2.12 Supporting procedures in computer hardware

This section has been set as optional by your instructor.

A *procedure* or function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time; parameters act as an interface between the procedure and the rest of the program and data, since they can pass values and return results. We describe the equivalent to procedures in Java in COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java), but Java needs everything from a computer that C needs. Procedures are one way to implement **abstraction** in software.

Procedure: A stored subroutine that performs a specific task based on the parameters with which it is provided.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his or her tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a "need to know" basis, so the spy can't make assumptions about the spymaster.

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. LEGv8 software follows the following convention for procedure calling in allocating its 32 registers:

- `X0–X7`: eight parameter registers in which to pass parameters or return values.
- `LR (X30)`: one return address register to return to the point of origin.

In addition to allocating these registers, LEGv8 assembly language includes an instruction just for the procedures: it branches to an address and simultaneously saves the address of the following instruction in register `LR (X30)`. The *branch-and-link instruction* (`BL`) is simply written

```
BL ProcedureAddress
```

Branch-and-link instruction: An instruction that branches to an address and simultaneously saves the address of the following instruction in a register (`LR` or `X30` in LEGv8).

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This "link", stored in register `LR` (register 30), is called the *return address*. The return address is needed because the same procedure could be called from several parts of the program.

Return address: A link to the calling site that allows a procedure to return to the proper address; in LEGv8 it is stored in register `LR (X30)`.

To support the return from a procedure, computers like LEGv8 use the *branch register* instruction (`BR`), introduced above to help with case statements, meaning an unconditional branch to the address specified in a register:

```
BR    LR
```

The branch register instruction branches to the address stored in register `LR`—which is just what we want. Thus, the calling program, or *caller*, puts the parameter values in `X0 – X7` and uses `BL X` to branch to procedure `X` (sometimes named the *callee*). The callee then performs the calculations, places the results in the same parameter registers, and returns control to the caller using `BR LR`.

Caller: The program that instigates a procedure and provides the necessary parameter values.

Callee: A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the *program counter*, abbreviated *PC* in the LEGv8 architecture, although a more sensible name would have been *instruction address register*. The `BL` instruction actually saves PC + 4 in register `LR` to link to the byte address of the following instruction to set up the procedure return.

Program counter (PC): The register containing the address of the instruction in the program being executed.

PARTICIPATION ACTIVITY 2.12.1: Procedure basics.

Start 2x speed

Idea
Main program:
InstrA
InstrB
Call ProcX with f, g
Instr operating on r
...

ProcX:
Instrs operating on f and g

LEGv8
Main program:
InstrA
InstrB
Set X0-X7
BL ProcXAddress *(sets LR to next Instr's address)*
Instr operating on X0-X7
...

ProcX:
Instrs operating on X0-X7

Return r

Set X0-X7
BR LR

PARTICIPATION ACTIVITY 2.12.2: Basic procedure call and return.

1) A main program will call a procedure Power for computing x^y. Currently, x is in X19, y is in X20. How might the program pass the parameter values to Power?
   - `ADD X0, X19, XZR`
`ADD X1, X20, XZR`
   - `ADD X19, X0, XZR`
`ADD X20, X1, XZR`
   - `ADD X9, X19, XZR`
`ADD X10, X20, XZR`

2) A first part of a main program calls procedure Power to compute x^y, where x is in X19, y is in X20. Later, the program is to call Power again, but this time x is in X21 and y is in X26. How might the program pass the parameter values to Power?
   - Copy X21 to X0, and X26 to X1.
   - Not possible; x and y must be in X19 and X20.

3) A main program calls a Power procedure using the instruction: `BL Power`. That instruction is at address 1000. What happens to LR?
   - Nothing; BL is unrelated to LR.
   - LR is set to 1000.
   - LR is set to 1004.

4) A procedure Power computes X0 to the power of X1. In which register should Power write the result before returning?
   - X0
   - X19

5) A procedure Power computes X0 to the power of X1. How should the procedure branch back to the next instruction in the caller?
   - `BR Caller`
   - `BR LR`
   - `BL LR`

Using more registers

Suppose a compiler needs more registers for a procedure than the eight argument registers. Since we must cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the *Hardware/Software Interface* in COD Section 2.3 (Operands of the computer hardware).

