Owl Tech Industries

Software Engineering Internship Program

Project Assignment: BitBoard Game Engine

Project Type: Individual Assignment
Department: Low-Level Systems Division
Language: C (Required)
Submission: See course schedule

Welcome to Owl Tech!

Welcome to the Low-Level Systems Division at Owl Tech Industries! You'll be developing a bitboard-based checkers game that demonstrates efficient bit manipulation techniques. This project builds on your C programming skills while introducing advanced concepts used in high-performance game engines.

1 Project Overview

1.1 What You'll Build

You'll create a checkers game that uses bitboards - a technique where each bit in an integer represents a board square. This approach is used in chess engines and other performance-critical applications.

1.2 Learning Objectives

- Master bitwise operations (AND, OR, XOR, NOT, shifts)
- Implement efficient data structures using bit manipulation
- Apply Boolean algebra to practical programming
- Build a working game in C

1.3 Grading Overview

Your project will be evaluated in four equal parts:

Component	Weight	Focus
Phase 1: Bit Operations Completeness	25%	All required functions implemented
Phase 1: Bit Operations Correctness	25%	Functions work properly
Phase 2: Game Completeness	25%	Core game features present
Phase 2: Game Correctness	25%	Game works using bitboards

2 Technical Background

2.1 Understanding Bitboards

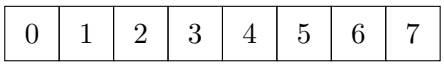
A bitboard uses the bits of an integer to represent a game board. For an 8×8 checkers board, we use a 64-bit unsigned integer where each bit represents one square.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Bit positions on an 8×8 board

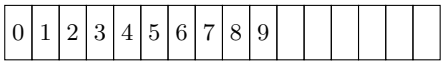
2.2 Data Types You'll Use

`unsigned char:`



(8 bits)

`unsigned int:`



... 31 (32 bits)

`unsigned long long:`

64 bits (perfect for 8×8 board!)

Phase 1: Bit Manipulation Operations

Phase 1 Expectations

Completeness (25% of grade): Implement ALL listed bit manipulation functions
Correctness (25% of grade): Functions must work correctly with proper edge case handling

2.3 Core Bit Manipulation Capabilities

Your bit manipulation library needs these capabilities - but feel free to implement them your way!

```
// Basic bit operations - implement these or equivalent functions
unsigned int SetBit(unsigned int value, int position);    // Set bit to 1
unsigned int ClearBit(unsigned int value, int position);  // Set bit to 0
unsigned int ToggleBit(unsigned int value, int position); // Flip bit
int GetBit(unsigned int value, int position);             // Get bit value

// Counting and finding
int CountBits(unsigned int value);                        // Count 1s

// Shift operations
unsigned int ShiftLeft(unsigned int value, int positions);
unsigned int ShiftRight(unsigned int value, int positions);

// Display functions
void PrintBinary(unsigned int value);                     // Show in binary
void PrintHex(unsigned int value);                        // Show in hex
```

Listing 1: Suggested Bit Manipulation Functions

Be Creative!

These function names and signatures are **suggestions**. If you have a better approach, use it! For example:

- Want to combine set/clear/toggle into one function with a parameter? Go for it!
- Prefer different names like `EnableBit` or `FlipBit`? That's fine!
- Want to add helper functions like `CreateMask` or `IsBitSet`? Excellent!

The key is that you can manipulate bits effectively, not that you follow a specific API.

2.4 What "Completeness" Means for Phase 1

To earn full completeness points, you need to demonstrate:

- Ability to set bits to 1 (however you implement it)
- Ability to clear bits to 0
- Ability to check bit values
- Ability to shift bits left and right

- Ability to display bit patterns for debugging
- Any additional helper functions that support these operations

2.5 What "Correctness" Means for Phase 1

Your implementation should:

- Actually manipulate bits using bitwise operators (&, —, ^, ~, `<<`, `>>`)
- Handle edge cases reasonably (like invalid bit positions)
- Produce correct results for typical use cases
- Work with your Phase 2 game implementation

2.6 Example Implementations and Helper Functions

```
// Traditional approach
unsigned int SetBit(unsigned int value, int position) {
    if (position < 0 || position >= 32) return value;
    return value | (1 << position);
}

// Alternative: Helper function approach
unsigned int CreateMask(int position) {
    return 1 << position;
}

unsigned int SetBitWithMask(unsigned int value, int position) {
    return value | CreateMask(position);
}

// Alternative: Combined function approach
unsigned int ModifyBit(unsigned int value, int position, int operation) {
    // operation: 0=clear, 1=set, 2=toggle
    switch(operation) {
        case 0: return value & ~(1 << position);
        case 1: return value | (1 << position);
        case 2: return value ^ (1 << position);
    }
    return value;
}
```

Listing 2: Example Implementation Approaches

Experiment and Explore!

Don't feel constrained by traditional approaches. Some creative ideas:

- Build a bit manipulation library that fits YOUR game's needs
- Create specialized functions like `SetMultipleBits` or `ClearRange`
- Develop helper functions that make your game logic cleaner
- Try different error handling strategies

2.7 Testing Your Phase 1 Functions

Create simple tests to verify your functions work:

```c
int main() {
    unsigned int test = 0;

    // Test SetBit
    test = SetBit(test, 3);  // Should set bit 3
    printf("After setting bit 3: ");
    PrintBinary(test);  // Should show: 0000 0000 0000 1000

    // Test CountBits
    printf("Number of 1s: %d\n", CountBits(test));  // Should print 1

    return 0;
}
```

Listing 3: Testing Example

Phase 2: Checkers Game Implementation

Phase 2 Expectations

Completeness (25% of grade): Implement core game features
Correctness (25% of grade): Game must work using bitboard operations
Note: You have flexibility in HOW you implement these features!

