COMP 264 Fall 2015 Lecture Notes

R. I. Greenberg

October 07, 2015

1 Machine-Level Representation of Programs

1.1 Getting High-Level Language Programs to Run on Specific Machines

- We've seen `gcc -o hello hello.c` to go through 4 phases, of preprocessing, compilation, assembly, and linking. (Could also have C program spread across multiple files and can allow compiler to do a specified level of optimizations, e.g., `gcc -O2 -o prog part1.c part2.c`. Here `.c` files are individually transformed to `.i`, `.s`, and `.o`; then linker combines with each other and library functions to produce executable `prog`.)

- Biggest step is compilation. Assembly code output by compiler is essentially same as the machine code output by the assembler but in a more readable textual form, so this is our focus for "machine-level code".

- We'll focus on gcc assembly code GAS for the Intel processor line that has gone under various names, e.g., "IA32" and then "Intel64", or the colloquial "x86". We will focus on the now-dominant 64-bit version that we refer to as `x86-64` extension(Can see some history in *04-machine-basics slides* 3–11.)

An aside: Sometimes code is interpreted rather than compiled (or there may even be an in-between approach). We can talk about that a bit in class.

1.2 Assembly Language Instructions

We'll see some x86-64 assembly language details soon; first we discuss some general features of assembly languages.

1.2.1 Instruction Types

One way to classify the main types of instructions is as follows:

- Data Transfer Instructions: Move (really copy) data from one location to another. Some varieties are as follows:
  - Load data from memory into the microprocessor.
  - Store data from the microprocessor into memory.
  - Input data to the microprocessor from an external device (e.g., keyboard, mouse).
  - Output data from the microprocessor to an output device (e.g., printer, monitor).

  The above four categories of instructions may be split into two groups (I/O for the latter two bullet items, and regular data transfer for the first two). (Note that moving data within the microprocessor is best viewed as a degenerate case of the data operation instructions discussed next.)

- Data Operation Instructions: Typically perform some operation using one or two data values (operands) and store the result. Some examples are:
  - Integer arithmetic instructions such as add, subtract, multiply, or divide.
  - Floating point arithmetic instructions.
  - Logic instructions such as bitwise AND, OR, or XOR of two data values, or negation of a single value.
  - Shifts (can be left or right, and circular, arithmetic, or logical).

- Program Control Instructions: Transfer execution from one place in the program to another. High-level languages generally provide structured control such as for loops, while loops, etc. Assembly languages provide more elementary mechanisms such as the unconditional jump and the conditional (e.g., if and only if a certain register contains 0) jump. (Some authors distinguish between unconditional and conditional by referring to them as "jumps" and "branches", respectively.)

  Typically also instructions to call subroutines and return from them. Main difference from ordinary jumps is that when a subroutine is called, the program location of the call is remembered so that the next return from the subroutine will go to that location. This is necessary so that the same subroutine can be called from many different locations.

  A microprocessor may be designed to accept *interrupts* (temporarily stop what doing and execute other instructions). May include hardware interrupts triggered by external devices (e.g., keyboard), as well as software interrupts generated by special instructions in the assembly language. *Exceptions* or *traps* are a similar situation, where a special handling routine is invoked to deal with valid instructions that perform invalid operations (e.g., division by 0 or referring to an out-of-bounds memory location)

The halt instruction is also rather trivial control instruction to tell the microprocessor that the end of the program has been reached.

There may also be other types of instructions, e.g., data operation instructions that do string comparisons and manipulations, or specialized I/O instructions, e.g., for graphics.

1.2.2 Data Types

The assembly language may include different instructions to perform the same kind of operation on different data types such as the following:

- Unsigned integers (in the range 0 to $2^n - 1$ for an n-bit value)
- Signed integers (in the range -2^{n-1} to $2^{n-1} - 1$ for an n-bit value)
- Floating point numbers
- Boolean values, e.g., 0 for FALSE, and 1 for TRUE.
- Characters encoded using ASCII (traditionally usual), EBCDIC, or UNICODE (newer 16-bit standard facilitating a larger character set)

1.2.3 Addressing Modes

When an instruction specifies data to transfer or operate on or a program location to which control should pass, there are several different addressing modes that may be used to specify this *operand*. Examples for our assembly language below.

1.2.4 Instruction Formats

When assembly language instructions are translated into machine language, the instruction is represented by a binary encoding in which different groups of bits represent different parts of the instruction. Usually, there is a specific location that is the same for all instructions that provides the *opcode* (or at least the "primary opcode"), which gives the information (or most of it) as to what type of operation is to be performed. Other fields select operands for the instruction. Modes of operands may also be selected by additional fields in the instruction, or they may be determined by the opcode. What fields appear and how many bits are in each may be a uniform specification across all instructions, or there may be different formats for different instructions, the different instructions being recognizable from the opcode.

1.2.5 C Code Example with Assembly Language

Warning: Routine compilations on Loyola machines (shannon), which are 64-bit machines, produce assembly language vastly different than what you would get on one of our older machines like `xenon` and what you will see in our text. To get older style code, we'll use the -m32 flag in the compilation.

