Bitwise Data Manipulation

Bitwise operations

More on integers

bitwise operators

ex

Bitwise operators on fixed-width **bit vectors**.

AND & OR | XOR ^ NOT ~

```
  01101001        01101001        01101001
& 01010101      | 01010101      ^ 01010101      ~ 01010101
  --------      ----------      ----------      ----------
  01000001

                                                  01010101
                                                ^ 01010101
                                                ----------
```

Laws of Boolean algebra apply bitwise.

e.g., DeMorgan's Law: ~(A | B) = ~A & ~B

Aside: sets as bit vectors

ex

Representation: *n*-bit vector gives subset of {0, …, *n*–1}.

$a_i = 1 \equiv i \in A$

```
01101001    { 0, 3, 5, 6 }
76543210

01010101    { 0, 2, 4, 6 }
76543210
```

Bitwise Operations

```
&    01000001    { 0, 6 }
|    01111101    { 0, 2, 3, 4, 5, 6 }
^    00111100    { 2, 3, 4, 5 }
~    10101010    { 1, 3, 5, 7 }
```

Set Operations?

bitwise operators in C

ex

`& | ^ ~` apply to any *integral* data type

`long, int, short, char, unsigned`

Examples (`char`)

```
~0x41 =

~0x00 =

0x69 & 0x55 =

0x69 | 0x55 =
```

Many bit-twiddling puzzles in upcoming assignment

logical operations in C

ex

&& || ! apply to any "integral" data type
`long, int, short, char, unsigned`

0 is **false** **nonzero** is **true** **result** always **0 or 1**

early termination a.k.a. **short-circuit evaluation**

Examples (`char`)

```
 !0x41 =
 !0x00 =
!!0x41 =

0x69 && 0x55 =
0x69 || 0x55 =
```


Encode playing cards.

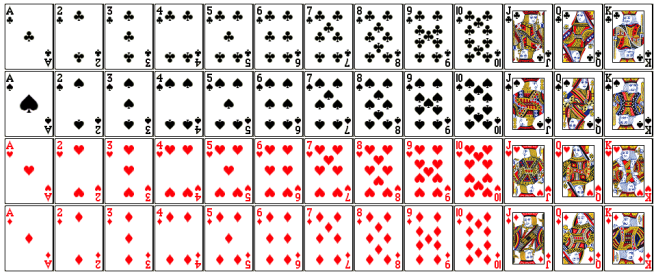
52 cards in 4 suits

How do we encode suits, face cards?

What operations should be easy to implement?

Get and compare rank

Get and compare suit



Two possible representations

52 cards – 52 bits with bit corresponding to card set to 1

"One-hot" encoding

Hard to compare values and suits independently

Not space efficient

4 bits for suit, 13 bits for card value – 17 bits with two set to 1

Pair of one-hot encoded values

Easier to compare suits and values independently

Smaller, but still not space efficient

Two better representations

Binary encoding of all 52 cards – only 6 bits needed

Number cards uniquely from 0

Smaller than one-hot encodings.

Hard to compare value and suit

Binary encoding of suit (2 bits) and value (4 bits) separately

Number each suit uniquely

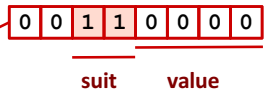
Number each value uniquely

Still small

Easy suit, value comparisons

Compare Card Suits

mask: a bit vector that, when bitwise ANDed with another bit vector *v*, turns all *but* the bits of interest in *v* to 0

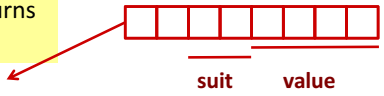



```
#define SUIT_MASK 0x30

int sameSuit(char card1, char card2) {
  return !((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));

  //same as (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}



char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
...
if ( sameSuit(hand[0], hand[1]) ) { ... }
```


Compare Card Values

ex

mask: a bit vector that, when bitwise ANDed with another bit vector *v*, turns all *but* the bits of interest in *v* to 0




```
#define VALUE_MASK

int greaterValue(char card1, char card2) {




}

char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
...
if ( greaterValue(hand[0], hand[1]) ) { ... }
```


Bit shifting

x 1 0 0 1 1 0 0 1

x << 2 1 0 0 1 1 0 0 1 0 0 **logical shift left 2**

lose bits on left *fill with zeroes on right*

1 0 0 1 1 0 0 1 x

fill with zeroes on left

logical shift right 2 0 0 1 0 0 1 1 0 0 1 **x >> 2**

lose bits on right

arithmetic shift right 2 1 1 1 0 0 1 1 0 0 1 **x >> 2**

fill with copies of MSB on left

unsigned **shifting** and **arithmetic**

unsigned
x = 27;
y = x << 2;
y == 108

0 0 0 1 1 0 1 1

0 0 0 1 1 0 1 1 0 0

logical shift left

logical shift right

1 1 1 0 1 1 0 1

0 0 1 1 1 0 1 1 0 1

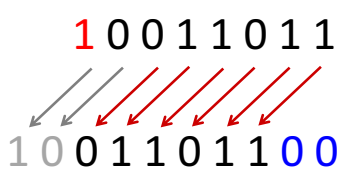
unsigned
x = 237;
y = x >> 2;
y == 59

two's complement **shifting** and **arithmetic**

signed
x = -101;

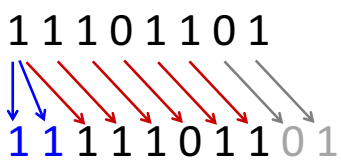
y = x << 2;

y == 108



logical shift left

arithmetic shift right



signed
x = -19;

y = x >> 2;

y == -5

shift-and-add

ex

Available operations

`x << k` implements `x * 2^k`

`x + y`

Implement `y = x * 24` using only <<, +, and integer literals

Shift gotchas

!!!

