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.

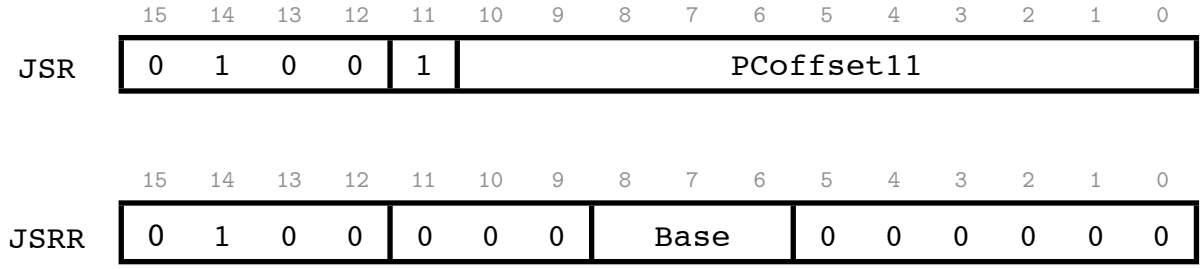
B. Outcomes

After this lecture, you should know

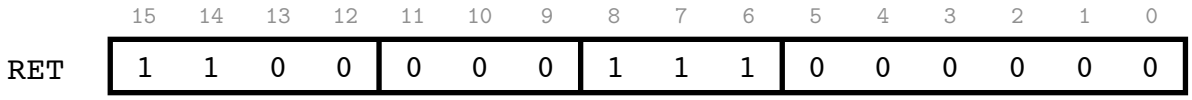
- How to write and use a non-recursive subroutine using the return address save/restore pattern.
- What the runtime stack is and how to use one to save procedure call information (arguments, caller information, results).

C. Simple Subroutines

- The LC-3 uses the `JSR` and `JSRR` commands to `J`ump to a `S`ub`R`outine.
- 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` + *offset*
- `JSRR` uses a base register to specify where to go to. A subtle issue: If `R7` is the base register, we need a temporary variable to swap `R7` and `PC` correctly.
  - *target* ← *R*[*Base*]; `R7` ← `PC`; `PC` ← *target*



- To return from a subroutine call, use `JMP` with `R7` as the base register.



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 traps. (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.)
- Before you return from the subroutine, restore the registers you saved.
  - Then `RET` using `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.)
- You can also have semantics like "This routine may change `R0`" — then the onus of saving/restoring is on the user.

E. Example: Printing a Newline

- It might seem like overkill to write a subroutine that just prints a newline, but it's a nice small example.


```
; newline.asm
;
; The main program exercises the NewLine subroutine to make
; sure it actually prints out a newline.
;
          .ORIG     x3000
           LEA      R0, string1   ; print first message
           PUTS
           JSR      NewLine       ; print newline

           LEA      R0, string2   ; print second message
           PUTS
           JSR      NewLine       ; print newline
           PUTS                   ; reprint second message
           JSR      NewLine       ; print newline
           HALT

string1   .STRINGZ  "Hello,"
string2   .STRINGZ  "world"
```

- Here's the code for the subroutine. Note that the labels for the subroutine all begin with `NL` — it's useful to give them a prefix that's unique to the routine so that you don't get them confused with the labels for some other routine.

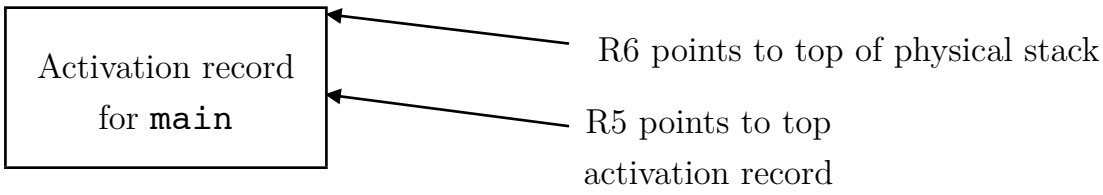
```
; NewLine: subroutine to print a newline character.
; Uses/restores R0.
;
NewLine    ST       R0, NLsave0  ; Save R0
           ST       R7, NLsave7  ; Save R7
           LD       R0, NLchar   ; Get newline into R0
           OUT                   ; and print it
           LD       R7, NLsave7  ; Restore R7
           LD       R0, NLsave0  ; Restore R0
           RET                   ; ... and return

NLchar     .FILL    x0A          ; ASCII newline char

; Save area for registers
NLsave0    .BLKW    1            ; Save area for R0
NLsave7    .BLKW    1            ; Save area for R7
           .END
```

F. Subroutine Calls in C and C++

- Higher-level languages support recursive subroutines, and the simple save/ restore technique for variables doesn't work for them.

