EECE 321: Computer Organization

Mohammad M. Mansour
Dept. of Electrical and Compute Engineering
American University of Beirut

Lecture 39: IO Basics

I/O Commands

- I/O devices are managed by I/O controller hardware
  - Transfers data to/from device
  - Synchronizes operations with software
- Command registers
  - Cause device to do something
- Status registers
  - Indicate what the device is doing and occurrence of errors
- Data registers
  - Write: transfer data to a device
  - Read: transfer data from a device

I/O Register Mapping

- Memory mapped I/O
  - Registers are addressed in same space as memory
  - Address decoder distinguishes between them
  - OS uses address translation mechanism to make them only accessible to kernel

- I/O instructions
  - Separate instructions to access I/O registers
  - Can only be executed in kernel mode
  - Example: x86

Polling

- Periodically check I/O status register
  - If device ready, do operation
  - If error, take action
- Common in small or low-performance real-time embedded systems
  - Predictable timing
  - Low hardware cost
- In other systems, wastes CPU time

Interrupts

- When a device is ready or error occurs
  - Controller interrupts CPU
- Interrupt is like an exception
  - But not synchronized to instruction execution
  - Can invoke handler between instructions
  - Cause information often identifies the interrupting device
- Priority interrupts
  - Devices needing more urgent attention get higher priority
  - Can interrupt handler for a lower priority interrupt

I/O Data Transfer

- Polling and interrupt-driven I/O
  - CPU transfers data between memory and I/O data registers
  - Time consuming for high-speed devices

- Direct memory access (DMA)
  - OS provides starting address in memory
  - I/O controller transfers to/from memory autonomously
  - Controller interrupts on completion or error

DMA/Cache Interaction

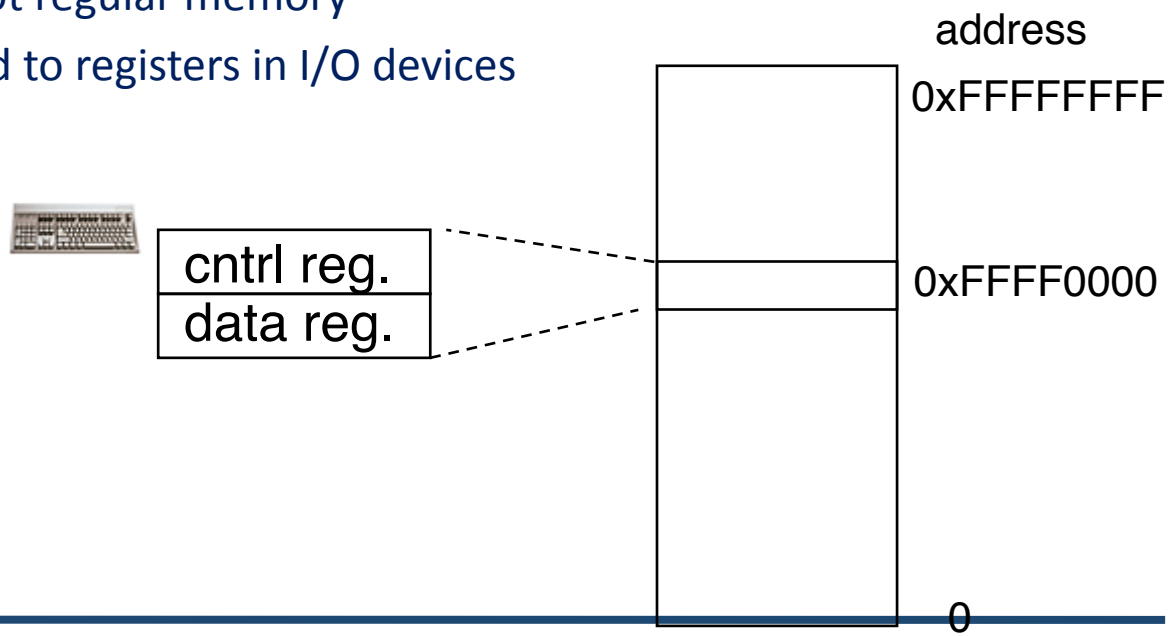
- If DMA writes to a memory block that is cached
  - Cached copy becomes stale
- If write-back cache has dirty block, and DMA reads memory block
  - Reads stale data
- Need to ensure cache coherence
  - Flush blocks from cache if they will be used for DMA
  - Or use non-cacheable memory locations for I/O

