Bit Vectors and Sets

Prof. Darrell Long and Dr. Kerry Veenstra

CSE 13S

Let's Have a Definition: Sets

- Well-defined unordered collections that are characterized by the elements they contain.
- Sets are equivalent *if and only if* they have exactly the same elements.
- Basic relation in set theory is membership.
- Operations:
  - Intersection: $A \bigcap B$
  - Union: $A \bigcup B$
  - Complement: $\bar{A}$ or A^C
  - Difference: $A - B = A \bigcap \bar{B}$

dogs = {  ,  ,  ,  ,  ,  ,  ,  }

Set Operations with Dogs

- Let's first define a universal set of dogs:

```
dogs = { shih tzu, poodle, pitbull, lab,
         corgi, chihuahua, rottweiler, beagle }
```

good_dogs = {  ,  ,  ,  }

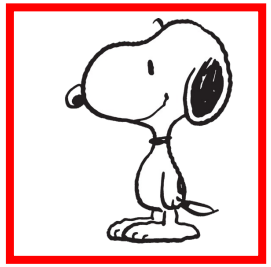
bad_dogs = {  ,  ,  }

good_dogs ∩ bad_dogs = {  }

Set Intersection (A ∩ B)

- The set of elements in A and B.
- `good_dogs = { pitbull, lab, corgi, beagle }`
- `bad_dogs = { shih tzu, poodle, pitbull }`
- `good_dogs ∩ bad_dogs = { pitbull }`

overrated_dogs = {  ,  }

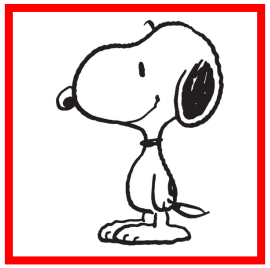
cute_dogs = {  ,  }

overrated_dogs ∪ cute_dogs = {  ,  ,  ,  }

Set Union (A ∪ B)

- The set of elements in A or B.
- `overrated_dogs = { shih tzu, poodle }`
- `cute_dogs = { corgi, beagle }`
- `overrated_dogs ∪ cute_dogs = { shih tzu, poodle, corgi, beagle}`

some_dogs = {  ,  ,  ,  }

cute_dogs = {  ,  }

some_dogs - cute_dogs = {  ,  }

Set Difference (A – B)

- The set of elements in A that aren't in B.
- `some_dogs = { shih tzu, poodle, corgi, beagle }`
- `cute_dogs = { corgi, beagle }`
- `some_dogs - cute_dogs = { shih tzu, poodle }`

dogs = {  ,  ,  ,  ,  ,  ,  ,  }

cute_dogs = {  ,  }

cute_dogs^C = {  ,  ,  ,  ,  ,  }

Set Complement ($\bar{A}$ or A^C)

- The set of elements not in A.
- Also defined as U – A, where U is the universal set.
- `dogs = { shih tzu, poodle, pitbull, lab, corgi, chihuahua, rottweiler, beagle }`
- `cute_dogs = { corgi, beagle }`
- `cute_dogs`^{C} `= { shih tzu, poodle, pitbull, lab, chihuahua, rottweiler }`