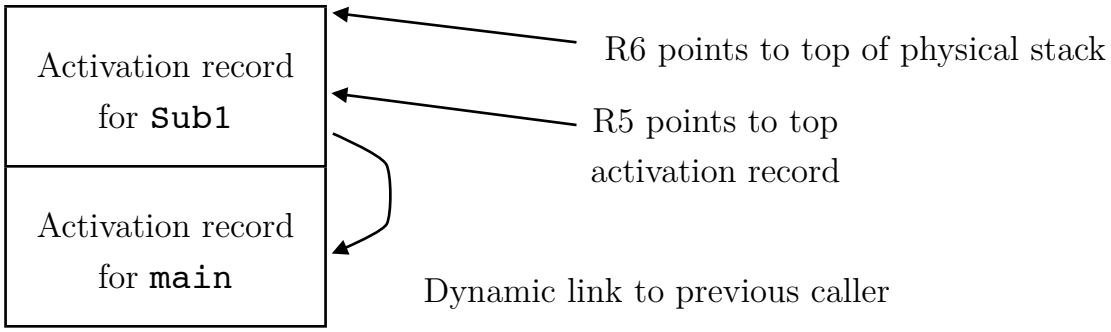
- In C and C++, the solution is to use the **runtime stack**. The elements of the runtime stack are called **activation records** or **activation frames**; each one holds the information we need to execute a call (what we're passing, what we need returned, where to return to, and the local variables and parameters of the routine we're calling).
  - If `main` calls *subroutine 1* calls *subroutine 2*, … calls *subroutine n*, then the runtime stack will contain an entry for each of these routines (`main`, *subroutine 1*, etc.) with `main` being the oldest.
  - The topmost frame is for the active call. When we finish with it, we go back to the most-recently paused call; this is why we need a stack (last in, first out).
  - It's the use of a stack that keeps us from returning pointers to local variables: When we return from the subroutine, the memory space we're pointing to will be deallocated and sent back to the runtime system for re-use.
- The structure of the runtime stack used by the textbook is fairly complicated.
  - The size of an activation record can vary; it depends on how many and what kinds of local variables and parameters the routine being called has.
  - To implement such a stack, a linked list is a natural structure. In the textbook's implementation of the runtime stack, `R5` points to the top activation record, and each record points to its predecessor.
  - To implement this linked list stack, the textbook uses a word-based stack. If the activation record is N words long, then the record is created by doing N one-word pushes onto this underlying physical stack. (And it's removed by doing N one-word pops.) The textbook uses `R6` to point to the top of this physical stack.
  - You might think that `R5` = `R6`, but that's not the case because the address of an activation record isn't at its physical beginning; it's in the middle. (Yes, it sounds weird; we'll look at this more later.)
- **Example**:

  (1) OS calls `main` (OS initializes stack & pushes record for call of `main`).



