Signed Binary Numbers

Unsigned *Binary Numbers*

We write numbers with as many digits as we need:

`0, 99, 65536, 15000, 1979, …`

However, memory locations and CPU registers always hold a constant, fixed number of bits:

- 4 bits: **`0000, 1010, 0111, 1101, …`**
- 8 bits: **`00000000, 11011001,`**
  `00000110, 10111010, …`

This means there is a biggest *possible* number:

- 4 bits: $\mathbf{1111_2}$ **`= 15`** is the biggest.
- 8 bits: $\mathbf{1111\ 1111_2}$ **`= 65535`** is the biggest.

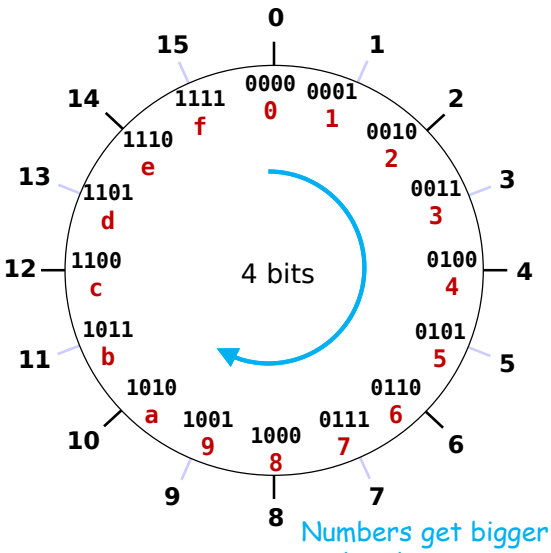
Using n bits allows 2^n distinct values in the range:

`{ 0 .. 2`^{n}**`-1 }`**

4-bit Unsigned Values …

- $2^4 = 16$ distinct values: {***0***, ***1***, …, ***15***}
- Numbers start with $0000b = \mathbf{0}$
- Numbers "wrap around": after $1111b = \mathbf{15}$, back to $0000b = \mathbf{0}$ again

this is
Unsigned Binary



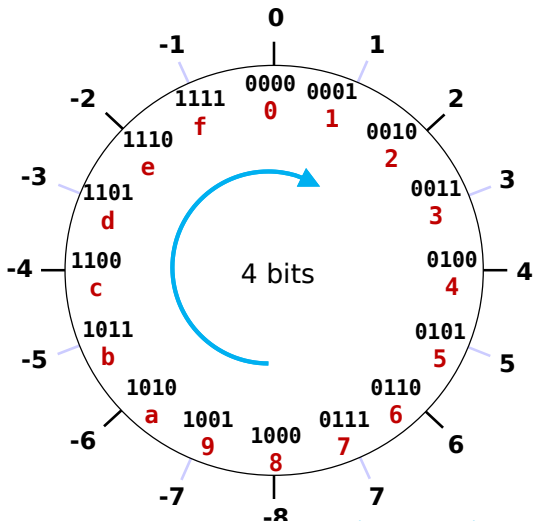
Positive and Negative Numbers

- Subtraction leads to the possibility of numbers smaller than 0:
  - 9 - 7 = 2, while 7 - 9 = -2
- How do we represent negative numbers in binary?
  - Fixed number of bits → finite set of values
  - 4 bits: **Unsigned** uses this set of 16 values:
    { 0 .. 2^4-1 } = {0, 1, …, 15}
    but this leaves *no room for negatives*
- Use a set of 16 values that includes negatives:
  - 4 bits: use values in the set
    { -2^3 .. 2^3-1 } = {-8, -7, …, 7 }

… 4-bit Signed Values

- Start with **0000** ⇔ ***0***, count in both directions
- Negatives start at **-1**
- Numbers "wrap around" at ***-8***

this is
2's Complement



2's-Complement and Hexadecimal

- Hexadecimal – special case
  - Used as "short-hand" representation of binary
- What about 2's-complement (signed) binary?
  - Ignore the sign, just express the binary as hex digits
  - "**Sign-extend**" to get 4 bits per hex digit
- Examples:
  - 01101101 ⇿ 6d (0x6d in C notation)
  - 11010011 ⇿ d3 (0xd3 in C notation)
  - 011011 ⇾ **00**011011 ⇿ 1b (0x1b in C notation)
  - 11010 ⇾ **111**11010 ⇿ fa (0xfa in C notation)

2's Complement Characteristics

- All negative values have a one in the leftmost bit
- All positive values have a zero in the leftmost bit
  - This implies that 0 is a positive value
- Negated zero is still zero (this is a Good Thing!)
- The negative values "go further" than the positive values
  - 4 bits:
    -8, -7, -6, -5, -4, -3, -2, -1, 0, +1, +2, +3, +4, +5, +6, **+7**
  - 8 bits:
    -128, -127, .. 0, .. 126, **+127**
  - 32 bits:
    -2147483648 .. 0, .. **+2147483647**
- *Arithmetic is the same as for unsigned binary!*

