CS 2461: Computer Architecture I

Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~bhagiweb/cs2461/

The von Neumann Model

- Memory: holds both data and instructions
- Processing Unit: carries out the instructions
- Control Unit: sequences and interprets instructions
- Input: external information into the memory
- Output: produces results for the user

I/O: Connecting to Outside World

- So far, we’ve learned how to:
  - compute with values in registers
  - load data from memory to registers
  - store data from registers to memory
- But where does data in memory come from?
  - wide variety of Input devices
- And how does data get out of the system so that humans can use it?
  - Wide variety of output devices

I/O: Connecting to the Outside World

- Types of I/O devices characterized by:
  - behavior: input, output, storage
    - input: keyboard, motion detector, network interface
    - output: monitor, printer, network interface
    - storage: disk, CD-ROM
  - data rate: how fast can data be transferred?
    - keyboard: 100 bytes/sec
    - disk: 30 MB/s
    - network: 1 Mb/s - 1 Gb/s
- We stick to keyboard and display
  - Cover basic concepts of I/O processing
  - Similar solutions used in real processors
Interacting with I/O Devices

- What do we need to know about I/O devices?

- Only two aspects:
  - Are they ready to process CPU’s request?
  - Where to send the data to be processed by I/O device?

I/O Devices and Controllers

Most I/O devices are not purely digital themselves …

- Electro-mechanical: e.g., keyboard, mouse, disk, motor
- Analog/digital: e.g., network interface, monitor, speaker, mic

... all have digital interfaces presented by I/O Controller

- CPU (digital) talks to controller
- Not super-interested in controller/device internals for now.

I/O Controller Interface: Abstraction

- I/O Controller interface presented as device registers
  - Control/status: may be one register or two
  - Data: may be more than one of these
- For input:
  - CPU checks status register if input is available
  - Reads input from data register (or waits if no input)

- For output:
  - CPU checks status register to see if it can write (device free)
  - Writes output to data register
- Device electronics
  - performs actual operation
  - pixels to screen, bits from disk, characters from keyboard

Programming Interface

- How are device registers identified?
  - Memory-mapped vs. special instructions

- How is timing of transfer managed?
  - Asynchronous vs. synchronous

- Who controls transfer?
  - CPU (polling) vs. device (interrupts)
Transfer Timing

- I/O events generally happen much slower than CPU cycles.

Synchronous
- data supplied at a fixed, predictable rate
- CPU reads/writes every X cycles

Asynchronous
- data rate less predictable
- CPU must synchronize with device, so that it doesn’t miss data or write too quickly
  - How: some protocol is needed

Synchronous or Asynch.

- TV and Remote?
- Mail delivery person and you?
- Mouse and PC?

TV and Remote: synchronous
- TV samples at specific intervals to see if key on remote has been pressed

Mail delivery: asynchronous
- Use mailbox as synchronization mechanism

Mouse and PC: synchronous
- PC samples mouse at specific intervals

How are Device Register Reads/Writes Performed?

Two options (aren’t there always?)

I/O instructions
- Designate opcode(s) for I/O
- Register and operation encoded in instruction

Memory-mapped I/O
- Assign a memory address to each device register
- Use conventional loads and stores
- Hardware intercepts loads/stores to these address
- No actual memory access performed
- LC3 (and most other platforms) do this

Memory-Mapped vs. I/O Instructions

Instructions
- designate opcode(s) for I/O
- register and operation encoded in instruction

Memory-mapped
- assign a memory address to each device register
- use data movement instructions (LD/ST) for control and data transfer
Memory-mapped I/O (Table A.3)

