

Lab 2 - Group 5

We started this lab by having some trouble with the setup. Originally the problem was that we all used the same port. After we changed that to 5042 and 5082 we were ready to go. It was a lot of work just figuring out the syntax of ARM. That means a lot of searching and reading in the documentation.

Task 1:

```
.data                                @ Data section for declaring variables

string: .asciz "%d\n"               @ Variable string with \0

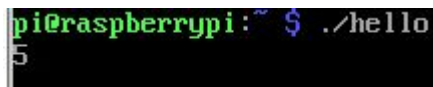
.global main                        @ Defining function main
.extern printf                      @ Setting up external function

main:
    mov r0, #2                     @ Load immediate value 2 into r0
    mov r2, #3                     @ Load immediate value 3 into r2
    add r1, r0, r2                 @ Add value in r0 with r2, store in r1
    ldr r0, =string                @ Load address of string into r0, r1 as arg
    bl printf                      @ Branch and pass r0 to printf
    bx lr                         @ Branch to link register
```

This code was compiled in the following way:

```
as hello.s -o hello.o
gcc hello.o -o hello
```

This created the executable hello. When run, the following result was produced:



```
pi@raspberrypi:~$ ./hello
5
```

2 + 3 = 5 makes the output correct.

Task 2:

Here we had some problems with linking them together but eventually we found out that we needed to make some changes when we compiled.

C-function that takes an integer argument and prints the value in hexadecimal notation

```
#include <stdio.h>
```

```
void int_out(int num)
{
    printf("%#x\n", num);
}
```

Main function for testing the function int_out

```
int main()
{
    int_out(5);
    return 0;
}
```

Produces output: 0x5

The int_out function could be compiled to an object file using the command:

```
gcc int_out.c -o int_out.o -c
```

Assembly code which loads an immediate value 4 into a register and calls int_out.

```
.global main      @ Defining function main
.extern int_out    @ Setting up external function

main:
    mov r0, #4      @ Load immediate value 4 into r0
    bl int_out      @ Call external function with arg r0
    bx lr          @ Branch to link register
```

The assembly program could be compiled to an object file using the command:

```
as hex.s -o hex.o
```

The two object files could then be linked using the command:

```
gcc hex.o int_out.o -o hex
```

Executing the file produces the output:

```
pi@raspberrypi:~$ ./hex
0x4
Segmentation fault
```

Bitshifting 0xBD5B7DDE

```
.data                                @ Data section for defining variables

number: .word 0xBD5B7DDE @ Stores the number in variable

.global main
```

```
.extern int_out

main:
    ldr r1, number      @ Stores number in r1
    asr r0, r1, #1      @ Arithmetic right shift by 1
    bl int_out           @ Call int_out
    bx lr               @ Branch to link register
```

Compiling and running produces the output:

```
pi@raspberrypi:~$ ./hex
0xdeadbeef
Segmentation fault
```

0xBD5B7DDE is 10111101010110110111110111011110 in binary.

0xDEADBEEF is 11011110101011011011111011101111 in binary.

While doing an arithmetic right shift, the sign extension replaces the new most significant bit with a copy of the old one. As can be seen in this example, All the numbers have been shifted right and the most significant bit has been replaced with a 1. The result is correct.

Using printf inside assembly

```
.data                                @ Data section for defining variables
number:
    .word 0xBD5B7DDE                @ Number stored in word

string:
    .asciz "%#x\n"                  @ String stored in ascii with \0 at end

.text                                @ Code section
.global main
.extern printf

main:
    ldr r0, =number                 @ Stores address of number in r0
    ldr r2, [r0]                    @ Stores value at address r0 in r2
    asr r1, r2, #1                  @ Arithmetic right shift of r2, store in r1
    ldr r0, =string                  @ Load address of string into r0, r1 as arg
    bl printf                        @ Call external function printf
    bx lr                           @ Branch to link register
```

Compiling and running produces the output:

```
pi@raspberrypi:~$ ./hex
0xdeadbeef
```

Which again is the correct output.

Task 3:

C code for calculating the XOR of two values a and b.

```
#include <stdio.h>

int xor(int a, int b)
{
    return a ^ b; // Bitwise XOR operator
}

int main()
{
    printf("%d\n", xor(0, 1));
    return 0;
}
```

Testing with different values matches the truth table for the XOR.
0 and 0 gives 0, 0 and 1 gives 1, 1 and 0 gives 1, 1 and 1 gives 0.

The function was rewritten in assembly

```
.global axor    @ Defines name of function

axor:
    eor r0, r1    @ XOR operation on operand r0 and r1
    bx lr        @ Branch to link register
```

The C-program was modified to use the assembly function

```
#include <stdio.h>

extern int axor(int, int);

int main()
{
    int i = 0;
    int j = 0;
    while(i < 4)
    {
        while(j < 4)
        {
            printf("%d\n", axor(i, j));
            j++;
        }
        j = 0;
        i++;
    }
}
```

```
}  
  
return 0;  
}
```

The parameters correspond to register r0 and r1. The operation is therefore made on these registers. The value is stored in r0 and returned to the C-program.

After compiling and running, the following result was produced:

```
pi@raspberrypi:~$ gcc xor.c axor.s -o xor  
pi@raspberrypi:~$ ./xor  
0 0 xor 0 = 0  
1 00 xor 01 = 01  
2 00 xor 10 = 10  
3 00 xor 11 = 11  
1 01 xor 00 = 01  
0 01 xor 01 = 00  
3 01 xor 10 = 11  
2 01 xor 11 = 10  
2 10 xor 00 = 10  
3 10 xor 01 = 11  
0 10 xor 10 = 0  
1 10 xor 11 = 01  
3 11 xor 00 = 11  
2 11 xor 01 = 10  
1 11 xor 10 = 01  
0 11 xor 11 = 00  
pi@raspberrypi:~$
```