lecture 20

Input / Output (I/O) 2

- isolated vs. memory mapped I/O
- program controlled I/O: polling
- direct memory access (DMA)

Wed. March 23, 2016

Isolated I/O

In MARS simulator of MIPS, we uses syscall for I/O e.g. reading from keyboard, printing to console.

Conceptually, what happens there?

You can think of MARS as pausing, and your computer's operating system performing the requested operation.

Real MIPS processors do not use syscalls for I/O.

Isolated I/O is a more general and common approach. The processor has *special instructions for specific I/O* operations e.g. which refer to specific I/O registers.

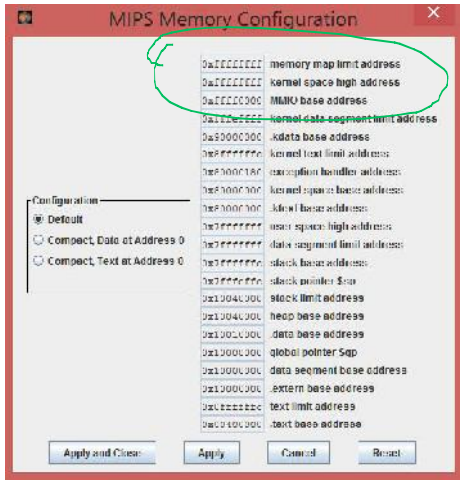
Memory Mapped I/O

"Memory mapped" I/O (MMIO) is a different approach.

MMIO is used in real MIPS processors.

It uses addresses from 0xffff0000 to 0xffffffff (in kernel).
(64 KB ~ 2^16 bytes)

MARS can be configured to use MMIO.



console output data register

0x ffff 000c

console output control register

0x ffff 0008

console input data register

0x ffff 0004

console input control register

0x ffff 0000


```
lui   $t0, 0xffff
lw    $s0, 4($t0)
```

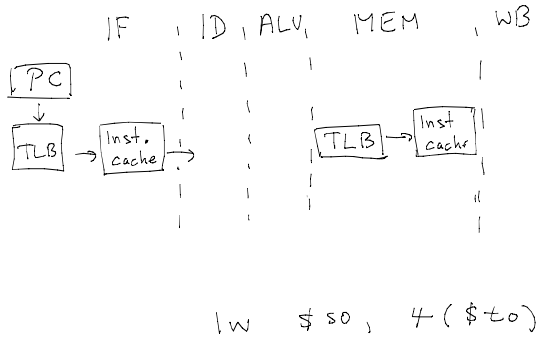
It only loads a byte, but let's ignore that detail.

console input data register

0x ffff 0004

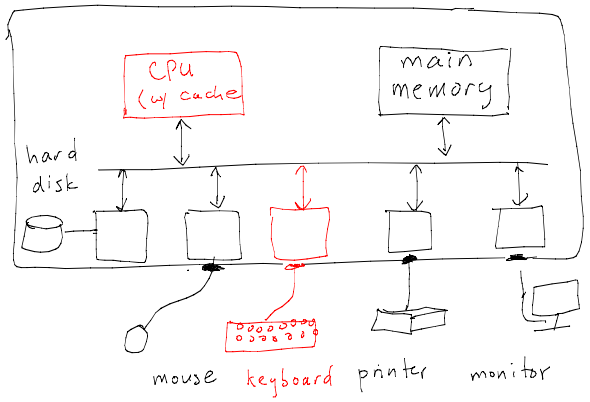
What happens physically ?

There is a circuit that detects that this is a memory mapped address, and that loads the word from the register instead....



Rather than reading from memory in the usual way, there would be a branch to the I/O exception handler.