Step 1: Choose an Order for Set Members

dogs = {  7 ,  6 ,  5 ,  4 ,  3 ,  2 ,  1 ,  0 }

some_dogs = {  ,  ,  ,  ,  }

some_dogs = 1 0 0 1 1 0 1 1

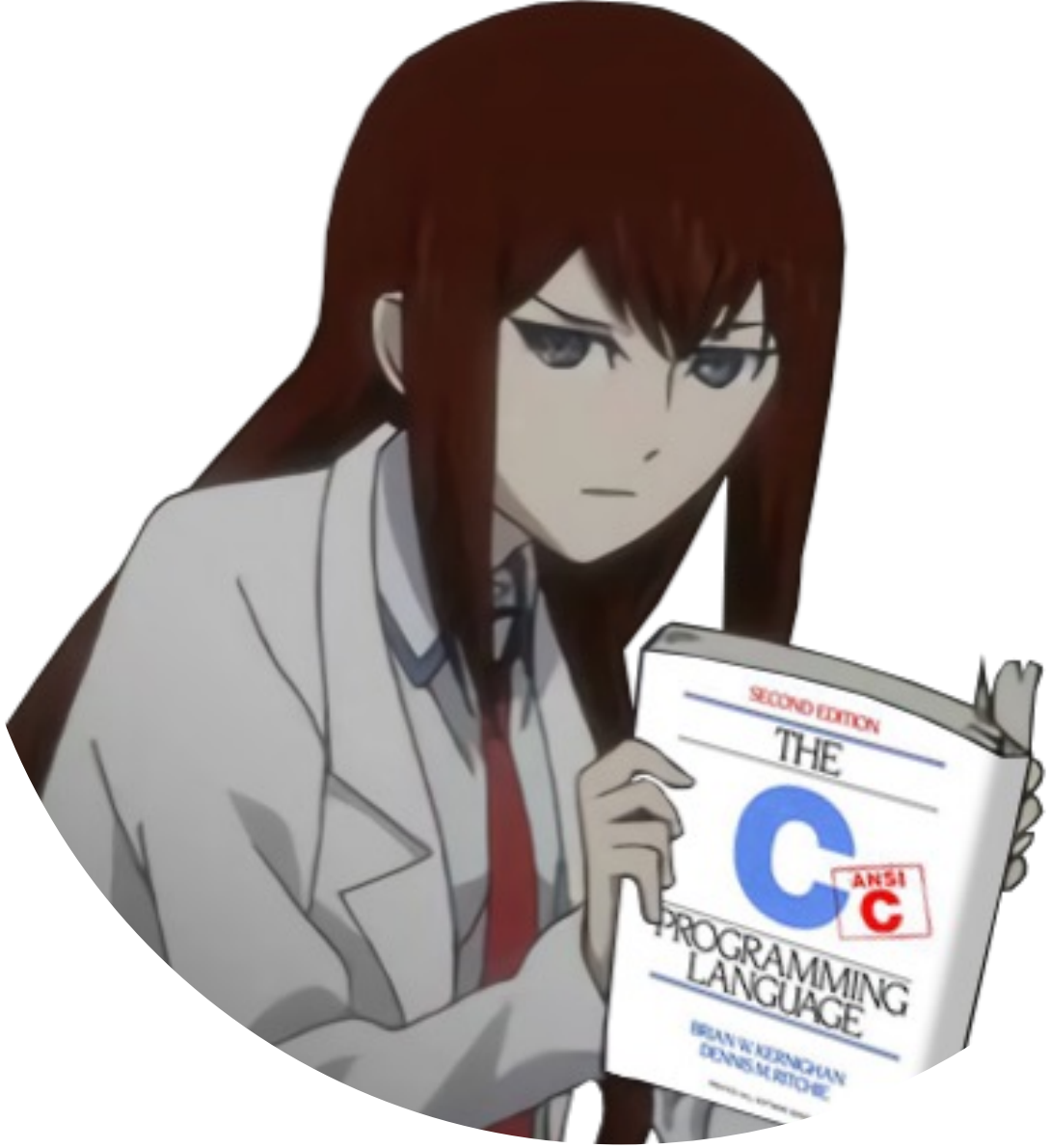
Step 2: In a binary number, use 1 to indicate set membership; 0 otherwise.

Representing Sets with Bits

- Sets can be represented with bits.
- A `0` indicates that the element is not a member of the set.
- A `1` indicates that the element is a member of the set.

Review:
Bitwise Operations in C

&	AND
|	OR
~	NOT
^	XOR
<<	Left shift
>>	Right shift



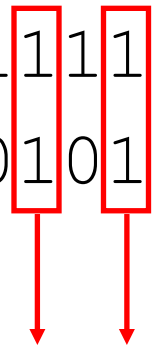
AND (&) Truth Table (per bit)

&	0	1
0	0	0
1	0	1

AND & Example

```
uint32_t a = 0xf0;    // 11110000 binary
uint32_t b = 0x55;    // 01010101 binary

uint32_t c = a & b;   // 01010000 binary
```