Negation

- Negation has to be *invertible:*
  $f(x) = -x$
  $f(-x) = x$
- The negation operation must work on the *bit patterns*.
- An algorithm for negations:
  (1) take the bitwise inverse of the number
  (2) add 1 to the inverse
  - Symbolically:

$$-x = \bar{x} + 1 \quad (or, \quad -x = x' + 1)$$

Some examples

```
   7 =   0111
invert:  1000
add 1:  +   1
         1001
```

```
  -7 =   1001
invert:  0110
add 1:  +   1
         0111
```

```
   0 =   0000
invert:  1111
add 1:  +   1
         0000
```

```
   4 =   0100
invert:  1011
add 1:  +   1
         1100
```

```
  -4 =   1100
invert:  0011
add 1:  +   1
         0100
```

```
  -8 =   1000
invert:  0111
add 1:  +   1
         1000
```

Oops!

```
   1 =   0001
invert:  1110
add 1:  +   1
         1111
```

```
  -1 =   1111
invert:  0000
add 1:  +   1
         0001
```

2's-Complement Sign Extension

- Sign extension is the process of extending a value to use more bits
  - The value must remain the same!
- Using more bits in unsigned binary:
  - Just add leading (leftmost) zeroes.
- Using more bits in 2's complement:
  - Copy the leftmost bit into the new positions
  - Positive values stay positive
  - Negative values stay negative

```
  +7 =  0000 0111
invert:  1111 1000
add 1:  +         1
  -7 =  1111 1001
```

```
  -7 =  1001
extend to
eight bits:
      1111 1001
```

Signed Conversions

Algorithms to convert between Base-10 and Base-2 signed numbers

Decimal to Binary

- Convert in two (and a half) steps:
  1. Work with absolute value of base-10 number
  2. Convert absolute value to binary
     - maintain a leading zero, because this is a positive number!
  3. **IF** the original number was negative, **THEN** negate the binary value

-17 → 17 → 010001 → 101110 → 101111

17 → 17 → 010001 → *(done)*

Decimal to Binary - Code

```
int main(int argc, char **argv)
{
   char bitstring[65], temp[65] ;
   int value;

   while ( *++argv != NULL) {
      value = atoi( *argv);

      if (value < 0) {
         unsigned2binary_with_leading_0( temp, -(value) );
         negate(bitstring, temp);
      } else
         unsigned2binary_with_leading_0( bitstring, value );

      printf("%i --> %s\n", value, bitstring);
   }
}
```

important

Binary to Decimal - 1

- Invert the decimal-to-binary approach:
  1. **IF** binary number is negative,
     THEN negate to get positive value
  2. Convert to decimal as if unsigned
  3. **IF** original binary number was negative,
     THEN negate the decimal value

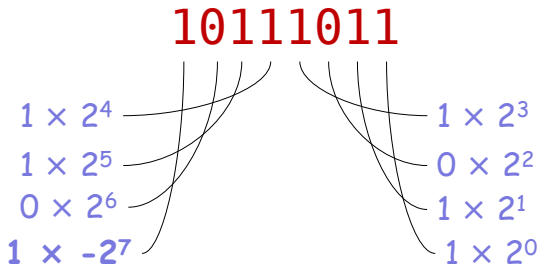
011010 → 011010 → 26 → *(done)*

111010 → 000110 → 6 → -6

This approach is easy for machines/programs

Binary to Decimal - 2

- Alternative approach:
- Convert as if unsigned, BUT
- if most significant bit is 1, SUBTRACT its value instead of adding it



This approach is easier for humans

Binary Negation and Coding

- Negating a binary value represented by a bitstring:
  a) copy the bitstring, with 0s and 1s interchanged
  b) add one to the result…
- Example: "110010100000"
  - "1100 1010 0000" → "0011 0101 1111"
  - "0011 0101 1111" + 1 → "0011 01**10 0000**"
    - rightmost "1" bits changed back to "0"
    - rightmost "0" bit changed back to "1"
    - remaining (more-significant) bits unchanged

Use spaces, underscores, etc. to make bitstrings more readable

Coding approach for "adding 1"

- Start at the *rightmost* end of the inverted bitstring
  - working left, from least-significant bit to most-significant bit
- Copy each bit to a new string
  - while rightmost bits are "1":
    a) turn each "1" bit back into a "0" bit
    b) turn rightmost "0" bit back into a "1" bit
  - copy remainder of bits unchanged
- Remember to add a '\0' bit to the end of the string!

