CS61: Systems Programing and Machine Organization

Fall 2011

Section 2: Monday 19 September – Friday 23 September

Topics to be covered:

- **Flags**
- **Jumps**
- **Loops**
- **Procedure Calls**

Condition Codes

EFLAGS is a 32 bit register that contains separate bits for each of the condition flags, which are set automatically by the CPU to represent the result of the previously executed instruction. Examples of condition flags include the following:

CF: Carry Flag.	The most recent operation generated a carry out of the most significant bit. Used to detect overflow of unsigned operations.
ZF: Zero Flag.	The most recent operation yielded zero.
SF: Sign Flag.	The most recent operation yielded a negative value.
OF: Overflow Flag.	The most recent operation caused a two's-complement overflow -- either negative or positive.

Typically these flags are set or cleared as the result of an instruction (e.g. `add, sub, cmp,` etc.) and can then be used to conditionally set a single byte (`set`), jump to a new part of the program (`jmp`) or transfer some data (`mov`).

Q1: For each one of the following, determine which flags are set by the add instruction and why.

[a]

```
movl $0x40, %eax
movl $0xffffffc0, %ebx
addl %eax, %ebx
```

[b]

```
movl $0x2a, %eax
movl $0xffffffc0, %ebx
addl %eax, %ebx
```

[c]

```
movl $0x7FFFFFF0, %eax
movl $0x2c, %ebx
addl %eax, %ebx
```

Jumps

There are two methods of performing jumps: *direct* and *indirect*. For direct jumps, the destination is specified as a label (e.g. `jmp .L1` or, after compiling, `jmp 0x8049994`) and is encoded as part of the instruction. For indirect jumps, the jump target is read from a register or a memory location and is preceded by a '*'. For example:

```
jmp *%eax
```

uses the value in register `%eax` as the jump target.

Certain jumps are combined with certain condition flags to create conditional jumps:

Instruction		Synonym	Description
`je`	*Label*	`jz`	Equal / zero
`jne`	*Label*	`jnz`	Not equal / not zero
`js`	*Label*		Negative
`jns`	*Label*		Nonnegative
`jg`	*Label*	`jnle`	Greater
`jge`	*Label*	`jnl`	Greater or equal
`jl`	*Label*	`jnge`	Less
`jle`	*Label*	`jng`	Less or equal
`ja`	*Label*	`jnbe`	Above
`jae`	*Label*	`jnb`	Above or equal
`jb`	*Label*	`jnae`	Below
`jbe`	*Label*	`jna`	below or equal

Q2: Which of the condition flags do each of the above jump instructions use in determining if it will execute the jump?

Control Flow: Loops

Let us now see how loops are implemented using conditional jumps. The following is a simple function to compute a Fibonacci sequence:

```
int fibonacci(int n) {
    int i = 0;
    int val = 0;
    int nval = 1;
    do {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    } while (i < n);
    return val;
}
```

Generate the assembly code in the cs61 machine:

```
$ gcc -O2 -S -m32 fibonacci.c
```

Let's look at the code of this function, and focus on the code inside the loop.

Register	Variable	Initially
%ecx	i	0
%ebx	val	0
%edx	nval	1
%esi	n	n
%eax	t	0

```
fibonacci:
     pushl   %ebp               # save old value of %ebp
     xorl  %ecx, %ecx           # i = 0
     movl  %esp, %ebp           # %ebp = base of current stack frame
     movl  $1, %edx             # nval = 1
     pushl   %esi               # save previous value of %esi
     movl  8(%ebp), %esi        # load n into %esi
     pushl   %ebx               # save previous value of %ebx
     xorl  %ebx, %ebx           # val = 0
     jmp   .L2                  # jump to .L2
.L7:
     movl  %eax, %edx           # nval = t
.L2:
     addl  $1, %ecx             # i++
     cmpl  %esi, %ecx           # compare i to n
     leal  (%edx,%ebx), %eax    # t = val + nval
     movl  %edx, %ebx           # val = nval
     jl    .L7                  # Jump if i < n
     popl  %ebx                 # restore %ebx
     movl  %edx, %eax           # Set nval (==val) as the ret. value
     popl  %esi                 # restore %esi
     popl  %ebp                 # restore %ebp
     ret                        # pop return address and jump to it
```

Note that assembly code instructions do not always appear in the same order as the corresponding code in the C program. For example, `i` is incremented near the beginning of the loop in the assembly program, but is incremented at the end of the loop in the C source program. The compiler is free to re-arrange the order of the instructions as long as it does not change the meaning, or behavior, of the code.

