Bits & File I/O

Intro to Computer Systems, Fall 2021

Instructor: Travis McGaha

Upcoming Due Dates

- LC4 Simulator HW (Part 1)
  - Due Friday @ 11:59 pm

Any Logistical Questions?
Thoughts? Feelings?

Anything?

Outline

- **Bits & Bytes**
  - **Binary & Hexadecimal**
  - **Endianness**
  - **Bit manipulation**
- File I/O
- Hexdump demo

Bits & Bytes Reminder

- A bit is a singular 1 or 0 that is used by the computer to represent data
- A byte is a collection of 8 bits
  - In most systems a byte is the smallest addressable unit
  - (In LC4 everything is 16 bits… which is 2 bytes)
- There most/least significant bits/bytes.
  - These are the bits/bytes that would most greatly affect the magnitude of the data if we read the bits/bytes as a number
    - E.g the most significant bit (msb) in <u>0</u>1101100 is '0'
- EVERYTHING IS STORED AS BITS IN A COMPUTER

The Meaning of Bits

- *A sequence of bits can have many meanings!*

- Consider the hex sequence 0x4E6F21
  - Common interpretations include:
  - The decimal number 5140257
  - The characters "No!"
  - The background color of this slide
  - The real number 7.203034 $\times 10^{-39}$

- A series of bits can also be code!

- It is up to the program/programmer to decide how to ***interpret*** the sequence of bits

Hexadecimal

- Base 16 representation of numbers
- Allows us to represent binary with fewer characters
  - 0b11110011 == 0xF3
    ^ **b**inary ^ he**x**

- In C, you can **not** define binary literals!
  - `int x = 0b0011; // illegal`

- Hexadecimal has THE SAME bits as a binary number.
- One hex "digit" is 4 bits.
  Two hex "digits" is one byte.

Decimal	Binary	Hex
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

Bitwise operations

- Various operations can be performed on bits in C
  - &
    - Bitwise AND
      - `0x9 & 0x3 = 0x1`
      - `0b1001 | 0b0011 = 0b0001`
  - |
    - Bitwise OR
      - `0xA | 0x9 = 0xB`
      - `0b1010 | 0b1001 = 0b1011`
  - ^
    - Bitwise XOR
      - `0x3 ^ 0xD = 0xE`
      - `0b0011 ^ 0b1101 = 0b1110`

Bitwise operations

- Various operations can be performed on bits
  - ~
    - Bitwise NOT or “compliment”
      - `~0x5 = 0xA`
      - `~0b0101 = 0b1010`
  - <<
    - Logical Left shift
      - `0x2 << 2 = 0x8`
      - `0b0010 << 2 = 0b1000`
  - >>
    - Right shift (arithmetic if signed, logical if unsigned)
      - `0x4 >> 1 = 0x2`
      - `0b0100 >> 1 = 0b0010`

Bitwise Practice

- Given a 16 bit LC4 shift instruction, extract the sub-opcode and return it
  - SLL should return 0
  - SRA should return 1
  - SRL should return 2

SLL Rd Rs UIMM4	1010 ddds ss00 uuuu
SRA Rd Rs UIMM4	1010 ddds ss01 uuuu
SRL Rd Rs UIMM4	1010 ddds ss10 uuuu

```
unsigned short int shift_subop(unsigned short int insn) {



}
```

Bitwise Practice

- Given a 16 bit LC4 shift instruction, extract the sub-opcode and return it
  - SLL should return 0
  - SRA should return 1
  - SRL should return 2

SLL Rd Rs UIMM4	1010 ddds ss00 uuuu
SRA Rd Rs UIMM4	1010 ddds ss01 uuuu
SRL Rd Rs UIMM4	1010 ddds ss10 uuuu

THERE ARE OTHER POSSIBLE SOLUTIONS

```
unsigned short int shift_subop(unsigned short int insn) {
  unsigned short int mask = 0x30;
  unsigned short int sub_op = insn & mask;
  sub_op = sub_op >> 4;
  return sub_op;
}
```

```
unsigned short int shift_subop(unsigned short int insn) {
  return (insn & 0x30) >> 4;
}
```

