Lab 6: Assembly

Lab sessions Tue Nov 07 to Thu Nov 09

Lab written by Julie Zelenski

Learning goals

This lab is designed to give you a chance to:

1. study the relationship between C source and its assembly translation
2. use `objdump` and `gdb` to disassemble and trace assembly code
3. write a little assembly of your own

Find an open computer and somebody new to sit with. Share your joy, angst, and relief from last week's midterm.

Get started

Clone the repo by using the command below to create a lab6 directory containing the project files.

```
git clone /afs/ir/class/cs107/repos/lab6/shared lab6
```

Open the lab checkoff form (https://web.stanford.edu/class/cs107/cgi-bin/lab6) .

Lab exercises

When doing code study on assembly language, we recommend you use the nifty GCC Explorer online interactive compiler. Use this link https://godbolt.org/g/WAq2MM (https://godbolt.org/g/WAq2MM) we have pre-configured for myth's version of GCC and compiler flags from the CS107 makefiles. (To manually configure GCC Explorer: compiler is `x86-64 gcc 4.8.5` , flags are `-Og -std=gnu99 -xc`).

In GCC Explorer, you can type in a C function, see its generated assembly, then tweak the C source and observe how those changes are reflected in the assembly instructions. You could instead make these same observations on myth using gcc and gdb, but GCC Explorer makes it easier to do those tasks in a convenient environment for exploration. Try it out!

A note on register use: Lecture has not yet reached the assembly for function calls and how registers are used for passing values in and out of functions, so here is a quick preview. The first three arguments are passed in `%rdi` , `%rsi` , and `%rdx` respectively and the return value is written to `%rax` . In the body of the function, these registers (and a few others) are used to hold intermediate results. A reference to the `e` -named register accesses the lower 32-bit subregister, e.g. `%edi` is the subregister of `%rdi` . Operations on pointers and longs (64-bit types) will use the full `r-` named registers, and operations on ints will use the `e` -named subregisters.

1) Assembly code study

Addressing modes

Review the two `deref` functions below.

```
void deref_one(char *ptr, long index)
{
    *ptr = '\0';
}

void deref_two(int *ptr, long index)
{
    *ptr = 0;
}
```

Open GCC Explorer https://godbolt.org/g/WAq2MM (https://godbolt.org/g/WAq2MM) and copy and paste the above functions into the left pane and examine the generated assembly shown in the right pane.

1. There is one difference between the assembly instruction sequences for the two functions as shown above. What is the difference? Why it is different?

2. Edit the first function to add the typecast `*(float *)ptr` to the assignment statement and the two assembly sequences become the same. But the C source for the two functions exhibit clearly different intentions! How is possible that both can generate the same assembly instructions?
3. Edit both functions to instead assign `ptr[7]` (no typecasts) to zero. How does this change the addressing mode using for the destination operand of the `mov` instruction? Do both `deref` functions change in the same way?
4. Edit both functions to now assign `ptr[index]` to zero. How does this change the addressing mode using for the destination operand of the `mov` instruction? Do both `deref` functions change in the same way?
5. Change the assignment statement to `*ptr = *(ptr + 1)`. For both functions, the assembly sequence is one instruction longer. Previously, only one `mov` instruction was sufficient to implement the assignment statement. This assignment statement requires two instructions. Why?

Signed/unsigned arithmetic

The following two functions perform the same arithmetic operation on their arguments, but those arguments differ in type (signedness).

```
int s_arith(int a, int b)
{
    return (a - b) * a;
}

unsigned int u_arith(unsigned int a, unsigned int b)
{
    return (a - b) * a;
}
```

Copy and paste the following code into GCC Explorer.

1. Observe that both functions use the same assembly instructions for the arithmetic. The choice of two's complement representation allows the exact same add/sub/imul instruction to work for both unsigned and signed types. Neat!
2. Edit both functions to perform a right shift on one of its arguments. When doing a right-shift, does gcc emit an arithmetic (`sar`) or logical (`shr`) shift?

