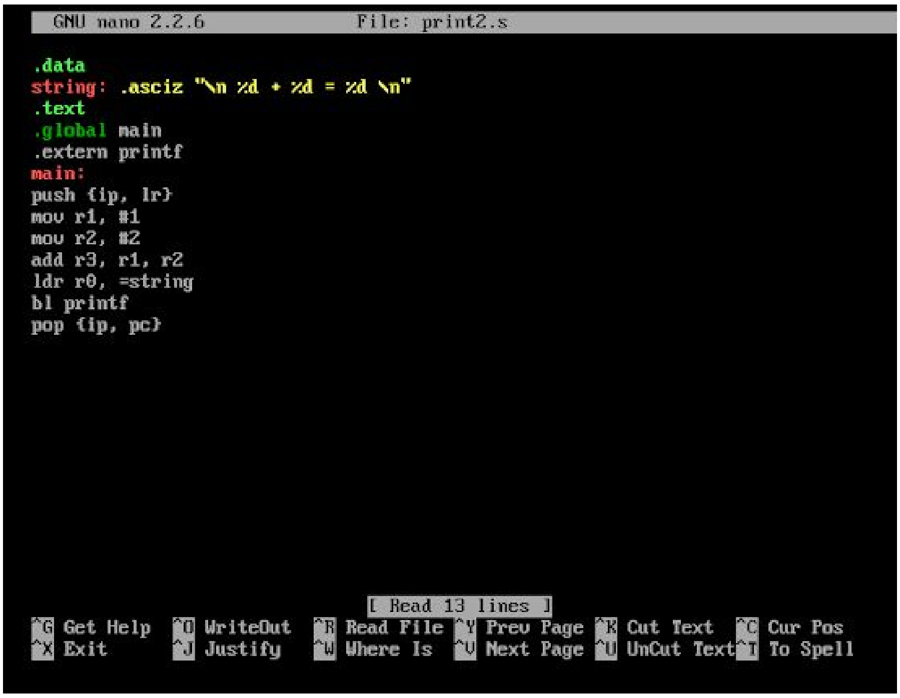
Lab2

Vide Hopper & Martin Karlsson

Task 1

We began this lab with starting up a virtual machine with an Rasbian as operating system. The OS was configured to emulate an ARM processor found on the original Raspberry Pi. We used qemu as our emulator. The next step was to create and edit a s-file, which we used nano for. Then we proceded to load two integers to register 1 and 2. We then used *add* to sum the integers and to store the result in register 3.



Code 1, task 1 (print 2 integers)

By following the example given in the description of the lab we could figure out how to set up the external c function; *printf*. We used a string in the data field to format the output of *printf*. The string was then loaded in to register 0 which is the register used when printing. The following registers will take place of correspondent *%d*. This is equivalent to the same call but in C *printf("\n %d + %d = %d , r1,r2,r3 \n").*

To run the program you first need to assemble the file using as

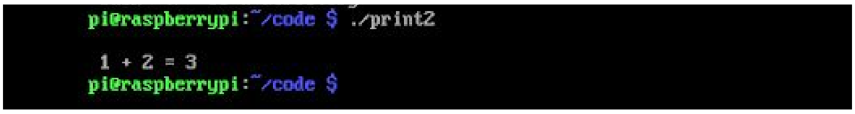
as -o file.o file.s

and then use gcc to make it executable.