Q3: Which line in the assembly actually causes the code to loop? What lines are important in making sure that we don't loop forever?

Now we'll look at fibonacci defined slightly differently:

```
int fibonacci(int n) {
      // ignoring negative n
      if(n == 0 || n == 1)
        return n;
      else
        return fibonacci(n-2) + fibonacci(n-1);
}
```

Q4: What is the stack going to look like midway through a call to, say, fibonacci(100000)?

Let's try one more time:

```
int fibonacci(int n) {
    if(n < 3)
        return 1;
    else
        return fibonacci_helper(n-2,1,1);
}

int fibonacci_helper(int n, int n0, int n1) {
    if(n == 0)
        return n1;
    return fibonacci_helper(n-1, n1, n0+n1);
}
```

Q5 (Bonus): What's so different about this particular implementation of fibonacci? What happens to the stack / what does the stack look like midway through a call to fibonacci(100000)?

Q6: Consider the following assembly code:

```
# x at %ebp+8, n at %ebp+12
   movl    8(%ebp), %esi
   movl    12(%ebp), %ebx
   movl    $-1, %edi
   movl    $1, %edx
.L2:
   movl    %edx, %eax
   andl    %esi, %eax
   xorl    %eax, %edi
   movl    %ebx, %ecx
   sall    %cl, %edx
   testl   %edx, %edx
   jne     .L2
   movl    %edi, %eax
```

The preceding code was generated by compiling C code that had the following overall form:

```
int loop(int x, int n)
{
    int result = __________;
    int mask;
    for (mask = ___________; mask ___________; mask = __________) {
        result ^= ___________;
    }
    return result;
}
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%eax`. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables.

a. Which registers hold program values `x`, `n`, `result`, and `mask`?
b. What are the initial values of `result` and `mask`?
c. What is the test condition for `mask`?
d. How does `mask` get updated?
e. How does `result` get updated?
f. Fill in all the missing parts of the C code.

Bonus: Why are the arguments to `loop` pushed on the stack in reverse order (i.e., x ends up closer to loop's %ebp than n)? Why can't we do it the other way around?

Procedure Calls

Q7: Lets say we are given the following assembly code for a function:

```
pushl %edi
pushl %esi
pushl %ebx
sub $0x24, %esp
movl 24(%ebp), %eax
imull 16(%ebp), %eax
movl 24(%ebp), %ebx
leal 0(, %eax, 4), %ecx
addl 8(%ebp), %ecx
movl %ebx, %edx
subl 12(%ebp), %edx
……
popl %ebx
popl %esi
popl %edi
```

A: Why are `%edi, %esi, and %ebx` pushed onto the stack at the beginning of this function and popped off at the end?

B: What about `%eax, %edx, and %ecx`? Why aren't they put on the stack?

C: What do `24(%ebp)` and `16(%ebp)` refer to?

D. Why do we subtract 0x24 from %esp? What might be put in that area?

Q8: Lets say we are given the following assembly code for a function:

```
int proc(void) {
    int x, y;
    scanf("%x %x", &y, &x);
    return x - y;
}
```

and the corresponding assembly code generated is:

```
1 proc:
2   pushl %ebp
3   movl %esp, %ebp
4   subl $24, $esp
5   addl $-4, %esp
6   leal -4(%ebp), %eax
7   pushl %eax
8   leal -8(%ebp), %eax
9   pushl %eax
10  pushl $.LC0         # Pointer to string "%x %x"
11  call scanf
12  movl -8(%ebp), %eax
13  movl -4(%ebp), %edx
14  subl %eax, %edx
15  movl %edx, %eax
16  movl %ebp, %esp
17  popl %ebp
18  ret
```

Lets assume procedure `proc` starts executing with the following register values:

`%esp` = `0x800040`
`%ebp` = `0x800060`

Suppose proc calls scanf (line 11) and scanf reads values 0x46 and 0x53 from the standard input. Assume the string "%x %x" is stored at memory location `0x300070` (i.e., the label `.LC0` is translated to the address 0x300070).

a. What value does `%ebp` get on line 3?
b. At what addresses are local variables `x` and `y` stored?
c. What is the value of `%esp` after line 10?
d. What does the stack frame look like before line 11? If the line numbers all the way on the left were the addresses of the instructions, what value would the call instruction push onto the stack?