The Virtual Machine: Subroutines

Prof. Naga Kandasamy
ECE Department, Drexel University

We discuss how to design mechanisms to support subroutines — also called as procedures, functions, or methods — of procedural or object-oriented languages. Consider the following snippet of C code wherein the *main()* function calls *add()* which adds two numbers and returns the result back to *main()*.

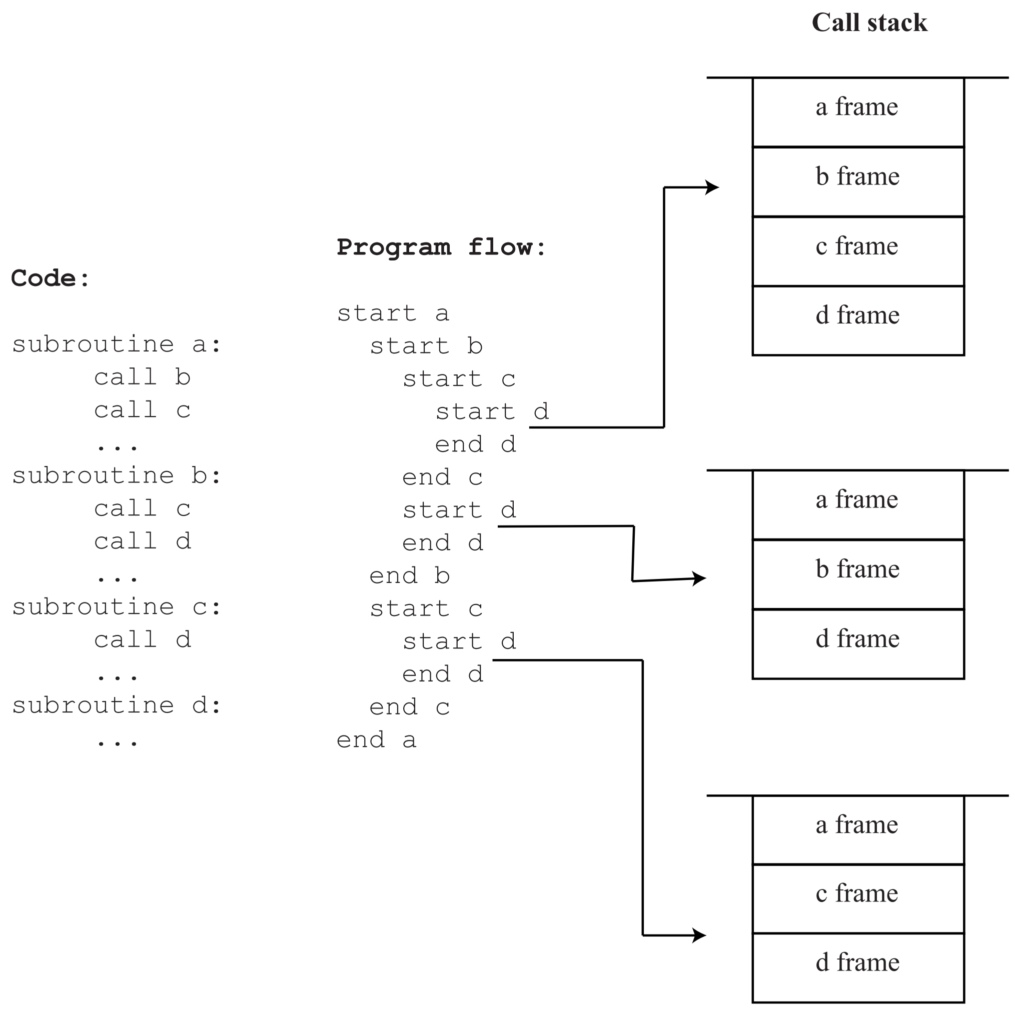
```c
/* Add c = a + b */
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

/* Main function */
int main(int argc, char **argv)
{
    int a = 5;
    int b = 10;
    int c, d;

    c = add(a, b);
    d = add(a, c);

    return;
}
```

To support subroutines such as *add()*, the compiler must handle the following details behind the scene:[1]

- Passing parameters (or arguments) from *main()* to the called subroutine *add()*.
- Saving the state of *main()* before switching to execute *add()*.
- Allocating space for the local variables declared within *add()*.
- Jumping to execute *add()*.
- Returning values from *add()* back to the caller function *main()*.
- Reclaiming the memory space occupied by *add()*, once it returns.
- Restoring state of *main()*.
- Jumping to execute code in *main()* immediately following the spot where we left it.

