Because learning changes everything.®

From Bits and Gates to C and Beyond

Functions

Chapter 14

Implementing Functions

Each function needs space for its local variables and other state.
Where should it be allocated?

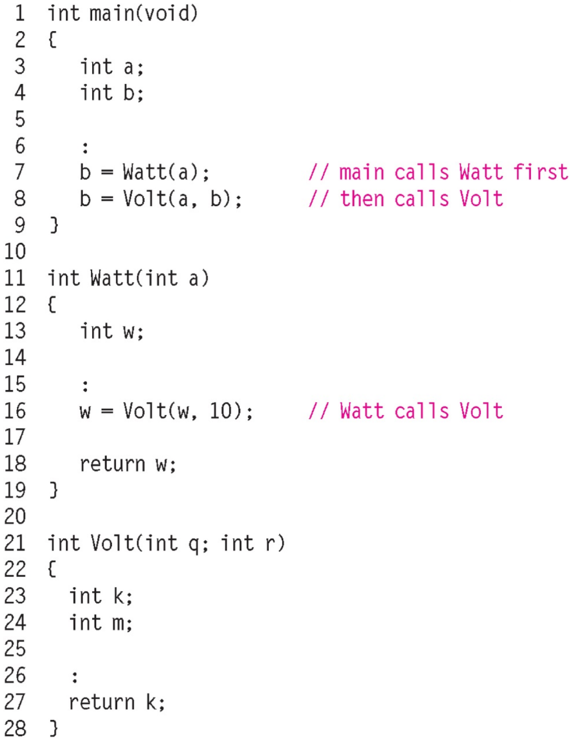
Option 1: Compiler allocates a memory space for each function.

- Does not allow recursion -- local variables would be overwritten.
- Allocates memory for functions that may not be called.

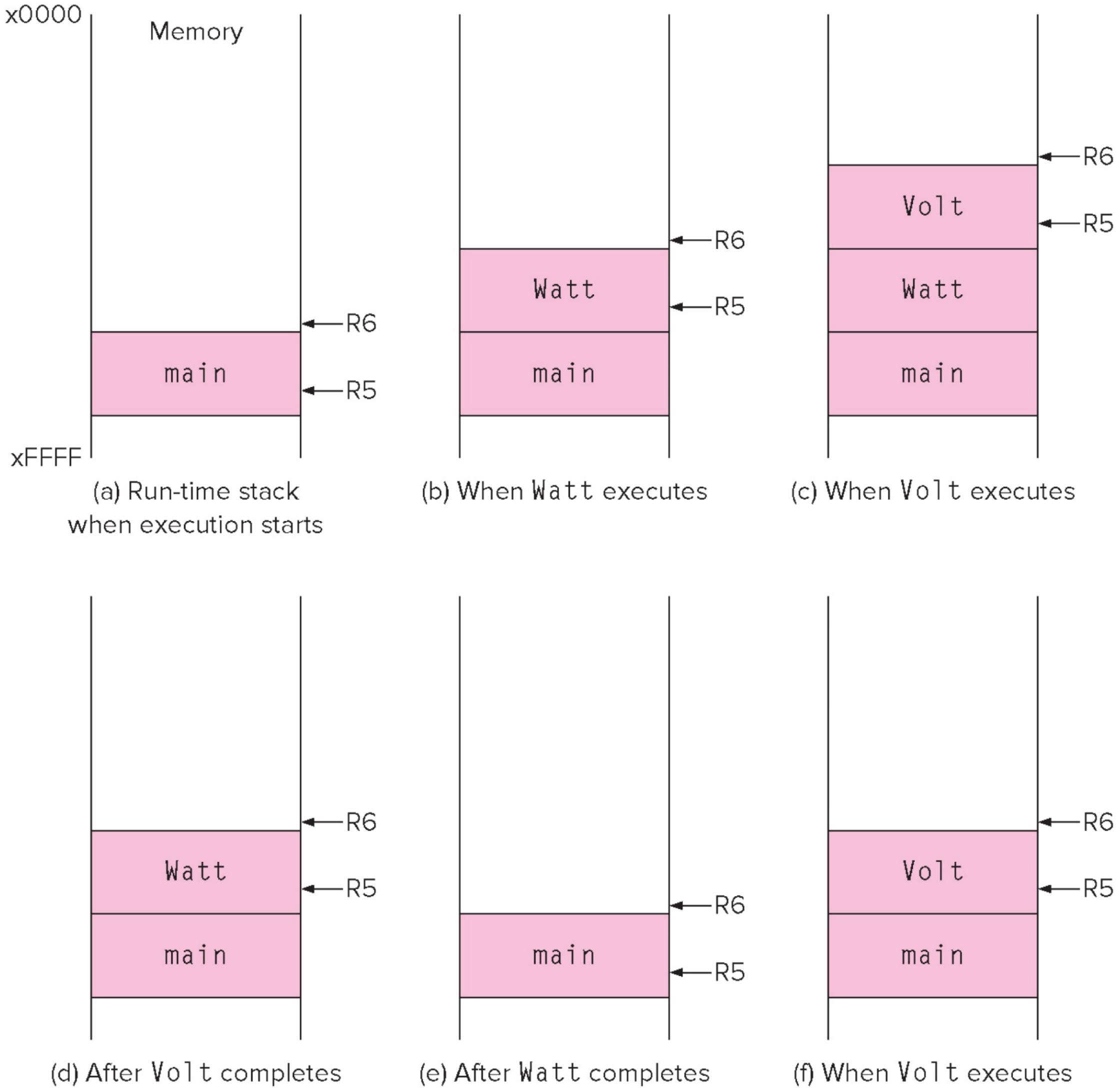
Option 2: Program allocates space on a run-time stack.

