Programming Assignment #3

Question 1

Step 1. Write a program in GNU assembly language that uses macros to print the following messages on the screen [20 marks]:

> Hello, programmers!
> Welcome to the world of,
> Linux assembly programming!

Before I show the assembly language, I just want to state that we are not really taught how to do what this question asks. In addition, trying to piece together the information with the various different sources such as the textbook, the powerpoint notes, the disassembler programs themselves, and online resources, is quite frustrating as there are always a variety of differenes between them.

For example, **copying and pasting the program in the module 6 notes into a vi file on the linux server supplied by the school and executing the commands as recommended gives an error** saying there is no main method*. Whether this is just a standard that is no longer used or not, I don't know, but assembly code derived from the gcc compiler on the linux machine supplied by the school does not use _start anywhere. This kind of surefire confusion is frustrating and leaves a student with more questions than answers.

*

```
[musfiqcomp213136@cs2 kuby]$ as Hello.s
Hello.s: Assembler messages:
Hello.s:3: Error: no such instruction: `_main'
[musfiqcomp213136@cs2 kuby]$
```

See the following page for my answer to the question.

C code:

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

#define msg1 printf("Hello, programmers!\n")
#define msg2 printf("Welcome to the world of,\n")
#define msg3 printf("Linux assembly programming!\n")

int main()
{
    msg1;
    msg2;
    msg3;
    return 0;
}
```

Assembly code derived from the GCC compiler on the supplied linux machine:

```
	.file	"question1.c"
	.section	.rodata.str1.1,"aMS",@progbits,1
.LC0:
	.string "Hello, programmers!"
.LC1:
	.string "Welcome to the world of,"
.LC2:
	.string "Linux assembly programming!"
	.text
.globl main
	.type	main, @function
main:
.LFB18:
	.cfi_startproc
	subq	$8, %rsp
	.cfi_def_cfa_offset 16
	movl	$.LC0, %edi
	call	puts
	movl	$.LC1, %edi
	call	puts
	movl	$.LC2, %edi
	call	puts
	movl	$0, %eax
	addq	$8, %rsp
	.cfi_def_cfa_offset 8
	ret
	.cfi_endproc
.LFE18:
	.size	main, .-main
	.ident	"GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-23)"
	.section	.note.GNU-stack,"",@progbits
```

Here we clearly see the read only data section, housing the macros, as well as the .text section that contains the main method. Main simply creates a bit of space on the run time stack, loads each macro address into %edi and calls puts, which sequentially prints the macros to screen. Finally, the return value is set to zero and %rsp incremented before execution finishes via ret.

Question 2

a) Write a version of the function using a for loop

```
long forsum (long start, long finish)
{
    long acc = start;
    long i;

    for (i = start+1; i <= finish; i++)
    {
        acc += i;
    }
    return acc;
}
```

b) Write a version of the function using a while loop

```
long whilesum (long start, long finish)
{
    long acc = start;
    long i = start+1;

    while (i <= finish)
    {
        acc += i;
        i++;
    }
    return acc;
}
```

c) Write a version of the function using a do loop

```
long dosum (long start, long finish)
{
    long acc = start;
    long i = start+1;

    do
    {
        acc += i;
        i++;
    } while (i <= finish);
    return acc;
}
```

d) Write a version of the function using a goto loop

```c
long gotosum (long start, long finish)
{
    long acc = start;
    long i = start +1;

    loop:
        acc += i;
        i++;
        if (i <= finish)
            goto loop;
    return acc;
}
```

e) Is the assembly language version of each loop function the same or different? If different, identify the differences. Your comparison should be based on:

- Number of registers used
- Number of jumps (iterations)
- Total number of operations

The following is the assembly code for the for loop and the while loop.

```
3   _forsum:
4   LFB4:
5       movq    %rdi, %rax      ; long acc = start                  note: acc in %rax and start in %rdi
6       leaq    1(%rdi), %rdx   ; long i = start + 1                note: i in %rdx
7       cmpq    %rsi, %rdx      ; compare i to finish --> guard     note: finish in %rsi
8       jg  L1                  ; guarded do                        if i > finish, return.
9       addq    $1, %rsi        ; finish++ (??)
10  L3:
11      addq    %rdx, %rax      ; acc += i
12      addq    $1, %rdx        ; i++
13      cmpq    %rsi, %rdx      ; compare i to finish
14      jne L3                  ; if finish != i, jump to L3        !! whilesum and forsum are exactly the same.
15  L1:
16      ret
```

