Lab Manual Section 1: The C Programming Language

1-1: Shifts and Masks

Objectives

1. Understand how to use shifts to do multiplication and division by two
2. Understand differences between a logical and a bitwise AND
3. Understand how arrays work in C, how to use them, and how they are stored in memory
4. Be able to use bitwise operators on bit fields of various lengths

Helpful Information

Arrays

An array in the C programming language is handled in the same fashion as one in Java. To instantiate an array, you declare the array type and size. For example:

```
double myArray[100];
```

Bitwise Operations

C has six operators intended to directly manipulate bits within integer variables:

Operator	Operation
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	left shift
>>	right shift
~	1's complement (invert all bits)

Bitwise AND

For the purposes of this lab, we are only concerned with the bitwise AND operator. A bitwise AND takes two binary-represented numbers of equal binary length and performs the logical AND operation bit by bit. For example:

```
 00011001
&11011000
=00011000
```

Bit Masking

Bit masking is the use of the bitwise AND operator to obtain, or "mask," only the specific bits we are

interested in. Consider the case in which you are interested in determining whether a number is odd or even. You would only need to check if the least-significant bit is a one or a zero. If we were operating on a 4-bit integer, then the mask would be `0001`. ANDing it with a 6 in binary would yield:

```
 0110
&0001
=0000
```

Since we got a zero and not a one, the number is even.

Bit Shifting

Another bitwise operation that is very useful is bit shifting. You can shift to the left or to the right. There are two types of shifts: logical shifts and arithmetic shifts. In a right arithmetic shift, the most-significant bit is preserved to maintain the sign of the number. In logical shifts of either direction, no bits are preserved; instead, the bits shifted out are replaced with zeros. A left arithmetic shift is the same as a left logical shift. In C, all shifts of unsigned numbers are logical and all shifts of signed numbers are arithmetic. A right shift will divide a number by 2^{-n}, where n is the number of shifts. For example, if we had the number 20 and we did a right shift by 3 bits (`20 >> 3`), we would obtain $20 * 2^{-3}$: $20 / 2^3 = 20 / 8 = 2.5 = 2$. Examples of right shifting follow:

Operation	Binary		Decimal
Arithmetic right shift	Signed	`1110 1100 >> 1 = 1111 0110`	`-20 >> 1 = -10`
Logical right shift	Unsigned	`1000 0010 >> 1 = 0100 0001`	`130 >> 1 = 65`

A shift left has the opposite effect. Instead of dividing by powers of 2, we are multiplying by powers of 2. If we have a number, say, 10, and we shift to the left n times, it is equivalent to obtaining $10 * 2^n$, where n is the number of shifts. For example, say we have the number 10 and we shift it left 3 times (`10 << 3`). The operation is equivalent to obtaining $10 * 2^3 = 10 * 8 = 80$.

Operation	Binary	Decimal
Left shift	`0000 1010 << 3 = 0101 0000`	`10 << 3 = 80`

Assignment

The following program takes in an integer and prints out its binary representation:

```
#include <stdio.h>

/* Prototype functions */
void printBinary(short);
char asciiDigit(int);

int main(void)
{
    printBinary(255);

    return 0;
}

void printBinary(short number)
{
    /* Shorts are two bytes on the x86 */
    const char LENGTH_OF_SHORT = 16;
```

```
    char digits[LENGTH_OF_SHORT];

    int i;
    char digit;

    for (i = 0; i < LENGTH_OF_SHORT; i++){
        /* Obtain the least-significant bit of number */
        digit = number & 1;

        /* Right-shift the number once to evaluate the next bit */
        number = number >> 1;

        /* Store the result in the array */
        digits[(LENGTH_OF_SHORT - 1) - i] = asciiDigit(digit);
    }

    for (i = 0; i < LENGTH_OF_SHORT; i++){
        /* Print the digit */
        putchar(digits[i]);
    }
}
```

Write a function in C that takes a `short` argument and converts it to its hexadecimal representation. Your program must use shifts and masks to accomplish this and you may not use the standard C libraries (for example, `stdio.h`)—however, you may incorporate code you wrote for your previous lab assignments.

Be sure to include a test program (in a separate file) that demonstrates that your "print hex" function works correctly. Your Makefile should compile both source files and link them into a single executable file.

1-0: ASCII **1-1: Shifts and Masks** 1-2: Pointers