User Subroutines and the Runtime Stack

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Subroutines are the most basic way to share executable code.
- Information for a procedure call is stored in an activation frame. At runtime, the activation frames form a stack (in C and C++) or heap (in Java).

B. Outcomes

After this lecture, you should

- Understand how simple subroutines can be defined and used.
- What activation frames look like and how the runtime stack is used to save procedure call information.

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: Multiply Subroutine

- Our earlier multiplication program can be easily made into a subroutine. The first question to ask is the protocol for calling the routine. Since we're passing and returning numbers, it seems natural to use registers for that purpose.
- In `multiply_subr.asm`, the main program calls a `Mult` subroutine to set `Z` = `X * Y` by first setting `R1` = `X`, `R2` = `Y`. Then it calls `Mult` to set `R1` = `R1 * R2` and then stores `R1` into `Z`.

```
; multiply_subr.asm -- Multiplication as a subroutine
; ...
; Main program: Calculate X * Y, store result in product
;
        .ORIG    x3050
         LD      R1, X
         LD      R2, Y
         JSR     Mult           ; R1 = R1 * R2
         ST      R1, Z
         HALT

X       .FILL    12
Y       .FILL    8
Z       .BLKW    1              ; Holds X*Y at end
```

- It's important to comment the subroutine call interface.

```
; Multiply subroutine
; Set R1 = R1 * R2 (if R2 >= 0); set R1 = 0 if Y < 0.
;
; Property:
;    Y  < 0 implies product = 0, k = Y
;    Y >= 0 implies product = X * (Y-k) and 0 <= k <= Y
;        (So when k = 0, product = X * Y)
;
; Register usage: R1 = X, R2 = Y; R3 = product, R4 = k
;
```

- The subroutine itself is defined using the framework already described, so it begins by saving the registers. Note all the labels for the routine begin with `MU`, just to keep them separate from the labels from other future routines we might add.

```
Mult     ST      R7, MU_R7      ; Save registers
         ST      R4, MU_R4
         ST      R3, MU_R3
```

- Then we initialize the product and counter and fall into the loop.

```
; Establish product and k properties
         AND     R3, R3, 0       ; product = 0
         ADD     R4, R2, 0       ; counter k = Y

; If Y < 0, product = 0; otherwise, maintain
; product = X * (Y - k) while k = Y, Y-1, ....
; Stop when k = 0
;
MU_Loop  BRNZ    MU_Done         ; until k <= 0
         ADD     R3, R3, R1      ;   product += X
         ADD     R4, R4, -1      ;   k--
         BR      MU_Loop
```

- After the loop, we set the result register, restore the other registers, and return. Note we didn't actually change `R7`, so we didn't really need to save it, but it's easy to imagine a future change that might call a `TRAP` (to print an error message, for example.)

```
; After loop, R3 = X * Y (if Y >= 0) or 0 (if Y < 0)
; Set R1 to result, restore registers, and return
;
MU_Done  ADD     R1, R3, 0       ; R1 = X * Y
         LD      R3, MU_R3       ; Restore registers
         LD      R4, MU_R4
         LD      R7, MU_R7
         JMP     R7              ; return to caller

; Save area for registers
;
MU_R1   .BLKW   1
MU_R3   .BLKW   1
MU_R4   .BLKW   1
MU_R7   .BLKW   1
```

F. Example: Reading a String

- The `readstring` program from the previous lecture can also 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?
- Since the I/O traps `GETC`, `IN`, `OUT`, and `PUTS` use R0, it seems reasonable to have `ReadLine` to use `R0`; the user should set `R0` to point to the buffer before calling `ReadLine`. Aside from saving and restoring the registers and renaming some labels, the rest of `ReadLine` is similar to the earlier version.
- The main program calls `ReadLine` twice. The first call shows how to call a close-by routine: It uses `LEA` to get the address of the routine and `JSR` to call it. The second call loads a pointer to `ReadLine` and uses `JSRR` to call it; this will work whether `ReadLine` is close by or far away from the subroutine call.

```
; read_subr.asm
;
; The main program exercises the ReadLine subroutine.
;
            .ORIG   x3000
             LEA    R0, msg1        ; read first message
             JSR    ReadLine

             LD     R0, addr_msg2   ; read second message
             LD     R7, addr_read   ;
             JSRR   R7
             HALT

; Buffers to read lines into
;
msg1        .BLKW   100
msg2        .BLKW   100

; Pointers &ReadLine and &msg2 to illustrate JSRR
;
addr_read   .FILL   ReadLine
addr_msg2   .FILL   msg2
```