- Allocates space as each function is called; space is deallocated when function returns. Only uses the space that is needed.
- Allows recursion -- each function invocation gets its own set of locals.
- Stack's LIFO behavior matches the call-return action of functions.

Push/Pop Stack Frames



```
int main(void)
{
   int a;
   int b;

   :
   b = Watt(a);        // main calls Watt first
   b = Volt(a, b);     // then calls Volt
}

int Watt(int a)
{
   int w;

   :
   w = Volt(w, 10);    // Watt calls Volt

   return w;
}

int Volt(int q; int r)
{
  int k;
  int m;

  :
  return k;
}
```



(a) Run-time stack when execution starts

(b) When Watt executes

(c) When Volt executes

(d) After Volt completes

(e) After Watt completes

(f) When Volt executes

Access the text alternative for slide images.

Implementation using LC-3 Instructions

Note: The approach of using a run-time stack is common to all processors, not just the LC-3. While the specific instructions will be different, the concepts are the same for different ISAs.

Three places where instructions are needed:

- Caller function -- pass arguments, transfer control to callee, use return value when control is given back.
- Callee function -- preamble to save state and allocate local variables.
- End of callee function -- return statements deallocates local variables and restores state before returning to caller.

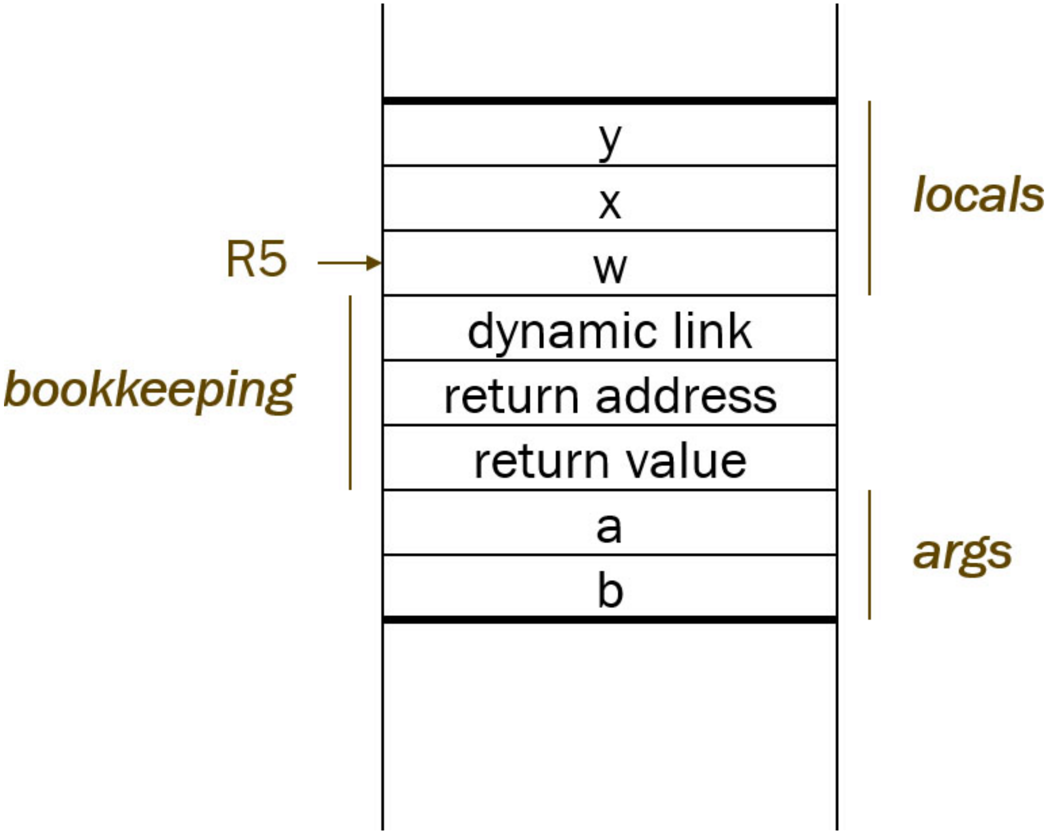
LC-3 Activation Record 1

The structure of a function's activation record is shown below.

