Assembly Language Mini-project

Foundations of Computational Systems (FSC)

December 2023

1 Introduction

The objective of this project is to design and test a set of assembly language routines for unsigned integers of 128 bits (16 bytes).[1] You show your routines at work, you should create a program that reads a value n and prints out the corresponding factorial ($n!$).

You should develop and test your work with RARS.

2 Project Description and Overall Design

The 128-bit unsigned numbers should be represented by a vector of four 32-bit integers (words). The least-significant word is the stored in the first position.

An 128-bit number can be declared like this:

```
.data
qnum:    .word  0x1000000A,0x2000000B,0x3000000C,0x4000000D
# qnum is 0x4000000D3000000C2000000B1000000A
# decimal 85070592775056009246371375814044286986
```

You should design and test the following routines:

1. `qadd`: adds two 128-bit numbers;
2. `qmult`: multiplies two 128-bit numbers;
3. `qtoa`: converts an 128-bit number to ASCII (in hexadecimal);
4. `atoq`: converts an ASCII string (representing a number in hexadecimal) to an 128-bit number;
5. `itoq`: converts a 32-bit unsigned number to an 128-bit number.

The details of the subroutines are described in Table 1

[1] These are sometimes called "quadwords."

Name	Arguments	Result
`qadd`	`a0`: address of a q-number `a1`: address of a q-number `a2`: address of result	sum stored at address `a2`; `a0`: 0 if no overflow, 1 if overflow.
`qmult`	`a0`: address of a q-number `a1`: address of a q-number `a2`: address of result	product stored at address `a2`; `a0`: 0 if no overflow, 1 if overflow.
`qtoa`	`a0`: address of q-number `a1`: address of area for string	string stored at address `a2`
`atoq`	`a0`: address of string `a1`: address of q-number	Converted value stored at address `a1`; `a0`: 0 if no error, 1 if error.
`itoq`	`a0`: 32-bit value `a1`: address of q-number	Converted value stored at address `a1`.

Table 1: Subroutines to be implemented

For example, qadd could be used as follows.

```
.data
qnum1: .word  0x1000000A,0x2000000B,0x3000000C,0x4000000D
# decimal 85070592775056009246371375814044286986
qnum2: .word  0x00000000,0x20000000,0x30000000,0x40000000
# decimal:  85070591745089896339574058165945237504
qres:    .space 16      # reserve 16 bytes
.text
    la      a0,qnum1
    la      a1,qnum2
    la      a2,qres
    call    qadd
    bne     a0, zero, error
    ...     # result is in memory at qres
            # 0x8000000d6000000c4000000b1000000a
            # stored as 0x1000000a, 0x6000000c, 0x4000000b, 0x8000000d
error: ...  # handle error (there should be none in this case)
```

Test your subroutines individually; afterwards use them to create a program that:

1. asks the user for an unsigned 32-bit number;
2. calculates the factorial using 128-bit arithmetic (use a loop, not recursion);
3. prints the result.

Using 128-bit arithmetic, you should obtain correct results up to 34!.

The expected values are:

```
1! = 0x1
2! = 0x2
3! = 0x6
4! = 0x18
5! = 0x78
6! = 0x2d0
7! = 0x13b0
8! = 0x9d80
9! = 0x58980
10! = 0x375f00
11! = 0x2611500
12! = 0x1c8cfc00
13! = 0x17328cc00
14! = 0x144c3b2800
15! = 0x13077775800
16! = 0x130777758000
17! = 0x1437eeecd8000
18! = 0x16beecca730000
19! = 0x1b02b9306890000
20! = 0x21c3677c82b40000
21! = 0x2c5077d36b8c40000
22! = 0x3ceea4c2b3e0d80000
23! = 0x57970cd7e2933680000
24! = 0x83629343d3dcd1c00000
25! = 0xcd4a0619fb0907bc00000
26! = 0x14d9849ea37eeac91800000
27! = 0x232f0fcbb3e62c3358800000
28! = 0x3d925ba47ad2cd59dae000000
29! = 0x6f99461a1e9e1432dcb6000000
30! = 0xd13f6370f96865df5dd54000000
31! = 0x1956ad0aae33a4560c5cd2c000000
32! = 0x32ad5a155c6748ac18b9a580000000
33! = 0x688589cc0e9505e2f2fee5580000000
34! = 0xde1bc4d19efcac82445da75b00000000
```