Let's examine this in more detail (AND MORE GENERAL).
(console input = keyboard)



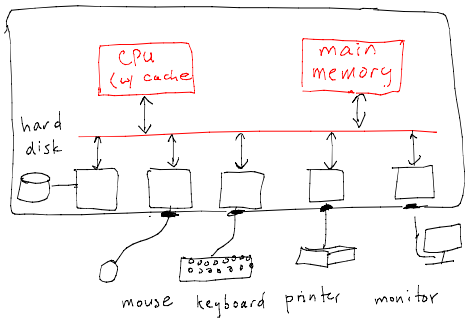
- branch to kernel's I/O exception handler
- CPU translates virtual address 0xffff0004 to I/O device address, namely keyboard ID (details not specified. hardware implementation dependent)
- CPU write I/O device address on the address bus and sets ReadIO control to 1
- keyboard controller puts byte onto the data bus (last lecture)
- CPU reads data bus

last lecture

- kernel jumps back to user program, which continues execution (and byte is loaded into register)

Note similarity to TLB or instruction/data cache miss.

Addresses, data, controls need to be put on bus.



- same idea for I/O output

```
lui   $t0, 0xffff
sw    $s0, 12($t0)
```

We sort of skipped an issue:

CPU to device : "Ready or not ? "

- Does input device have a (new) input ?
- Has output device displayed the previous output (s) ?

Program contolled I/O: Polling

(Are you ready? Are you ready? .....)

console input control register

0x ffff 0000

console input data register

0x ffff 0004

```
    lui  $t0, 0xffff
Wait:
    lw     $t1, 0($t0)          # load from the input control register
    andi   $t1, $t1, 0x0001     # reset (clear) all bits except LSB
    beq    $t1, $zero, Wait     # if not ready, then loop back

    lw     $s2, 4( $t0)         # input device is ready, so read
```

Kernel would only let a process run for a limited number of clock cycles before pausing and giving control to another process, and later returning. So, no worries about an infinite loop.

In principle, polling can be done with isolated IO or memory mapped IO. (The concept is general.)

We cannot say how MIPS implements polling. MIPS is specified at the assembly language level.

Polling (e.g. output)

console output data register

0x ffff 000c

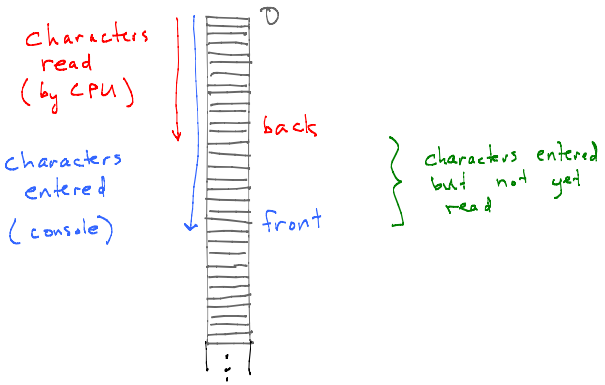
console output control register

0x ffff 0008

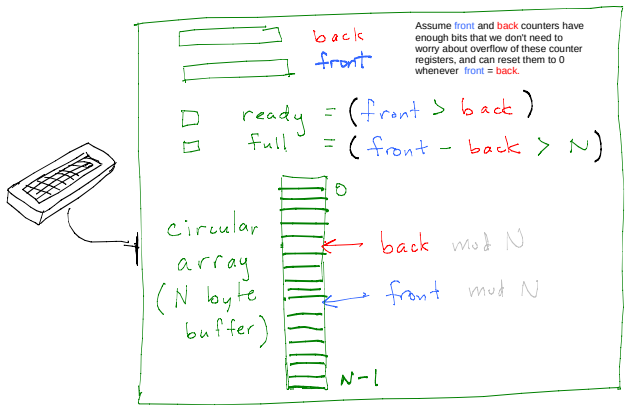

```
    lui  $t0, 0xffff
Wait:
    lw     $t1, 8($t0)          # load the output control reg
    andi   $t1, $t1, 0x0001     # reset all bits except LSB
    beq    $t1, $zero, Wait     # if not ready, then loop back

    sw     $s2, 12( $t0 )       # output device ready, so write
```

User types on keyboard and characters are echoed in display window. But sometimes the echoed characters are delayed (CPU or bus is busy). Then a burst occurs as the CPU catches up.