Logical or arithmetic shift right: how do we tell?

C: compiler chooses

Usually based on type: rain check!

Java: >> is arithmetic, >>> is logical

Shift an *n*-bit type by at least 0 and no more than n-1.

C: other shift distances are undefined.

anything could happen

Java: shift distance is used modulo number of bits in shifted type

Given int x: x << 34 == x << 2

Shift and Mask: extract a bit field

ex

Write C code:

extract *2nd most significant byte* from a 32-bit integer.

given x = 01100001 01100010 01100011 01100100

should return: 00000000 00000000 00000000 01100010

All other bits are zero.

Desired bits in least significant byte.

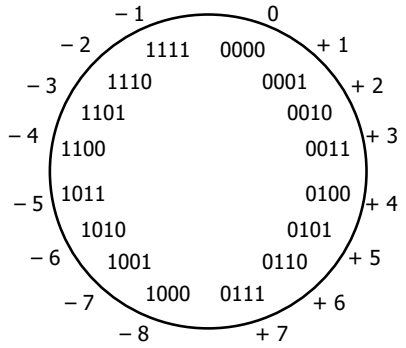
What does this function compute?

ex

```
unsigned puzzle(unsigned x, unsigned y) {
  unsigned result = 0;
  for (unsigned i = 0; i < 32; i++){
    if (y & (1 << i)) {
      result = result + (x << i);
    }
  }
  return result;
}
```


multiplication

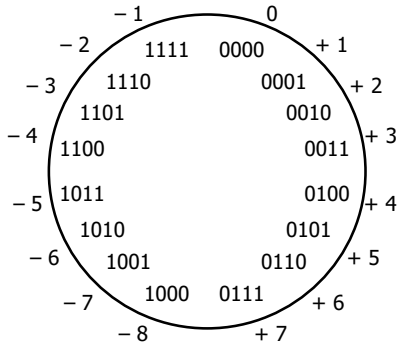
2	0010
x 3	x 0011
6	00000100
-2	1110
x 2	x 0010
-4	11111100



Modular Arithmetic

multiplication

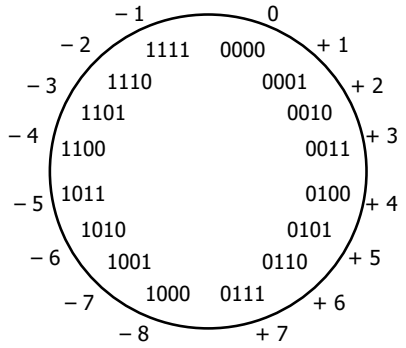
5	0101
x 4	x 0100
~~20~~	00010100
4	
-3	1101
x 7	x 0111
~~-21~~	11101011
-2	



Modular Arithmetic

multiplication

5	0101
x 5	x 0101
~~25~~	00011001
-7	
-2	1110
x 6	x 0110
~~-12~~	11110100
4	



Modular Arithmetic

***Convert/cast** signed number to **larger** type.*

0 0 0 0 0 0 1 0	8-bit 2
_ _ _ _ _ _ _ _ 0 0 0 0 0 0 1 0	16-bit 2
1 1 1 1 1 1 0 0	8-bit -4
_ _ _ _ _ _ _ _ 1 1 1 1 1 1 0 0	16-bit -4

Rule/name?

Casting Integers in C

!!!

Number literals: `37` is signed, `37U` is unsigned

Integer Casting: *bits unchanged, just reinterpreted.*

Explicit casting:

```
int tx = (int) 73U;            // still 73
unsigned uy = (unsigned) -4;   // big positive #
```

Implicit casting:	**Actually does**
`tx = ux;`	`// tx = (int)ux;`
`uy = ty;`	`// uy = (unsigned)ty;`
`void foo(int z) { ... }`	
`foo(ux);`	`// foo((int)ux);`
`if (tx < ux) ...`	`// if ((unsigned)tx < ux) ...`

More Implicit Casting in C

!!!

If you **mix unsigned** and **signed** in a single expression, then ***signed** values are **implicitly cast to unsigned**.*

How are the argument bits interpreted?

Argument$_1$	**Op**	**Argument$_2$**	**Type**	**Result**
`0`	`==`	`0U`	unsigned	1
`-1`	`<`	`0`	signed	1
`-1`	`<`	`0U`	unsigned	**0**
`2147483647`	`<`	`-2147483648`		
`2147483647U`	`<`	`-2147483648`		
`-1`	`<`	`-2`		
`(unsigned)-1`	`<`	`-2`		
`2147483647`	`<`	`2147483648U`		
`2147483647`	`<`	`(int)2147483648U`		

Note: T_{min} = -2,147,483,648 T_{max} = 2,147,483,647