As an example, the NoName function has two parameters (a and b) and three local variables (w, x, and y).

```
int NoName(int a, int b) {
  int w, x, y;
  ...
}
```

Name	Type	Offset	Scope
a	int	4	NoName
b	int	5	NoName
w	int	0	NoName
x	int	−1	NoName
y	int	−2	NoName



y

x

R5 → w

dynamic link

return address

return value

a

b

locals

bookkeeping

args

Access the text alternative for slide images.

LC-3 Activation Record 2

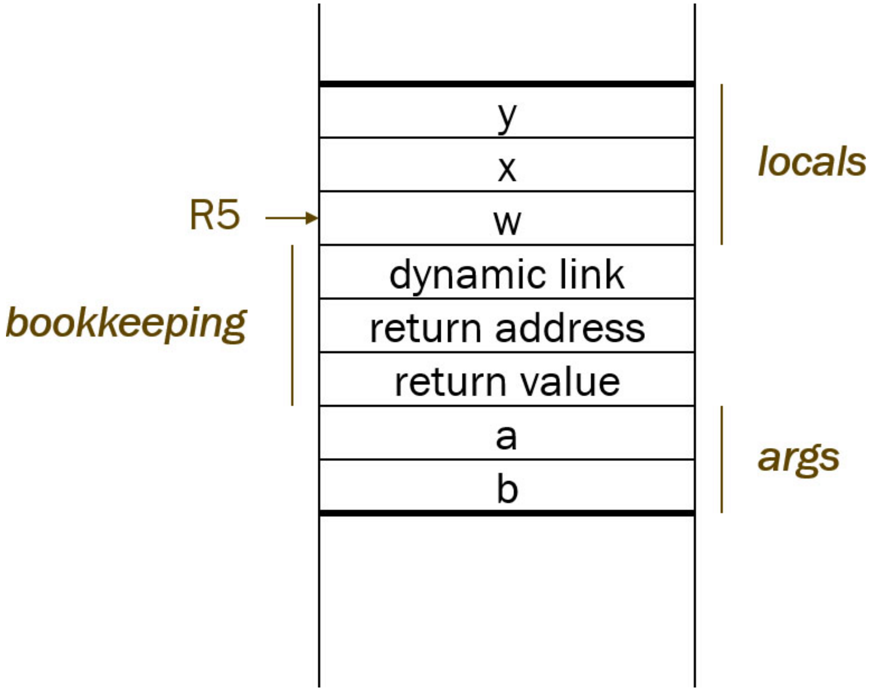
Local variables (if any) are allocated in stack order. R5 is the frame pointer, and points to the highest address of the local variable area.

The dynamic link is the saved frame pointer from the caller function. This allows us to restore the caller's state when we return.

The return address is the location of the next instruction in the caller. This is a saved version of R7, so that this function can make other function calls (JSR).

The return value will be placed above the argument values that were pushed by the caller.

The parameters (arguments) are pushed by the caller before the call. The names are known only by the callee, but the values are provided by the caller.



Caller Function

Consider this statement in the `Watt` function:

```
w = Volt(w, 10);
```

To execute the statement, we need to pass two values to the `Volt` function, call the `Volt` function, get the return value, and store it to `w`.

At the time of the call, the stack frame (activation record) for `Watt` is currently at the top of the run-time stack.

