CS61 Section Notes 3

(Week of 10/1 - 10/5)

0. GDB Primer

Some gdb commands you should know (most helpful ones are starred):

A. Execution

- `*run` (`r`) - run (or restart) the program; pass parameters here
- `*quit` (`q`) - exit gdb
- `*continue` (`c`) - continue execution of the program until the next breakpoint
- `*si` - step forward one (or more) assembly instructions, entering function calls
- `*ni` - step forward one (or more assembly instructions, skipping function calls
- `step` (`s`) - step forward one line of source code, entering function calls (if source is available)
- `next` (`n`) - step forward one line of source code, skipping function calls (if source is available)

B. Breakpoints

- `*break` (`b`) - set a breakpoint at a function, address, or line number
- `delete` (`d`) - delete a given breakpoint (or all breakpoints if no parameter given)
- `info breakpoints` (`i b`) - list all breakpoints
- `disable` - disables breakpoints, and does not delete them
- `enable` - enables a disabled breakpoint

C. Examining code/data

- `*print` (`p`) - can print a register value or variable value
- `*x` - print memory value(s) at a given address (ie, examine memory contents)
- `disas` - disassemble a function
- `info registers` (`i r`) - prints values in register
- `info frame` (`i f`) - prints info about current stack frame
- `backtrace` (`bt`) - prints stack backtrace
- `list` (`l`) - list source code, including line numbers, functions, and more

D. Miscellaneous

- `help` - list commands, or get help on a particular command
- Tip - you can dictate the format of printed values using switches like `/x`, `/d`, `/b`, and more.

See http://cs61.seas.harvard.edu/wiki/Useful_GDB_commands for examples

1. Assembly Operand specifiers

Assembly instructions can take three different types of operands: a constant, or immediate, value, a register value, or a memory value.

Type	Form	Operand value	Example
Immediate	*$Imm*	*Imm*	$42
Register	E_a	$R[E_a]$	%eax
Memory	$Imm(E_b, E_i, s)$	$M[Imm] + R[E_a] + R[E_i] * s$	$42(%esp, %edx, 4)

Things to note:

1. *b* for “base”, i for “index”, s for “scale”
2. scale has to be one of 1, 2, 4, or 8

This last operand form is one of many ways of accessing memory, though it is the most general. The full list of memory operand specifiers is given in Figure 3.3 of the text (pg. 169). This is the most useful form to remember though, because we can derive all of the others from it. Basically, the other forms leave of some of the arguments. One way to think of those forms is as supplying “default” arguments to this specifier, where the defaults are 0 for *Imm*, $R[E_b]$, and $R[E_i]$, and 1 for *s*.

Q1: Assume the following values are stored at the indicated memory addresses and registers.

Address	Value	Register	Value
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x1
0x108	0x13	%edx	0x3
0x10C	0x11		

Fill in the missing value for each operand:

Operand	Value
%eax	
0x104	
$0x108	
(%eax)	
4(%eax)	
9(%eax, %edx)	
260(%ecx, %edx)	
0xFC(, %ecx, 4)	
(%eax, %edx, 4)	

Q2: A function with prototype int decode(int x, int y, int z); is compiled into assembly. The body of the code is as follows:

```
1 # x at %ebp+8, y at %ebp+12, z at %ebp+16
2 movl  12(%ebp), %edx
3 subl  16(%ebp), %edx
4 movl  %edx, %eax
5 sall  $31, %eax
6 sarl  $31, %eax
7 imull 8(%ebp), %edx
8 xorl  %edx, %eax
```

Parameters `x`, `y`, and `z` are stored at memory locations with offsets 8, 12, and 16 relative to the address in register `%ebp`. The code stores the return value in register `%eax`. Write the C code for decode that will have an effect equivalent to our assembly code.

```
int decode(int x, int y, int z) {




}
```

2. 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`).

Q3: 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
```

3. 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

Q4: Which of the condition flags do each of the above jump instructions use in determining if it will execute the jump?

4. 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.

Q5: 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);
}
```

Q6: 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);
}
```

Q7(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)?

Q8: Consider the following assembly code:

```
# x at %ebp+8, n at %ebp+12
1     movl    8(%ebp), %esi
2     movl    12(%ebp), %ebx
3     movl    $-1, %edi
4     movl    $1, %edx
5  .L2:
6     movl    %edx, %eax
7     andl    %esi, %eax
8     xorl    %eax, %edi
9     movl    %ebx, %ecx
10    sall    %cl, %edx
11    testl   %edx, %edx
12    jne     .L2
13    movl    %edi, %eax
```

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

```
1  int loop(int x, int n)
2  {
3      int result = _________;
4      int mask;
5      for (mask = __________; mask __________; mask = _________) {
6          result ^= __________;
7      }
8      return result;
9  }
```

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?

5. Procedure Calls

Q9: Lets say we are given the following assembly code for a function:

```
1 pushl %edi
2 pushl %esi
3 pushl %ebx
4 sub $0x24, %esp
5 movl 24(%ebp), %eax
6 imull 16(%ebp), %eax
7 movl 24(%ebp), %ebx
8 leal 0(, %eax, 4), %ecx
9 addl 8(%ebp), %ecx
10 movl %ebx, %edx
11 subl 12(%ebp), %edx
……
20 popl %ebx
21 popl %esi
22 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?

Q10: 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?