The compiler relieves the programmer from having to take care of the above housekeeping duties. The stack structure used in our low-level implementation supports the above-described housekeeping very well.

[1]Note that *main()* is also a subroutine from the executing program's perspective. It is the entry function into a C program.



```
Code:

subroutine a:
     call b
     call c
     ...
subroutine b:
     call c
     call d
     ...
subroutine c:
     call d
     ...
subroutine d:
     ...
```

```
Program flow:

start a
  start b
    start c
      start d
      end d
    end c
    start d
    end d
  end b
  start c
    start d
    end d
  end c
end a
```

Fig. 1: Call stack of an executing program.

Let us now define two very important terms:

- *Frame:* This refers to the subroutine's local variables, the arguments on which it operates, its working stack, and the other memory segments that support its operation.
- *Call stack:* This global memory area is a stack structure that stores frames of active subroutines of a program — frames of the currently active subroutine and all the subroutines waiting for it to return The main reason for having a call stack is to keep track of the point to which each active subroutine should return control when it finishes executing. An active subroutine is one that has been called, but is yet to complete execution, after which control should be handed back to the point of call. Though the subroutine calling chain may be arbitrarily deep as well as recursive, at any given point in time only one subroutine executes at the top of the chain, while all the other subroutines down the calling chain are waiting for it to terminate. When subroutine `a` calls subroutine `b`, we can save `a`'s frame on the stack and branch to execute `b`. When `b` returns, we can restore `a`'s frame off the stack, and continue executing `a` as if nothing happened. This execution model is illustrated in Fig. 1.

Our VM language features three function-related commands:

- function *f k*: This command starts the code of a function named *f* which has *k* local variables.
- call *f n*: This command calls function *f*, stating that *n* arguments have already been pushed onto the stack by the caller.
- return: This command returns to the calling function.

A function has a symbolic name whose scope is global — all functions in all files are seen by each other and may call each other using the function name.that is used globally to call it. The function name is an arbitrary string composed of any sequence of letters, digits, underscore (_), dot (.), and colon (:) that does not begin with a digit.

The following code snippets illustrate how the VM implementation supports subroutines. The overall program consists of four files, each containing code for a single function: *sys.vm*, which contains the initialization routine; *main.vm*, which is the entry point into the program; and two other files *add.vm* and *sub.vm*. We detail the mechanism at a high level for now and delve into the specific implementation later.

The *init* function initializes the starting memory location of the call stack and invokes *main*, the entry point to our program. This example declares no local variables within the function body.

```
// sys.vm
// Code to intialize the system

function init 0
    set sp 256
    call main 0
    end
```

The main function calls a sequence of functions that implement simple arithmetic operations. Four local variables are declared within the function body, corresponding to variables names, say *a*, *b*, *c*, and *d*. These variables are stored in the *local* segment of the function, at offsets 0, 1, 2, and 3.

```
// main.vm
// Main function calls a sequence of arithmetic operations

function main 4
    push constant 5
    pop local 0             // a = 5
    push constant 10
    pop local 1             // b = 10
    push local 0
    push local 1
    call add 2              // a + b
    pop local 0             // a = a + b
    push constant 100
    pop local 2             // c = 100
    push constant 80
    pop local 3             // d = 80
    push local 2
    push local 3
    call add 2
    pop local 2             // c = c + d
```

```
21      push local 0
22      push local 2
23      call sub 2            // a - c
24      pop local 0           // a = a - c
25      push local 0          // push a as the return value
26      return
```

Lines 5 through 7 initialize variables *a* and *b* to values 5 and 10, respectively, through the working stack of the function. Two arguments — in this case, *a* and *b* — are pushed on the stack prior to calling the `add` function in line 11. Just prior to returning, *add* places the result on top of the stack. The result is then popped and stored back into variable *a*, in line 12. The other calls are handled similarly.

