COSC 243

Input / Output

Introduction

- This Lecture
  - Input/Output
  - Source: Chapter 7 (10th edition)

- Next Lecture (until end of semester)
  - Zhiyi Huang on Operating Systems

Memory

- RAM
  - ***R***andom ***A***ccess ***M***emory
  - Read / write memory
  - Loses its contents on power-off (Volatile)

- ROM
  - ***R***ead ***O***nly ***M***emory
  - Random access memory
  - Keeps its contents on power-off (non-volatile)
  - PROM / EPROM / EEPROM can be used as substitutes

Power-On / Reset

- What happens when you reset the CPU?
  - On the 6502 8-bit CPU
    - Load PC with the value of memory location FFFC and FFFD
      - Called the “reset vector” or “reset interrupt vector”
    - Branch to that address
  - On the ARM 32-bit CPU
    - Load PC with 0000
    - Branch to that address
- On the 6502
  - What number is stored at FFFC / FFFD?
  - How do you get it there?

Recall...

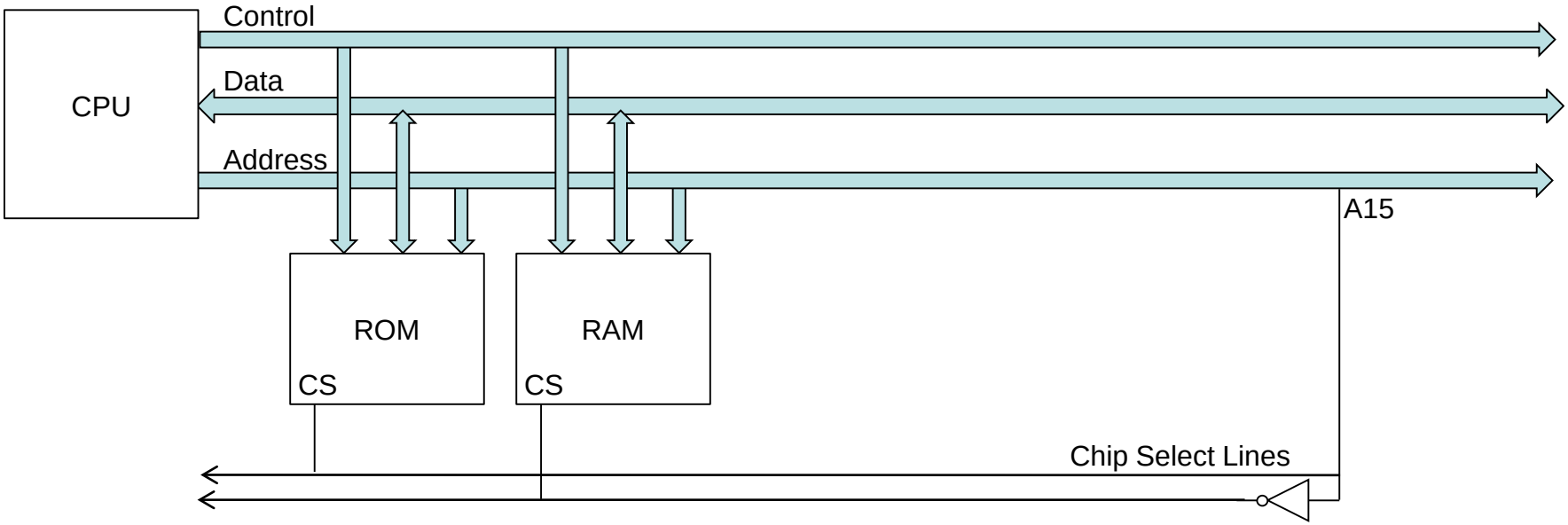
- CPU puts the contents of PC onto the address bus
- Then reads the value on the data bus

- For the 6502 we want to have the value of the reset vector (and the location it branches to) already there at power-on

- We make sure those memory locations are in ROM
- For this we need to split the address space into two

