Bit Array Documentation

Yi Lu

March 27, 2024

Bit Array Documentation

Introduction

A bit array is a data structure that encapsulates an array for bitwise operations on individual bits. It provides a compact representation for manipulating sequences of bits efficiently. This documentation provides an overview of the bit array data structure implemented in C. The content of this document has been polished by GPT.

Properties

- **base_type**: The data type used as the base structure for each element in the array.
  - `unsigned char` ;
  - `unsigned int` ✓.
- **baseSize**: The number of bits in each base type.
  - `unsigned char`: **baseSize** = 8;
  - `unsigned int`: **baseSize** = 32 (in 64-bit system) ✓.
- **cell**: A unit for operations, consisting of a specific number of bits.
  - **cellNum**: The number of cells in the array.
  - **cellSize**: The number of bits in each cell.
- **bitLength**: The total number of bits in the array.
- **arrayLength**: The total number of elements in the array.

Relationships between properties

$$cellNum \times cellSize = bitLength \leq arrayLength$$

Ordering of bits:

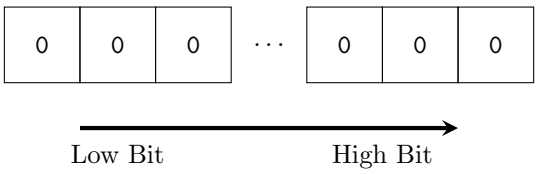

Figure 1: Illustration of Bit Array Bits

Private Functions

getIndex

This private function calculates the index of the integer containing the bit specified by the given position.

```
/**
 * Calculates the index of the integer containing the bit specified by the given
    position.
 *
 * @param pos       The position of the bit.
 * @param bitSize   The size of the base data type in bits.
 * @return          The index of the integer containing the bit.
 */
static int getIUndex(int pos, int bitSize)
{
    return pos / bitSize;
}
```

Listing 1: get_index Function

getOffset

This private function calculates the offset within the integer containing the bit specified by the given position.

```
/**
 * Calculates the offset within the integer containing the bit specified by the given
    position.
 *
 * @param pos       The position of the bit.
 * @param bitSize   The size of the base data type in bits.
 * @return          The offset within the integer containing the bit.
 */
static int getOffset(int pos, int bitSize)
{
    return pos % bitSize;
}
```

Listing 2: get_offset Function

Methods

bitAnd

This method performs bitwise AND operations between two unsigned integer arrays.

```
/**
 * Performs bitwise AND operations between the unsigned integer arrays 'arr1' and '
    arr2' and stores the result in 'result'.
 *
 * @param arr1      The first unsigned integer array.
 * @param arr2      The second unsigned integer array.
 * @param result    The unsigned integer array to store the result.
 * @param len       The length of arrays 'arr1', 'arr2', and 'result'.
 */
void bitAnd(unsigned int* arr1, unsigned int* arr2, unsigned int* result, int len)
{
    for (int i = 0; i < len; i++) {
        result[i] = arr1[i] & arr2[i];
    }
}
```

Listing 3: Bitwise AND Operation

bitOr

This method performs bitwise OR operations between two unsigned integer arrays.

```
/**
 * Performs bitwise OR operations between the unsigned integer arrays 'arr1' and '
    arr2' and stores the result in 'result'.
```

```
 *
 * @param arr1      The first unsigned integer array.
 * @param arr2      The second unsigned integer array.
 * @param result    The unsigned integer array to store the result.
 * @param len       The length of arrays 'arr1', 'arr2', and 'result'.
 */
void bitOr(unsigned int* arr1, unsigned int* arr2, unsigned int* result, int len)
{
    for (int i = 0; i < len; i++) {
        result[i] = arr1[i] | arr2[i];
    }
}
```

Listing 4: Bitwise OR Operation

bitEq

This method checks if two unsigned integer arrays are equal.

```
/**
 * Checks if the unsigned integer arrays 'arr1' and 'arr2' are equal.
 *
 * @param arr1      The first unsigned integer array.
 * @param arr2      The second unsigned integer array.
 * @param len       The length of arrays 'arr1' and 'arr2'.
 * @return          Returns true if arrays 'arr1' and 'arr2' are equal, otherwise
    returns false.
 */
bool bitEq(unsigned int* arr1, unsigned int* arr2, int len)
{
    for (int i = 0; i < len; i++)
    {
        if (arr1[i] != arr2[i]) {
            return false;
        }
    }
    return true;
}
```

Listing 5: Bitwise Equality Check

clone

This method clones an unsigned integer array.

```
/**
 * Clones the unsigned integer array 'arr' and stores the result in 'result'.
 *
 * @param arr       The unsigned integer array to be cloned.
 * @param result    The unsigned integer array to store the cloned array.
 * @param len       The length of arrays 'arr' and 'result'.
 */
void clone(unsigned int* arr, unsigned int* result, int len)
{
    for (int i = 0; i < len; i++) {
        result[i] = arr[i];
    }
}
```