The *add* function first locates the two arguments passed to it by *main* and pushes them onto its working stack, in lines 5 and 6. It then invokes the *add* operation. Recall that this binary operation pops two operands from the stack, performs the addition, and pushes the result back to the working stack. Prior to returning, *add* re-positions the return value in the call stack appropriately such that *main* can pop it off.

```
1  // add.vm
2  // Adds argument1 + argument2
3
4  function add 0
5      push argument 0
6      push argument 1
7      add
8      return
```

The *sub* function behaves similarly.

```
// sub.vm
// subtracts argument1 - argument2

function sub 0
    push argument 0
    push argument 1
    sub
    return
```

Let's take a closer look at the low-level protocol implementation details pertaining to calling a function.

- Before calling a function, the caller must push the necessary arguments onto the stack. For example,

```
    // Some code
    push local 0
    push local 2
    push local 5
    call foo 3
label return_address
    // Rest of code
```

pushes three arguments to the stack for function *foo*. Figure 2 shows the call stack's state at this point.

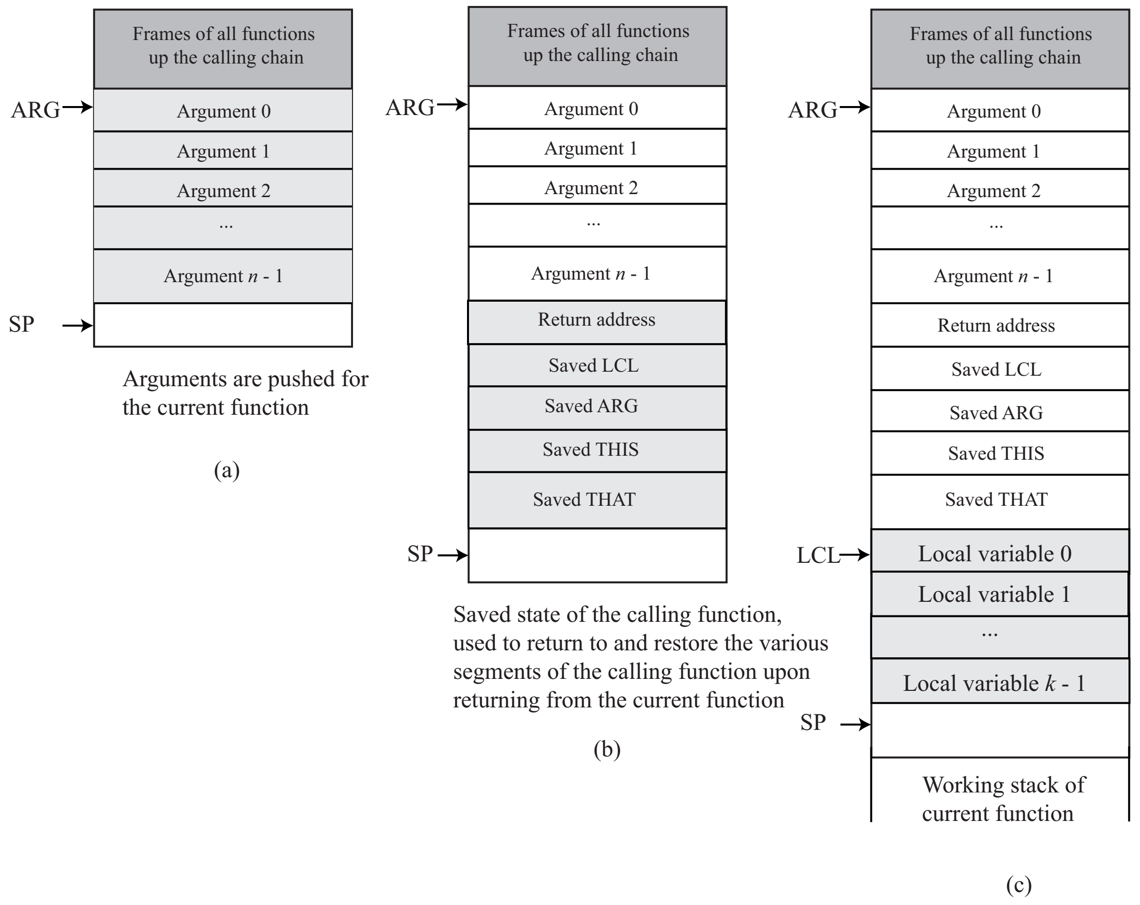

Frames of all functions up the calling chain
ARG
Argument 0
Argument 1
Argument 2
...
Argument n - 1
SP
Arguments are pushed for the current function
(a)
Frames of all functions up the calling chain
ARG
Argument 0
Argument 1
Argument 2
...
Argument n - 1
Return address
Saved LCL
Saved ARG
Saved THIS
Saved THAT
SP
Saved state of the calling function, used to return to and restore the various segments of the calling function upon returning from the current function
(b)
Frames of all functions up the calling chain
ARG
Argument 0
Argument 1
Argument 2
...
Argument n - 1
Return address
Saved LCL
Saved ARG
Saved THIS
Saved THAT
LCL
Local variable 0
Local variable 1
...
Local variable k - 1
SP
Working stack of current function
(c)

Fig. 2: Evolution of the call stack when a function is called.

- Once arguments have been placed on the stack, the caller invokes function *foo* using:

```
call foo n
```

This call provides information that n arguments have been previously pushed to the stack. So, the implementation of *call* is as follows:

```
push return_address     // Use the label declared below
push LCL                // Push LCL register of the calling function
push ARG                // Push ARG register of the calling function
push THIS               // Push THIS register of the calling function
push THAT               // Push THAT register of the calling function
ARG = SP - 5 - n        // Reposition ARG to point to start of ARG segment
LCL = SP                // Reposition LCL
goto foo                // Transfer control to function
(return_address)        // Declare a label marking the return point
```