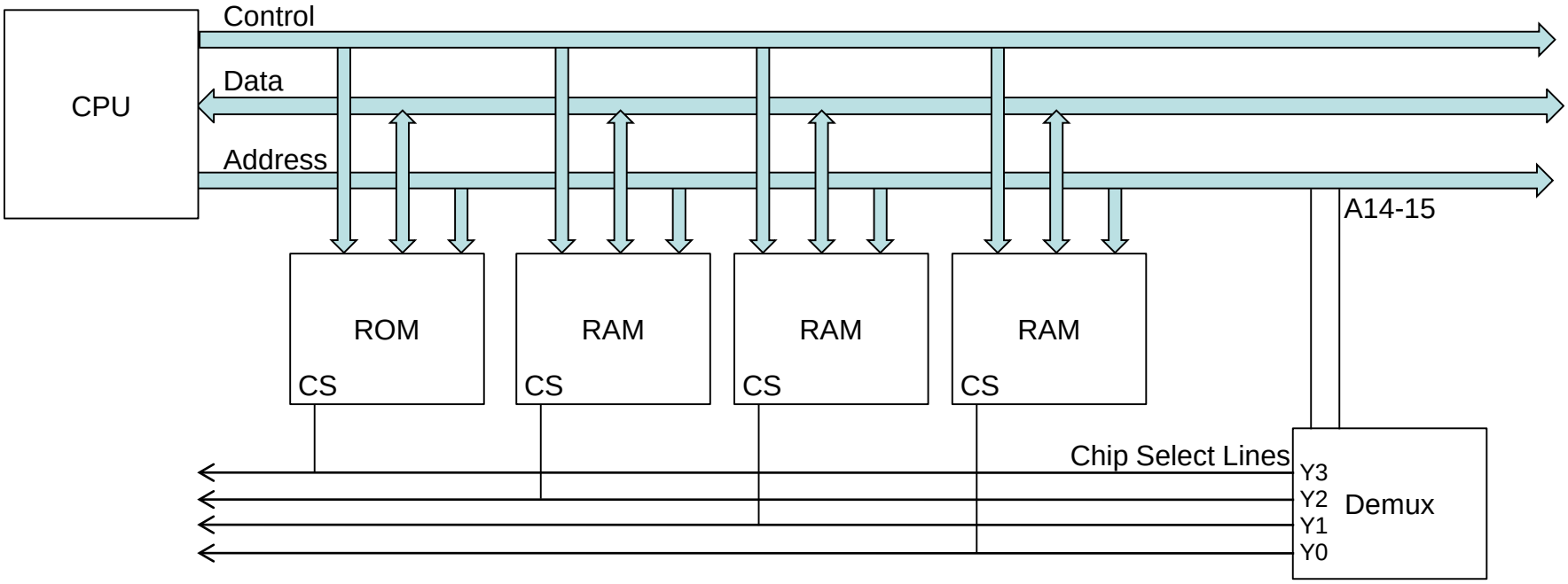
Partition the Address Space

- E.g. Divide the “address space” into two
  - One half is ROM, remainder is RAM



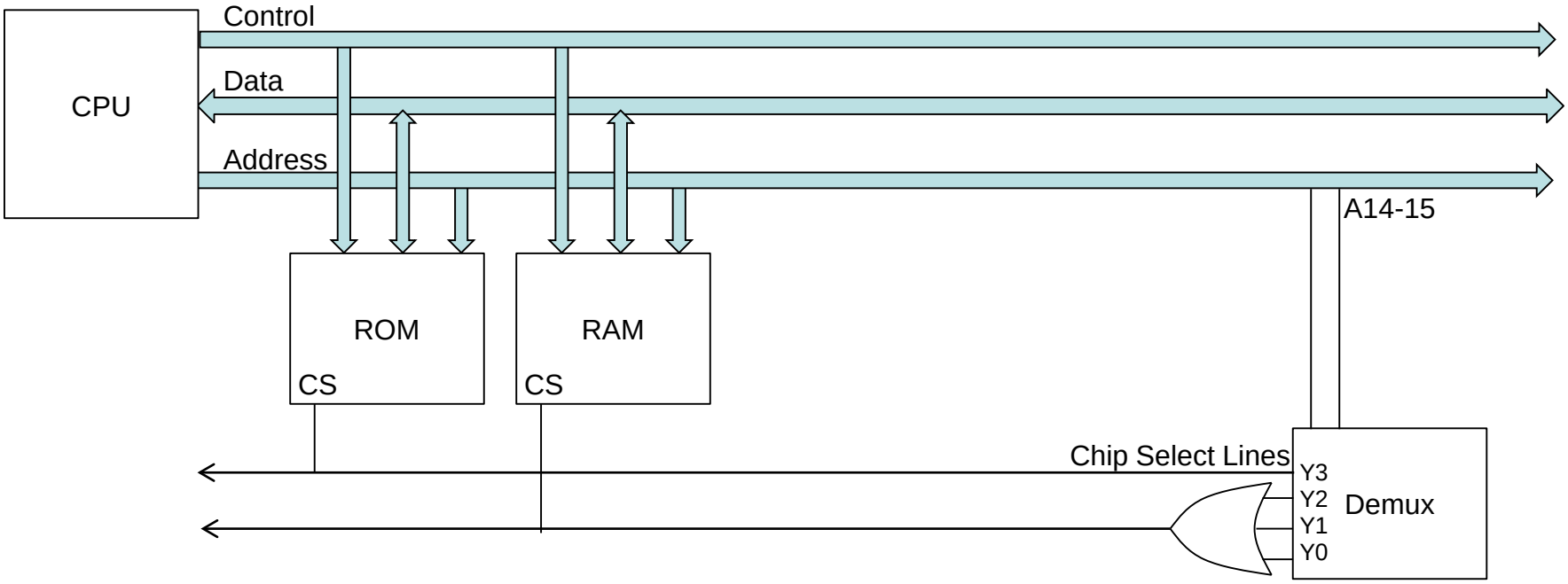
Partition the Address Space

- E.g. Divide the “address space” into four
  - One quarter is ROM, remainder is RAM



