



中国科学技术大学  
University of Science and Technology of China

计算系统概论A  
Introduction to Computing Systems  
( CS1002A.03)

# Chapter 8 Data Structures

陈俊仕  
cjuns@ustc.edu.cn  
2023 Fall

计算机科学与技术学院  
School of Computer Science and Technology

# Outline



**1 Review**

---

**2 Subroutines**

---

**3 Control Instructions for Subroutines**

---

**4 Memory Model for Program Execution**

---

**5 The Stack**

---

**6 Implementing Functions in C**

---

# Outline



- 1 Review
- 2 Subroutines
- 3 Control Instructions for Subroutines
- 4 Memory Model for Program Execution
- 5 The Stack
- 6 Implementing Functions in C**

# A C program that uses a function to print a banner message



```
1  #include <stdio.h>
2
3  void PrintBanner();    // Function declaration
4
5  int main(void)
6  {
7      PrintBanner();    // Function call
8      printf("A simple C program.\n");
9      PrintBanner();
10 }
11
12 void PrintBanner()    // Function definition
13 {
14     printf("=====\n");
15 }
```



# Function in C

- **Smaller, simpler, subcomponent of program**
- **Provides abstraction**
  - hide low-level details
  - give high-level structure to programmer, easier to understand overall program flow
  - enables separable, independent development
- **C functions**
  - zero or multiple arguments passed in
  - single result returned (optional)
  - return value is always a particular type
- **In other languages, called procedures, subroutines, methods ...**



## Example of High-Level Structure

```
main()
{
    SetupBoard();      /* place pieces on board */
    DetermineSides(); /* choose black/white */

    /* Play game */
    do {
        WhitesTurn();
        BlacksTurn();
    } while (NoOutcomeYet());
}
```

Structure of program is evident, even without knowing implementation.

# Functions in C



```
double ValueInDollars(double amount, double rate);
```

**declaration**

```
main()
```

```
{  
  ...  
  dollars = ValueInDollars(francs, DOLLARS_PER_FRANC);  
  printf("%f francs equals %f dollars.\n", francs, dollars);  
  ...  
}
```

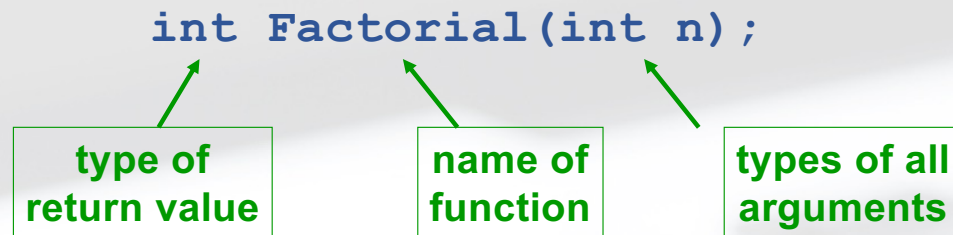
**function call (invocation)**

**definition**

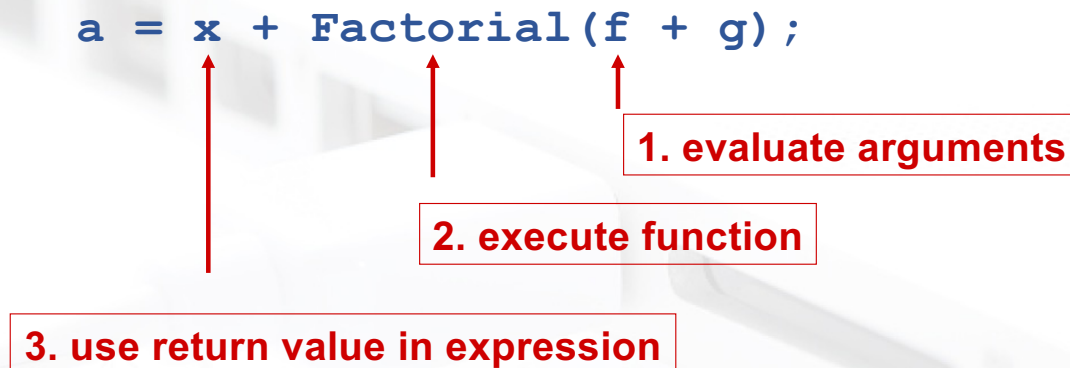
```
double ValueInDollars(double amount, double rate)
```

```
{  
  return amount * rate;  
}
```

## ■ Function Declaration (also called prototype)



## ■ Function Call -- used in expression







# Function Definition

## ■ Return type, function name, types of arguments

- must match function declaration
- give name to each argument (doesn't have to match declaration)

```
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

← gives control back to calling function and returns value

# Implementing Functions in C



- **Functions** in C are the high-level equivalent of **subroutines** at the LC-3 machine level.
- Functions in C are implemented using a similar set of mechanisms as assembly level subroutines.
- There are four basic phases in the execution of a function call:
  - (1) argument values from **the caller** are passed to **the callee**
  - (2) control is transferred to the callee, ——JSR/JSRR
  - (3) the callee executes its task
  - (4) control is passed back to the caller, along with a return value.
- We will examine how all of this is accomplished on the LC-3.



## Storage Requirements

- **Code** must be stored in memory so that we can execute the function.
- **Parameters** must be sent from the **caller** to the **callee** so that the function receives them.
- **Local/global variables** for the function must be stored somewhere, is one copy enough?
- **Return address** must be stored so that control can be returned to the **caller**.
- **Return values** must be sent from the **callee** to the **caller**, that's how results are returned.

```
// main program
Int a = 10;
Int b = 20;
Int c = foo(a, b);

Int foo(int x, int y)
{
    Int z;
    z = x + y;
    return z;
}
```

### ■ What needs to be stored?


- Code, parameters, local/global variables, return address/values

# Possible Solution: Mixed Code and Data

## ■ Function implementation:

```
// main program
Int a = 10;
Int b = 20;
Int c = foo(a, b);

Int foo(int x, int y)
{
    Int z;
    z= x + y;
    return z;
}
```



```
foo          BR   foo_begin  ;skip over data
; Memory allocation
foo_rv       .BLKW 1         ;return value
foo_ra       .BLKW 1         ;return address
foo_paramx   .BLKW 1         ;'x' parameter
foo_paramy   .BLKW 1         ;'y' parameter
foo_localz   .BLKW 1         ;'z' local variable

foo_begin    ST R7, foo_ra   ;save return
...
LD R7, foo_ra ;restore return
RET
```

## ■ Calling sequence

