Lab Week #6
CIS 314

November 5, 2017

1 Y86 Assembly Simulator

After loading the Y86 simulator (`https://xsznix.github.io/js-y86/`), you'll be greeted with the following code:

```
.pos 0
Init:
   irmovl Stack, %ebp
   irmovl Stack, %esp

.pos 0x100
Stack:
```

The above creates some variable "Stack", and makes it an alias of the memory location 0x100. In addition, it has two instructions to set the stack and base pointer to the hex value 0x100.

From here, we can start doing useful things by adding a "main" procedure:

```
.pos 0
Init:
   irmovl Stack, %ebp
   irmovl Stack, %esp
   call Main
   halt

Main:
   pushl %ebp
   rrmovl %esp, %ebp //function prologue

   irmovl $2, %eax

   popl %ebp
   ret             //function epilogue

.pos 0x100
Stack:
```

This procedure isn't very interesting – it only moves the value '2' in to the register eax...

2 An Add Procedure

Let's make a more interesting procedure, one to add two numbers:

```
.pos 0
Init:
   irmovl Stack, %ebp
   irmovl Stack, %esp
   call Main
   halt

Add:   //int add(int a, int b)
   pushl %ebp
   rrmovl %esp, %ebp //prologue


   mrmovl 8(%ebp), %eax  //get first argument
   mrmovl 12(%ebp), %ecx //get second argument
   addl %ecx, %eax       //add the two arguments

   popl %ebp
   ret        //epilogue


Main:
   pushl %ebp
   rrmovl %esp, %ebp //prologue

   irmovl $2, %eax
   pushl %eax
   irmovl $3, %eax
   pushl %eax
   call Add

   brk        //at this point, the result is stored in %eax

   popl %ebp
   ret        //epilogue

.pos 0x100
Stack:
```

Uh oh! At the end of the execution, esp and ebp weren't put back to where they started! We need to clean up the stack in Main after we have fiddled with it!

```
.pos 0
Init:
   irmovl Stack, %ebp
   irmovl Stack, %esp
   call Main
   halt

Add:   //int add(int a, int b)
   pushl %ebp
   rrmovl %esp, %ebp //prologue


   mrmovl 8(%ebp), %eax  //get first argument
   mrmovl 12(%ebp), %ecx //get second argument
   addl %ecx, %eax       //add the two arguments

   popl %ebp
   ret        //epilogue


Main:
   pushl %ebp
   rrmovl %esp, %ebp //prologue

   irmovl $2, %eax
   pushl %eax
   irmovl $3, %eax
   pushl %eax
   call Add

   rrmovl %ebp, %esp //at this point, the result is stored in %eax

   popl %ebp
   ret        //epilogue

.pos 0x100
Stack:
```

That's better.

3 Mult

Now for an even harder procedure. Since y86 doesn't have a multiplication instruction, if we want to multiply two numbers, we'll need to repeatedly add. Let's have a first attempt:

```
Mult: //int mult(int a, int b)
   pushl %ebp
   rrmovl %esp, %ebp

   mrmovl 8(%ebp), %eax
   mrmovl 12(%ebp), %ecx
   irmovl $1, %edx

   loop:
      addl %eax, %eax
      subl %edx, %ecx
      jg loop


   popl %ebp
   ret
```

For some reason, this code actually serves to left shift! A moment's thought reveals that the problem is in the 'add' instruction – we're always doubling eax! Let's try this instead:

```
Mult: //int mult(int a, int b)
    pushl %ebp
    rrmovl %esp, %ebp
    pushl %ebx          //since ebx is callee save, we need to back it up

    mrmovl 8(%ebp), %eax
    rrmovl %eax, %ebx
    mrmovl 12(%ebp), %ecx
    irmovl $1, %edx

    loop:
        addl %ebx, %eax
        subl %edx, %ecx
        jg loop

    popl %ebx
    popl %ebp
    ret
```

That still doesn't work! To figure out why, consider the case where we multiply something by 1 – we'll always get in to the loop, which implies we will always multiply by at least two! The solution is to decrement the counter variable before entering the loop!

```
Mult: //int mult(int a, int b)
    pushl %ebp
    rrmovl %esp, %ebp
    pushl %ebx

    mrmovl 8(%ebp), %eax
    rrmovl %eax, %ebx
    mrmovl 12(%ebp), %ecx
    irmovl $1, %edx
    subl %edx, %ecx
    je end

    loop:
        addl %ebx, %eax
        subl %edx, %ecx
        jg loop

    end:
    popl %ebx
    popl %ebp
    ret
```

There are some obvious problems with the procedure – it won't work for 0 or anything negative – but it serves to show some fun things in y86.

4 Pointers in Y86

Up until now, we've only seen how to deal with values... what about dealing with pointers? We can declare an array as follows:

```
.pos 0x104 //make the array start at location 0x104
.align 4   //make each element in the array occupy 4 bytes
array:
    .long 0x1  //element 0
    .long 0x2  //element 1
    .long 0x3  //element 2
```

Neat! What if we modify the add procedure to take pointers, and we return the result of adding the dereferenced values together:

```
Add:   //int add(int* a, int* b)
    pushl %ebp
    rrmovl %esp, %ebp //prologue


    mrmovl 8(%ebp), %eax  //get first argument
    mrmovl 12(%ebp), %ecx //get second argument
    mrmovl (%eax), %eax   //dereference first argument
    mrmovl (%ecx), %ecx   //dereference second argument
    addl %ecx, %eax       //add the two arguments

    popl %ebp
    ret        //epilogue
```

That's all fine and good, but how do we send a pointer to 'Add'?

```
Main:
    pushl %ebp
    rrmovl %esp, %ebp //prologue

    irmovl array, %eax
    irmovl $4, %ecx
    pushl %eax
    addl %ecx, %eax
    pushl %eax
    call Add

    rrmovl %ebp, %esp

    popl %ebp
    ret        //epilogue
```

The above will add array[0] + array[1]. Fun!