Bitwise Operations

A *Bit* of Review

- Bit
  - lowest level of storage
  - has a value of 0 or 1 (0 = off / false, 1 = on / true)
  - is actually an electrical switch that either holds a charge (on) or has no charge (off)

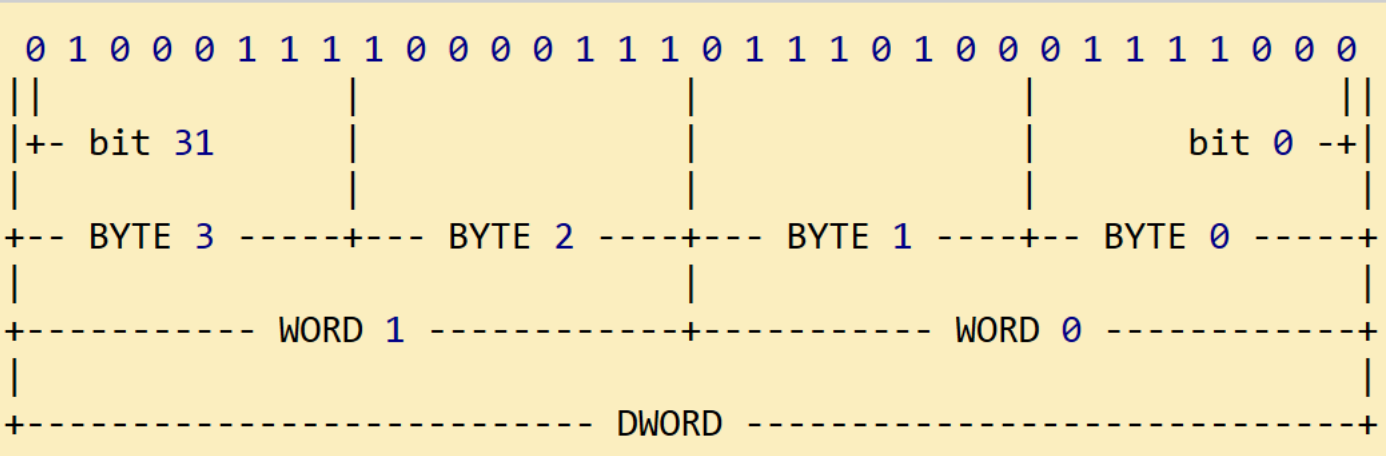
- Byte
  - made up of 8 bits
  - each "cell" of RAM is 1 byte
  - data types take up 1 or more bytes

- All things are stored at the bit (binary) level even though we sometimes forget this very basic fact about computers.

A Bit More Words

- WORD: made up of 2 bytes or 16 bits

- DWORD: made up of 2 WORDS or 4 bytes or 32 bits



Bitwise Terminology

- **mask or bitmask**: data that is used to isolate specific bit(s) of another value, while ignoring other bits.

- **least significant bit**: the lowest bit in a byte, word, dword, etc. (the bit all the way to the right)
  - ex: 0000000**0**
  - ex: 00000000 00000000 0000000 0000000**0**

- **most significant bit**: the highest bit in a byte, word, dword, etc. (the bit all the way to the left)
  - ex: **0**0000000
  - ex: **0**0000000 00000000 0000000 00000000

Bitwise Terminology

- **least-significant byte:** the lowest byte in a data type made up of multiple bytes
  - ex: 00000000 00000000 00000000 **00000000**

- **most-significant byte:** the highest byte in a data type made up of multiple bytes
  - ex: **00000000** 00000000 00000000 00000000

- **big-endian:** storage scheme where the most significant byte of a word is placed in the smallest address sequentially and the least significant byte is stored in the largest address sequentially.

- **little-endian**: storage scheme where the least significant byte is stored in the smallest address sequentially and the most significant byte is stored in the largest address sequentially. (intel machines use this.)

Bitwise Operators

- Java has bit-wise operators which can perform operations on a value at the **bit** level.

- Normally these are applied to positive (unsigned) integer data types.
  - doubles present some issues when performing bit-wise operations.
  - negatives can have issues due to the sign bit

- Two shift operators: >>, <<

- Four logical operators: &, |, ^, ~

- Five assignment operators: >>=, <<= &=, |=, ^=

The Left Shift Operator <<

- Syntax: `[variable or literal] << [number of places]`

- shifts all of the bits to the left by the number of places specified and zeros are padded on the right

- every shift is the equivalent of multiplying by a power of 2.

