COMP1521

W5 - Binary and Bitwise Operations

Overview

Admin

Binary and other bases (Q2)

Bitwise operations (Q3)

How we use bitwise operations (Q4, Q7)

Admin

Assignment 1 is due this Friday

W4 weekly test is due this Thursday

This weeks lab (W5 lab) will be due Monday W7 (extended due to flex week)

Help sessions are running daily up until the assignment deadline, in person in K17 Room 113 and online also.

Hopefully forum posts will be seen to by a small army of tutors

Binary and other bases

What is a number?

A number is something that represents a value.

When we are looking at different bases, there are multiple numbers we could use to represent the same value.

For example, consider the number 23, which represents the value “23” (decimal).

The number 0x17 also represents the value “23” (hexadecimal).

The number 027 also represents the value “23” (octal).

The number 0b10111 also represents the value “23” (binary).

What do all of those “numbers” really mean?

Decimal:

23 = 2 * 10^1 + 3 * 10^0 = “23”

Hexadecimal:

0x17 = 1 * 16^1 + 7 * 16^0 = “23”

Octal:

027 = 2 * 8^1 + 7 * 8^0 = “23”

Binary:

0b10111 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = “23”

What were those prefixes?

No prefix = decimal

0x prefix = hexadecimal

0 prefix = octal

0b prefix = binary

How do computers store values?

We typically write value in our programs using their decimal “number”.

However, when they are stored in a computer, because computers use all 1s and 0s, computers store values as their binary numbers.

This line of C:
int x = 23;

Would cause some of the memory in your computer to load this binary number:

RAM (addr 0x10010000) = 00000000 00000000 00000000 000101111

They both still represent the same value; they MEAN the same thing

It’s just a different way of storing it.

Representing numbers in binary

For conversion, effective strategy is to just keep taking away the largest power of 2 that fits in the number.

Like for 59:

-32 = 27
-16 = 11
-8 = 3
-4? no
-2 = 1
-1 = 0, done.

So the result is 111011.

More effective conversion (with a snake)

Open your terminal and type “python3”.

Type a non-decimal number (eg. 0b100100 or 0x123) and it will echo the decimal equivalent.

Use the corresponding function bin(), hex() or oct() with an input number at it will convert it to that base (eg. bin(20) or hex(0b111000) or bin(0x123))

Type ctrl+d to escape.

Bits and Bytes

A bit is a single binary “digit”, in the same way that a “digit” is a single decimal digit.

A byte is a collection of 8 bits. A byte is a nice chunk of memory for computers, because the size of types (like char, int) generally fits nicely into a multiple of bytes.

For example, a register in MIPS stores 32 bit numbers (or equivalently it has 4 bytes of storage).

So, a register just stores a 32 digit binary number, (32 bit).

Tutorial Q2

2. How can you tell if an integer constant in a C program is decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2)?

   Sidenote: do you think this is good language design?

   Language trivia: what base is the constant **0** in C?

   Show what the following decimal values look like in 8-bit binary, 3-digit octal, and 2-digit hexadecimal:

   a. `1`
   b. `8`
   c. `10`
   d. `15`
   e. `16`
   f. `100`
   g. `127`
   h. `200`

   How could I write a C program to answer this question?

Bitwise operations

sidenote

For the following slides we are treating 0 = false and 1 = true

Logical AND/OR/NOT

Below are the logical AND/OR/NOT operations that we are familiar with:

Logical &&	0	1
0	0	0
1	0	1

Logical ||	0	1
0	0	1
1	1	1

Logical !	0	1
	1	0

Bitwise?

It just does the regular AND/OR operation for every bit in the number

So for example,

00001111 & 00000011 = 00000011 (for every bit, perform logical AND)

00001111 | 00000011 = 00001111 (for every bit, perform logical OR)

~ 00001111 = 11110000 (for every bit, perform logical NOT)

And also XOR

XOR is the weird bitwise op that rarely gets used (Exclusive OR).

In practice, it’s mainly used for encryption.

This is the logical XOR:

Logical ^	0	1
0	0	1
1	1	0

So bitwise XOR is like this:

10101111 ^ 11110000 = 01011111

And also bitwise SHIFT operations

The left shift << and right shift >> operators will “shift” the bits in a number in a certain direction a certain number of times.

So, if we take 0100 << 1, we get 1000

Or if we take 0010 >> 1, we get 0001

Bits that get shifted “over the edge” of the number are erased!!

The new bits that enter the number will typically be zeroes, as long as you are using uint[xx]_t types (we will talk about signed/unsigned next week).

Bitwise SHIFT for multiplication and division

0001 << 1 = 0010

1 << 1 = 2

<< 1 is effectively multiplication by 2, since each bit moves to the left where it represents a value 2x larger.

0010 >> 1 = 0001

2 >> 1 = 1

>> 1 is effectively integer division by 2, for a similar reason.

If we >> or << x where x > 1, then it is repeated division/multiplication by 2 (or individual multiplication/division by 2^x)

Trick question

What types in C can you use bitwise operations on?

Only on binary types??

int?

char?

*int (pointer)?

Other types?

Tutorial Q3

3. Assume that we have the following 16-bit variables defined and initialised:

```
uint16_t a = 0x5555, b = 0xAAAA, c = 0x0001;
```

What are the values of the following expressions:

a. `a | b` (bitwise OR)
b. `a & b` (bitwise AND)
c. `a ^ b` (bitwise XOR)
d. `a & ~b` (bitwise AND)
e. `c << 6` (left shift)
f. `a >> 4` (right shift)
g. `a & (b << 1)`
h. `b | c`
i. `a & ~c`

Give your answer in hexadecimal, but you might find it easier to convert to binary to work out the solution.

