data
- “Computer organization” refers to the structure, function and interconnection of a computer’s functional units.
- How does it differ from “computer architecture”?

Computer Architecture

- Computer architecture refers to the set of resources provided by hardware – anything hardware-provided that the low level programmer can/must interact with directly.
- is the apparent or virtual set of resources available to the low-level (assembly language) programmer

Computer Architecture

- Main functional units:
  - CPU → executes instructions
  - Main memory → stores data (+ instructions)
  - I/O interfaces → read from / write to outside world
    - without these the computer is useless
- … all interconnected by a bus (or busses), and synchronized by a clock
- Between which units can data flow?

Data Flow on Buses

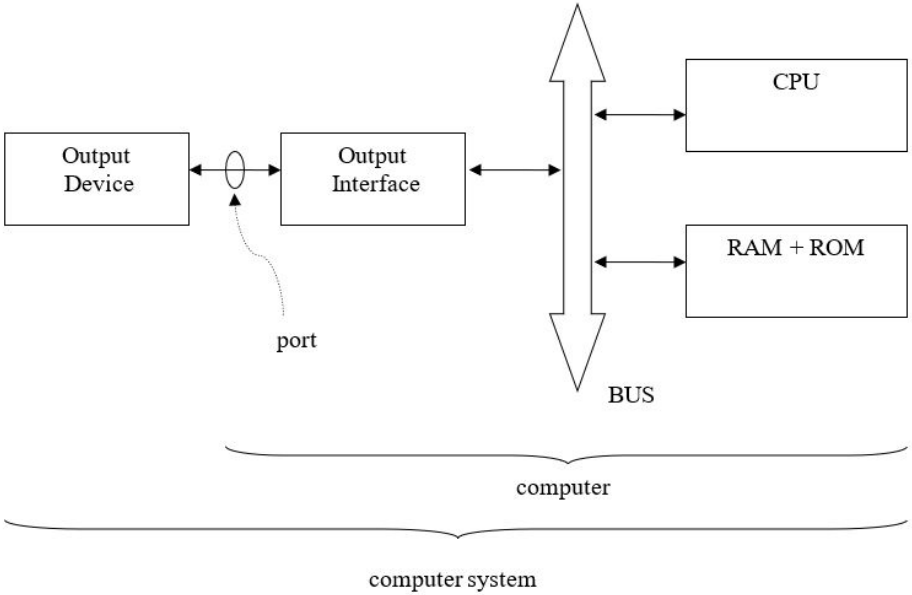
- Ideally all, but in reality the CPU controls the busses (both internal and external)
- Therefore:
  - CPU ←→ Memory
  - CPU ←→ I/O
  - Memory ←→ I/O will be discussed
  - I ←→ O why bother

I/O Interface vs. I/O Device

- Important: I/O interface ≠ I/O device
- I/O interface = a digital circuit which matches / connects an I/O device to a computer bus
    - (sometimes called an I/O module, I/O controller, interface circuit, ...)
- I/O device = a device which translates information between the computer and the environment
    - Can be electronic, electro-mechanical, etc.
- I/O devices are sometimes called peripherals

I/O Interface vs. I/O Device

- Notice that the device – interface connection is bi-directional.
- Does information ever flow into the computer from the output device?



Information Flow Direction

- YES – feedback from the device, ex. A printer indicating out of paper
    - But this is ONLY done through the interface
- Physically, the computer's units may all be on the same motherboard (or even on the same chip – a system on a chip - "SoC")
- I/O interfaces may or may not be on expansion cards
- I/O devices may or may not be external to the computer's case

Common I/O Devices

- What are some common input, output and input/output devices?
  - Input: KBD, Mouse, Joystick
  - Output: Printer, Monitor
  - Both: Modem, Touch Screen Monitor
  - Etc.
- Note: secondary storage devices (e.g. hard drives) are considered I/O devices, from our perspective.
- (all USB devices)

I/O Interfaces

- An I/O interface acts as a “buffer”
    - an electronic circuit which enables the connection of two different units
- Note: here, the term “buffer” means something different than a buffer in main memory
- Buffer in memory is: an array of characters

I/O Interfaces

- A buffer in relation to an I/O interface:
  - a person who shields another especially from annoying routine matters
  - a temporary storage unit (as in a computer); especially : one that accepts information at one rate and delivers it at another
- https://www.merriam-webster.com/dictionary/buffer

I/O Interface Buffering

- Why is buffering between I/O devices and the bus (CPU) necessary?
  - speed differences
  - data format differences (incl. serial vs. parallel)
  - electronic differences
  - etc.

