Bit manipulation used in monitoring and control

Bit manipulation is commonly used in monitoring and control systems to efficiently represent and manipulate binary data. It offers efficient and effective ways to handle data, control devices, and manage system states.

* Status Monitoring: In a monitoring system, bit manipulation techniques can be used to check the status of various sensors or devices. For example, each sensor could have its own dedicated bit within a status register. Each bit in a status register can represent a different condition or flag, such as over-temperature, under-voltage, or communication errors. By reading this register and performing bitwise AND operations with specific masks, it's possible to determine whether a particular sensor has triggered an event or not.
* Device Control: In embedded systems, bit manipulation is often used to control hardware devices. For example, setting, clearing, or toggling bits in a control register can enable or disable specific features of a device, such as turning on an LED or activating a motor.
* Data Compression: Monitoring and control systems often need to transmit data efficiently over networks. Bit manipulation can be used to compress data by packing multiple data points or flags into a single byte or word, reducing the amount of data that needs to be transmitted, which is crucial for bandwidth-limited or power-constrained systems.
* Configuration Settings: Bit manipulation is used to set or read configuration settings where different bits represent different configuration options. This allows multiple settings to be stored compactly in a single register or memory location, making it easy to read or modify multiple settings simultaneously.
* Access Control and Security: In some systems, bit manipulation techniques are used to manage access control and security settings. For example, individual bits might be used to enable or disable access to certain features or to control read/write permissions on memory sections.
* Efficient Data Handling: For systems that process large amounts of data, bit manipulation can provide a way to efficiently handle, analyze, and transform data. Operations like bit shifting can be used for mathematical calculations, such as multiplication or division by powers of two, more efficiently than traditional arithmetic operations.
* Interrupt Management: In real-time systems, bit manipulation is often used in the management of interrupts. Specific bits in an interrupt register can be set or cleared to enable or disable interrupts, determine interrupt priorities, and acknowledge interrupt handling, allowing for precise control over system responses to events.
* Timing and Synchronization: Bit fields can be used to represent timing and synchronization information, such as timestamps, counters, or schedules. Manipulating these bits allows for precise control over timing-related functions in a system.

Consider a control system that uses an 8-bit register to control a set of eight lights. If the bit that controls a light is 1, the light is on. If it is 0, the light is off.

In the table below, the contents of the register show that lights 1, 4, 5, and 8 are on. Lights 2, 3, 6, and 7 are off.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Light numbers** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Register** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |

Two of the lights (lights 3 and 4) are security lights outside the building. To isolate the bits that control these two external lights, the mask 00110000 is used.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Light numbers** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Mask** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

**Use of the operation to check selected bits**

To **check** whether any of the external lights are on, you can use a logical bitwise **AND** operation, in conjunction with the mask, on the contents of the 8-bit register (that reflects the status of all of the lights). If the result is **not** zero (00000000), then at least one of the lights is on.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Register** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| **Mask** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **AND** | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

#### Use of the operation to clear selected bits

To turn off the external lights, you can use a logical bitwise **AND** operation in conjunction with the same mask to **clear** the relevant bits. However, this time, the mask must have 0 in the positions to clear, and 1 in all of the other positions, so you need to start by performing a logical bitwise **NOT** operation on the mask to flip the bits.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Mask** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **NOT** | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |

Then, you can use the inverted mask and a logical bitwise **AND** operation to clear the selected bits

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Register** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| **Inverted mask** | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| **AND** | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |

Notice that the operation has only reset the bit in position 4 (that controls light 4). Light 3 was already off, so the bit in position 3 did not need to change. All of the other bits have maintained their original values

Logical OR OPERATIONS

The **logical bitwise OR operation** takes two binary strings of equal length and performs the logical **OR** operation on each pair of corresponding bits. The result in each position is 1,1 if either or both bits are 1,1; if both bits are 0,0, the result is 0,0.

You have already seen how a logical bitwise **AND** operator can be used with a mask to **check** or **clear** one or more bits in a binary string. The logical bitwise **OR** operator can be used with a mask to **set** one or more bits.

Consider again the 8-bit register that holds the status of a set of lights:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Light numbers** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Register** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |

#### Use of the operation to set selected bits

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Register** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| **Mask** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **OR** | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |

To **set** (turn on) the external lights, you can use the same mask as before and a logical bitwise **OR** operation.