- The `ReadLine` routine begins with usage comments and register saving.

```
; ReadLine: Read a return-terminated string into a
; buffer pointed to by R0.  Uses the same pseudocode as in
; readstring.asm.  Registers are restored before return.
;
; Register usage:
;    R0 = GETC/OUT char,  char *R1 = bp (buffer position),
;    R2 = -(return char), R3 = temporary
;

ReadLine     ST     R0, RS_R0    ; Save R0
             ST     R1, RS_R1    ; Save R1
             ST     R2, RS_R2    ; Save R2
             ST     R3, RS_R3    ; Save R3
             ST     R7, RS_R7    ; Save R7
```

- Then we do some initialization and prompt for and read the first character

```
; Calculate R2 = -(ASCII char for return)
             LD     R2, RS_rc    ; R2 = return char
             NOT    R2, R2       ; R2 = -(return char) - 1
             ADD    R2, R2, 1    ; R2 = -(return char)

; Initialize bp = &buffer, prompt user for string,
; and read first char
;
             ADD    R1, R0, 0    ; bp = &buffer
             LEA    R0, RS_msg   ; get prompt message
             PUTS                ; prompt for input
             GETC                ; read char into R0
             ADD    R3, R0, R2   ; calculate R0 - return char
```

- The loop repeatedly echoes / stores / reads a character until we see `'\n'`. Then we end the string with a `'\0'` and print it.

```
; Repeat printing the char, storing it, and reading the next
; char until we read in a return char
;
RS_Loop      BRZ    RS_Done      ; until char read = return
             OUT                 ;     print char read in
             STR    R0, R1, 0    ;     *bp = char read in
             ADD    R1, R1, 1    ;     bp++
             GETC                ;     read next char
             ADD    R3, R0, R2   ;     calc char - return char
             BR     RS_Loop      ; continue loop
```

```
; When we see the end of the line, terminate the string
; in the buffer and print it out
;
RS_Done      OUT                ; echo the return char in R0
             AND    R3, R3, 0   ; get a null char ('\0')
             STR    R3, R1, 0   ; terminate buffer string
             LD     R0, RS_R0   ; point to buffer
             PUTS               ; print the string we read in
             LD     R0, RS_rc   ; get a newline
             OUT                ; end this line of output
```

- In addition to restoring the registers and returning, we have to declare some constants and a save area for the registers.

```
; Restore the registers and return
;
             LD     R7, RS_R7
             LD     R3, RS_R3
             LD     R2, RS_R2
             LD     R1, RS_R1
             LD     R0, RS_R0
             JMP    R7

RS_rc       .STRINGZ "\n"         ; ASCII newline char
RS_msg      .STRINGZ "Enter chars (then return): "

; Save area for registers
RS_R0        .BLKW   1
RS_R1        .BLKW   1
RS_R2        .BLKW   1
RS_R3        .BLKW   1
RS_R7        .BLKW   1
```

G. Recursive Routines and the Runtime Stack

- We know that saving call information with the code for the called routine doesn't work for recursive routines because the saved information would be overwritten by the next call.

- Let's study the following pseudocode for factorial(n), annotated with positions A – E

fact(*n*):

    (A) **if** $n < 1$ **then**
            returned value = 1 (B)
        else
            call *fact*(*n*-1);
            (C) set *temp* = callee's *returned value*
            (D) our *returned value* = *temp* * *n*
            (E) **return**

- Each call of factorial gets its own **activation record** (or "frame") that includes
    - local variable *temp* (set and used by callee)
    - local variables to save registers (set and used by callee)
    - return address (saved and used by caller)
    - returned value (value set by callee, used by caller)
    - parameter *n* (value set by caller)
- The **Runtime Stack** holds the frames; each call pushes a frame onto the stack; the top frame is for the active call; returning pops the frame from the stack.
    - The stack is usually implemented as a linked list, so each frame also includes a "dynamic link" pointer to the previous stack element.
    - In C/C++, on return, we deallocate the space for the local variables of the called routine. That's why in these languages a routine should not return a pointer to a local variables.
    - In Java, the space is not deallocated and the garbage collector will recover the space if necessary.
- Illustration: Call *fact*(3), show state at (B) of base case call *fact*(0)
    - (Note: Not bothering to show register save areas Also not showing link to top frame of stack.)

