Chapter 8 & 9.1
I/O and Traps
Aside Off-Topic: Memory Hierarchy

**SPEED**
- CPU Registers: Very Fast (1 ns)
- Cache: Very Fast (10 ns)
- RAM: Fast (100 ns)
- Hard Disk: Slow (10 ms)
- Off-Line Storage (Tape Drives, etc.): Very Slow (in seconds)

**SIZE**
- Very Small (512 Bytes)
- Small (12 MB)
- Large (8 GB)
- Very Large (2 TB)
- Potentially Huge (PBs)
- Least Expensive

**COST**
- Very Expensive (part of CPU)
- Very Expensive ($150/MB)
- Inexpensive ($0.58/MB)
- Very Inexpensive ($0.0025/MB)
Why memory hierarchy?

Performance of Computer processor VS Memory Processor
Memory Hierarchy

- Smaller
- Faster
- More expensive per byte

- CPU
  - Regs
- L1 cache
  - SRAM
- L2 cache
  - SRAM
- Main memory
  - DRAM
- Hard disk
  - Magnetics

- Larger
- Slower
- Cheaper per byte
Why do caches work or do not work?

Principle of Locality

- Temporal locality If the location A is accessed now, it’ll be accessed again soon
- Spatial locality If the location A is accessed now, the location nearby (e.g., A+1) will be accessed soon

```java
for (i = 0; i < cars.length; i++) {
    text += cars[i] + "<br>";
}
```
I/O: Connecting to Outside World

So far, we’ve learned how to:

• compute with values in registers
• alter the sequence of instructions
• load data from memory to registers
• store data from registers to memory

But where does data in memory come from?

And how does data get out of the system so that humans can use it?
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
I/O Controller Registers

- CPU interacts with IO devices through registers

- How many registers do we need to interact with the I/O controller?
- How do we interface with I/O registers, since we can reference only eight registers in LC3 ISA and all eight have been mapped to LC-3 ISA registers?
How many registers do we need to interact with the I/O controller?

• I/O device and LC3 ALU need to interact with each other
  ➢ 1 Data Register
  ➢ 16 bits

• For a Keyboard one key generates only 8-bit ASCII number
  ➢ We will design a 16 bit data register but only use 8 bits from it
  ➢ 16 bit register is set up only to simplify the interface design
How many registers do we need to interact with the I/O controller?

• How will the LC-3 know when to use the keyboard data register (KBDR) value?
  ➢ 1 Data Register
  ➢ 16 bits

• If LC-3 keeps sourcing KBDR, it may get stale data from KBDR

• We need the LC-3 to source KBDR only when it has a valid entry
  ➢ For this we set up a Status Register to see if the data value in Data register is Valid or not
  ➢ 16 bit status register
  ➢ use only one bit from the status register
How many registers do we need to interact with the I/O controller?

- We need min of two registers in the IO controller to interact with LC-3 (any CPU)
  - 8 bit data register
  - 1 bit status register
  - we use 16 data and status register to simplify the interface design
I/O Controller (INPUT)

Control/Status Registers
- LC-3 checks whether data is ready -- read status register

Data Registers
- LC-3 transfers data to/from device

Device electronics
- performs actual operation
I/O Controller (OUTPUT)

Control/Status Registers

- LC3 tells device what to do -- write to control register
- LC3 checks whether task is done -- read status register

Data Registers

- LC3 transfers data to/from device

Device electronics

- performs actual operation
  - pixels to screen, bits to/from disk, characters from keyboard
Programming Interface

How do we interface with I/O registers, since we can reference only eight registers in LC3 ISA and all eight have been mapped to LC-3 ISA registers?

• Memory-mapped vs. special instructions
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)
Memory-Mapped vs. I/O Instructions

Instructions

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

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>Device</td>
<td>Op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory-mapped

• assign a memory address to each device register
• use data movement instructions (LD/ST) for control and data transfer
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
Transfer Control

Who determines when the next data transfer occurs?

