Representing Data with Bits

bits, bytes, numbers, and notation

positional number representation

base = 10 (*decimal*)

2	4	0
100	10	1
10^2	10^1	10^0
2	1	0

weight

position

$= 2 \times 10^2 + 4 \times 10^1 + 0 \times 10^0$

Base determines:

Maximum digit (base – 1). Minimum digit is 0.

Weight of each position.

Each position holds a digit.

Represented value = sum of all position values

position value = digit value × base^{position}

binary = base 2

Binary digits are called ***bits***: 0, 1

base = 2 (*binary*)

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)

Powers of 2:
memorize up to ≥ 2^{10} (in base ten)

ex

Power: $2^?$	Decimal value
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	

conversion from binary to decimal

ex

$$1\,0\,1\,1\,0\,1_2 = ?_{10}$$

Interpret the positional representation according to the base:
sum the place weights where 1 appears (in either direction).

conversion from decimal to binary

ex

$$19_{10} = ?_2$$

Divide-by-2 Approach (Right to Left)

64	32	16	8	4	2	1

Quotient *Remainder?*

Powers-of-2 Approach (Left to Right)

64	32	16	8	4	2	1

Value *Power that fits?*

binary arithmetic

ex

$110_2 + 1011_2 = ?_2$

$1101_2 - 1011_2 = ?_2$

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

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.

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

`char`: representing characters

A C-style string is represented by a series of bytes (`char`s).

- One-byte ASCII codes for each character.
- ASCII = American Standard Code for Information Interchange

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

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

		(size in bytes)	
Java Data Type	**C Data Type**	**[word = 32 bits]**	**[word = 64 bits]**
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!

bitwise operators

bit = Boolean
0 = false
1 = true

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

AND &   OR | XOR ^   NOT ~

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

ex

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

Laws of Boolean algebra apply bitwise.

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

bitwise operators in C

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

`long, int, short, char, unsigned`

Examples (**`char`**)

```
~0x41 =

~0x00 =

0x69 & 0x55 =

0x69 | 0x55 =
```

ex

Many bit-twiddling puzzles in upcoming assignment

basics of C (vs. Java)

		Java	C
Similar	*Interaction*	compile, then run	
	Variable declaration; arithmetic	`int a = 5;` `a = a * 5;`	
	Loops	`for (int i = 0; i < N; i++) {…}`	
Different	*Data*	Objects/references	Pointers/structs/allocated memory
	Management	Garbage collection	Manual memory management
	Functions	`public static int max(int a, int b)`	`int max(int a, int b)`
	Main, printing		

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

```c
#include<stdio.h>
int main(void) {
    printf("Hello\n");
    return 0;
}
```

https://introcs.cs.princeton.edu/java/faq/c2java.html

Representation Example 1:
Sets as Bit Vectors

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

$a_i = 1 \equiv i \in A$

a = 0b01101001 A = { 0, 3, 5, 6 }
76543210

b = 0b01010101 B = { 0, 2, 4, 6 }
76543210

Bitwise Operations		**Set Operations**
`a & b = 0b01000001`	{0, 6}	Intersection
`a | b = 0b01111101`	{0, 2, 3, 4, 5, 6}	Union
`a ^ b = 0b00111100`	{2, 3, 4, 5}	Symmetric difference
`~ b = 0b10101010`	{1, 3, 5, 7}	Complement