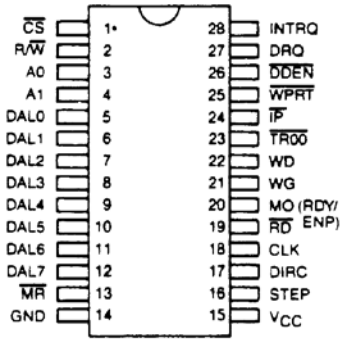
Partition the Address Space

- E.g. Divide the “address space” into four
  - One quarter is ROM, remainder is RAM



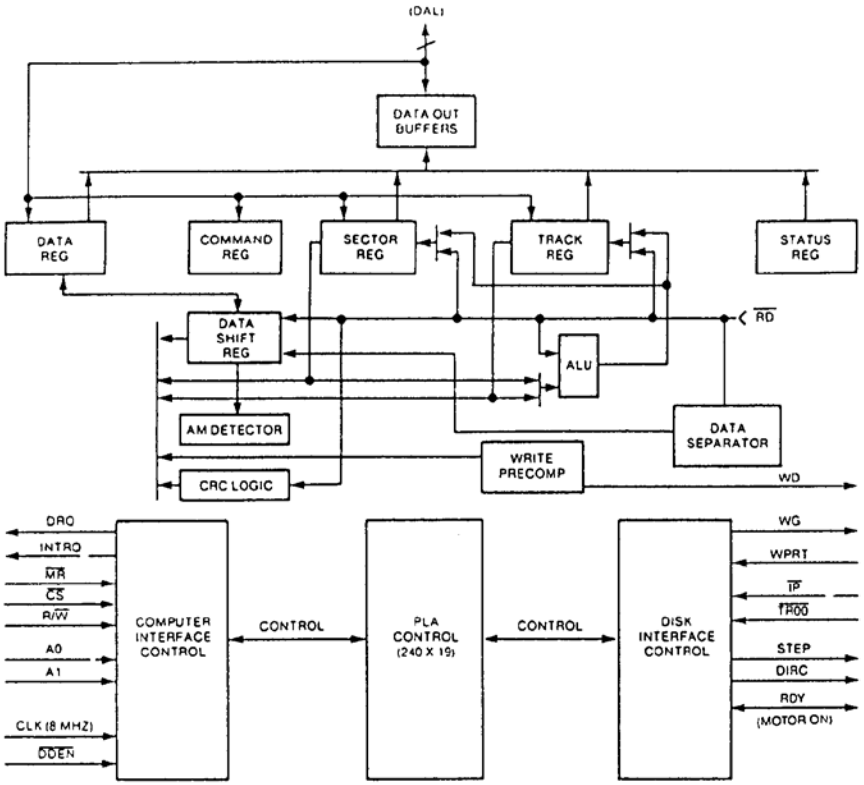
Power-On

- Create a ROM with interrupt vector set to branch to somewhere also in the ROM
  - That program does a Power On Self Test (POST)
  - Then starts the Operating System
    - Seek the disk head to track 0 sector 0 (the boot sector)
    - Load track 0 sector 0 into memory
    - Run the program just loaded from disk (the boot loader)
  - Also (often) contains the Basic Input Output System (BIOS)
    - Read / write to disk
    - Read keyboard
    - Write to screen
    - All of which are needed in the boot process.
- The remainder of the address space is RAM

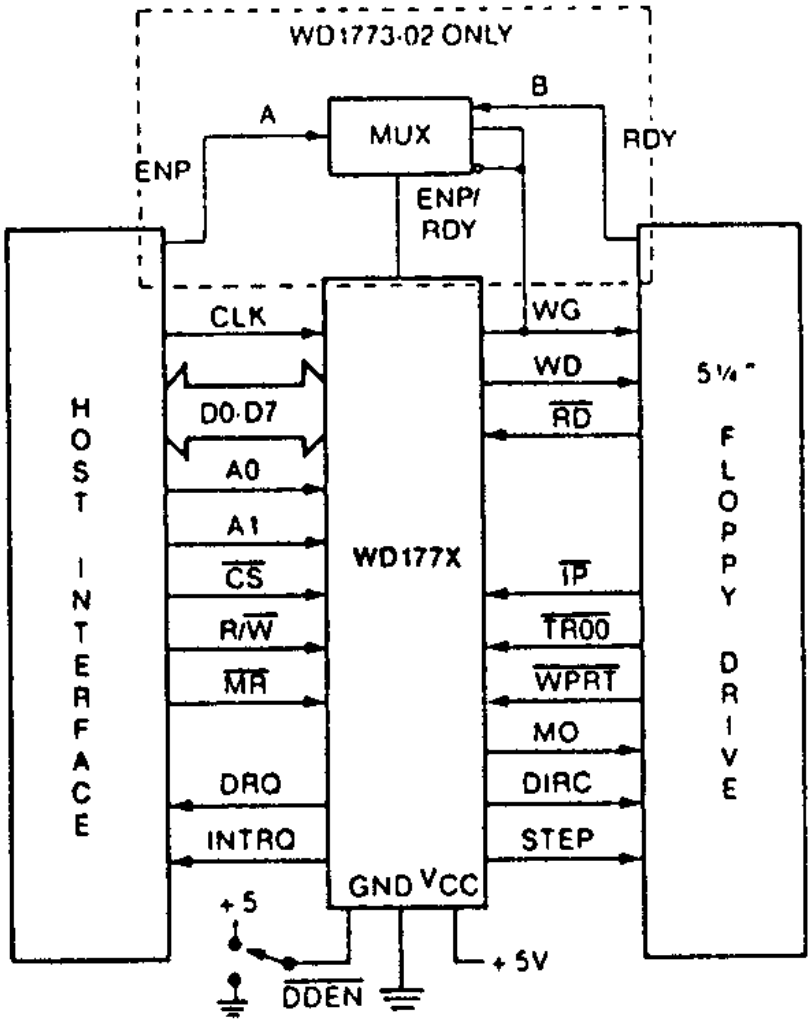


Peripheral Devices

- How do we get the contents of the disk into memory?
  - WD1771 (Floppy Disk Drive Formatter/Controller)