Listing 6: Clone Method

is_zero

This method checks if an unsigned integer array contains only zero values.

```
/**
 * Checks if the unsigned integer array 'arr' contains only zero values.
 *
 * @param arr   The unsigned integer array to be checked.
 * @param len   The length of the array 'arr'.
 * @return      Returns true if all elements in the array are zero, otherwise returns
     false.
 */
bool is_zero(unsigned int* arr, int len) {
    for (int i = 0; i < len; i++) {
        if (arr[i] != 0) {
            return false;
        }
    }
    return true;
}
```

Listing 7: is_zero Method

getBitItem

This method retrieves a single bit from the unsigned integer array at the specified index.

```
/**
 * Retrieves a single bit from the unsigned integer array 'arr' at the specified bit
    index and stores the result in 'result'.
 *
 * @param arr         The unsigned integer array from which to retrieve the bit.
 * @param result      The unsigned integer array to store the result.
 * @param bit_index   The index of the bit to retrieve.
 */
void getBitItem(unsigned int* arr, unsigned int* result, int bit_index) {
    const int index = getIndex(shiftAmount, 32);
    const int offset = getOffset(shiftAmount, 32);
    *result = (arr[index] >> offset) & 1u;
}
```

Listing 8: getBitItem Method

getBitItemSlice

This method retrieves a slice of bits from the unsigned integer array within the specified range and step.

```
/**
 * Retrieves a slice of bits from the unsigned integer array 'arr' within the
    specified range and step and stores the result in 'result'.
 *
 * @param arr       The unsigned integer array from which to retrieve the bits.
 * @param result    The unsigned integer array to store the result.
 * @param start     The starting index of the slice.
 * @param end       The ending index of the slice (exclusive).
 * @param step      The step size for indexing the array.
 */
void getBitItemSlice(unsigned int* arr, unsigned int* result, int start, int end, int
    step) {
    for (int i = 0;; i++) {
        if (start + step * i >= end) {
            break;
        }
        getBitItem(arr, result + i, start + step * i);
    }
}
```

Listing 9: getBitItemSlice Method

shiftLarray

This method performs bitwise shift operations towards the low bit.

```
/**
 * Shifts the unsigned integer array 'arr' towards the low bit by 'shiftAmount' bits and stores the result in 'result'.
 *
 * @param arr           The unsigned integer array to be shifted.
 * @param result        The unsigned integer array to store the result.
 * @param len           The length of arrays 'arr' and 'result'.
 * @param shiftAmount   The number of bits to shift towards the low bit.
 */
void shiftLarray(unsigned int* arr, unsigned int* result, int len, int shiftAmount)
{
    // Calculate the number of integers to shift and the remaining shift amount
    const int index = getIndex(shiftAmount, 32);
    const int offset = getOffset(shiftAmount, 32);
    int n = len - index - 1;

    // Initialize the result array with zeros
    set0array(result, len);

    // Perform shifting
    if (offset != 0) {
        for (int i = 0; i < n; i++) {
            result[i] = (arr[index + i] >> offset);
            result[i] |= (arr[index + i + 1] << (sizeof(int) * 8 - offset));
        }
        result[n] = (arr[index + n] >> offset);
    } else {
        for (int i = 0; i < n; i++) {
            result[i] = arr[index + i];
        }
        result[n] = arr[index + n];
    }
}
```

Listing 10: Bitwise Shift to Low Bit

shiftHarray

This method performs bitwise shift operations towards the high bit.

```
/**
 * Shifts the unsigned integer array 'arr' towards the high bit by 'shiftAmount' bits and stores the result in 'result'.
 *
 * @param arr           The unsigned integer array to be shifted.
 * @param result        The unsigned integer array to store the result.
 * @param len           The length of arrays 'arr' and 'result'.
 * @param shiftAmount   The number of bits to shift towards the high bit.
 * @param bitLength     The total number of bits in the array.
 */
void shiftHarray(unsigned int* arr, unsigned int* result, int len, int shiftAmount, int bitLength)
{
    const int index = getIndex(shiftAmount, 32);
    const int offset = getOffset(shiftAmount, 32)
    const int n = len - index - 1;
    const int bound_offset = (8 * sizeof(int) * len - bitLength);

    // Initialize the result array with zeros
    set0array(result, len);

    // Perform shifting
    if (offset != 0) {
```

```
        for (int i = 0; i < n; i++) {
            result[len - 1 - i] = (arr[len - 1 - (index + i)] << offset);
            result[len - 1 - i] |= (arr[len - 1 - (index + i + 1)] >> (sizeof(int) *
                8 - offset));
        }
        result[len - 1 - n] = (arr[len - 1 - (index + n)] << offset);
    } else {
        for (int i = 0; i < n; i++) {
            result[len - 1 - i] = arr[len - 1 - (index + i)];
        }
        result[len - 1 - n] = arr[len - 1 - (index + n)];
    }

    // Handle boundary case
    if (bound_offset != 0) {
        result[len - 1] = (result[len - 1] << bound_offset) >> bound_offset ;
    }
}
```

Listing 11: Bitwise Shift to High Bit

Conclusion

In conclusion, a bit array provides a convenient interface for manipulating individual bits efficiently. By encapsulating bitwise operations and providing methods for accessing and modifying bits and cells, it serves as a versatile tool for various applications in computer science and programming.