```
ST R1, foo_paramx ; R1 has 'x'
ST R2, foo_paramy ; R2 has 'y'
JSR foo           ; Function call
LD R3, foo_rv     ; R3 = return value
```



## Possible Solution: Mixed Code and Data

### ■ Advantages:

- Code and data are close together
- Conceptually easy to understand, code generation is relatively simple
- Minimizes register usage for variables
- Data persists through life of program
- Few instructions are spent moving data

### ■ Disadvantages:

- Code is vulnerable to self-modification
- Cannot handle recursion or parallel execution
- Consumes resource for inactive functions



## Possible Solution: Separate Code and Data

### ■ Memory allocation

```
// main program
Int a = 10;
Int b = 20;
Int c = foo(a,b);

Int foo(int x,int y)
{
    Int z;
    z= x+y;
    return z;
}
```

```
; Code for foo() and bar() are somewhere else

foo_rv      .BLKW 1      ; foo return value
foo_ra      .BLKW 1      ; foo return address
foo_paramx  .BLKW 1      ; foo 'x' parameter
foo_paramy  .BLKW 1      ; foo 'y' parameter
foo_localz  .BLKW 1      ; foo 'z' local

bar_rv      .BLKW 1      ; bar return value
bar_ra      .BLKW 1      ; bar return address
bar_paramw  .BLKW 1      ; bar 'w' parameter
```

### ■ Function code call is similar to mixed solution



## Possible Solution: Separate Code and Data

### ■ Advantages:

- Code can be marked 'read only'
- Conceptually easy to understand
- Early Fortran used this scheme
- Data persists through life of program

### ■ Disadvantages:

- Cannot handle recursion or parallel execution
- Consumes resource for inactive functions



# Run-time Stack Requirements

## ■ Consider what has to happen in a function call:

- Caller must pass parameters to the callee.
- Caller must transfer control to the callee.
- Caller must allocate space for the return value.
- Caller must save the return address.
- Callee requires space for local variables.
- Callee must return control to the caller.

## ■ Parameters, return value, return address, and locals are stored on the stack.

## ■ The order above determines the responsibility and order of stack operations.



## ■ What is a Run-time Stack?

- First In, Last Out (FILO) data structure
- PUSH adds data, POP removes data
- Overflow condition: push when stack full
- Underflow condition: pop when stack empty
- Stack grows and shrinks as data is added and removed
- Stack grows downward from the end of memory space
- Function calls allocate a **stack frame**
- Return cleans up by freeing the **stack frame**
- Corresponds nicely to nested function calls
- **Stack Trace shows current execution (Java/Eclipse)**



## Run-time Stack : Stack frame (or **activation record** )

■ **Definition:** A **stack frame** or **activation record** is the memory required for a function call:

- **Stack frame below** contains the function that called this function.
- **Stack frame above** contains the functions called from this function.
- **Caller** pushes parameters.
- **Callee** allocates the return value, saves the return address, allocates/frees local variables, and stores the return value.
- Most offsets are small, this explains LDR/STR implementation.
- Base register stores pointer, signed Parameters offset accesses both directions.



