Mc Graw Hill Education

Because learning changes everything.®

From Bits and Gates to C and Beyond

Variables and Operators

Chapter 12

Bitwise Operators

Table 12.2 Bitwise Operators in C

Operator symbol	Operation	Example usage
~	bitwise NOT	~x
&	bitwise AND	x & y
|	bitwise OR	x | y
^	bitwise XOR	x ^y
«	left shift	x « y
»	right shift	x » y

C bitwise operators apply a logical operation across the individual bits of its operands.

Only applies to integers, or to smaller types that can be treated as integer (`char` or `bool`). The operation is applied to the 2's complement binary representation of the integer value.

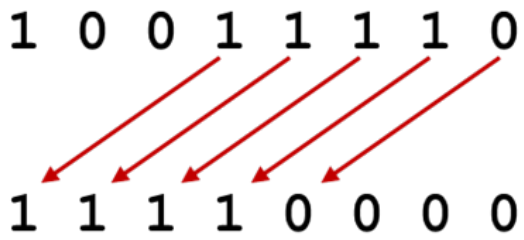
Expression	Operation	Result
`0x1234 | 0x5678`	OR	`0x567c`
`0x1234 & 0x5678`	AND	`0x1230`
`0x1234 ^ 0x5678`	XOR	`0x444c`
`~0x1234`	NOT	`0xedcb`
`1234 & 5678`	AND	`1026`

Hex and decimal are both integers. Don't let the notation confuse you. The same binary operation is performed, no matter whether you write the values in base-10 or base-16.

Shift Operators

The << and >> operators are used to perform a bitwise shift of an integer value, to the left (<<) or right (>>). The left operand is the value to be shifted, and the right operand is the number of bit positions to shift.

For the purposes of illustration, let's assume that integer values are 8 bits. Evaluate `0x9e << 3`



Each bit is shifted left by three positions. The leftmost bits are "shifted off" and we fill in from the right with zeroes. Result: `0xF0`

Access the text alternative for slide images.

Using Bitwise Operators

We've encountered the bitwise operators before, in Chapter 2, but let's review how they are typically used:

AND: clearing bits

- Create a mask with 0's for the bits you want to clear, 1's for the bits you want to keep.

OR: setting bits

- Create a mask with 1's for the bits you want to set, 0's for the bits you want to keep.

XOR: flipping bits

- Create a mask with 1's for the bits you want to flip, 0's for the bits you want to keep.

Machine-independent (portable) C code should not assume that you know the size of an `int` or any other data type.

Shift operators are useful to move bits where you need them to be. You might want to shift the mask, or you might want to shift the value being masked...

C Program

```
#include <stdio.h>   // need this to use printf and scanf
int main(void)
{
   int amount;  // number of bytes to transfer -- get from user
   int rate;    // avg network transfer rate -- get from user
   int time;    // transfer time in seconds
   int hours, minutes, seconds;  // convert to hr:min:sec

   // Get input from user:  file size (bytes) and network speed (bytes/sec)
   printf("How many bytes of data to be transferred? ");
   scanf("%d", &amount);
   printf("What is the transfer rate (in bytes/sec)? ");
   scanf("%d", &rate);

   // Calculate time in seconds
   time = amount / rate;    // truncates any fractional part

   // Use arithmetic operators to calculate hours, mins, secs
   hours = time / 3600;             // throws away remainder
   minutes = (time % 3600) / 60;    // remainder, converted to minutes
   seconds = ((time % 3600) / 60);

   // Output results
   printf("Time : %dh %dm %ds\n", hours, minutes, seconds);
}
```

Translating C Code to LC-3 Assembly Language

Before we learn about more features of the C language, let's consider how these basic C statements would be translated into LC-3 instructions.

What is compiler's task?

1. Allocate memory for variables in a systematic way.
   Will build a **symbol table** to keep track of variable type, size, location.
2. Generate instruction sequences that carry out the computations specified by operators and statements.

For this class, we will compile "by hand" to get a sense of how these translations occur.