DMA/VM Interaction

- OS uses virtual addresses for memory
  - DMA blocks may not be contiguous in physical memory
- Should DMA use virtual addresses?
  - Would require controller to do translation
- If DMA uses physical addresses
  - May need to break transfers into page-sized chunks
  - Or chain multiple transfers
  - Or allocate contiguous physical pages for DMA

Instruction Set Architecture for I/O

- What must the processor do for I/O?
  - Input: reads a sequence of bytes
  - Output: writes a sequence of bytes
- Some processors have special input and output instructions
- Alternative model (used by MIPS):
  - Use loads for input, stores for output
  - Called "Memory Mapped Input/Output"
  - A portion of the address space is dedicated to communication paths to Input or Output devices (no memory there)
- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices



Processor-I/O Speed Mismatch

- 1GHz microprocessor can execute 1 billion load or store instructions per second, or 4,000,000 KB/s data rate
  - I/O devices data rates range from 0.01 KB/s to 1,000,000 KB/s

- Input: device may not be ready to send data as fast as the processor loads it
  - Also, might be waiting for human to act

- Output: device may not be ready to accept data as fast as processor stores it

- What to do?

Processor Checks Status before Acting: Polling

- Path to device generally has 2 registers:
  - Control Register, says it's OK to read/write (I/O ready) [think of a flagman on a road]
  - Data Register, contains data

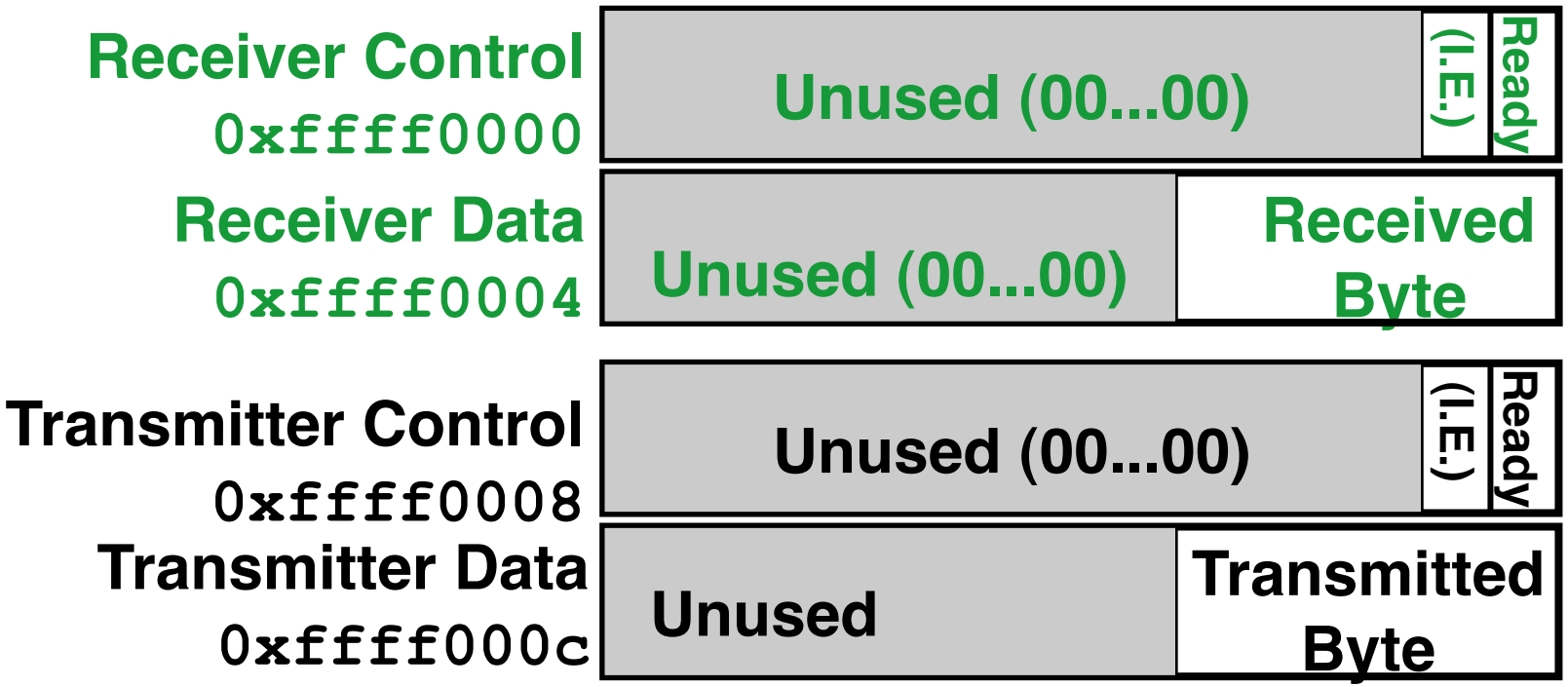
- Processor reads from Control Register in loop, waiting for device to set Ready bit in Control reg (0 ⇒ 1) to say its OK