I/O Interface Functions

- There are a number of functions that an I/O interface performs:
  - communication with CPU (etc.) via bus
  - communication with device
  - data buffering (this is the normal meaning of buffer!)
    - e.g. if data arrives from input device, hold it until CPU can read it
  - error detection
  - allow CPU to control I/O operation
  - allow CPU to monitor I/O status

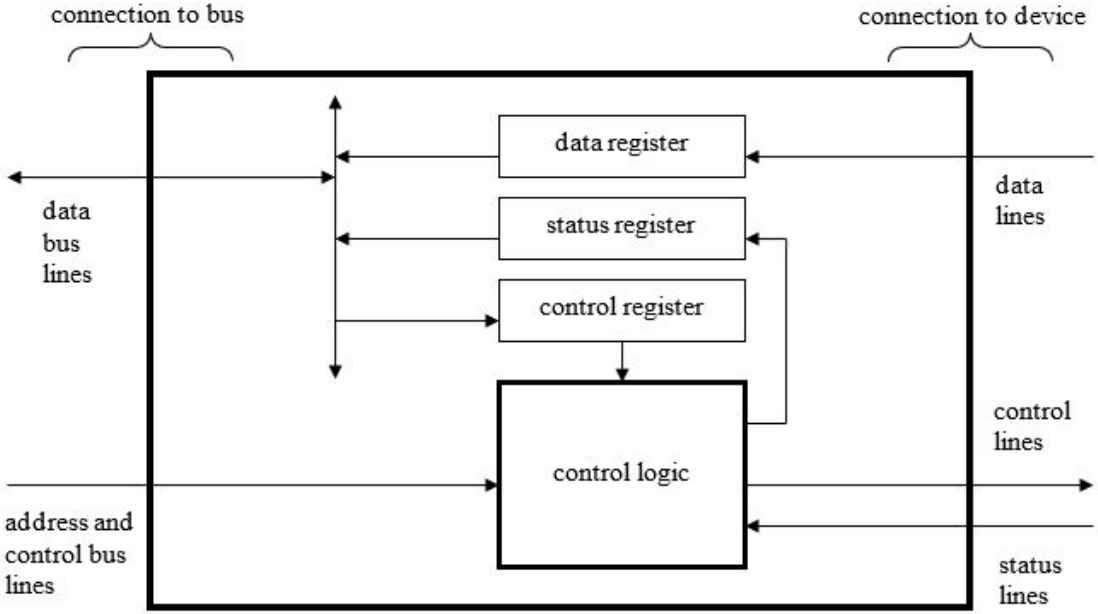
I/O Interface

- A typical I/O interface has various onboard registers:
  - input and/or output data register(s)
    - “buffers”
  - status register(s)
    - so CPU can monitor I/O status, e.g. is device “idle” or “busy”?
  - control registers(s)
    - so CPU can control I/O modes of operation, e.g. disable error detection

I/O Interface

- The CPU can access [read and write] these registers
- This is the method by which the CPU can interact with an I/O interface.
- Access may be limited, i.e. read only or write only.
- Reason:
  - What would the result of writing to the keyboard input data register?
  - Writing to the status register?

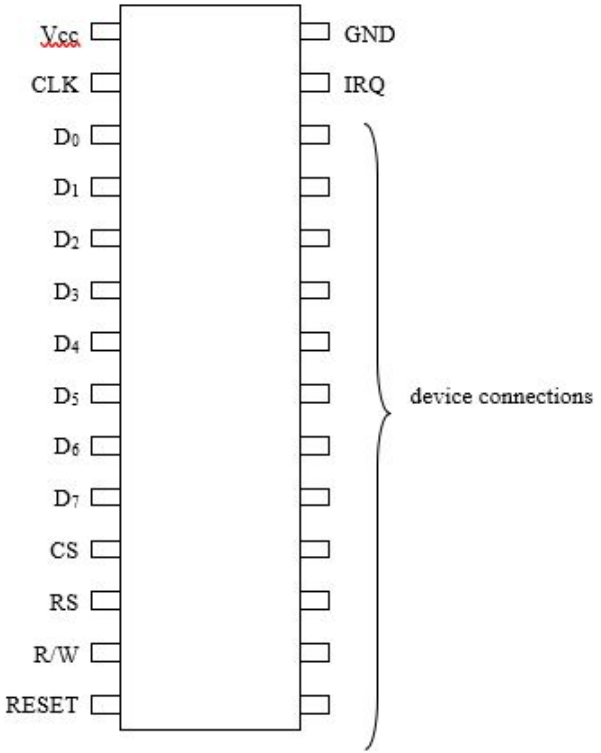
Generalized Input Interface Block Diagram



