Owl Tech Industries

Systems Programming Division

Assignment 2: Advanced Data Representation

Course:	CS 3503 - Comp Org & Arch
Assignment:	A2 - Data Representation & Mapping
Language:	C Programming
Topics:	Direct mapping, signed representations

Continuing Your Work at Owl Tech

Building on Assignment 1
In Assignment 1, you built a number base conversion utility using division and subtraction algorithms. Now you'll extend that work with more advanced techniques: direct mapping between number systems and signed number representations used in modern CPUs.
The good news: You can reuse your test framework, file parsing, and utility functions from Assignment 1. Focus your efforts on implementing the new conversion methods.

1 Assignment Overview

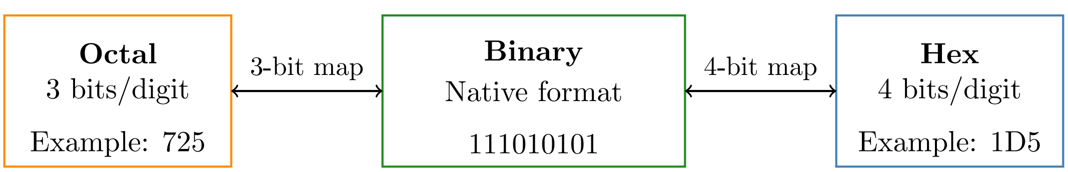
1.1 The Next Challenge

The CPU design team needs tools for analyzing how different architectures represent signed numbers. You'll implement:

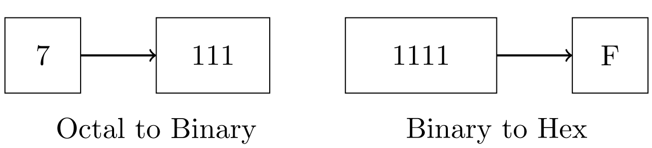
- **Direct mapping functions** - Fast conversions using bit patterns
- **Signed representations** - How computers actually store negative numbers

1.2 Visual Overview: Number System Mappings

Direct Mapping Between Number Systems



Mapping Example:



2 Function Specifications

2.1 Part 1: Direct Mapping Functions

These functions use the mathematical relationship between bases that are powers of 2.

2.1.1 Function 1: oct_to_bin

Purpose: Convert octal to binary using 3-bit mapping
Prototype: `void oct_to_bin(const char *oct, char *out)`
Key Insight: Each octal digit = exactly 3 binary digits

Octal to Binary Mapping Table

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

Example: "725" → "111" + "010" + "101" = "111010101"

2.1.2 Function 2: oct_to_hex

Purpose: Convert octal to hexadecimal via binary
Prototype: `void oct_to_hex(const char *oct, char *out)`
Algorithm: Octal → Binary → Group by 4 → Hex

2.1.3 Function 3: hex_to_bin

Purpose: Convert hexadecimal to binary using 4-bit mapping
Prototype: `void hex_to_bin(const char *hex, char *out)`
Note: Handle both uppercase and lowercase hex digits

2.2 Part 2: Signed Number Representations

Three Ways Computers Represent Negative Numbers

Sign-Magnitude	One's Complement	Two's Complement
1 bit for sign 31 bits for value	Flip all bits for negative	Flip bits + 1 (Standard today)

Example: Representing -5

Sign-Mag: `10000000000000000000000000000101`
↑ sign bit
One's Comp: `11111111111111111111111111111010`

Two's Comp: `11111111111111111111111111111011`

2.2.1 Function 4: to_sign_magnitude

```
// For positive: output as-is with leading zeros
// For negative: set bit 31 to 1, keep magnitude in bits 0-30
if (n >= 0) {
    // Regular binary with 32 bits
} else {
    // Set sign bit + magnitude of absolute value
}
```

Listing 1: Sign-Magnitude Algorithm

2.2.2 Function 5: to_ones_complement

```
// For positive: output as-is with leading zeros
// For negative: flip ALL bits
if (n >= 0) {
    // Regular binary with 32 bits
} else {
    // Get positive representation, then flip every bit
}
```

Listing 2: One's Complement Algorithm

2.2.3 Function 6: to_twos_complement

```
// For positive: output as-is
// For negative: flip all bits and add 1
// This is how modern CPUs actually store integers!
```

Listing 3: Two's Complement Algorithm

3 Implementation Tips

3.1 Reusing Code from Assignment 1

Work Smarter!

You've already built:

- File parsing routines
- Test framework
- Output formatting
- Build system

Copy these from Assignment 1 and focus on the new functions!