```
31  _whilesum:
32  LFB6:
33      movq    %rdi, %rax      ; long acc = start                  note: acc in %rax and start in %rdi
34      leaq    1(%rdi), %rdx   ; long i = start + 1                note: i in %rdx
35      cmpq    %rsi, %rdx      ; compare i to finish --> guard     note: finish in %rsi
36      jg  L9                  ; guard                             if i > finish, return.
37      addq    $1, %rsi        ; finish++ (??)
38  L11:
39      addq    %rdx, %rax      ; acc += i
40      addq    $1, %rdx        ; i++
41      cmpq    %rsi, %rdx      ; compare i to finish
42      jne L11                 ; if finish != i, jump to L3        !! whilesum and forsum are exactly the same.
43  L9:
44      ret
```

We see here that the assembly code of the for loop and the while loop are identical. Both use exactly four registers and store: acc in%rax; start in %rdi; finish in %rsi; i in %rdx. Both loops have a guard (on lines 8 and 36, respectively) with conditional jump instructions that cause the loop to iterate zero times if start > finish.

In the event that the guard test fails and the loop does iterate, the compiler makes a slight tweak to the C code by incrementing finish (lines 9 and 37), then uses a do… while loop with a test that queries whether i != finish. This is similar to iterating while i < finish in C code, which is functionally the same as how the code was written since finish has been incremented.

The total number of instructions executed are 5 if start > finish, and a minimum of 10 if start <= finish. Each iteration requires 4 instructions, so the number of instructions executed can by expressed as

x = 5, if start > finish
x = 6 + 4(n), where n = number of loop iterations, if start <= finish.

The following is the assembly code for the do loop and the goto loop.

```
LFE4:
    .globl _dosum
_dosum:
LFB5:
    movq    %rdi, %rax      ; long acc = start                    note: acc in %rax and start in %rdi
    leaq    1(%rdi), %rdx   ; long i = start + 1                  note: i in %rdx
L7:
    addq    %rdx, %rax      ; acc += i
    addq    $1, %rdx        ; i++
    cmpq    %rsi, %rdx      ; compare i to finish
    jle L7                  ; if i <= finish, jump to L7          !! goto1sum and dosum are identical
    ret
```

```
LFE6:
    .globl _goto1sum
_goto1sum:
LFB7:
    movq    %rdi, %rax      ; long acc = start                    note: acc in %rax and start in %rdi
    leaq    1(%rdi), %rdx   ; long i = start + 1                  note: i in %rdx
L15:
    addq    %rdx, %rax      ; acc += i
    addq    $1, %rdx        ; i++
    cmpq    %rsi, %rdx      ; compare i to finish
    jle L15                 ; if i <= finish, jump to L15         !! goto1sum and dosum are identical
    ret
```

We see in these two cases that the assembly code of the do loop and the goto loop are also identical. Both use exactly four registers and store: acc in %rax; start in %rdi; i in %rdx, and finish in %rsi.

Neither loops have a guard and so both loops are guaranteed to iterate at least one time. As such, functionally these two procedures will give different answers from the for and while loops for instances where start >= finish. For example, if start = 5 and finish = 0, both the do and goto loops written here will return a value of 11, while the for and while loops return an answer of 5. It is fair to say that this does not represent the intention of the programmer in this sense.

With respect to the actual assembly code, the compiler very faithfully represents the C code in both cases, with the instructions being near identical representations of what was written in C.

Both of these procedures execute a minimum of 7 instructions due to the fact that they guarantee at least a single execution. Each additional loop iteration requires another 4 instructions.

The number of instructions executed can be expressed via

$$x = 3 + 4(n), \text{ where } n = \text{number of loop iterations and } n >= 1$$

It is worth noting that the goto loop has various implementations, and it just so happened that the version I wrote mirrored the do loop. The use of goto code gives the programmer utmost control, and so it would be possible to write a goto loop that mirrors the assembly code of the for and while loops if so desired (by including a guard with a goto label that returns the function prior to iterating, for example).

Question 3

Using the C Programming language, write a program that sums an array of 50 elements. Next, optimize the code using loop unrolling. Loop unrolling is a program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration. Generate a graph of performance improvement. Tip: Figure 5.17 in the textbook provides an example of a graph depicting performance improvements associated with loop unrolling. [30 marks]

Original program sumarray:

```c
int sumarray(int input[AMOUNT])
{
    int i;
    int acc = 0;
    for (i = 0; i < AMOUNT; i++)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc += input[i];
    }
    return acc;
}
```

sumarray1 which uses 2 x 1 unrolling:

```c
int sumarray1(int input[AMOUNT])
{
    int i;
    int acc = 0;
    int limit = AMOUNT-1;

    for (i = 0; i < limit; i+=2)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc = (acc + input[i]) + input[i+1];
    }

    for (; i < AMOUNT; i++)
        acc += input[i];

    return acc;
}
```

sumarray2 which uses 2 x 1 unrolling with re-association:

```
int sumarray2(int input[AMOUNT])
{
    int i;
    int acc = 0;
    int limit = AMOUNT-1;

    for (i = 0; i < limit; i+=2)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc += (input[i] + input[i+1]);
    }

    for (; i < AMOUNT; i++)
        acc += input[i];

    return acc;
}
```

sumarray3 which uses 5 x 1 unrolling and re-association:

```
int sumarray3(int input[AMOUNT])
{
    int i;
    int acc = 0;
    int limit = AMOUNT-4;

    for (i = 0; i < limit; i+=5)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc += (input[i] + input[i+1]);
        acc += (input[i+2] + input[i+3]);
        acc += input[i+4];
    }

    for (; i < AMOUNT; i++)
        acc += input[i];

    return acc;
}
```

sumarray4 which uses 5 x 5 unrolling:

```c
int sumarray4(int input[AMOUNT])
{
    int i;
    int acc0 = 0;
    int acc1 = 0;
    int acc2 = 0;
    int acc3 = 0;
    int acc4 = 0;
    int limit = AMOUNT-4;

    for (i = 0; i < limit; i+=5)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc0 += input[i];
        acc1 += input[i+1];
        acc2 += input[i+2];
        acc3 += input[i+3];
        acc4 += input[i+4];
    }

    for (; i < AMOUNT; i++)
        acc0 += input[i];

    return acc0 + acc1 + acc2 + acc3 + acc4;
}
```

sumarray5 which uses 10 x 10 unrolling:

```
int sumarray5(int input[AMOUNT])
{
    int i;
    int acc0 = 0;
    int acc1 = 0;
    int acc2 = 0;
    int acc3 = 0;
    int acc4 = 0;
    int acc5 = 0;
    int acc6 = 0;
    int acc7 = 0;
    int acc8 = 0;
    int acc9 = 0;
    int limit = AMOUNT-4;

    for (i = 0; i < limit; i+=10)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc0 += input[i];
        acc1 += input[i+1];
        acc2 += input[i+2];
        acc3 += input[i+3];
        acc4 += input[i+4];
        acc5 += input[i+5];
        acc6 += input[i+6];
        acc7 += input[i+7];
        acc8 += input[i+8];
        acc9 += input[i+9];
    }

    for (; i < AMOUNT; i++)
        acc0 += input[i];

    return (acc0 + acc1 + acc2 + acc3 + acc4 + acc5 + acc6 + acc7 + acc8 + acc9);
}
```

main:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 1000
#define AMOUNT 50
#define DELAY 0xfffffff
 int main ()
 {
     int array[AMOUNT];
     int i;
     srand(time(NULL));
     for (i = 0; i < AMOUNT; i++)
         array[i] = rand() % MAX; // use remainder operator to limit the size.

     int sum = sumarray(array);
     printf("The sum of the values of the array is %d\n", sum);

     int sum1 = sumarray1(array);
     printf("The sum of the values of the array is %d\n", sum1);

     int sum2 = sumarray2(array);
     printf("The sum of the values of the array is %d\n", sum2);

     int sum3 = sumarray3(array);
     printf("The sum of the values of the array is %d\n", sum3);

     int sum4 = sumarray4(array);
     printf("The sum of the values of the array is %d\n", sum4);

     int sum5 = sumarray5(array);
     printf("The sum of the values of the array is %d\n", sum5);

     exit (0);
 }
```