Use of bitwise operations

What’s the point?

So far these probably all seem kinda useless. Shift could be used for division and multiplication, but really we should just use the dedicated div(/) and mul(*) operators for that.

The main use of these operators concerns reading and writing the bits inside numbers.

We can use bitwise operators to write and read certain bits of a number, which can allow us to store information in a very compact manner (memory efficient!).

How can we write 1 bits to a number

Generally, we can use bitwise OR (|) to write 1 bits to a number.

If we consider some unknown target binary number (????????).

And we would like to “write” a 1 to the first bit, we can use the | operator like so: 10000000 | ????????

= 1???????

And we have effectively “written” a 1 to the first bit.

How can we write 1 bits to a number?

We could also have used some shift operators to generate the “10000000” value.

For example, 1 << 7 would have worked well.

00000001 << 7 = 10000000

How can we read bits from a number?

Generally, we can use bitwise AND (&) to read bits from a number.

Consider some unknown binary number we would like to read from (????????),

Assuming we want to read the leftmost 4 bits.

If we perform an operation like:
11110000 & ???????? = ????0000

We can effectively “read” 4 bits from the number.

How can we read bits from a number?

After reading my value “????0000” I might like to check what it is equal to.

But if I check it now, it will appear 16 times larger due to those 0s on the right.

(This is like if a look at the number 2300, it is 100 times larger than 23 due to the extra zeros on the right)

So, maybe we can use a bitwise shift operator (>>) to move it to the right:

????0000 >> 4 = 0000????

How can we write 0 bits to a number?

We can use bitwise AND (&) to write 0 bits, by reading all of the bits except those we wish to set to 0.

For example, to set the last bit of a mystery 8 bit number ???????? to 0:

???????? & 11111110

= ????????0

Practical use

One very notable example of where bits are “packed” into a number to compactly represent information is a MIPS instruction.

CPU Arithmetic Instructions back to top

✓	ADD	R_d, R_s, R_t	$R_d = R_s + R_t$	INTEGER OVERFLOW	000000ssssstttttddddd00000100000

On the right, see that we use a 32 bit number (size of an int in C) and we compactly store information about s (a source register from 0-31), t (a temp register from 0-31), d (a destination register from 0-31).

Because each register is 0-31 we can represent it using 5 binary digits (5 bits)

Tutorial Q4

4. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

Tutorial Q5

5. You've been hired to write a driver to control a printer/scanner combo. The printer communicates through a series of flags that can be either 0 or 1:
   - `NO_INK` (read/write): The printer sets this flag when it's out of ink. You must unset it when the ink is replaced.
   - `COLOUR` (write): You set this flag to tell the printer to scan/print in colour.
   - `SELECT_PRINT` (write): You set this flag to select printing mode.
   - `SELECT_SCAN` (write): You set this flag to select scanning mode.
   - `START` (read/write): You set this flag to do the selected task (print or scan). The printer will unset this when it's finished.

   Don't worry about how the actual file you're printing/scanning gets to and from the printer. We're only interested in the control signals.

   One way to implement this is to have a variable for each flag:

```
int NO_INK = 0;       // Ink levels OK
int COLOUR = 1;       // Printing/scanning in colour
int SELECT_PRINT = 1; // Print mode selected
int SELECT_SCAN = 0;  // Scan mode not selected
int START = 0;        // Printing/scanning hasn't started
```

   However, this is a waste of space, and in hardware, every bit matters! Each integer takes up 32 bits, but we only need to store 1 bit of information for each flag. Instead, we can pack all the flags into a single 8 bit (1 byte) integer, and use the individual bits to represent the flags:

```
printerControl = 0 0 0 0 0 0 0 0
                 ^ ^ ^ ^ ^ ^
                 | | | | |
                 | | | | L [NO_INK]
                 | | | L [COLOUR]
                 | | L [SELECT_PRINT]
                 | L [SELECT_SCAN]
                 L [START]
```

   The most significant 3 bits are unused.

In C, that would look like:

```
#include <stdint.h>

uint8_t printerControl = 0; // 0b 0000 0000

// Whether the printer is out of ink
#define NO_INK (0x1)       // 0b 0000 0001
// Whether to print/scan in colour
#define COLOUR (0x2)       // 0b 0000 0010
// Select print mode
#define SELECT_PRINT (0x4) // 0b 0000 0100
// Select scan mode
#define SELECT_SCAN (0x8)  // 0b 0000 1000
// Start print/scan
#define START (0x10)       // 0b 0001 0000
```

For the following questions, assume the C code above is included globally. Don't change any other flags other than the ones specified.

Write a function:

a. that prints (to terminal) whether the printer is out of ink.
b. that tells the printer the ink has been replaced.
c. to use colour and select scan mode. Assume no mode has been selected yet.
d. that toggles between print and scan mode. Assume 1 mode is already selected.
e. *(Extension question)* to start printing/scanning. It should:
   i. check that one (and only one) mode is selected
   ii. check there's ink if printing.
   iii. print (to terminal) what it's doing and any error messages.
   iv. wait until the printing/scanning is finished and print a 'finished' message. Since there isn't an actual printer on the other side, a correct implementation of this will infinite loop and never print 'finished'.