Reminder

- Binary/Decimal/Hexadecimal Demonstrator

  https://www.kerryveenstra.com/bin-dec-hex.html

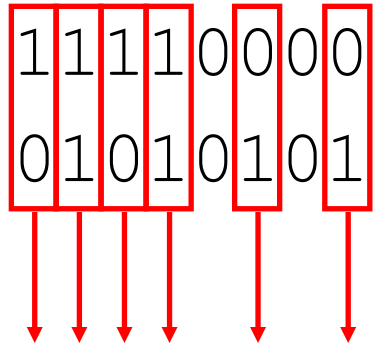
OR (|) Truth Table (per bit)

|	**0**	**1**
0	0	1
1	1	1

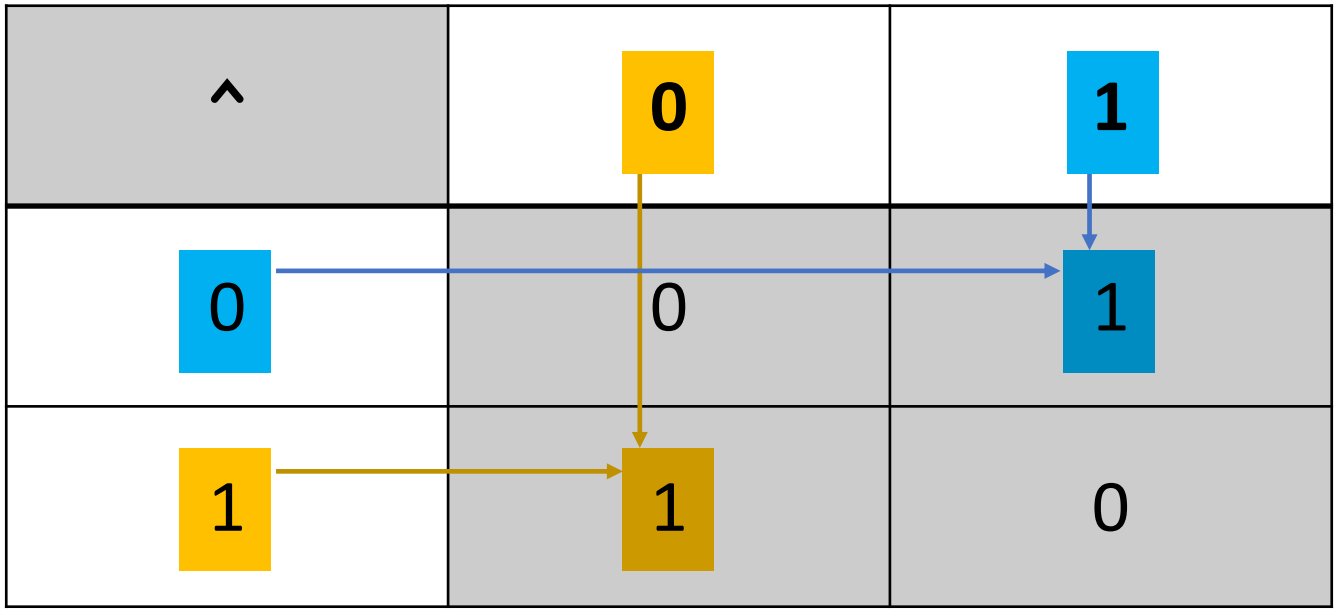
OR | Example

```
uint32_t a = 0xf0;    // 11110000 binary
uint32_t b = 0x55;    // 01010101 binary

uint32_t c = a | b;   // 11110101 binary
```



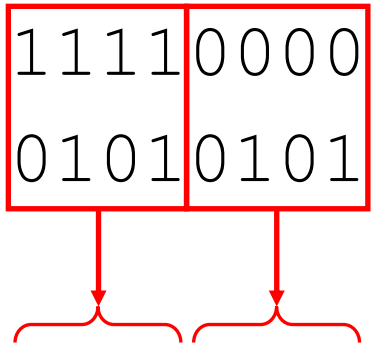
XOR (^) Truth Table (per bit)



XOR ^ Example

```
uint32_t a = 0xf0;    // 11110000 binary
uint32_t b = 0x55;    // 01010101 binary

uint32_t c = a ^ b;   // 10100101 binary
```