gcc -o nameofprogram file.o

To run the executable you need to write

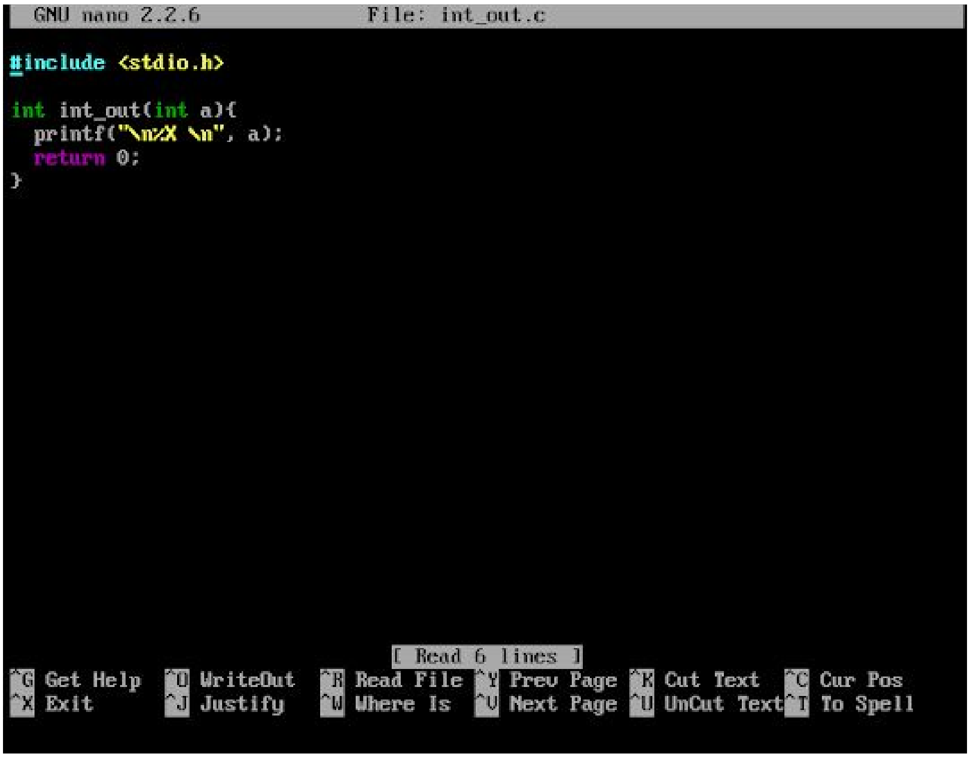
./nameofprogram



Output 1, task 1 (print 2 integers)

Task 2 Shifting integers

The first part of this task was to create a file c function *int_out* converted an integer to hexadecimal and printed the result. We created a c-script that called the function to verify that we got the the right output from the function. The C-code also needed to be compiled using gcc.



Code 2, The int_out function

The next step was to create an assembly program which loads the value 4 and then uses the function *int_out* to convert and print the result in hexadecimal. In the assembly code we had to include the external function *int_out* then loaded the value 4 into register 0 and called the external funcion.

```
  GNU nano 2.2.6                    File: ez.s

.data
string: .asciz "\n %x \n"

.text
.global main
.extern int_out

main:
push {ip, lr}
mov r0, #4
bl int_out
pop {ip, pc}

                               [ Read 13 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Code 3, calling our int_out function

To link the functions together we needed to use gcc with both of the .o-files included as seen in *Output 2*.

```
pi@raspberrypi:~/code $ gcc -o namn ez.o int_out.o
pi@raspberrypi:~/code $ ./namn

