LC-3 Subroutines

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Subroutines are the most basic way to share executable code.

B. Outcomes

After this lecture, you should

- Understand how simple subroutines can be defined and used.

C. Simple User-Written Subroutines

- The LC-3 uses the `JSR` and `JSRR` commands to Jump to a SubRoutine.
- Both instructions set `R7` ← `PC` before the jump so that the subroutine knows where to return to.
- `JSR` uses an 11-bit `PC` offset to find the address of the subroutine to go to:
  - `R7` ← `PC`; `PC` ← `PC` + `Sext(PCoffset11)`
- `JSRR` uses a base register to specify where to go to.
  - *target* ← $R[Base]$; `R7` ← `PC`; `PC` ← *target*
  - (A subtle issue: When the base register is `R7`, we need the temporary variable to correctly swap `PC` and `R7`. If the base register is not `R7`, then the semantics above is equivalent to `R7` ← `PC`; `PC` ← $R[Base]$.)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSR	0	1	0	0	1	PCoffset11										

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

- To return from a subroutine call, use **JMP R7** (jump with **R7** as the base register). The assembler also allows **RET** as a substitute mnemonic.

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

D. Framework For a Simple Subroutine

- Write comments that specify what registers should contain parameters, which ones will contain results, and which ones get modified but not restored.
- Begin by saving the registers you will modify and restore. These can include
  - Registers you're using for intermediate calculations.
  - `R0`, if you're going to use any of the I/O traps (**GETC** and **IN** read a character into `R0`; **OUT** prints `R0`, and **PUTS** requires a pointer in `R0`).
  - `R7`, if you're going to call any other subroutines or a **TRAP**. (Executing **JSR**, **JSRR**, or **TRAP** will cause the value of `R7` you need to return with to be overwritten.)
  - The easiest way to save registers is to store them into some variables set aside for that purpose. (This doesn't work for recursive subroutines.)
  - Alternatively, you can have semantics like "This routine may change `R0`" — then the onus of saving/restoring is on the user.
- Before you return from the subroutine, restore the registers you saved.
  - Then return using **RET** or **JMP R7** (they're equivalent).
- Note that unless you save and restore `R7`, the **JMP R7** will go to the instruction after the most recent **JSR**, **JSRR**, or **TRAP** in your subroutine. (If you're lucky, this will cause some sort of obvious problem — a bad calculation or an infinite loop.)

E. Example: Reading a String

- The `readstring` program from the previous lecture can be made into a subroutine fairly easily.