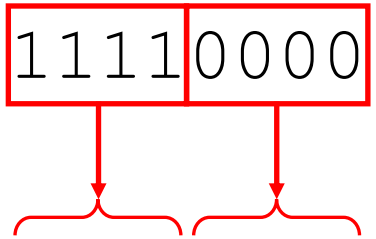
NOT (~) Truth Table (per bit)

a	~a
0	1
1	0

NOT ~ Example

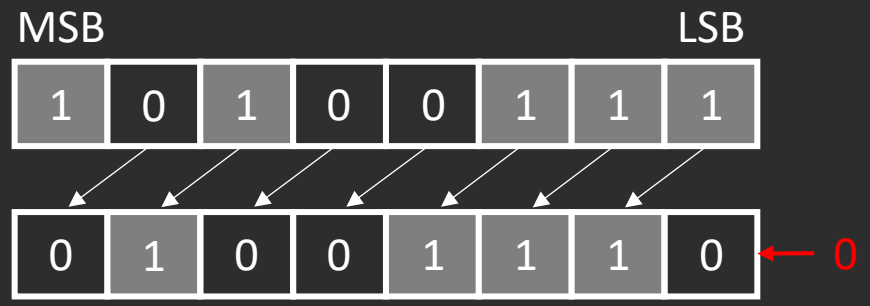
```
uint32_t a = 0xf0;     // 11110000 binary

uint32_t b = ~a;       // 00001111 binary
```

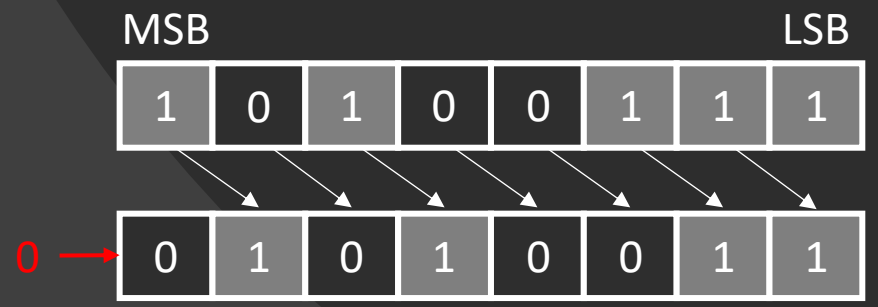


Logical Shift (`a << 1` and `a >> 1`)

- Logical shift left: zeroes are shifted in on the right.

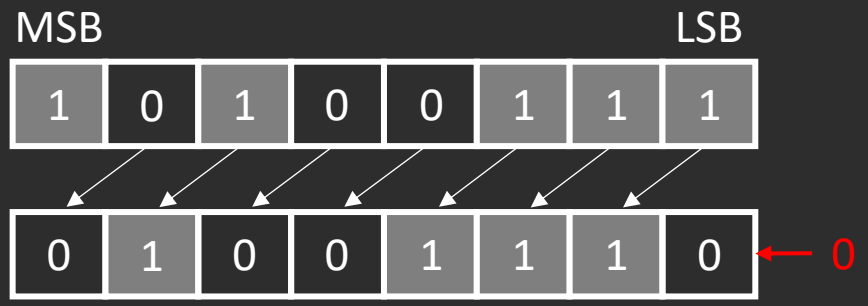


- Logical shift right: zeroes are shifted in on the left.
- **Unsigned integers only**

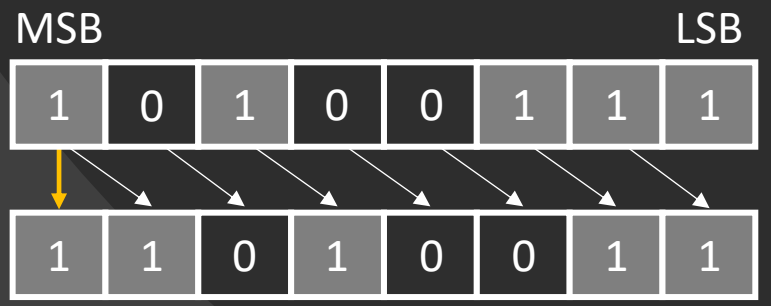


Arithmetic Shift (also `a << 1` and `a >> 1`)

- Arithmetic shift left: zeroes are shifted in on the right.