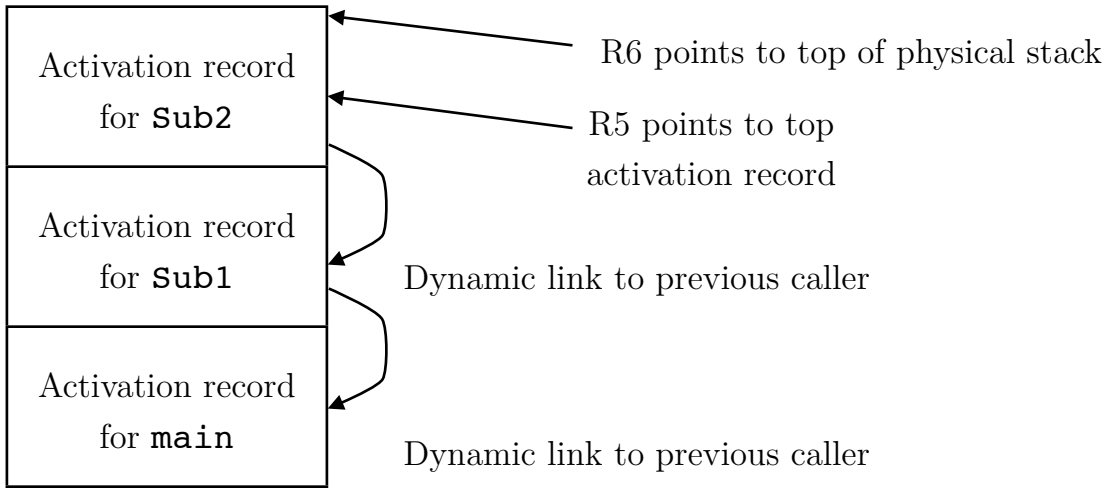
Runtime Stack (OS calls `main`)

(2) `main` calls `Sub1` (push record for call of `Sub1`)



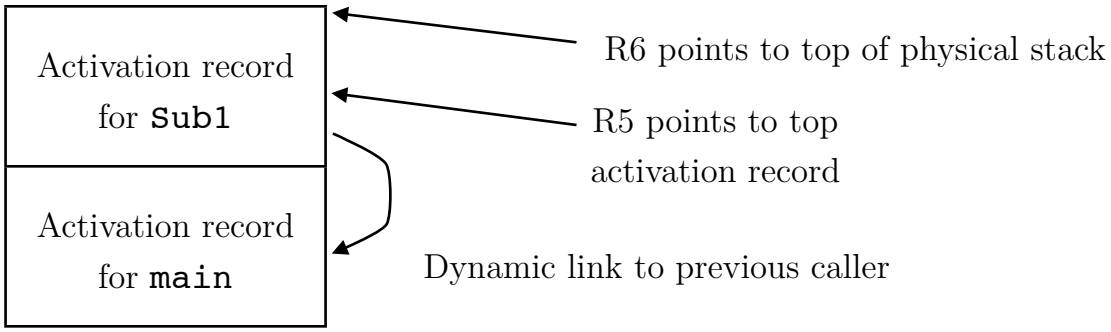
Runtime Stack (OS calls `main` calls `Sub1`)

(3) `Sub1` calls `Sub2` (push record for call of `Sub2`)



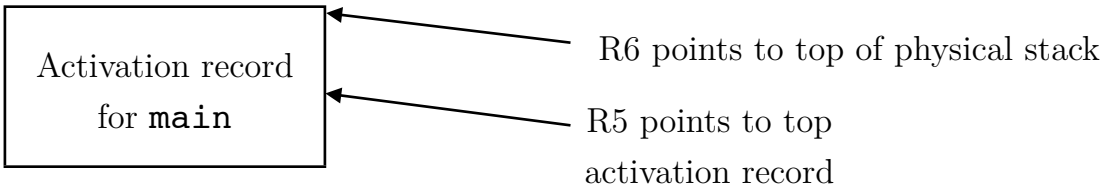
Runtime Stack (`main` calls `Sub1` calls `Sub2`)

(4) `Sub2` returns to `Sub1` (pop record for `Sub2`; `Sub1` record at top)



Runtime Stack (After `Sub2` returns to `Sub1`)

(5) `Sub1` returns to `main` (pop record for `Sub1`; `main` record at top)

