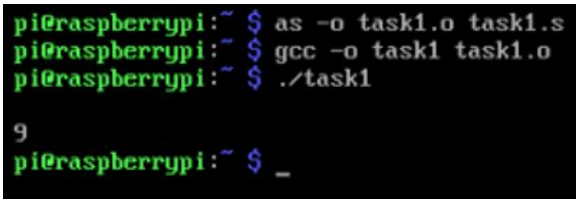
Kinfai Chin & Zacharias Jöbo

Lab 2: ARM Assembly Language

Task 1: Basic Setup (5 points)

The first task was to write two immediate integer values into the registers. Those two values should be added together and be displayed via the printf function.

Syntax:	Result:
.data /* This part sets the string to show a integer */ string: .asciz "\n%d\n" .text .global main .extern printf main: push {ip, lr} mov r2, #1 @sets the value 1 to r2 mov r3, #8 @sets the value 8 to r3 add r1, r2 ,r3 /*adds r2 and r3 and stores it in r1*/ ldr r0, = string /* loads the string into r0 */ bl printf /* the printf takes the string stored in r0 and uses the value in the next registry (r1)*/ pop {ip, pc}	

Task 2: Shifting Integers (10 points)

The second task was to work with integers. The work started with creating an C function “int_out” that takes an integer argument and prints out the value in hexadecimal form and then write an main that tests the function.

Syntax:	Result:

`#include <stdio.h>`
 	
`void int_out(int x){`
`  printf(“\n %x \n”) /*Prints out the argument x in hexform*/`
 	
`  return;`
`}`
 	
`int main(){`
`  int_out(1);`
`  int_out(10);`
`  int_out(16);`
`  return 0;`
`}`
	

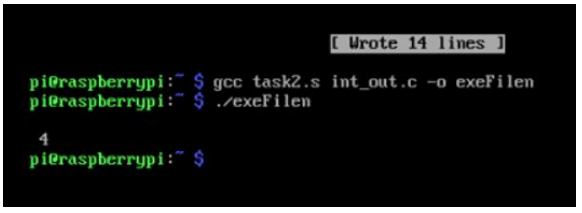
The next part was to write an assembly program which loads the immediate value of 4 and then calls the external function int_out to print it in a hexadecimal form. The assemby code was written first and the the following code was written into raspberry pi command prompt.

```
gcc task2.s int_out.c -o exeFilen
```

This creates an executable file called exeFilen that links the assembly code task2.s and the external functions from int_out.c

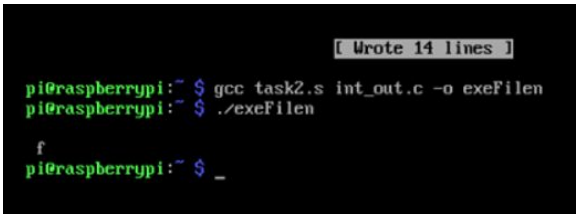
Kinfai Chin & Zacharias Jöbo

Syntax:	Result:

`.global main`
`.extern int_out @declaration of external function`
 	
`main:`
 	
`push(ip,ir)`
`mov r0,#15  /*Puts the value 15 into register 0*/`
 	
`bl int_out  /*Calls the external function to print out*/`
 	
`pop (ip,pc)`
	

The output was like expected but since 4 is the same in hexadecimal form as well another number was tested to confirm that the function was giving the correct output.

Syntax:	Result:

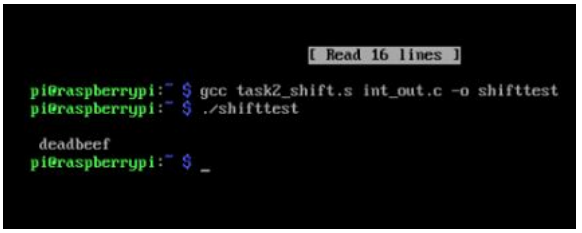
`.global main`
`.extern int_out`
 	
`main:`
 	
`push(ip,ir)`
`mov r0,#15  /*Puts the value 15 into register 0*/`
 	
`bl int_out  /*Calls the external function to print out*/`
 	
`pop (ip,pc)`
	

Next part was to load the integer 0xBD5B7DDE and verify that sign extension works as expected when bit shifting to the right.