3.2 Buffer Management

Buffer Size Guidelines

Variable Length	32-bit Fixed
Allocate generously: `char buffer[100];`	Exactly 33 chars: `char buffer[33];`

Remember: +1 for null terminator!

3.3 Quick Reference: Bit Patterns

```c
// Extract 3 bits for octal
int octal_digit = oct_char - '0';  // '7' -> 7

// Build binary from octal
// 7 -> "111"
buffer[pos++] = (octal_digit >> 2) & 1 ? '1' : '0';
buffer[pos++] = (octal_digit >> 1) & 1 ? '1' : '0';
buffer[pos++] = (octal_digit >> 0) & 1 ? '1' : '0';

// Check if hex char is valid
if ((c >= '0' && c <= '9') ||
    (c >= 'A' && c <= 'F') ||
    (c >= 'a' && c <= 'f')) {
    // Valid hex digit
}
```

Listing 4: Common Patterns You'll Need

4 Testing Your Implementation

4.1 Test Strategy

Your main.c from Assignment 1 should work with minimal changes:

```c
// Your existing test structure probably looks like:
typedef struct {
    char function_name[50];
    char input[100];
    char expected[100];
} TestCase;

// Just update to handle new function names!
if (strcmp(test.function_name, "oct_to_bin") == 0) {
    oct_to_bin(test.input, actual_output);
} else if (strcmp(test.function_name, "to_sign_magnitude") == 0) {
    int32_t n = atoi(test.input);
    to_sign_magnitude(n, actual_output);
}
// ... etc
```

Listing 5: Reusing Your Test Framework

4.2 Expected Output Format

```
Test 1: oct_to_bin("725") -> Expected: "111010101", Got: "111010101" [PASS]
Test 2: to_twos_complement(-5) -> Expected: "11111111111111111111111111111011", Got: "11111111111111111111111111111011" [PASS]
...
Summary: 60/60 tests passed
```

Listing 6: Test Output

5 Submission Requirements

5.1 What to Submit

GitHub Repository

- convert.c (updated)
- main.c (from A1)
- README.md
- A2_tests.txt

D2L Submission

- convert.c
- main.c
- output.txt
- GitHub URL

Same process as Assignment 1!

5.2 README Template

README Flexibility

Feel free to use your own format! Just include build instructions and results.

```
# CS 3503 Assignment 2 - Data Representation and Mapping

## Author
[Your Name]

## Description
Advanced data representation functions for Owl Tech's CPU design team.

## What's New
- Direct mapping functions (oct/hex/bin)
- Signed number representations
- Reused test framework from A1

## Build Instructions
```bash
gcc -o convert convert.c main.c
./convert
```

Test Results
[Your test summary here]
```

Listing 7: Example README.md

6 Resources and Support

6.1 Helpful Links

- Two's Complement Visualization: `https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html`
- Number Systems: `https://www.tutorialspoint.com/computer_logical_organization/number_system_conversion.htm`
- Your Assignment 1 code (best reference!)

6.2 Office Hours Support

Getting Help

- Teams Channel for quick questions
- Office hours for debugging help
- Check for GTA and Main Lab TA assistance.

A Appendix: Understanding 32-Bit Representations

A.1 Why Exactly 32 Bits?

32-Bit Integer in Memory

Byte 3	Byte 2	Byte 1	Byte 0
31-24	23-16	15-8	7-0

Total: 32 bits = 4 bytes = 1 word (on 32-bit systems)

Signed range: -2,147,483,648 to 2,147,483,647
Unsigned range: 0 to 4,294,967,295

Modern computers work with fixed-size chunks:

- **8-bit**: Historic computers, embedded systems
- **16-bit**: Early PCs, some microcontrollers
- **32-bit**: Standard for decades (your functions)
- **64-bit**: Modern CPUs and operating systems

A.2 Quick Reference: String Building

```c
void to_32bit_binary(uint32_t value, char *out) {
    for (int i = 31; i >= 0; i--) {
        out[31 - i] = ((value >> i) & 1) ? '1' : '0';
    }
    out[32] = '\0';  // Don't forget!
}

// For negative numbers in two's complement:
// The bit pattern IS the two's complement
// Just cast and print:
int32_t negative = -5;
uint32_t bit_pattern = (uint32_t)negative;
to_32bit_binary(bit_pattern, output);
```

Listing 8: Building 32-bit Binary Strings

You've Got This!

Assignment 2 builds on your success with Assignment 1. You already know:

- How to parse test files
- How to build and test C programs
- How to work with strings and buffers
- How to use Git for version control

Now you're adding specialized knowledge about how CPUs represent data. These concepts appear everywhere in systems programming - from network protocols to graphics programming.

Good luck with your second Owl Tech project!