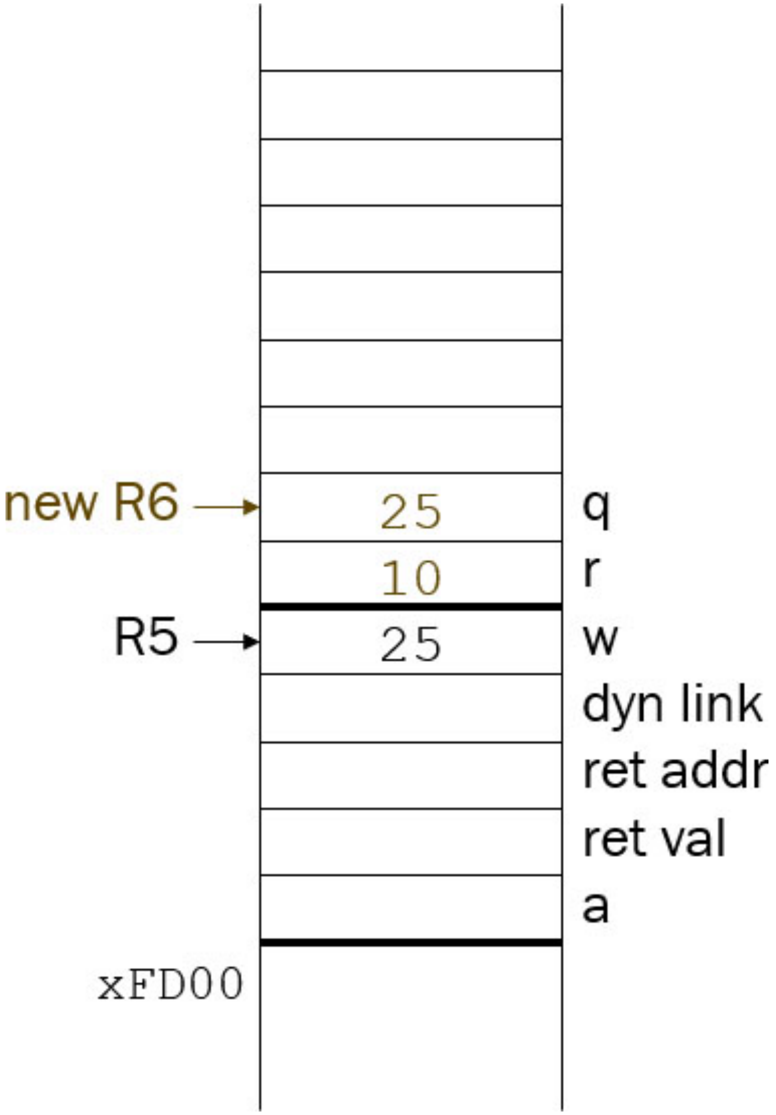
- R6 is the stack pointer -- it points to the top address on the stack.
- R5 is the frame pointer -- it points to the local variables of the current function (`Watt`).

Caller Function: Push the Arguments

To pass the arguments to a function, the caller pushes them onto the run-time stack. It pushes the arguments in right-to-left order.

```
; Volt(w, 10)
AND R0, R0, #0 ; push 10
ADD R0, R0, #10
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #0 ; push w
ADD R6, R6, #-1
STR R0, R6, #0
```

For this illustration, we show that w has a value of 25.



Access the text alternative for slide images.

Caller Function: Transfer Control to the Callee

Now that the values are on the stack, we use JSR to transfer control to the callee function (Volt). For our purposes we assume that the subroutine has a label that matches the function name.

```
JSR Volt
```

We use JSR instead of BR or JMP so that the return address will be saved in R7. The callee can use RET to give control back to the caller.

