Laboratory 9

Title of the Laboratory Exercise: Inline assembly language programs for code optimisation

1. Introduction and Purpose of Experiment

   Students will create C programs with inline assembly code for code optimisation

2. Aim and Objectives

   Aim

   To develop inline assembly language program for code optimisation

   Objectives

   At the end of this lab, the student will be able to

   - Identify inline assembly language calls
   - Explain optimization of program by exploiting architectural features in target computer
   - Create C programs with inline assembly code

3. Experimental Procedure

   1. Write algorithm to solve the given problem
   2. Translate the algorithm to assembly language code
   3. Run the assembly code in GNU assembler
   4. Create a laboratory report documenting the work

4. Questions:

   Develop a C program without inline assembly instructions and find out the code size and memory used for the program. Develop the C program with inline assembly instructions and find out the code size and memory used for the program. Compare the results.

5. Calculations/Computations/Algorithms

```c
#include <stdio.h>
#define di(i) printf("\nDEBUG--#"#i " : %d#\n", i);

/* a = b + (c * d) */
int a, b = 1, c = 2, d = 3;

int main() {
    di(b); di(c); di(d);
    printf("\nBefore Operation:\n");
    di(a);
    __asm__("movl c, %eax \n\t"
            "movl d, %edx \n\t"
            "mull %edx \n\t"
            "add b, %eax \n\t"
            "movl %eax,a \n\t");
    printf("\nAfter Operation:\n");
    di(a);
    return 0;
}
```

Figure 0-1 Inline Assembly C Code

```c
#include <stdio.h>
#define di(i) printf("\nDEBUG--#"#i " : %d#\n", i);

/* a = b + (c * d) */
int a, b = 1, c = 2, d = 3;

int main() {
    di(b); di(c); di(d);
    printf("\nBefore Operation:\n");
    di(a);
    a = b + (c * d);
    printf("\nAfter Operation:\n");
    di(a);
    return 0;
}
```

Figure 0-2 Native C Code

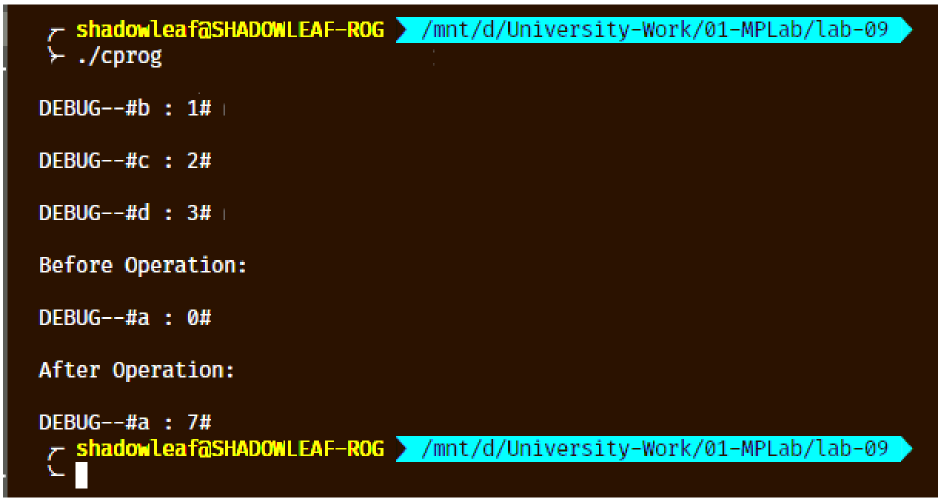
```
	.file	"assem.c"
	.comm	a,4,4
	.globl	b
	.data
	.align 4
	.type	b, @object
	.size	b, 4
b:
	.long	1
	.globl	c
	.align 4
	.type	c, @object
	.size	c, 4
c:
	.long	2
	.globl	d
	.align 4
	.type	d, @object
	.size	d, 4
d:
	.long	3
	.section	.rodata
.LC0:
	.string "\nDEBUG--#b : %d#\n"
.LC1:
	.string "\nDEBUG--#c : %d#\n"
.LC2:
	.string "\nDEBUG--#d : %d#\n"
.LC3:
	.string "\nBefore Operation:"
.LC4:
	.string "\nDEBUG--#a : %d#\n"
.LC5:
	.string "\nAfter Operation:"
	.text
	.globl	main
	.type	main, @function
main:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	b(%rip), %eax
	movl	%eax, %esi
	movl	$.LC0, %edi
	movl	$0, %eax
	call	printf
	movl	c(%rip), %eax
	movl	%eax, %esi
	movl	$.LC1, %edi
	movl	$0, %eax
	call	printf
	movl	d(%rip), %eax
	movl	%eax, %esi
	movl	$.LC2, %edi
	movl	$0, %eax
	call	printf
	movl	$.LC3, %edi
	call	puts
	movl	a(%rip), %eax
	movl	%eax, %esi
	movl	$.LC4, %edi
	movl	$0, %eax
	call	printf
#APP
# 11 "assem.c" 1
	movl c, %eax
	movl d, %edx
	mull %edx
	add b, %eax
	movl %eax,a

# 0 "" 2
#NO_APP
	movl	$.LC5, %edi
	call	puts
	movl	a(%rip), %eax
	movl	%eax, %esi
	movl	$.LC4, %edi
	movl	$0, %eax
	call	printf
	movl	$0, %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	main, .-main
	.ident	"GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609"
	.section	.note.GNU-stack,"",@progbits

