Chapter 9

Subroutines and TRAPs

- *Privileged Instructions*
- *TRAP Routines*
- *Subroutines*

Privileged Instructions

- There are several instructions that are best executed by a *supervisor* program (OS) rather than a *user* program:
  - I/O instructions
  - Interacting with system/device (memory-mapped) registers
  - Resetting the clock
  - Halt

  i.e. instructions where one program can affect the behavior of another.
- Most modern CPUs are designed to enforce at least two modes of operation:
  - User Mode
  - Privileged Mode (aka. supervisor, kernel, monitor mode)
- Only the supervisor program (OS) can execute privileged instructions.
- But, how do we ALLOW user programs to access privileged functionality?
- There are two issues to address: *Policy* and *Mechanism*

TRAP Instructions

- TRAPs insulate critical tasks from the user
  - with or without privilege enforcement
- The TRAP mechanism:
  - A set of trap service routines or TSRs (part of the CPU OS)
    - We have already seen the basic I/O SRs
  - A table of the starting addresses of these service routines
    - Located in a pre-defined block of memory …
    - … called the Trap Vector Table or System Control Block
    - In the LC-3: from x0000 to x00FF (only 5 currently in use)
  - The TRAP instruction
    - which loads the starting address of the TSR into the PC
  - Return link
    - from the end of the TSR back to the original program.

LC-3 TRAP Routines

- GETC (TRAP x20)
  - Read a single character from KBD.
  - Write ASCII code to R0[7:0], clear R0[15:8].
- OUT (TRAP x21)
  - Write R0[7:0] to the monitor.
- PUTS (TRAP x22)
  - Write a string to monitor (address of first character of string is in R0).
- IN (TRAP x23)
  - Print a prompt to the monitor and read a single character from KBD.
  - Write ASCII code to R0[7:0], clear R0[15:8], echo character to the monitor.
- HALT (TRAP x25)
  - Print message to monitor & halt execution.
- PUTSP (TRAP x24)
  - Print packed string to monitor (address in R0)

⋮	⋮
x0020	x0400
x0021	x0430
x0022	x0450
x0023	x04A0
x0024	x04E0
x0025	xFD70
⋮	⋮

Trap vector table

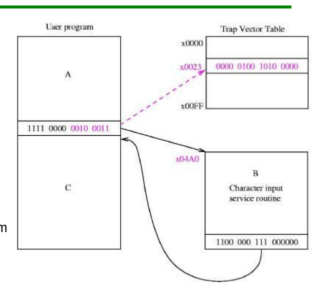
TRAP Instructions

15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1	0 0 0 0	trapvector8

- TRAP: A special instruction
  - A form of subroutine call used to invoke a service routine.
  - If privilege is being enforced, it switches the execution to *privileged* mode, and reverts back to *user* mode when the TSR completes.
    - R7 ← (PC) ; the current PC is stored in R7
    - PC ← Mem[Zext(IR[7:0])] ; the 8-bit trap vector is loaded to the PC
- RET – return instruction
  - The TSR ends with the RET instruction
    - PC ← (R7) ; the program now picks up where it left off

TRAP Example

- Trap Vector Table
  - Or System Control BLock
  - In LC-3
    - 8 bits specify one of 256 locations (x0000 to x00FF)
    - The location contains the address of the TRAP service routine.
- TRAP & Interrupts
  - Similar mechanisms
  - A TRAP is an instruction (event internal to a program).
  - An interrupt is external to a program (from an I/O device)
  - Both invoke a supervisor service routine.



Character Output TSR (OUT)

```
01              .ORIG   X0430        ; System call starting address
02              ST      R1, SaveR1   ; R1 will be used for polling
03
04      ; Write the character
05      TryWrite LDI    R1, DSR      ; Get status
06              BRzp    TryWrite     ; bit 15 = 1 => display ready
07      WriteIt STI     R0, DDR      ; Write character in R0
08
09      ; Return from TRAP
0A      Return  LD      R1, SaveR1   ; Restore registers
0B              RET                  ; Return (actually JMP R7)
0C      DSR     .FILL   xFE04        ; display status register
0D      DDR     .FILL   xFE06        ; display data register
0E      SaveR1  .BLKW   1
0F              .END

ALSO
01              .ORIG   x0021
02              .FILL   x0430
```


HALT TSR

- Clears the RUN latch MCR[15]:

```
01              .ORIG   XFD70            ; System call starting address
02              ST      R0, SaveR0       ; Saves registers affected
03              ST      R1, SaveR1       ; by routine
04              ST      R7, SaveR7       ;
05
06      ; Print message that machine is halting
07              LD      R0, ASCIINewLine
08              TRAP    x21              ; Set cursor to new line
09              LEA     R0, Message      ; Get start of message
0A              TRAP    x22              ; and write it to monitor
0B              LD      R0, ASCIINewLine
0C              TRAP    x21
0D
0E      ; Clear MCR[15] to stop the clock
0F              LDI     R1, MCR          ; Load MC register to R1
10              LD      R0, MASK         ; MASK = x7FFF (i.e. bit 15 = 0)
11              AND     R0, R1, R0       ; Clear bit 15 of copy of MCR
12              STI     R0, MCR          ; and load it back to MCR
```