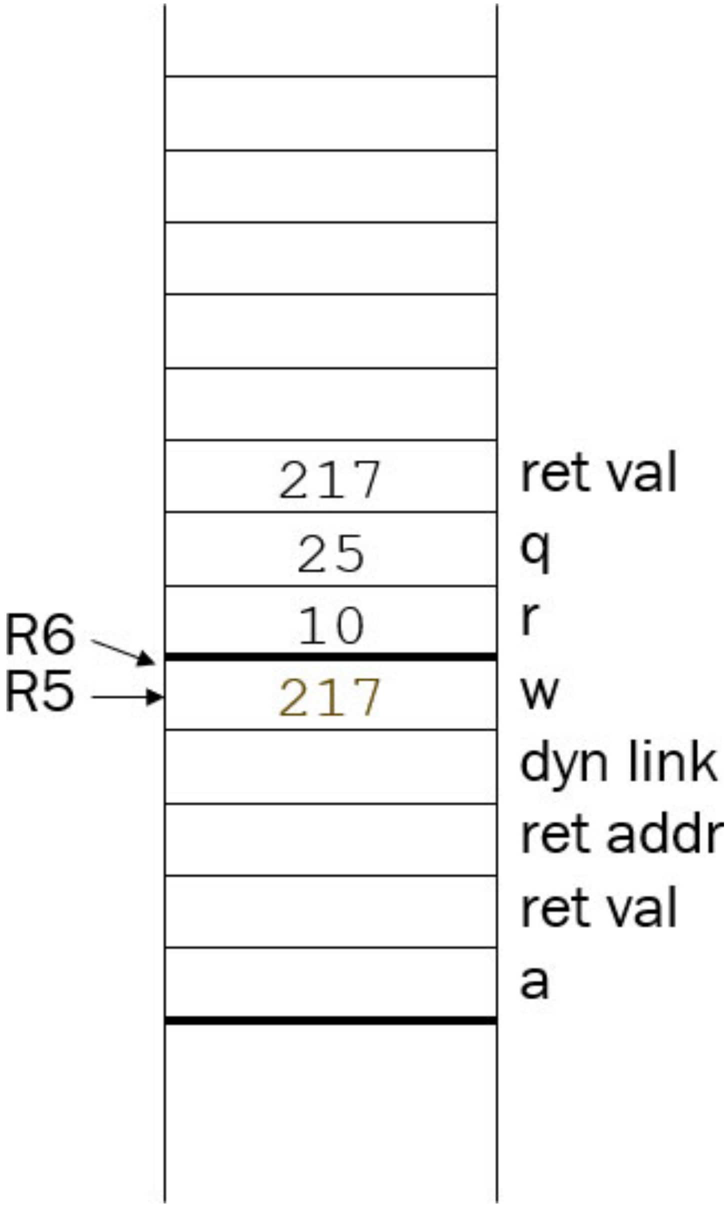
Caller Function: Using the Return Value

When the caller resumes, the return value is on the top of the stack, above the locations where the arguments were pushed.

The caller will then (a) pop the return value, and (b) pop the arguments.

Then the return value is used as directed by the caller (in this case, store to w).

```
LDR R0, R6, #0  ; pop return value
ADD R6, R6, #1
ADD R6, R6, #2  ; pop arguments
; w = Volt(...);
STR R0, R5, #0  ; store to w
```



Access the text alternative for slide images.

© McGraw Hill

Caller: Complete Code

```
; w = Volt(w,10);

AND R0, R0, #0    ; push 10
ADD R0, R0, #10
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #0    ; push w
ADD R6, R6, #-1
STR R0, R6, #0

JSR Volt     ; call function

LDR R0, R6, #0    ; pop return value
ADD R6, R6, #1
ADD R6, R6, #2    ; pop arguments
STR R0, R5, #0    ; store r.v. to w
```

At this point, the processor will start executing the Volt subroutine.
But that code does not belong here.
We will show the callee code next.

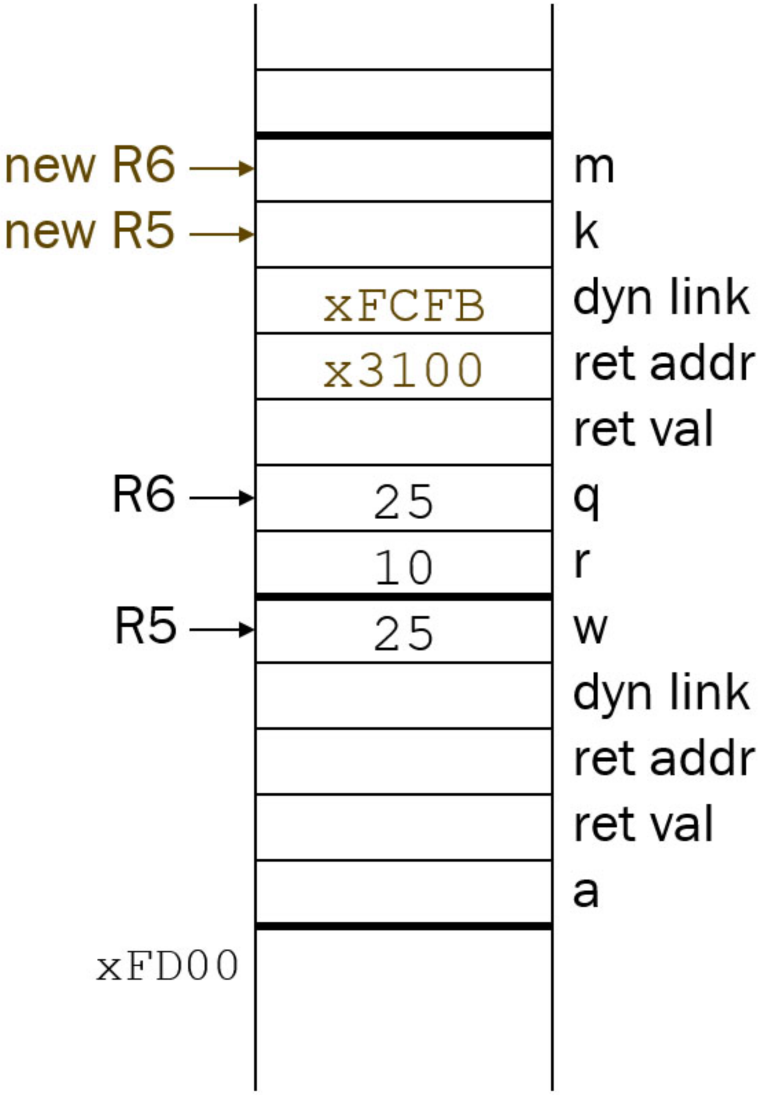
When callee returns, execution will resume here.

