

Tutorial 6: Functions

Problem 1: 40-bit Addition

We like to design a function that does addition in 40 bits. First try the following 40-bit additions by hand. Next design this function by developing a C code for it. Modify this function to do addition as well as subtraction.

$0x08\ 0000\ 0020 + 0x02\ 0000\ 0040 = 0x0a\ 0000\ 0060 = +\ 42949673056 > (+\ 2147483647$ that is possible in 32 bits)

$0x08\ 0000\ 0020 + 0xf7\ 0000\ 0040 = 0xff\ 0000\ 0060 = -\ 4294967200 < (-\ 2147483648$ that is possible in 32 bits)

$0x08\ 0000\ 0020 + 0x07\ ffff\ ffff = 0x10\ 0000\ 001f = +\ 68719476767 > (+\ 2147483647$ that is possible in 32 bits)

A function for 40-bit addition will do this in two steps. First step is to declare a structure for the representation of a 40-bit number.

```
struct int40 {char hi8;  
              int lo32};
```

In this structure a 40-bit number is represented as concatenation of a 32-bit number to represent lower 32 bits, and an 8-bit number to represent the upper 8 bits. The addition of two 40-bit operands (m) and (n) can be carried out by adding the ($lo32$) members of the two operands together and the ($hi8$) members serially together with the carry out from the first addition as shown in Figure 1.

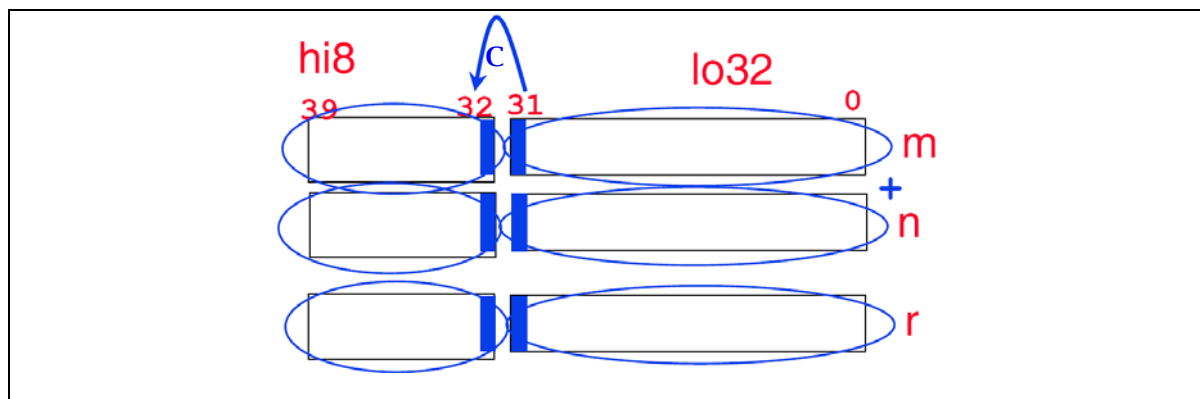


Figure 1: 40-bit Addition Procedure

The challenge is to figure out what the value of the carryout (C) from the addition of ($m.lo32$) and ($n.lo32$) is. We recognise the fact that are 3 conditions that generate the carryout (C). One condition is that bit 31 of (m) and (n) are both 1, i.e. ($m.lo32 = 1$) and ($n.lo32 = 1$). The other condition is that either of (m) or (n) has its bit 31 set to (1) and bit 31 of (r) is set to 0, i.e. ($(m.lo32 = 1) \text{ or } (n.lo32 = 1)$) and ($r.lo32 = 0$). The C function 4-bit adder is presented in Figure 2.

```

#include <stdio.h>

struct int40 { char hi8;
               int lo32;
               };

struct int40 add_40(struct int40 m, struct int40 n)
{
    struct int40 r;
    unsigned msb_m_lo32 = m.lo32 >> 31, msb_n_lo32 = n.lo32 >> 31, msb_r_lo32 ,C;

    r.lo32 = m.lo32 + n.lo32;
    msb_r_lo32=r.lo32 >> 31;

    C = ((msb_m_lo32 & msb_n_lo32) || (msb_m_lo32 & ~msb_r_lo32) ||
         (msb_n_lo32 & ~msb_r_lo32));
    r.hi8 = m.hi8 + n.hi8 + C;
    return r;
}

int main (void)
{
    struct int40 a = {0x8,0x20}, b = {0x2,0x40}, c;
    c = add_40(a, b);
    printf("%x%8.8x\n\n", c.hi8, c.lo32);
    b.hi8 = 0xf7;
    c = add_40(a, b);
    printf("%x%8.8x\n\n", c.hi8, c.lo32);
    b.lo32 =0xffffffff;
    b.hi8 =0x7;
    c = add_40(a, b);
    printf("%x%8.8x\n\n", c.hi8, c.lo32);
}