- Arithmetic shift right: sign bits are shifted in on the left.
  - Maintain the sign (value of the leftmost bit)
- **Signed integers only**



C's Vexatious Right Shift (>>)

- The result of `v >> p` is `v` right-shifted by `p` bit positions.
- If `v` is unsigned, or is signed with a non-negative value…
  - `(v >> p) ==` integral part of the quotient `v / 2^p`
- If `v1` is signed with a negative value…
  - `(v >> p) ==` implementation defined
    - We don't know for sure.

C's Vexatious Left Shift (<<)

- The result of `v << p` is `v` left-shifted by `p` bits.
  - Zeroes are filled.
- If `v` is unsigned…
  - `(v << p) == (v * 2^p) % n`
    - `n` is the (maximum value of the resulting type) + 1.
- If `v` is signed with a non-negative value…
  - `(v << p) == (v * 2^p)`
    - If `(v * 2^p)` is representable in the resulting type
- Else, the behavior is undefined.

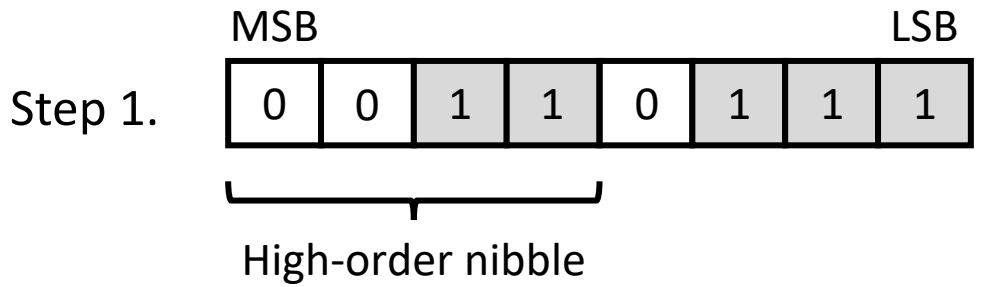
Units of Information

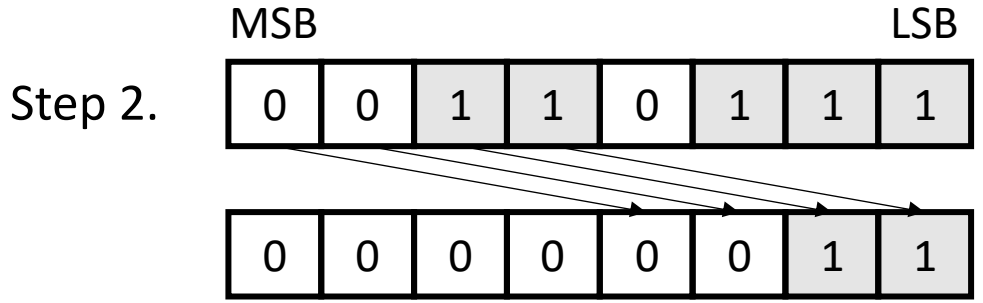
Unit	Size in Bits	Value	Notes
Bit	1	0/1	Smallest
Nibble	4	Hex digit	
Byte	8	ASCII	Smallest addressable
Half word	16		
Word	32		Native size, register length
Long Word	64		Native size, register length

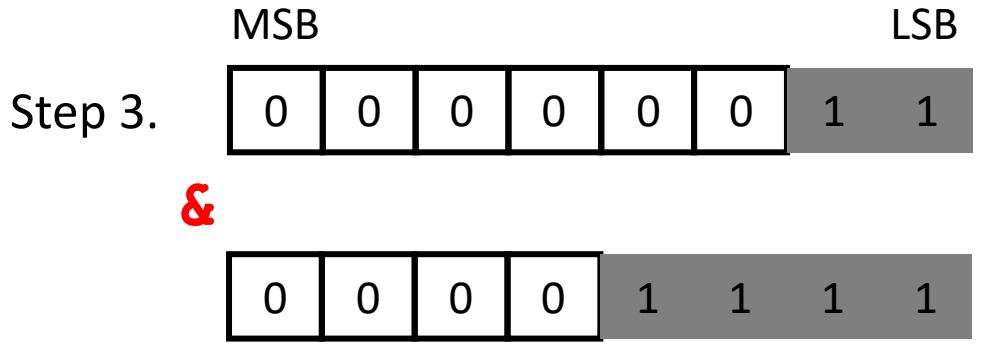
Getting A High-Order Nibble

Step

1. A high-order nibble in a byte means the most significant 4 bits.
2. Bit-shift right 4 times so that the high-order nibble takes the place of the low-order nibble
3. AND with `0x0F`