<table>
<thead>
<tr>
<th>Location</th>
<th>I/O Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xFE00</td>
<td>Keyboard Status Reg (KBSR)</td>
<td>Bit [15] is one when keyboard has received a new character.</td>
</tr>
<tr>
<td>xFE02</td>
<td>Keyboard Data Reg (KBDR)</td>
<td>Bits [7:0] contain the last character typed on keyboard.</td>
</tr>
<tr>
<td>xFE04</td>
<td>Display Status Register (DSR)</td>
<td>Bit [15] is one when device ready to display another char on screen.</td>
</tr>
<tr>
<td>xFE06</td>
<td>Display Data Register (DDR)</td>
<td>Character written to bits [7:0] will be displayed on screen.</td>
</tr>
</tbody>
</table>

Input from Keyboard

- When a character is typed:
  - It is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
  - the "ready bit" (KBSR[15]) is set to one
  - keyboard is disabled -- any typed characters will be ignored

- When KBDR is read: the keyboard HW
  - KBSR[15] is set to zero
  - keyboard is enabled

Basic Input Routine

Check if Keyboard ready
Keep checking....
How?
Once ready, it reads from Keyboard into register R0

Simple Implementation: Memory-Mapped Input

Address Control Logic determines whether MDR is loaded from Memory or from KBSR/KBDR. If address = xFE00 then KBSR

KBSR

KBDR

Keyboard data

read character

new char?

Polling

NO

YES

Keep checking....

How?

Once ready, it reads from Keyboard into register R0
Basic Input Routine

- **New char?**
  - **Polling**
  - **Read character**
  - **POLL LDI R0, KBSRPtr**
  - **BRzp POLL**
  - **LDI R0, KBDRPtr**
  - **KBSRPtr .FILL xFE00**
  - **KBDRPtr .FILL xFE02**

Output to Monitor

- When Monitor is ready to display another character:
  - The “ready bit” (DSR[15]) is set to one
- When data is written to Display Data Register:
  - DSR[15] is set to zero
  - Character in DDR[7:0] is displayed
  - Any other character data written to DDR is ignored (while DSR[15] is zero)

Keyboard Echo Routine

- Usually, input character is also printed to screen.
  - User gets feedback on character typed and knows it’s ok to type the next character.

Some Questions

- What is the danger of not testing the DSR before writing data to the screen?
- What is the danger of not testing the KBSR before reading data from the keyboard?

What if the Monitor were a synchronous device, e.g., we know that it will be ready 1 microsecond after character is written.
  - Can we avoid polling? How?
  - What are advantages and disadvantages?
Who writes the I/O code?

- Trap Routines/Service calls

- Not a good idea to let programmers write their code to do I/O?
- Send the request to the "system"
  - OS will service the request and return control back to user program
  - Eg: Printf

System Calls

- Certain operations require specialized knowledge and protection:
  - Specific knowledge of I/O device registers and the sequence of operations needed to use them
  - I/O resources shared among multiple users/programs; a mistake could affect lots of other users!
- Not every programmer knows (or wants to know) this level of detail
- Provide service routines or system calls (part of operating system) to safely and conveniently perform low-level, privileged operations

System Call - steps

- 1. User program invokes system call.
- 2. Operating system code performs operation.
- 3. Returns control to user program.

Role of the OS
how do I get my code to “ask” the OS for I/O?
  - Call a special subroutine, called a “TRAP”
  - Also called syscall or callgate
- don’t simply use a Branch or Function call:
  - not “secure” enough
  - User can’t set privilege bit themselves
  - great temptation to give themselves powers they shouldn’t have

In the Real World

- System call/TRAP Specifics:
  - User can call to a restricted set of function addresses
  - Can upgrade privilege only through these channels
- This system call mechanism is commonly used in actual systems
  - As an example the BIOS (Basic Input Output System) on many Intel PCs provided precisely this functionality to allow programs to access basic input and output devices, keyboards, displays, timers etc.
- More modern systems use EFI (Extensible Firmware Interface) which is a more sophisticated version of the same thing.

