Cora English | 127007634

CSCE 312-201 Lab 5

6.1 Problem 1: The following "C" program (code fragment) is given –

```
int i,j;
…..
if (i > j) {
        i= i+5;
} else {
        i=0;
        j++;
}
```

Activities to do

1. Provide the Y86 assembly language code for the "C" program.

```
.pos 0
Init:
    irmovl Stack, %ebp
    irmovl Stack, %esp
Main:
    irmovl 1, %ecx      #i
    irmovl 4, %edx      #j
    pushl %edx          #push value onto stack
    subl %ecx, %edx     #j = j - i
    jle if              #jump if i > j
    irmovl 0, %ecx      #set i to 0
    popl %edx           #pop value to use
    irmovl 1, %ebx      #add one to temp reg
    addl %ebx, %edx     #incr j by 1
    rrmovl %edx, %eax   #value to be returned
    halt
if:
    irmovl 5, %ebx      #move 5 to temp reg
    addl %ebx, %ecx     #add 5 to i
    rrmovl %ecx, %eax   #value to be returned
    halt

.pos 0x100
Stack:
```

```
1  .pos 0
2  Init:
3      irmovl Stack, %ebp
4      irmovl Stack, %esp
5
6  Main:
7      irmovl 1, %ecx      #i
8      irmovl 4, %edx      #j
9      pushl %edx          #push value onto stack
10     subl %ecx, %edx     #j = j - i
11     jle if              #jump if i > j
12     irmovl 0, %ecx      #set i to 0
13     popl %edx           #pop value to use
14     irmovl 1, %ebx      #add one to temp reg
15     addl %ebx, %edx     #incr j by 1
16     rrmovl %edx, %eax   #value to be returned
17     halt
18
19 if:
20     irmovl 5, %ebx      #move 5 to temp reg
21     addl %ebx, %ecx     #add 5 to i
22     rrmovl %ecx, %eax   #value to be returned
23     halt
24
25 .pos 0x100
26 Stack:
```

2. Verify this assembly code using either the Y86 tool set.
3. Report your assumptions used to test this code. (e.g. the value of i and j)
    - Used i = 1, j = 4 and i = 6, j = 4 with eax storing 5 and 11 respectfully.

6.2 Problem 2: For the following "C" program (code fragment) –

```
int j,k;
.....
for (int i=0; i <5; i++) {
       j = i*2;
       k = j+1;
}
```

Activities to do

1. Provide the Y86 assembly language code for the "C" program.

```
.pos 0
Init:
    irmovl Stack, %ebp
    irmovl Stack, %esp
Main:
    irmovl 0, %eax      #initialize counter
    pushl %eax          #push to stack
    irmovl 1, %edx      #const val 1
for:
    popl %eax           #update counter
    irmovl 0, %esi      #j
    addl %eax, %esi     #mult j by 2
    addl %eax, %esi     #^
    rrmovl %esi, %edi   #k = j
    addl %edx, %edi     #k = j + 1
    irmovl 5, %ecx      #loop cond
    irmovl 1, %ebx      #store 1
    addl %ebx, %eax     #update count by 1
    pushl %eax          #save counter on
stack
    subl %ecx, %eax     #check count > cond
    jl for              #loop if needed
.pos 0x100
Stack:
```

```
1  .pos 0
2  Init:
3      irmovl Stack, %ebp
4      irmovl Stack, %esp
5
6  Main:
7      irmovl 0, %eax      #initialize counter
8      pushl %eax          #push to stack
9      irmovl 1, %edx      #const val 1
10 for:
11     popl %eax           #update counter
12     irmovl 0, %esi      #j
13     addl %eax, %esi     #mult j by 2
14     addl %eax, %esi     #^
15     rrmovl %esi, %edi   #k = j
16     addl %edx, %edi     #k = j + 1
17     irmovl 5, %ecx      #loop cond
18     irmovl 1, %ebx      #store 1
19     addl %ebx, %eax     #update count by 1
20     pushl %eax          #save counter on stack
21     subl %ecx, %eax     #check count > cond
22     jl for              #loop if needed
23
24 .pos 0x100
25 Stack:
```

2. Verify this assembly code using the Y86 tool set and state your assumptions.
    - j, k = 0 at beginning of loop and will be reassigned each time loop is executed.
    - Final values: j = 8, k = 9