Lea, multiplication by a constant

The `lea` instruction packs a lot of computation in one instruction: two adds and a multiply by constant 1, 2, 4 or 8. It was designed for address arithmetic, but the math is compatible with regular integer operations and it is often used by compiler to do an efficient add/multiply combo.

```
int combine(int x, int y)
{
    return x + y;
}

int scale(int x)
{
    return x * 4;
}
```

Copy and paste the above functions into GCC Explorer.

1. Look at the generated assembly for `combine` and you'll see a `lea` instead of the expected `add`. Interesting! `lea` can be used for `add`, but what else can it do?
2. Edit the `combine` function to `return x + 2*y` or `return x + 8*y -17` and observe how a single `lea` instruction can also compute these more complex expressions.
3. Edit to `return x + 23*y` and the result will no longer fit the pattern for an `lea`. What sequence of assembly instructions is generated instead?
4. Multiply is one of the more expensive instructions and the compiler prefers cheaper alternatives where possible. The `scale` function multiplies its argument by 4. Look at its generated assembly-- what instruction does the compiler use? Edit the `scale` function to multiply by 16. What assembly instruction is used now?
5. It is perhaps unsurprising that the compiler treats multiplication by powers of 2 as a special case given the underlying binary representation, but it's got a few more tricks up its sleeve than just that! Edit the `scale` function to instead multiply its argument by a small constant that is not a power of 2. Try 3, then 17, and then 25. For each, look at the

assembly and see what instructions are used. Gcc sure goes out of its way to avoid multiply!

6. Experiment to find a small integer constant C such that `return C*x` is expressed as a true `imul` instruction.

Constant folding

Below are some functions that we examined earlier in the quarter. Each has a complex expression that involve only constants. The `abs_val` function could have right-shifted `x` by a hard-coded `31` or `has_zero_byte` might have assigned `highs` to a hard-coded `0x8080808080808080`), but instead the functions build up these values. I find this approach pleasing as it documents the construction of these values as opposed to dropping in some magic number. An objection might be that repeatedly calculating the value incurs a performance penalty, but no need to worry! An expression that involves only constants can be evaluated at compile-time and have its result substituted into the generated assembly. This compiler feature is called *constant folding*.

```
#include <limits.h>

int abs_val(int x)
{
    int sn = x >> (sizeof(int)*CHAR_BIT -1);
    return (x ^ sn) - sn;
}

int has_zero_byte(unsigned long val)
{
    unsigned long ones = ~0UL/UCHAR_MAX;
    unsigned long highs = ones << (CHAR_BIT - 1);
    return (((val - ones) & highs) & ~val) != 0;
}

int constant_fold(int x)
{
    return 10 + ((22*45 + 48)*13 - 100)*x - 1;
}
```

1. Copy and paste the above functions into GCC Explorer and look at the generated assembly instructions. Can you see where constant folding was applied? If you hover over the large decimal constants in the assembly for `has_zero_byte` , GCC Explorer will show the hex equivalent, which will make the constants less inscrutable.

2) Tools

Deadlisting with `objdump`

As part of the compilation process, the assembler takes in assembly instructions and encodes them into binary machine form. *Disassembly* is the reverse process that converts binary-encoded instructions back into human-readable assembly text. `objdump` is a tool that operates on object files (i.e. files containing machine instructions in binary). It can dig out all sorts of information from the object file, but one of the more common uses is as a disassembler. Let's try it out!

1. Invoking `objdump -d` extracts the instructions from an object file and outputs the sequence of binary-encoded machine instructions alongside the assembly equivalent. This dump is called a deadlist ("dead" to distinguish from the study of "live" assembly as it executes). Use `make` to build the lab programs and then `objdump -d code` to get a sample deadlist.
2. The `countops.py` python script reports the most heavily used assembly instructions in a given object file. Try out `countops.py code` for an example. The script uses `objdump` to disassemble the file, tallies instructions by opcode, and reports the top 10 most frequent. Use `which` to get the path to a system executable (e.g. `which vim` or emacs or gcc) and then use `countops.py` on that path. Try this for a few executables. Does the mix of assembly instructions seem to vary much by program?

GDB commands for live assembly debugging

Below we introduce a few of the gdb commands that allow you to work with code at the assembly level.