for fact(3)	for fact(2)	for fact(1)	for fact(0)
temp	temp	temp	temp
Dynamic link to frame for `main`	d.l. to frame for fact(3)	d.l. to frame for fact(2)	d.l. to frame for fact(1)
ret addr to (C)	ret addr to (C)	ret addr to (C)	ret addr to (C)
ret val	ret val	ret val	ret val = 1
n = 3	n = 2	n = 1	n = 0
Paused at C	Paused at C	Paused at C	Active at B

for fact(3)	for fact(2)	for fact(1)	for fact(0)
temp	temp	temp	
d.l. to frame for `main`	d.l. to frame for fact(3)	d.l. to frame for fact(2)	
ret addr to (C)	ret addr to (C)	ret addr to (C)	
ret val	ret val	ret val	ret val = 1
n = 3	n = 2	n = 1	n = 0
Paused at C	Paused at C	Active at C	Returned

for fact(3)	for fact(2)	for fact(1)
temp	temp	temp = 1
d.l. to frame for `main`	d.l. to frame for fact(3)	d.l. to frame for fact(2)
ret addr to (C)	ret addr to (C)	ret addr to (C)
ret val	ret val	ret val = 1
n = 3	n = 2	n = 1
Paused at C	Paused at C	Active at E

for fact(3)	for fact(2)
temp	temp = 1
d.l. to frame for `main`	d.l. to frame for fact(3)
ret addr to main pgm call	ret addr to (C)
ret val	ret val = 2
n = 3	n = 2
Paused at C	Active at E

for fact(3)
temp = 2
d.l. to frame for `main`
ret addr to (C)
ret val = 6
n = 3
Active at E

- In the example above, all the frames were for the same routine, so the frames were the same size.
- More generally, the size of a stack frame will depend on how many parameters and local variables the called routine has.

General Method for Routine P calling Routine Q

- The caller *P* sets up the parameters of *Q* by calculating the argument values; then adds them to the activation we're building for Q.
- Then *P* does a `JSR` to the routine being called (saving `R7` ← `PC` to establish the address to return to).
- The callee *Q* sets up the rest of the activation record; *Q* has to allocate space for

- The return value
- The return address (and set it to `R7`)
- The dynamic link (the pointer to the old top of the runtime stack)
- Any desired local variables, including for register saves

- We reset the top-of-runtime-stack pointer to the activation frame we just built.
- Then we jump to the code for the body of *Q* and start executing it.

Returning from Q to P (in C and C++)

- For *Q* to return to *P*, we have to pop the activation record for its call. *Q* can:
  - Deallocate space for its local variables.
  - Get the dynamic link so we can reset the top of the runtime stack (and deallocate the space for the dynamic link).
  - Get the return address and restore `R7` (and deallocate the space for the return address).
  - Copy the value being returned by *Q* (if there is one) into its designated space.
  - Use R7 to jump back to *P*.
- *P* can get the returned value (if there is one) and deallocate the space for it.
- *P* also has to deallocate the space for the arguments passed to *Q*.

