Computer Labs: C for Lab 2

2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

September 22, 2011

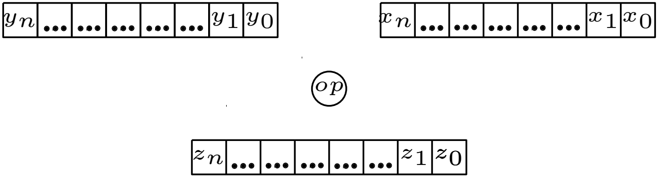
Contents

Bitwise Operators and Shifts

C Unions

Bitwise Operations

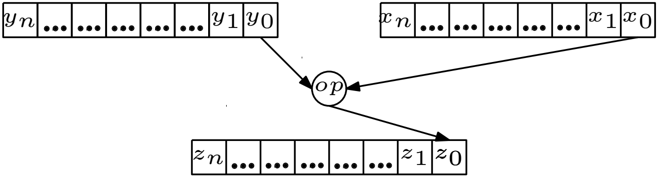
- Bitwise operations
  - are boolean operations, either binary or unary
  - take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
  - apply the operation on every bit of these operands



y_n ... ... y_1 y_0
x_n ... ... x_1 x_0
op
z_n ... ... z_1 z_0

Bitwise Operations

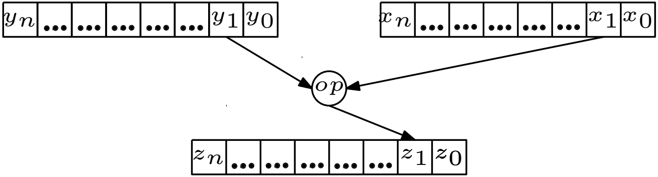
- Bitwise operations
  - are boolean operations, either binary or unary
  - take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
  - apply the operation on every bit of these operands



y_n ... ... y_1 y_0
x_n ... ... x_1 x_0
op
z_n ... ... z_1 z_0

Bitwise Operations

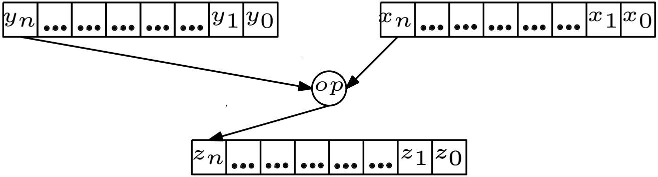
- Bitwise operations
  - are boolean operations, either binary or unary
  - take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
  - apply the operation on every bit of these operands



y_n ... ... y_1 y_0
x_n ... ... x_1 x_0
op
z_n ... ... z_1 z_0

Bitwise Operations

- Bitwise operations
  - are boolean operations, either binary or unary
  - take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
  - apply the operation on every bit of these operands



Bitwise Operators

- Bitwise operators:

  & bitwise AND

  | bitwise inclusive OR

  ^ bitwise exclusive OR

  ~ one's complement (unary)

- Do not confuse them with the logical operators which evaluate the truth value of an expression:

  && logical and

  || logical or

  ! negation

Bitwise Operators: Application

- Use with bit masks:

```
uchar mask = 0x80;     // 10000000b
...
if ( flags & mask )    // test value of flags MS bit
   ...
flags = flags | mask;  // set flags MS bit
flags ^= mask;         // toggle flags MS bit
mask = ~mask;          // flags becomes 01111111b
flags &= mask;         // reset flags MS bit
```

- In Lab 1, you can use the `|` operator to compose the attribute byte:

```
#define RED 0x04         // Foreground color RED
#define GREEN_BACK 0x20  // Background color GREEN

uchar ch_attr = RED | GREEN_BACK;
```

Shift Operators

- Similar to corresponding assembly language shift operations

  >> left shift of left hand side (LHS) operand by the number of bits positions given by the RHS operand

  - Vacated bits on the left are filled with:

    0 if the left operand is unsigned (logical shift)

    either 0 or 1 (machine/compiler dependent] if the left operand is signed

  << right shift

  - Vacated bits on the right are always filled with `0`’s

- LHS operand must be of an integral type
- RHS operand must be non-negative

Shift Operators: Application

- Integer multiplication/division by a power of 2:

  ```
  unsigned int n;

  n <<= 4;   // multiply n by 16 (2^4)
  n >>= 3;   // divide n by 8 (2^3)
  ```

- In Lab 1, we can use them to avoid mistakes in the definition of the attributes:

  ```
  #define BLUE       (1<<0)
  #define GREEN      (1<<1)
  #define RED        (1<<2)
  #define BACK_SHIFT 4
  #define GREEN_BACK (GREEN << BACK_SHIFT)
  ```

Contents

Bitwise Operators and Shifts

C Unions

C Unions

- Syntatically, a union data type appears like a struct:

```
union reg_a {
   unsigned char a;    // 8080 A register
   unsigned short ax;  // 8086 AX register
   unsigned long eax;  // 80386 EAX register
} xax;
```

  - Access to a union's members is via the dot operator

- However semantically, there is a big difference:

  Union contains space to store any of its members, but not all of its members simultaneously

  - The name **union** stems from the fact that a variable of this type can take any

  Struct contains space to store all of its members simultaneously

Question What are unions good for?

C Union and Type Conversion

```
union reg_a {
   struct {
        unsigned char al, ah, _eax[2]; // access as 8-bit re
   } b;
   struct {
        unsigned short ax, _eax;  // access as 16-bit regis
   } w;
   struct {
        unsigned long eax;  // access as 32-bit register
   } l;
} ia32_a;
```

- This allows us to initialize the union as a 32-bit register

```
ia32_a.l.eax = 0xD0D0DEAD;
```

- And later access any of the smaller registers available in the IA 32 architecture

```
printf("EAX = 0x%p \t AX = 0x%x \t AH = 0x%x \t AL = 0x%x \n",
       ia32_a.l.eax, ia32_a.w.ax, ia32_a.b.ah, ia32_a.b.al);
```

Question What are the assumptions underlying this code?