# Representing Data with Bits

bits, bytes, numbers, and notation

## binary = base 2

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 8 | 4 | 2 | 1 |
| $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 3 | 2 | 1 | 0 |

$= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

*weight*

*position*

When ambiguous, subscript with base:

$101_{10}$ Dalmatians (movie)

$101_2$-Second Rule (folk wisdom for food safety)

irony



Show powers, strategies.

## conversion and arithmetic

ex

$19_{10} = ?_2$

$1001_2 = ?_{10}$

$240_{10} = ?_2$

$11010011_2 = ?_{10}$

$101_2 + 1011_2 = ?_2$

$1001011_2 \times 2_{10} = ?_2$



*What do you call 4 bits?*

## *byte* = 8 bits

a.k.a. octet

**Smallest unit of data**

*used by a typical modern computer*

**Binary** $00000000_2$ -- $11111111_2$

**Decimal** $000_{10}$ -- $255_{10}$

**Hexadecimal** $00_{16}$ -- $FF_{16}$

**Byte = 2 hex digits!**

Programmer's hex notation (C, etc.):

`0xB4` = $B4_{16}$

Octal (base 8) also useful.

Why do 240 students often confuse Halloween and Christmas?

| Hex | Decimal | Binary |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

# *word* |wərd|, n.

**Natural unit of data used by processor.**

- **Fixed size** (e.g. 32 bits, 64 bits)
  - Defined by ISA: Instruction Set Architecture
- machine instruction operands
- word size = register size = address size

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

**Java/C int = 4 bytes:** 11,501,584

**MSB: most significant bit**

**LSB: least significant bit**



# fixed-size data representations

| Java Data Type | C Data Type | (size in bytes) 32-bit | 64-bit |
|---|---|---|---|
| boolean | | 1 | 1 |
| byte | char | 1 | 1 |
| char | | 2 | 2 |
| short | short int | 2 | 2 |
| int | int | 4 | 4 |
| float | float | 4 | 4 |
| | long int | 4 | 8 |
| double | double | 8 | 8 |
| long | long long | 8 | 8 |
| | long double | 8 | 16 |

**Depends on word size!**



ex

# *bitwise* operators

**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



ex

# *bitwise* operators in C

`& | ^ ~` 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 ex

# *logical* operations in C

**&& || !** 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.

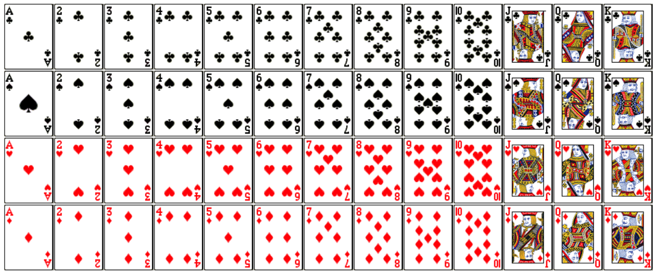
**52 cards in 4 suits**

How do we encode suits, face cards?

**What operations should be easy to implement?**

Get and compare rank

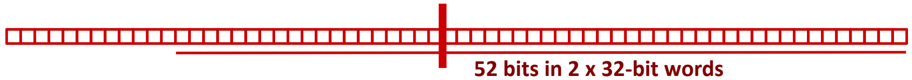
Get and compare suit





# 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

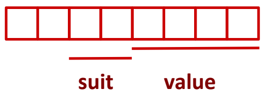

**Binary encoding of suit (2 bits) and value (4 bits) separately**

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

| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

suit value

```
#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

| | | | | | | | |
|---|---|---|---|---|---|---|---|

suit value

```
#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*



## **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.