Consider the following C code from the file `exchange.c`:

```
long exchange(long *xp, long y) {
  long x = *xp;
  *xp = y;
  return x;
}
```

Reviewing the blue aside in textbook section 3.4.3 will help you understand the use and workings of this procedure and, more generally, improve your facility with the C `&` and `*` operators.

Compiling with just the `-S` flag on a machine like shannon, we get this in `exchange.s`:

```
.file "exchange.c"
.text
.globl exchange
.type exchange, @function
exchange:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movq %rdi, -24(%rbp)
movq %rsi, -32(%rbp)
movq -24(%rbp), %rax
movq (%rax), %rax
movq %rax, -8(%rbp)
movq -24(%rbp), %rax
movq -32(%rbp), %rdx
movq %rdx, (%rax)
movq -8(%rbp), %rax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size exchange, .-exchange
.ident "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
.section .note.GNU-stack,"",@progbits
```

This is a bit of a complicated version with no optimizations. Note that, in general, when we compile things we may get somewhat different results than in the book, since there are many different versions of the gcc compiler as it continuously undergoes tuning to generate more efficient code. With the `-O2` option added to get a typically desirable level of optimization, we get the following assembly code:

```
.file "exchange.c"
.text
.p2align 4,,15
.globl exchange
.type exchange, @function
exchange:
.LFB0:
.cfi_startproc
movq (%rdi), %rax
movq %rsi, (%rdi)
ret
.cfi_endproc
.LFE0:
.size exchange, .-exchange
.ident "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
.section .note.GNU-stack,"",@progbits
```

Before we examine the assembly code more closely, note also that there is a handy facility available to recover assembly code from object code. After `gcc -O2 -c exchange.c` creates `exchange.o`, we can do `objdump -d exchange.o` to get:

```
exchange.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <exchange>:
0:48 8b 07              mov    (%rdi),%rax
3:48 89 37              mov    %rsi,(%rdi)
6:c3                    retq
```

Same as `.s` except (not all points exemplified in this short code fragment):

- Omits directives to be used by the assembler and linker.
- Shows hexadecimal encoding of instructions and memory offsets.
- Constants in the assembly code are in hex instead of decimal.
- Instruction names are written without the "q" suffix, except "q" added to "ret" and "call".

Assembly code in our text is typically annotated to explain something of what is going on in relation to the original C program, e.g., for the inner part of `exchange.s` with the `-O2` optimizations:

```
# long exchange(long *xp, long y)
```

```
# xp in %rdi, y in %rsi
exchange:
        movq    (%rdi), %rax    # Get x at xp. Set as return value.
        movq    %rsi, (%rdi)    # Store y at xp.
        ret                     # Return
```

Figures 3.1–6, and 3.8–10 explain most of what we need to understand assembly code.

Figure 3.1 shows us that instructions ending in "q" operate on "quad words". Careful!! The quad word terminology is based on the origins of this processor family as a 16-bit processor, so a quad word is 64 bits. Other suffixes for integer ops.: "b" for byte, "w" for a "word" of 2 bytes, "l" for a "double word" of 4 bytes. Suffixes for floating point ops: "s" for SP (4 bytes) and "l" for DP (8 bytes).

Figure 3.2 shows there are sixteen 64-bit (quad word) registers called (different naming groups based on historical use and development):

- `%rax`, `%rbx`, `%rcx`, `%rdx`
- `%rsi`, `%rdi`
- `%rbp`, `%rsp`(still reserved for special purposes: 'frame pointer" and "stack pointer")
- `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`

Also can operate on just lower order 32 bits:
`%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, `%esp`, `%r8d`,, `%r9d`, `%r10d`, `%r11d`, `%r12d`, `%r13d`, `%r14d`, `%r15d` or lower 16 bits:
`%ax`, `%bx`, `%cx`, `%dx`, `%si`, `%di`, `%bp`, `%sp`, `%r8w`,, `%r9w`, `%r10w`, `%r11w`, `%r12w`, `%r13w`, `%r14w`, `%r15w` or lower 8 bits:
`%al`, `%bl`, `%cl`, `%dl`, `%sil`, `%dil`, `%bpl`, `%spl`, `%r8b`,, `%r9b`, `%r10b`, `%r11b`, `%r12b`, `%r13b`, `%r14b`, `%r15b` .
Operations that set just the lower byte or lower two bytes leave the rest of the register unchanged, but part of the convention set for the transition from IA32 to x86-64 is that operations that set the bottom four bytes will also zero out the upper four bytes.