WD1771


WD1773-02 ONLY
A
B
MUX
RDY
ENP
ENP/
RDY
CLK
WG
WD
RD
D0-D7
A0
A1
CS
R/W
MR
WD177X
IP
TR00
WPRT
MO
DIRC
STEP
DRQ
INTRQ
GND
VCC
+5
+5V
DDEN
HOST INTERFACE
5¼" FLOPPY DRIVE

WD1771 Details

A1-A0	Read	Write
00	Status Register	Command Register
01	Track Register	Track Register
10	Sector Register	Sector Register
11	Data Register	Data Register

WD1771 Addresses

Command	Meaning
08	Restore (seek Track 0)
5A	Step In
7A	Step Out
88	Read Sector
A8	Write Sector

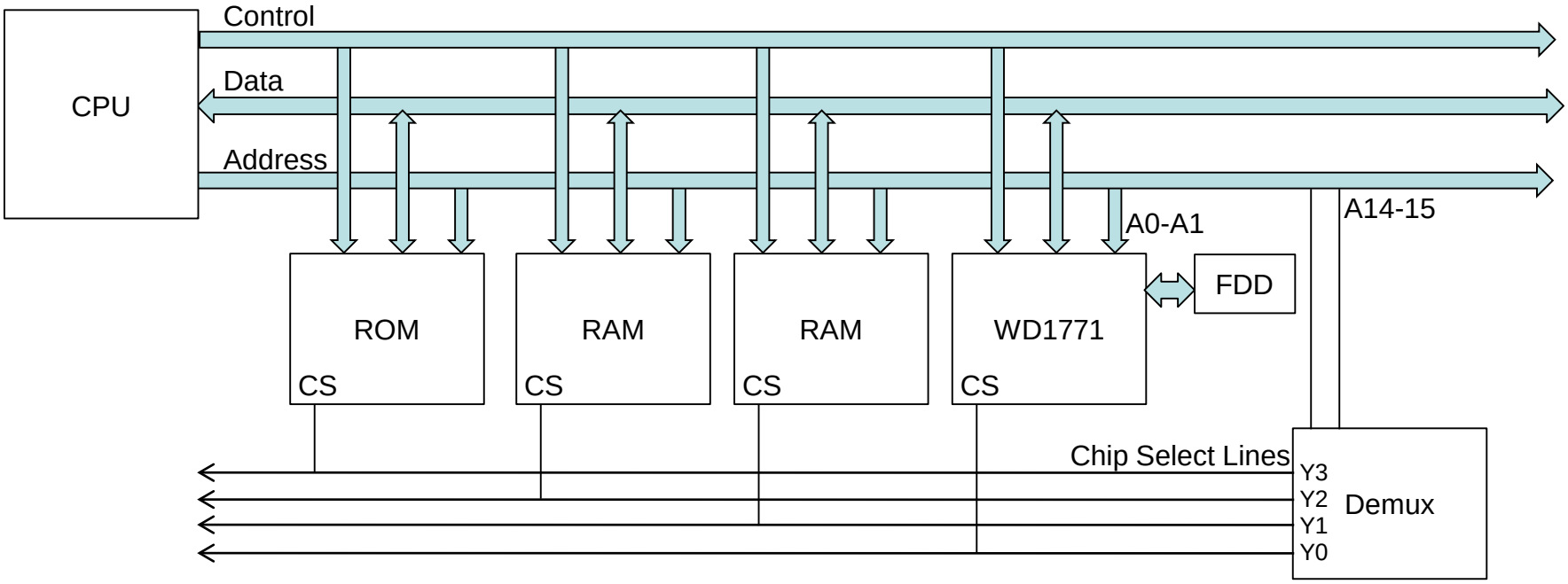
WD1771 Commands (abridged)

Programming the WD1771

- To Seek to Track 0
  - Load the 08 into a CPU register
  - Copy the CPU register into the WD1771 command register

Memory Mapped I/O

- Map the WD1771 controller into the address space
  - This is called ***Memory Mapped I/O***



Programming the WD1771

- To Seek to Track 0
  - Load the 08 into a CPU register
  - Store the CPU register in the computer's memory
    - At the location that the WD1771 command register is located
  - The WD1771 gets the command
    - Performs the seek operation