Endianness

- In other architectures, there is one byte at each address location
  - For multi-byte data, how do we order it in memory?
  - Data should be kept together, but what order should it be?
  - Example, store the 4-byte (32-bit) int:
    0x A1 B2 C3 D4

Each byte has its own address

Most significant Byte

Least significant Byte

- The order of the bytes in memory is called endianness
  - Big endian vs little endian

Endianness

- Consider our example 0x A1 B2 C3 D4

  Most significant Byte Least significant Byte

- Big endian
  - Least significant byte has highest address
  - Looks the most like what we would read
  - The standard for storing information on files/the network

	0x2000	0x2001	0x2002	0x2003	
	A1	B2	C3	D4	

Least significant Byte

- Little Endian
  - Least significant byte has lowest address
  - What your VM probably uses

Note how the hex digits within a byte are still in the same order

	0x2000	0x2001	0x2002	0x2003	
	D4	C3	B2	A1	

Endianness practice

- Complete the convert() function, which converts from little endian to big endian for a 16 bit input

```
unsigned short int convert(unsigned short int input) {



}
```

Endianness practice

- Complete the convert() function, which converts from little endian to big endian for a 16 bit input

```
unsigned short int convert(unsigned short int input) {
   unsigned short int upper = (input & 0xFF00) >> 8;
   unsigned short int lower = input & 0x00FF;
   unsigned short int result = (lower << 8) | (upper);
   return result;
}
```

Endianness functions

- There are some functions out there that convert byte orderings
  - `htons()` -> Host to Network short (16 bits)
    - Converts from Host byte ordering to network byte ordering
  - `ntohs()` -> Network to Host short (16 bits)
    - Converts from network byte ordering to host byte ordering

- "Network byte order" is big endian. Your "host" machine is little endian

- More info in <arpa/inet.h>
  - Variants also exist for 32 bit and 64 bit conversion

Outline

- Bits
  - Binary & Hexadecimal
  - Endianness
  - Bit manipulation
- **File I/O**
- Hexdump demo

Thinking about files in C

- In C (and unix based operating systems), a file is just a sequence of bytes
  - It is up to programs and users to interpret those bytes for various applications

- Basic Operations:
  - Open
  - Close
  - Read
  - Write

- ALL FILES ARE SEQUNCES OF BYTES
  - For some of these files, the bytes translate to ASCII Characters

FILE*

- C stdio provides FILE* and various functions for reading/writing files
  - FILE* and the associated functions can be used as a “file iterator”
- Main operations:
  - `fopen()`
  - `fclose()`
  - `fread()`
  - `fwrite()`
  - `feof()`
- Three streams provided by default: `stdin`, `stdout`, `stderr`

C FILE Functions (1 of 3)

- Some FILE* functions (complete list in `stdio.h`):
  - `FILE* fopen(filename, mode);`
    - Returns **NULL** on error (CHECK THIS)
    - Opens the specified file in specified file access mode
      - Some format access modes:
        - » "r" -> read from file
        - » "w" -> write to file (remove old content if file already exists)
        - » "a" -> append to file (write to end of file if it already exists)
        - » "rb" -> read in binary mode
        - » "wb" -> write in binary mode
  - `int fclose(FILE* f);`
    - Closes the specified file.

C FILE Functions (2 of 3)

- Some FILE functions (complete list in `stdio.h`):

Returns the number of elements read/written

  - `size_t fwrite(ptr, size, count, file);`
    - Writes an "array" of *count* elements of *size* bytes from *ptr* to *file*
  - `size_t fread(ptr, size, count, file);`
    - Reads an "array" of *count* elements of *size* bytes from *file* to *ptr*
- Each read/writes (`size * count`) number of bytes
- Note: These functions read/write bits directly.
  - If we wrote an integer, the bits of the integer are written NOT the characters.
    E.g. if we had `short int x = 13`, we would write the bits `0000000000001101` and NOT the characters "13".

