1. Convert -6.625 to 32-bit IEEE single precision format
   1. 110.101 .625
   2. 1.10101 -.5
   3. 1.10101 x 2^2 .125
   4. 127 + 2 = 129 10000001
   5. 1100 0000 1101 0100 0000 0000 0000 0000
   6. C 0 D 4 0 0 0 0 0
   7. 0xC0D40000
2. Convert 123.457 to 32 32-bit IEEE single precision format.
   1. 123.457 .457

1111011.011101 -.25

1.111011011101 x 2^6 -.125

127 + 6 = 133 -.0625

10000101 front -.015625

0 so 011101

0100 0010 1111 0110 1110 1000 0000 0000

4 2 F 6 E 8 0 0

0x42F6E800

1. What floating-point number is represented by **0x41BA0000**.
   1. 0100 0001 1011 1010 0000 0000 0000 0000
   2. 131 – 127 = 4
   3. 011 1010 0000 0000 0000 0000
   4. 10111.010…

d. 10111.01

e. 23.25

1. Assume we are multiplying the unsigned integers **1011 X 1011**. Trace the values of the multiplicand, multiplier, and result at every step. (We are not covering this algorithm until Monday April 11).

|  |  |  |
| --- | --- | --- |
| multiplicand | multiplier | Result |
| 1011 | 1011 | 1011 |
| 10110 | 101 | 10110 |
| 101100 | 10 | 000000 |
| 1011000 | 1 | 1011000 |
|  | Final result: | 1111001 |

1. The swap function below exchanges the two double values pointed to by **x** and **y**. Write **swap** as an ARM assembly language function. Full credit for the most concise version.

**void swap(double \*x, double \*y) {**

**double tmp = \*x;**

**\*x = \*y;**

**\*y = tmp;**

**}**

**Swap:**

**Ldr d0, [r0]**

**Ldr d1, [r1]**

**Str d0, [r1]**

**Str d1, [r0]**

**Bx lr**

1. Write a recursive C function that implements the declaration below. **popcount** counts the number one bits in the binary representation of its argument. For example, **popcount(30)** is 4 because 30 in binary is 11110, which has four one bits.

**extern int popcount(unsigned int n);**

int popcount\_rec(int n) {

if (n == 0) {

return 0;

}else{

int sum = (n & 1);

sum = sum + popcount(n >> 1);

}

}

1. Write **popcount** as an ARM assembly language function.

.global popcount\_rec

.cpu cortex-a53

.text

popcount\_rec:

push { r4-r6, lr}

mov r4, r0

cmp r4, #0

beq endif

bl else

endif:

mov r0, #0

pop { r4-r6, lr}

bx lr

else:

and r5, r4, #1

lsr r4, r4, #1

mov r0, r4

bl popcount\_rec

add r5, r5, r0

mov r0, r5

pop { r4-r6, lr}

bx lr

1. Consider the logic function with three inputs **A**, **B**, **C** and one output **Out**. **Out** should be 1 when exactly two inputs are 1.
   1. Draw the truth table for this function.

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | out |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |

* 1. Write the sum-of-products logic equation for this function.

The letter that are underlined are the negated ones

S =~~A~~BC + A~~B~~C + AB~~C~~

* 1. Minimize the logic equations

^ is the equivalent of excusive or

B(A ^ C) + ABC

* 1. Diagram

     Description automatically generatedDraw the circuit diagram for the logic equation.

1. Write a C function scale that takes a factor and multiplies each item in the array by the factor.

**extern void scale(double factor, double [] vec, int n);**

void scale(double factor, double [] vec, int n) {

for (int i = 0; i < n; i++) {

double \*add = &vec[i];

\*add = factor \* vec[i];

}

}

1. Make sure you understand the four areas of program memory; code, global data, stack, and heap and how memory is allocated for each.
2. Static function local variables in C are allocated in/on \_\_global data\_\_\_ memory.
3. Local variables in C are allocated in/on \_\_\_\_stack\_\_\_\_ memory.
4. Memory allocated using **malloc** is \_\_\_\_heap\_\_\_ memory.
5. What does the **-g** flag on the gcc compiler do?

It compiles it in a way that allows it to be ready for debugging.

1. What does the **-S** flag on the gcc compiler do?

This created a direct translation of your c program into arm assembly.

1. What does the **-o** flag on the gcc compiler do?

This compiles your program and creates an executable file that you can call and run.

1. What does the **-O3** flag on the gcc compiler do?

Optimizes your assembly code to the 3rd level of performance.

1. What does the **-c** flag on the gcc compiler do?

This is just the compile flag and checks to see if your program will compile.

1. What program do we use to reverse engineer machine code files?

We use obj dump -d to reverse engineer machine code files.

1. How many bytes is a C **double**?

8 bytes

1. Briefly describe what a *memory leak* is?

When a program over time uses more ram memory/ memory and just keeps growing, this is caused if the memory is never freed.

1. Consider the following C program. Why might it have a segmentation fault?

**#include <stdio.h>**

**int \*seven() {**

**int x = 7;**

**return &x;**

**}**

**int main() {**

**int \*y = seven();**

**printf("%d\n", \*y);**

**}**

Due to x not being static it is treated as a local variable and therefore cannot be accessed outside of the scope like the function call in main. The way that this is implemented gives x or 7 a temporary place in memory therefore it could be overwritten.

1. The following variation of the program seems to work OK. Why?

**#include <stdio.h>**

**int \*seven() {**

**static int x = 7;**

**return &x;**

**}**

**int main() {**

**int \*y = seven();**

**printf("%d\n", \*y);**

**}**

Since x is now a static variable it is now treated as if it were a global variable this means that it will not be overwritten therefore it will provide an actual value.

1. Write a function **rev** that takes an unsigned integer **x** and reverses the bits in **x**. Use bit operations only, don’t use strings or arrays.
   1. Modify the **add** function in **adder.c** we wrote to call **rev**.

#include "adder.h"

#include <stdlib.h>

//make a type for a bit

typedef u\_int32\_t bit\_t;

//reverse the bits in x

u\_int32\_t rev(u\_int32\_t x) {

u\_int32\_t reverse = 0;

while(x) {

reverse = reverse << 1;

reverse = reverse | x & 1;

x = x >> 1;

}

Return reverse;

}

u\_int32\_t add(u\_int32\_t x, u\_int32\_t y) {

u\_int32\_t s = 0; //running sum

bit\_t C = 0; //initial carry in is a zero

for (int i = 0; i < 32; i++) {

bit\_t X = x & 1;

bit\_t Y = y & 1;

bit\_t S = X ^ Y ^ C;

C = (X & Y) | (X & C) | (Y & C);

x = x >> 1;

y = y >> 1;

//get S into s

s = (s << 1) | S;

}

return rev(s);

}

1. There is a simple fix to the **add** function in **adder.c** file that does not need to reverse the bits. What is it?

u\_int32\_t add(u\_int32\_t x, u\_int32\_t y) {

u\_int32\_t s = 0; //running sum

bit\_t C = 0; //initial carry in is a zero

for (int i = 0; i < 32; i++) {

bit\_t X = x & 1;

bit\_t Y = y & 1;

bit\_t S = X ^ Y ^ C;

C = (X & Y) | (X & C) | (Y & C);

x = x >> 1;

y = y >> 1;

//get S into s

s = s | (S << i);

}

return s;

}