Callee Code: Preamble 1

When the callee function begins, the arguments are on the stack, and R5 points to the caller's local variables.

We need to allocate and fill in the rest of the callee function's activation record. And we need to set R5 to point to this function's local variables.

We call this the preamble code. It is executed at the beginning of every function. The results are shown by "new R5" and "new R6".

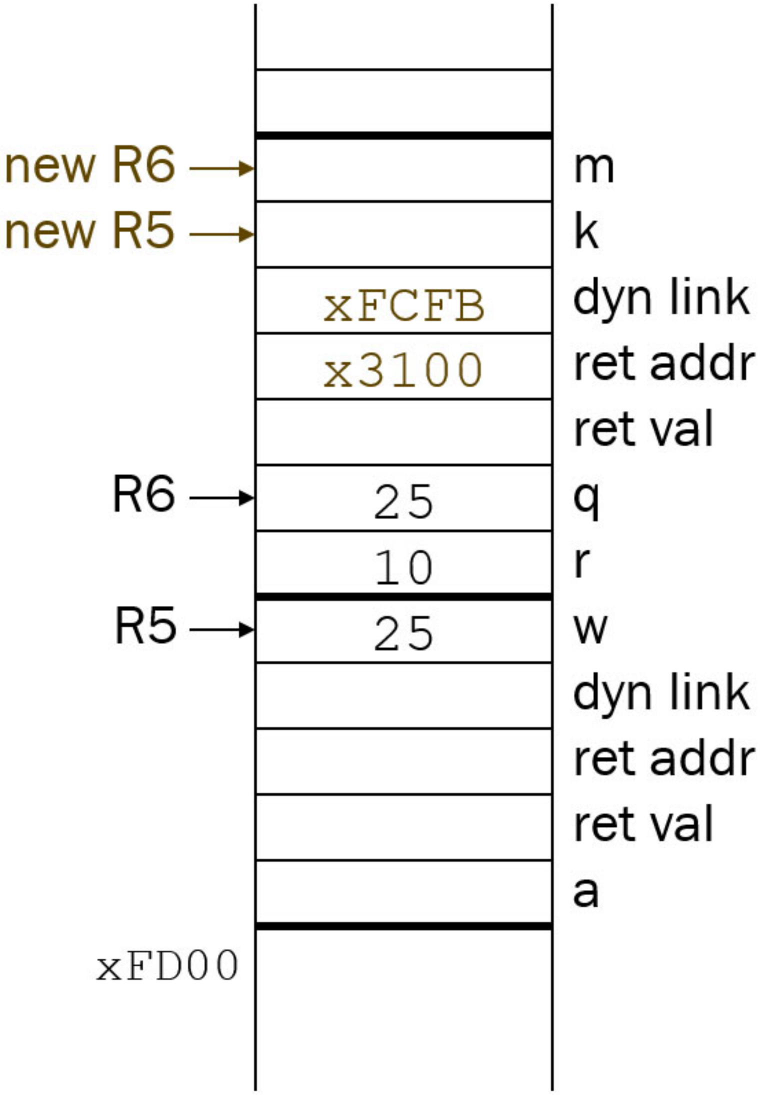


Access the text alternative for slide images.