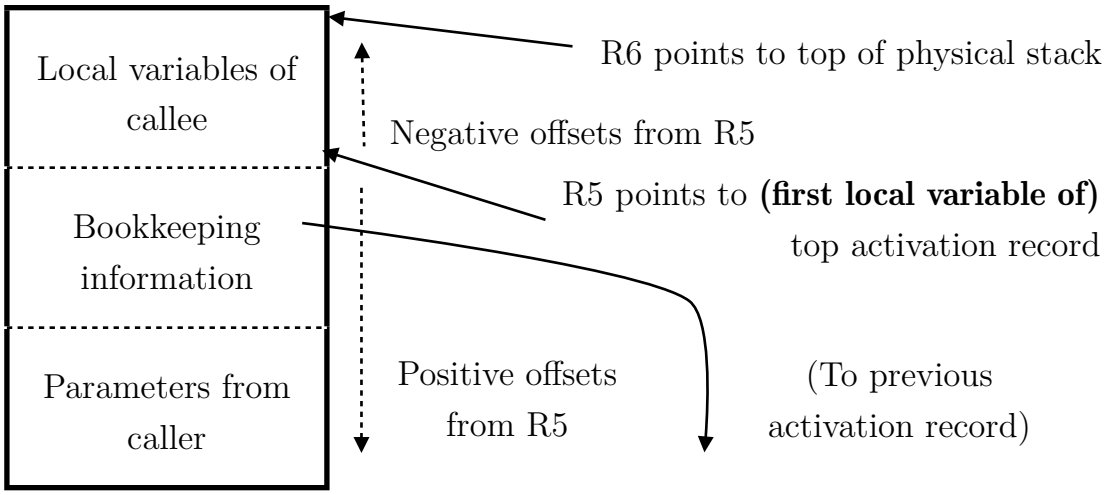


Runtime Stack (After `Sub1` returns to `main`)

(6) `main` returns to OS (pop record for `main`; stack empty)

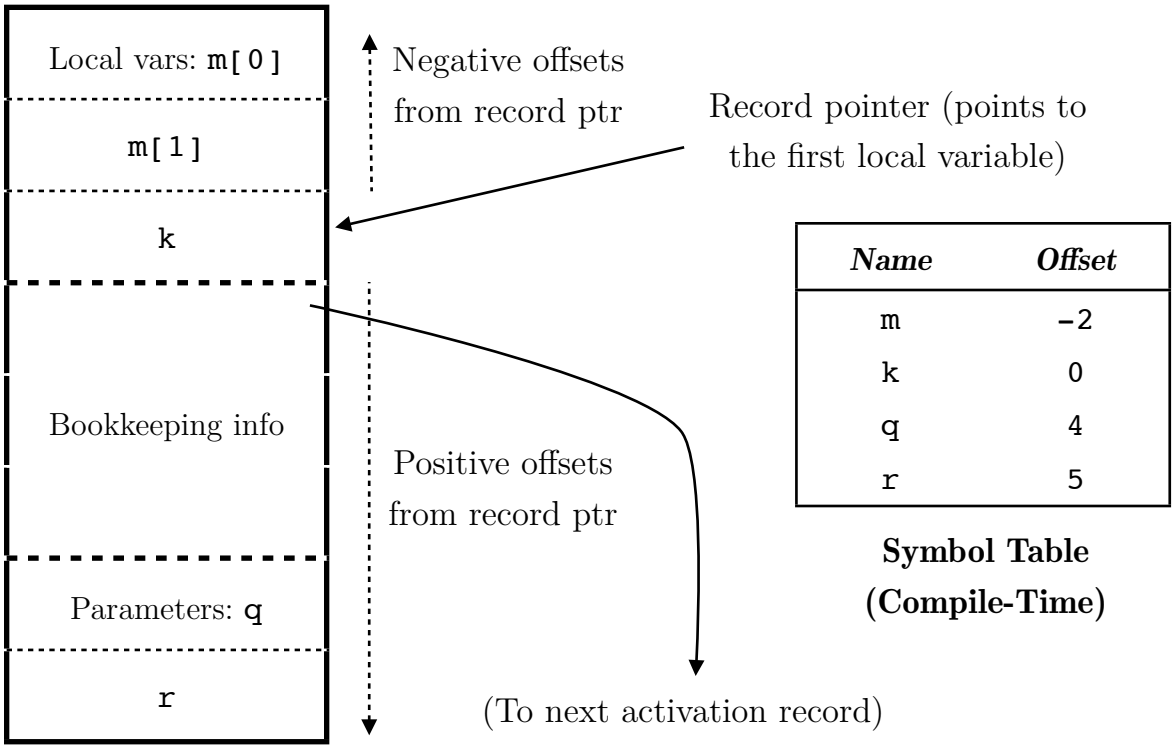
G. Contents of Activation Record

- An activation record for a call of procedure P contains
  - Space for the local variables of P. (Size depends on P.)
  - Bookkeeping information (fixed size).
  - Space for the parameters of P (filled in by the caller, with the values of the arguments). (Size depends on P.)
- The record pointer points to the first local variable so that the bookkeeping information is always at the same offsets regardless of the number and size of local variables and parameters.