## Run-time Stack : Stack frame

- Each function has a **memory template** where it stores its local variables, some bookkeeping information, and its parameter variables .This template is called its **stack frame** or **activation record**.
- Whenever a function is called, its **stack frame** will be allocated somewhere **in memory**.
- Because the calling pattern of functions naturally follows a stack-like pattern, this allocation and deallocation will follow the pushes and pops of a stack.



## Run-time Stack : Stack Pointers(SP/R6) and Frame Pointers(FP/R5)

- Clearly we need a variable to store the **stack pointer (SP)**, LC3 assembly uses **R6**.
  - Stack execution is ubiquitous, so hardware has a stack pointer, sometimes even instructions.
  - Problem: stack pointer is difficult to use to access data, since it moves around constantly.
- Solution: allocate another variable called a **frame pointer (FP)**, LC3 assembly uses **R5**.
  - Where should frame pointer point? Our convention sets it to point to the first local variable.



## Run-time Stack

- In the previous solutions, the compiler allocated parameters and locals in fixed memory locations.
- Using an Run-time Stack means parameters and locals are constantly moving around.
- **The frame pointer** solves this problem by using fixed offsets instead of addresses.
- The compiler can generate code using offsets, without knowing where the stack frame will reside.
- **Frame pointer** needs to be saved and restored around function calls. How about the stack pointer?

# Implementing Functions in C Using a Run-Time Stack



## ■ Activation record

- information about each function, including arguments and local variables
- stored on run-time stack

## Calling function

1. push new activation record
2. copy values into arguments
3. call function

4. get result from stack

## Called function

1. execute code
2. put result in activation record
3. pop activation record from stack
4. return

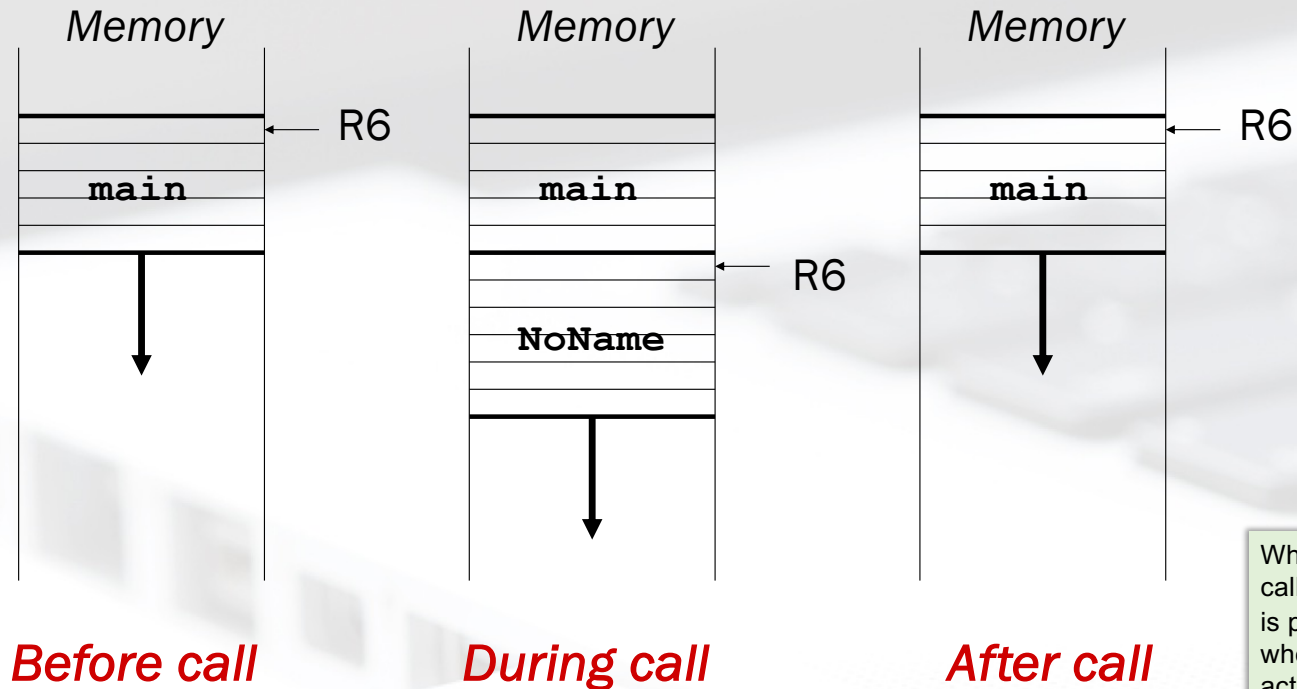
# Run-Time Stack



Stack pointer (R6) points to the beginning of a region of memory that stores local variables for the current function.

That region of memory is called an **activation record**

Recall that **local variables** are stored on the run-time stack



When a new function is called, its activation record is pushed on the stack; when it returns its activation record is popped off of the stack.

Note: Using R6 this way is incompatible with storing interrupt state on the stack.

# Activation Record



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

Name	Type	Offset	Scope
a	int	3	NoName
b	int	4	NoName
w	int	5	NoName
x	int	6	NoName
y	int	7	NoName

## Return value

- always first word in activation record
- holds value returned by function
- allocated even if function does not return a value

## Return address

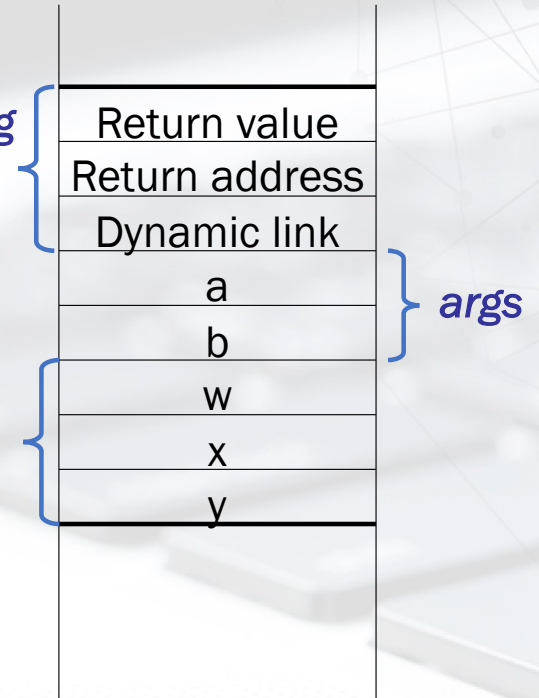
- save pointer to next instruction in calling function
- convenient location to store R7 in case another function (JSR) is called

## Dynamic link

- address of previous activation record
- used to pop this activation record from stack

*bookkeeping*

*locals*

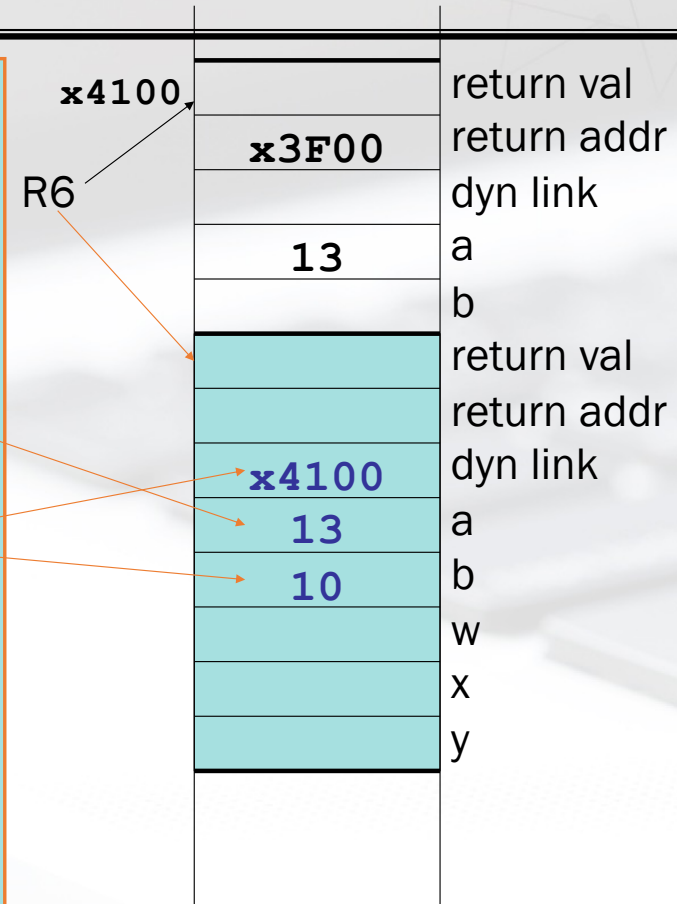




# Calling the Function



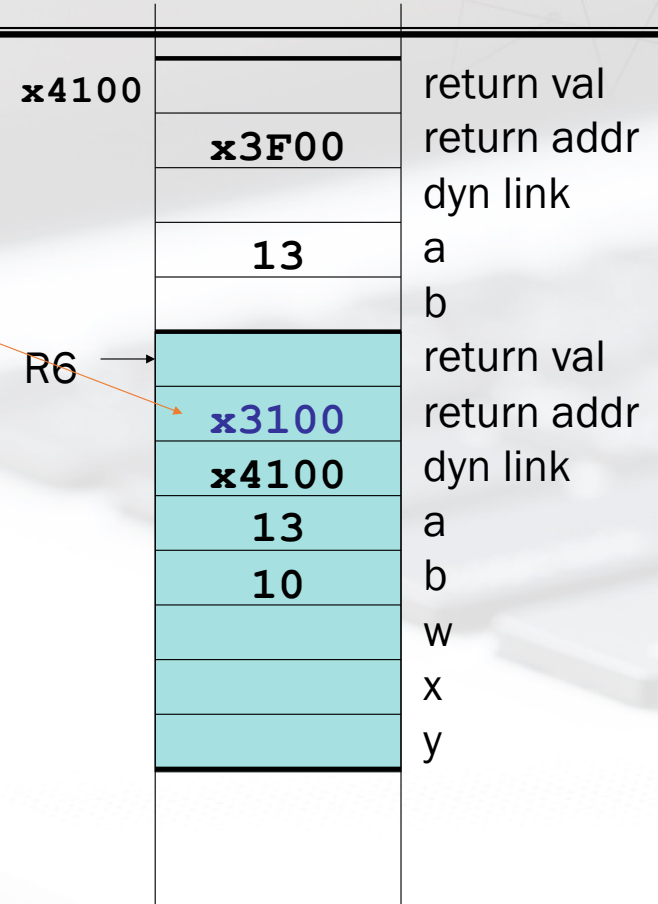
```
b = NoName(a, 10);  
  