Callee Code: Preamble 2

```
Volt  ADD R6, R6, #-1 ; push r.v.
      ; save return address
      ADD R6, R6, #-1 ; push R7
      STR R7, R6, #0
      ; save caller's frame ptr
      ADD R6, R6, #-1 ; push R5
      STR R5, R6, #0
      ; set R5 to callee locals
      ADD R5, R6, #-1
      ; push space for locals
      ADD R6, R6, #-2
```

After generating code for the preamble, the compiler will start generating code for the statements in the function definition.



Callee Code: `return` [1]

To execute the return statement, the function must:

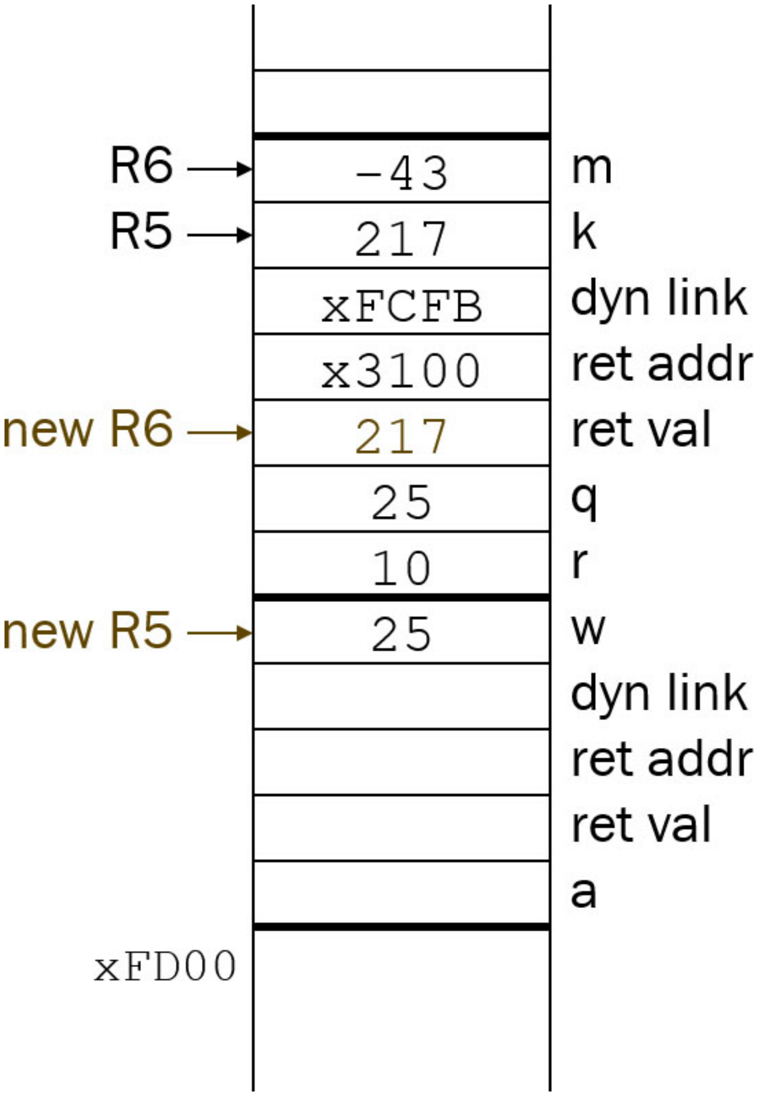
(a) evaluate the return expression.

(b) store the value to the return value.

(c) pop (deallocate) local variables.

(d) restore caller's frame pointer and return address.

(e) return to the caller, with the return value left on the top of stack.