Symbol Table

For the LC-3 assembler, the symbol table tracked the memory address for each label in the program.

For the C compiler, we need to track the type and scope information. The type tells us how big each variable's memory object should be. The scope tells us where the memory object should be allocated.

Identifier	Type	Location (as an offset)	Scope	Other Info...
amount	int	0	main	...
hours	int	−3	main	...
minutes	int	−4	main	...
rate	int	−1	main	...
seconds	int	−5	main	...
time	int	−2	main	...

This is a symbol table generated for the sample program we just wrote.

There is one entry for each declared variable.,

The offset will be explained shortly.

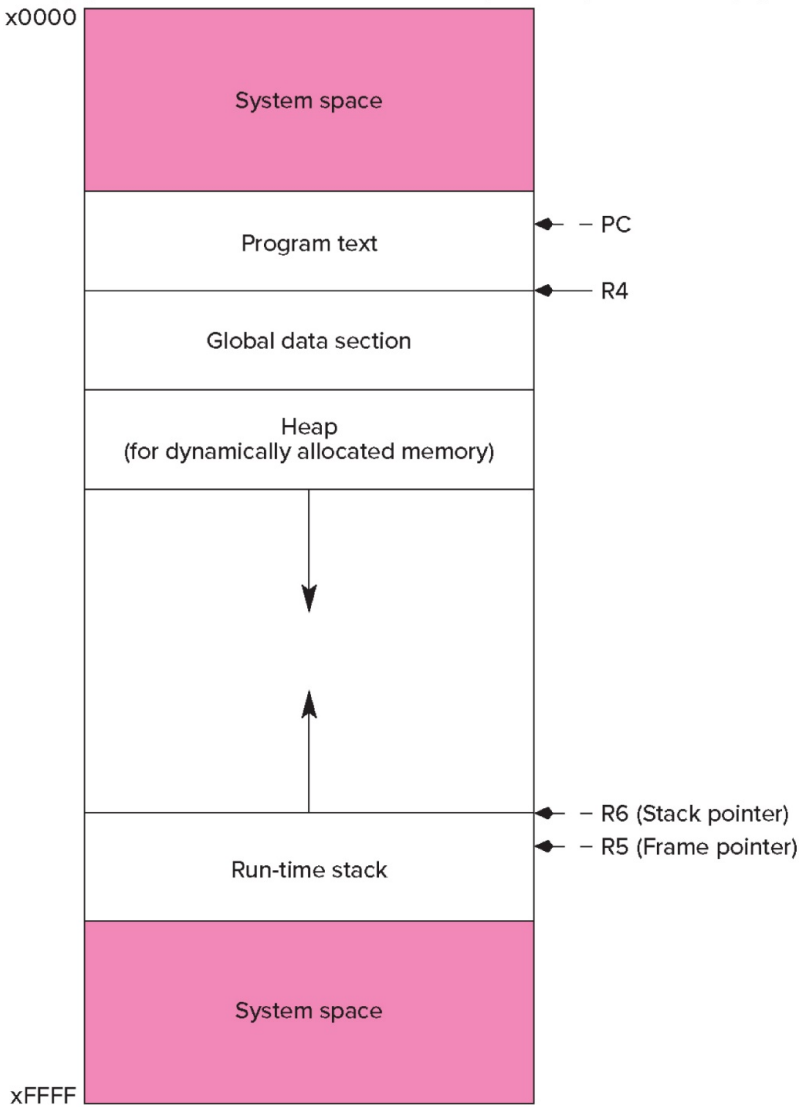
LC-3 Memory Map

The LC-3 operating system reserves some of the memory address space for trap vectors, service routine code, and memory-mapped I/O.

Program instructions will be placed in the "program text" section.

Global variables are allocated next. R4 is set to the first allocated address for globals.

Local variables are stored on the run-time stack. R5 points to the local variables of the currently-executing function.



Access the text alternative for slide images.