6.3 Problem 3: For the following two "C" programs –

Activities to do:

1. Use gcc to generate the equivalent assembly codes for these two "C" programs.
2. Generate the executable codes from the generated assembly codes.
    - Can be found in zipped folder.
3. Compare and analyze the structure of the two assembly codes generated by gcc, identify the various assembly code segments and their respective roles on the printouts.
    - These two codes have a similar structure based on the fact that they both output strings. The difference seen in the second program is the inclusion of incrementing an integer and outputting that in the printf as well as the initialized string.

```
//Program 1, file name "lab5_prob3_1.c"
#include <stdio.h>
int main(int argc, char *argv[])
{
	printf("Hello, world\n");
	return 0;
}
```

```
  .file "lab5_prob3_1.c"          # file name
  .section  .rodata
.LC0:
  .string "Hello, world"          # initialize string to be used later in program
  .text
  .globl  main
  .type main, @function
main:                             # entering main of program
.LFB0:
  .cfi_startproc
  pushq %rbp                      # push frame pointer onto stack
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq  %rsp, %rbp                # copy contents of stack pointer to
  .cfi_def_cfa_register 6            frame pointer to maintain original val
  subq  $16, %rsp                 # increase stack pointer by 16 to increase space
  movl  %edi, -4(%rbp)            # store edi at 4 above start of stack
  movq  %rsi, -16(%rbp)           # store rsi at end of stack
  movl  $.LC0, %edi               # move string into edi register on stack
  call  puts
  movl  $0, %eax                  # store 0 in eax since func returns 0
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size main, .-main
  .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)"
  .section  .note.GNU-stack,"",@progbits
```

```
//Program 2, file name “lab5_prob3_2.c”
#include <stdio.h>
int main(int argc, char *argv[])
{
	int i = 1;
	i++;
	printf("The value of i is %d\n", i);
	return 0;
}
```

```
  .file "lab5_prob3_2.c"            # file name
  .section  .rodata
.LC0:
  .string "The value of i is %d\n"  # initialize string for later use
  .text
  .globl  main
  .type main, @function
main:                               # entering main of program
.LFB0:
  .cfi_startproc
  pushq %rbp                        # push frame pointer onto stack
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq  %rsp, %rbp                  # copy contents of stack pointer to
  .cfi_def_cfa_register 6             frame pointer to maintain original val
  subq  $32, %rsp                   # increase stack pointer by 32 to increase space
  movl  %edi, -20(%rbp)             # store edi at 20 above start of stack
  movq  %rsi, -32(%rbp)             # store rsi at end of stack
  movl  $1, -4(%rbp)                # store 1 at 4 above start of stack
  addl  $1, -4(%rbp)                # increment previous location by 1
  movl  -4(%rbp), %eax              # move previous value to eax
  movl  %eax, %esi                  # move value from eax to esi
  movl  $.LC0, %edi                 # move string into edi
  movl  $0, %eax                    # move 0 into eax since function returns 0
  call  printf                      # call printf to print blurb and i
  movl  $0, %eax                    # repeat the same action ?
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size main, .-main
  .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)"
  .section  .note.GNU-stack,"",@progbits
```

6.4 Problem 4: Following single "C" file are given –

```
//File 1, named "lab5_prob4_main.c"
#include <stdio.h>
void print_hello();
int main(int argc, char *argv[])
{
        print_hello();
        return 0;
}
void print_hello(){
        printf("Hello, world\n");
};
```

Activities to do

1. Generate assembly codes for this file. Compare it with the assembly code generated from the first file named "lab5_prob3_1.c" of the previous prob# 3, analyze and explain how the function call "print_hello()" was materialized inside the computer (compiler, OS and the hardware) for the above example. Use figures and text descriptions if necessary.

