LC-3

Subroutines and Traps

(Textbook Chapter 9)
Subroutines

• Blocks can be encoded as *subroutines*
• A *subroutine* is a program fragment that:
  – lives in user space
  – performs a well-defined task
  – is invoked (called) by a user program
  – returns control to the calling program when finished

• 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
  – requires less memory
In-line vs. Subroutine Execution

Need a mechanism to return to Calling location
JSR

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

PC

Register File

Sext

Instruction Reg

ALU

R7

IR[10:0]
JSRR

- Just like JSR, except Register addressing mode.
  - target address is Base Register
  - bit 11 specifies addressing mode

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

• **RET (JMP R7)** gets us back to the calling routine by taking the PC value saved in R7 by the JSR instruction and putting it back into the PC.
RET

RET is same as JMP R7.
Ex: Subroutine to Negate the value in R0

- 2sComp
  - NOT R0, R0 ; flip bits
  - ADD R0, R0, #1 ; add one
  - RET ; return to caller

To call from a program (within 1024 instructions):

; need to compute R4 = R1 - R3
  - ADD R0, R3, #0 ; copy R3 to R0
  - JSR 2sComp ; negate
  - ADD R4, R1, R0 ; add to R1
  ...

Passing Information to/from Subroutines

• Arguments
  – A value passed in to a subroutine (or trap) is called an argument.
  – This is a value needed by the subroutine to do its job.
  – Examples:
    • In 2sComp routine, R0 is the number to be negated
    • 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 2sComp routine, negated value is returned in R0.
    • In GETC service routine, character read from the keyboard is returned in R0.
Using Subroutines

• In order to use a subroutine, a programmer must know:
  – its address (or the 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
  – its arguments (where to pass data in, if any)
  – its return values (where to get computed data, if any)
  – registers that are changed by Subroutine
Saving and Restoring Registers
(not in text)

• 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
Who saves the registers?

• The 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

• The Calling routine -- “caller-save”
  – Save all registers that will be needed later before the call
  – Restore after return.
  – In special cases, avoid using those registers altogether

• Values are saved by storing them in memory.

• Generally use “callee-save” strategy, except for return values

• Remember to save R7 before any other call (incl. TRAPs!) or you won’t be able to return
Traps

- Sometimes called “System Calls”
- Subroutines called by the TRAP instruction.
- Executed by the Computer’s Operating System.
- Provide services that users are not allowed to modify
  - Input and Output
  - Access to other peoples files
  - Etc.
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 (TRAP instruction has 8 bit trap vector)

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 link back to the user program.
   want execution to resume
   immediately after the TRAP instruction
1. User program executes **TRAP**: load indirect address of TRAP routine code

2. Execute TRAP routine and RETurn to instruction following the **TRAP** in user program
**TRAP**

- **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}`
Return: RET (JMP R7)

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

• We saved old PC in R7.
  – RET 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”.

• Service routine must 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 of TRAP Instructions

```
.ORIG x3000
LD   R2, TERM  ;
LD   R3, ASCII ;
AGAIN
  TRAP  x23  ; IN (R0)
  ADD   R1, R2, R0
  BRz  EXIT
  ADD   R0, R0, R3 ; Change to lowercase
  TRAP  x21  ; OUT (R0) to monitor
  BRnzp AGAIN ;
TERM  .FILL  xFFC9  ; -'7'
ASCII .FILL  x0020  ; lowercase bit
EXIT   TRAP  x25  ; HALT
.END
```
Output Service Routine (explain in I/O section)

```
.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 xFE04
CRTDR .FILL xFE06
SaveR1 .FILL 0
SaveR7 .FILL 0
.END
```

stored 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>PUTC</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>
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.
Example

```
LEA R3, Binry
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 ; -x30
COUNT .FILL #10
Binry .BLKW #10
```

What’s wrong with this routine?
Exercises due Nov. 9th

- Ex 9.2, 9.4, 9.5
- Ex 9.17, 9.18
- **Recommended:** The keyboard registers are supposed to be “privileged” which means that you cannot access them in your user code. Try to write to the screen without using the PUTC or any other TRAP to see if this is true.
Recommended exercises and **Homework** due Feb. 11

• Ex 9.2, 9.5, 9.6, 9.7, 9.13