 4
pi@raspberrypi:~/code $ _
```

Output 2, the output from our int_out function

As can be seen on *Output 2* our output was correct, the value 4 in decimal is 4 in hexadecimal.

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Tabel 1, translation table

The next step was to right shift one bit using a loaded number. To load the number we create a variable *num*. The integer was loaded in register 0, we used *asrs* to right bit shift the integer and then used the external function *int_out* to print the result.

The value

0xBD5B7DDE

is

0b1011 1101 0101 1011 0111 1101 1101 1110

in binary.

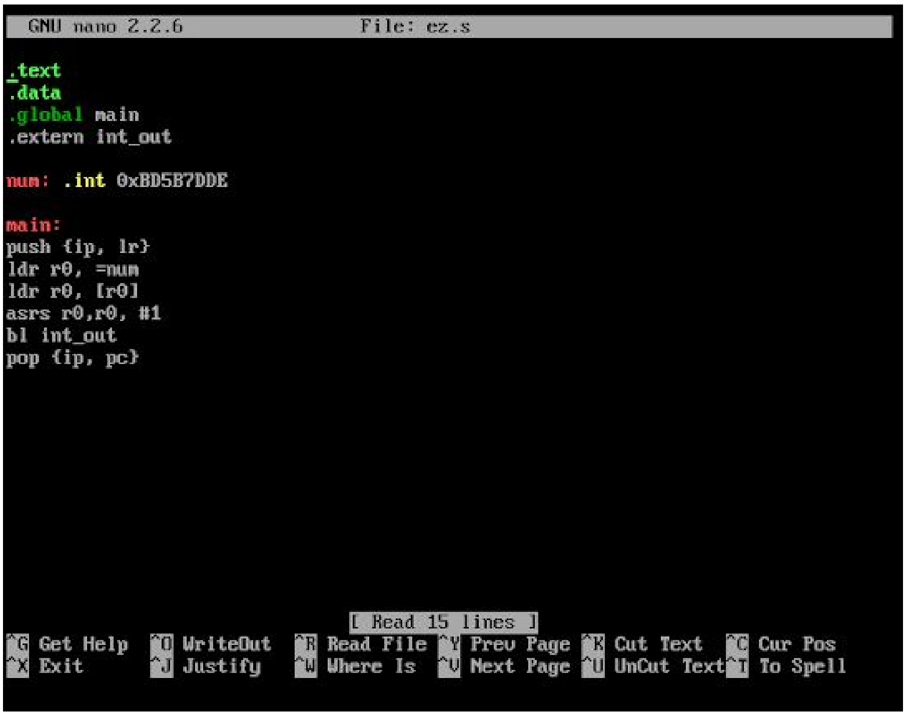
If we do a right bit shift on the binary number above we get

0b1101 1110 1010 1101 1011 1110 1110 1111

assuming that the most significant bit is a signed bit.

By quickly looking at *Tabel 1* you can easily translate the shifted bits to hexadecimal which translates to

0xDEADBEEF



Code 4, ASRS with int_out

Here we see the binary number, 0b1101 1110 1010 1101 1011 1110 1110 1111 in hexadecimal

```
pi@raspberrypi:~/code $ ./asd

DEADBEEF
pi@raspberrypi:~/code $ _
```

Output 3, deadbeef using int_out

The next step was to use *printf* instead of our own function. The string was loaded into register 0 and the *num* variable to register 1.

```
  GNU nano 2.2.6                File: ez.s

.data
string: .asciz "\n %x \n"
num: .int 0xBD5B7DDE

.text
.global main
.extern printf

main:
push {ip, lr}
ldr r0, =string
ldr r1, =num
ldr r1, [r1]
asrs r1, r1, #1
mov r1, r1
bl printf
pop {ip, pc}

                         [ Read 18 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Code 5, ASRS with printf

The difference is our function uses uppercase with the %X and the string provided in assembly uses lowercase, %x.

```
pi@raspberrypi:~/code $ ./asdf

 deadbeef
pi@raspberrypi:~/code $
```

Output 4, deadbeef using printf

Conclusion

For task one it’s easy to verify that the number printed from our function is the correct representation in hexadecimal. This can be seen in the *Tabel 1* which is a translation table.

When a right bit shift is performed on the given binary

0b1011 1101 0101 1011 0111 1101 1101 1110

it can give two different outputs depending on if the most significant bit is a sign bit or not. If we just did a right bit shift without keeping track of the signed bit then the shift result in

0b01011 1101 0101 1011 0111 1101 1101 111

which is

0x5EADBEEF

in hexadecimal. This is what most online calculator results in when performing this operation which caused a lot of confusion. In the assignment it clearly state that the given number is a signed number. This means that we need to keep track of the sign bit which is the most significant bit. When bit shifting is performed and we keep the sign the same we get

0b1101 1110 1010 1101 1011 1110 1110 1111

which is

0xDEADBEEF

this indicates that our output is correct.