; store a to 1st argument  
LDR R0, R6, #3  
STR R0, R6, #8  
  
; store 10 to 2nd argument  
AND R0, R0, #0  
ADD R0, R0, #10  
STR R0, R6, #9  
  
; store R6 into dynamic link  
STR R6, R6, #7  
  
; move R6 to start of new record  
ADD R6, R6, #5  
  
; call subroutine  
JSR NoName
```



Note: Caller needs to know number and type of arguments, doesn't know about local variables.

# Starting the Callee Function

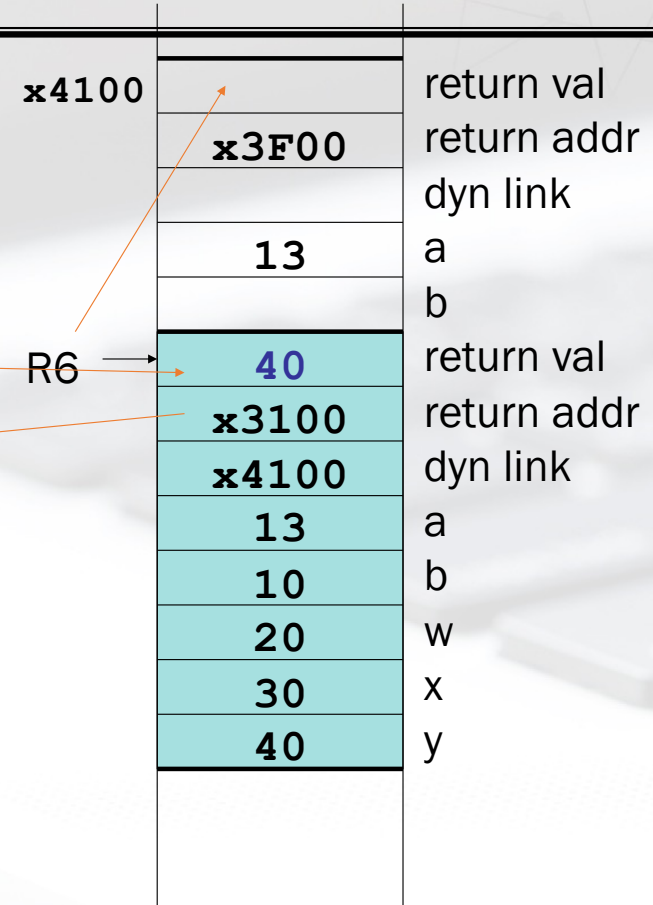
```
; store return address  
STR R7, R6, #1
```



# Ending the Callee Function



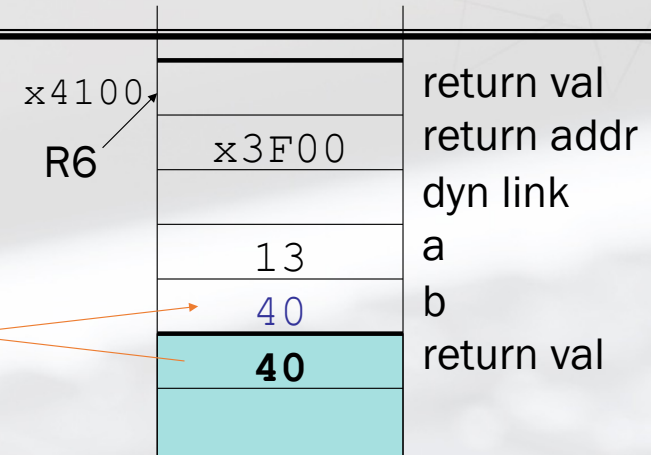
```
return y;  
  
; copy y into return value  
LDR R0, R6, #7  
STR R0, R6, #0  
  
; load the return address  
LDR R7, R6, #1  
  
; load the dynamic link  
  
; (pops the activation record)  
LDR R6, R6, #2  
  
; return control to caller  
RET
```



# Resuming the Caller Function



```
b = NoName (a, 10) ;  
  
; load return value  
LDR R0, R6, #5  
; store result into b  
STR R0, R6, #4
```





## Example: LC-3 Code for ToUpper

```
/* Function ToUpper:
 * If the argument is lower case,
 * return its upper case ASCII value */

char ToUpper(char inchar)
{
    int outchar = inchar;

    if ('a' <= inchar && inchar <= 'z')
        outchar = inchar - ('a' - 'A');

    return outchar;
}
```

```
ToUpper  STR  R7, R6, #1          ; save return addr
         LDR  R0, R6, #3          ; load parameter (inchar)
         STR  R0, R6, #4          ; initialize outchar
         LD   R1, neg_a           ; load -'a'
         ADD  R1, R0, R1          ; inchar - 'a'
         BRn  FALSE              ; br if inchar < 'a'
         LD   R1, neg_z           ; load -'z'
         ADD  R1, R0, R1          ; inchar - 'z'
         BRp  FALSE              ; br if inchar > 'z'
         LD   R1, neg_upper       ; load -('a' - 'A')
         ADD  R0, R0, R1          ; inchar - ('a' - 'A')
         STR  R0, R6, #4          ; store to outchar
FALSE    LDR  R0, R6, #4          ; load outchar
         STR  R0, R6, #0          ; store in result
         LDR  R7, R6, #1          ; load return address
         LDR  R6, R6, #2          ; load dynamic link
         RET
```

- Compile this function to LC-3 assembly language.



## Run-time Stack: Stack frame

### ■ Consider what has to happen in a function call:

- Caller must pass parameters to the callee.
- Caller must transfer control to the callee.
- Caller need to allocate space for the return value.
- Caller need to save the return address.
- Callee requires space for local variables.
- Callee must return control to the caller.
- Callee need to save the **frame pointer of the caller**

■ So, parameters, **return value**, return address, frame pointer, and local variables are stored on the stack.

# Run-time Stack: stack-like nature of function calls