Programming the WD1771

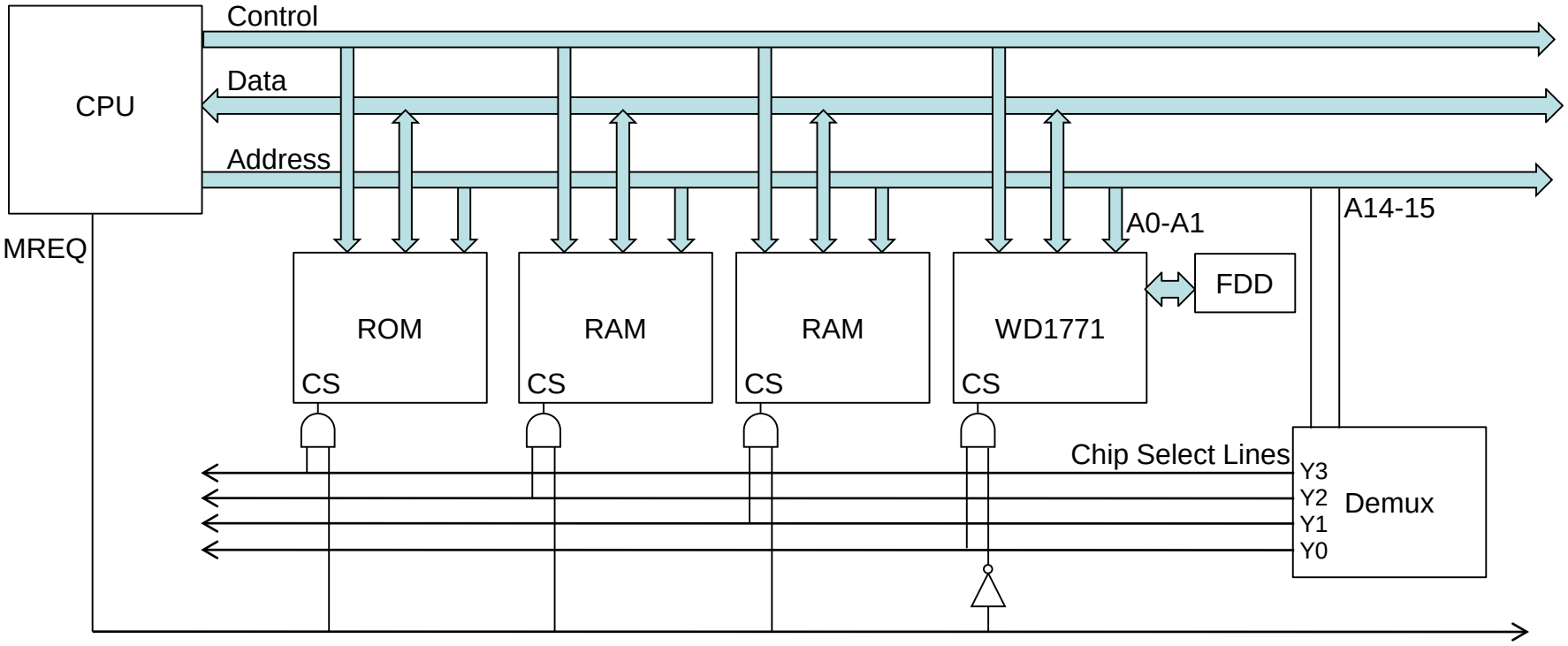
- Read a sector
  - Load track number into CPU register
  - Store in the WD1771 track register
  - Load the sector number in the CPU register
  - Store in the WD1771 sector register
  - Load 88 into the CPU register
  - Store in the WD1771 command register
  - Read (byte at a time) from the WD1771 data register

Isolated I/O (Port Mapped I/O)

- A disadvantage with ***Memory Mapped I/O*** is that the devices take some of the memory space
- To get around this some CPUs (Z80, 8086, etc) use a separate control line to control I/O
  - Special `IN` and `OUT` instructions in the CPU
  - Normally write contents of register to I/O “port”
- A disadvantage of ***Isolated I/O*** is that it requires special hardware and instructions in the CPU

Isolated I/O (Port Mapped I/O)

- Map the WD1771 controller into the port space



Reading from a Device

- The CPU is much faster than the Floppy Disk Drive
- After we issue a read sector command, how do we know when there is data ready to read?
- One approach is called ***Polling***
  - The WD1771 has a ***status register*** that signals when data is ready

Bit	Name	Meaning
7	Not Ready	Drive is not ready
6	Write Protect	Set when a write fails because the disk is write-protected
5	Record Type	Valid or “deleted” sector on the disk
4	Not Found	Selected Track/Sector/Side were not found
3	CRC Error	Error on disk
2	Lost Data	The CPU didn’t read the byte before the next arrived
1	Data Request	Data register is full (read) or empty (write)
0	Busy	Busy performing a command

Polled I/O

- In the ***polled I/O*** approach (reading from WD1771)
  - In a loop:
    - Read the status register to see data is ready
    - If Data Request bit is set
      - Read a byte from the Data Register
    - Read the status register to see if finished
    - If Busy bit is set
      - Finished the (read) command
    - Else
      - Read more bytes