Figure 3.3 shows different addressing modes for operands. Most general form: $Imm(r_b, r_i, s)$ means $\mathsf{M}[Imm + \mathsf{R}[r_b] + \mathsf{R}[r_i] \cdot s]$. Some important special or variant cases:

- $\$Imm$ for Imm ("Immediate")
- r_a for $\mathsf{R}[r_a]$ ("Register")
- Imm for $\mathsf{M}[Imm]$ ("Absolute")
- (r_a) for $\mathsf{M}[\mathsf{R}[r_a]]$ ("Indirect")
- $Imm(r_a)$ for $\mathsf{M}[Imm + \mathsf{R}[r_a]]$ ("Displacement)

Can classify into three broad categories: immediate, register, and memory.

An actual sample instr.:

```
leaq 9(%rcx,%rdx,2), %rbx
```

will set `%rbx` to "effective address" of the operand, i.e., twice the content of `%rdx` plus the content of `%rcx` plus 9. `leaq` (forthcoming in Figure 3.10) is an exceptional instruction in that it uses the computed memory address as the operand, whereas any other instruction involving a memory addressing mode would get the operand from the indicated location in memory. For example,

```
movq 9(%rcx,%rdx,2), %rbx
```

will compute the same effective address as in the `leaq` example, call it A, and then it will copy $M[A]$ to `%rbx`.

Figures 3.4–6 shows the data movement instrs. The most basic ones, to which we can limit most of our attention, are in Figure 3.4, whereas Figures 3.5 and 3.6 include instructions for moving smaller data items into larger ones with zero-extension of with sign-extension. Even `movq` in Figure 3.4 sometimes involves sign-extension, though, since the largest immediate value that can be used is 32 bits (and the `movabsq` instruction at the end of Figure 3.4 is for working with a 64-bit immediate). In general, the `movb`, `movw`, `movl`, and `movq` instructions can have operands that are immediate, in a register, or in memory, but can't have both in memory. (Can see slide 27 of *04-machine-basics slides* for move illustrations.)

The version of the `exchange` assembly code in which we did not use the `-O2` flag used `pushq` and `popq` as well as movq. The push and pop instructions manipulate the stack (*asm/stack.pdf*).

(Can see in *04-machine-basics slides* 29–38. an illustration of assembly code for a `swap` procedure similar to our text's `exchange`.)

1.2.6 Arithmetic and Logical Operations

Key integer arithmetic opers. in Figure 3.10 (and most in *05-machine-control slides* 7–8). `leaq` doesn't actually access memory but computes an addr. to store in a register. Next are unary operations (source=dest). Next are binary ops. (Second op. is a source as well as dest.; order of ops. makes sense from standpoint of S being used to do something to D.) Last group is shifts; both arithmetic and logical right shifts available (are same for left shifting). Except for `leaq`, all the entries in this table actually stand for a class of operations with different operand sizes. For example, there is `incb`, `incw`, `incl`, and `incq`. (Shifts have a special rule that the first operand, the no. of bits by which the second operand is shifted, must be an immediate operand or the register `%cl`, and only the lowest-order bits are actually used so that the shift amount will be less than the number of bits in the second operand.)

Also, some multiplication and division operations in Figure 3.12. Actually had `IMUL` before in Figure 3.10, where `imulq` would multiply two 64-bit values to produce a 64-bit value, but there could be trunctation in the result. Here mechanisms are provided for

128-bit operations, using `%rdx` (high-order part) and `%rax` (low-order part) together when needing to represent a 128-bit value. The name `imulq` is reused here — the assembler can recognize this form from the presence of just one operand; the other source operand implicitly is `%rax`. New instruction names also are provided for unsigned full multiplication and for signed and unsigned division with implicit dividend formed by `%rdx` and `%rax`, and quotient and remainder going into `%rax` and `%rdx`, respectively.

2 Appendix: Instruction Set Architecture Design

An obviously important consideration in instruction set design is *completeness*, which is the straightforward idea that the ISA must incorporate sufficiently many instructions and registers to make the computer useful for the tasks it is intended for. Conversely, it is desirable to avoid excessive, redundant functionalities that make the implementation more complicated. Another feature that is pleasant for the compiler writer but may complicate the hardware is *orthogonality*.

Orthogonality means that different features are independent, for example, allowing every instruction to use every addressing mode for each of its operands. A higher degree of orthogonality means more flexibility for compiling code into the relevant machine language but more complexity in the implementation of the machine language. To give a better sense of the concept of orthogonality, here are some examples of non-orthogonality at the high-level language level, specifically in C. I've included a few related comments as well, all from `http://mindprod.com/jgloss/orthogonal.html`.

> Here are some examples of non-orthogonality in C:
>
> 1. C has two kinds of built in data structures, arrays and records (structs). Records can be returned from functions, but arrays cannot.
> 2. A member of a struct can have any type except void or a structure of the same type.
> 3. An array element can be any data type except void or a function.
> 4. Parameters are passed by value, unless they are arrays, in which case they are passed by reference.
> 5. a+b usually means that they are added, but if a is a pointer, the value of b may be changed before the addition takes place.
>
> Non-orthogonality means exceptions to the general language rules, which make it harder to learn. It means that you cannot combine language features in all possible ways. Excessive orthogonality makes it possible to say silly things in the language that complicate the compilers.