- This process is called polling.

- Processor then loads from (input) or writes to (output) data register
  - Load from or Store into Data Register resets Ready bit (1 ⇒ 0) of Control Register

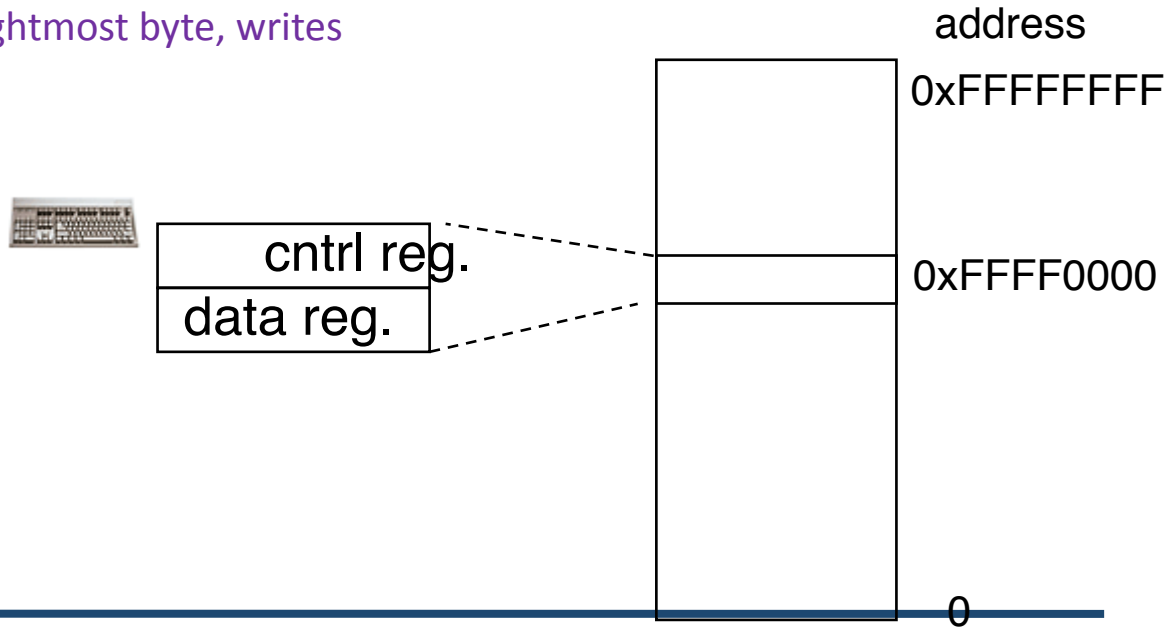
SPIM I/O Simulation

- SPIM simulates 1 I/O device: memory-mapped terminal (keyboard + display)
  - Read from keyboard (receiver); 2 device registers
  - Writes to terminal (transmitter); 2 device registers



SPIM I/O

- Control register rightmost bit (0): Ready
  - Receiver: (keyboard) Ready==1 means character in Data Register not yet been read;
    1 ⇒ 0 when data is read from Data Reg
  - Transmitter: (terminal) Ready==1 means transmitter is ready to accept a new character;
    0 ⇒ Transmitter still busy writing last char
    - I.E. bit discussed later
- Data register rightmost byte has data
  - Receiver: last char from keyboard; rest = 0
  - Transmitter: when write rightmost byte, writes char to display