- An output interface would be similar
- NOTICE arrow direction(s) and which units communicate with each other

Generalized 8-Bit I/O Interface Pin Layout

- The d0-d7 are part of the external data bus that is connected to the CPU
    - These are **NOT** registers d0-d7!!
- The device connections are connected to a port that we see on the computer case!



Generalized 8-Bit I/O Interface Pin Layout

- Notice there are NO address lines!!
- Above, CS = “chip select” and RS = “register select”.
- As will be seen below, they are derived from the address bus.
- If the chip is not being selected i.e. CS = 0, it ignores the bus.
- If the chip is being selected, i.e. CS = 1,
  - Then RS + R/W determine which register is being accessed
    - The data is either read or written on D0-7

Generalized 8-Bit I/O Interface Pin Layout

- Notice for this “general” interface
  - How many visible registers are there? 3
  - How many RS pins? 1
  - Is there any problem / contradiction here?
    - Yes!!
  - With only 1 pin (bit) can only select between 2 items!!
  - This is an example of limited access!

Generalized 8-Bit I/O Interface Pin Layout

- The fact that there is only ONE RS pin indicates that there are only TWO visible registers from the CPU's perspective
  - the data register, which is read-only
  - the status register (read-only) and control register (write-only). Since these registers are only used in one mode they can be at the same address
- The number of visible internal registers will dictate the number of RS inputs the I/O interface will have

Memory Mapping

- How are I/O registers (“I/O ports”) addressed?
- Solution #1: assign them addresses from the CPU’s address space. They look like ordinary memory locations, but they’re not. This is called memory mapped I/O.
- E.g. 68000-based systems use memory mapped I/O.

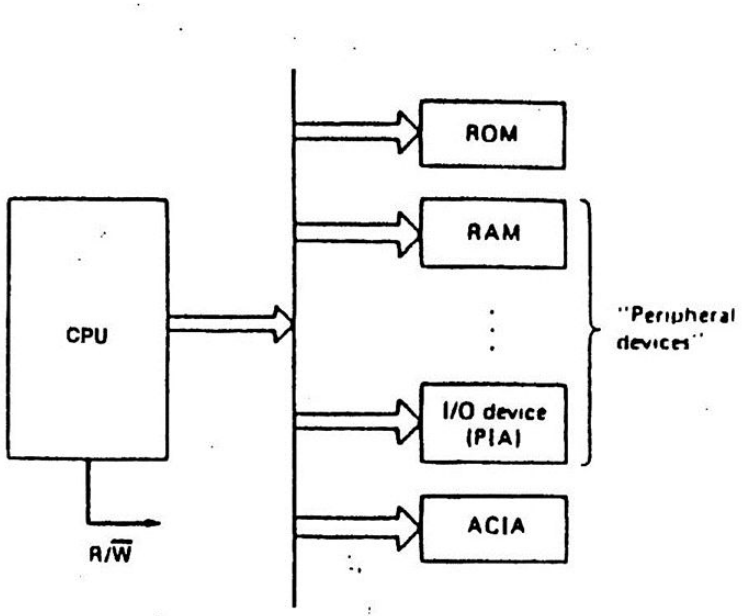
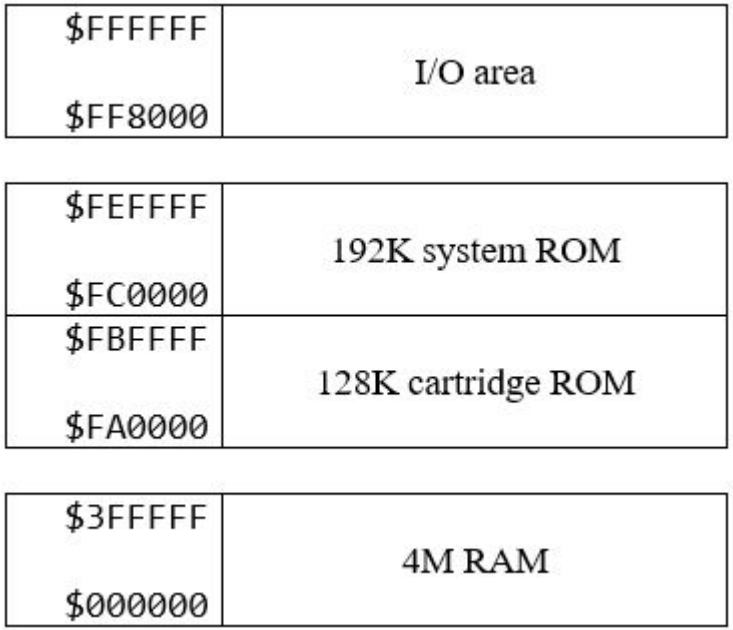