Polling
• CPU keeps checking status register until new data arrives OR device ready for next data
• “Are we there yet? Are we there yet? Are we there yet?”

Interrupts
• Device sends a special signal to CPU when new data arrives OR device ready for next data
• CPU can be performing other tasks instead of polling device.
• “Wake me when we get there.”
Points Covered So Far

• Setting up a Memory Hierarchy
  • Offset the memory latency
  • Registers to Off-Line Storage
  • Principle of Locality

• Connecting an IO device with LC-3
  • Two registers required
    ➢ One for data transfer
    ➢ One for Synchronization

• Programing Interface
  • Memory-mapped vs Special Instructions
  • Asynchronous vs Synchronous
  • Polling vs Interrupts
**LC-3**  
**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>

**Asynchronous devices**  
- synchronized through status registers  

**Polling and Interrupts**  
- the details of interrupts will be discussed in Chapter 10
Input from Keyboard

When a character is typed:
- its ASCII code 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:
- KBSR[15] is set to zero
- keyboard is enabled
Basic Input Routine

Polling

new char?

POLL  LDI  R0, KBSRPtr  
BRzp  POLL  
LDI  R0, KBDRPtr  

KBSRPtr .FILL xFE00  
KBDRPtr .FILL xFE02
Address Control Logic determines whether MDR is loaded from Memory or from KBSR/KBDR.
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)
Exercise: Write the Basic Output Routine

Screen Ready?

NO

Polling

YES

write character

Input Routine for reference

```
POLL    LDI    R0, KBSRPtr
        BRzp   POLL
        LDI    R0, KBDRPtr
        ...

KBSRPtr  .FILL  xFE00
KBDRPtr  .FILL  xFE02
```
**Basic Output Routine**

**Polling**

- screen ready?
  - NO
  - write character
  - YES
  - POLL LDI R1, DSRPtr
  - BRzp POLL
  - STI R0, DDRPtr
  
  ...  

  - DSRPtr .FILL xFE04
  - DDRPtr .FILL xFE06
Simple Implementation: Memory-Mapped Output

Sets LD.DDR or selects DSR as input.

Note: Writing data values to the Data Register.
How would you know if you CPU is reference the right character?

- Echo out the input key in the Display
Write an 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.
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.

```
POLL1   LDI  R0, KBSRPtr
        BRzp POLL1
        LDI  R0, KBDRPtr
POLL2   LDI  R1, DSRPtr
        BRzp POLL2
        STI  R0, DDRPtr

...  
KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
DSRPtr  .FILL xFE04
DDRPtr  .FILL xFE06
```
In-Class Exercise
2014Fall Exam4 Q5

• Complete the code snippet below to display the string.

.ORIG x3000
LEA R3, STRING

NEXT
LDR R0, R3, #0

______________________
______________________
______________________
; TODO

______________________
ADD R3, R3, #1 ; increment pointer to next char
BR NEXT

END HALT

STRING .STRINGZ "Have a Happy Thanksgiving!" ; string to print
DSR .FILL xFE04
DDR .FILL xFE06
.END
• Complete the code snippet below to display the string.

```
.ORIG x3000
LEA R3, STRING

NEXT   LDR R0, R3, #0
       BRz  END

POLL   LDI R1, DSR
       BRzp POLL
       STI R0,DDR
       ADD R3, R3, #1 ; increment pointer to next char
       BR   NEXT

END    HALT

STRING .STRINGZ "Have a Happy Thanksgiving!" ; string to print
DSR    .FILL xFE04
DDR    .FILL xFE06
.END
```
Interrupt-Driven I/O

External device can:
(1) Force currently executing program to stop;
(2) Have the processor satisfy the device’s needs; and
(3) Resume the stopped program as if nothing happened.