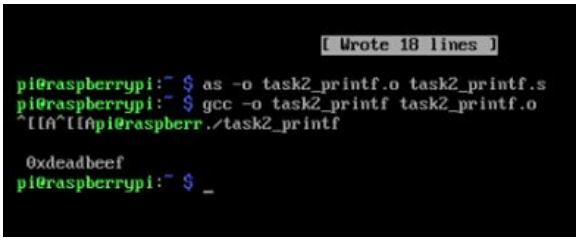
Kinfai Chin & Zacharias Jöbo

Syntax:	Result:

```
.data
number: .int 0xBD5B7DDE
.global main
.extern int_out

main:
push(ip,ir)

ldr r0, =number /*Assigns the
adress of number into register 0.*/

ldr r0, [r0] @
/* Assigns the value of number into
register 0 */

asr r0, #1 /*Uses asr to bit shift
1 step to the right*/

bl int_out /*Calls the external
function to print out*/

pop (ip,pc)
```



Now the code gets modified to use printf instead of the custom function “int_out”

Syntax:	Result:
.data number: .int 0xBD5B7DDE string: .asciz “\n %#010x \n” .global main .extern printf main: push(ip,ir) ldr r0, =number /*Assigns the address of number into register 0.*/ ldr r0, [r0] /* Assigns the value stored in number into register 0 */ asr r0, #1 /*Uses asr to bit shift 1 step to the right*/ ldr r0, =string /*Sets register 0 to the string defined above*/ bl printf pop (ip,pc)	

Kinfai Chin & Zacharias Jöbo

Task 3: Assembly in C (10 points)

The first two parts in task 3 was to write C code for xor bit-wise and test it.

Syntax (C-implemenation part and test):

```
int xor(int a, int b){
/* In C there's a operator for
bitwise xor operation which is ^ */
     a = a^b;
return a;
}

int main(){
/*Testing the function*/
     int a = 15;
     int b = 0;
     int x = xor(a,b);
     printf("\nOutput = %d\n", x);
return 0;
}
```

Result:

```
pi@raspberrypi:~ $ gcc -o xor xor.c
pi@raspberrypi:~ $ ./xor

Output = 15
pi@raspberrypi:~ $
```

Table below shows 15 in binary and 0 in binary. The result shows the after a bitwise xor.

15	1	1	1	1
0	0	0	0	0
Result:	1	1	1	1

The output 15 is correct.

Syntax (Testing the C-function in assembly):

```
.text
.global main
.extern xor

main:

push {ip,lr}

mov r0, #13 @ sets the value 13 in r0
move r1, #9 @ set the value 9 in r1
bl xor /*calls the function xor from
the previous C-implementation*/
pop {ip, pc}
```

Result:

```
pi@raspberrypi:~ $ gcc task3_xor.s xor.c -o xorex
pi@raspberrypi:~ $ ./xorexe

 4
pi@raspberrypi:~ $ _
```

13	1	1	0	1
9	1	0	0	1
Result:	0	1	0	0

The output 4 is correct.

The last two parts was to write an assembly version of the xor bit-wise.

Syntax (xor function in assembly):

```
.global axor
axor:
eor r0, r0 ,r1 /* eor is assembly xor bitwise operator, stores the
result in r0 */
mov pc, lr
```

Syntax:

```
#include <stdio.h>
/* takes in the external function
xor, that was made previously in
assembly*/
extern int axor(int a, int b);

int main (){
     int x = axor(0xf, 0x0);
/*the parameters is 15 and 0 in
hexadecimals*/
     printf = (“\n %x \n”, x);
     return 0;
}
```

Result:

```
pi@raspberrypi:~ $ gcc axor.s axor.c -o axorexe
pi@raspberrypi:~ $ ./axorexe

 f
pi@raspberrypi:~ $ _
```

15	1	1	1	1
0	0	0	0	0
Result:	1	1	1	1

The output F is correct.