Computer Labs: C Topics for Lab 2

2º MIEIC

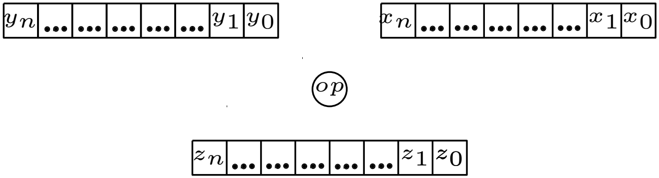
Pedro F. Souto (`pfs@fe.up.pt`)

September 24, 2013

Contents

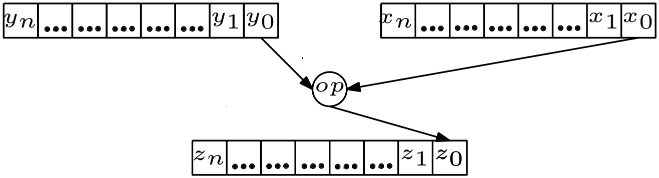
Bitwise Operations

- 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

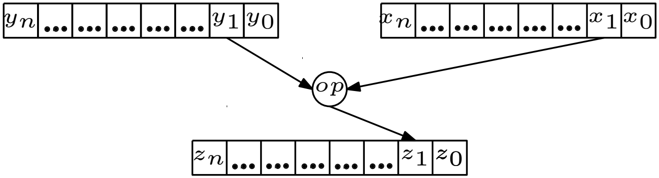
- Bitwise operations
  - are boolean operations, either binary or unary
  - take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
  - apply the operation on every bit of these operands



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

Bitwise Operations

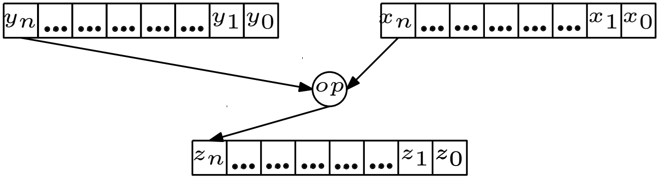
- Bitwise operations
  - are boolean operations, either binary or unary
  - take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
  - apply the operation on every bit of these operands



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



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

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;            // mask becomes 01111111b
flags &= mask;           // reset flags MS bit
```

- In Lab 2, you can use the `|` operator to select a graphics mode using the linear memory model

```
#define LINEAR_MODEL_BIT 0x40

mode |= LINEAR_MODEL_BIT;
```

Shift Operators

- Similar to corresponding assembly language shift operations

  $>>$ right 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 LHS is unsigned (logical shift)

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

  $<<$ left 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)
```

- Flags definitions (to avoid mistakes)

```
#define LINEAR_MODEL_BIT 14

#define BIT(n) (0x1 << (n))

mode |= BIT(LINEAR_MODEL_BIT);
```

Contents

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 of the types of its members

  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);
```