I/O Polling Example

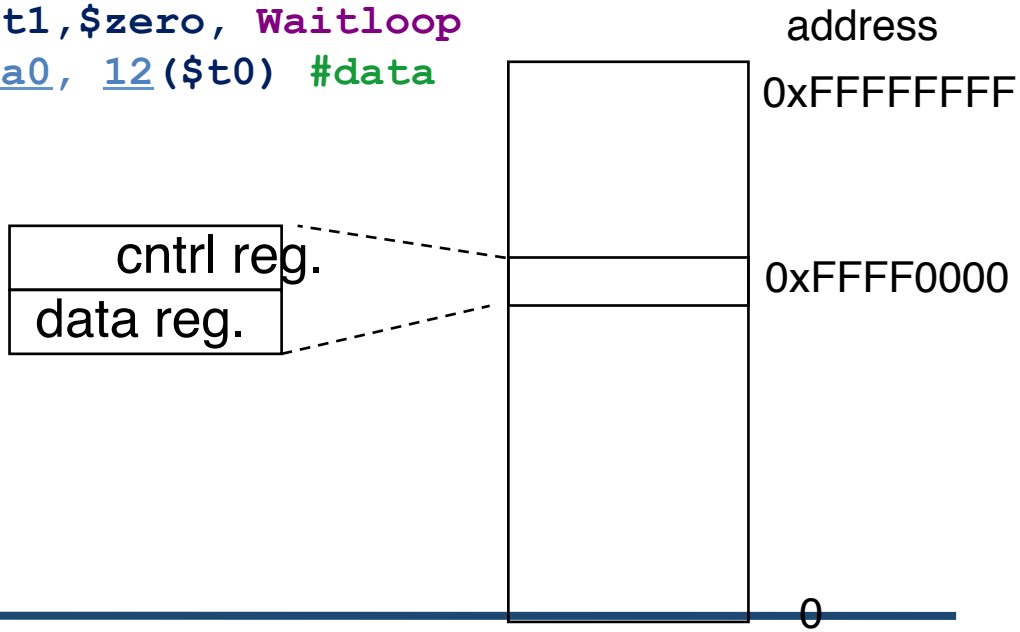
- Input: Read from keyboard into $v0

```
                    lui     $t0, 0xffff #ffff0000
Waitloop:           lw      $t1, 0($t0) #control
                    andi    $t1,$t1,0x1
                    beq     $t1,$zero, Waitloop
                    lw      $v0, 4($t0) #data, then extract byte
```

- Output: Write to display from $a0

```
                    lui     $t0, 0xffff #ffff0000
Waitloop:           lw      $t1, 8($t0) #control
                    andi    $t1,$t1,0x1
                    beq     $t1,$zero, Waitloop
                    sw      $a0, 12($t0) #data
```

- Processor waiting for I/O called "Polling"



Cost of Polling: A mouse, floppy disk, and a hard disk

- Assume for a processor with a 1GHz clock, it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning). Determine % of processor time for polling, assuming that you poll often enough not to miss data and that the devices are always busy: (Here assume 1K=1000)

- Mouse: Given that mouse has to be polled 30 times/sec so as not to miss user movement

  Mouse Polling, Clocks/sec

  = 30 [polls/s] * 400 [clocks/poll] = 12K [clocks/s]

  % Processor for polling:

  $12*10^3$ [clocks/s] / $1*10^9$ [clocks/s] = 0.0012%

  ⇒ Polling mouse little impact on processor

Cost of Polling a Floppy Disk

- Assume for a processor with a 1GHz clock, it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning). Determine % of processor time for polling, assuming that you poll often enough not to miss data and that the devices are always busy: (Here assume 1K=1000)

- Floppy disk: Transfers data in 2-Byte units and has a data rate of 50 KB/second. No data transfer can be missed.

  Frequency of Polling Floppy (must poll faster than the disk can generate data)

  = 50 [KB/s] / 2 [B/poll] = 25K [polls/s]

  Floppy Polling, Clocks/sec

  = 25K [polls/s] * 400 [clocks/poll] = 10M [clocks/s]

  % Processor for polling:

  $10*10^6$ [clocks/s] / $1*10^9$ [clocks/s] = 1%

  ⇒ OK if not too many I/O devices

Cost of Polling a Hard Disk

- Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second. Again, no transfer can be missed.

