CS-200

Computer Organization and Assembly Language

Bitwise Operators

Bitwise Operators

- C and C++ allow operations on bits
- Actually, they work on byte boundaries
- The data is always assumed to be unsigned integer
- Operations on other types may have unpredictable results.

Bitwise Operators

- Why bother with bitwise operations?
  - Data compression
  - Encryption
  - Speed
- But there's a price in complexity, flexibility, and maintainability

Bitwise Operators

SHIFT OPERATORS

Shift Operators

- Left Shift (<<) moves all bits to the left, padding with 0s.
- << is equivalent to multiplication by powers of 2
- But the overflow is lost, so it's not true multiplication

```
int num = 4;                // 00000100
int result;                 // ??
result = num << 3;          // 00100000 = 32
```

Shift Operators

- Right Shift (>>) moves all bits to the right.
- Unsigned numbers are padded with 0s
- Signed numbers may preserve sign.
- Can be thought of as division by powers of 2
- Same limitations as multiplication

Bitwise Operators

COMPLEMENT

Complement Operator

- Complement (~) flips all bits
- Not the same as logical NOT (!)
  - !00011000 = 0, ~00011000 = 11100111
  - !0 = 1, ~0 = 11111111
- Useful for finding max unsigned integer value

```
long int maxval = ~0;      // ?? depends on system
```

- Good for building masks (more later)

Bitwise Operators

LOGICAL OPERATORS

Logical Operators

- AND (&) – not to be confused with &&
- Operates on bit pairs in different words

```
int num1 = 107;
int num2 = 54;
int result = num1 & num2;        // result =  34
```

01101011 &

00110110 =

00100010

Logical Operators

- OR (|) – not to be confused with ||
- Operates on bit pairs in different words

```
int num1 = 107;
int num2 = 54;
int result = num1 | num2;          // result =  127
```

01101011 &

00110110 =

01111111

Logical Operators

- XOR (^) – no corresponding operator
- Operates on bit pairs in different words

```
int num1 = 107;
int num2 = 54;
int result = num1 ^ num2;          // result =  93
```

01101011 &

00110110 =

01011101

Bitwise Operators

COMPRESSION

Compression

- Not all data needs a whole byte to store
- How many bits to store letters and numbers?
- 26 + 10 = 36 is less than 7 bits
- Even if we use upper/lowercase, 6 bits are enough
- So we could store 4 characters in 24 bits (3 bytes)
- But how to do it?

Compression

Data: 00101101 00011100 00001111 00101010

Result: 10110101 11000011 11101010

- 1st (or 5th, 9th, and so on) character is easy: shift left by 2 and make it byte 1

```
int byte1 = char1 << 2;  // 10110100
```

- 2nd character must be split across bytes 1 and 2

```
byte1 = byte1 + char2 >> 4;  //10110101
int byte2 = char2 << 4;  // 11000000
```

Compression

Data: 00101101 00011100 00001111 00101010

Result: 10110101 11000011 11101010

- 3rd character is split across bytes 2 and 3

  byte2 = byte2 + char3 >> 2; // 11000011

  int byte3 = char3 << 6; // 11000000

- The last character is easiest of all, just add it

  byte3 = byte3 + char4; // 11101010

- Repeat for the next 4 characters until you run out.
- You've saved 25% space at the cost of a little time.

Compression

- Getting the characters back is a similar process
- 1st (or 4th, 7th, and so on) byte holds char1 and part of char2:

```
int char1 = byte1 >>2;

Int char2 = (byte1 << 6) >> 2;
```

- Moving left 6 bits erases the first character and then moving right 2 bits puts the remainder in the right spot

10110101 -> 01000000 -> 00010000

Compression

- 2nd byte holds the rest of char2 and part of char3:

  char2 = char2 + (byte2 >>4);

  Int char3 = (byte2 << 4) >> 2;

- 3rd byte holds the rest of char3 and all of char4:

  char3 = char3 + (byte3 >>6);

  Int char4 = (byte3 << 2) >> 2;

- Rinse and repeat until you run out of bytes.