LC-3 TRAP Mechanism

1. A set of service routines.
   - part of operating system -- routines start at arbitrary addresses (convention is that system code is below x3000)
   - up to 256 routines
2. Table of starting addresses.
   - stored at x0000 through x00FF in memory
   - called System Control Block in some architectures
3. TRAP instruction.
   - used by program to transfer control to operating system
   - 8-bit trap vector names one of the 256 service routines
4. A linkage back to the user program.
   - want execution to resume immediately after the TRAP instruction

LC3 TRAP Routines and their Assembler Names

<table>
<thead>
<tr>
<th>vector</th>
<th>symbol</th>
<th>routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>x20</td>
<td>GETC</td>
<td>read a single character (no echo)</td>
</tr>
<tr>
<td>x21</td>
<td>OUT</td>
<td>output a character to the monitor</td>
</tr>
<tr>
<td>x22</td>
<td>PUTS</td>
<td>write a string to the console</td>
</tr>
<tr>
<td>x23</td>
<td>IN</td>
<td>print prompt to console, read and echo character from keyboard</td>
</tr>
<tr>
<td>x25</td>
<td>HALT</td>
<td>halt the program</td>
</tr>
</tbody>
</table>
**TRAP Instruction**

<table>
<thead>
<tr>
<th>TRAP</th>
<th>Trap vect8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

- **Trap vector**
  - Identifies which system call to invoke
  - 8-bit index into table of service routine addresses
  - In LC-3, this table is stored in memory at 0x0000 - 0x00FF
  - 8-bit trap vector is zero-extended into 16-bit memory address

- **Where to go**
  - Lookup starting address from table; place in PC
  - Load contents at trap vector address into the PC!

- **How to get back**
  - Save address of next instruction (current PC) in R7
  - Last instruction in TRAP program sets PC equal to R7

**NOTE:** PC has already been incremented during instruction fetch stage.

---

**RET (JMP R7)**

- How do we transfer control back to instruction following the TRAP?

- We saved old PC in R7.
  - JMP R7 gets us back to the user program at the right spot.
  - LC-3 assembly language lets us use RET (return) in place of “JMP R7”.

- Must make sure that service routine does not change R7, or we won’t know where to return.
**TRAP Mechanism Operation**

1. Lookup starting address.
2. Transfer to service routine.
3. Return (JMP R7).

**Example: Using the TRAP Instruction**

```
.ORIG x3000
  ; user code
  TRAP x23  ; input character into R0
  ADD R1, R2, R0  ; use R0
  ADD R0, R0, R3  ; load output data into R0
  TRAP x21  ; Output to monitor...
  EXIT  TRAP x25  ; halt
.END
```

**Question**

- Can a service routine call another service routine?
- If so, is there anything special the calling service routine must do?
How do actual I/O interactions take place – Protocols?

Two schemes for interacting with I/O devices

What we have seen so far is polling

- “Are we there yet? Are we there yet? Are we there yet?”
- CPU keeps checking status register in a loop
- Very inefficient, multi-tasking CPU has better things to do

Alternative scheme is called interrupts

- “Wake me when we get there.”
- Device sends special signal to CPU when status changes
- CPU stops current program, saves its state
- CPU “handles interrupt”: checks status, moves data
- CPU resumes stopped program, as if nothing happened!!!!!!!

Example 1

- Download, assemble and run
  - IO-example1.asm
  - Enter 2 for first number
  - Enter 3 for second number
  - What does program print out?
  - Why?

Example 2

- ASCII to Integer Conversion
- Download, assemble and run
  - IO-example2.asm
  - Enter 2 for first number
  - Enter 3 for second number
  - What does program print out?

Example 3

- Download, assemble and run
  - IO-example3.asm
  - Enter 2 for first number
  - Enter 3 for second number
  - What does program print out?
  - Why? What is going wrong?
Example 4

- Use Windows LC3 simulator
- Download
  - Data1.obj
  - Data2.obj
- IO-example4.asm
- First load data1.obj
- Next load data2.obj
- Finally load IO-example4.obj
  - (this is same as code in Example 2 to add two numbers)
- Run program…
- ???
- Maybe bug in example4
  - Load IO-example2.obj and run….
- what is going on?