Memory & Data III

CSE 351 Winter 2021

Instructor:

Mark Wyse

Teaching Assistants:

Kyrie Dowling

Catherine Guevara

Ian Hsiao

Jim Limprasert

Armin Magness

Allie Pfleger

Cosmo Wang

Ronald Widjaja

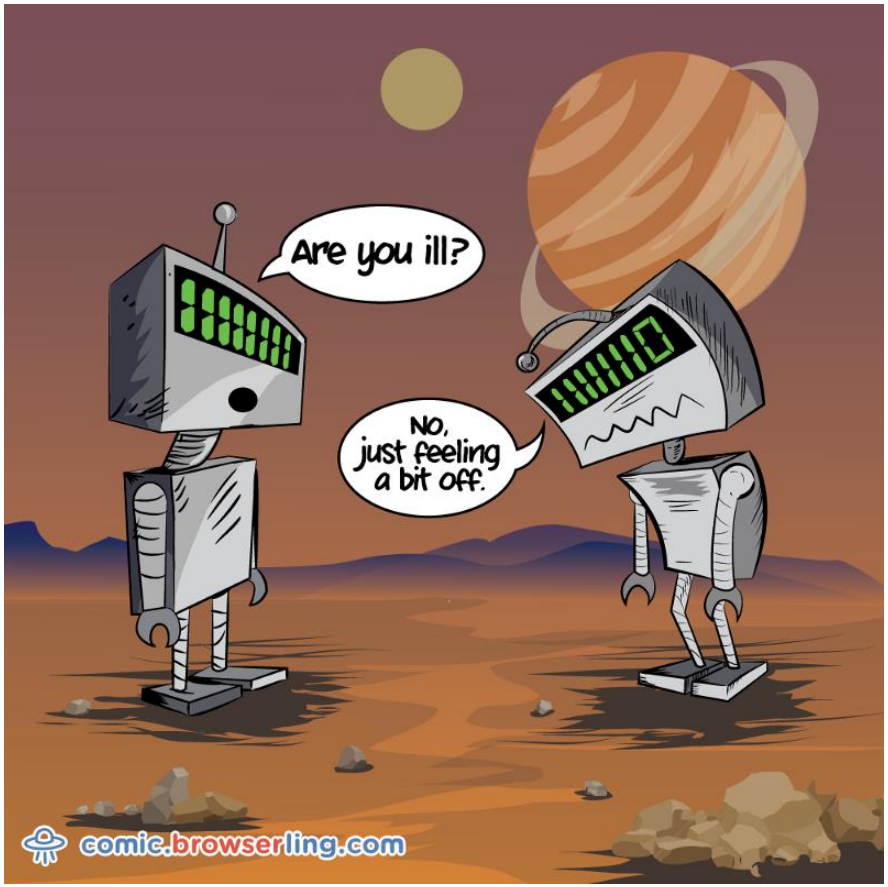


image source

Administrivia

- hw3 due Wednesday, hw4 due Friday
  - **by 11:00am PST!**

- Lab 1a
  - Workflow:
    1) Edit `pointer.c`
    2) Run the Makefile (`make clean` followed by `make`) and check for compiler errors & warnings
    3) Run ptest (`./ptest`) and check for correct behavior
    4) Run rule/syntax checker (`python3 dlc.py`) and check output
  - Due Friday 1/15
    - We grade just your *last* submission

Reading Review

- Terminology:
  - Bitwise operators (&, |, ^, ~)
  - Logical operators (&&, ||, !)
  - Short-circuit evaluation

- Questions from the Reading?
  - about Bitwise and Logical Operators

Bitmasks

- Typically binary bitwise operators (&, |, ^) are used with one operand being the "input" and other operand being a specially-chosen bitmask (or *mask*) that performs a desired operation

- Operations for a bit b (answer with 0, 1, b, or $\bar{b}$):

$b \;\&\; 0 =$ 0 "clear" set ∅ $\quad b \;\&\; 1 =$ b "keep"

$b \mid 0 =$ b keep "no-op" $\quad b \mid 1 =$ 1 "set"

$b \;\hat{}\; 0 =$ b keep "no-op" $\quad b \;\hat{}\; 1 =$ $\bar{b}$ "flip"

Bitmasks

- Typically binary bitwise operators (&, |, ^) are used with one operand being the "input" and other operand being a specially-chosen bitmask (or *mask*) that performs a desired operation

- Example: $b|0 = b,\ b|1 = 1$

"set" "keep"

```
  01010101  ← input
| 11110000  ← bitmask
----------
  11110101
```