  Frequency of Polling Disk

  = 16 [MB/s] / 16 [B] = 1M [polls/s]

  Disk Polling, Clocks/sec

  = 1M [polls/s] * 400 [clocks/poll]

  = 400M [clocks/s]

  % Processor for polling:

  $400*10^6$ [clocks/s] / $1*10^9$ [clocks/s] = 40%

  ⇒ Unacceptable!

Alternative to Polling

- What is the alternative to polling?
- Wasteful to have processor spend most of its time “spin-waiting” for I/O to be ready.
- Would like an unplanned procedure call that would be invoked only when I/O device is ready.
- Solution:
  - Use exception mechanism to help I/O.
- Interrupt program when I/O is ready, return when done with data transfer.

I/O Interrupt

- An I/O interrupt is like overflow exceptions, except that:
- An I/O interrupt is “asynchronous” (can happen any time even within a clock cycle)
  - More information needs to be conveyed
- An I/O interrupt is asynchronous with respect to instruction execution stream:
- I/O interrupt is not associated with any instruction, but it can happen in the middle of any given instruction.
- I/O interrupt does not prevent any instruction from completion.
  - For example, every keystroke generates an interrupt signal.
- Interrupts can also be generated by other devices, such as a printer, to indicate that some event has occurred.
  - These are called hardware interrupts.
- Interrupt signals initiated by programs are called software interrupts.
  - A software interrupt is also called a trap (synchronous) or an exception (asynchronous).

PC I/O Interrupts

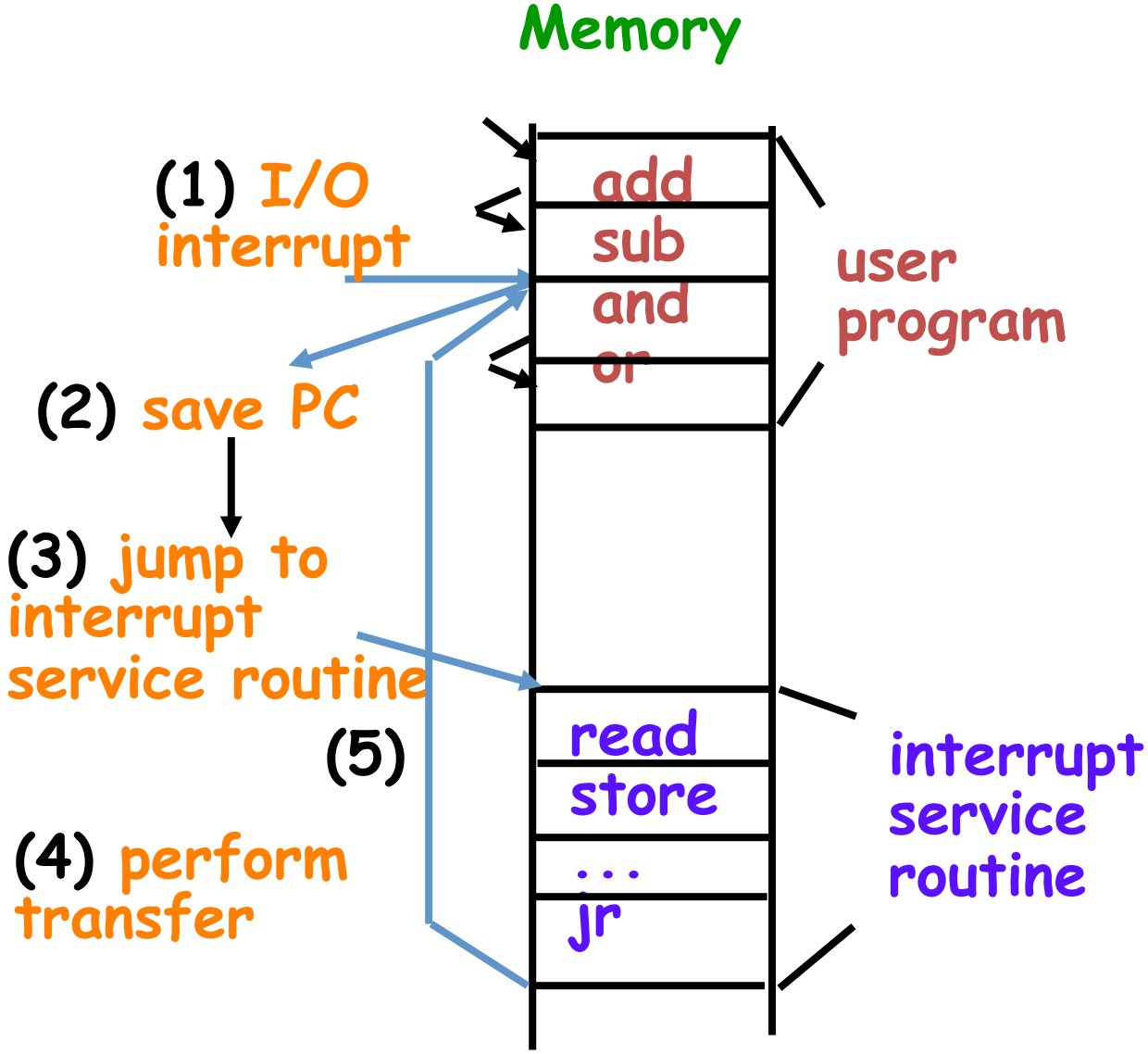
- PCs support:
  - 256 types of software interrupts, and
  - 15 hardware interrupts.
- Each type of software interrupt is associated with an *interrupt handler* or *interrupt service routine* -- a routine that takes control when the interrupt occurs.
- For example, when you press a key on your keyboard, this triggers a specific interrupt handler.
- The complete list of interrupts and associated interrupt handlers is stored in a table called the *interrupt vector table*, which resides in the first 1K of addressable memory.