C FILE Functions (2 of 3)

- Some FILE functions (complete list in `stdio.h`):
  - `size_t fwrite(ptr, size, count, file);`
    - Writes an "array" of *count* elements of *size* bytes from *ptr* to *file*
  - `size_t fread(ptr, size, count, file);`
    - Reads an "array" of *count* elements of *size* bytes from *file* to *ptr*
- Each read/writes (`size * count`) number of bytes
- Example:

```
#define BUFSIZE 128
int main(int argc, char** argv) {
  FILE *f = // for this example assume f is opened
  int readbuf[BUFSIZE];
  size_t readlen;
  readlen = fread(readbuf, sizeof(int), BUFSIZE, f);
  // ...
```

C FILE Functions (2 of 3)

- Some FILE functions (complete list in `stdio.h`):
  - `size_t fwrite(ptr, size, count, file);`
    - Writes an “array” of *count* elements of *size* bytes from *ptr* to *file*
  - `size_t fread(ptr, size, count, file);`
    - Reads an “array” of *count* elements of *size* bytes from *file* to *ptr*
- Can be used to read in one item instead of many
- Example:

```
int main(int argc, char** argv) {
  FILE *f = // for this example assume f is opened
  int read_val;  // only reading one integer
  if (!fread(&read_val, sizeof(int), 1, f)) {
    // error handling
  }
  // ...
```

C FILE Functions (3 of 3)

- Some FILE* functions (complete list in `stdio.h`):
  - `int fprintf(stream, format, ...);`
    - Writes a formatted C string
      - `printf(...);` is equivalent to `fprintf(stdout, ...);`
  - `int fscanf(stream, format, ...);`
    - Reads data and stores data matching the format string

FILE & Endianness

- If we are writing bits that represent elements larger than a byte, we need to consider what is the endianness of the bytes we write.
  - The endianness should usually be big endian
  - Note that ascii characters are 1 byte each, so endianness doesn't apply to them

- We prefer writing the bits of an integer instead of it's string equivalent UNLESS a human is supposed to read the file.
  - If we had an integer 432134, it would take 6 bytes to write the string "432134" but only 4 bytes if it is a 32 bit integer.

File I/O Practice

- Finish the following program so that we write the array to a file called "output.bytes" with the data in big endian

```
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char** argv) {
  unsigned short int to_write[3] = {33219, 30902, 152};
```

File I/O Practice

- Finish the following program so that we write the array to a file called "output.bytes" with the data in big endian

```
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char** argv) {
  unsigned short int to_write[3] = {33219, 30902, 152};
  for (int i = 0; i < 3; i++) {
    to_write[i] = htons(to_write[i]);
  }
  FILE* f = fopen("output.bytes", "wb");
  if (f == NULL) {
    printf("Error: could not open file for writing\n");
    return EXIT_FAILURE;
  }
  fwrite(to_write, sizeof(unsigned short int), 3, f);
  fclose(f);
}
```

Outline

- Bits
  - Binary & Hexadecimal
  - Endianness
  - Bit manipulation
- File I/O
- **Hexdump demo**

Hexdump

- Tool for looking at the contents of a binary file.

- Example:

```
hexdump -C divide.obj
```

- Want to store the output in a file?

```
hexdump -C divide.obj > hex.txt
```

Hexdump Output

- Example from doing `hexdump -C divide.obj`

Offset (in hex) into the file

Result of trying to read the bytes as ASCII

```
00000000  ca de 00 00 00 06 98 00  19 21 14 93 07 fd 19 3f  |.........!.....?|
00000010  0f ff c3 b7 00 01 00 04  4c 4f 4f 50 c3 b7 00 06  |........LOOP....|
00000020  00 03 45 4e 44 c3 b7 00  05 00 0d 49 4e 46 49 4e  |..END......INFIN|
00000030  49 54 45 5f 4c 4f 4f 50                           |ITE_LOOP|
00000038
```

Contents of the file in hex, with spacing between each byte