CS 321 Lecture Notes

Orion Sky Lawlor
Department of Computer Science, University of Alaska at Fairbanks
http://lawlor.cs.uaf.edu/ olawlor olawlor@acm.org

1 2005/02/02 Lecture Notes

1.1 x86 Assembly Language

The Linux standard compiler GCC and assembler GAS use the UNIX or "AT&T" assembly convention: the destination is listed last. Thus under GCC,

```
mov %eax,%ebx
```

would copy the value in register eax into the register ebx. Register names are always prefixed by a percent sign in the UNIX convention, and in gcc inline assembly are prefixed by two percent signs. The GNU tool "objdump -d " is very useful to disassemble any binary file (.o object file, static or dynamic library, executable, or even Windows .exe!).

Microsoft Visual C++ and NASM, by contrast, use the Intel assembly convention: the destination register is listed first. Thus under NASM or Visual C++,

```
mov eax,ebx
```

would copy the value in ebx into eax!

1.2 x86 Subroutine Calls

A typical x86 call sequence is:

1. Caller pushes arguments onto the stack, starting from the rightmost argument. This means the first (leftmost) argument is sitting on the top of the stack.

2. Caller executes the `CALL` instruction, which pushes the caller's program counter and jumps to the given subroutine. This normally looks like:

```
call   804838c <foo>
```

3. A subroutine usually begins by setting up a "frame pointer", which is a register (always ebp) that points into the stack and is used to access arguments and local variables. This data could also be accessed relative to the stack pointer, but because the stack pointer keeps changing (e.g., as arguments are pushed on and popped off), it's a bit less confusing to access everything relative to the frame pointer. The compiler doesn't get confused, and hence can be instructed not to bother with the frame pointer—the gcc argument for this is "–fomit-frame-pointer", and this normally speeds things up a tad.

   The assembly code you normally see at the start of any routine to set up the frame pointer is:

```
push   %ebp
mov  %esp,%ebp
sub  $0x8,%esp
```

   The "push" saves the old frame pointer; the "mov" sets up the frame pointer ebp; and the "sub" makes some room on the stack (here, eight bytes) for the routine's local variables. Note that the stack grows down, so "sub" really does make room on the stack (for a regular array, which grows up, you'd use "add"!).

4. The subroutine then does whatever it needs to do. It access its arguments at positive offsets from ebp, and accesses local variables at negative offsets.

5. Before it finishes, a subroutine restores the frame pointer using the `LEAVE` instruction. This restores esp from ebp, and pops the old ebp. This is hence the opposite of the frame pointer setup at the start of the subroutine. The subroutine

6. The subroutine returns by executing the `RET` instruction, which pops the old program counter from the stack and starts executing there. Note that this is corresponds to the `CALL`, which pushed the old program counter before entering the routine. This normally returns control back to the calling routine (unless the stack has been corrupted!).

7. In C, the calling routine has to pop its arguments back off the stack. It usually does this by just adding a constant to the stack pointer, like this:

   ```
   add $0x10,%esp
   ```