Figure 7.2. Memory-Mapped I/O.

Atari Memory Map

- The Atari ST memory map is shown to the right
- The RAM area is subdivided by TOS (e.g. for stack, frame buffer, etc.) as we have already seen.



Atari Memory Map

- The I/O area on the previous slide is further subdivided:

I/O interface:	Start Address for Registers:	
2 6850 ACIAs	$FFFC00	
68901 MFP	$FFFA00	
PSG	$FF8800	
DMA & FDC	$FF8600	
Video Registers	$FF8200	← SHIFTER & MMU
Data Configuration	$FF8000	← MMU

- These can be found in the reference\memory maps folder in the class directory and on D2L

Memory Mapping

- Solution #2: provide separate address spaces memory and I/O. This is called isolated I/O. Also called independent I/O or port mapped I/O
- E.g. Intel x86-based systems use isolated I/O. The instruction set includes `in` and `out` instructions.

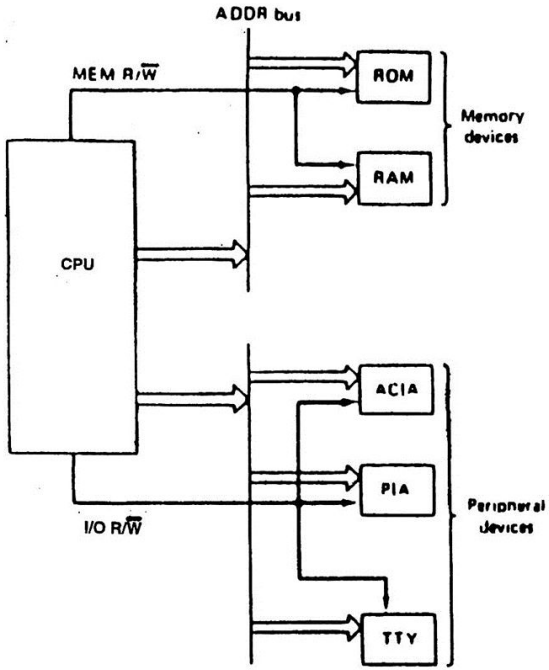


Figure 7.1. Independent I/O.

Memory Mapping

- Do there need to be two distinct busses, physically?
  - NO!
- There does need to be separate I/O and Memory R/W lines
  - Why?
- But the address and data lines can be common
  - i.e. multiplexed

Memory Mapping

- What are some pros and cons of memory mapped I/O?
  - pro: simpler CPU design (e.g. no specialized I/O instructions or logic)
  - pro: simpler for programmer too
  - con: some main memory address space is sacrificed
    - but is this really a problem?

Memory Mapping

- How does a particular I/O interface know that it is being addressed?
- One possibility: route all address bus lines to each I/O interface.
  - Each “listens” for its address (i.e. decodes the address bus signals) to determine if it is being addressed.
  - Why isn’t this normally done?

Memory Mapping

- Why isn't this normally done?
  - Each chip would need to access ALL of the address pins on the bus
    - Pins cost money so would make interface chips more expensive
    - different organizations/architectures have different sized address buses so need interface chip for each organization/architecture
    - the interface chip would have a fixed address – possible conflicts
    - would require extra logic on the interface chip
  - Want to have generic, cheap interface chips that can be used with any processor!