HALT TSR (cont.)

```
13      ; Return from the HALT routine
14      ; (how can this ever happen, if the clock is stopped on line 12??)
15      ;
16                      LD      R7, SaveR7   ; Restores registers
17                      LD      R1, SaveR1   ; before returning
18                      LD      R0, SaveR0
19                      RET                  ; JMP R7
1A
1B      ; constants
1C      ASCIINewLine    .FILL   x000A
1D      SaveR0          .BLKW   1
1E      SaveR1          .BLKW   1
1F      SaveR7          .BLKW   1
20      Message         .STRINGZ    "Halting the machine"
21      MCR             .FILL   xFFFE
22      MASK            .FILL   x7FFF
23                      .END
```


Saving & restoring registers

- Protect your values! High-level "Scope" rules don't apply at low-level
  - Any routine may change values currently stored in any register/memory.
- Caller Save
  - Sometimes the calling program ("caller") knows what needs to be protected, so it saves the endangered register before calling the subroutine.
    - e.g. in the HALT routine, which has itself been called by another program, the caller knows that it has precious cargo in R7, which will be overwritten by the TRAP instructions (why??), so it saves R7 to memory at the start of the routine, and restores it from memory before returning to the main program.
- Callee save
  - Other times it will be the called program ("callee") that knows what registers it will be using to carry out its task.
    - again in the HALT routine, R0 and R1 are used as temporary working space to hold addresses, masks, ASCII values, etc., so they are both saved to memory at the start of the routine, and restored from memory before returning to the main program.

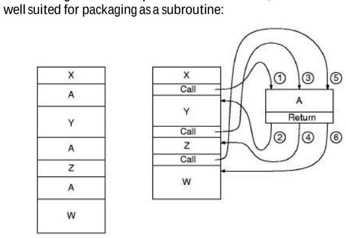
Subroutines

- Used for
  - Frequently executed code segments
  - Library routines
  - Team-developed systems
    - in other words, all the same reasons for using subroutines in higher level languages, where they may be called functions, procedures, methods, etc.
- Requirements:
  - Pass parameters and return values, via registers or memory.
  - Call from any point & return control to the same point.

  - First, we'll pass values via registers. (Easy, but many limitations)
  - Later, we'll pass values via memory. (Uses memory as a "stack", powerful!)

The Call / Return mechanism

- The figure illustrates the execution of a program comprising code fragments A, W, X, Y and Z.
  - Note that fragment A is repeated several times, and so is well suited for packaging as a subroutine:



(a) Without subroutines (b) With subroutines

Jump to Subroutine : JSR/JSRR

- A = IR[11] specifies the addressing mode

15 14 13 12	11	10 9 8 7 6 5 4 3 2 1 0
0 1 0 0	1	Address eval. bits
JSR(R)	A	

- JSR: jump to subroutine (PC-Relative), IR[11] = 1
  - R7 ← (PC) i.e. PC is saved to R7
  - PC ← (PC) + Sext(IR[10:0]) i.e PC-Relative addressing,
  - using 11 bits => label can be within +1024 / -1023 lines of JSR instruction

15 14 13 12	11	10 9 8	7 6	5 4 3 2 1 0
0 1 0 0	0		BaseR	0 0 0 0 0 0
JSR(R)	A			

- JSRR: jump to subroutine (relative base+offset), IR[11] = 0:
  - R7 ← (PC) i.e. PC is saved to R7
  - PC ← (BaseR) i.e Base+Offset addressing, with offset = 0

Subroutine call example

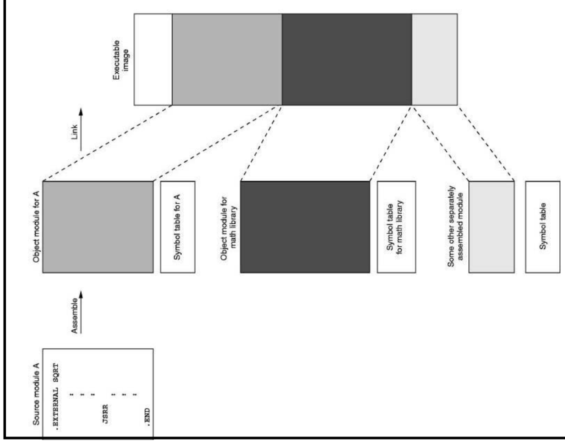
```
; Calling program
        .ORIG x3000
        LD      R1, num1
        LD      R2, num2
        JSR     multi
        ST      R3, prod
        HALT
;
; Input data & result
num1    .FILL   x0006
num2    .FILL   x0003
prod    .BLKW   1
```

```
; Subroutine multi
; Multiply 2 positive numbers
; Parameters:
; In: R1, R2;   Out: R3
;
multi   AND     R3, R3, #0
        ADD     R4, R1, #0
        BRz     zero
loop    ADD     R3, R2, R3
        ADD     R1, R1, # -1
        BRp     loop
zero    RET
        .END
```

Notice any undesirable side-effects?

Library Routines

- Library
  - A set of routines for a specific domain application.
  - Example: math, graphics, GUI, etc.
  - Defined outside a program.
- Library routine invocation
  - Labels for the routines are defined as *external*. In LC-3:
    .External Label
  - Each library routine contains its own symbol table.
  - A linker resolves the external addresses before creating the executable image.



Practice Problems

- 9.2, 9.7, 9.10, 9.13, 9.15