& → keep or clear

| → keep or set

^ → keep or flip

Short-Circuit Evaluation

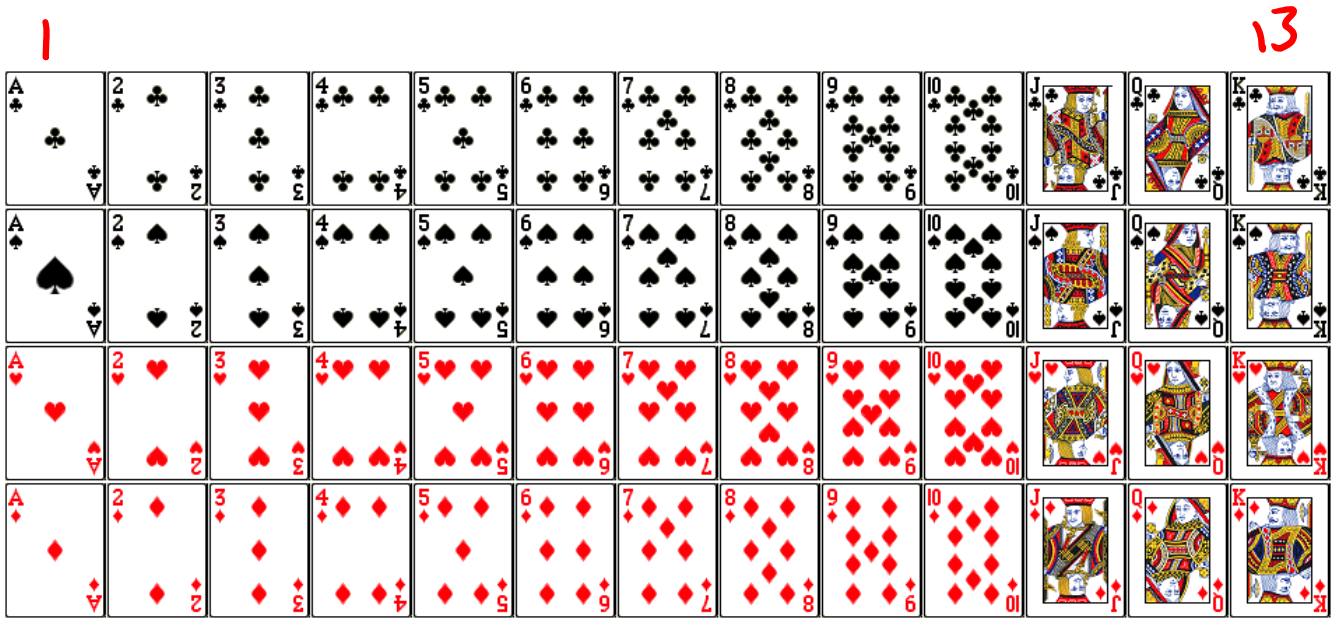
- If the result of a binary logical operator (&&, ||) can be determined by its first operand, then the second operand is never evaluated
  - Also known as *early termination*

- Example: if (p && *p) for a pointer p to "protect" the dereference
  - Dereferencing NULL (0) results in a segfault

if(p) → true if p≠0 (NULL)
→ false if p == NULL

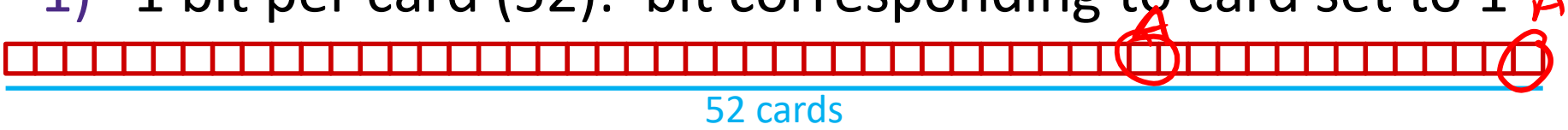
Numerical Encoding Design Example

- Encode a standard deck of playing cards
  - 52 cards in 4 suits
- Operations to implement:
  - Which is the higher value card? 1→13
  - Are they the same suit? →CSDH →4 suits



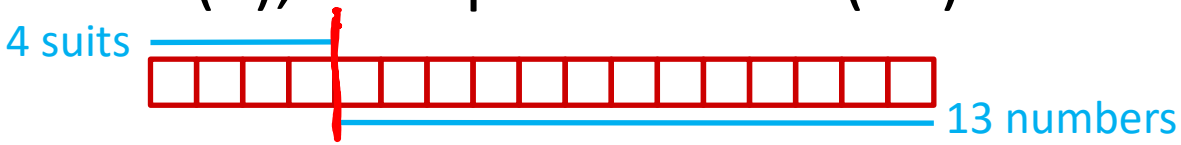
Representations and Fields

1) 1 bit per card (52): bit corresponding to card set to 1



- "One-hot" encoding (similar to set notation)
- Drawbacks:
  - Hard to compare values and suits
  - Large number of bits required