```
1 int main(void)
2 {
3     int a;
4     int b;
5
6     :
7     b = Watt(a);    // main calls Watt first
8     b = Volt(a, b); // then calls Volt
9 }
10
11 int Watt(int a)
12 {
13     int w;
14
15     :
16     w = Volt(w, 10); // Watt calls Volt
17
18     return w;
19 }
20
21 int Volt(int q; int r)
22 {
23     int k;
24     int m;
25
26     :
27     return k;
28 }
```

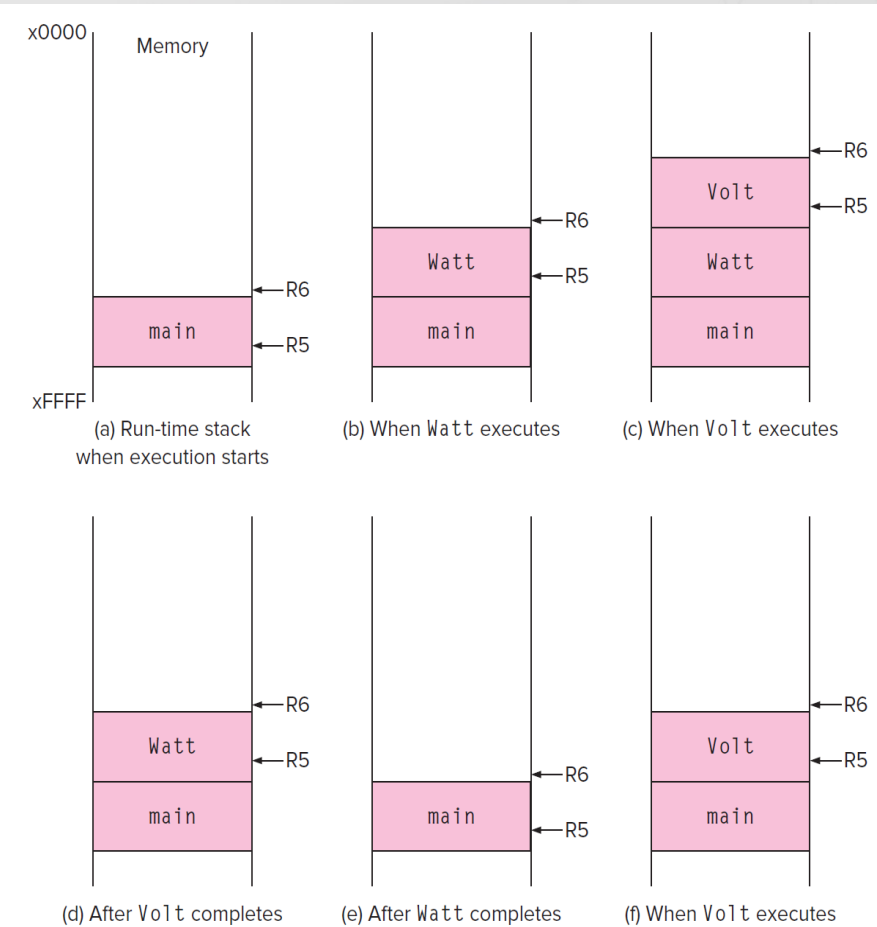


Figure 14.5 Several snapshots of the run-time stack while the program outlined in Figure 14.4 executes.

# Run-time Stack: frame pointer & stack pointer

- We need some easy way to access the data in each function' s stack frame and also to manage the pushing and popping of stack frames.
- For this, we will use R5 and R6.
  - R5 points to some **internal location** within the **stack frame** at the top of the stack—it may point to the base of the local variables for the currently executing function. We call it the **frame pointer (FP)**.
  - R6 always points to the very top of the stack. We call it the **stack pointer (SP)**.

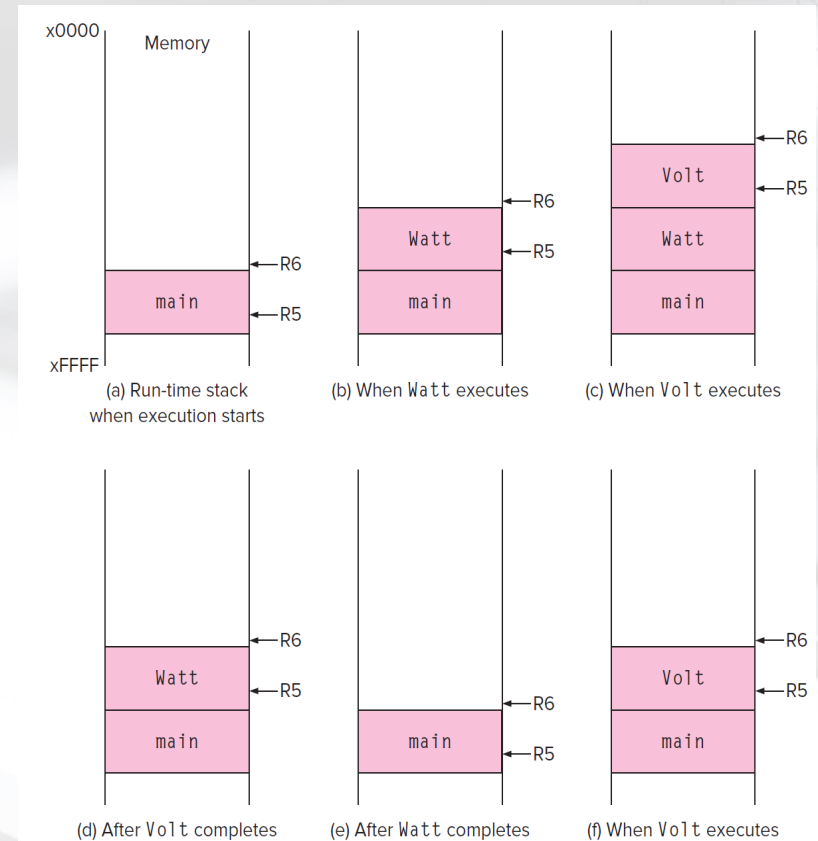


Figure 14.5 Several snapshots of the run-time stack while the program outlined in Figure 14.4 executes.





# Real Solution: Run-time Stack

- Instead of allocating the space for local variables **statically** (i.e., in a fixed place in memory), the space is allocated **once the function starts executing**.