What's going on (with the burst) ?



(Sketch only) Suppose the keyboard controller has a buffer which is a circular array.



User presses a key:

```
if (front - back <  N)    //  buffer is not full
   front = front + 1
   buffer[ front mod N] = key
```

CPU reads a character:

```
buffer[ back mod N]  is put onto data bus
back = back + 1
```

Burst is due to CPU rapidly reading a sequence of characters from the buffer, once system bus is free. (Assume CPU echos each character right away to the console.)

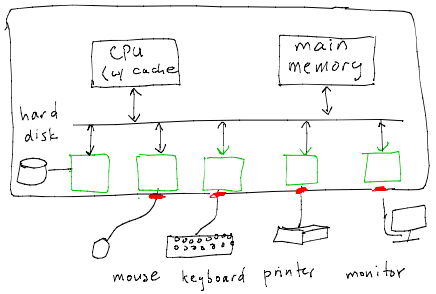
lecture 20

Input / Output (I/O) 2

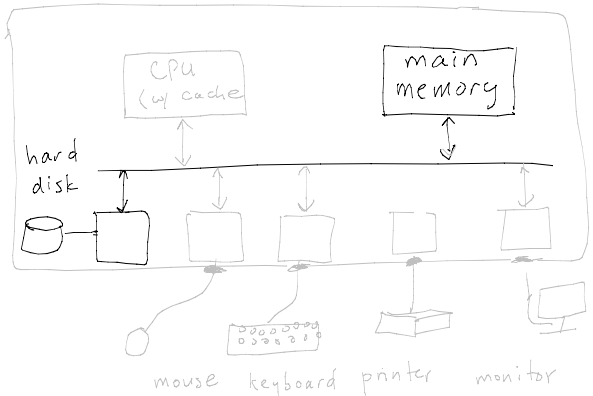
- isolated vs. memory mapped I/O
- program controlled I/O: polling
- direct memory access (DMA)

Wed. March 23, 2016

Suppose page fault occurs. Then, kernel must coordinate a page swap. How ? (sketch)



Direct Memory Access (DMA)

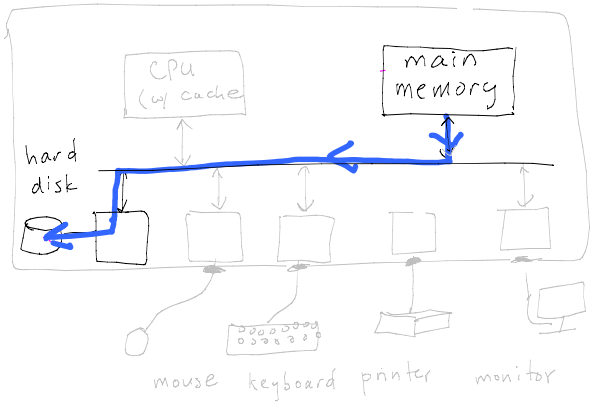


DMA (for page fault): Step 1

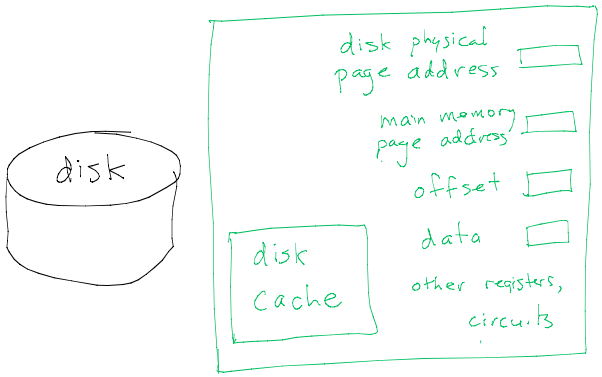
CPU writes onto system bus

- HDD controller's address (device ID) (on address bus)
- instruction for HDD controller (on control bus)
- physical *address* of pages (on *data* bus)

DMA (for page fault): Step 2 (desired)

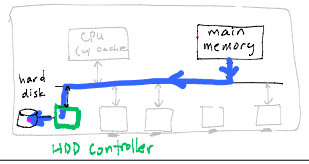


HDD controller takes over.