Two Versions of "add_one()" Function

```
/*
* Two versions of a routine to "add 1" to a bitstring.
*  1st version uses a while loop, and a switch statement
*  2nd version uses a for loop, and an if-else if-else statement
* Both versions preserve any bits other than "0" and "1",
* so spacing characters are unchanged.
* Example:
*  "1100 0101" becomes "1100 0110", space character is preserved.
*/
void add_one_whileloop(char *dst, char *dst_end)
{
   while (dst_end >= dst) {
      switch (*dst_end) {
         default:
            break;  // do nothing to this char
         case '1':
            *dst_end = '0'; // flip bit back to "0"
            break;
         case '0':
            *dst_end = '1'; // flip bit back to "1"
            return;        // ...and quit the routine!
      }
      dst_end--;
   }
}
```

```
void add_one_forloop(char *dst, char *dst_end)
{
   for ( ; dst_end >= dst; dst_end--) {
      if (*dst_end == '1')
         *dst_end = '0';
      else if (*dst_end == '0') {
         *dst_end = '1';
         break;  // ...out of for loop
      } else
         ;      // leave other char's unchanged
   }
}
```

Other Representations

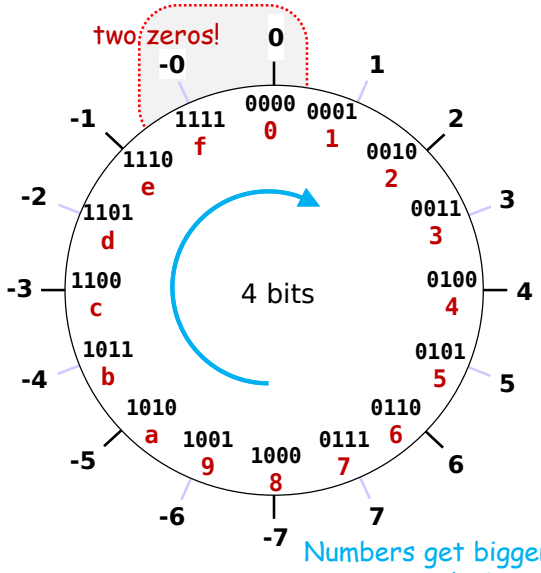
Other ways to represent negative numbers on binary computers

Other ways to make negative numbers:

How about... just invert the bits (without adding 1)?

Two zeros.
Arithmetic is tricky.
Discontinuity at
7 ↔ -7

this is
1s' Complement

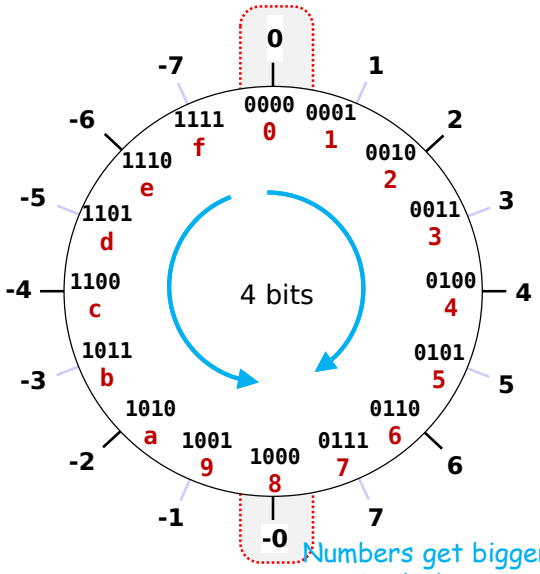


Other ways to make negative numbers:

How about... just set a *sign* bit?

Two zeros, two discontinuities. Numbers get "bigger" in two directions.

this is
Sign-Magnitude



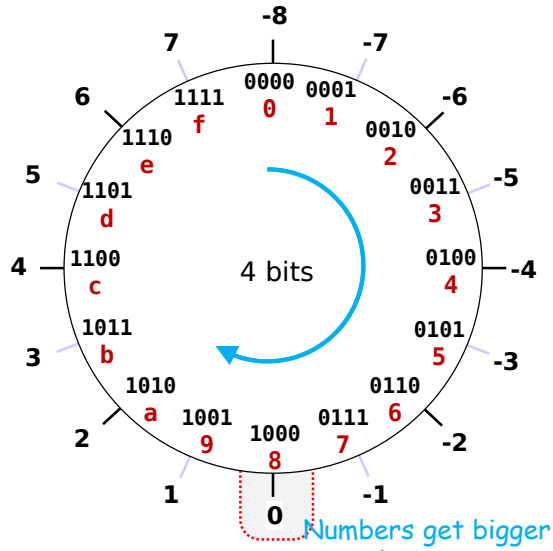
Numbers get bigger in both directions

Other ways to make negative numbers:

How about... shift by a constant (such as 8) before converting to binary?

Similar to 2's-complement, but sign bit is reversed.

this is
Excess-8



Numbers get bigger going clockwise again

Convert Between Decimal and Excess-N

- Decimal → Excess-N:
  1. **Add** N to the **decimal** value
  2. Convert the result as if for unsigned binary

-7 → -7 + 8 → 1 → 0001

- Excess-N → Decimal:
  1. Convert the binary value as if for unsigned binary
  2. **Subtract** N from the result

1101 → 13 → 13 – 8 → +5

Decimal <--> Excess-8

Summary: Other Representations

- 1s' complement
  - Two representations of zero
  - Arithmetic works differently than for unsigned binary
- Sign-magnitude
  - Two representations of zero
  - Not monotonic – arithmetic is very different

These have been used in the past, but no longer

- Excess-N
  - N can be chosen arbitrarily
  - Arithmetic works the same as for unsigned binary
  - Single representation of zero
  - Negation is more work
  - Used for exponents in IEEE-754 floating-point notation