```

Figure 2: The C-Program for 40-bit Addition Function

For subtraction operation, the addition of carryout (c) with (hi8) members should be replaced with subtraction by borrow from (B). We recognise the fact that there are 3 conditions that generate the borrow from (B). One condition is that bit 31 of (m) is set to 0 and bit 31 of (n) is set to 1, i.e. (m.lo32 = 0) and (n.lo32 = 1). The other condition is that bits 31 of both (m) and (n) are both 1 or 0 and bit 31 of (r) is set to 1, i.e. ((m.lo32 = n.lo32) and (r.lo32 = 1)). The corresponding program to do both addition and subtractions in 40-bit data representation is presented in Figure 3. The by hand arithmetic below is to check the program in Figure 3.

0x08 0000 0020 + 0x02 0000 0040 = 0x0a 0000 0060 = + 42949673056 > (+ 2147483647 that is possible in 32 bits)

0x08 0000 0020 - 0xf7 0000 0040 = 0x10 0000 0060 = + 68719476832 > (+ 2147483647 that is possible in 32 bits)

0x08 0000 0020 - 0x07 ffff ffff = 0x00 0000 0021 = + 33

```

#include <stdio.h>

struct int40 { char hi8;
               int lo32;
               };

struct int40 add_sub_40(struct int40 m, struct int40 n, char operation)
{
    struct int40 r;
    unsigned msb_m_lo32 = m.lo32 >> 31, msb_n_lo32 = n.lo32 >> 31, msb_r_lo32 ,C,
        B;

    if (operation == '+')
    {
        r.lo32 = m.lo32 + n.lo32;
        msb_r_lo32=r.lo32 >> 31;

        C = ((msb_m_lo32 & msb_n_lo32) || (msb_m_lo32 & ~msb_r_lo32) ||
            (msb_n_lo32 & ~msb_r_lo32));
        r.hi8 = m.hi8 + n.hi8 + C;
        return r;
    }

    if (operation == '-')
    {
        r.lo32 = m.lo32 - n.lo32;
        msb_r_lo32=r.lo32 >> 31;

        B = ((~msb_m_lo32 & msb_n_lo32) || (!(msb_m_lo32 ^ msb_n_lo32) &&
            msb_r_lo32));

        r.hi8 = m.hi8 - n.hi8 - B;
        return r;
    }
}

int main (void)
{
    struct int40 a = {0x8,0x20}, b = {0x2,0x40}, c;
    c = add_sub_40(a, b, '+');
    printf("%x%8.8x\n\n", c.hi8, c.lo32);
    b.hi8 = 0xf7;
    c = add_sub_40(a, b, '-');
    printf("%x%8.8x\n\n", c.hi8, c.lo32);
    b.lo32 =0xffffffff;
    b.hi8 =0x7;
    c = add_sub_40(a, b, '-');
    printf("%x%8.8x\n\n", c.hi8, c.lo32);
}

```

Figure 3: The C-Program for 40-bit Addition and Subtraction Function

If the C compiler supports (long long) type (gcc does!) then a simpler function can be written for the 40-bit arithmetic operations as shown in Figure 4.

```

#include <stdio.h>

long long arithmetic_40(long long m, long long n, char operation)
{
    long long r;

    if (operation == '+')
    {
        r = (m + n) & (0x000000fffffffffff);
        return r;
    }
    if (operation == '-')
    {
        r = (m - n) & (0xfffffffffffffffffff);
        return r;
    }
    if (operation == '*')
    {
        r = (m * n) & (0x000000fffffffffff);
        return r;
    }
    if (operation == '/')
    {
        r = (m / n) & (0x000000fffffffffff);
        return r;
    }
}

int main (void)
{
    long long a = 0x800000020, b =0x200000040, c;
    int p1;
    char p2;
    c = arithmetic_40(a, b, '+');
    p1 = (int) c;
    p2 = (char) (c >>32);
    printf("%2.2x%8.8x\n\n", p2,p1);
    b = 0xf70000020;
    c = arithmetic_40(a, b, '-');
    p1 = (int) c;
    p2 = (char) (c >>32);
    printf("%2.2x%8.8x\n\n", p2,p1);
    b =0x7fffffff;
    c = arithmetic_40(a, b, '-');
    p1 = (int) c;
    p2 = (char) (c >>32);
    printf("%2.2x%8.8x\n\n", p2,p1);
    c = arithmetic_40(a, b, '*');
    p1 = (int) c;
    p2 = (char) (c >>32);
    printf("%2.2x%8.8x\n\n", p2,p1);
    c = arithmetic_40(a, b, '/');
    p1 = (int) c;
    p2 = (char) (c >>32);
    printf("%2.2x%8.8x\n\n", p2,p1);
}

```

Figure 4: The C-Program for 40-bit Arithmetic Function Using long long type

Problem 2: Reverse Polish Calculator

Write an assembly program for calculation in Reverse Polish Notations.