Bitwise Operators

ENCRYPTION

Encryption

- Early data in programs was easy to find and read.
- Simple encryption works similarly to compression
  - Use a key that is between 0 and maxint - maxchar
  - Add the key to data bytes to make them unreadable
  - Subtract the key (if you know it) to make it readable
- Unfortunately, a computer can quickly try all possible keys
- So, add a step that rotates the bits as well

Encryption

- Rotation is just a combination of left and right shift
  - int data = 103; // 01100111
  - int encrypt_data = (data<<5) + (data>>3) // 11101100
- This multiplies the possibilities by 16: 8 bit positions X 2 (either before or after adding the key)
- Still pretty easy for a computer to break but keeps out the casual snoops

Bitwise Operators

BIT TEST AND SET

Bit Test and Set

- Many hardware devices are controlled by registers
  - Think of a hardware register as a bank of switches
  - The register is ‘memory mapped’
  - We control the switches by ‘setting’ the bit to 1 or ‘resetting’ the bit to 0.
  - We can also see what the bits are to see how the device is set
- This technique also works for software ‘switches’ if we need to save space

Bit Test and Set

- Testing a bit to see if it is set or reset requires a ‘mask’
- If our mask has a 1 where we want to test and 0 elsewhere, we can use the & operator to test with
- Ex: mask = 00000100 (we want to test the 3rd bit)

_____1__	_____0__ data
& 00000100	& 00000100 mask
00000100	00000000 result

- The other bits in the data don’t matter

Bit Test and Set

_____1__	_____0__ data
& 00000100	& 00000100 mask
00000100	00000000 result

- A nonzero result means the bit is set (1)
- A zero result means the bit is reset (0)
- We can only test one bit at a time
- If our mask is 00010010 and we get a non-zero result, which bit (or was it both) was set?
- However, sometimes we don't care which one.

Bit Test and Set

- Many headers files will define masks for the possible bit positions

```
bit0mask = 1;  // 00000001
bit1mask = 2;  // 00000010
bit2mask = 4;  // 00000100   and so on…
```

- But we can be cleverer: mask = 00000001;
- Now we can use left shift to test any bit

```
int testbit = 3;  // we want to test the third bit
result = data & (mask<<(testbit – 1));
```

Bit Test and Set

- The convention is usually to number the bits from the right starting at 0
- So the third bit would actually be bit 2 and we don't need the subtraction in the last example.
- That's faster and is a big reason the convention was adopted.
- The shift method of creating a mask is nearly as fast as fetching from memory … sometimes faster.

Bit Test and Set

- Now we can test a bit, but how to change one?
- The same mask works here as well.
- If you wish to set the bit, use OR (|)

  data | (mask << 3); // sets the 4th bit

  It doesn't matter if the bit was already set

- If you wish to reset the bit, subtract the mask

  data – (mask << 3); // resets the 4th bit

  But it only works correctly if the bit was set, so test first

Bit Test and Set

- Unlike testing, we can set or reset multiple bits at a time

```
mask = 49;  // 00110001 sets/resets bits 0, 4, and 5
```

- Shifting of these masks are generally useless so defining them in a header is the way to go
- We can define the bit positions and add them to create any mask
  - Remember the masks from slide 28?

```
mask = bit5mask + bit4mask + bit0mask;
```

Bit Test and Set

- Back on slide 10, mask building with complement was mentioned
- Good if you want to set/reset all except
- Mask the bits you wish to except
- Use the complement function to get actual mask

```
mask = 9;  // 00001001 single out bits 3 and 0
~mask;  // 11110110 now all but 3 and 0 are masked
```

Bit Test and Set

- Suppose you want to flip a bit, no matter what it's current value.
- Then you simply use the XOR operator.
  - Any mask bit set to 1 flips the data bit.

    m = 1: d = 1, r = 0 d = 0; r = 1
  - Any mask bit set to 0 leaves the data bit alone

    m = 0: d = 1, r = 1 d = 0; r = 0

Bitwise Operators

END OF SECTION