Pranav Velamakanni
Assignment 3
07/26/2015

Basic functionality of Mystery

- Mystery.s has three functions named:

1. add
2. compute_fib
3. main

- Looking at the simplest function add and the following snippet

```
.globl add
        .type   add, @function
add:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
        ret
        .size   add, .-add
```

We come to know that it simply intakes 2 arguments and return their sum.

- Looking at the code snippet at the end of main

```
        .comm num,800,32
```

This is allocating contigous 800 bytes with 32 bit alignment , it simply means that it is allocating an array of size 200 elements with each element having ability to store 4 bytes (possibly integers)

- subl $36, %esp

This line allocates 9 local variables for the function main.

- Now having look at the following snippet

```
        movl    %eax, -12(%ebp)
        movl    $0, -8(%ebp)
        jmp     .L16
.L17:
        movl    -8(%ebp), %eax
        movl    $-1, num(,%eax,4)
        addl    $1, -8(%ebp)
.L16:
        cmpl    $199, -8(%ebp)
        jle     .L17
```

Its a loop with loop variable looping from 0 to 199 and this is stored at -8(%ebp)
Its loop body is label L17 which initializes each element of num array to -1.(Consider
movl $-1, num(,%eax,4))

- This snippet in main function
  ```
  movl    (%eax), %eax
  movl    %eax, (%esp)
  call    atoi
  ```

Converts some String to Integer with atoi call and saves it to 3rd local variable on stack i.e. -12(%ebp)

- Then connecting above snippet with this

Line 101-104
```
movl    -12(%ebp), %eax
movl    %eax, (%esp)
call    compute_fib
```

It calls compute_fib for that variable

- Then these lines of code:
  ```
  movl    %eax, 4(%esp)
  movl    $.LC0, (%esp)
  call    printf
  ```

Call printf function. Format specifier verifies that result of compute_fib was an integer.

- Next lines
  ```
  movl    $0, %eax
  addl    $36, %esp
  popl    %ecx
  popl    %ebp
  leal    -4(%ecx), %esp
  ret
  ```

Returns 0 from main and deallocates the local variables and restores the registers used.

- Coming back to compute_fib function. The lines
  ```
  pushl   %ebp
  movl    %esp, %ebp
  pushl   %ebx
  ```

stores the old base pointer , sets base pointer to stack frame pointer and then save the value of ebx. (there is a chance that ebx will be used in this function and can be restored from the stack later at the end of the function.)

- Line
```
subl    $28, %esp
```
Allocates 7 local variables on stack in compute_fib
- Lines of code
```
movl    8(%ebp), %eax
movl    num(,%eax,4), %eax
cmpl    $-1, %eax
```

Checks the first argument of the function compute_fib and checks that index in num array allocated earlier, then compares the value with -1
If it is not -1 then continues to following lines.(-1 was the value to which it was initialized earlier.)
```
movl    8(%ebp), %eax
movl    num(,%eax,4), %eax
movl    %eax, -24(%ebp)
jmp     .L6
```

These lines move this value in array num to 6th local variable in stack and then jumps to L6
- At L6

it stores the 6th variable into %eax and then gets ready to return by restoring the variables allocated earlier. This whole control in compute_lib tells if there was non -1 value in the index passed in the argument of compute_fib, it will simply return that value.
- At L4:

Now we consider the case when the value in num array at the index is -1 , so program control goes to label L4. We have this code at L4
```
movl    $-1, -8(%ebp)
cmpl    $0, 8(%ebp)
jne     .L7
movl    $0, -8(%ebp)
jmp     .L9
```
We initialize the first local variable to -1
Then compare the first and only argument of this function with 0. If it is 0, then we initialize first local variable to 0. and then we jump to L9, save the value 0 and return it.
Second case is when it is not 0 then we move to L7 and compare it with 1. If it is 1, then we initialize the first local variable to 1 and then we jump to L9 save the value 1 to num variable and return it

- Another case in this function is when argument is neither 0 nor 1, then we move to label L10.

- At label L10

```
movl    8(%ebp), %eax
subl    $2, %eax
movl    %eax, (%esp)
call    compute_fib
movl    %eax, %ebx
```

This snippet calls compute_fib again. If initially argument was n then it will be compute_fib(n-2)
Then last line saves the result to %ebx

```
movl    8(%ebp), %eax
subl    $1, %eax
movl    %eax, (%esp)
call    compute_fib
```

This snippet also calls compute_fib again. If initially argument was n then it will be compute_fib(n-1)

```
movl    %ebx, 4(%esp)
movl    %eax, (%esp)
call    add
movl    %eax, -8(%ebp)
```

This snippet adds the result of 2 recursive calls and save it to -8(%ebp) which is second local variable of the function.

- At L9 It checks if the value at index=1st argument of the function.
- If it is -1 then it stores the value calculated inside -8(%ebp) to appropriate index in num array
- If it is not -1 then it simply goes to Label L12 then it stores the value to %eax and returns

Optimization to speed up the computation

The program involves an optimization to speed up the computation. It uses an array to speed up the process. Here is an example to explain it. Suppose we are calculating 5th fibonacci term. Then recursion tree will go like this.

```
                    fib(5)
                  /           \
             fib(4)                fib(3)
           /      \              /     \
       fib(3)      fib(2)    fib(2)    fib(1)
      /     \       /    \      /    \
 fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
```

```
  /    \
fib(1) fib(0)
```

As you can see that function is called 4 times for 1 and 3 times for 0. 3 times for 2, which will again call itself to calculate the 2nd term. Hence time needed to calculate fibonacci term increases dramatically.

To improve computation the program uses auxiliary array of integers, which will store the temporary results so on each call the function checks whether the requested term is calculated. This drastically improves runtime.

Compiler Optimization

Next thing is when what does the compiler does to optimize the assembly generated with -O option.
Few points to note:

1. Considering following lines of code
   ```
   subl    $20, %esp;optimized code
   ```
   Optimized code allocates 20 bytes of local variable space which was exactly what was required and un-optimized code allocates 36 bytes of local variable space which was actually indicated by the source code but was not actually required.
2. Optimized code makes use of %esi register and not %ebp variable.
3. Consider the code from unoptimized version in last label before return
   ```
   movl    8(%ebp), %eax
   movl    num(,%eax,4), %eax
   movl    %eax, -16(%ebp)
   movl    -16(%ebp), %eax
   ```
   This uses a local variable to store the final value and then return it to accumulator. Now consider from optimized version of code in last label.
   ```
   movl    num(,%ebx,4), %eax
   ```