Polled I/O

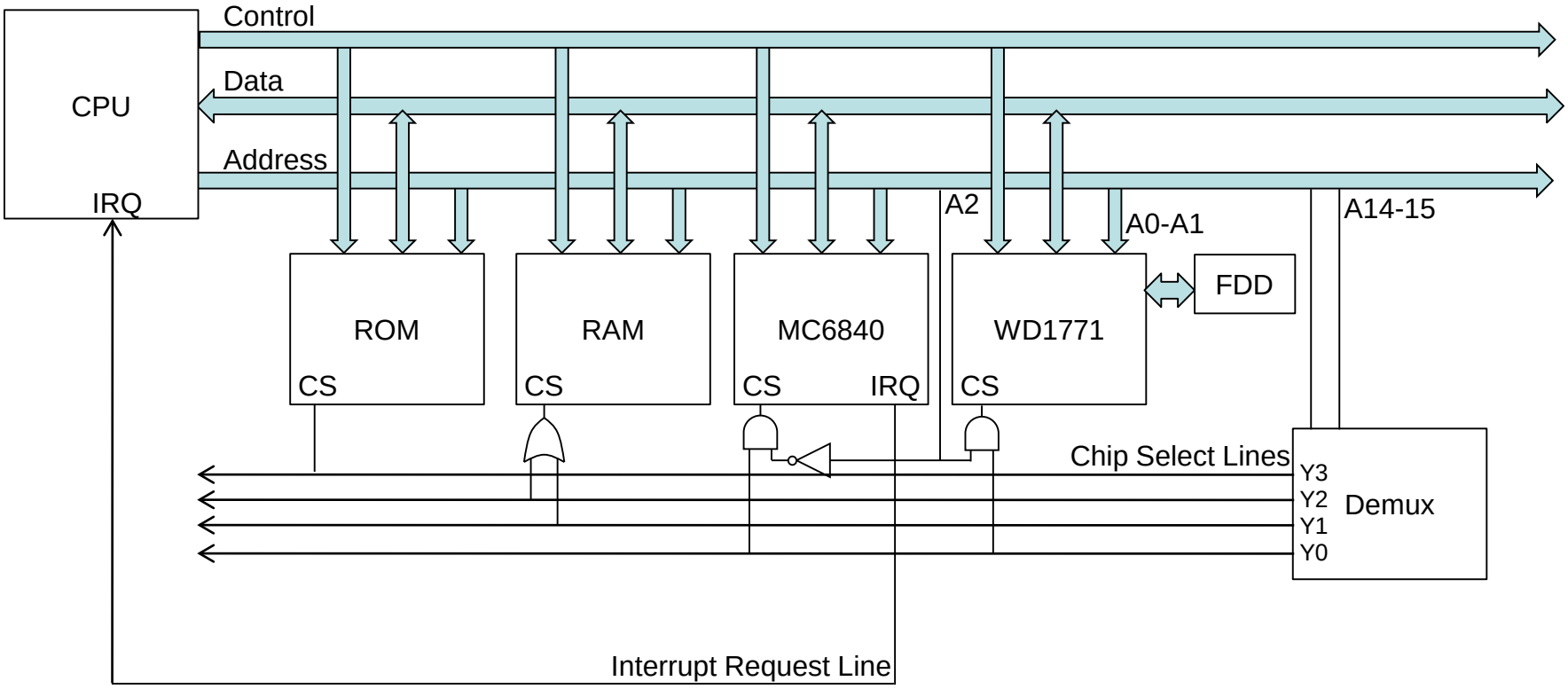
- With ***polled I/O*** we poll over and over again to see if there is something we should do
  - If you have multiple devices attached then you poll each one in turn to see if there is anything that needs doing
- If the device is many times slower than the CPU then you waste CPU cycles checking the status
- There is no way to know when work must be done
  - So CPU time is spent checking when there's nothing to do

Interrupts

- When reading from disk the CPU initiates the request
- When lying in bed in the morning the alarm clock interrupts your sleep
- When the CPU is busy and an alarm goes off it is interrupted
- When reading from a serial port the sender initiates
  - Perhaps this should be ***interrupt*** driven too?

Interrupts

- The MC6840 clock can be programmed to ***interrupt*** the CPU. Map it into the address space



