UNIVERSITY OF DUBLIN

XMA12621

TRINITY COLLEGE

Faculty of Science

school of mathematics

JF Maths
SF TSM

Trinity Term 2010

Mathematics 1261, 1262: Computation

Thursday, May 06 RDS - MAIN 09.30 12.30

Dr. A. Nolan, Dr. C. Ó Dúnlaing

Students should attempt three questions from each module.

Log tables are available from the invigilators, if required.

Non-programmable calculators are permitted for this examination,—please indicate the make and model of your calculator on each answer book used.

Module 1261

1. (a) Convert -1054 and 1307 to 2s complement short integers (in hexadecimal), and compute the sum as a short integer. 'Little endian' form is not needed.

```
     65 r 14      4 r 1     4,1,14 = 041e   ffff   fbe1+1   fbe2
16)1054       16)65                        - 41e
     81  r 11     5 r 1     5,1,11 = 051b                     051b
16)1307                                                       ----
                                                             100fd
answer 00fd
```

   (b) Calculate the single-precision floating point form of $-136.0/9$, little endian.

```
sign 1
136 = 8 * 17
exponent 3 -> 130; 1000 0010.  It is best computed as
follows
   0111 1111
  +       11
  ----------
   1000 0010
mantissa 17/9
```

To convert this to a ‘repeating binary fraction,’

$$1.b_1b_2b_3\ldots$$

First subtract 1: 8/9. Thus $8/9 = .b_1b_2b_3\ldots$
To get b_1, double: $16/9 = b_1.b_2b_3\ldots$
This is ≥ 1, so $b_1 = 1$.
Subtract 1: $7/9 = 0.b_2b_3\ldots$
Double : $14/9 = b_2.b_3b_4\ldots$
This is ≥ 1, so $b_2 = 1$; because $b_2 = 1$, we subtract 1:
$5/9 = 0.b_3\ldots$ and so on. Briefly put, we develop the following sequence.

```
17/9  8/9 7/9 5/9 1/9 2/9 4/9 8/9
1.111000111000111000
Check:
1+7*8*(1/64 + 1/64^2 ...) = 1 + 56/63 = 1 + 8/9
mantissa
1110 0011 1000 1110 0011 100 | 0 1110 0011 1000
Round down
1 10000010 1110 0011 1000 1110 0011 100

1100 0001 0111 0001 1100 0111 0001 1100
   c    1    7    1    c    7    1    c

1c c7 71 c1 little endian.
```

2. (a) Simulate the following program, showing what gets printed.

```
void xxx ( int n )
{
  int x,y, i;

  x = 0;
  y = 1;
  for ( i=0; i<n; ++i )
  {
    x = x + y;
    y = y + 2;
  }

  printf("%d, %d, %d\n", n, x, y);
}

main()
{
   xxx ( 4 );
}
```

```
  n   x   y   i  i < n?
  4   0   1
              0    1
      1   3
              1    1
      4   5
              2    1
      9   7
              3    1
     16   9
              4    0
prints:
4,16,9
```

(b) In general (in terms of n), what gets printed?
$n, n^2, 2n+1$.

(c) In the above code, change the statement

```
y = y+2;
```

to

```
y = y + 1;
```

Then simulate the altered program.

```
  n   x   y   i  i < n?
  4   0   1
              0    1
      1   2
              1    1
      3   3
              2    1
      6   4
              3    1
     10   5
              4    0
prints:
4,10,5
```

3. (a) Write a program which reads (integers) m, n from the command line and calculates and prints m^n. It should use a for-loop, and *not* use functions from `math.h` Assume $n \geq 0$.

```
#include <stdio.h>
#include <stdlib.h>
main( int argc, char * argv[] )
{
  int x,m,n,i;
  m = atoi( argv[1] );
  n = atoi( argv[2] );

  x = 1;
  for (i=0; i<n; ++i)
    x = x * m;

  printf("%d\n", x);
}
```

(b) Write a function `char * copy_string ( char * s )` which returns a copy of the string s. It should use `malloc` or `calloc` to get enough memory in which to store the copy.

```
char * copy_string ( char * s )
{
  int size;
  char * x;

  size = strlen ( s ) + 1;
  x = ( char * ) malloc ( size );
  snprintf ( x, len, "%s", s );
  return x;
}
```

4. (a) Given the declaration

```
  int a[3][4];
```

suppose `a[0][0]` begins at address 1000. What is the value of `a[2]`? What is the address of `a[1][2]`? $1000 + 2 \times 4 \times 4 = 1032$
$1000 + 4 \times 4 + 2 \times 4 = 1024$.

(b) Given `x` is a `double`, explain why one of the following statements is right and one is wrong.

```
    scanf("%f", &x);
    printf("%f\n", x);
```

The first is wrong, as it only fills the first 4 bytes of x. The second is all right, as all floats are converted to double before printing, and there is no need for C to distinguish between them.

(c) What gets printed by the following code?

```
printf("%d\n", ( 1+2/3   ) > 1);
printf("%d\n", ( 1.0+2/3 ) > 1);
printf("%d\n", ( 1+2/3.0 ) > 1);
```

First printf: $1 + 2/3$ evaluates to 1, so the relation evaluates to 0, and 0 is printed.
Second: $1.0 + 2/3$ evaluates to double-precision 1.0, and 1 is converted to double precision, the relation evaluates to 0, and 0 is printed.
Third: $1 + 2/3.0$ evaluates to $1.6666\ldots$ double precision, and 1 is converted to double, so the relation is true and evaluates to 1: 1 is printed.

Thus the following is printed.

```
0
0
1
```

Module 1262

In 2010, this module was numerical analysis. This year the second module will be C++ programming.