Bit field

A **bit field** is a data structure used in computer programming It consists of a number of adjacent computer memory locations which have been allocated to hold a sequence of bits, stored so that any single bit or group of bits within the set can be addressed.[1][2] A bit field is most commonly used to represent integral types of known, fixed bit-width.

The meaning of the individual bits within the field is determined by the programmer; for example, the first bit in a bit field (located at the field's base address) is sometimes used to determine the state of a particular attribute associated with the bit field.[3]

Within microprocessors and other logic devices, collections of bit fields called "flags" are commonly used to control or to indicate the intermediate state or outcome of particular operations.[4] Microprocessors typically have a status register that is composed of such flags, used to indicate various post-operation conditions, for example an arithmetic overflow. The flags can be read and used to decide subsequent operations, such as in processing conditional jump instructions. For example, a *je* (Jump if Equal) instruction in the x86 assembly language will result in a jump if the Z (zero) flag was set by some previous operation.

A bit field is distinguished from a bit array in that the latter is used to store a large set of bits indexed by integers and is often wider than any integral type supported by the language.[5] Bit fields, on the other hand, typically fit within a machine word,[3] and the denotation of bits is independent of their numerical index.[2]

Contents

Implementation

Bit fields can be used to reduce memory consumption when a program requires a number of integer variables which always will have low values. For example, in many systems storing an integer value requires two bytes (16-bits) of memory; sometimes the values to be stored actually need only one or two bits. Having a number of these tiny variables share a bit field allows efficient packaging of data in the memory.[6]

In C and C++, native implementation-defined bit fields can be created using unsigned int, signed int, or (in C99:) _Bool. In this case, the programmer can declare a structure for a bit field which labels and determines the width of several subfields.[7] Adjacently declared bit fields of the same type can then be packed by the compiler into a reduced number of words, compared with the memory used if each 'field' were to be declared separately

For languages lacking native bitfields, or where the programmer wants strict control over the resulting bit representation, it is possible to manually manipulate bits within a larger word type. In this case, the programmer can set, test, and change the bits in the field using combinations of masking and bitwise operations. [8]

Examples

C programming language

Declaring a bit field in C and C++:

```c
// opaque and show
#define YES 1
#define NO  0

// line styles
#define SOLID  1
#define DOTTED 2
#define DASHED 3

// primary colors
#define BLUE  4   /* 100 */
#define GREEN 2   /* 010 */
#define RED   1   /* 001 */

// mixed colors
#define BLACK   0                   /* 000 */
#define YELLOW  (RED | GREEN)       /* 011 */
#define MAGENTA (RED | BLUE)        /* 101 */
#define CYAN    (GREEN | BLUE)      /* 110 */
#define WHITE   (RED | GREEN | BLUE)  /* 111 */

const char * colors[8] = {"Black", "Red", "Green", "Yellow", "Blue", "Magenta", "Cyan", "White"};

// bit field box properties
struct box_props
{
     unsigned int opaque       : 1;
     unsigned int fill_color   : 3;
     unsigned int              : 4; // fill to 8 bits
     unsigned int show_border  : 1;
     unsigned int border_color : 3;
     unsigned int border_style : 2;
     unsigned char             : 0; // fill to nearest byte (16 bits)
     unsigned char width       : 4, // Split a byte into 2 fields of 4 bits
                   height      : 4;
};
```

[9]

The layout of bit fields in a C `struct` is implementation-defined For behavior that remains predictable across compilers, it may be preferable to emulate bit fields with a primitive and bit operators:

```c
/* Each of these preprocessor directives defines a single bit,
   corresponding to one button on the controller.  Button order
   matches that of the Nintendo Entertainment System. */
#define KEY_RIGHT    (1 << 0)   /* 00000001 */
#define KEY_LEFT     (1 << 1)   /* 00000010 */
#define KEY_DOWN     (1 << 2)   /* 00000100 */
#define KEY_UP       (1 << 3)   /* 00001000 */
#define KEY_START    (1 << 4)   /* 00010000 */
#define KEY_SELECT   (1 << 5)   /* 00100000 */
#define KEY_B        (1 << 6)   /* 01000000 */
#define KEY_A        (1 << 7)   /* 10000000 */

int gameControllerStatus  = 0;

/* Sets the gameControllerStatus using OR */
void keyPressed(int key) {
     gameControllerStatus  |= key;
}

/* Turns the key in gameControllerStatus off using AND and ~ (binary NOT)*/
void keyReleased(int key) {
    gameControllerStatus  &= ~key;
}

/* Tests whether a bit is set using AND */
int isPressed(int key) {
    return gameControllerStatus  & key;
}
```

Processor status register

A simple example of a bitfield status register is included in the design of the eight-bit 6502 processor. One eight-bit field held seven pieces of information:[10]

- Bit 7. Negative flag
- Bit 6. Overflow flag
- Bit 5. Unused
- Bit 4. Break flag
- Bit 3. Decimal flag
- Bit 2. Interrupt-disable flag
- Bit 1. Zero flag
- Bit 0. Carry flag

Unix process exit code

Another example is the Unix exit status code, which can be used as a flag word to pass status information to another process. For example, a program which monitors the status of eight burglar alarm switches could set the bits in the exit code, passing along to another process information about which of the switches are closed or open.