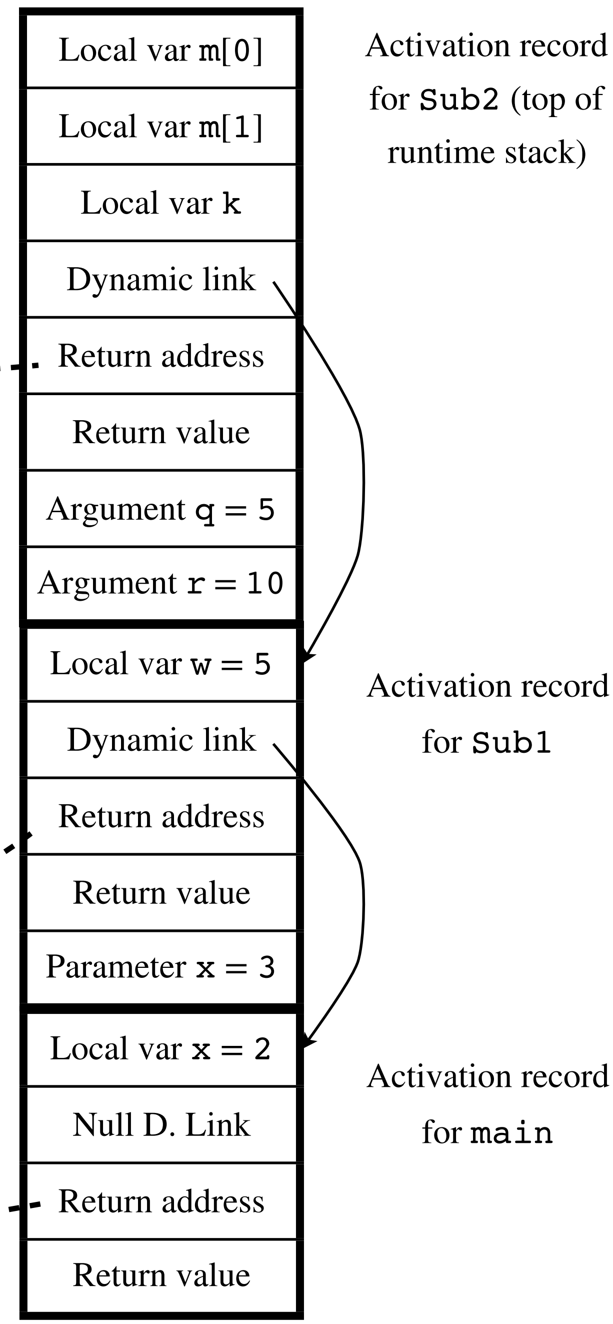
H. Sample of Nested Call

- In the example below, the `main` program calls `Sub1` which calls `Sub2`:

```
main() {
  int x = 2;
  Sub1(x+1);
}

int Sub1(int x) {
  int w = x+2;
  w = Sub2(w,10);
  return w;
}

int Sub2(int q, int r) {
  int k; int m[2];
  // *** Here ***
  return k;
}
```

- Note that since the `x` of the `main` program and the `x` of `Sub1` are in different routines, their values must be stored in different places.
  - (Note the runtime stack below is drawn vertically.)
- At the point marked “`Here`” in `Sub2`, the activation stack looks like this:

```
main() {
  int x = 2;
  Sub1(x+1);
}

int Sub1(int x) {
  int w = x+2;
  w = Sub2(w,10);
  return w;
}

int Sub2(int q, int r) {
  int k; int m[2];
  // *** here ***
  return k;
}
```



LC-3 Subroutines and the Runtime Stack

CS 350: Computer Organization & Assembly Language Programming

A. Why?

- Subroutines are the most basic way to share executable code.
- In operating systems, stacks can help track nested subroutine calls.

A. Outcomes

After this activity, you should be able to

- Understand how simple subroutines can be defined and used.
- Sketch the runtime stack at various points during the execution of a program.

B. Problems

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`.
5. Suppose in the `Sub1`/`Sub2` example, `Sub2` returns `8`. Sketch the runtime stack after `Sub2` returns to `Sub1` and `Sub1` assigns `w = Sub2(...)`.
6. Continuing from Question 1, sketch the runtime stack just after `Sub1` returns to the `main` program.
7. If a subroutine wants to pass an array instead of an integer, then in C-like languages, we just copy the address of the initial element. What would the pros and cons be of copying the entire array onto the stack instead?

8. If routine P calls routine Q, which of the following would you find or not find in the activation record for the call?

   (a) The return address to the code for P.

   (b) The return address from P back to its caller.

   (c) A link to the activation record for P.

   (d) A link to the activation record for Q.

   (e) Space for the local variables of P.

   (f) Space for the local variables of Q.

   (g) Space for the arguments being passed to Q.

   (h) Space for the arguments P received from its caller.

   (i) Space for the value that Q will return to P.

   (j) Space for the value that P will return to its caller.