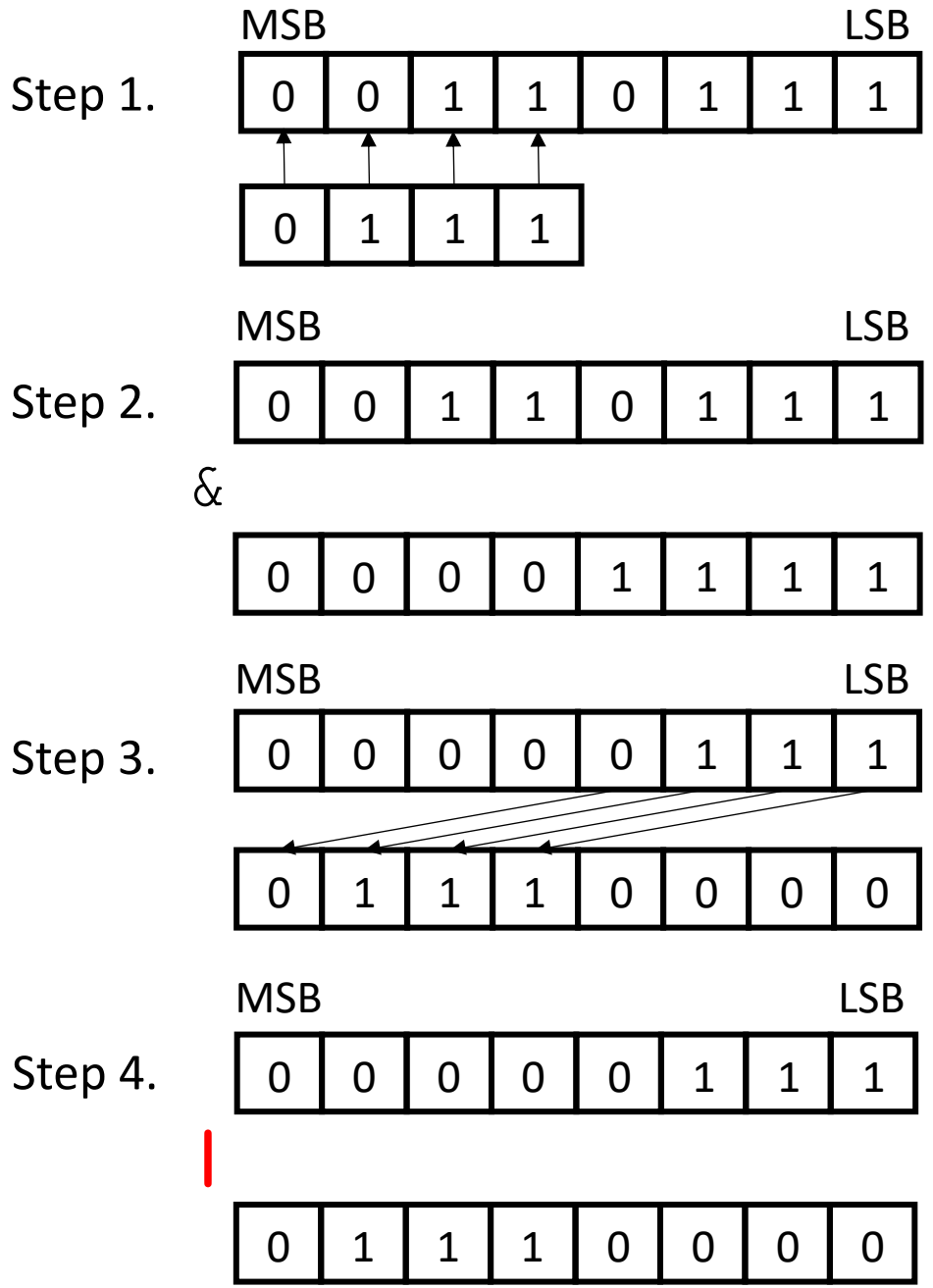




Setting A High-Order Nibble

Step

1. We want to place a nibble into the higher-order bits of a byte
2. AND byte with `0x0F`
3. Bit-shift nibble left 4 times
4. OR byte with bit-shifted nibble