Memory Mapping

- Better: a platform-specific “system controller” decodes the address bus and asserts the chip select signal of whichever unit is being addressed
- E.g. on the Atari ST, the GLUE performs this duty (diagram shown later)
- Actually, a few of the least significant bits of the address bus may be passed on as register select signals

Programmed I/O

- How can a program perform I/O?
  - i.e. directly, without using a library function or a system call
- It can read/write the relevant I/O registers directly!
- Consider the following example:
- Pretend a program wants to write the string “hello, world!” to an output device
  - e.g. a printer

Programmed I/O

- Assume the output interface has the following registers:
  - status (8-bits) mapped at $FFFF00
  - data (8-bits) mapped at $FFFF02
- Notice: very similar addresses, i.e. close together in memory. In fact they only differ by one bit – *bit 1*
  - This bit would be connected to the Register Select pin
- Assume also that bit 3 of the status register is an “idle” flag: when set it means that the device is ready to accept the next character

Programmed I/O

- The following C code writes the string:

```
#define IDLE 0x08
volatile const char *out_status = 0xFFFF00;
volatile char *out_data = 0xFFFF02;

char str[] = "hello, world!";
int i;

for (i = 0; str[i] != '\0'; i++) {
        while (!(*out_status & IDLE))         /* BUSY WAIT */
                ;
        *out_data = str[i];
}
```

Programmed I/O

- The keyword `volatile` tells the compiler not to optimize accesses
- Notice that out_status is only accessed in the while loop test
- Since it isn't set anywhere else the compiler will tend change this code so that it is read ONCE before the loop
- By making it `volatile` the compiler will not make this change
- In this technique, the I/O is under direct control of the CPU. For this reason, it is referred to as programmed I/O

Programmed I/O

- This leads to an important question:
  - How is `out_status` changed so the loop knows the idle status?
- This is done using the technique called polling
  - polling is when the CPU repeatedly queries the I/O interface to determine when the next data value can be read or written
  - i.e. the status register has its data received repeatedly to get updated status information from the device (see the earlier block diagram)

Programmed I/O

- This leads to an important question:
  - Is polling efficient?
    - NO!
  - This code will spend 90% of the time in the busy wait.
    - Why?
    - Because the processor is far faster than the device. Remember this was one of the reasons for having I/O interfaces
    - Alternatives to programmed I/O include “interrupt-driven I/O” and “DMA”

Supervisor Mode

- Consider: what if dereferencing a wild pointer accidentally reprogrammed a register of, say, the hard drive controller?

```
char *ptr;
...
*ptr = 'a';     /* oops, forgot to initialize ptr! */
```

Supervisor Mode

- Consider: what if dereferencing a wild pointer accidentally reprogrammed a register of, say, the hard drive controller?

```
char *ptr;
...
*ptr = 'a';      /* oops, forgot to initialize ptr! */
```

- Many CPUs operate in one of 2 (or more) modes:
  1. user mode ← restricts use of certain instructions and addresses
  2. supervisor mode ← unrestricted access

Supervisor Mode

- The 68000 supports these two modes ...
  - normal programs operate in user mode
  - what is meant to operate in supervisor mode?
    - access to all memory and to all instructions
  - what error results when a user-mode program attempts to use a protected address?
    - A bus error

Supervisor Mode

- How about a privileged instruction?
  - move, andi, eori and ori where the status register is an operand
- SR contains a supervisor bit
  - Bit 13, but in order to access this bit the processor must be in supervisor mode!
- user mode uses A7 (USP) as its stack pointer
- supervisor mode uses A7’ (SSP)

Supervisor Mode

- Amazingly, TOS provides the Super system call for switching into supervisor mode!
  - It takes the value to load into the SSP (0 leaves it as-is).
  - It returns the original value of SSP
    - which should be saved so it can be restored when returning to user mode

Supervisor Mode - Example

- E.g. on the Atari ST, omitting Super will result in a bus error:

```
volatile const char *kybd_status = 0xFFFC02;
long old_ssp;
old_ssp = Super(0);
if (*kybd_status == ...)
    ...
Super(old_ssp);
```

- The minimum amount of work possible should be done in supervisor mode
- **Note:** All of the above can also be done in assembly!

Supervisor Mode

- How does the CPU (68000) know which addresses are protected?
- It doesn't – this must be handled by an external item
- On the Atari this is done by an Atari custom chip the GLUE using the address and the CPU's state (mode) makes this determination

Supervisor Mode

- The 68000 outputs its state on the three “function code” pins, FC0-2
    - This includes an indication of whether the CPU is in user or supervisor mode
- The FC values can be found on the reference card in the column beside the Status Register