Chapter 9
TRAP Routines and Subroutines
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
     (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
TRAP

NOTE: PC has already been incremented during instruction fetch stage.
RET (JMP R7)

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

We saved old PC in R7 as part of the TRAP instruction.

- **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

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 difference
                   ; e.g. ‘a’ = ‘A’ + x0020
EXIT   TRAP  x25   ; halt
.END
```
Example: Output Service Routine (TRAP x21)

```assembly
.ORIG x0430 ; syscall address
ST R1, SaveR1 ; Save R1

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

; ----- Return from TRAP
Return LD R1, SaveR1 ; restore R1
RET ; back to user

DSR .FILL xFE04
DDR .FILL xFE06
SaveR1 .BLKW 1
.END
```

stored in table, location x0021
### 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>
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

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

What’s wrong with this routine?
What happens to R7?
Saving and Restoring Registers

Called routine -- "callee-save"
- Before start, save any registers that will be altered (unless altered value is not needed by calling program!)
- Before return, restore those same registers

Calling routine -- "caller-save"
- Save registers destroyed by own instructions (e.g., R7) 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/restored to/from memory by store/load instructions.
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:

• is located 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
JSR Instruction

Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.

- target address is PC-relative \( (PC + \text{Sext}(\text{IR}[10:0])) \)
NOTE: PC has already been incremented during instruction fetch stage.
JSRR Instruction

Just like JSR, except Register addressing mode.

- target address is in Base Register
- bit 11 specifies addressing mode

What important feature does JSRR provide that JSR does not?
NOTE: PC has already been incremented during instruction fetch stage.
Returning from a Subroutine

RET (JMP R7) gets us back to the calling routine.

• just like from TRAP
Example: Negate (2’s complement) the value in R0

\[ 2sComp \quad \text{NOT} \quad R0, \ R0 \quad ; \quad \text{flip bits} \]

\[ \text{ADD} \quad R0, \ R0, \ #1 \quad ; \quad \text{add one} \]

\[ \text{RET} \quad ; \quad \text{return to caller} \]

To call from a program (within 1024 instructions):

\[ \begin{array}{l}
; \quad \text{need to compute } R4 = R1 - R3 \\
\text{ADD} \quad R0, \ R3, \ #0 \quad ; \quad \text{copy } R3 \text{ to } R0 \\
\text{JSR} \quad 2sComp \quad ; \quad \text{negate} \\
\text{ADD} \quad R4, \ R1, \ R0 \quad ; \quad \text{add to } R1 \\
\end{array} \]

\[ \ldots \]

Note: Caller should save R0 if the caller will need it value later!
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 2sComp routine, \(R0\) is the number to be negated
  - 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 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 at least a label that will be bound to its address)
• its function (what does it do?)
  
  Ø NOTE: The programmer does not need to know how the subroutine works, but what changes are visible in the machine’s state through its return values after the routine.
• its arguments (where to pass data in, if any)
• its return values (where to get computed data, if any)
Saving and Restore Registers

Since subroutines are just like service routines, we also need to save and restore registers, if needed.

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

- Save anything that the subroutine will alter internally that shouldn’t be visible when the subroutine returns.
- It’s good practice to restore incoming arguments to their original values (unless overwritten by return value).

**Remember**: You MUST save R7 if you call any other subroutine or service routine (TRAP).

- Otherwise, you won’t be able to return to the caller.
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)

```
.
.EXTERNAL SQRT
.
LD R2, SQAddr ; load SQRT addr
JSRR R2
.
SQAddr .FILL SQRT
```

Using JSRR, because we don’t know whether SQRT is within 1024 instructions.
Mechanisms to get attention from the operating system (OS)
  • System calls (via TRAP instruction)
  • Interrupts
  • Exceptions

Subroutines
  • invoked via JSR or JSRR instructions (return address in R7)
  • arguments
  • return values
  • need to save/restore registers