The command creates a label for the return point in the code of the caller function and saves it as the return address, saves register values for the caller, and re-positions the `ARG` and `LCL` registers for the called function. It then transfers control to it. This situation is shown in Fig.2(b).

- Once control is transferred to *foo*, we hit the entry point

  ```
  function foo k
  ```

  in the VM code. Here k denotes the number of local variables declared within the function. The *function* command is implemented as follows:

  ```
  (foo)                   // Label for the entry point
  Repeat k times:
      push 0              // Push 0 into the stack
  ```

  This provides the label for the caller to jump to. It also initializes variables that live within the function's local segment to 0. This scenario is shown in Fig. 2(c). When the called function starts executing, its argument segment has been initialized with actual argument values passed by the caller and its local variables segment has been allocated and initialized to zeros. The static segment that the called function sees has been set to the static segment of the VM file to which it belongs, and the working stack that it sees is empty. Segments *this*, *that*, *pointer*, and *temp* are undefined upon entry.

- Once the function has finished execution, control is passed back to the caller using the VM command

  ```
  return
  ```

  The implementation is as follows:

  ```
  1 FRAME = LCL          // FRAME is stored in temporary register, say R14
  2 RET = *(FRAME - 5)   // Store return address in temporary register, say R15
  3 *ARG = pop() // Reposition return value for caller, to top of ARG segment
  4 SP = ARG + 1         // Restore SP for the caller
  5 THAT = *(FRAME - 1)  // Restore THAT of the caller
  6 THIS = *(FRAME - 2)  // Restore THIS of the caller
  7 ARG = *(FRAME - 3)   // Restore ARG of the caller
  8 LCL = *(FRAME - 4)   // Restore LCL of the caller
  9 goto RET             // Jump to return address
  ```

  First, we fix a reference point called `FRAME` within the function's stack, which is the starting location of the LCL segment. This reference point makes it easy to restore the various registers for the caller prior to passing control. The '*' notation denotes the de-referencing operation — that is, we treat the contents of `ARG` and `FRAME` registers as addresses themselves and '*' denotes the value stored at that address. Lines 3 and 4 require some explanation. Before returning, the called function must push a value onto the stack. Line 3 pops the return value, which is on top of the working stack of this function, and copies it to the first location of the `ARG` segment. `SP` is re-positioned to point to the very next location. This is illustrated in Fig. 3. We do this so that once control returns to the caller, the return value is on top of it's working stack for use.

