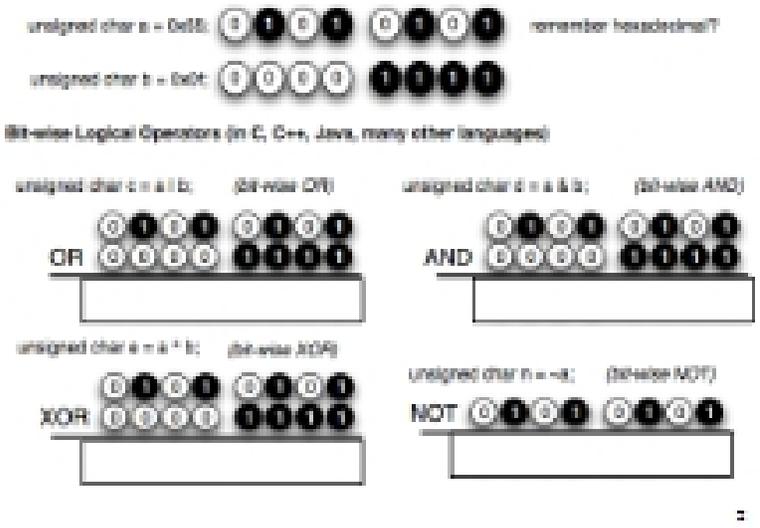
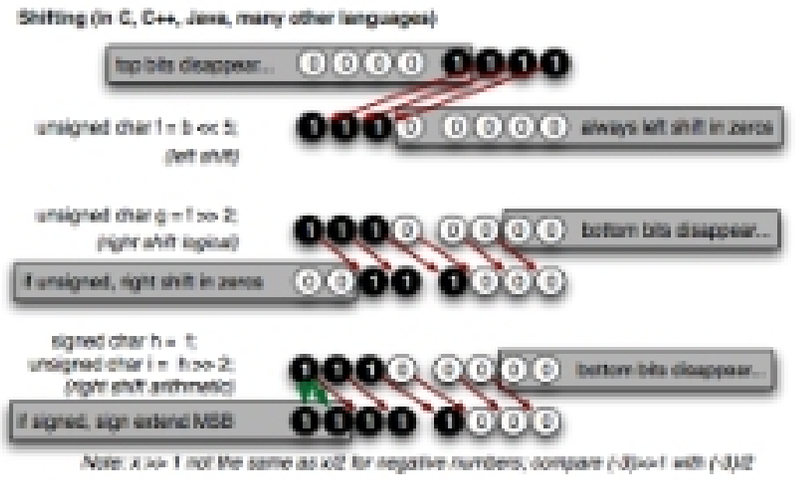
Lecture 4: Bitwise Operations

(CPEG323: Intro. to Computer System Engineering)

Bit Operators (I)



Bit Operators (II)



Note: x >> 1 not the same as x/2 for negative numbers, compare (-3)>>1 with (-3)/2

Bit-wise Arithmetic 1

```
int a = 5;
int f = a << 2;
```

What is f?

1) 0x10
2) 0x12
3) 0x14
4) 0x07

Bit-wise Arithmetic 2

```
int a = 5;
int g = a | 3;
```

What is g ?

1) 0x5
2) 0x7
3) 0x8
4) 0x12

Bit-wise Arithmetic 3

```
int b = -7;
int h = b >> 2;
```

What is contained in h ?

1) 0
2) -1
3) -2
4) -5

Bit-wise Arithmetic 4

```
int b = -7;
int h = b >> 2;
int i = ((unsigned)b) >> 2;
```

Which is true ?

1) h > i
2) h == i
3) h < i

Bit-wise Arithmetic 5

```
int b = -7;
int j = b & 0xf;
```

What is j ?

1) 0x7
2) 0x9
3) 0xa
4) 0xf

Low-level Programming in "High-level" Languages

- Very often it is necessary to store a large number of very small data items.

Low-level Programming in "High-level" Languages

- Very often it is necessary to store a large number of very small data items.
- Example: A Social Security Number (SSN) registry
  - Needs to keep track of which SSNs have already been allocated.
- How much space is required?