The ideal data structure for spilling registers is a *stack*—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found. The *stack pointer* (`SP`), which is just one of the 32 registers, is adjusted by one doubleword for each register that is saved or restored. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a *push*, and removing data from the stack is called a *pop*.

Stack: A data structure for spilling registers organized as a last-in- first-out queue.

Stack pointer: A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In LEGv8, it is register `SP`.

Push: Add element to stack.

Pop: Remove element from stack.

By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Elaboration

As mentioned above, in the full ARMv8 instruction set, the stack pointer is folded into register 31. In some instructions -- data transfers and arithmetic immediates that don't set flags when it is the destination register or first source register -- register 31 indicates `SP` but in the rest, such as arithmetic register instructions or in flag setting instructions, it indicates the zero register (`XZR`). Given that this trick only saves one register, would complicate the datapath in COD Chapter 4 (The Processor), and it is a bit confusing, LEGv8 just assumes `SP` is one of the other 31 general purpose registers; we use X28 for `SP`.

Example 2.12.1: Compiling a C procedure that doesn't call another procedure.

Let's turn the following example into a C procedure:

```
long long int leaf_example (long long int g, long long
int h, long long int i, long long int j)
{
      long long int f;

      f = (g + h) - (i + j);
      return f;
}
```

What is the compiled LEGv8 assembly code?

Answer

The parameter variables `g`, `h`, `i`, and `j` correspond to the argument registers `X0`, `X1`, `X2`, and `X3`, and `f` corresponds to `X19`. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example in Section 2.2 (Operations of the computer hardware), which uses two temporary registers (`X9` and `X10`). Thus, we need to save three registers: `X19`, `X9`, and `X10`. We "push" the old values onto the stack by creating space for three doublewords (24 bytes) on the stack and then store them:

```
SUBI SP, SP, #24       // adjust stack to make room for 3 items
STUR X10, [SP,#16]     // save register X10 for use afterwards
STUR X9, [SP,#8]       // save register X9 for use afterwards
STUR X19, [SP,#0]      // save register X19 for use afterwards
```

The animation below shows the stack before, during, and after the procedure call. The next three statements correspond to the body of the procedure:

```
ADD X9,X0,X1           // register X9 contains g + h
ADD X10,X2,X3          // register X10 contains i + j
SUB X19,X9,X10         // f = X9 - X10, which is (g + h) - (i + j)
```

To return the value of f, we copy it into a parameter register:

```
ADD X0,X19,XZR         // returns f (X0 = X19 + 0)
```

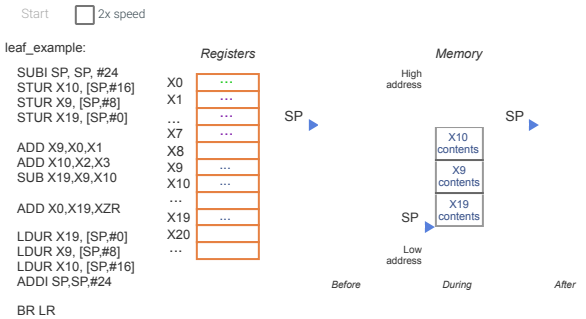
Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
LDUR X19, [SP,#0]      // restore register X19 for caller
LDUR X9, [SP,#8]       // restore register X9 for caller
LDUR X10, [SP,#16]     // restore register X10 for caller
ADDI SP,SP,#24         // adjust stack to delete 3 items
```

The procedure ends with a branch register using the return address:

```
BR LR                  // branch back to calling routine
```

PARTICIPATION ACTIVITY 2.12.3: The values of the stack pointer and the stack before, during, and after the procedure call (COD Figure 2.11).



The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, LEGv8 software separates 19 of the registers into two groups:

- `X9 – X17`: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `X19 – X28`: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller does not expect registers `X9` and `X10` to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore `X19`, since the callee must assume that the caller needs its value.

PARTICIPATION ACTIVITY 2.12.4: Procedure call using the stack.

1) The stack is a region in the set of registers.
   - ○ True
   - ○ False
2) The BL instruction copies registers to the stack.
   - ○ True
   - ○ False
3) A procedure should copy all of registers X9-X17 and X19-X28 to the stack, before executing the procedure's computations.
   - ○ True
   - ○ False
4) If a procedure will update registers X19, X20, X21, and X22, the procedure should make room on the stack by adding 32 to SP.
   - ○ True
   - ○ False
5) Upon computing a value to return, the procedure might copy that value into register X0.
   - ○ True
   - ○ False
6) LEGv8 allows a procedure to modify registers X9-X17 without saving those registers to the stack and restoring those registers upon returning.
   - ○ True
   - ○ False

Nested procedures

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, attention must be paid when invoking nonleaf procedures.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register `X0` and then using `BL A`. Then suppose that procedure A calls procedure B via `BL B` with an argument of 7, also placed in `X0`. Since A hasn't finished its task yet, there is a conflict over the use of register `X0`. Similarly, there is a conflict over the return address in register `LR`, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

One solution is to push all the other registers that must be preserved on the stack, just as we did with the saved registers. The caller pushes any argument registers (`X0 – X7`) or temporary registers (`X9 – X17`) that are needed after the call. The callee pushes the return address register `LR` and any saved registers (`X19 – X25`) used by the callee. The stack pointer `SP` is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory, and the stack pointer is readjusted.

Example 2.12.2: Compiling a recursive C procedure, showing nested procedure linking.

Let's tackle a recursive procedure that calculates factorial:

```
long long int fact (long long int n)
{
      if (n < 1) return (1);
            else return (n * fact(n – 1));
}
```

What is the LEGv8 assembly code?

Answer

The parameter variable `n` corresponds to the argument register `X0`. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and `X0`:

```
fact:
      SUBI SP, SP, #16 // adjust stack for 2 items
      STUR LR, [SP,#8] // save the return address
      STUR X0, [SP,#0] // save the argument n
```

The first time `fact` is called, `STUR` saves an address in the program that called `fact`. The next two instructions test whether `n` is less than 1, going to `L1` if `n ≥ 1`.

```
SUBIS ZXR,X0, #1       // test for n < 1
B.GE  L1               // if n >= 1, go to L1
```

If `n` is less than 1, `fact` returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in `X1`. It then pops the two saved values off the stack and branches to the return address:

```
ADDI  X1,XZR, #1       // return 1
ADDI  SP,SP,#16        // pop 2 items off stack
BR    LR               // return to caller
```

Before popping two items off the stack, we could have loaded `X0` and `LR`. Since `X0` and `LR` don't change when `n` is less than 1, we skip those instructions.

If `n` is not less than 1, the argument `n` is decremented and then `fact` is called again with the decremented value:

```
L1: SUBI X0,X0,#1      // n >= 1: argument gets (n - 1)
    BL fact            // call fact with (n - 1)
```

The next instruction is where `fact` returns. Now the old return address and old argument are restored, along with the stack pointer:

```
LDUR X0, [SP,#0]       // return from BL: restore argument n
LDUR LR, [SP,#8]       // restore the return address
ADDI SP, SP, #16       // adjust stack pointer to pop 2 items
```

Next, the value register `X1` gets the product of old argument `X0` and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until COD Chapter 3 (Arithmetic for Computers):

```
MUL X1,X0,X1           // return n * fact (n - 1)
```

Finally, `fact` branches again to the return address:

```
BR LR                  // return to the caller
```

Hardware/Software Interface

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. Example types include integers and characters. C has two storage classes: *automatic* and *static*. Automatic variables are local to a procedure and are discarded when the procedure exits. Static variables exist across exits from and entries to procedures. C variables declared outside all procedures are considered static, as are any variables declared using the keyword *static*. The rest are automatic. To simplify access to static data, some LEGv8 compilers reserve a register, called the *global pointer*, or `GP`. For example `X27` could be reserved for `GP`.

Global pointer: The register that is reserved to point to the static area.

The figure below summarizes what is preserved across a procedure call. Note that several schemes preserve the stack, guaranteeing that the caller will get the same data back on a load from the stack as it stored onto the stack. The stack above `SP` is preserved simply by making sure the callee does not write above `SP`; `SP` is itself preserved by the callee adding exactly the same amount that was subtracted from it; and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there.

Figure 2.12.1: What is and what is not preserved across a procedure call (COD Figure 2.12).

If the software relies on the global pointer register, discussed in the following subsections, it is also preserved.

Preserved	Not preserved
Saved registers: X19-X27	Temporary registers: X9-X15
Stack pointer register: X28(SP)	Argument/Result registers: X0-X7
Frame pointer register: X29(FP)	
Link Register (return address): X30(LR)	
Stack above the stack pointer	Stack below the stack pointer

PARTICIPATION ACTIVITY 2.12.5: Non-leaf procedures.

Consider a procedure P that calls another procedure Q.

1) P is a leaf procedure.
   - ○ True
   - ○ False

2) P should always save LR on the stack.

- True
- False

3) If P will write to X0 to pass a parameter to Q, P might first need to save X0 to the stack.
   - True
   - False

4) When P calls Q, P should expect that Q might pop less from the stack than Q pushed to the stack.
   - True
   - False

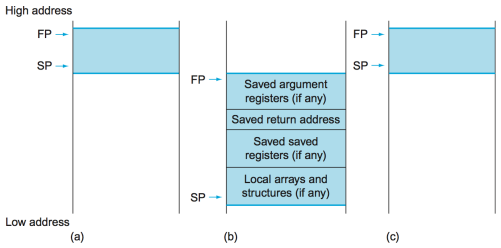
Allocating space for new data on the stack

The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure's saved registers and local variables is called a *procedure frame* or *activation record*. The figure below shows the state of the stack before, during, and after the procedure call.

Procedure frame: Also called ***activation record***. The segment of the stack containing a procedure's saved registers and local variables.

Figure 2.12.2: Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call (COD Figure 2.13).

The frame pointer (`FP` or `X29`) points to the first doubleword of the frame, often a saved argument register, and the stack pointer (`SP`) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in `SP` on a call, and `SP` is restored using `FP`.



Some ARMv8 compilers use a *frame pointer* (`FP`) to point to the first doubleword of the frame of a procedure. A stack pointer might change during the procedure, and so references to a local variable in memory might have different offsets depending on where they are in the procedure, making the procedure harder to understand. Alternatively, a frame pointer offers a stable base register within a procedure for local memory-references. Note that an activation record appears on the stack whether or not an explicit frame pointer is used. We've been avoiding using `FP` by avoiding changes to `SP` within a procedure: in our examples, the stack is adjusted only on entry to and exit from the procedure.

Frame pointer: A value denoting the location of the saved registers and local variables for a given procedure.

PARTICIPATION ACTIVITY 2.12.6: Activation record / procedure frame.

1) Procedure P needs 2 local variables. P will almost certainly need to put those local variables on the stack.
   - True
   - False

2) Procedure X needs 30 local variables. P will almost certainly need to put some of those local variables on the stack.
   - True
   - False

3) Whether a procedure puts a local variable in a register or on the stack doesn't impact performance.
   - True
   - False

4) Local variables v and w saved to the stack can be accessed as offsets from the frame pointer.
   - True

- False

5) All of the saved registers and local variables for a procedure call are referred to as an activation record.
   - True
   - False

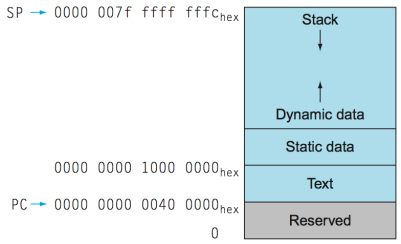
Allocating space for new data on the heap

In addition to automatic variables that are local to procedures, C programmers need space in memory for static variables and for dynamic data structures. The figure below shows the LEGv8 convention for allocation of memory when running the Linux operating system. The stack starts in the high end of the user addresses space (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)) and grows down. The first part of the low end of memory is reserved, followed by the home of the LEGv8 machine code, traditionally called the *text segment*. Above the code is the *static data segment*, which is the place for constants and other static variables. Although arrays tend to be a fixed length and thus are a good match to the static data segment, data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the *heap*, and it is placed next in memory. Note that this allocation allows the stack and heap to grow toward each other, thereby allowing the efficient use of memory as the two segments wax and wane.