- Example:

```
int x = 8;                  // 8 in binary is 00001000
int y = x << 2;               // shift the bits of x by 2
System.out.println(y);   // will print 32
```

Another Left Shift Example

- Here we will show shifting an integer which has 4 bytes.
- 127 << 2 //shift 127 to the left by 2 bits
- Convert 127 to Binary:

  00000000 00000000 00000000 01111111
- Shift 127 two places to the left, padding the right with 0's

  00000000 00000000 00000001 11111100
- Convert back to Decimal: 508

The right-shift Operator >>

- Syntax: `[variable` or `literal]` `>>` `[number of places]`

- shifts all of the bits to the right by the number of places specified and zeros are padded on the left

- every shift is the equivalent of integer division by 2

- Example:

```
int x = 8;                  // 8 in binary is 00001000
int y = x >> 2;             // shift the bits of x by 2
System.out.println(y); // will print 2
```

Other Shift Notes

- If you use a data type made up of multiple bytes, then the shifting operations are applied to the entire sequence of bits.

- What happens if we shift a value and we run out of bits in either direction?
  - Example: 128 could be stored using only 1 byte and the binary of 128 is 10000000. If you shift left 1 time then the result is 00000000.

- NOTE: Using left and right shift results in faster code than using division or multiplication.

Bitwise AND Operator &

- Performs boolean AND operation on the individual bits of two values.
  - remember 0 = false, 1 = true
  -
- Example 100 & 50 = 32

```
1   &   1   ==   1
1   &   0   ==   0
0   &   1   ==   0
0   &   0   ==   0
```

```
  0110  0100  =  100
& 0011  0010  =  50

-------------------

  0010  0000  =  32
```

Bitwise OR Operator |

- Performs boolean OR operation on the individual bits of two values.
  - remember 0 = false, 1 = true

```
1   |   1   ==   1
1   |   0   ==   1
0   |   1   ==   1
0   |   0   ==   0
```

- Example: 100 | 50 = 118

```
  0110  0100  =  100
| 0011  0010  =  50
-------------------
  0111  0110  =  118
```

Bitwise XOR Operator

- Performs boolean XOR operation on the individual bits of two values.
  - remember 0 = false, 1 = true

```
1   ^   1   ==   0
1   ^   0   ==   1
0   ^   1   ==   1
0   ^   0   ==   0
```

- Example: 100 ^ 50 = 86

```
  0110  0100  =  100
^ 0011  0010  =  50
-------------------
  0101  0110  =  86
```

Bitwise Compliment Operator ~

- Performs Boolean NOT on the individual bits of one value (flips the bits to their opposite or compliment).
  - remember 0 = false, 1 = true

- Example: ~100, for this example assume 100 is of the byte type.

$$\sim(0110\ 0100) = 1001\ 1011 = 155$$

Using Bitmasks

- Bitmasks are items that are used to manipulate one or more bits of a value.

- Normally we want to target specific bits so we need to "mask" the bits we are not interested in and focus on the one or more bits we want.

- NOTE: The char type is 2 bytes and can actually be used in the shifting operations. Remember that characters are stored using their Unicode values and when you perform a bit-wise operation on a char type, you are performing the operation on its Unicode value.

- For simplicity, the follow examples will all use the unsigned char type.

Using Bitmasks: Targeting a Specific Bit

- If you want to perform an operation to only 1 bit of a value, then you need to be able to target that bit first.

- To target the nth bit:
  - Create a mask to target the nth bit.

Using Bitmasks: Targeting a Specific Bit

- Turning a specific bit on:
  - perform an OR operation using a bitmask targeting the specific bit.

- First create the correct mask to turn on the nth bit
  - We can do this by starting with value 1, and shifting it n - 1 times to the left
  - i.e. `char mask = 1 << (n - 1)`

- Example: Turn the 4th bit on.

  `char mask = 1 << (4 – 1)`

  `char x = 149` //is 1001**0**101 in binary

  `char result = x | mask` //result is 157, 1001**1**101 in binary

Using Bitmasks: Turning a Specific Bit Off

- Turning a specific bit off:
  - perform an AND operation using a bitmask targeting the specific bit.

- First create the correct mask to turn off the nth bit
  - We can do this by starting with value 1, and shifting it n - 1 times to the left THEN flipping all the bits to their opposite.
  - i.e. `char mask = ~(1 << (n - 1))`

- Example: Turn the 4th bit off.

  `char mask = ~(1 << (4 – 1))`

  `char x = 157` //is 1001**1**101 in binary

  `char result = x & mask` //result is 149, 1001**0**101 in binary

Using Bitmasks: Checking the Status of a Bit

- Use the same type of mask that we create for turning a bit on, and then AND the value with the mask.
  - if the result of the operation is 0 then the bit was off
  - if the result of the operation is any other value, then the bit was on.

- Example: Check the status of the fourth bit

```
char mask = 1 << (4 – 1);
char x = 149; //is 10010101 in binary
char y = 157; //is 10011101 in binary


System.out.println(x & mask); //prints 0 (bit was off)
System.out.println(y & mask); //prints 8 (bit was on)
```