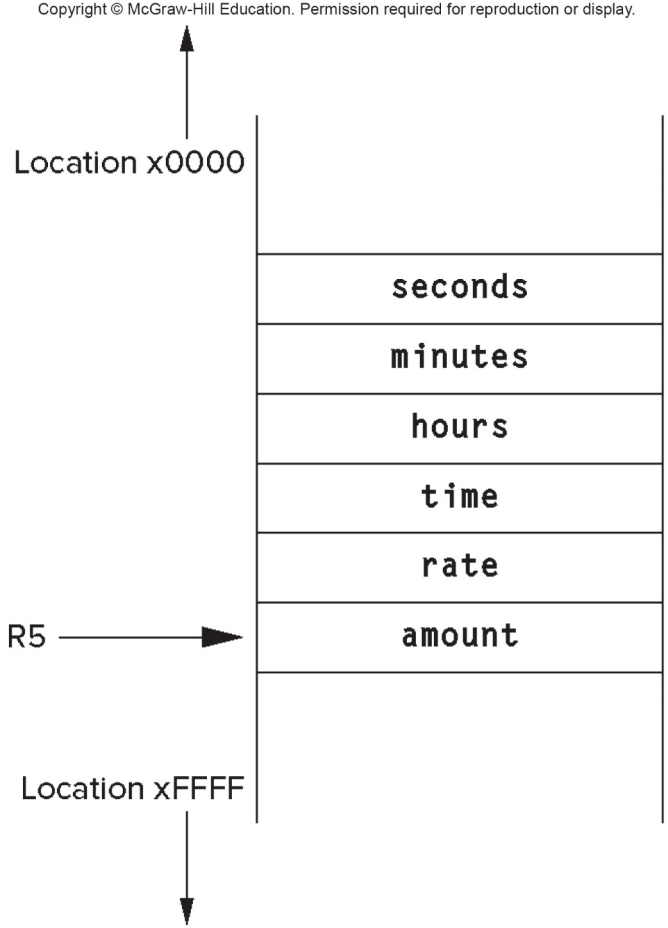
Stack Frame and Local Variables

When the program is running, the run-time stack is used to keep track of state for active functions (subroutines). Each function that has been called gets a **stack frame** or **activation record** allocated on the stack.

Part of the stack frame is used to hold local variables, as shown below.

R5 points to the region where local variables are stored, and variables are allocated in the order in which they are declared.

The symbol table offset is the amount to be added to R5 to get the address of each variable. For example, the offset for `amount` is 0, and the offset for `time` is -2.



Access the text alternative for slide images.

Generating LC-3 Instructions

The symbol table corresponds to the stack frame, as shown.

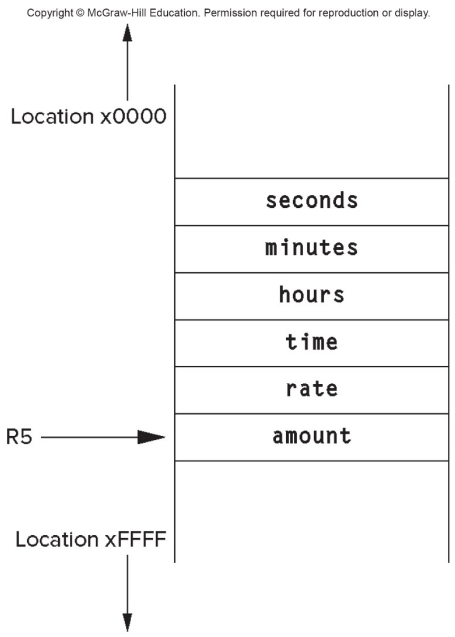
If we need to load the value of the `amount` variable into R0...

```
LDR R0, R5, #0
```

Suppose we've calculated the value to be stored into the hours variable, and the value is in R3...

```
STR R3, R5, #-3
```


Identifier	Type	Location (as an offset)	Scope	Other Info...
amount	int	0	main	...
hours	int	−3	main	...
minutes	Int	−4	main	...
rate	int	−1	main	...
seconds	int	−5	main	...
time	int	−2	main	...