- When the function returns to the caller, its space is **reclaimed** to be assigned later to another function.
- If the function is **called from itself**, the new invocation of the function will get its **own space** that is **distinct from** its other currently active invocations.
- The simple part
  - At the assembly level, a function is just a sequence of instructions that is called **using a JSR instruction**.
  - **The RET instruction** returns control back to the caller.
- The stickier issues
  - how arguments are passed
  - how the return value is returned,
  - and the allocation of local variables.
- The solution to these issues involves : **the run-time stack**.
- We need a way to “activate” a function when it is called. That is, when a function starts executing, its local variables must be allocated somewhere in memory. There are many possible solutions, and here we’ ll explore two options.

# Run-time Stack: Stack frame



**stack frame**  
**points to the**  
**base of the**  
**local**  
**variables for**  
**the currently**  
**executing**  
**function.**

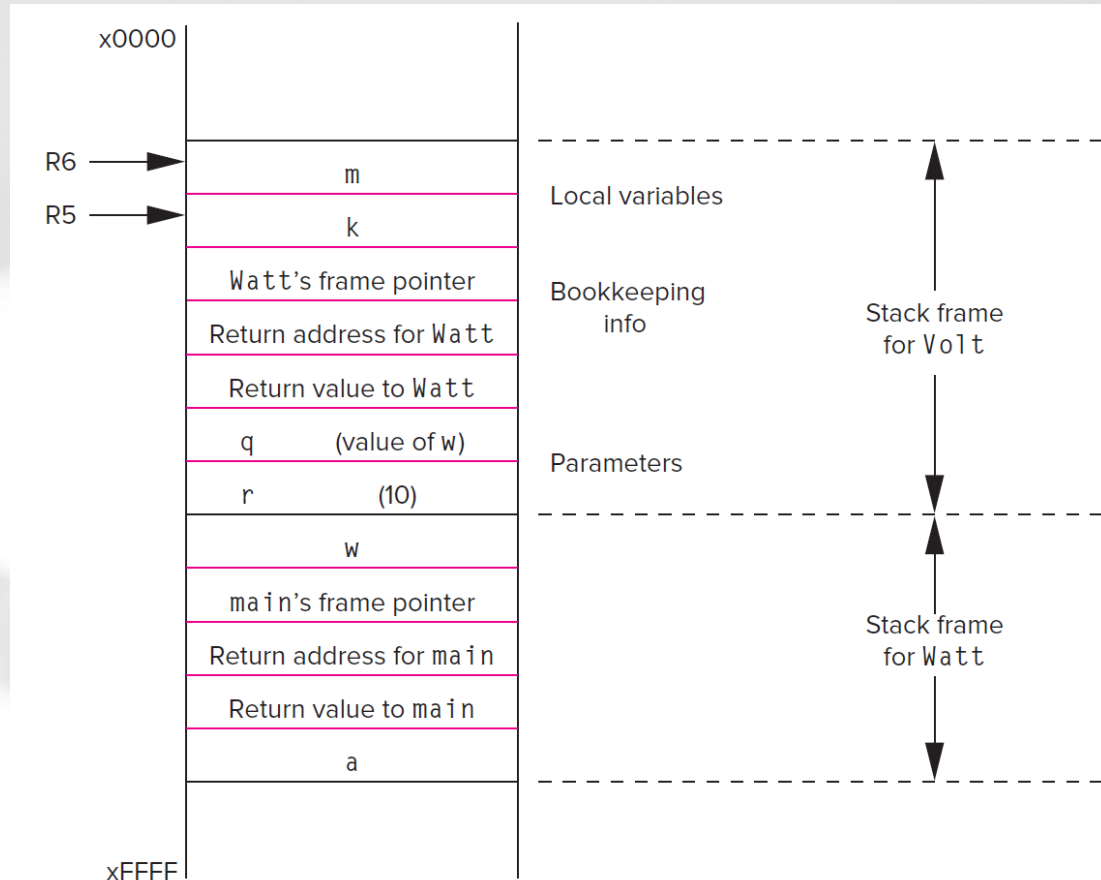
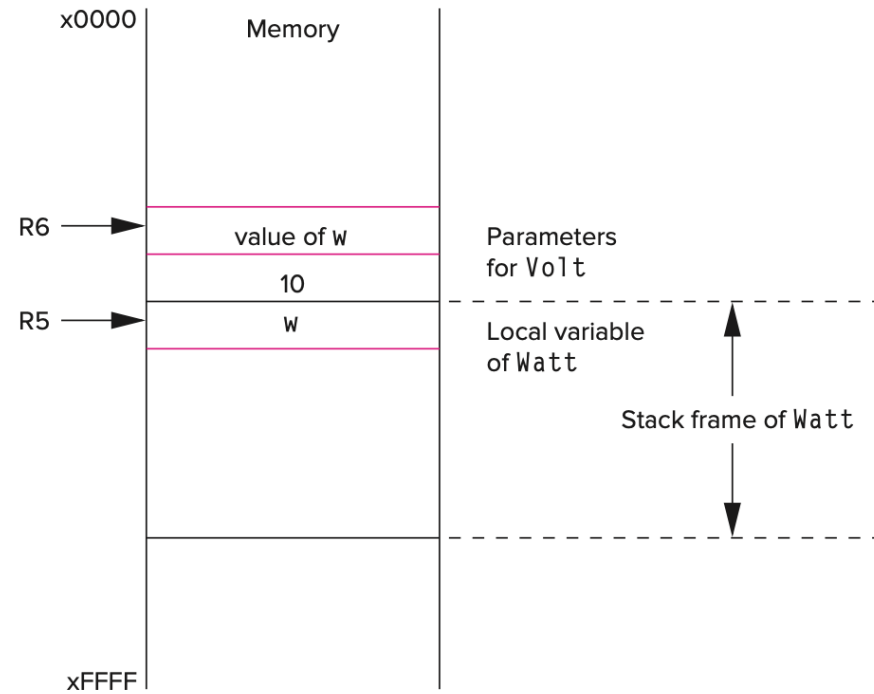


Figure 14.7 The run-time stack after the stack frame for `Volt` is pushed onto the stack.

# Run-time Stack: Stack frame

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

