CS 61 Scribe Notes

Annie Lin, Lukas Missik, Marcus Powers, Curtis Stone

Thursday, 10/09/14

1 Announcements

- Sam Fishman/Victor (Part of the Crimson): Oct 18-19 Harvard Hackfest at the iLab (sponsored by LiveRamp). Grand prize $500, Junior category grand prize $250 (per team member). Contact sam.fishman@thecrimson.com
- Problem Set: Office hours on Friday. Grading server is being fixed.
- Midterm next Thursday (different location). Sample questions will be posted.

2 Machine Instructions and Machine programming

- **Instruction**: A series of bytes that the processor interprets to perform a specific task.
- **Assembly Language**: A very low-level language where each machine instruction is given a mnemonic name.
- **Memory Hierarchy:** Registers are at the top, followed by caches, then primary memory. Since registers are on the CPU itself, they are extremely fast, but there are only about 8 general purpose registers with 32 bits each.

Today, we will try to guess what .c files could have produced the .s assembly code.

- Strategy: Make and test assumptions, look for patterns, infer meanings from the names of instructions.

◊ f00.s

```
.file   "f00.c"
.text
.globl  f
.type   f, @function
```

```
f:
.LFB0:
        ret
.LFE0:
        .size   f, .-f
        .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
        .section        .note.GNU-stack,"",@progbits
```

From `.type f, @function`, we can infer the function name is f. This function just returns, and does nothing else. How could we figure out the machine code version of the `ret`instruction?

- Option 1: Make the object code file, then use `objdump -S f00.o`
- Option 2: Use hexdump (from a previous lecture), and give it `f`'s address.

In this way, we may find that `ret` has the hexadecimal value `c3`.

◇ f01.s

```
        .file   "f01.c"
        .text
        .globl  f
        .type   f, @function
f:
.LFB0:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $0, %eax
        popl    %ebp
        ret
.LFE0:
        .size   f, .-f
        .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
        .section        .note.GNU-stack,"",@progbits
```

- From `pushl` and `popl`, we can infer that we're using a stack, which has the push (add to top) and pop (remove from top) operations. Since push and pop cancel each other out, we see that the state of the stack is the same at the beginning of the function as at the end (right before return). We use the stack to store local variables.
- `movl` will just put the value in the source (first argument) into the destination (second argument). The `l` here means the data being moves is 32 bits, and dates from a time when the word size on machines was only 16 bits, so 32 bits was considered "long."

- The dollar sign `$` represents a constant value. The percent sign `%` represents a register. `%eax` is the most important register, called the accumulator, and stores the return value.
- Thus, the line `movl $0, %eax` stores the value 0 in `%eax`.
- We see that in f01.c, the function just returns 0 (a 32-bit integer).

When optimizing, we find that the move into `%eax` and return are necessary, but the push and pop don't have any effect and so may be optimized out.

◇ f02.s
This is optimized. From here on out, we'll just be showing the important parts of the file.

```
movl    a, %eax
addl    b, %eax
ret
```

- `a` and `b` are global variables. From the suffix `l` in the commands, we determine they are 32 bit values. However, they could have type int, float, char*, unsigned int, array of 4 chars, etc.
- New command: `addl src dst` in pseudo-C is `dst += src`.
- The C file is just `return a + b`, two ints. Note that in the machine, there are no types (these exist only in the abstract machine).

◇ f03.s

- The contents are the same as f02.s.
- In the .c file, we adds two integers, but one is unsigned. This tells us the instruction is the same for adding signed and unsigned integers (a result of using Two's Complement Arithmetic in the machine representation of signed integers).
- We note that if he changes the order to `b + a` instead of `a + b` in the .c file, the assembly code remains the same. This is because addition is commutative, so the compiler is free to choose the order.