2.8 Required Game Features

Your checkers game must include:

1. **Board Representation**: Use bitboards to track pieces
2. **Display**: Show the current board state clearly
3. **Move Pieces**: Allow players to move their pieces
4. **Capture Pieces**: Remove opponent pieces when jumped
5. **King Promotion**: Regular pieces become kings at the far edge
6. **Win Detection**: Recognize when a player has won

2.9 What "Completeness" Means for Phase 2

To earn full completeness points:

- All 6 features above must be attempted
- Game must be playable (even if not perfect)
- Players can input moves somehow (your choice how)
- Board displays in a understandable format

2.10 What "Correctness" Means for Phase 2

To earn full correctness points:

- Pieces move according to checkers rules
- Captures actually remove opponent pieces
- Kings can move backwards
- Game uses bitwise operations (not just arrays)
- Win/loss conditions work properly

2.11 Implementation Flexibility

You have freedom in HOW you implement these features:

Your Design Choices

- **Input Method**: Number coordinates? Letter-number? Click positions? Your choice!
- **Display Style**: ASCII art? Simple grid? Unicode symbols? Up to you!
- **Game Flow**: Two-player turns? Single player vs simple AI? Either works!
- **Code Structure**: One big file? Separate functions? Helper utilities? Your design!
- **Extra Features**: Add an AI, undo moves, save games - these are bonuses!

2.12 Example Structures and Approaches

```
// Approach 1: Simple and Direct
typedef struct {
    unsigned long long red_pieces;
    unsigned long long black_pieces;
    // Maybe that's all you need!
} SimpleGame;

// Approach 2: More Detailed
typedef struct {
    unsigned long long player1_pieces;
    unsigned long long player1_kings;
    unsigned long long player2_pieces;
    unsigned long long player2_kings;
    int current_turn;
} DetailedGame;

// Approach 3: Your Creative Solution
// Maybe you track moves differently, or add history, or...?
```

Listing 4: Different Valid Approaches - Choose or Create Your Own

Helper Functions are Encouraged!

Consider creating helper functions that make your code cleaner:

- `IsValidMove()` - Check if a move follows rules
- `GetPossibleMoves()` - Find all legal moves for a piece
- `BoardToIndex()` - Convert board coordinates to bit positions
- `PrintBoardPretty()` - Make a nice display
- Whatever helps you build a working game!

2.13 Using Bitboards Creatively

The key requirement is using bit operations for your game logic. Here are some ideas:

```
// Example 1: Check if a square is occupied
int IsOccupied(GameState* game, int position) {
    unsigned long long all = game->player1_pieces | game->player2_pieces;
    return (all >> position) & 1;
}

// Example 2: Find all empty squares
unsigned long long GetEmptySquares(GameState* game) {
    unsigned long long all = game->player1_pieces | game->player2_pieces;
    return ~all;  // Invert to get empty squares!
}

// Example 3: Creative move generation
unsigned long long GenerateMoves(unsigned long long pieces, int direction) {
    // Shift pieces to simulate movement
    if (direction == NORTHEAST)
        return (pieces << 9) & VALID_SQUARES;
    // Your creative solution here!
}
```

Listing 5: Different Ways to Use Bitboards

The Fun Part: Experimentation!

This project is about learning bit manipulation through experimentation. Try different approaches:

- Maybe you discover a clever way to detect captures using XOR
- Perhaps you find an elegant king promotion check using masks
- You might create an efficient win detection using bit counting

There's no single "correct" implementation - explore and learn!

Appendix: Quick Reference Guide

2.14 A. Bitwise Operations Visual Guide

C Bitwise Operators

Operation	C Operator	What it Does	Example
AND	`&`	Both bits must be 1	`5 & 3 = 1`
OR	`|`	Either bit can be 1	`5 | 3 = 7`
XOR	`^`	Bits must be different	`5 ^ 3 = 6`
NOT	`~`	Flips all bits	`~5 = -6`
Left Shift	`<<`	Moves bits left	`5 << 1 = 10`
Right Shift	`>>`	Moves bits right	`5 >> 1 = 2`

Visual Examples:

AND:
```
  0101 (5)
& 0011 (3)
  ------
  0001 (1)
```

OR:
```
  0101 (5)
| 0011 (3)
  ------
  0111 (7)
```

Left Shift:
```
  0101 (5)
<< 1
  ------
 1010 (10)
```

2.15 B. Common Patterns You'll Need

```c
// Set bit n to 1
value = value | (1 << n);

// Clear bit n to 0
value = value & ~(1 << n);

// Toggle bit n
value = value ^ (1 << n);

// Check if bit n is set
if (value & (1 << n)) {
    // Bit is 1
}

// Clear all bits
value = 0;

// Set all bits
value = ~0;
```

Listing 6: Useful Bit Manipulation Patterns

2.16 C. Debugging Tips

1. **Print bits often**: Use your PrintBinary function liberally
2. **Test simple cases first**: Start with one piece, then add complexity
3. **Watch for overflow**: Shifting by 32+ is undefined for 32-bit integers
4. **Remember bit numbering**: Bit 0 is rightmost, bit 31/63 is leftmost

2.17 D. Checkers-Specific Hints

- Regular pieces can only move forward (different for each player)
- Kings can move both forward and backward
- Captures are mandatory in standard checkers (optional for this project)
- Only dark squares are used (32 of the 64 squares)

2.18 E. Common Mistakes to Avoid

1. **Wrong operator precedence**: Use parentheses! `1 << n + 1` != `(1 << n) + 1`
2. **Signed vs unsigned**: Use unsigned types for bitboards
3. **Not checking bounds**: Validate bit positions before shifting
4. **Forgetting endianness**: Be consistent with your bit numbering