```
1 AND R0, R0, #0 ; R0 <- 0
2 ADD R0, R0, #10 ; R0 <- 10
3 ADD R6, R6, #-1 ;
4 STR R0, R6, #0 ; Push 10 onto stack
5
6 LDR R0, R5, #0 ; Load w
7 ADD R6, R6, #-1 ;
8 STR R0, R6, #0 ; Push w
9 JSR Volt
```



# Run-time Stack: Stack frame

```

1 Volt: ADD R6, R6, #-1 ; Allocate spot for the return value
2
3     ADD R6, R6, #-1 ;
4     STR R7, R6, #0 ; Push R7 (Return address)
5
6     ADD R6, R6, #-1 ;
7     STR R5, R6, #0 ; Push R5 (Caller's frame pointer)
8
9     ADD R5, R6, #-1 ; Set frame pointer for Volt
10    ADD R6, R6, #-2 ; Allocate memory for Volt's local variables

```

```

1 LDR R0, R5, #0 ; Load local variable k
2 STR R0, R5, #3 ; Write it in return value slot, which will always
3                ; be at location R5 + 3
4
5 ADD R6, R5, #1 ; Pop local variables
6
7 LDR R5, R6, #0 ; Pop the frame pointer
8 ADD R6, R6, #1 ;
9
10 LDR R7, R6, #0 ; Pop the return address
11 ADD R6, R6, #1 ;
12 RET

```

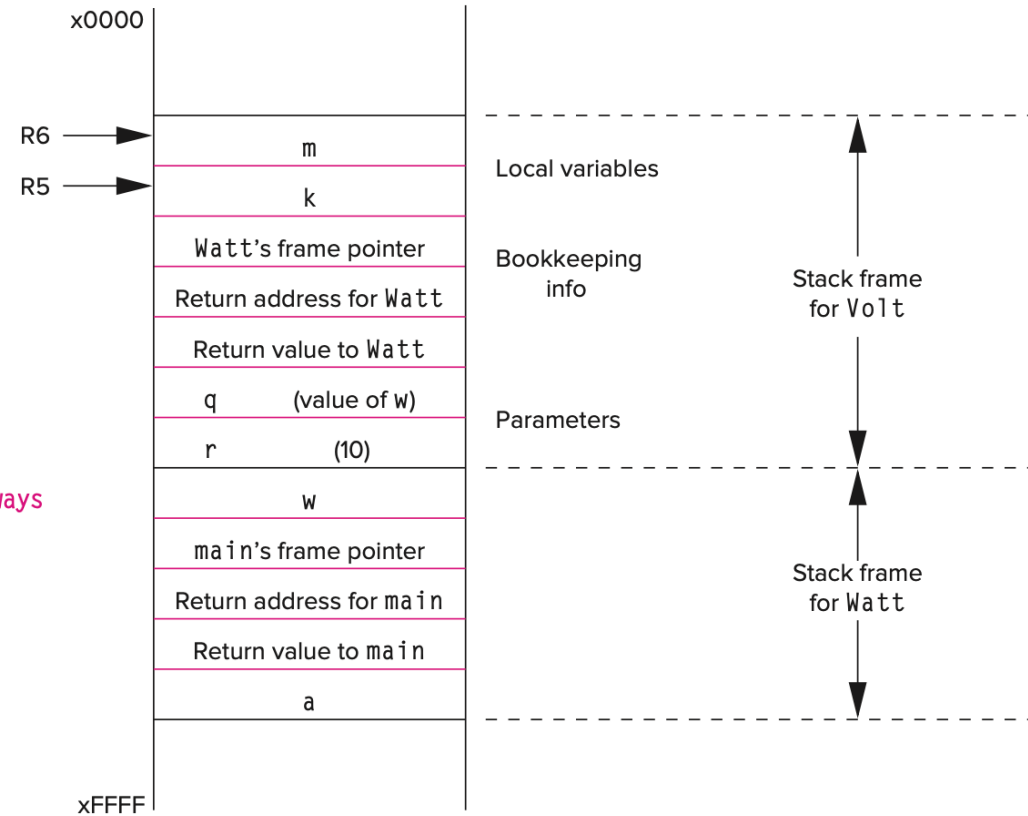


Figure 14.7 The run-time stack after the stack frame for Volt is pushed onto the stack.

# Run-time Stack: Stack frame



```

1 JSR Volt
2 LDR R0, R6, #0 ; Load the return value
3 STR R0, R5, #0 ; w = Volt(w, 10);
4 ADD R6, R6, #1 ; Pop return value
5
6 ADD R6, R6, #2 ; Pop arguments
    
```

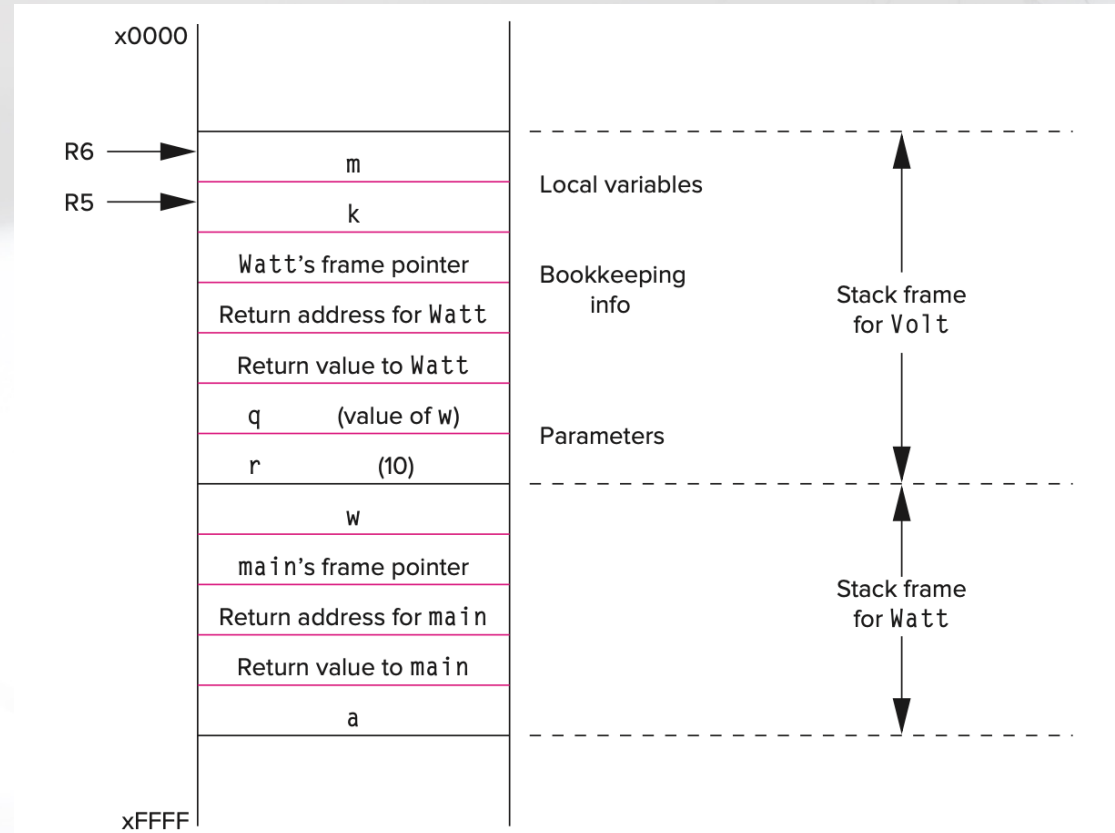


Figure 14.7 The run-time stack after the stack frame for Volt is pushed onto the stack.