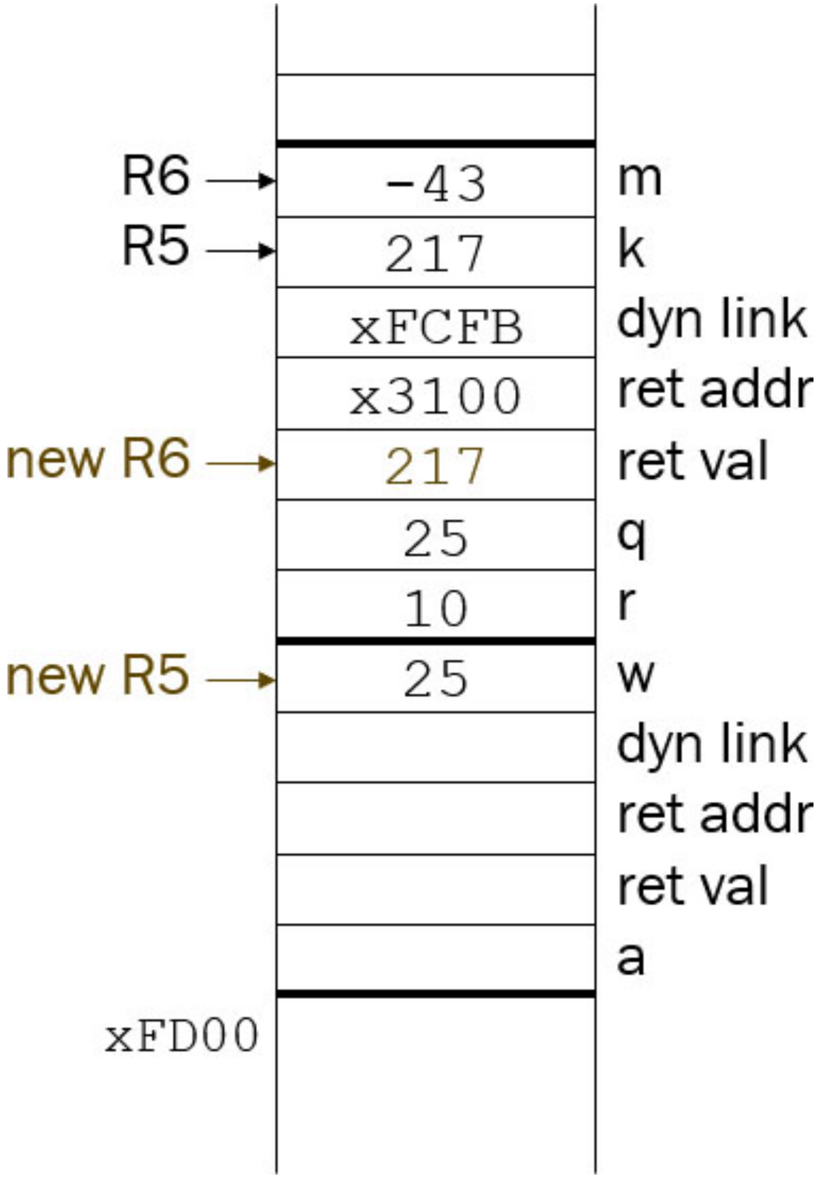


R6 →	−43	m
R5 →	217	k
	xFCFB	dyn link
	x3100	ret addr
new R6 →	217	ret val
	25	q
	10	r
new R5 →	25	w
		dyn link
		ret addr
		ret val
		a
xFD00		

Access the text alternative for slide images.

Callee Code: `return` 2

```
; return k;
LDR R0, R5, #0    ; k
STR R0, R5, #3    ; -> ret value
ADD R6, R6, #2    ; pop locals
; restore caller's frame ptr
LDR R5, R6, #0
ADD R6, R6, #1
; restore return address
LDR R7, R6, #0
ADD R6, R6, #1
; return to caller
RET
```



Summary: Caller Code

1. Push argument values, from right to left.

```
; evaluate argument, assume it's in R0
ADD R6, R6, #-1   ; push
STR R0, R6, #0
; repeat as necessary
```

2. Call function.

```
JSR function
```

3. Pop return value from stack, save in a register.

```
LDR R0, R6, #0  ; doesn't have to be R0
ADD R6, R6, #1  ; assuming a one-word return value
```

4. Pop arguments from stack.

```
ADD R6, R6, #___  ; depends on what was pushed in part 1
```

Summary: Callee Code - Preamble

1. Push space for return value.

```
ADD R6, R6, #-1     ; assume one-word return value
```

Value depends on type of return value.
If no return value, instruction is not needed.

2. Push return address.

```
ADD R6, R6, #-1
STR R7, R6, #0
```

3. Push dynamic link.

```
ADD R6, R6, #-1
STR R5, R6, #0
```

4. Set new dynamic link.

```
ADD R5, R6, #-1
```

5. Push space for local variables.

```
ADD R6, R6, #-___   ; depends on local variables
```

Summary: Callee Code - Return

1. Evaluate return value and store into activation record.

```
; evaluate argument, assume it's in R0
STR R0, R5, #3
```

2. Pop local variables.

```
ADD R6, R6, #___   ; depends on local variables
```

3. Pop/restore dynamic link.

```
LDR R5, R6, #0
STR R6, R6, #1
```

4. Pop/restore return address.

```
LDR R7, R6, #0
STR R6, R6, #1
```

5. Return to caller.

```
RET  ; or JMP R7
```

McGraw Hill Education

Because learning changes everything.®

www.mheducation.com