In reverse polish calculators the operators are post fix. For example the infix expression $(3 + 8) * 5 - (2 + 6) * 4$ is represented in post fix reverse polish notations as: $3 8 + 5 * 2 6 + 4 * -$. In post fix notation the operators follow the operands. Also, there is no need for parentheses.

We can implement this calculator using a stack. The operands are read from the input and progressively pushed onto the stack, until an operator is detected, upon which the arithmetic operation is performed and the result is again pushed onto the stack. Figure 5 is an example of implementation of reverse polish calculation. In the rudimentary example of Figure 5 only three operators “+”, “-” and “*” are included, and only integer numbers are considered. Each input is terminated with an “Enter Key” (ASCII = 10 = \D). The stream is ended with (\Z).

```

__start:

b main

main: str lr, [sp,#-4]!; store return address
      ldr a1,=input_stream
loop: bl get_input
      mov v1, a1
      bl isnumber
      cmp a1, #1 ; Input is a number
      streq v1, [sp,#-4]! ; push number onto stack
      b loop
      cmp v1, '\z' ; End of Stream
      beq fin
oper: ldr v2, [sp] ,#+4
      ldr v3, [sp] ,#+4
      cmp v1, #'+'
      addeq v2,v2,v3
      streq v2, [sp,#-4]!
      beq loop
      cmp v1, #'-'
      subeq v2,v2,v3
      streq v2, [sp,#-4]!
      beq loop
      cmp v1, #'*'
      muleq v1,v2,v3
      streq v1, [sp,#-4]!
      beq loop
      bne exit

fin: ldr v1, [sp] ,#+4
     ldr a1,=result

exit: ldr pc, [sp],#4; load return address

input_stream:
.ascii "3\D8\D+\D*\D2\D6\D+\D4\D*\D-\D*\D*\Z" ; Each input
; is ended with Enter
; key '\D' and the stream is ended
; with '\Z'

result:
.word 0

get_input:
---
---
---
mov pc lr

_isnumber:
---
---
---
mov pc lr

```

Figure 5: Rudimentary ARM Assembly Program for Reverse Polish Calculation

Problem 3: Parameter Passing to Function

Consider the C code in Figure 6 and the printout from it in Figure 7.

Explain if you find anything unusual with the printouts in Figure 7. Specifically look at correspondence between the addresses and values of the array elements printed out in Figure 7.

```
#include <stdio.h>

void array_check(int a, int b, int c, int d)
{
    printf("Address      Value \n",&a, a, &b, b, &c, c, &d, d);
    printf("%x      %x\n%x      %x\n%x      %x\n",&a, a, &b, b,
    &c, c, &d, d);
}

int main(void)
{
    int arr [] = {1,2,3,4,5,6};

    array_check(arr[0], arr[1], arr[2], arr[3]);
    array_check(arr[3], arr[2], arr[1], arr[0]);

    return 0;
}
```

Figure 6: Passing Parameters in C-Program

Address	Value
bffff270	1
bffff274	2
bffff278	3
bffff27c	4
Address	Value
bffff270	4
bffff274	3
bffff278	2
bffff27c	1

Figure 7: The Outputs of printf Statements

In C Program of Figure 6 we call function (array_check) twice by passing parameters (arr[0], arr[1], arr[2], arr[3]) and (arr[3], arr[2], arr[1], arr[0]). Although the right values are printed out, the addresses are the same. That means (arr[0] = 1) and (arr[3] = 4) share the same address (0xbffff270) in two calls to function (array_check). Similarly, (arr[1] = 2) and (arr[2] = 3) share the same address (0xbffff274) in two calls to function (array_check). The same behaviour is demonstrated for the pairs (arr[2] = 3), (arr[1] = 2), and (arr[3] = 4), (arr[0] = 1).

The reason is that function (array_check) gets a copy of array elements through (int a, int b, int, c, int d). It is called parameter passing by value. The printf statement in function (array_check) prints the addresses of the copies of the array elements and not addresses of the original array elements. The memory location for copies are allocated on the function stack frame in the same order each time function (array_check) is called. Therefore, in the first call to (array_check) copies of (arr[0], arr[1], arr[2], arr[3]) are pushed onto the stack at addresses (0xbffff270, 0xbffff274, 0xbffff278, 0xbffff27c). In the second call to (array_check) copies of (arr[3], arr[2], arr[1], arr[0]) are pushed onto the stack at same addresses (0xbffff270, 0xbffff274, 0xbffff278, 0xbffff27c).