Extracting bits from flag words

A subset of flags in a flag field may be extracted by ANDing with a mask. In addition, a large number of languages, due to the shift operator's (<<) use in performing power-of-two (`(1 << n)` evaluates to 2^n) exponentiation, also support the use of the shift operator (<<) in combination with theAND operator (&) to determine the value of one or more bits.

Suppose that the status-byte 103 (decimal) is returned, and that within the status-byte we want to check the 5th flag bit. The flag of interest (literal bit-position 6) is the 5th one - so the mask-byte will be $2^5 = 32$. ANDing the status-byte 103 (`0110 0111` in binary) with the mask-byte 32 (`0010 0000` in binary) evaluates to 32, our original mask-byte, which means the flag bit is set; alternatively, if the flag-bit had not been set, this operation would have evaluated to 0.

Thus, to check the **n**th bit from a variable **v**, we can perform the operation:

```
bool nth_is_set = ( v & (1 << n)) != 0;
bool nth_is_set = ( v >> n) & 1;
```

Changing bits in flag words

Writing, reading or toggling bits in flags can be done only using the OR, AND and NOT operations - operations which can be performed quickly in the processor. To set a bit, OR the status byte with a mask byte. Any bits set in the mask byte or the status byte will be set in the result.

To toggle a bit, XOR the status byte and the mask byte. This will set a bit if it is cleared or clear a bit if it is set.

See also

- Bitboard, used in chess and similar games.
- Bit array
- Word (computer architecture)
- Mask (computing)
- Program status word

- Status register
- FLAGS register (computing)
- Control register

External links

- Explanation from a book
- Description from another wiki
- Use case in a C++ guide
- C++ libbit bit library (alternative URL)

References

1. Penn Brumm; Don Brumm (August 1988). *80386 Assembly Language: A Complete Tutorial and Subroutine Library* (https://books.google.com/books?id=qjkiAQAAIAAJ). McGraw-Hill School Education Group. p. 606. ISBN 978-0-8306-9047-3.
2. Steve Oualline (1997). *Practical C Programming* (https://books.google.com/books?id=RzmsANQ4gaAC&pg=PA403). "O'Reilly Media, Inc.". pp. 403–. ISBN 978-1-56592-306-5.
3. Michael A. Miller (January 1992). *The 68000 Microprocessor Family: Architecture, Programming, and Applications* (https://books.google.com/books?id=6qAkAQAAIAAJ). Merrill. p. 323. ISBN 978-0-02-381560-7.
4. Ian Griffiths; Matthew Adams; Jesse Liberty (30 July 2010). *Programming C# 4.0: Building Windows, Web, and RIA Applications for the .NET 4.0 Framework* (https://books.google.com/books?id=8graYKZ7rhIC&pg=PA81). "O'Reilly Media, Inc.". pp. 81–. ISBN 978-1-4493-9972-6.
5. *Data Structures* (https://books.google.com/books?id=aYxSZurAGXEC&pg=PA104). PediaPress. pp. 104–. GGKEY:JHA7HYXLECN.
6. Tibet Mimar (1991). *Programming and Designing with the 68000 Family: Including 68000, 68010/12, 68020, and the 68030* (https://books.google.com/books?id=arFQAAAAMAAJ). Prentice Hall. p. 275. ISBN 978-0-13-731498-0.
7. Prata, Stephen (2007). *C primer plus* (5th ed.). Indianapolis, Ind: Sams. ISBN 0-672-32696-5.
8. Mark E. Daggett (13 November 2013). *Expert JavaScript* (https://books.google.com/books?id=MPBZAgAAQBAJ&pg=PA68). Apress. pp. 68–. ISBN 978-1-4302-6097-4.
9. Prata, Stephen (2007). *C primer plus* (5th ed.). Indianapolis, Ind: Sams. ISBN 0-672-32696-5.
10. *InCider* (https://books.google.com/books?id=AFpRAAAAYAAJ). W. Green. January 1986. p. 108.