◇ f04.s
The code is the same. Guess the types! In this case, we're adding the 0-th element of an integer array and an integer. Note that the address of the first element of the array is the same as the address of the variable referencing the array.
If we recompile with -O0, and read the result in pseudo-C, we get:

```
%edx = a;
%eax = b;
%eax += %edx;
```

◇ Questions following the break

- `.LFB0:` and `.LFE0:` mark the beginning and end of the function. Labels that start with a period are used by the compiler only, and disappear from the object file. Labels such as `f:` are accessible at runtime and represent addresses in the machine code, for example the beginning of a function.
- The keyword `extern` in C applied to a variable means the variable is defined in a different C file.
- The assembly code we're looking at is machine specific (only x86 processors can run it), but not C-specific. For example, a Java program may eventually compile to this same assembly code.

◇ f05.s
Same assembly as f04.s. We find out the C file is doing address arithmetic with a char* and an int, returning a value of type char*.

◇ f06.s

- Same assembly as f05.s. The variable a has type unsigned long long (8 bytes) and b has type int (4 bytes). The C code is `return (a & 0xFFFFFFFF) * b`. Note that a number is stored with the least significant byte at the lowest address in memory (a system called Little Endian), so the number 1 as an integer would be laid out in memory as `01 00 00 00`, where each pair of numbers constitutes one byte.
- The bitwise-and has the effect of "masking" the lower 32 bits (note 8 hex digits is 32 bits). This means that the higher 32 bits are thrown away, so the compiler can treat this like a 32-bit integer.

◇ f07.s

```
movl    a, %eax
subl    $3, %eax
ret
```

The C code is `return a - 3`, where a is an int.

◇ f08.s

- Same assembly as f07.s. Guess the types!
- Well, a is still an int, but the code is `return a + 4294967293;`.
- In the machine, this number is the "same" as -3. This is because when we add 3 to it, we get 2^{32} in the world of math, which is 0 in computer arithmetic, since the highest order bit is dropped as overflow.

◇ f09.s
Skipped in lecture

◇ f10.s

```
f:
.LFB0:
        pushl     %ebp
        movl      %esp, %ebp
        movl      x, %edx
        movl      y, %eax
        addl      %edx, %eax
        movl      %eax, a
        popl      %ebp
        ret
.LFE0:
        .size        f, .-f
        .globl        g
        .type        g, @function
g:
.LFB1:
        pushl    %ebp
        movl     %esp, %ebp
        movl     x, %edx
        movl     y, %eax
        subl     %eax, %edx
        movl     %edx, %eax
        movl     %eax, b
        popl     %ebp
        ret
.LFE1:
        .size        g, .-g
        .globl        h
        .type        h, @function
h:
.LFB2:
        pushl   %ebp
        movl    %esp, %ebp
        movl    x, %edx
```

```
        movl    y, %eax
        imull   %edx, %eax
        movl    %eax, c
        popl    %ebp
        ret
.LFE2:
```

Here we see three functions, f, g, and h, that manipulate the global variables a, b, c, x, and y. The first sets a = x + y, the second b = x - y, and the third c = x * y. Note use of the command `subl` and `imull`.

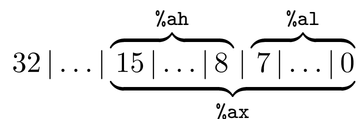
◇ f11.s
The new commands we see are:

```
        movzwl   x, %edx
        movzwl   y, %eax
        ...
        movw     %ax, c
```

- We know that `movl` moves a 32-bit number. The `w` stands for “word,” which was 16 bits long ago, so `movw` moves a 16-bit value.
- But `movzwl` has both `l` and `w`... The `z` stands for zero, so what it’s actually doing is moving 16 bits into the destination register, and extending the other, higher-order 16 bits with zero.
- We see the C code is the same as in f10, but the variables have type unsigned short (2 bytes) instead of unsigned int (4 bytes).

