Lab Week #4
CIS 314

October 23, 2016

1 A Note on `sall`

In the last assignment, we saw how to use `sall` where the first operand was an immediate value (namely, `$31`), but this assignment sees us using the same instruction with a register as the first value.

After looking at the textbook (specifically the diagram on page 168), we can see that `%cl` is simply the lowest order byte in register `%ecx`. Therefore, one may assume that in order to write an equivalent shift in C, we'd have to mask off the low order byte like this:

```
//Assume 'a' is in eax, 'c' is in ecx
// We're attempting to write code that is functionally equivalent to 'sall %cl, %eax'
a = a << (c & 0xFF);
```

While the above C code is *semantically* equivalent to the assembly, it turns out that the compiler will do this masking automatically for us. To see why, consider the fact that an `int` is 4 bytes (32 bits), and the first operand to the `sall` instruction is the number of bits to shift by. If that operand evaluated to a number which was greater than 32, our shift would result in undefined behavior. To solve this, the compiler will implicitly mask only the *lowest 5 bits* of the value contained in `%ecx` (in this example) to shift by[1], so we can simply write the analogous code as follows:

```
//Assume 'a' is in eax, 'c' is in ecx
// The below C code is functionally equivalent to 'sall %cl, %eax'
a = a << c;
```

To verify this, try compiling your example C program down to assembly!

2 Converting Iteration to Jumps/Goto

As we have seen, there are no 'iteration' instructions in assembly – the computer seems to magically transform our loops into combinations of conditional jumps and comparisons! To see how this works, we'll start by writing some (contrived) C program which uses a `for` loop, and transform it step-by-step into a program which only uses labels and conditional `goto`'s.

```
//A function which adds up the numbers from 0 to 10

int i;
int result=0;
for(i=0; i<=10; i++){
  result = result + i;
}
```

Our first transformation will turn the above into its equivalent representation using a `while` loop. This is as simple as moving the initialization of our index variable before the loop, and putting the increment operator in the body:

```
//A function which adds up the numbers from 0 to 10
int i=0;
int result=0;
while(i<=10){
  result = result + i;
  i++;
}
```

[1] Recall that the maximum unsigned number represented by k bits is $2^k - 1$

The book mentions (somewhere) that the assembly produced by the compiler resembles the `do...while()` variant of `while` loops. Therefore, the next transformation will do just that[2]:

```
//A function which adds up the numbers from 0 to 10
int i=0;
int result=0;
do{
  result = result + i;
  i++;
}while(i<=10);
```

Now we have something which looks somewhat familiar! To get to a conditional `goto` statement, we just mechanically replace `do` with some label, and the `while` condition with a conditional `goto`:

```
//A function which adds up the numbers from 0 to 10
int i=0;
int result=0;
loop:
  result = result + i;
  i++;
if(i<=10){
  goto loop;
}
```

3 GCC Inline Assembly

As it turns out, the previous section is *not* the farthest we can take C in our attempt to write assembly... GCC will allow us to write assembly right in our C program! Unfortunately, it requires a little bit more work than simply writing the assembly, but we then have access to all of our regular C functions! As an example, below is a program which will print the number 10:

```
#include <stdio.h>
//A function which prints 10
//Compile with:
//    gcc [filename] -m32

int main(){
  int result;
  //__volatile__ tells the compiler to never optimize the assembly away
  asm __volatile__(
    //Moving the value 10 into register %eax (we have to escape the first percent sign...)
    "movl $10, %%eax;"
    //This says that the output (=) register (a) should place its value in result
    : "=a"(result));

  printf("Result is %d\n", result);
}
```

[2] **Careful!** In general, this transformation *may not* be this straightforward... specifically, we're moving the guard which can result in different behavior.

That's neat! Now we have enough knowledge to write the conditional `goto` code from the last section in inline assembly:

```
#include <stdio.h>
//A function which prints 10
//Compile with:
//    gcc [filename] -m32
int main(){
  int result;
  asm __volatile__(
    //%ebx is callee saved...
    "pushl %%ebx;"
    //%eax is where we're storing 'result'
    "movl $0, %%eax;"
    //%ebx is where we're storing 'i'
    "movl $1, %%ebx;"
    // Create a label
    "loop:\n\t"
    // result = result + i
    "addl %%ebx, %%eax;"
    // i++
    "addl $1, %%ebx;"
    // if(i <= 10){goto loop;}
    "cmpl $10, %%ebx;"
    "jle loop;"
    "popl %%ebx;"
    : "=a"(result));

  printf("Result is %d\n", result);
}
```

We can also write procedure calls in inline assembly, though we have to trick the compiler into placing the generated assembly in the correct place. To do this, we can simply create a dummy function to 'house' the assembly of our procedure, and call it from inline assembly in main. The following example will create a procedure labelled `F:` which serves as the identity function (it returns its argument):

```
#include <stdio.h>
//An example of inline assembly procedure calls
//Compile with:
//    gcc [filename] -m32
int result;
void dummy(){
  asm __volatile__(
    "F:\n\t"
    "pushl %%ebp;"
    "movl %%esp, %%ebp;"
    //Our single argument is located at 8(%ebp)
    "movl 8(%%ebp), %%eax;"
    "popl %%ebp;"
    //What happens if you leave this line off? Why?
    "retl;"
    : "=a"(result));
}
int main(){
  asm __volatile__(
    //Boilerplate procedure call set-up
    "pushl %%ebp;"
    "movl %%esp, %%ebp;"
    //I'm going to send the number 1 as an argument to F
    "pushl $1;"
    "call F;"
    //Shrink the stack back down
    "addl $4, %%esp;"
    "popl %%ebp;"
    : "=a"(result));

  printf("Result is %d\n", result);
}
```

3.1 Assembly Resources

- Some decent resources on procedure calling conventions are outlined here:
  `https://www.cs.virginia.edu/~evans/cs216/guides/x86.html`
- How to do inline assembly in GCC, straight from `GNU` itself:
  `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`
- More inline assembly information:
  `https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s6`

4 System Calls

We can in fact use this inline assembly to make calls directly to the operating system! In the following example, we call the `sys_exit` function (specified by the value 1 in register `%eax`[3]), and then we tell the operating system to execute this instruction by calling the interrupt vector 0x80:

```
#include <stdio.h>

int main(void)
{
  int eax;
  asm __volatile__("movl $1, %%eax;"
                   "int $128;"
                   : "=a" (eax));

  printf("I'll never get called :( \n");
}
```

[3] A comprehensive list of all system functions is available here: `http://syscalls.kernelgrok.com/`