CS 135: Computer Architecture I

Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~bhagiweb/cs135/

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?
- 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
- 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 Controller

- Control/Status Registers
  - CPU tells device what to do – write to control register
  - CPU checks whether task is done – read status register
- Data Registers
  - CPU transfers data to/from device
- Device electronics
  - performs actual operation
  - pixels to screen, bits to/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)

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
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

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.”

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 later

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**

- **new char?**
  - **Polling**
  - **NO**
  - **YES**
- **POLL**
- **LDI**
- **R0, KBSRPtr**
- **BRzp POLL**
- **LDI**
- **R0, KBDRPtr**
- ...  
- **KBSRPtr** .FILL xFE00
- **KBDRPtr** .FILL xFE02

**Simple Implementation: Memory-Mapped Input**

- 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)

**Basic Output Routine**

- **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.

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
```

Interrupt Driven I/O

- What is it?
- Why does it exist?
- Generation of Interrupt Signal
  - Device
  - Priority
  - FSM Mods

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?
Some Questions

- Do you think polling is a good approach for other devices, such as a disk or a network interface?

- Why use LDI/STI for accessing device registers?

Trap Routines/Service calls

- Do you really want programmer to 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

TRAP Mechanism

- Set of Service Routines
- Table of Starting Addresses
- TRAP Instruction
- Linkage
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 x0FF 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

- **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 – 0x0FF
    - 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

TRAP

- PC
- Register File
- Memory
- Instruction Reg
- MAR
- MDR
- RD
- R7
- R[7:0]

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.

Example: Using the TRAP Instruction

- .ORIG x3000
- ...
- ... ; user code
- TRAP x23 ; input character into R0
- ADD R1, R2, R0 ; use R0
- ... ; user code
- ADD R0, R0, R3 ; load output data into R0
- TRAP x21 ; Output to monitor...
- ... ; ... User program...

EXIT TRAP x25 ; halt

Example: Output Service Routine

- .ORIG x0430 ; syscall address
- ...
- ...
- LD R1, SaveR1 ; restore R1 & R7
- LD R7, SaveR7 ; restore R7 & R1
- RET ; back to user

stocked in table, location x21

Example: Using the TRAP Instruction

- .ORIG x3000
- ...
- ... ; user code
- TRAP x23 ; input character into R0
- ADD R1, R2, R0 ; use R0
- ... ; user code
- ADD R0, R0, R3 ; load output data into R0
- TRAP x21 ; Output to monitor...
- ... ; ... User program...

EXIT TRAP x25 ; halt

Example: Output Service Routine

- .ORIG x0430 ; syscall address
- ...
- ...
- LD R1, SaveR1 ; restore R1 & R7
- LD R7, SaveR7 ; restore R7 & R1
- RET ; back to user

stocked in table, location x21
## 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>PUTF</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>

### Example

```assembly
LEA R3, Array
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 #4
```

What's wrong with this routine? What happens to R7?

### Saving & Restoring Registers

- Why should we save a register?
- When should we save a register?
- Who should save the register?

### 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
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.

---

Question

- Can a service routine call another service routine?

- If so, is there anything special the calling service routine must do?

---

What about User Code?

- Service routines provide three main functions:
  1. Shield programmers from system-specific details.
  2. Write frequently-used code just once.
  3. Protect system resources from malicious/clumsy programmers.

- Are there any reasons to provide the same functions for non-system (user) code?

---

Subroutines

- A **subroutine** is a program fragment that:
  - lives in user space
  - performs a well-defined task
  - is invoked (called) by another user program
  - returns control to the calling program when finished

- Like a service routine, but not part of the OS
  - not concerned with protecting hardware resources
  - no special privilege required

- Reasons for subroutines:
  - reuse useful (and debugged!) code without having to keep typing it in
  - divide task among multiple programmers
  - use vendor-supplied library of useful routines
Call/Return Mechanism

| JSR | 0000 | 100 | 1 | 000000 |
| JSRR | 0000 | 000 | 000000 |

JSR Instruction

- Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.
  - saving the return address is called “linking”
  - target address is PC-relative (PC + Sext(IR[10:0]))
  - bit 11 specifies addressing mode
    - if =1, PC-relative: target address = PC + Sext(IR[10:0])
    - if =0, register: target address = contents of register IR[8:6]

JSR 135

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

JSRR Instruction

- Just like JSR, except Register addressing mode.
  - target address is Base Register
  - bit 11 specifies addressing mode
- JSRR R4 ; calls subroutine whose address is in R4
  - R4 should have been loaded with address of subroutine before the JSRR instruction
    - LD R4, example
    - example JFLL x1234

- What important feature does JSRR provide that JSR does not?
Returning from a Subroutine

- RET (JMP R7) gets us back to the calling routine.
  - just like TRAP

TRAP vs JSR(R)

- TRAP
  - Uses trap vector table
  - (Can get to from anywhere)
  - Normally do system functions
  - Written very carefully!
  - Typically tied into some sort of system protection mechanism

- JSR(R)
  - Local (JSR)
  - Anywhere (JSRR)
  - Routine abstraction
  - Code reuse/libraries
  - Written
  - No protection mechanism

Passing Information to/from Subroutines

- Arguments
  - A value passed in to a subroutine is called an argument.
  - This is a value needed by the subroutine to do its job.
  - Examples: in assembly programming, arguments are passed using registers
    - In OUT service routine, R0 is the character to be printed.
    - In PUTS routine, R0 is address of string to be printed.

- Return Values
  - A value passed out of a subroutine is called a return value.
  - This is the value that you called the subroutine to compute.
  - Examples: in assembly, return values are passed using registers
    - In GETC service routine, character read from the keyboard is returned in R0.
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 R8 before TRAP x23 (input character)
  - Or avoid using those registers altogether

- Values are saved by storing them in memory.

Using Subroutines

- In order to use a subroutine, a programmer must know:
  - its address (or at least a label that will be bound to its
    address)
  - its function (what does it do?)
    - NOTE: The programmer does not need to know
      how the subroutine works, but
      what changes are visible in the machine’s state
      after the routine has run.
  - its arguments (where to pass data in, if any)
  - its return values (where to get computed data, if any)

- User code must save registers used to pass arguments
  - If subroutine uses other registers, then save them
    before use and restore before returning.

Dot product of 2 vectors

- Vectors A, B
  - Stored in Arrays array1, array2
- \[ A \cdot B = \sum_{i=1}^{n} A(i) \times B(i) \]

- Need to call Multiplication subroutine

- Load data for Array 1, Array 2

Library Routines

- Vendor may provide object files containing useful subroutines
  - don’t want to provide source code -- intellectual property
  - assembler/linker must support EXTERNAL symbols
    (or starting address of routine must be supplied to user)
  - Using JSRR, because we don’t know whether SQRT
    is within 1024 instructions.
**Linking**

- Libraries provide set of subroutines/functions
  - `Pseudo-op .EXTERNAL` specifies it is an external subroutine
    - Written by someone else/provided by vendor
- Create one executable image at link time
  - Combine multiple modules at link time to produce one executable image
    - Static linking

**Finally, last concept at Machine Level.**

The Stack: An abstract data type (ADT)

- An important abstraction you will encounter in many applications
- Abstract Data Type
  - Defined by behavior not implementation
- Stack
  - Last in/First Out
  - Many ways to implement
  - Many uses in CS
    - Interrupt drive I/O, Saving state of Processor, function calls, etc.
  - Push/Pop