Interrupt Request Lines (IRQ)

- IRQs are hardware lines over which devices can send interrupt signals to the μprocessor.
  - When a new device is added to a PC, its IRQ number must be set by a DIP switch.
  - This specifies which interrupt line the device may use.

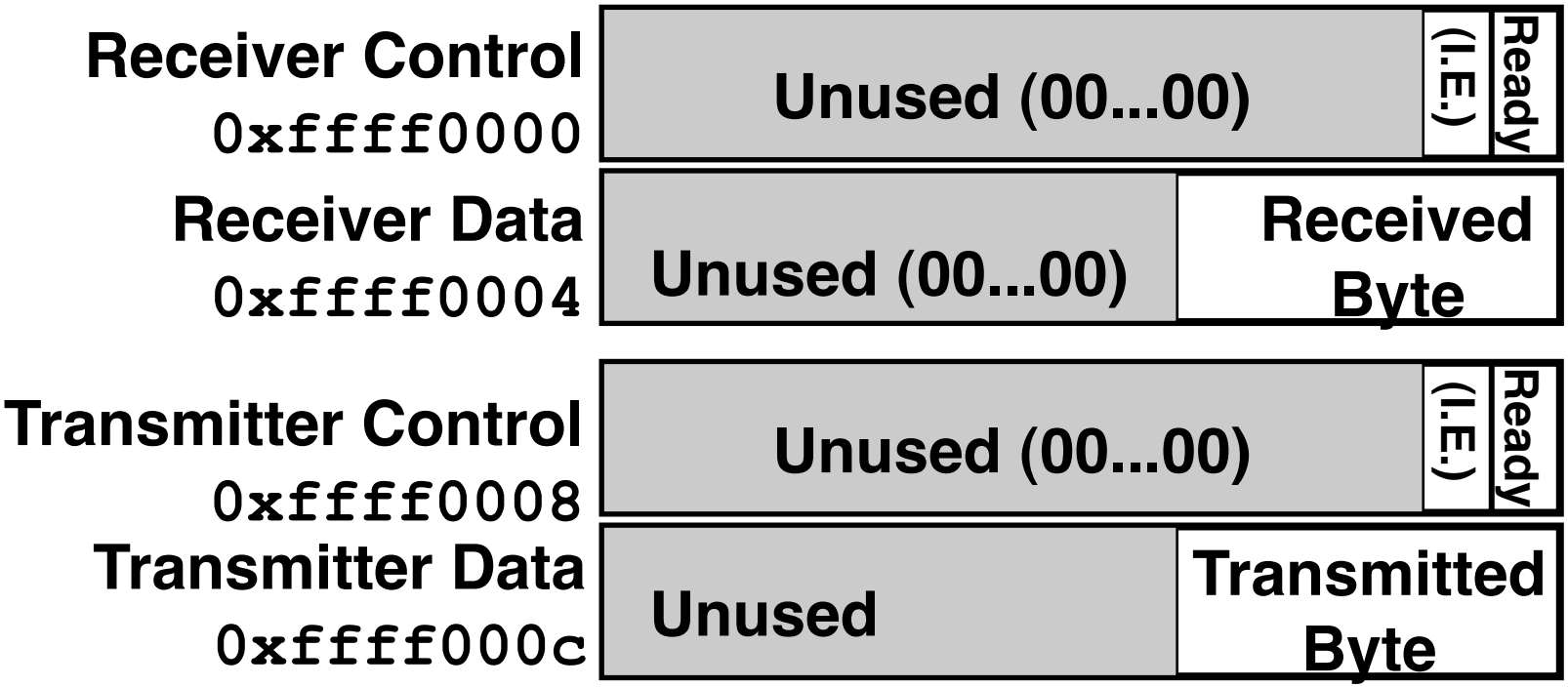
IRQ Number	Typical Use	Description
IRQ 0	System timer	This interrupt is reserved for the internal system timer. It is never available to peripherals or other devices.
IRQ 1	Keyboard	This interrupt is reserved for the keyboard controller. Even on devices without a keyboard, this interrupt is exclusively for keyboard input.
IRQ 2	Cascade interrupt for IRQs 8-15	This interrupt cascades the second interrupt controller to the first.
IRQ 3	Second serial port (COM2)	The interrupt for the second serial port and often the default interrupt for the fourth serial port (COM4).
IRQ 4	First serial port (COM1)	This interrupt is normally used for the first serial port. On devices that do not use a PS/2 mouse, this interrupt is almost always used by the serial mouse. This is also the default interrupt for the third serial port (COM3).
IRQ 5	Sound card	This interrupt is the first choice that most sound cards make when looking for an IRQ setting.
IRQ 6	Floppy disk controller	This interrupt is reserved for the floppy disk controller.
IRQ 7	First parallel port	This interrupt is normally reserved for the use of the printer. If a printer is not being used, this interrupt can be used for other devices that use parallel ports.
IRQ 8	Real-time clock	This interrupt is reserved for the system's real-time clock timer and can not be used for any other purpose.
IRQ 9	Open interrupt	This interrupt is typically left open on devices for the use of peripherals.