Storing collections of bits as integers

- Store N bits in each N-bit integer, only need $10^9/N$ integers
  - Requires $10^9/8$ bytes = 125MBs of storage (Fits on a CD)
- Allocate array:
  int array_size = 1000000000/_____
  unsigned int SSN_registry[array_size];

- Want two operations on this array:
  - check_SSN: returns 1 if SSN is used, 0 otherwise
  - set_SSN: marks an SSN as used.

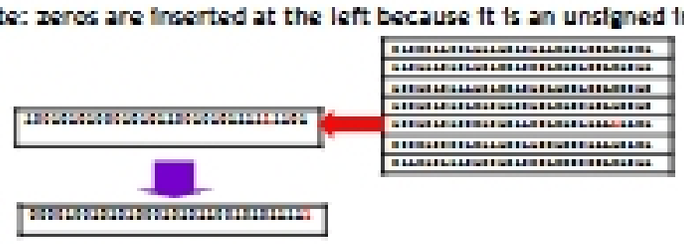
SSN #7
SSN #68

check_SSN

```
int check_SSN(unsigned int SSN_registry[], int ssn) {
  int word_index = ssn / (8*sizeof(int));
  int word = SSN_registry[word_index];
```

check_SSN

```
int check_SSN(unsigned int SSN_registry[], int ssn) {
    int word_index = ssn / (8*sizeof(int));
    int word = SSN_registry[word_index];
    int bit_offset = ssn % (8*sizeof(int)) // % is the remainder operation
    word = word >> bit_offset;  // >> shifts a value "right"
          (note: zeros are inserted at the left because it is an unsigned int)
```



check_SSN

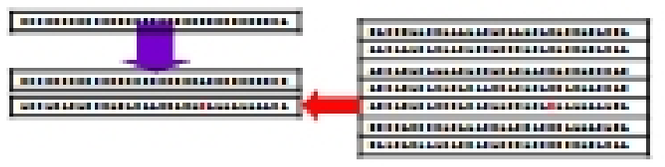
```
int check_SSN(unsigned int SSN_registry[], int ssn) {
    int word_index = ssn / (8*sizeof(int));
    int word = SSN_registry[word_index];
    int bit_offset = ssn % (8*sizeof(int))
    word = word >> bit_offset;
    word = (word & 1);  // & is the bit-wise logical AND operator
    return word;              (each bit position is considered independently)
}
```

set_SSN

```
void set_SSN(unsigned int SSN_registry[], int ssn) {
    int bit_offset = ssn % (8*sizeof(int))
    int new_bit = (1 << bit_offset) // "left" shift the 1 to the desired spot
                                    (always shifts in 0's at the right)
```

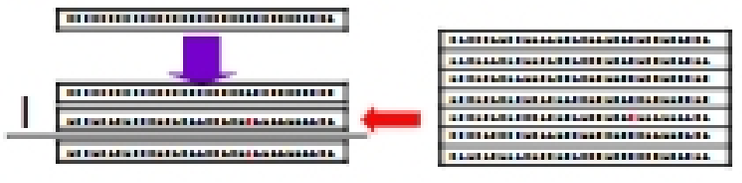
set_SSN

```
void set_SSN(unsigned int SSN_registry[], int ssn) {
    int bit_offset = ssn % (8*sizeof(int))
    int new_bit = (1 << bit_offset)
    int word_index = ssn / (8*sizeof(int));
    int word = SSN_registry[word_index];
```



set_SSN

```
void set_SSN(unsigned int SSN_registry[], int ssn) {
    int bit_offset = ssn % (8*sizeof(int))
    int new_bit = (1 << bit_offset)
    int word_index = ssn / (8*sizeof(int));
    int word = SSN_registry[word_index];
    word = word | new_bit;  // bit-wise logical OR sets the desired bit
```



set_SSN

```
void set_SSN(unsigned int SSN_registry[], int ssn) {
    int bit_offset = ssn % (8*sizeof(int))
    int new_bit = (1 << bit_offset)
    int word_index = ssn / (8*sizeof(int));
    int word = SSN_registry[word_index];
    word = word | new_bit;
    SSN_registry[word_index] = word;  // write back the word into array
}
```

Shorthand for last 3 lines: SSN_registry[word_index] |= new_bit;