COMP 1521

WEEK 5 TUTE

CONTENTS

Announcements

- Lab start this week!
  - Lab5 has been released, due on **Week 7 Monday 12:00:00 (midday)**

- weekly quiz will be **due Week 7 Thursday 21:00:00**
- **Assignment1** will due on Week 5 Friday 18:00:00

Different Binary Representations

Binary	Octal	Hexadecimal	Decimal
10100111	247	A7	167

Different Binary Representations

decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2)

Binary	Octal	Hexadecimal	Decimal
1 bit = 1 digit	3bits = 1 digit	4 bits = 1 digit	
10100111	247	A7	167

```
10100111
= 10 100 111
= 2 4 7
= 247
```

split into groups of 3

10100111
= 1010 0111
= A 7

split into groups of 4

Different Binary Representations

Binary

1 bit = 1 digit

10100111

Octal

3bits = 1 digit

247

10100111
= 10 100 111
= 2 4 7
= 247

Hexadecimal

4 bits = 1 digit

A7

10100111
= 1010 0111
= A 7

Decimal

167

from octal:

$$247 = 2 * 8^2 + 4 * 8^1 + 7 * 8^0 = 167$$

from hex:

$$A7 = A * 16^1 + 7 * 16^0 = 167$$

Different Binary Representations

Binary

1 bit = 1 digit

10100111

Octal

3bits = 1 digit

247

10100111
= 10 100 111
= 2 4 7
= 247

Hexadecimal

4 bits = 1 digit

A7

10100111
= 1010 0111
= A 7

Decimal

167

from octal:

$$247 = 2 * 8^2 + 4 * 8^1 + 7 * 8^0 = 167$$

from hex:

$$A7 = A * 16^1 + 7 * 16^0 = 167$$

IN C:

0b10100111

0247

0xA7

167

Converting Decimal to Binary

100

1: decompose it into powers of two
2: the powers are just the position of ‘1’s in bits!

Converting Decimal to Binary

1

- 100 = 64 + 32 + 4

1: decompose it into powers of two

2: the powers are just the position of ‘1’s in bits!

Converting Decimal to Binary

1. 100 = 64 + 32 + 4

= 2^6 + 2^5 + 2^2

1: decompose it into powers of two

2: the powers are just the position of ‘1’s in bits!

Converting Decimal to Binary

1. 100 = 64 + 32 + 4

   = 2^6 + 2^5 + 2^2

2. = 0110 0100

1: decompose it into powers of two

2: the powers are just the position of ‘1’s in bits!

Converting Decimal to Binary

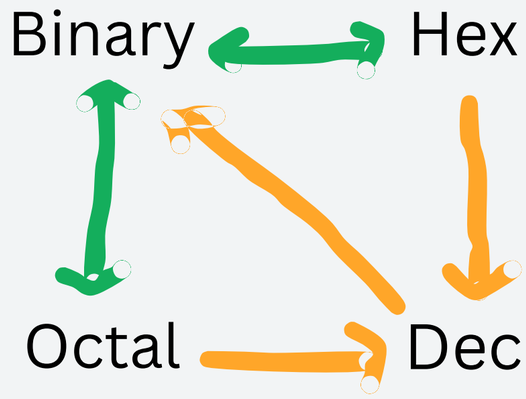
1. 100 = 64 + 32 + 4
   = $2^6 + 2^5 + 2^2$
2. = 0110 0100

7654 3210

1: decompose it into powers of two

2: the powers are just the position of ‘1’s in bits!

Conversion graph



Stay on the path :)

Some hex good to knows.

- every **digit** is 4 bits.

0xA023FD is a 24 bit number.

  - **F** -> loaded one’s (1111)
- **7** -> loaded one’s except the first (0111)
- digits are independent of each other in binary.

  0xF7 is 1111 0111

  F 7

  13(dec) in binary is 1101. -> not independent!! (decimal bad)

q2 a) d) h) g)

Bitwise operations

```
x & y     // bitwise and
x | y     // bitwise or
x ^ y     // bitwise exclusive-or (XOR)
~ x       //  bitwise not
x << n    // left shift
x >> n    // right shift
```

Bitwise operations

```
x & y     // bitwise and
x | y     // bitwise or
x ^ y     // bitwise exclusive-or (XOR)
~ x       //  bitwise not
x << n    // left shift
x >> n    // right shift
```

```
AND | 0  1
----|------
  0 | 0  0
  1 | 0  1
```

```
 OR | 0  1
----|------
  0 | 0  1
  1 | 1  1
```

```
00100111 << 2
--------
10011100
```

```
00100111 << 8
--------
00000000
```