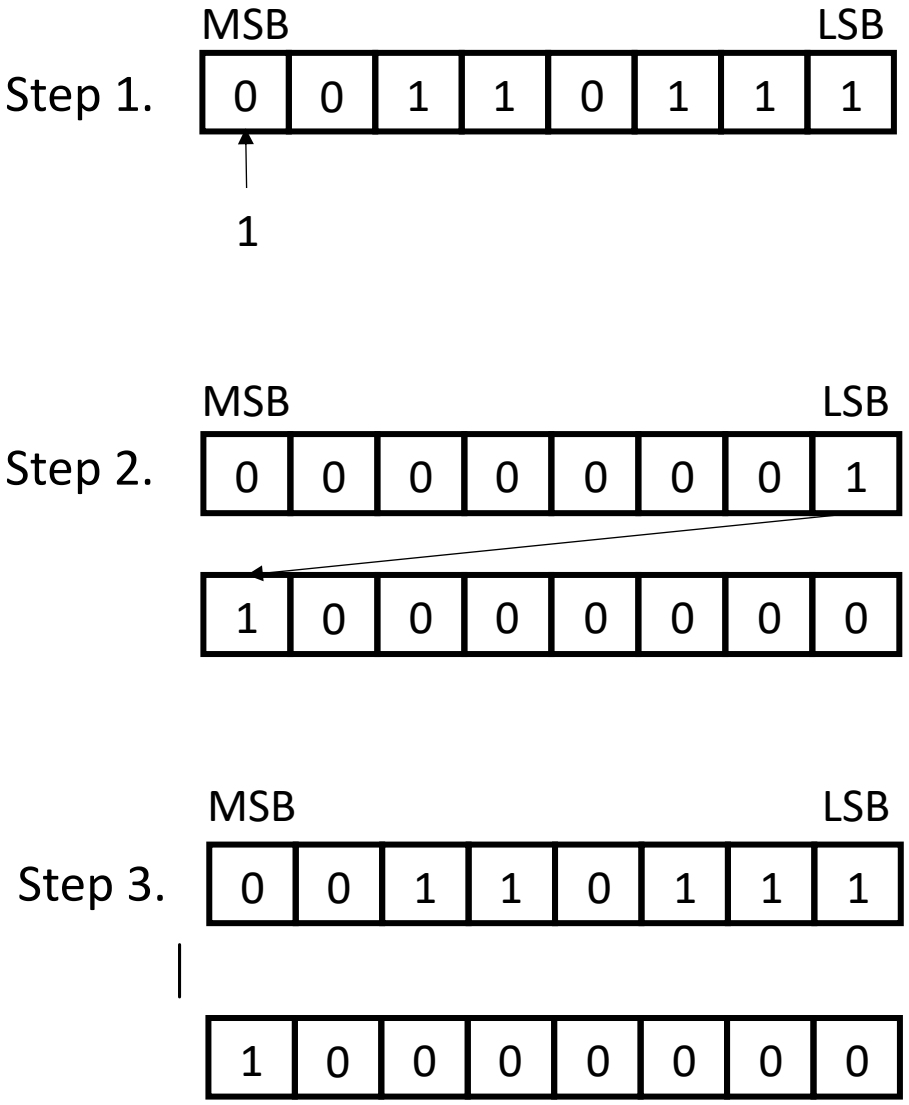
Choose a Bit Position for Each Set Member Using the C Programming Language

```
#define BEAGLE    0
#define ROTWEILER 1
#define CHIHUAHUA 2
#define CORGI     3
#define LAB       4
#define PITBULL   5
#define POODLE    6
#define SHIH_TZU  7
```

How to Set A Bit?