Why?
• Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation.
• Example: Process previous input while collecting current input. (See Example 8.1 in text.)
Interrupt-Driven I/O

To implement an interrupt mechanism, we need:

- A way for the I/O device to signal the CPU that an interesting event has occurred.
- A way for the CPU to test whether the interrupt signal is set and whether its priority is higher than the current program.

Generating Signal

- Software sets "interrupt enable" bit in device register.
- When ready bit is set and IE bit is set, interrupt is signaled.
Priority

Every instruction executes at a stated level of urgency.

LC-3: 8 priority levels (PL0-PL7)

• Example:
  ➢ Payroll program runs at PL0.
  ➢ Nuclear power plant control program runs at PL6.

• It’s OK for PL6 device to interrupt PL0 program, but not the other way around.

Priority encoder selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.
## Priority Encoder

### 4 to 2 Simple Encoder

<table>
<thead>
<tr>
<th>$I_3$</th>
<th>$I_2$</th>
<th>$I_1$</th>
<th>$I_0$</th>
<th>$O_1$</th>
<th>$O_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### 4 to 2 Priority Encoder

<table>
<thead>
<tr>
<th>$I_3$</th>
<th>$I_2$</th>
<th>$I_1$</th>
<th>$I_0$</th>
<th>$O_1$</th>
<th>$O_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$x$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$x$</td>
<td>$x$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$x$</td>
<td>$x$</td>
<td>$x$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Testing for Interrupt Signal

CPU looks at signal between STORE and FETCH phases. If not set, continues with next instruction. If set, transfers control to interrupt service routine.

More details in Chapter 10.
Full Implementation of LC-3 Memory-Mapped I/O

Because of interrupt enable bits, status registers (KBSR/DSR) must be written, as well as read.
In-Class Exercise

The 2 methods device registers are 1) memory-mapped and 2) special instructions. What is the difference between memory-mapped and special instructions?

The 2 methods of transfer timing is 1) synchronous and 2) asynchronous. When would you use one over the other?

The 2 methods of data transfer is 1) polling and 2) interrupts. Which is better if data transfer occurs frequently?
Note for all

- HW7 due today, HW8 will be uploaded online
  - remember to staple your submissions
- Exam-4 will have Chapter 7 to 9
- No final exam
- Go through your marks at Learn@UW for HW and midterms
- if you have your laptops with you, try and implement the assembly code for:
  - keyboard echo subroutine
  - Print a string “Happy easter”
Topics covered in previous class

- Implementing memory-mapped IO
  - using memory addresses to map IO data/status register
- Basic Input Routine Implementation
- Basic Output Routine Implementation
- Keyboard Echo Routine implementation
- Input Driven IO device
  - Use a priority encode to set priority
Chapter 9.1: 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

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

In LC-3, this is done through the **TRAP mechanism**.
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
TRAP Instruction

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>trapvect8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

How to get back

- save address of next instruction (current PC) in R7
NOTE: PC has already been incremented during instruction fetch stage.
TRAP Example

PC is currently 0x4000
mem[x0023] is currently x04A0

What is the contents of R7 and PC after the following instruction is executed?

0  0  1  0  0  0  1  1

R7 ← PC', PC ← mem[ZEXT(trapvect8)]

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.
RET Example

PC is currently 0x0220
R7 is currently 0x4001

What is the contents of PC after the following instruction is executed?

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
RET  1 1 0 0 0 0 0 0 Base 0 0 0 0 0 0 0
RET  1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0
```
TRAP Mechanism Operation

1. Lookup starting address.
2. Transfer to service routine.
3. Return (JMP R7).
Example: Using the TRAP Instruction

.ORIG x3000
LD R2, TERM ; Load negative ASCII ‘7’
LD R3, ASCII ; Load ASCII difference
AGAIN
TRAP x23 ; input character
ADD R1, R2, R0 ; Test for terminate
BRz EXIT ; Exit if done
ADD R0, R0, R3 ; Change to lowercase
TRAP x21 ; Output to monitor...
BRnzp AGAIN ; ... again and again...
TERM .FILL xFFC9 ; ‘7’
ASCII .FILL x0020 ; lowercase bit
EXIT TRAP x25 ; halt
.END
Example: Output Service Routine