Bitwise operations

```
x & y     // bitwise and
x | y     // bitwise or
x ^ y     // bitwise exclusive-or (XOR)
~ x       //  bitwise not
x << n    // left shift
x >> n    // right shift
```

```
AND | 0  1
----|------
  0 | 0  0
  1 | 0  1
```

“if both are 1, enter 1”

```
 OR | 0  1
----|------
  0 | 0  1
  1 | 1  1
```

“if either are 1, enter 1”

```
00100111 << 2
--------
10011100
```

```
00100111 << 8
--------
00000000
```

left shift by 1 bit multiplies your number by 2

left shift by n bits multiplies your number by 2^n

Why bitwise?

C does not have functions to interpret bit -
you cannot access the third bit of 10100110 by doing number[2]. It is not an array

Instead you must extract it using bitwise operations

Why bitwise?

C does not have functions to interpret bit -
you cannot access the third bit of 10100110 by doing number[2]. It is not an array

Instead you must extract it using bitwise operations

```
  10100 1 10
& 00100000
-----------
  00100000
```

(because 1 & 1 = 1 but 1 & 0 = 0)

AND	0	1
0	0	0
1	0	1

Why bitwise?

C does not have functions to interpret bit -
you cannot access the third bit of 10100110 by doing number[2]. It is not an array

Instead you must extract it using bitwise operations

```
  10100 1 10
& 00100000
-----------
  00100000
```

(because 1 & 1 = 1 but 1 & 0 = 0)

AND	0	1
0	0	0
1	0	1

result = 00100000 >> 5
result = 0000 0001.

When to use each operator

```
x & y       // bitwise and
x | y       // bitwise or
x ^ y       // bitwise exclusive-or (XOR)
~ x         //  bitwise not
x << n      // left shift
x >> n      // right shift
```

read/copy or remove bits in a number (010 -> 000)

add 1 bits to a number (000 -> 010)

Flipping bits (010 -> 101)

Flipping logic (0 -> 1) or removing 1 bits (010 -> 000)

positioning your mask

positioning your mask

q5

5. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

a, d, f

5. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

a) This guarantees that LOCKED is set to 1 while leaving other bits unchanged.

5. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

a) This guarantees that LOCKED is set to 1 while leaving other bits unchanged.

Mark the device as locked for reading bytes

We need to set the following flags **without changing other bits**:

- ✅ Set `READING to 1 (indicates the device is being read)`
- ✅ Set `AS_BYTES to 1 (indicates byte-wise access mode)`
- ✅ Set `LOCKED to 1 (locks the device)`
- ⚠️ `Leave all other bits unchanged`

5. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

0 | 1 = 1 (**Forces the bit to 1 if it was 0**)
1 | 0 = 1 (**Keeps the bit 1 if it was already 1**)

a) This guarantees that LOCKED is set to 1 while leaving other bits unchanged.

device = (READING | AS_BYTES | LOCKED);

5. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

d) Clearing the LOCKED Flag - This ensures that LOCKED is now 0 while other bits remain unchanged.

5. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

1 & 1 = 1 (**Keeps the bit 1 if it was already 1**)
1 & 0 = 0 (**Forces the bit to 0 if it was 1**)

d) Clearing the LOCKED Flag - This ensures that LOCKED is now 0 while other bits remain unchanged.

device = device & ~LOCKED

5. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

F) This means if READING was set, it is now cleared and WRITING is set, effectively swapping modes

5. Consider a scenario where we have the following flags controlling access to a device.

```
#define READING   0x01
#define WRITING   0x02
#define AS_BYTES  0x04
#define AS_BLOCKS 0x08
#define LOCKED    0x10
```

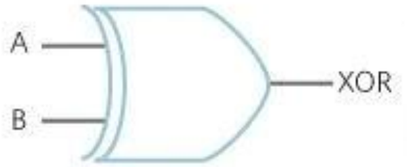
The flags are contained in an 8-bit register, defined as:

```
unsigned char device;
```

Write C expressions to implement each of the following:

a. mark the device as locked for reading bytes
b. mark the device as locked for writing blocks
c. set the device as locked, leaving other flags unchanged
d. remove the lock on a device, leaving other flags unchanged
e. switch a device from reading to writing, leaving other flags unchanged
f. swap a device between reading and writing, leaving other flags unchanged

$$X = A \oplus B$$



A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

When we want to switch (toggle) a bit between 0 and 1, we use bitwise XOR (^).
1 ^ 1 = 0 (**Flips 1 to 0**)
0 ^ 1 = 1 (**Flips 0 to 1**)

F) This means if READING was set, it is now cleared and WRITING is set, effectively swapping modes

device = device ^ (READING | WRITING);

Masks

Mask is an arbitrary binary value we use to help us in bitwise operations.