The gdb command `disassemble` with no argument will print the disassembled instructions for the currently executing function. You can also give an optional argument of what to disassemble, a function name or code address.

```
(gdb) disass myfn
Dump of assembler code for function myfn:
   0x0000000000400651 <+0>: push   %rbx
   0x0000000000400652 <+1>: sub    $0x20,%rsp
   0x0000000000400656 <+5>: movl   $0x1,(%rsp)
...
```

The hex number in the leftmost column is the address in memory for that instruction and in angle brackets is the offset of that instruction relative to the start of the function.

You can set a breakpoint at a specific assembly instruction by specifying its address `b *address` or an offset within a function `b * myfn+8` . Note that the latter is not 8 instructions into main, but 8 *bytes* worth of instructions into main. Given the variable-length encoding of instructions, 7 bytes can correspond to one or several instructions.

```
(gdb) b *0x400609      break at specified address
(gdb) b *myfn+8       break at instruction 8 bytes into myfn()
```

The gdb commands `stepi` and `nexti` allow you to single-step through assembly instructions. These are the assembly-level equivalents of the source-level `step` and `next` commands. They can be abbreviated `si` and `ni` .

```
(gdb) stepi            executes next single instruction
(gdb) nexti            executes next instruction (step over fn calls)
```

The gdb command `info reg` will print all integer registers. You can print or set a register's value by name. Within gdb, a register name is prefixed with `$` instead of the usual `%` .

```
(gdb) info reg
rax            0x4005c1 4195777
rbx            0x0  0
....
(gdb) p $rax            show current value in %rax register
(gdb) set $rax = 9      change current value in %rax register
```

The `tui` (text user interface) we showed in lecture splits your session into panes for simultaneously viewing the C source, assembly translation, and/or current register state. The gdb command `layout <argument>` starts tui mode. The argument specifies which pane you want (`src` , `asm` , `regs` , `split` or `next`).

```
(gdb) layout split
```

Tui mode is great for tracing execution and observing what is happening with code/registers as you `stepi` . Occasionally, tui trips itself and garbles the display. The gdb command `refresh` sometimes works to clean it up. If things get really out of hand, `ctrl-x a` will exit tui mode and return you to ordinary non-graphical gdb.

Reading and tracing assembly in GDB

Let's try out all these new gdb commands! Read over the program `code.c` . Compile the program and load into gdb.

1. Compile the program and load into gdb. Disassemble `myfn` .
2. Use the disassembly to figure out where `arr` is being stored. How are the values in `arr` initialized? What happened to the `strlen` call on the string constant to init the last array element?
3. What instructions were emitted to compute the value assigned to `count` ? What does this tell you about the `sizeof` operator?
4. Set a breakpoint at `myfn` and run the program. Use the gdb command `info locals` to show the local variables. Compare this list to the declarations in the C source. You'll see some variables are shown with values ("live"), some are `<optimized out>` , but others don't show up at all. Look at the disassembly to figure out what happened to these entirely missing variables. Step through the function repeating the `info locals` command to observe which variables are live at each step. Examine the disassembly to explain why there is no step at which both `total` and `squared` are live.

3) Writing assembly

The program `code` in the lab repo is written in a mix of C and assembly code. The file `code.c` has C source, the file `asm.s` has assembly instructions, and those two files are compiled/assembled and then linked together in one executable. There is no C source for the asm side, so the only means to understand what that code accomplishes is to read the assembly itself.

1. The program `code` calls the function `sample` on its command-line arguments and prints the result. The `sample` function was written in assembly, not compiled from C. Open the `asm.s` file to see its definition. Run in gdb and `stepi` through the execution of a call to `sample` and observe its operation. Once you understand what it's doing, jot down an equivalent C version of the `sample` function.
2. And lastly, try out hand-generating a little assembly by editing the `asm.s` file to implement your own version of the `mine` function to return the absolute value of the difference between its two arguments. Compile and test with sanitycheck to verify that your assembly code is correct.

Check off with TA

Before you leave, be sure to submit your checkoff sheet and have your lab TA come by and confirm so you will be properly credited. If you don't finish everything before lab is over, we strongly encourage you to finish the remainder on your own. Double-check your progress with self check (/class/cs107/selfcheck.html#lab6).