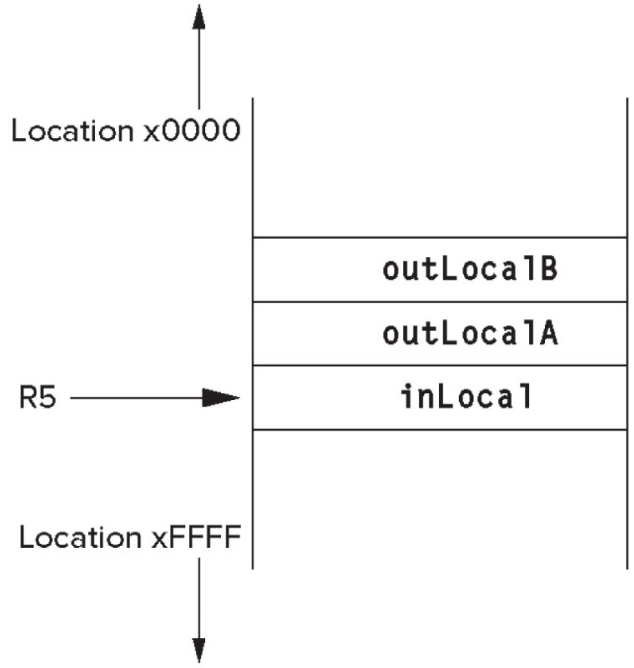
A More Complete Example


```
 1  #include <stdio.h>
 2
 3  int inGlobal;   // inGlobal is a global variable.
 4                  // It is declared outside of all blocks
 5
 6  int main(void)
 7  {
 8     int inLocal = 5;   // inLocal, outLocalA, outLocalB are all
 9     int outLocalA;     // local to main
10     int outLocalB;
11
12     // Initialize
13     inGlobal = 3;
14
15     // Perform calculations
16     outLocalA = inLocal & ~inGlobal;
17     outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);
18
19     // Print results
20     printf("outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);
21  }
```

Symbol Table and Stack Frame

Identifier	Type	Location (as an offset)	Scope	Other info...
inGlobal	int	0	global	...
inLocal	int	0	main	...
outLocalA	int	–1	main	...
outLocalB	int	–2	main	...

(a) Symbol table



(b) Activation record for main

The variable `inGlobal` is not in the stack frame because it is not local to the main function. Its offset is relative to R4.

Access the text alternative for slide images.

LC-3 Code for Each C Statement 1

```
; int inLocal = 5;
; We don't normally generate code for declarations, but
; but we need to initialize the variable.
AND R0, R0, #0   ; create value 5 to put in variable
ADD R0, R0, #5
STR R0, R5, #0   ; inLocal is local, offset = 0

; inGlobal = 3;
AND R0, R0, #0   ; create value 3 to put in variable
ADD R0, R0, #3
STR R0, R4, #0   ; inGlobal is global, offset = 0

; outLocalA = inLocal & ~inGlobal;
LDR R0, R5, #0   ; get value of inLocal
LDR R1, R4, #0   ; get value of inGlobal
NOT R1, R1       ; bitwise NOT of R1 (~inGlobal)
AND R2, R0, R1   ; perform AND
STR R2, R5, #-1  ; store result to outLocalA (local, offset = -1)
```

LC-3 Code for Each C Statement 2

```
; outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal)
LDR R0, R5, #0     ; get inLocal
LDR R1, R4, #0     ; get inGlobal
ADD R0, R0, R1     ; calculate inLocal + inGlobal

LDR R2, R5, #0     ; get inLocal
LDR R3, R5, #0     ; get inGlobal
NOT R3, R3
ADD R3, R3, #1
ADD R2, R2, R3     ; calculate inLocal - inGlobal

NOT R2, R2         ; calculate -(inLocal - inGlobal)
ADD R2, R2, #1
ADD R0, R0, R2     ; calculate final value
STR R0, R5, #-2    ; store to outLocal B (local, offset = -2)
```