Common masks:

```
int mask = 1
mask = mask << x
```

shifting a bit mask of “1” to get to a certain bit

When to use?

When trying to address specific bits!

e.g. tell me if the following is negative or positive:
0b1011 0100

Common masks:

```
int mask = 0xFF
//mask = 1111 1111
```

using “F” to get “loaded 1’s“

When to use?

When trying to address specific bytes!

e.g. convert endian-ness (flip biggest bytes to smallest bytes)

01110010 10000000 -> 10000000 01110010

q3 a) to e)

Integers

uint64_t

u = unsigned

64 = 64 bits (8 bytes) long

int16_t

signed

16 = 16 bits (2 bytes) long

q1

Negative Integers

- for an n-bit binary number the representation of $-b$ is $2^n - b$

Two's complement binary	Decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

e.g. for a 4 bit number,

$-b$ is $2^n - b$

$-7 = 2^4 - 7 = 9$

9 is of course 0b1001

- The leftmost bit in a binary number is called the **Sign Bit**.
- **0** represents a positive number.
- **1** represents a negative number.

How to Convert a Negative Number to Two's Complement

1. Start with the positive binary representation.
2. **Invert all bits** (flip 0s to 1s and 1s to 0s).
3. **Add 1** to the result.

Two's complement binary	Decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

Example: Convert **-7** in 4-bit representation

1. **7 in binary (4-bit)**: 0111
2. **Invert bits**: 1000
3. **Add 1**: 1001
    1. **Final result**: 1001 (which represents -7 in two's complement).

Negative Integer padding (optional for week 5)

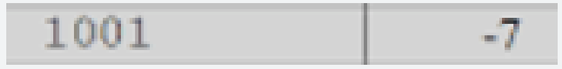
- Consider the scenario: you have a 4 bit negative integer (-7) but you want to make it 8 bits long. How can you convert it?

Consider the positive integer case first:

0111 -> 0000

0111

What about for -7?

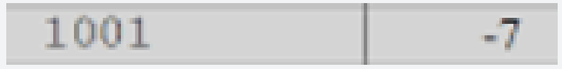


Negative Integer padding

- Consider the scenario: you have a 4 bit negative integer (-7) but you want to make it 8 bits long. How can you convert it?

Consider the positive integer case first:

0111 -> 0000 0111

What about for -7? 

To convert it, we use “padding”. For positive numbers we pad with “0s” but for negative numbers we pad with 1's

-7 = 1111 1001 = 2^8 - 7

BCD - binary coded decimal

BCD is where each byte is interpreted as 1 digit in decimal.

```
$ ./bcd 0x07
7
$ ./bcd 0x0102              # note: 0x0102 == 258 decimal
12
$ ./bcd 0x0402              # note: 0x0402 == 1026 decimal
42
```

BCD - binary coded decimal

BCD is where each byte is interpreted as 1 digit in decimal.

```
$ ./bcd 0x07
7
$ ./bcd 0x0102              # note: 0x0102 == 258 decimal
12
$ ./bcd 0x0402              # note: 0x0402 == 1026 decimal
42
```

In the 0x0402 case, we separate it into its bytes.

0x 04 02

  4 2

and thus we have 42!

BCD - binary coded decimal

BCD is where each byte is interpreted as 1 digit in decimal.

```
$ ./bcd 0x07
7
$ ./bcd 0x0102              # note: 0x0102 == 258 decimal
12
$ ./bcd 0x0402              # note: 0x0402 == 1026 decimal
42
```

In the 0x0402 case, we separate it into its bytes.

0x 04 02
     4   2

and thus we have 42!

Note that all BCDs must be between 0 and 9 inclusive - as that is the limit of a decimal.
0x0A02 would not be a valid BCD as A(10) cannot be represented as a decimal digit.

q4,
q8

```
uint32_t six_middle_bits(uint32_t u) { return (u >> 13) & 0x3F;}
```