Step

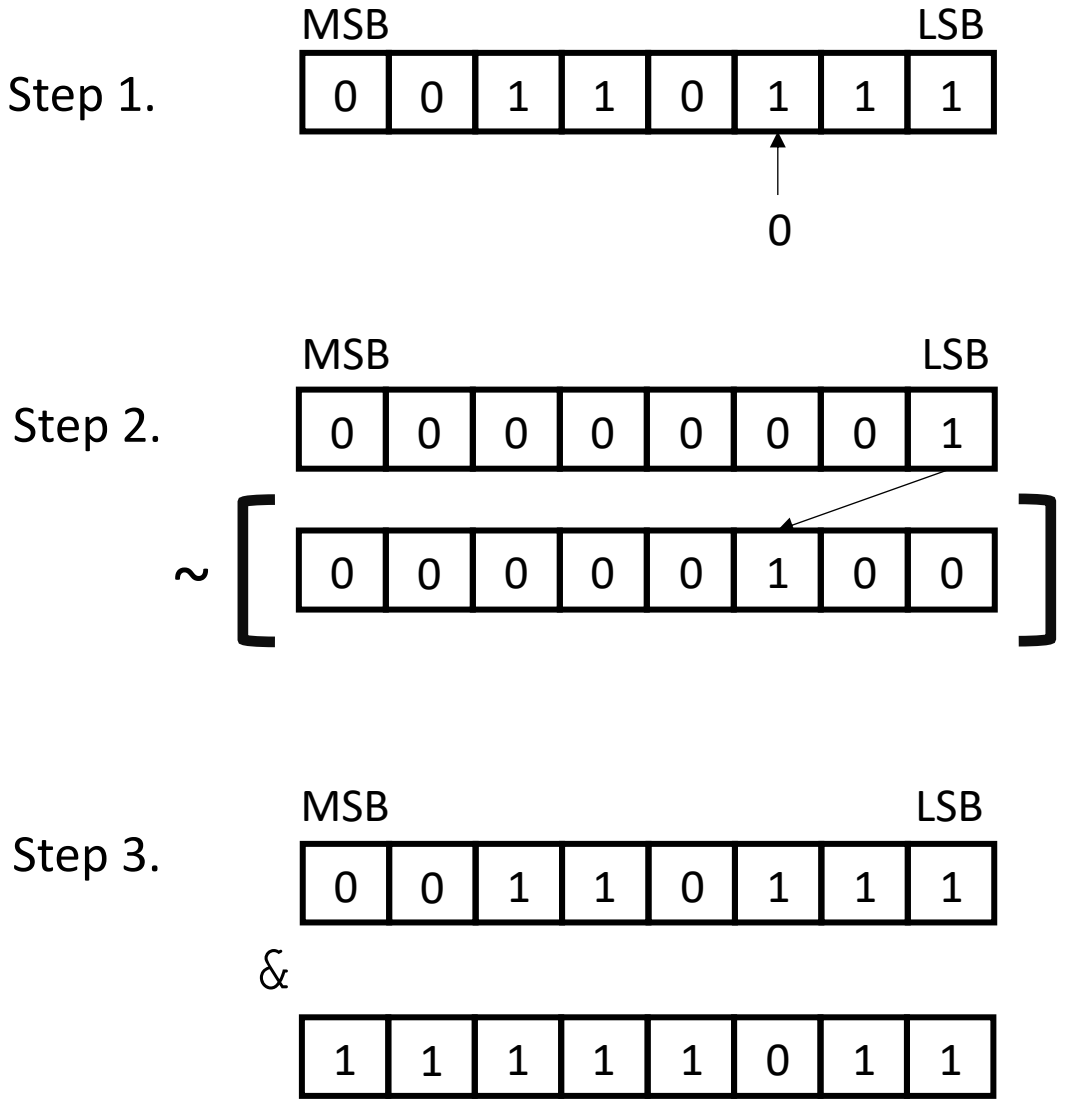
1. We want to set the bit at index 7 in a byte.
2. Take another byte with the bit at index 7 set
   - Can do this by shifting `0x1` left 7 times.
3. OR the bytes together to set the bit.



Clearing A Bit

Step

1. We want to clear the bit at index 2 in a byte.
2. Take another byte with all bits set *except* the bit at index 2.
   - Can do this by shifting `0x1` left 2 times and taking the bitwise NOT of the result.
3. AND the bytes together to clear the bit.



Getting A Bit

Step

1. We want to get, or return, the value of the bit at index 4 in a byte.
2. Take another byte with the bit at index 4 set.
   - Can do this by left-shifting `0x1` 4 times.
3. AND the bytes together to mask every bit except the bit at index 4.
4. Right-shift the AND-ed result 4 times to get the value.