Text segment: The segment of a UNIX object file that contains the machine language code for routines in the source file.

Figure 2.12.3: The LEGv8 memory allocation for program and data (COD Figure 2.14).

These addresses are only a software convention, and not part of the LEGv8 architecture. The user address space is set to 2^{39} of the potential 2^{64} total address space given a 64-bit architecture (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)). The stack pointer is initialized to `0000 007f ffff fffc`$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at `0000 0000 0040 0000`$_{hex}$. The static data immediately after the end of the text segment; in this example, we assume that address is `0000 0000 1000 0000`$_{hex}$. Dynamic data, allocated by `malloc` in C and by `new` in Java, is next. It grows up toward the stack in an area called the *heap*.



C allocates and frees space on the heap with explicit functions. `malloc()` allocates space on the heap and returns a pointer to it, and `free()` releases space on the heap to which the pointer points. C programs control memory allocation, which is the source of many common and difficult bugs. Forgetting to free space leads to a "memory leak," which ultimately uses up so much memory that the operating system may crash. Freeing space too early leads to "dangling pointers," which can cause pointers to point to things that the program never intended. Java uses automatic memory allocation and garbage collection just to avoid such bugs.

PARTICIPATION ACTIVITY 2.12.7: Heap and stack.

1) A procedure's local variables are normally stored on the heap.
   - True
   - False

2) If a running program should create a new array, the array is normally stored on the heap.
   - True
   - False

3) The heap typically grows upwards in memory, while the stack grows downwards.
   - True
   - False

4) If a program allocated huge arrays, the program may cause the heap to run into the stack, causing an error.
   - True
   - False

5) If a program has a procedure P that calls itself recursively, and a bug causes

P to just keep calling itself recursively without end, eventually the stack will run into the heap, causing an error.

- ○ True
- ○ False

The figure below summarizes the register conventions for the LEGv8 assembly language. This convention is another example of making the **common case fast**: most procedures can be satisfied with up to eight argument registers, nine saved registers, and seven temporary registers without ever going to memory.

Figure 2.12.4: LEGv8 register conventions (COD Figure 2.15).

`X8` is used by procedures that return a result via a pointer. ARM discourages the use of registers `X16` to `X18` as `X16` and `X17` may be used by the linker (see COD Section 2.12 (Translating and starting a program)), and `X18` may be used to create platform independent code, which would be specified by the platforms Application Binary Interface.

Name	Register number	Usage	Preserved on call?
X0-X7	0–7	Arguments/Results	no
X8	8	Indirect result location register	no
X9-X15	9–15	Temporaries	no
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	no
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	no
X18	18	Platform register for platform independent code; otherwise a temporary register	no
X19-X27	19–27	Saved	yes
X28 (SP)	28	Stack Pointer	yes
X29 (FP)	29	Frame Pointer	yes
X30 (LR)	30	Link Register (return address)	yes
XZR	31	The constant value 0	n.a.

Elaboration

What if there are more than eight parameters? The LEGv8 convention is to place the extra parameters on the stack below the frame pointer. The procedure then expects the first eight parameters to be in registers `X0` through `X7` and the rest in memory, addressable via the frame pointer.

The frame pointer is convenient because all references to variables in the stack within a procedure will have the same offset. The frame pointer is not necessary, however. The ARMv8 C compiler uses a frame pointer, but some C compilers do not; they treat register 29 as another save register.

Elaboration

Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with recursive procedure calls. For example, consider a procedure used to accumulate a sum:

```
long long int sum (long long int n, long long int acc) {
      if (n > 0)
         return sum(n - 1, acc + n);
      else
         return acc;
}
```

Consider the procedure call `sum(3, 0)`. This will result in recursive calls to `sum(2, 3)`, `sum(1, 5)`, and `sum(0, 6)`, and then the result 6 will be returned four times. This recursive call of sum is referred to as a tail call, and this example use of tail recursion can be implemented very efficiently (assume `X0 = n`, `X1 = acc`, and the result goes into X2):

```
sum: SUBS XZR, X0, XZR      // compare n to 0
     B.LE sum_exit          // go to sum_exit if n <= 0
     ADD X1, X1, X0         // add n to acc
     SUBI X0, X0, #1        // subtract 1 from n
     B sum                  // go to sum
sum_exit:
     ADD X2, X1, XZR        // return value acc
     BR LR                  // return to caller
```

PARTICIPATION ACTIVITY 2.12.8: Check yourself: Memory management in C and Java.

1) C programmers manage data explicitly, while data management is automatic in Java.

- ○

- True
- False

2) C leads to more pointer bugs and memory leak bugs than does Java.
   - True
   - False

Provide feedback on this section