  In summary, after the called function returns, the arguments that the caller has pushed before the call have disappeared from the stack, and a return value (that always exists) appears at the top of the stack; After the called function returns, the caller's memory segments *argument, local, static, this, that,* and *pointer* are the same as before the call, and the *temp* segment is undefined.

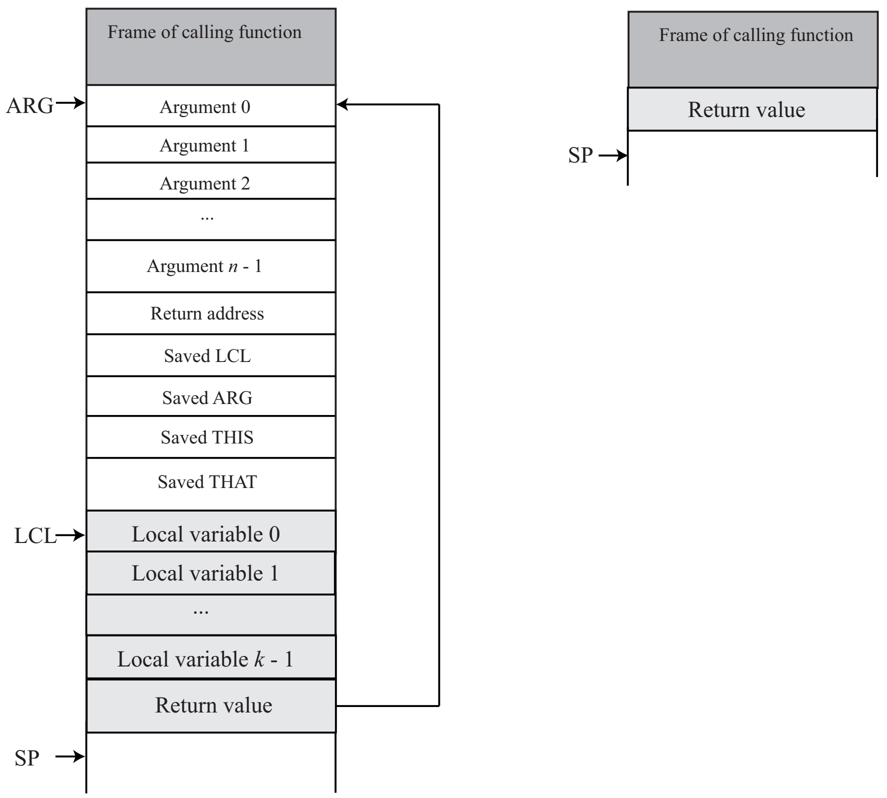


Fig. 3: Repositioning the return value and adjusting the stack pointer.

The VM commands discussed so far must be translated into appropriate sequences of Hack assembly-language commands. We will show a few examples of how to generate such code sequences.

- Generating assembly code to push various registers of the caller to the stack as part of the *call* command is straightforward. For example, the following code pushes the value of `LCL` to the stack.

```
@LCL
D = M
@SP
A = M
M = D       // Push LCL
@SP
M = M + 1   // Adjust SP
```

- Here are examples of copying the return address into `RET` and restoring `LCL` of the caller as part of the *return* command.

```
// Store LCL in temp register R14 (FRAME)
@LCL
D = M
@R14
M = D
// Store return address (RET) in temp register R15, RET = *(FRAME - 5)
@R14
D = M           // D <-- FRAME
```

```
@5
D = D - A       // D <-- FRAME - 5
A = D
D = M           // D <-- *(FRAME - 5)
@R15
M = D           // RET <-- *(FRAME - 5)
// Restore LCL = *(FRAME - 4)
@R14
D = M           // D = FRAME
@4
D = D - A       // D = FRAME - 4
A = D
D = M           // D = *(FRAME - 4)
@LCL
M = D           // LCL = *(FRAME - 4)
```

- And finally, assembly code to jump to the return address previously stored in `RET`.

```
// goto RET
@R15
A = M
0;JMP
```