```assembly
.ORIG x0430 ; syscall address
ST R7, SaveR7 ; save R7 & R1
ST R1, SaveR1

; ----- Write character
TryWrite LDI R1, CRTSR ; get status
BRzp TryWrite ; look for bit 15 on
WriteIt STI R0, CRTDR ; write char

; ----- Return from TRAP
Return LD R1, SaveR1 ; restore R1 & R7
LD R7, SaveR7

RET ; back to user

CRTSR .FILL xF3FC
CRTDR .FILL xF3FF
SaveR1 .FILL 0
SaveR7 .FILL 0
.END
```

stored in table, location x21
Assembler Names and their TRAP Routines

<table>
<thead>
<tr>
<th>Code</th>
<th>Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETC</td>
<td>TRAP x20</td>
<td>Read one character from keyboard. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>OUT</td>
<td>TRAP x21</td>
<td>Write one character (in R0[7:0]) to console.</td>
</tr>
<tr>
<td>PUTS</td>
<td>TRAP x22</td>
<td>Write null-terminated string to console. Address of string is in R0.</td>
</tr>
<tr>
<td>IN</td>
<td>TRAP x23</td>
<td>Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>HALT</td>
<td>TRAP x25</td>
<td>Halt execution and print message to console.</td>
</tr>
</tbody>
</table>
Basic Input Routine

```asm
; the TRAP vector table
.ORIG x0000

; start of user program
.GETC

; get character input
.HALT

; halt, end user program
.END

.LDI DR, label ; Load Indirect
DR ← mem[mem[PC' + SEXT(PCoffset9)]] also setcc()

OS_KBSR .FILL xFE00 ; keyboard status register
OS_KBDR .FILL xFE02 ; keyboard data register

; ; ; GETC - Read a single character of input from keyboard device into R0
.TRAP_GETC

; wait for a keystroke
 BRzp TRAP_GETC

; read it and return
.RET
```
Saving and Restoring Registers

Must save the value of a register if:

• Its value will be destroyed by service routine, and
• We will need to use the value after that action.

Who saves?

• caller of service routine?
  ➢ knows what it needs later, but may not know what gets altered by called routine
• called service routine?
  ➢ knows what it alters, but does not know what will be needed later by calling routine
Example

What’s wrong with this TRAP x23 usage?
What happens to R7?

LEA  R3, Binary
LD   R6, ASCII   ; char->digit template
LD   R7, COUNT   ; initialize to 10
AGAIN TRAP  x23   ; Get char
ADD  R0, R0, R6  ; convert to number
STR  R0, R3, #0  ; store number
ADD  R3, R3, #1  ; incr pointer
ADD  R7, R7, -1  ; decr counter
BRp  AGAIN       ; more?
BRnzp NEXT

ASCII  .FILL xFFD0
COUNT  .FILL #10
Binary .BLKW #10
Saving and Restoring Registers

Called routine -- "callee-save"

- Before start, save any registers that will be altered (unless altered value is desired by calling program!)
- Before return, restore those same registers

Calling routine -- "caller-save"

- Save registers destroyed by own instructions or by called routines (if known), if values needed later
  - save R7 before TRAP
  - save R0 before TRAP x23 (input character)
- Or avoid using those registers altogether

Values are saved by storing them in memory.
Summary

Chapter 8: Input/output
- Behavior and data rate of I/O device
- Asynchronous vs. synchronous
- Polled vs. interrupt-driven
- Programmed vs. memory-mapped
- Control registers, data registers

Chapter 9: Traps and System Calls
- Hide details of I/O device interaction
- TRAP/RET instructions
- Caller- vs callee-saved registers