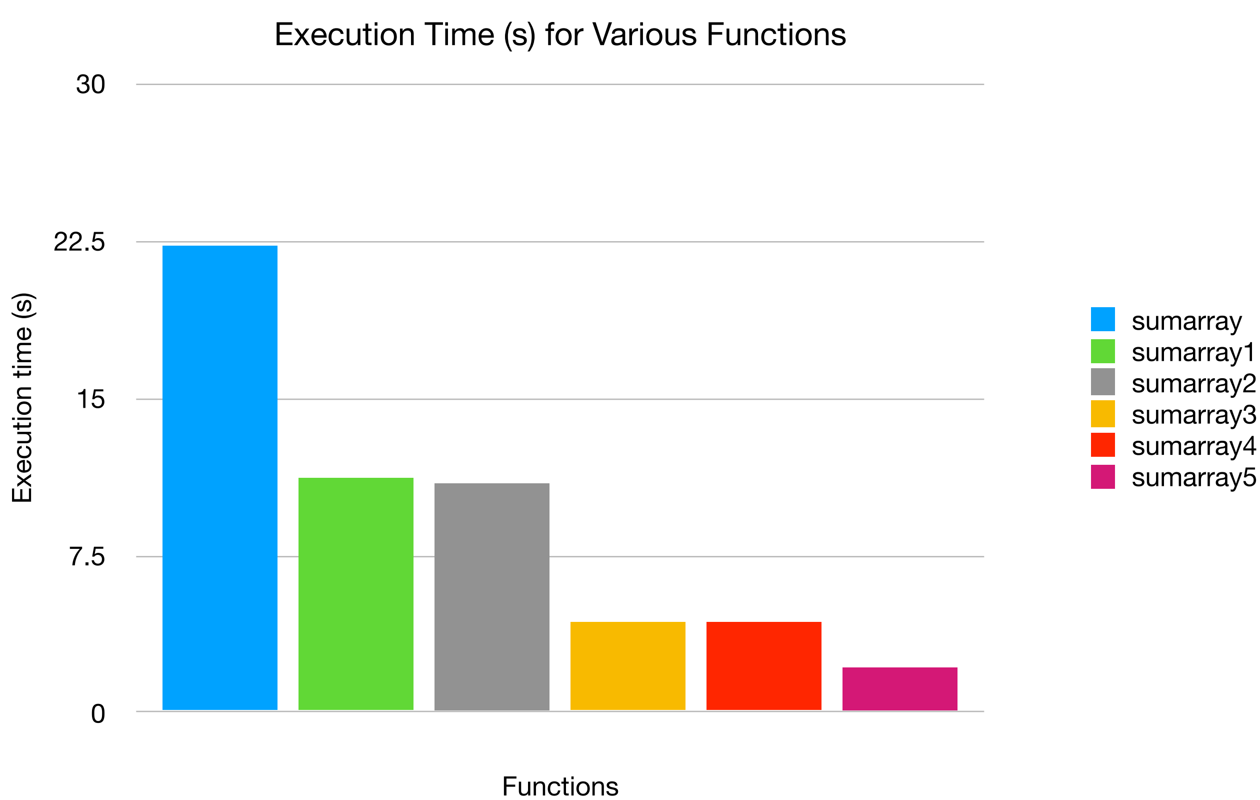
Execution results:

```
MichaelKuby@Michaels-iMac Question 3 % gcc-11 -O0 sumarray.c -o sumarray
MichaelKuby@Michaels-iMac Question 3 % ./sumarray
The sum of the values of the array is 24332
The sum of the values of the array is 24332
The sum of the values of the array is 24332
The sum of the values of the array is 24332
The sum of the values of the array is 24332
The sum of the values of the array is 24332
MichaelKuby@Michaels-iMac Question 3 % 
```

Analysis

By compiling with -pg and testing with GPROF we see some interesting results:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 40.66     22.31    22.31        1    22.31    22.31  sumarray
 20.38     33.49    11.18        1    11.18    11.18  sumarray1
 19.84     44.38    10.89        1    10.89    10.89  sumarray2
  7.85     48.68     4.31        1     4.31     4.31  sumarray4
  7.77     52.95     4.27        1     4.27     4.27  sumarray3
  3.86     55.07     2.12        1     2.12     2.12  sumarray5
```



The third column “self seconds” shows us the total run time in seconds for each function. Our original code sumarray takes over 22 seconds to execute. The fastest of our 5 functions is sumarray5, which utilizes 10 x 10 unrolling. The difference here, rounded to the hundredth, is 22.31 / 2.12 = 10.52, meaning we’ve achieved a speed up of over a factor of 10. Pretty impressive.

From sumarray to sumarray1, which utilizes 2 x 1 unrolling with suboptimal association, we see an immediately drastic improvement of about a factor of 2. Somewhat surprisingly we notice that the reassociation technique utilized by sumarray2 offers almost no improvement over sumarray1. Nevertheless, it does add some improvement, and should be utilized.

From sumarray2 to sumarray3 we see another drastic improvement. sumarray3 uses 5 x 1 unrolling and the proper associations. At this point we have hit the latency bound of the hardware.

Interestingly, sumarray4 has a very slight regression in performance in attempting to use multiple accumulators. It's hard for me to pinpoint exactly why this is happening. My first guess was that the regression in performance was because the numerous number of variables cannot all be held in registers, causing variables to be stored in memory. This seemed unlikely given the fact that there are only 5 accumulators and x86-64 hardware has 16 general purpose registers. The results of sumarray5 go to show that the issue was not spillage.

sumarray5 is our top performer, breaking through the latency limit and getting what I would assume is somewhere close to the throughput limit. sumarray5 uses 10 x 10 unrolling, so it's speed is achieved via the use of a large number of accumulators, which exploits the functional capabilities of the systems hardware.