```

1  int myadd(int x, int y) {
2      return x + y;
3  }
4
5  int add1(int x, int y) {
6      return myadd(x, y);
7  }
8
9  int add2(int x, int y) {
10     int z = 3;
11     return myadd(x, y) + z;
12 }
13
14 int add3(int x, int y) {
15     int z = 3;
16     int w = 4;
17     int v = 4;
18     return myadd(x, y) + z + w + v;
19 }

```

```

1  .globl> _myadd
2  _myadd:
3      sub>sp, sp, #16
4      str>w0, [sp, #12]
5      str>w1, [sp, #8]
6      ldr>w8, [sp, #12]
7      ldr>w9, [sp, #8]
8      add>w0, w8, w9
9      add>sp, sp, #16
10     ret
11
12     .globl> _add1
13     _add1:
14     sub>sp, sp, #32
15     stp>x29, x30, [sp, #16]
16     add>x29, sp, #16
17     stur> w0, [x29, #-4]
18     str>w1, [sp, #8]
19     ldur> w0, [x29, #-4]
20     ldr>w1, [sp, #8]
21     bl> _myadd
22     ldp>x29, x30, [sp, #16]
23     add>sp, sp, #32
24     ret

```

```

26     .globl> _add2
27     _add2:
28     sub>sp, sp, #32
29     stp>x29, x30, [sp, #16]
30     add>x29, sp, #16
31     stur> w0, [x29, #-4]
32     str>w1, [sp, #8]
33     mov>w8, #3
34     str>w8, [sp, #4]
35     ldur> w0, [x29, #-4]
36     ldr>w1, [sp, #8]
37     bl> _myadd
38     ldr>w8, [sp, #4]
39     add>w0, w0, w8
40     ldp>x29, x30, [sp, #16]
41     add>sp, sp, #32
42     ret

```

```

44     .globl> _add3
45     _add3:
46     sub>sp, sp, #48
47     stp>x29, x30, [sp, #32]
48     add>x29, sp, #32
49     stur> w0, [x29, #-4]
50     stur> w1, [x29, #-8]
51     mov>w8, #3
52     stur> w8, [x29, #-12]
53     mov>w8, #4
54     str>w8, [sp, #16]
55     str>w8, [sp, #12]
56     ldur> w0, [x29, #-4]
57     ldur> w1, [x29, #-8]
58     bl> _myadd
59     ldur> w8, [x29, #-12]
60     add>w8, w0, w8
61     ldr>w9, [sp, #16]
62     add>w8, w8, w9
63     ldr>w9, [sp, #12]
64     add>w0, w8, w9
65     ldp>x29, x30, [sp, #32]
66     add>sp, sp, #48
67     ret

```

### 14.7 Following is the code for a C function named Bump.

```
int Bump(int x)
{
    int a;
    a = x + 1;
    return a;
}
```

- a. Draw the stack frame for Bump.
- b. Write one of the following in each entry of the stack frame to indicate what is stored there.
  - (1) Local variable
  - (2) Argument
  - (3) Address of an instruction
  - (4) Address of data
  - (5) Other
- c. Some of the entries in the stack frame for Bump are written by the function that calls Bump; some are written by Bump itself. Identify the entries written by Bump.

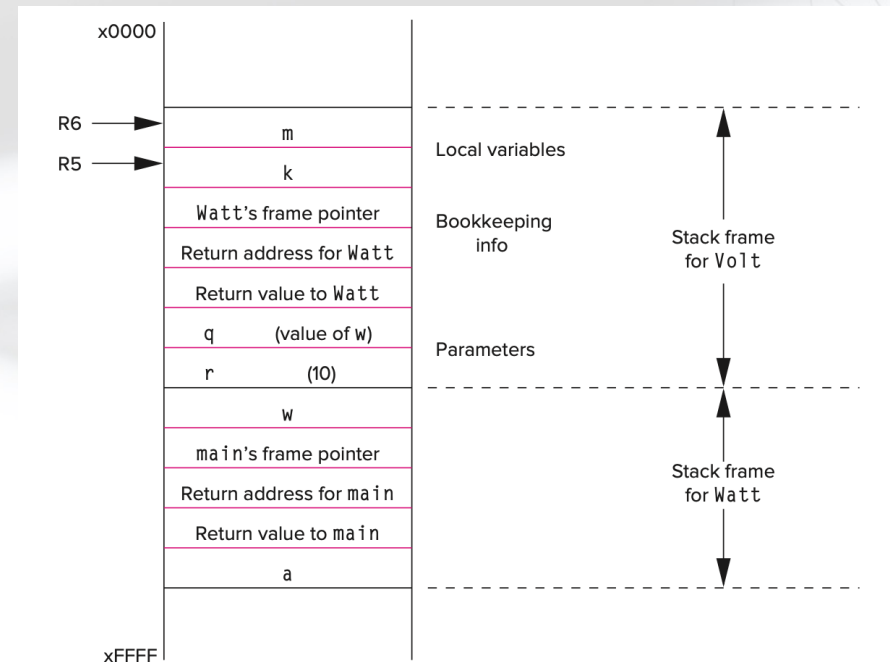


Figure 14.7 The run-time stack after the stack frame for Volt is pushed onto the stack.

**14.15** The following C program is compiled into LC-3 machine language and loaded into address x3000 before execution. Not counting the JSRs to library routines for I/O, the object code contains three JSRs (one to function *f*, one to *g*, and one to *h*). Suppose the addresses of the three JSR instructions are x3102, x3301, and x3304. Also suppose the user provides 4 5 6 as input values. Draw a picture of the run-time stack, providing the contents of locations, if possible, when the program is about to return from function *f*. Assume the base of the run-time stack is location xEFFF.

```
#include <stdio.h>
int f(int x, int y, int z);
int g(int arg);
int h(int arg1, int arg2);

int main(void)
{
    int a, b, c;

    printf("Type three numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    printf("%d", f(a, b, c));
}

int f(int x, int y, int z)
{
    int x1;

    x1 = g(x);
    return h(y, z) * x1;
}

int g(int arg)
{
    return arg * arg;
}

int h(int arg1, int arg2)
{
    return arg1 / arg2;
}
```

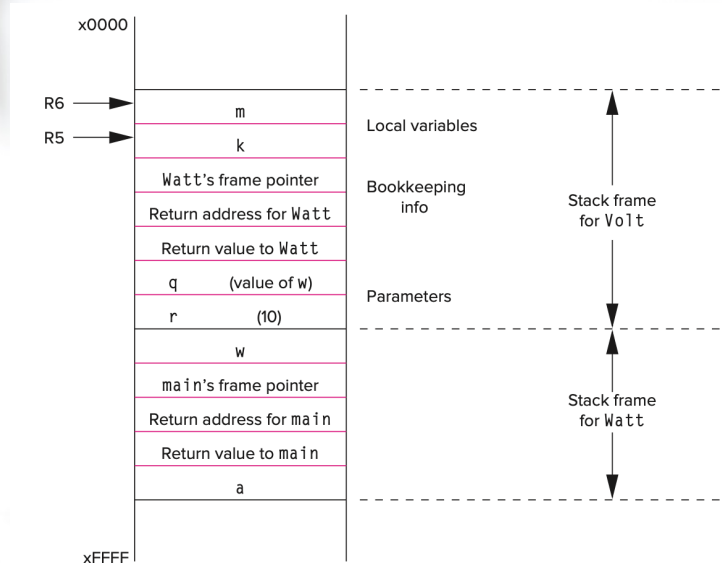


Figure 14.7 The run-time stack after the stack frame for *Volt* is pushed onto the stack.