```

Figure 0-3 Inline Assembly assembly code

```
	.file	"cprog.c"
	.comm	a,4,4
	.globl	b
	.data
	.align 4
	.type	b, @object
	.size	b, 4
b:
	.long	1
	.globl	c
	.align 4
	.type	c, @object
	.size	c, 4
c:
	.long	2
	.globl	d
	.align 4
	.type	d, @object
	.size	d, 4
d:
	.long	3
	.section	.rodata
.LC0:
	.string "\nDEBUG--#b : %d#\n"
.LC1:
	.string "\nDEBUG--#c : %d#\n"
.LC2:
	.string "\nDEBUG--#d : %d#\n"
.LC3:
	.string "\nBefore Operation:"
.LC4:
	.string "\nDEBUG--#a : %d#\n"
.LC5:
	.string "\nAfter Operation:"
	.text
	.globl	main
	.type	main, @function
main:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	b(%rip), %eax
	movl	%eax, %esi
	movl	$.LC0, %edi
	movl	$0, %eax
	call	printf
	movl	c(%rip), %eax
	movl	%eax, %esi
	movl	$.LC1, %edi
	movl	$0, %eax
	call	printf
	movl	d(%rip), %eax
	movl	%eax, %esi
	movl	$.LC2, %edi
	movl	$0, %eax
	call	printf
	movl	$.LC3, %edi
	call	puts
	movl	a(%rip), %eax
	movl	%eax, %esi
	movl	$.LC4, %edi
	movl	$0, %eax
	call	printf
	movl	c(%rip), %edx
	movl	d(%rip), %eax
	imull	%eax, %edx
	movl	b(%rip), %eax
	addl	%edx, %eax
	movl	%eax, a(%rip)
	movl	$.LC5, %edi
	call	puts
	movl	a(%rip), %eax
	movl	%eax, %esi
	movl	$.LC4, %edi
	movl	$0, %eax
	call	printf
	movl	$0, %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	main, .-main
	.ident	"GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609"
	.section	.note.GNU-stack,"",@progbits
```