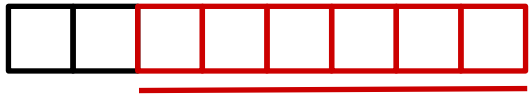
2) 1 bit per suit (4), 1 bit per number (13): 2 bits set



- Pair of one-hot encoded values
- Easier to compare suits and values, but still lots of bits used

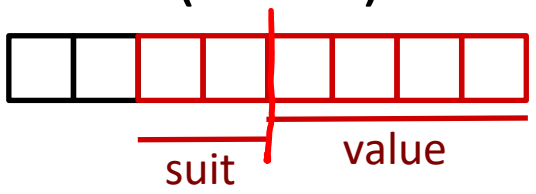
Representations and Fields

3) Binary encoding of all 52 cards – only 6 bits needed
   - $2^6 = 64 \geq 52$ (# cards)



   - Fits in one byte (smaller than one-hot encodings)
   - How can we make value and suit comparisons easier?

4) Separate binary encodings of suit (2 bits) and value (4 bits)



   - Also fits in one byte, and easy to do comparisons

K	Q	J	. . .	3	2	A
1101	1100	1011	. . .	0011	0010	0001

could be Ø

♣	00
♦	01
♥	10
♠	11

Compare Card Suits

mask: a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector *v*.
Here we turn all *but* the bits of interest in *v* to 0.

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

```
#define SUIT_MASK  0x30

int sameSuitP(char card1, char card2) {
  return (!((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK)));
  //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

returns **int**

SUIT_MASK = 0x30 =

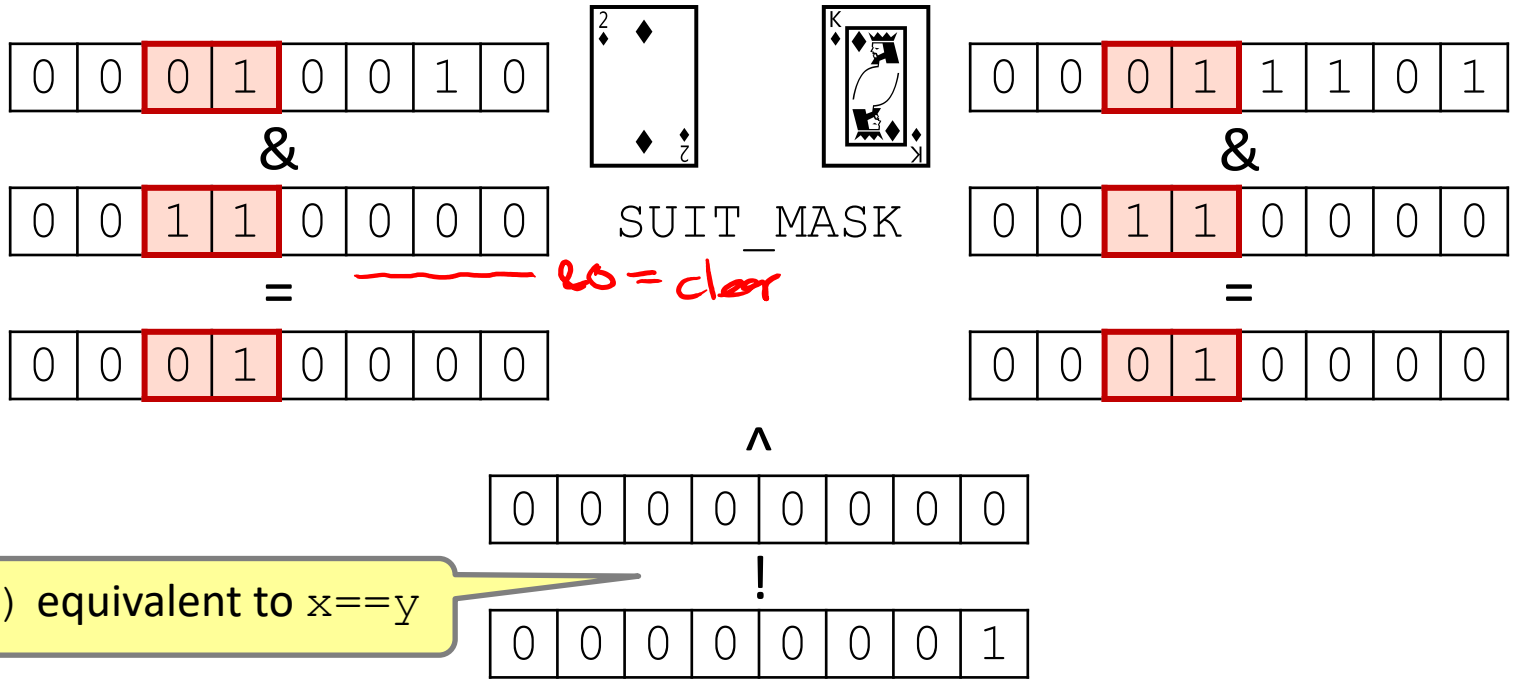
0	0	1	1	0	0	0	0

suit value

equivalent

Compare Card Suits