- It used `R0`, `R1`, `R2`, and `R3`, so we want to start by saving those registers in addition to `R7`. Instead of halting, we need to restore the registers and `JMP R7` to return to the caller.
- We also need to establish a protocol for calling the routine — should the subroutine contain the buffer to read into? Or should the caller pass the address of the buffer? If it passes the address, how should we do this?
- Let's say we decide to have the caller pass the address of the buffer, in `R0`. (`R0` seems reasonable because it's similar to how the I/O traps `GETC`, `IN`, `OUT`, and `PUTS` work.)
- The body of `readstring` is almost exactly the same as in the `readstring.asm` program; the only real difference is that to get the buffer address, we use `R0` instead of `LEA …, buffer`. In addition, all the labels have been modified to begin with `RS`. This is just a mnemonic to indicate they're part of the `readstring` routine.

```
; readSubroutine.asm
;
; The main program exercises the readline subroutine.
;
          .ORIG     x3000
           LEA      R0, string1   ; read first message
           JSR      readstring

           LEA      R0, string2   ; read second message
           JSR      readstring

           HALT

string1   .BLKW     100
string2   .BLKW     100


; readstring: Reads a return-terminated string into a
; buffer pointed to by R0.  Uses the same pseudocde as in
; readstring.asm
;
; This routine assumes R0 points to the buffer into which
; to read the string.  It saves and restores all registers.

; Save internally used registers:
```

```
;     R0 = GETC/OUT char, R1 = buffer_posn,
;     R2 = -(return char), R3 = temporary
readstring ST       R0, RSsave0   ; Save R0
           ST       R1, RSsave1   ; Save R1
           ST       R2, RSsave2   ; Save R2
           ST       R3, RSsave3   ; Save R3
           ST       R7, RSsave7   ; Save R7

           ADD      R1, R0, 0     ; buffer_posn = &buffer
           LEA      R0, RSmsg     ; get prompt message
           PUTS                   ; prompt for input
           GETC                   ; read char into R0
           LD       R2, RS_rc     ; R2 = return char
           NOT      R2, R2        ; R2 = -(return char) - 1
           ADD      R2, R2, 1     ; R2 = -(return char)
           ADD      R3, R0, R2    ; calculate R0 - return char
RSLoop     BRZ      RSDone        ; until char read = return
           OUT                    ; print char read in
           STR      R0, R1, 0     ; *buffer_posn = char read in
           ADD      R1, R1, 1     ; buffer_posn++
           GETC                   ; read next char
           ADD      R3, R0, R2    ; calc char - return char
           BR       RSLoop        ; continue loop
RSDone     OUT                    ; print the return char in R0
           AND      R3, R3, 0     ; get a null char ('\0')
           STR      R3, R1, 0     ; terminate buffer string
           LD       R0, RSsave0   ; point to bufer
           PUTS                   ; print the string we read in
           LD       R0, RS_rc     ; get a newline
           OUT                    ; end this line of output

; Restore registers and return
           LD       R7, RSsave7   ; Restore R7
           LD       R3, RSsave3   ; Restore R3
           LD       R2, RSsave2   ; Restore R2
           LD       R1, RSsave1   ; Restore R1
           LD       R0, RSsave0   ; Restore R0
           JMP      R7

RS_rc     .FILL     x0A           ; ASCII newline char
RSmsg     .STRINGZ  "Enter chars (then return): "

; Save area for registers
RSsave0   .BLKW     1             ; Save area for R0
RSsave1   .BLKW     1             ; Save area for R1
RSsave2   .BLKW     1             ; Save area for R2
```

```
RSsave3   .BLKW    1              ; Save area for R3
RSsave7   .BLKW    1              ; Save area for R7
          .END
```

- The `JSR` instruction can only access subroutines within an 11-bit `PC` offset. If the subroutine might be further away than that, you can load the address of the subroutine into a register and use `JSRR`:

```
; Alternate version for long jump to subroutine: (uses R3
; as a temporary variable).
           LD       R3, RSptr     ; Point to readstring subr
           JSRR     R3            ; Call readstring
            ...
RSptr     .FILL     readstring    ; &readstring subroutine
```

LC-3 Subroutines

CS 350: Computer Organization & Assembly Language Programming

A. Why?

- Subroutines are the most basic way to share executable code.

B. Outcomes

After this activity, you should be able to

- Understand how simple subroutines can be defined and used.

C. Questions

1. Why do we have the called subroutine save/restore registers, not the calling routine?
2. In general, what behavior do you get if you call a subroutine that doesn't save and restore `R7` before calling a `TRAP` or subroutine?
3. Given the save/restore register technique we used in the sample programs, what happens if you try to write a recursive subroutine?
4. Sketch the code for a simple subroutine that assumes `R0` points to a string, finds the length of the string, and returns the length in `R1`.

Solution

1. The caller might not know which registers need to be saved/restored. More importantly, the save/restore code would have to be written with each call instead of being written just once in the called routine. This would take extra space.
2. When the subroutine tries to return to the caller (via `JMP R7`), it will jump to the instruction after the most recently executed `JSR` or `TRAP` instead of back to the caller's code. An infinite loop can happen.
3. Since the register save areas will be rewritten with each recursive call, each return (whether from a base case or recursive case) will return to exactly the same instruction, with exactly the same registers. An infinite loop is again certainly possible.