Runtime Activation Record (at top of stack)

- The local variables have negative offsets from the record pointer; the parameters have positive offsets.

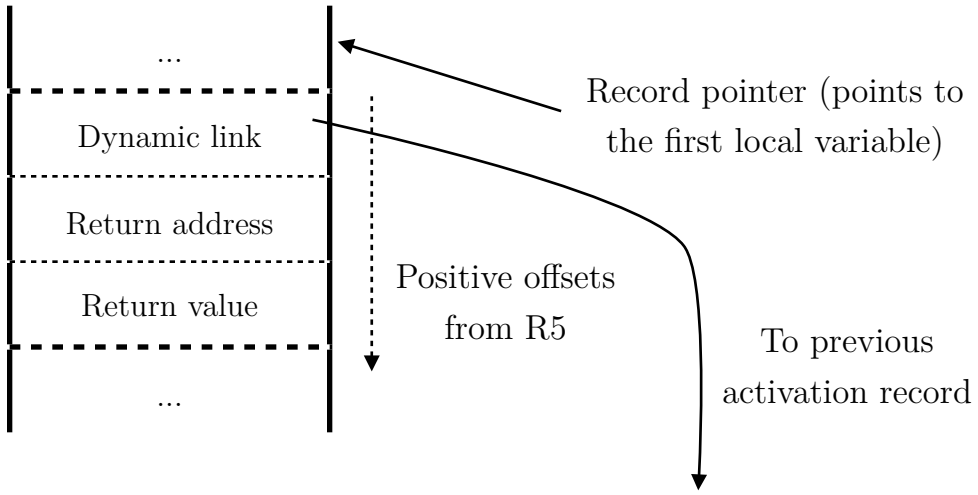


Name	*Offset*
m	-2
k	0
q	4
r	5

Symbol Table (Compile-Time)

Local Data for `Sub2` (assuming 4 words of bookkeeping info)

- Bookkeeping information



Bookkeeping Information in Activation Record

  - The **dynamic link** field points to the activation record of the caller.
  - The return address field points to the location we should return to, within the caller.
  - The return value field is space for whatever value the called routine wants to return (if any).

H. How to Call a Subroutine

- To push or pop an activation record requires pushes or pops of multiple words of data. The textbook uses `R6` as the pointer to the top of the underlying word-oriented stack that's used to implement the runtime stack.
  - To push the value in a *register* to the stack, we use

```
STR register,R6,0
ADD R6,R6,-1
```

