Computer Labs: C Topics for Lab 2

2º MIEIC

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

October 7, 2014

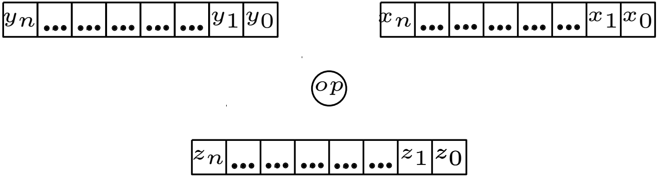
Contents

Bitwise and Shift Operators

C Integer Conversion

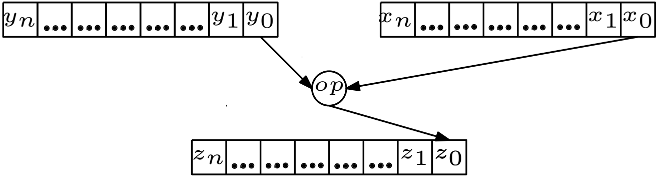
Bitwise Operations

- 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

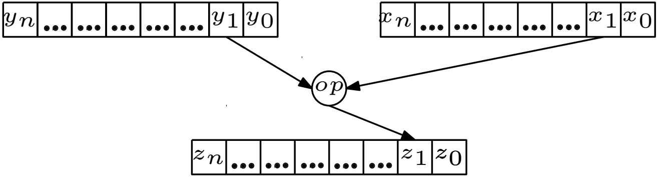
Bitwise Operations

- 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

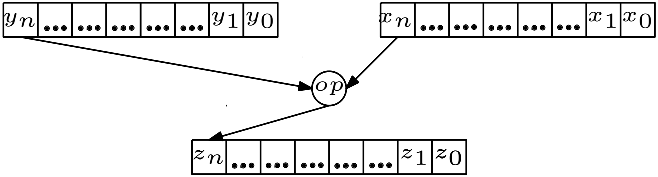
Bitwise Operations

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

- In Lab 2, you can use the `|` operator to select the `TIMER_RB_CMD` (Read-back Command)

```
#define TIMER_RB_CMD 0xC0

cmd |= TIMER_RB_CMD;
```

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 BIT(n) (0x1 << (n))

#define TIMER_RB_CMD (BIT(7)|BIT(6))

cmd |= TIMER_RB_CMD;
```

Contents

Bitwise and Shift Operators

C Integer Conversion

C Integer Conversion Rules

- C supports different integer types, which differ in their:

  Signedness i.e. whether they can represent negative numbers
  Precision i.e. the number of bits used in their representation

- The C standard specifies a set of rules for conversion from one integer type to another integer type so that:
  - The results of code execution are what the programmer expects
- One such rule is that:
  - Operands of arithmetic/logic operators whose type is smaller than `int` are promoted to `int` before performing the operation

  the rational for this is

  - To prevent errors that result from overflow. E.g:

```
signed char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3;
```

source: CMU SEI

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;
uint8_t result_8 = ( ~port ) >> 4;
```

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;
uint8_t result_8 = ( ~port ) >> 4;
```

Question: What is the value of `result_8`?

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;
uint8_t result_8 = ( ~port ) >> 4;
```

Question: What is the value of `result_8`?

Answer: Most likely, you'll think in terms of 8-bit integers:

Expr.	**8-bit**
`port`	`0x5a`
`~port`	`0xa5`
`(~port)>>4`	`0x0a`
`result_8`	`0x0a`

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;
uint8_t result_8 = ( ~port ) >> 4;
```

Question: What is the value of `result_8`?

Answer: ... but because of integer promotion, need to think in terms of `sizeof(int)`:

Expr.	8-bit	32-bit
`port`	`0x5a`	`0x0000005a`
`~port`	`0xa5`	`0xffffffa5`
`(~port)>>4`	`0x0a`	`0xfffffffa`
`result_8`	`0x0a`	`0xfa`

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;
uint8_t result_8 = ( ~port ) >> 4;
```

Question: What is the value of `result_8`?

Answer: ... but because of integer promotion, need to think in terms of `sizeof(int)`:

Expr.	8-bit	32-bit
`port`	`0x5a`	`0x0000005a`
`~port`	`0xa5`	`0xffffffa5`
`(~port)>>4`	`0x0a`	`0xfffffffa`
`result_8`	`0x0a`	`0xfa`

Solution: One way to fix this is to use a cast on the value after the complement:

```
uint8_t port = 0x5a;
uint8_t result_8 = (uint8_t) ( ~port ) >> 4;
```

This truncates the result of the complement to its LSB, and therefore the right shift works as expected

Further Reading

- INT02-C. Understand integer conversion rules