Notice that the bit in position 3 (that controls light 3) has been set to 1,1 (it was 0,0). All of the other bits are unchanged. Light 4 was already on, so the bit in position 4 did not need to change.

**Logical bitwise XOR operation**

The **logical bitwise XOR operation** takes two binary strings of equal length and performs the logical **XOR** operation on each pair of corresponding bits. The result in each position is 1 if either of the bits is 1; if both bits are 0 or both bits are 1, the result is 0.

You have now seen how a logical bitwise **AND** operator can be used with a mask to **check** or **clear** one or more bits in a binary string, and how a logical bitwise **OR** operator can be used with a mask to **set** one or more bits. The logical bitwise **XOR** operator can be used with a mask to **toggle** a bit.   
This means that if the bit is 1 it will be set to 0, and if the bit is 0, it will be set to 1.

Consider again the 8-bit register that holds the status of a set of lights:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Light numbers** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Register** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |

**Use of the operation to toggle selected bits**

To **toggle** the setting of the external lights, you can use the same mask as before and a logical bitwise **XOR** operation.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Register** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| **Mask** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **XOR** | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

Notice that the bits in positions 3 and 4 have been toggled (flipped). All of the other bits are unchanged.

**Use of the operation to clear all of the bits**

You can use the logical bitwise **XOR** operator as an efficient way to set the value of a register to zero. If you perform an XOR operation on a pair of corresponding bits with the same value, the result will be 0.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Register** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| **Mask (copy of the register values)** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| **XOR** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

This operation will generally require fewer clock cycles and less memory than loading a zero value and saving it to the register.

Logical Bitwise NOT

The **logical bitwise NOT operation** has the effect of flipping each bit in a binary string. Bits that are 0 become 1, and bits that are 1 become 0.

For example:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Binary string** | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| **NOT** | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

#### Shifting signed numbers

Logical shifts **cannot** be used to multiply or divide **signed** numbers, because the sign bit must not simply be shifted. If a number is signed, an arithmetic shift must be used.

An **arithmetic shift** must be used when the binary string represents a signed number. It is important to use arithmetic shifts when numbers are stored with a sign as the most significant bit (for example, with two's complement representation), so that shifting does not inadvertently cause a change of sign. With this type of shift, the sign of the number is preserved.

Consider an 8-bit register that holds the number minus, −52 stored in two's complement representation:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Place values** | -128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **Register** | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |

−128+64+8+4=−52

A simple right shift would result in the following bit pattern:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Place values** | -128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **Register** | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

This would evaluate to +102:

64+32+4+2=+102

**This is not the correct result**; a right shift should result in the number being divided by 2.

With an arithmetic shift, **the most significant bit (the sign bit) is not shifted**. The remaining bits are shifted and the empty places are padded as follows:

|  |  |
| --- | --- |
| Left shift of a positive number | Empty places are padded with 0 |
| Left shift of a negative number | Empty places are padded with 0 |
| Right shift of a positive number | Empty places are padded with 0 |
| Right shift of a negative number | Empty places are padded with 1 |

Pay particular attention to the fact that you must pad the empty places with 1 if you perform an arithmetic right shift of a negative number.

Consider again the number−52, and an arithmetic right shift of one place:

* The first bit (shaded pink) is the sign bit, so it **does not change**
* The other bits are shifted to the right (and the rightmost bit is discarded)
* The empty space in the second position (shaded yellow) is filled with a 1, because the sign bit is 1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Place values** | -128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **Register** | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

The result of the shift is −26:

−128+64+32+4+2=−26

If you had **incorrectly** filled the empty space in the second position with a 0, then the result would be:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Place values** | -128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **Register** | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

−128+32+4+2=−90

Cyclic Shift

In a **cyclic shift**, no bits are lost. The bits that are shifted out at one end are moved into the other end of the register.

The following example shows a cyclic right shift:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Original value** | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| **New value** | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

All of the bits have been shifted right. The least significant bit has not been dropped, but has instead been moved into the free space at the start of the string.

Cyclic shifts have some fairly niche uses. For example:

* In cryptography, the cyclic shift of a code will always yield another code. The Two fish cipher makes extensive use of cyclic shifts, because they are very fast operations.
* A cyclic left shift can be used for some operations in modular arithmetic.