- This code is different from the one generated in prob_3_1 because print_hello() is now a separate function that is called by the main instead of just a printf line called within the main. The way the function was materialized by first converting it to assembly after the program is compiled, then it is translated into machine code which the processor can actually understand. After that the machine code is passed to the CPU where it goes through a decoder to be converted into control signals which sends the data through the CPU's functional units and produces the expected outcome. At least that's what one source says.

```
1   .file "lab5_prob4_main.c"
2   .text
3   .globl  main
4   .type main, @function
5 main:
6 .LFB0:
7   .cfi_startproc
8   pushq %rbp
9   .cfi_def_cfa_offset 16
10  .cfi_offset 6, -16
11  movq  %rsp, %rbp
12  .cfi_def_cfa_register 6
13  subq  $16, %rsp
14  movl  %edi, -4(%rbp)
15  movq  %rsi, -16(%rbp)
16  movl  $0, %eax
17  call  print_hello
18  movl  $0, %eax
19  leave
20  .cfi_def_cfa 7, 8
21  ret
22  .cfi_endproc
23 .LFE0:
24  .size main, .-main
25  .section  .rodata
26 .LC0:
27  .string "Hello, world"
28  .text
29  .globl  print_hello
30  .type print_hello, @function
31 print_hello:
32 .LFB1:
33  .cfi_startproc
34  pushq %rbp
35  .cfi_def_cfa_offset 16
36  .cfi_offset 6, -16
37  movq  %rsp, %rbp
38  .cfi_def_cfa_register 6
39  movl  $.LC0, %edi
40  call  puts
41  popq  %rbp
42  .cfi_def_cfa 7, 8
43  ret
44  .cfi_endproc
45 .LFE1:
46  .size print_hello, .-print_hel
47  .ident  "GCC: (GNU) 4.8.5 2015
48  .section  .note.GNU-stack,"",@
```

6.5 Problem 5: Following two "C" files are given –

```
//File 1, "lab5_prob5_main.c"
void print_hello();
int main(int argc, char *argv[]){
       print_hello();
       return 0;
}
```

```
//File 2, "lab5_prob5_print.c"
#include <stdio.h>
void print_hello(){
    printf("Hello, world\n");
};
```

Activities to do

1. Generate assembly codes for these two files. Then using these assembly files generate the object code files, then link the generated object files to make a single executable file. Use gcc toolset. Provide printout (or copy paste) of the assembly files.
    - `gcc -c lab5_prob5_main.S -o lab5_prob5_main.o`
    - `gcc -c lab5_prob5_print.S -o lab5_prob5_print.o`
    - `gcc lab5_prob5_main.o lab5_prob5_print.o -o lab5_prob5`

```
  .file "lab5_prob5_main.c"
  .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
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  movq  %rsi, -16(%rbp)
  movl  $0, %eax
  call  print_hello
  movl  $0, %eax
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size main, .-main
  .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)"
  .section  .note.GNU-stack,"",@progbits
```

```
  .file "lab5_prob5_print.c"
  .section  .rodata
.LC0:
  .string "Hello, world"
  .text
  .globl  print_hello
  .type print_hello, @function
print_hello:
.LFB0:
  .cfi_startproc
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register 6
  movl  $.LC0, %edi
  call  puts
  popq  %rbp
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size print_hello, .-print_hello
  .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)"
  .section  .note.GNU-stack,"",@progbits
```

2. Compare the assembly codes generated in activity # 1 of this problem against the assembly code generated in the previous problem # 4. Is there any difference how the function call "print_hello()" was materialized for this example ? Explain the differences if any, in the assembly codes generated in this problem with that of problem # 4.
    - The only difference was that print_hello() was generated in another file. Other than that the function assembly code is exactly the same. There is more going on in the background though since each file is manually converted to an object file then linked together with the commands.

6.6 Problem 6: The following "C" code is given –

```
 //File named "lab5_prob6.c"
#include <stdio.h>
int very_fast_function(int i){
       if ( (i*64 +1) > 1024) return i++;
       else return 0;
}
int main(int argc, char *argv[])
{
       int i;
       i=40;
       printf("The function value of i is %d\n", very_fast_function(i) );
       return 0;
}
```

Activities to do

1. Rewrite the above "C" code such that the "very_fast_function()" is implemented in Y86 or IA32 assembly language embedded in the "C" code.

```
int very_fast_function(int i) {
       __asm__(
              "movl  %edi, %eax\n\t"
              "sall  $6, %eax\n\t"
              "addl  $1, %eax\n\t"
              "cmpl  $1024, %eax\n\t"
              "jle   .L2\n\t"
              "movl  -4(%rbp), %eax\n\t"
              "leal  1(%rax), %edx\n\t"
              "movl  %edx, -4(%rbp)\n\t"
              "jmp   .L3\n\t"
       ".L2:\n\t"
              "movl  $0, %eax\n\t"
       ".L3:\n\t"
              "popq  %rbp\n\t"
              "ret"
        );
      }
```