IRQ 10	Open interrupt	This interrupt is typically left open on devices for the use of peripherals.
IRQ 11	Open interrupt	This interrupt is typically left open on devices for the use of peripherals.
IRQ 12	PS/2 mouse	This interrupt is reserved for the PS/2 mouse on machines that use one. If a PS/2 mouse is not used, the interrupt can be used for other peripherals, such as network card.
IRQ 13	Floating point unit/coprocessor	This interrupt is reserved for the integrated floating point unit. It is never available to peripherals or other devices as it is used exclusively for internal signaling.
IRQ 14	Primary IDE channel	This interrupt is reserved for use by the primary IDE controller. On systems that do not use IDE devices, the IRQ can be used for another purpose.
IRQ 15	Secondary IDE channel	This interrupt is reserved for use by the secondary IDE controller.

Interrupt Driven Data Transfer



SPIM I/O Simulation: Interrupt Driven I/O

- I.E. stands for Interrupt Enable
- Set Interrupt Enable bit to 1 have interrupt occur whenever Ready bit is set



Benefit of Interrupt-Driven I/O

- Does Interrupt-driven I/O solve the problem with the hard disk in the previous example?
  - (Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second.)

- Find the % of processor consumed if the hard disk is only transferring data 5% of the time. Assuming 500 clock cycle overhead for each transfer, including interrupt:
  - Disk Interrupts/s = 16 MB/s / 16B/interrupt
    = 1M interrupts/s
  - Disk Interrupts, clocks/s
    = 1M interrupts/s * 500 clocks/interrupt
    = 500,000,000 clocks/s
  - % Processor consumed during transfer:
    $500*10^6 / 1*10^9 = 50\%$
- Disk active 5% $\Rightarrow$ 5% * 50% $\Rightarrow$ 2.5% busy