  - To pop the top of the stack into a *register*, we use

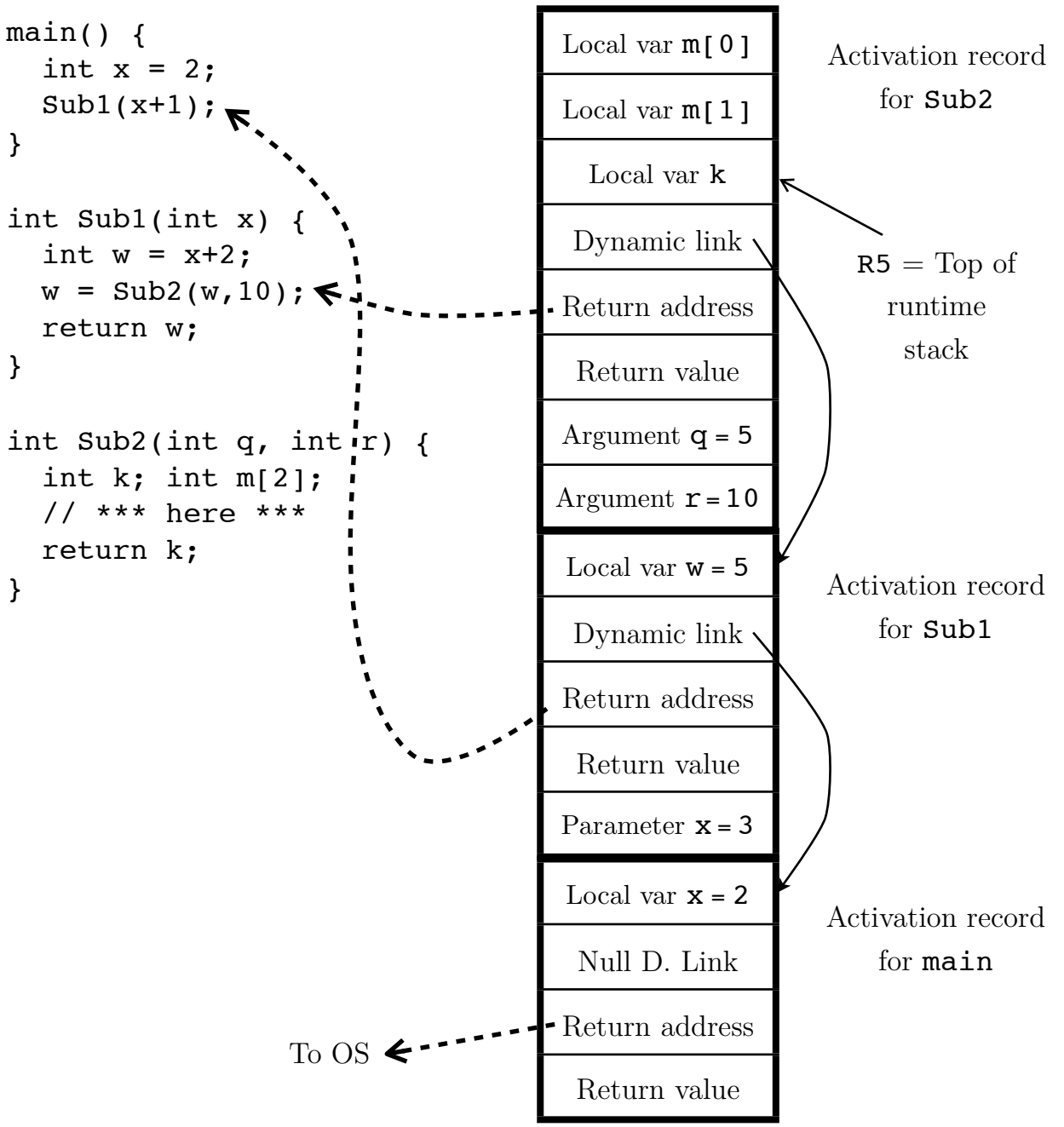
```
ADD R6,R6,1
LDR register,R6,0
```

- To call a procedure (say routine P calls Q)
  - The caller P has to set up part of the activation record
    - Push the arguments (right to left) onto the stack

        - Then `JSR` to the routine being called (saving `R7` ← `PC` as we go)
    - The callee Q sets up the rest of the activation record.
        - Push space for the return value.
        - Push return address in `R7` onto the stack.
        - Push activation record pointer in `R5` onto the stack. (This sets the dynamic link field.)
        - Set pointer to top activation record to address of first local variable (copy `R5` ← `R6`).
        - Push spaces for its local variables, if any.
- As the called routine Q executes, it can use `LDR`/`STR` *register*`, R5,` N to access the local variables and parameters. (The offset N depends on the variable/parameter.)
- To return from a call
    - The routine Q that is returning has to
        - Pop off any local variables
        - Pop the dynamic link from stack into `R5`. (This makes the activation record for the caller P the top record.)
        - Pop the caller's return address into `R7`.
        - Copy the return value to the top slot of the stack
        - Return (`RET` = `JMP R7`)
    - The routine P being returned to has to:
        - Pop off the return value that was pushed on by the called routine.
        - Pop off and throw away the arguments it pushed on before the call.

I. Sample of Nested Call

- The example below shows that the `main` program called `Sub1` which in turn called `Sub2`.
- Note that the `x` of the `main` program and the `x` of `Sub1` are stored at different places, since they're in different routines.



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.

B. Outcomes

After this activity, you should be able to

- Sketch the runtime stack at various points during the execution of a program.