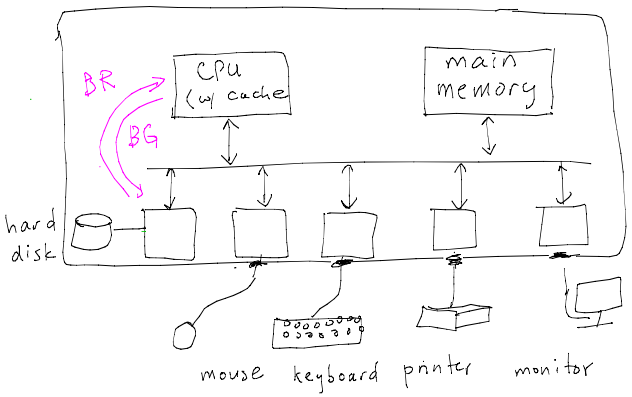
HDD controller

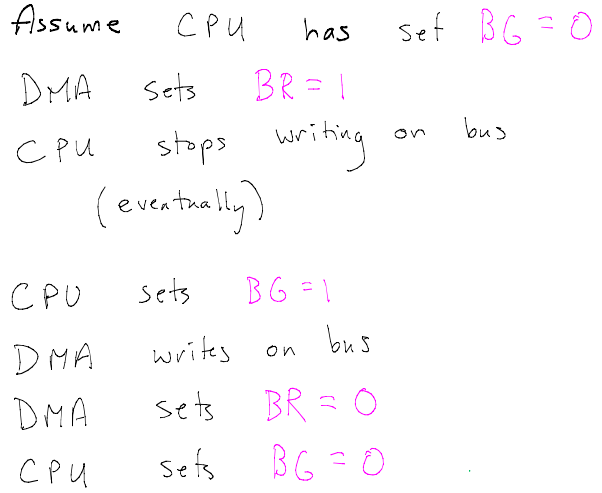
- gets write access to system bus (How?)
- instructs main memory to write on the system bus, and reads the page word by word, storing in local memory ("disk cache" on previous slide) on the HDD controller (How ?)
- transfers the page from disk cache to the hard disk when the HD is at the right physical position



How does the DMA (HDD) controller gets write access to system bus, after it has retrieved a page from disk ?

If it doen't currently have access to the bus, then how can it tell the CPU that it wants access ?

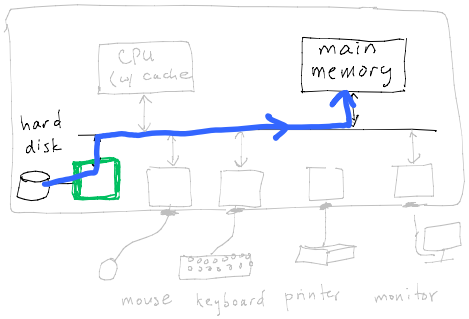




DMA controller (i.e. HDD controller) instructs main memory to write on the system bus. DMA controller then reads the page word by word, storing the words in its local memory. Here is pseudocode sketching what the DMA controller needs to do.

```
initialize baseReg to the address where the page will be go in the
  local memory ("disk cache")  of the HDD controller

while offset = 0 to pagesize -1 {
     put control signals on the control bus so main memory
     writes a word onto data bus,  and put address AddrMainMem
     on address bus

     read word from data bus and put it into disk cache  at
               address (baseReg + offset)
     AddrMainMem = AddMainMem + 4     //  1 word
}
```

Similar steps for transferring page from HDD to main memory.



ASIDE: Improving HDD performance ?

read/write multiple pages at a time

- good if neighboring physical pages correspond to neighboring virtual pages (disk fragmentation issue)
- order list of physical pages that have been requested