◇ Interlude on Registers

- x86 has about 8 general purpose registers, 6 of which are generally used for arithmetic. These are: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`. The `e` stands for “extended” and means the register holds 32 bits (earlier, registers held 16 bits).
- Looking at `%eax` (numbers represent bit addresses):



Here we see that `%ax` represents the lower 16 bits, the lower order byte is `%al`, and the second-lowest order byte is `%ah`. This naming pattern also applies to `%ebx`, `%ecx`, and `%edx`.

◇ f12.s
We see the same assembly code as f11 but with unsigned chars.

◇ f13.s
Skipped in lecture.

◇ f14.s
New commands are **andl**, **orl**, **xorl**, and **notl**, representing bitwise and, bitwise or, bitwise exclusive or, and bitwise complement (which is twiddle ~ in C).

◇ f15.s

```
xorl    %eax, %eax
ret
```

- This has a return value, as there is no reason to modify `%eax` if there were no return value. In pseudo-C, we can write this command as `%eax ^= %eax`, which has the effect of setting `%eax` to zero.
- How did he write C code to get this weird assembly that just returns zero?
- All he did was compile at -O3. It turns out this is more efficient because we don't need to load data into registers.
- If we compile f01.c with -O1, we get the command `movl $0, %eax`. Printing this using objdump, we see

  ```
  0:    b8 00 00 00 00        mov    $0x0, %eax
  5:    c3                    ret
  ```

  Whereas printing f15 gives us:

  ```
  0:    31 c0                 xor    %eax, %eax
  2:    c3                    ret
  ```

  We see an immense savings (50%) in code size!

◇ f16.s

```
movl    x, %eax
negl    %eax
movl    %eax, a
ret
```

New command: **negl** negates a value. This just sets a to -x.

◇ f17.s
The same assembly code as f16. However, the C code is different: `a = ~x + 1`. This means the compiler believes what he told us earlier, namely `-x == ~x + 1`.

◇ f18.s

```
pushl    %ebp
movl     %esp, %ebp
movl     x, %edx
movl     %edx, %eax
sall     $10, %eax
subl     %edx, %eax
movl     %eax, a
popl     %ebp
ret
```

- We know `x` is moved into `%edx`, then moved into `%eax`, so we're keeping 2 copies. Shifting between registers is faster than loading the value twice, since operations involving two registers are faster and their instructions more compact than those involving constants.
- `s` means shift. `sall` is a left-shift. In the above code, we're left-shifting the value of `%eax` by 10, which is equivalent mathematically to multiplying it by 2^{10}.
- Pseudo-C for the above code:

  ```
  %edx = %eax = x;
  %eax <<= 10;
  a = %eax - %edx
  ```
- Shifting is much faster for the processor than multiplying (expensive) or dividing (very expensive, ~100 cycles). So, if you know you're multiplying or dividing by a power of 2, do it with a shift (~1 cycle).
- The C code for this program is `a = x * 1023`. But we're multiplying x by 1024 (using left-shift) and then subtracting x at the assembly level to avoid the actual multiplication operation. The compiler is even doing this at -O0!

◇ f19.s
This has the same code as f18.s but we're not subtracting anything, so we just get `a = x * 1023`.

◇ f20.s
New command: `shrl`, which is a logical right-shift (will move zeros into the most significant side after shifting). Thus, shifting right by 10 is equivalent to dividing by 1024. Note that it's very important that the numbers are unsigned. The C code is `a = x / 1024`.

◇ f21.s

We get the same assembly code when the C code is `a = x >> 10`.

◇ f22.s

Same C code as f21.c, except we change the type to signed integers and see the command `sarl` appear in the assembly.

◇ f23.s

Here we're dividing a signed integer by 1024 in the C code. We get very nasty assembly. The point is that integer division is actually more mathematically complicated than unsigned division because the C abstract machine requires that we round to 0. For example, $-1/4$ evaluates to 0. But we know that an unsigned representation of -1 is a very large number ($2^{32} - 1$), and dividing this by 4 will definitely not yield zero. Thus, the division and multiplication instructions on the machine have both signed and unsigned variants.