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 =
```

ex

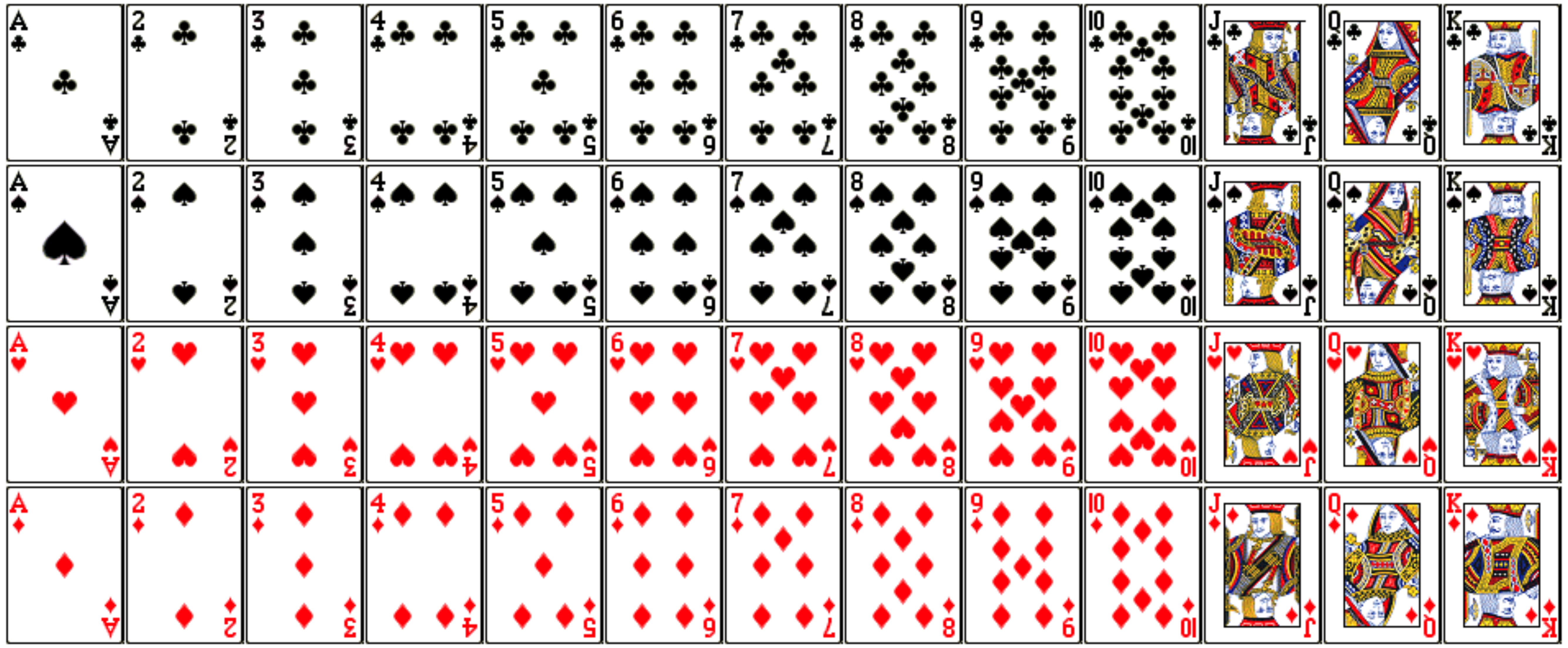
Representation Example 2: Playing Cards

52 cards in 4 suits

- How do we encode suits, face cards?

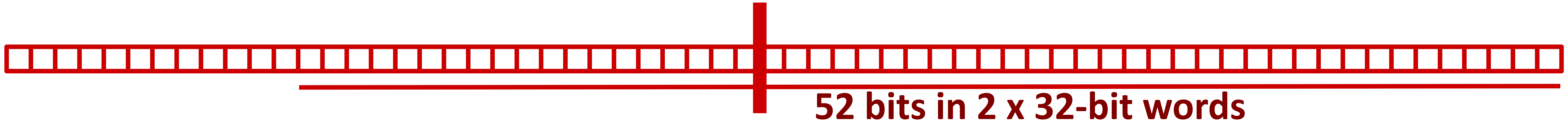
What operations should be easy to implement?

- Get and compare rank
- Get and compare suit



Two possible representations

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

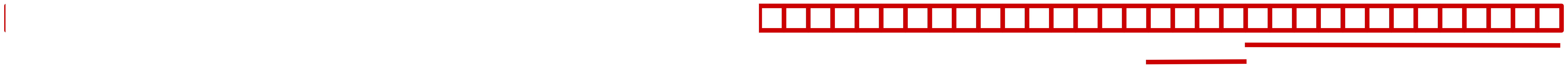


“One-hot” encoding

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



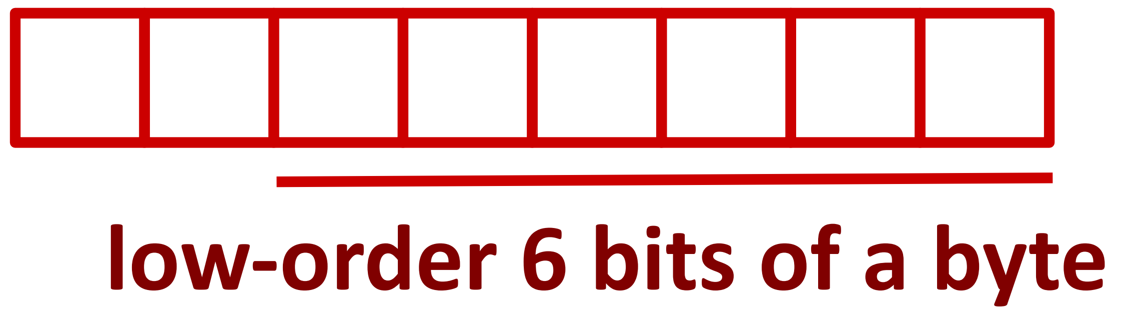
Pair of one-hot encoded values

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

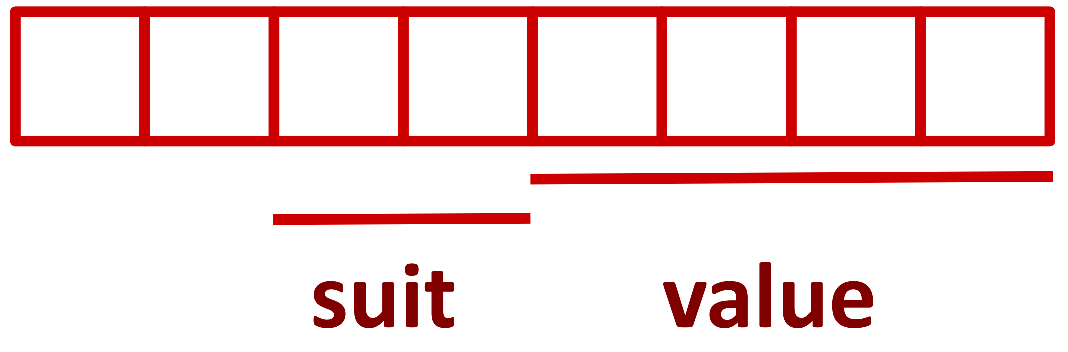


Number cards uniquely from 0

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

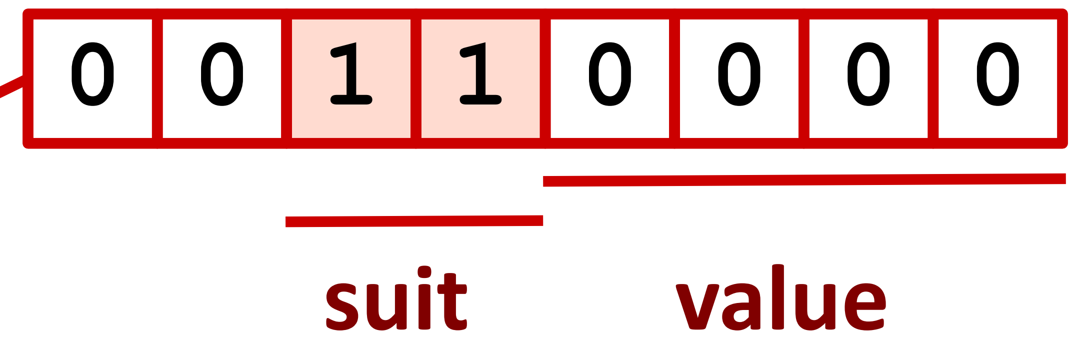
Number each value uniquely

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



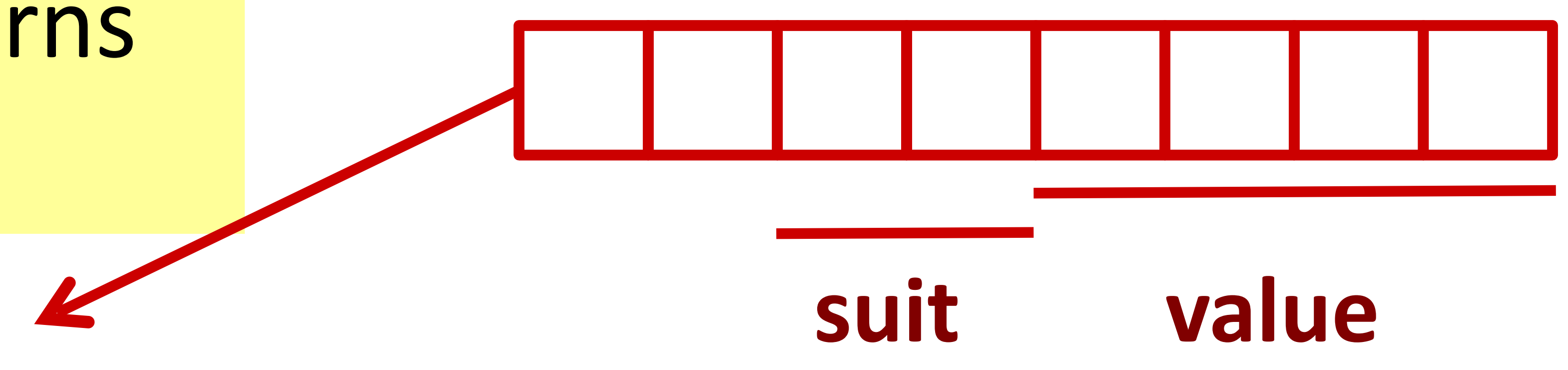

```
#define SUIT_MASK 0x30

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
...
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




```c
#define VALUE_MASK

int greaterValue(char card1, char card2) {




}
```

```c
char hand[5];          // represents a 5-card hand
...
if (greaterValue(hand[0], hand[1])) { ... }
```

Bit shifting

logical shift left 2

x << 2

1 0 0 1 1 0 0 1   x

1 0 | 0 1 1 0 0 1 0 0

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   **x >> 2**

0 0 1 0 0 1 1 0 | 0 1

lose bits on right

arithmetic shift right 2   **x >> 2**

1 1 1 0 0 1 1 0 | 0 1

fill with copies of MSB on left

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 a C function that
extracts the *2nd most significant byte*
from its 32-bit integer argument.

Example behavior:

argument:	0b	01100001	**[01100010]**	01100011	01100100
expected result:	0b	00000000	00000000	00000000	**[01100010]**

All other bits are zero.

Desired bits in least significant byte.

```
int get2ndMSB(int x) {
```