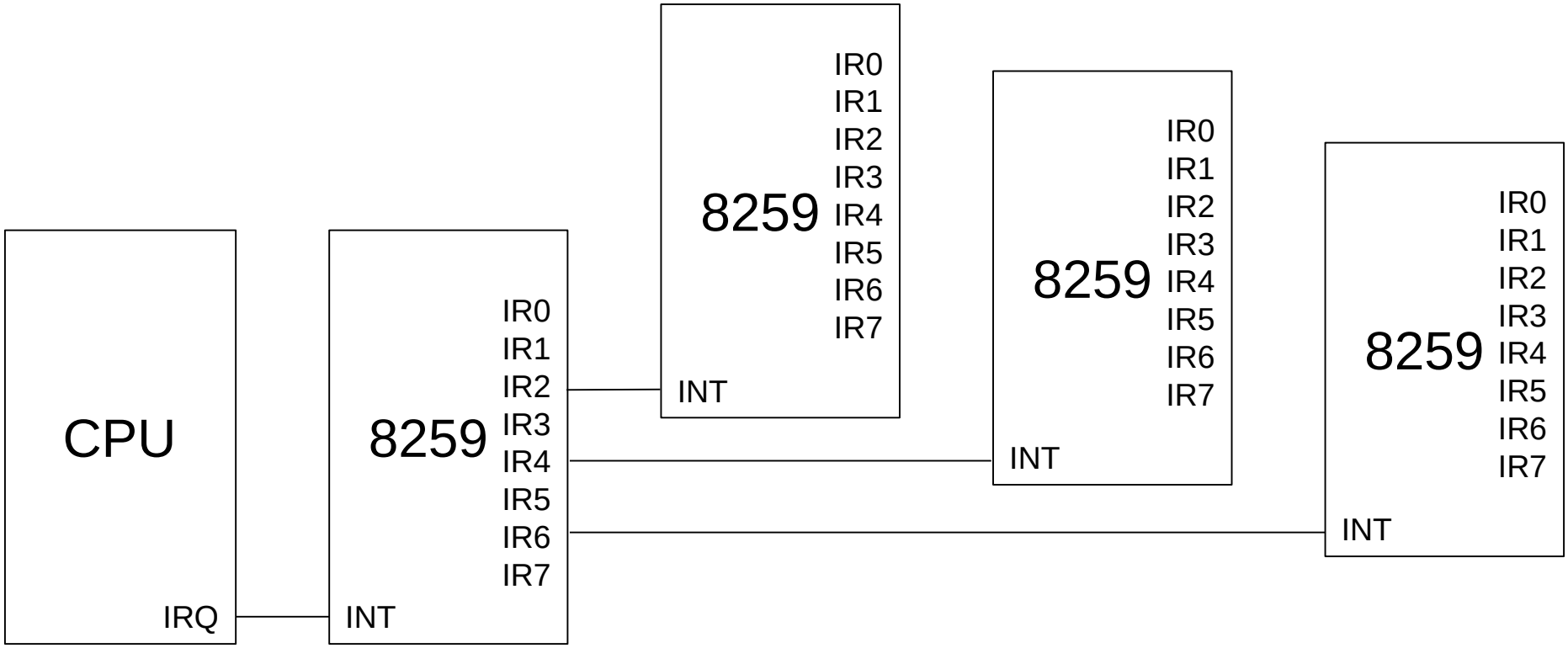
Interrupts

- The CPU is running a program
- At the end of each instruction it checks the IRQ line
- If it is signaled then it
  - Stops the program
  - Saves the state of the CPU (on the stack)
  - Loads a value from ROM
    - IRQ vector, FFFE and FFFF on 6502
  - Jumps to a program (typically) in ROM
  - Run that program
    - ***Interrupt handler / interrupt service routine***
    - The program finishes with a “return from interrupt” instruction
  - Restores the state of the CPU
  - Goes back to what it was doing

Multiple Interrupts

- More than one device might generate interrupts
- How do we determine the requesting device?
  - Multiple interrupt lines (NMI / IRQ in 6502)
    - Each has a different interrupt vector
      - NMI is at FFFA / FFFB on 6502
      - IRQ is at FFFE / FFFF on 6502
- How many interrupt vectors does the CPU need?
  - Only one!
    - If you can connect it to another device that can multiplex interrupts