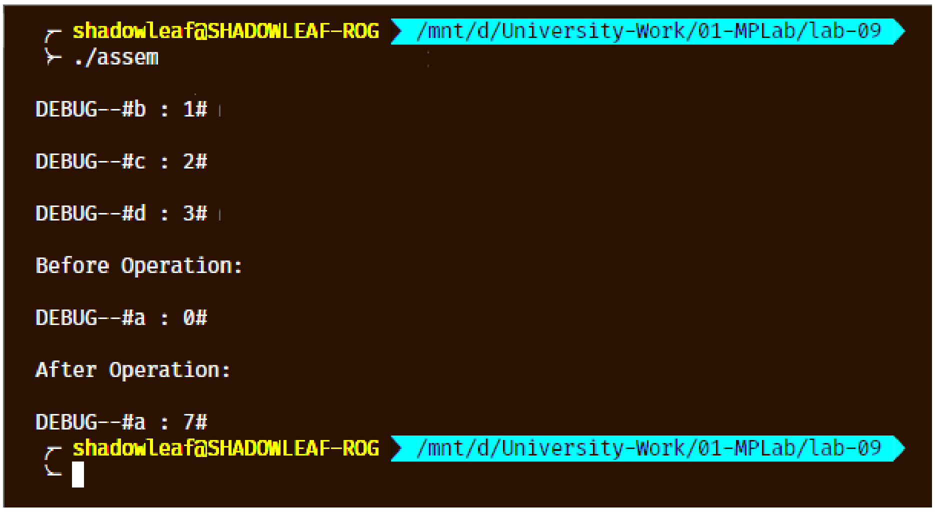
Figure 0-4 Native C, assembly code

6. Presentation of Results



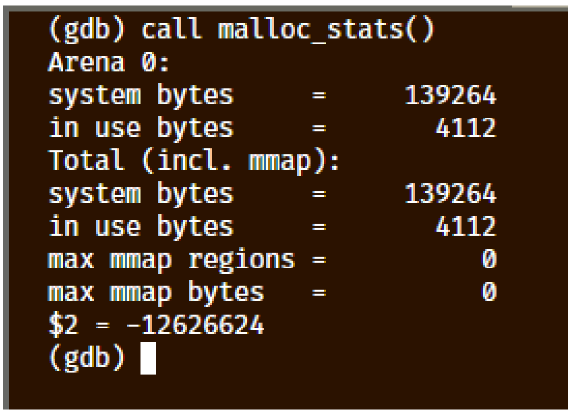
```
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/01-MPLab/lab-09
./cprog

DEBUG--#b : 1#

DEBUG--#c : 2#

DEBUG--#d : 3#

Before Operation:

DEBUG--#a : 0#

After Operation:

DEBUG--#a : 7#
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/01-MPLab/lab-09
```

Figure 0-5 Output of Native C Code



```
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/01-MPLab/lab-09
./assem

DEBUG--#b : 1#

DEBUG--#c : 2#

DEBUG--#d : 3#

Before Operation:

DEBUG--#a : 0#

After Operation:

DEBUG--#a : 7#
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/01-MPLab/lab-09
```

Figure 0-6 Output of Inline Assembly



```
(gdb) call malloc_stats()
Arena 0:
system bytes     =     139264
in use bytes     =       4112
Total (incl. mmap):
system bytes     =     139264
in use bytes     =       4112
max mmap regions =          0
max mmap bytes   =          0
$2 = -12626624
(gdb)
```

Figure 0-7 Native C Code Memory Usage

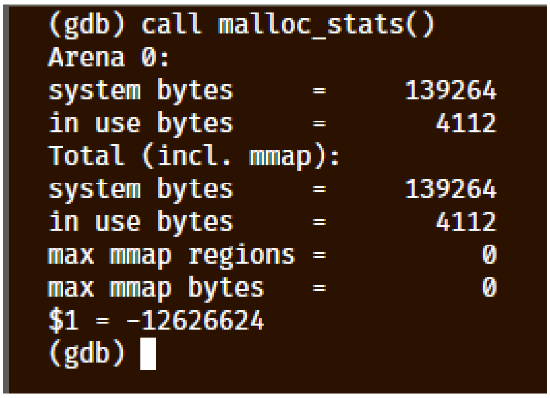


Figure 0-8 Inline Assembly Memory Usage

7. Analysis and Discussions

The format of basic inline assembly is very much straight forward. Its basic form is

```
asm("assembly code");
```

Example.

```
asm("movl %ecx %eax"); /* moves the contents of ecx to eax */

__asm__("movb %bh (%eax)"); /*moves the byte from bh to the memory pointed by eax
*/
```

We can use `__asm__` if the keyword `asm` conflicts with something in our program. If we have more than one instructions, we write one per line in double quotes, and also suffix a '`\n`' and '`\t`' to the instruction. This is because gcc sends each instruction as a string to as(GAS) and by using the newline/tab we send correctly formatted lines to the assembler.

If in our code we touch (ie, change the contents) some registers and return from asm without fixing those changes, something bad is going to happen. This is because GCC have no idea about the changes in the register contents and this leads us to trouble, especially when compiler makes some optimizations. It will suppose that some register contains the value of some variable that we might have changed without informing GCC, and it continues like nothing happened. What we can do is either use those instructions having no side effects or fix things when we quit or wait for something to crash. This is where we want some extended functionality. Extended asm provides us with that functionality.

8. Conclusions

Inline assembly can be used in C programs to write in low level language and have direct access to the CPU registers, this gives more control over the register's memory.

If we compare the code size of the trans-piled assembly code from the c source code, we find that the inline assembly code takes a greater number of lines in assembly than the native c code trans-piled to assembly code.

The memory usage of both the programs are similar with little to no differences in memory usage.

9. Comments

1. Limitations of Experiments

The Experiment is limited to a very simple C program, hence concluding the results in difficult for the same.

2. Limitations of Results

Since the operations performed were very simple, there is little to no differences in the two codes, the inline assembly and native c code, although for complex codes, the results might differ.

3. Learning happened

We learnt how to disassemble a C code into its assembly code using `gcc -S filename.c` command.

4. Recommendations

To have a better comparison between the two types of codes, take a larger and complex c code and trans-pile it to its assembly code.

Signature and date

Marks