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. Suppose in the `Sub1`/`Sub2` example, `Sub2` returns `8`. Sketch the runtime stack after `Sub2` returns to `Sub1` and `Sub1` assigns `w = Sub2(…)`.
5. Continuing from Question 1, sketch the runtime stack just after `Sub1` returns to the `main` program.
6. Sketch code that `Sub2` could use to implement `k = q+r;` Use the symbol table

Name	*Offset*
`m`	`-2`
`k`	`0`
`q`	`4`
`r`	`5`

Symbol Table

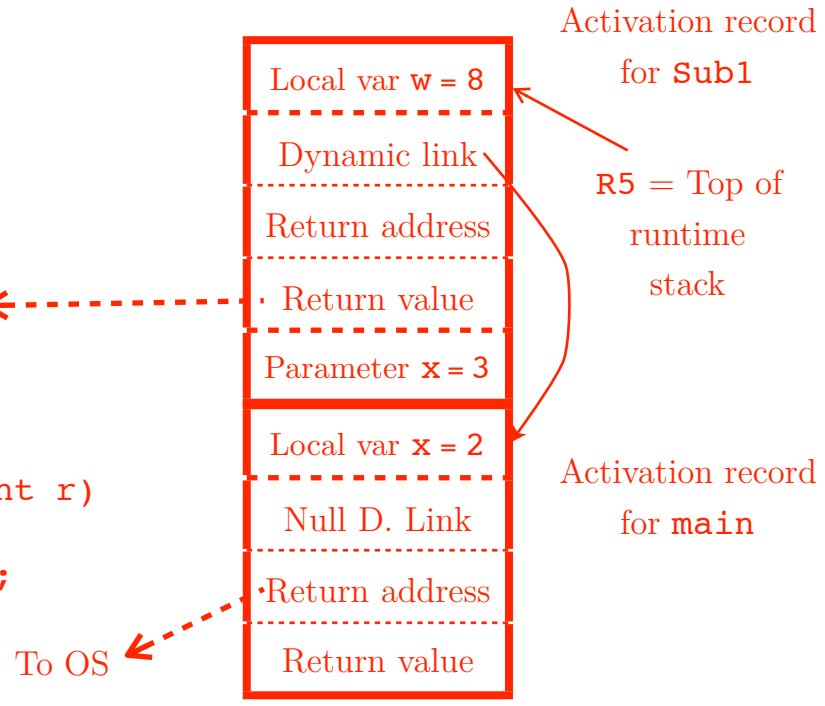
7. If a subroutine wants to pass an array instead of an integer, then in C-like languages, we just copy the address of the initial element. What would the pros and cons be of copying the entire array onto the stack instead?

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.

Solution

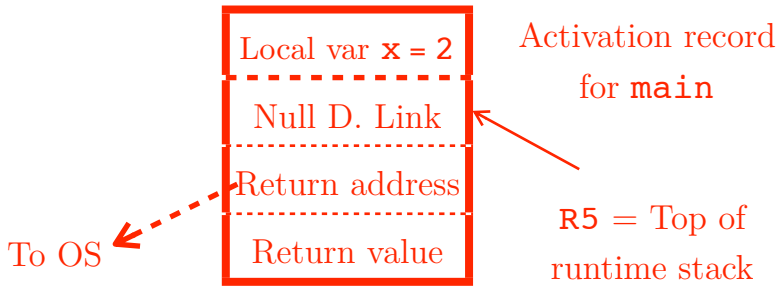
1. The caller might not know which registers need to be saved/restored. More importantly, the save/restore code would have to be written with each call instead of being written just once in the called routine. This would take extra space.
2. When the subroutine tries to return to the caller (via `JMP 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.
4. (Note change to value of `w`.)

```
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;
}
```



5.

```
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) {
   ...
}
```



6. To implement `k = q+r;` (using `R0` and `R1` as temporary registers)

```
LDR  R0,R5,4   ; R0 = q
LDR  R1,R5,5   ; R1 = r
ADD  R0,R1,R0  ; R0 = q+r
STR  R0,R5,0   ; k = q+r
```

7. If we copy the entire array, then the subroutine can make changes to its parameter without changing the argument array. On the other hand, copying the entire array takes more time and it uses more space on the runtime stack.

8. If P calls Q, the activation record for the call will contain
   (a) The return address to the code for P.
   (c) A link to the activation record for P.
   (f) Space for the local variables of Q.
   (g) Space for the arguments being passed to Q.
   (i) Space for the value that Q will return to P.