Interrupt Controller



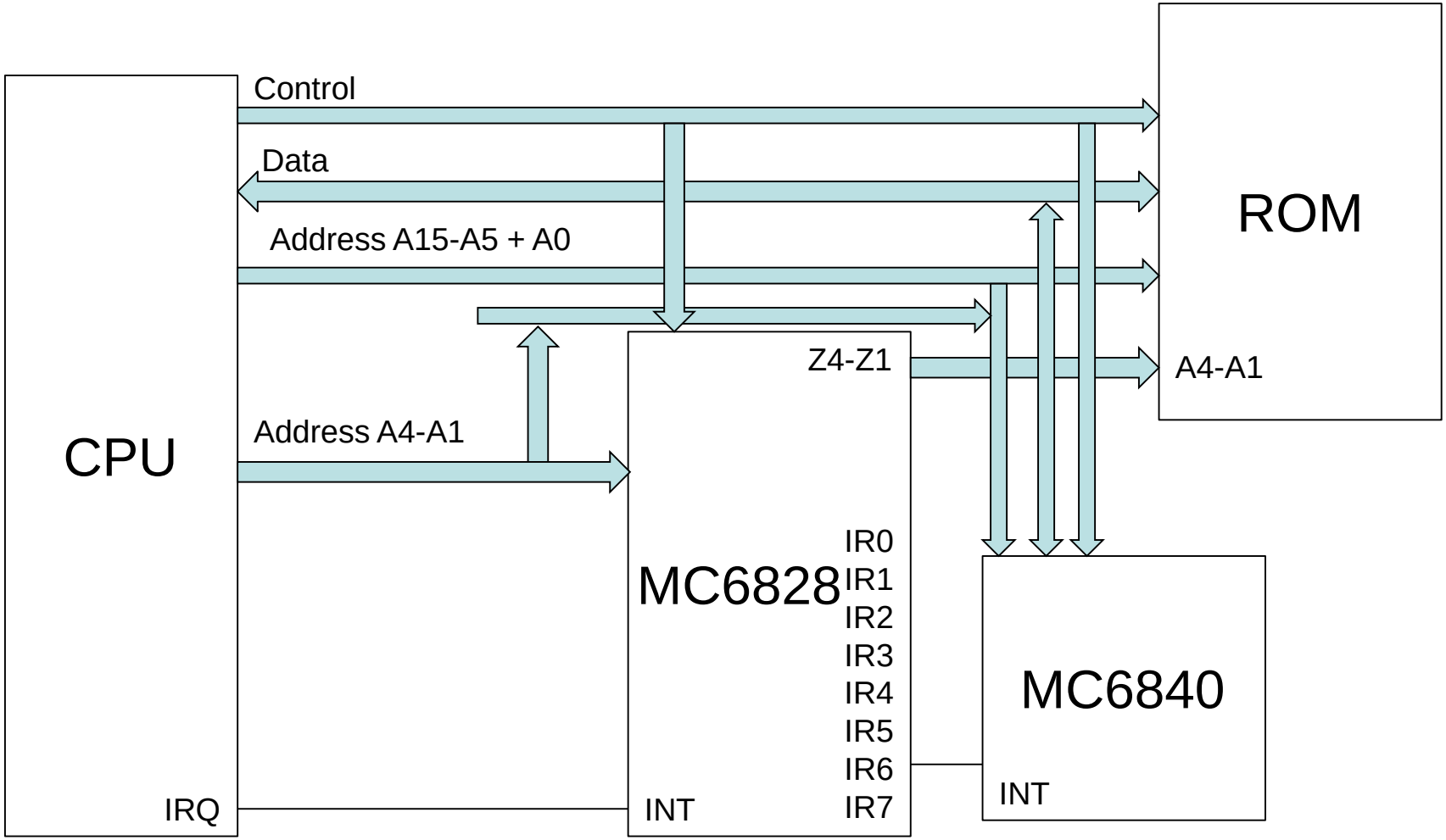
Multiple Interrupts

- Software Poll
  - When a device generates an interrupt, it sets a bit in its status register to indicate it wants service. The CPU polls the devices to see which one(s) need servicing
- Daisy chain
  - CPU sends an interrupt acknowledgement which propagates through each device until it reaches the signalling device
  - The interrupting I/O device places the address of its interrupt handler onto the data bus

Multiple Interrupts

- Interrupt Vector Expansion
  - The ***Interrupt Controller*** adds new interrupt vectors by sitting on the address bus between the CPU and the ROM
  - It receives the Interrupt, listens for the CPU accessing the address bus, and changes the address to a new address before it is seen by the address bus (connected to the ROM)
  - The ROM returns the appropriate data (an interrupt vector) to the CPU which then branches to the interrupt service routine at that (new) address
  - On the MC6809 CPU the MC6828 adds 8 interrupt vectors between memory locations FFE8 and FFF6 (each is 2 bytes)

Interrupt Controller



Interrupts

- The CPU can disable (some) interrupts
  - NMI = Non Maskable Interrupt on 6502 is non maskable

- Interrupts can have levels of priority
  - On MC6809 there are three levels:
    - IRQ: Interrupt Request, lowest priority
    - FIRQ: Fast IRQ, middle priority, can interrupt an IRQ
    - NMI: Non Maskable Interrupt, cannot be prevented

  - On the MC6828 the interrupts form a strict ordering
    - INT0 < INT1 < INT2 < INT3 < INT4 < INT5 < INT6 < INT7

Direct Memory Access (DMA)

- When the CPU reads from a device and writes to memory the data must pass up the data bus to the CPU then down the data bus to the memory. The program that does this takes CPU resources that might be used for other purposes.
- This can be overcome with special circuitry that (when the CPU isn't using the bus) reads from the device and writes to memory.
- That special circuitry is called the ***Direct Memory Access Controller***

DMA

- The CPU tells the DMA controller
  - Where to read from
  - Where to write to
  - How many bytes
- And then tells the DMA controller to do the transfer
- The DMA module mimics the CPU's bus accesses
  - Reads from the read address
  - Writes to the write address
  - Increments the write address
  - Decrements the byte count
- CPU is only involved at the beginning and end

Direct Memory Access (cont)

- Transfer is between device and memory without processor intervention
- Data is transferred via the system buses
  - DMA Controller uses the bus only when the CPU is not
  - This is done by the CPU signaling its non-use via the control bus
- Can either read from or write to devices
  - Read from incrementing location and write to fixed location
  - Some can increment the read and write address
    - Equivalent to a ***block move***
- Some devices talk to the DMA controller directly and transfer data when they are ready

Fast I/O

- Typically:
  - Device is memory mapped into CPU address space
  - Connected to the CPU interrupt line
    - Sometimes through an interrupt controller
  - Device causes an interrupt
  - CPU tells the DMA controller to do a transfer
  - CPU goes back to what it was doing
  - DMA controller does the transfer
  - DMA controller signals the CPU when finished (an interrupt)
  - CPU processes the data


"That's all Folks!"