Lecture 23 Bit Operations CSI333

Slide 1

Bitwise Logical Operators

apply to integer type objects (char, short, int, long int and their unsigned counterparts) and give integer results defined in terms of bits.

Advice: Use unsigned long int for bitstrings when 32 bits are needed. (long guarantees 32 bits from ANSI C.)

“All unsigned types use straight binary notation, regardless of whether the signed types use 2-s complement, 1-s compl. or sign magnitude ... the sign bit is treated as an ordinary bit.” (Harbison & Steele)

$^a$also bool after conversion to int

$^b$C/C++ on general purpose computers uses 2-s complement integers, but other sign representations are possible.
Since the MIPS Instruction Set Architecture is (1) defined in terms of how bit fields code machine language instructions, and (2) MIPS has bitwise operations, C++ bitwise operations are useful to simulate MIPS in C++.

First, single bit “Boolean” or truth value operations:

Unary “not” (complement): (also denoted by ¬, \(\overline{x}\), \(~\))

\[
\begin{array}{c|c}
 x & \text{not}(x) \\
 0 & 1 \\
 1 & 0 \\
\end{array}
\]

Binary “and,” “(inclusive) or,” “exclusive or”:
(Binary means “TWO OPERANDS” here)

\[
\begin{array}{c|c|c|c|c|c}
 x & y & x \text{ and } y & x \text{ or } y & x \text{ xor } y \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 1 & 0 & 1 & 1 \\
 1 & 0 & 0 & 1 & 1 \\
 1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Other Symbols: \(\wedge, \&, \cdot\) \(\lor, |, +\) \(\text{e}or, \oplus\)

Other names for logic values:

1: on, true, asserted, enabled

0: off, false, deasserted, disabled
C++ Bitwise Operators apply Boolean operations to the individual bits of the binary integer representations.

unsigned char X = 0x0C; // 0000 1100 (binary)
unsigned char Y = 0x0A; // 0000 1010
(X & Y) == 0x04 // 0000 1000 AND
(X | Y) == 0x0E // 0000 1110 OR
(X ^ Y) == 0x06 // 0000 0110 EOR
(~ X) == 0xF3 // 1111 0011 COMPL

---

Caution: It’s a common bug to confuse Bitwise operations with Logical AND, OR, NOT:

(X && Y) == 1 if X!=0 and Y!=0, 0 otherwise
(X || Y) == 1 if X!=0 or Y!=0, 0 otherwise
(! X) == 0 if X!=0, 1 if X==0

In C/C++, consider any non-zero int or pointer as “true,” and 0 as “false.”

(In C++, non-zero ints or pointers are converted to bool true, zero converts to false, true converts to 1 and false to 0).
Shifts For integral \( X \), \( AMT \), \( AMT \geq 0 \)
\( X << AMT \) is \( X \) shifted Left \( AMT \) bit positions. Zero bits are shifted in from the right.
Example: \((0x000F8001 << 3) = 0x007C0008\)
For unsigned \( X \) or \( X \geq 0 \)
\( X >> AMT \) is \( X \) shifted Right \( AMT \) bit positions. Zero bits are shifted in from the left.
Example: \((0x007C0008 >> 3) = 0x000F8001\)
For signed \( X \), \( X < 0 \), in \( X >> AMT \),
whether 0s or 1s are shifted in is IMPLEMENTATION DEPENDENT!!

Application to \texttt{tmips} project

```
long unsigned int IR;  //to hold TMIPS instr.
const int SHOP = 32 - 6;
const int MASKOP = 0x3F << SHOP;
const int OPSPECIAL = 0 << SHOP;
const int OPLW = 35 << SHOP;
switch( IR & MASKOP ) {
  case OPSPECIAL:  // handle insns with opcode==0
      break;
  case OPLW:      // handle LW instruction
      break;
  // ...
  default:  // handle illegal instruction, halt.
```
Another style..

```c
long unsigned int IR; //to hold TMIPS instr.
const int SHOP = 32 - 6;
const int OPSPECIAL = 0;
const int OPLW = 35;
switch( IR >> SHOP ) {
    case OPSPECIAL: // handle insns with opcode==0
        break;
    case OPLW:     // handle LW instruction
        break;
    // ...
    default:       // handle illegal instruction, halt.
}
```

Extracting an internal field

```c
int baseregno( long unsigned int Insn )
{
    const int SHBASE   = 5 + 16; //rt+offset bits
    const int MASKBASE = 0x1F;
    return ( Insn >> SHBASE ) & MASKBASE ;
    // alternative:
    //const int MASKBASE = 0x1F << SHBASE;
    //return ( Insn & MASKBASE ) >> SHBASE ;
}
```
Bitwise Op. Applications

1. Extract or compose bit fields when format is externally defined. (Hardware simulation, device driver software, network packet analysis/synthesis).

2. Work on small, fixed universe subsets efficiently. (ios flags, Strou. (6.2.4) and p.616-7.)

3. Efficient special case arithmetic operations:

```c
if( X & 3 ) { /* X is not a multiple of 4 */ } 
Y = X & (~0x3FF);
    /* round down to nearest 1K multiple */
if( (X & 1) == 0 ){ /* X is even */ }
```

How would you sort, using about 1 Megabyte of memory, a few million 7 digit telephone numbers from a disk file once every 1/2 hour, so that we can rapidly tell if a number is assigned?

A discussion is given in Jon Bentley’s “Programming Pearls” column in the December 1999 issue of Dr. Dobb’s Journal.
Saving Space (Strou. C.8)

1. Put more than one small object into a byte.

2. Use the same space to hold different (types) of objects at different times.

Bitwise (mask and shift) operations are portable: They denote the same mathematical operations on equal sized integers, regardless of Endianness. But other C++ features can help save space.

1. Bit Fields: Specify integral fields in a struct with given bit lengths. The compiler tries to pack them tightly, in a non-portable way: Depends on Byte-Ordering and Alignment Boundaries.

2. Unions: Specify two or more different fields with different types or sizes that will occupy the same memory space in a struct. Program logic or other fields must keep track of the kind of value currently in the union.

Unions “shouldn’t” be used for type conversion, since C++ has cast operations that check the validity of the conversion.
Unions can be used to explore what data representations a system actually uses.

If you try to access bytes by byte offsets in a char array overlaid on the TMIPS word array, the byte selection will depend on the C++ machine’s byte order.

For simplicity, the TMIPS machine **does not have** byte instructions.