```
#define SUIT_MASK   0x30

int sameSuitP(char card1, char card2) {
  return (!((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK)));
  //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

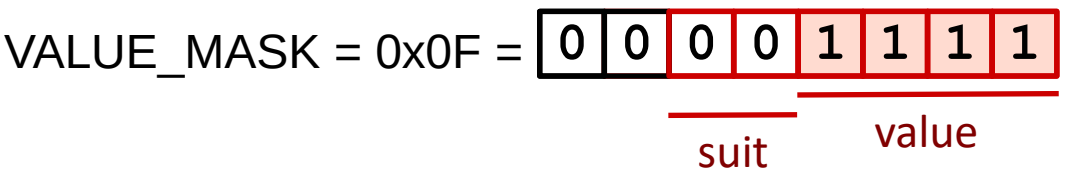


! (x^y) equivalent to x==y

Compare Card Values

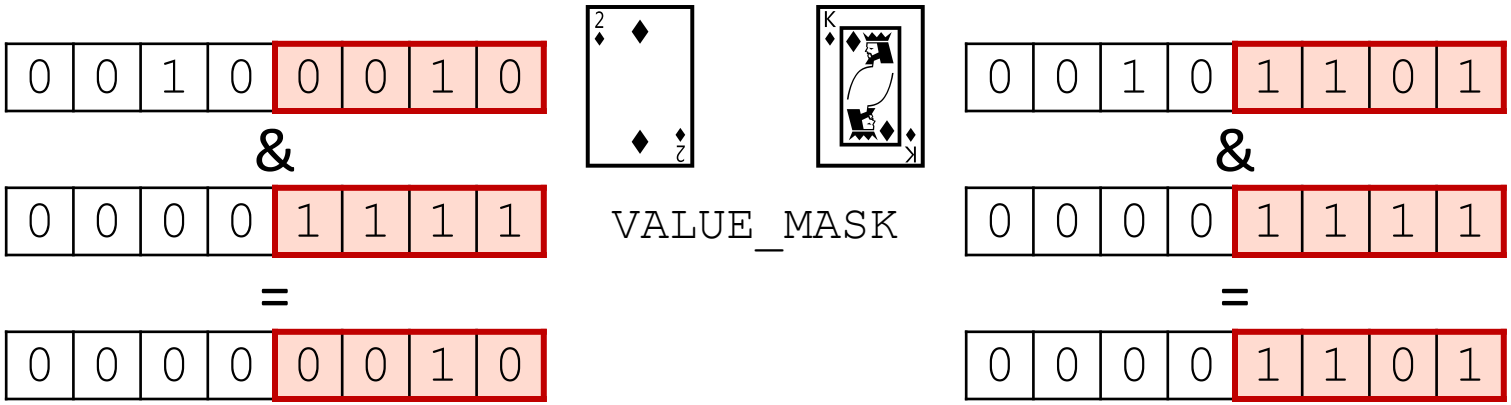
```
char hand[5];          // represents a 5-card hand
char card1, card2;     // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( greaterValue(card1, card2) ) { ... }
```

```
#define VALUE_MASK   0x0F

int greaterValue(char card1, char card2) {
  return ((unsigned int)(card1 & VALUE_MASK) >
          (unsigned int)(card2 & VALUE_MASK));
}
```



Compare Card Values

```
#define VALUE_MASK   0x0F

int greaterValue(char card1, char card2) {
  return ((unsigned int)(card1 & VALUE_MASK) >
          (unsigned int)(card2 & VALUE_MASK));
}
```



$2_{10} > 13_{10}$

0 (false)

Summary

- Bit-level operators allow for fine-grained manipulations of data
  - Bitwise AND (&), OR (|), and NOT (~) different than logical AND (&&), OR (||), and NOT (!)
  - Especially useful with bit masks
- Choice of *encoding scheme* is important
  - Tradeoffs based on size requirements and desired operations

Review Questions (Breakouts)

- Compute the result of the following expressions for `char c = 0x81;`
  - `c ^ c` = 0x00 → A
  - `~c & 0xA9` = 0x28 → B
  - `c || 0x80` c = 0x81 → true → 0x01 → B
  - `!!c` ! (!c) c → true !(!true) !(0 false) → true → B 0x01
    ↑ logical