CSO: Recitation 7

Goktug Saatcioglu

10.16.2019

1 Assembly Programmer's View

- Execution context
  - PC: Program counter
    - Address of next instruction: Called "RIP" (x86-64)
  - Registers
    - Heavily used program data
  - Condition code registers
    - Store status information about most recent arithmetic or logical operation
    - Used for conditional branching
- Memory
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

2 Assembly Data Types And Kinds Of Operations

- "Integer" data of 1, 2, or 4 bytes
  - Represent either data values
  - or addresses
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No arrays or structures
- 3 Kinds of Assembly Instructions
  - Perform arithmetic on register or memory data
    - Add, subtract, multiply, ...
  - Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches

3 Object Code

- Assembler
  - Translates .s (assembly code) into .o (object code)
  - Binary encoding of each instruction
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for malloc, printf
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution

4 x86-64 Integer Registers

- 8-bytes, low order 4-bytes, low-order 2-bytes, high-order of 1 byte of low-order 2 bytes, low-order 1 byte of low-order 2 bytes
- `%rax`, `%eax`, `%ax`, `%ah`, `%al`
- `%rbx`, `%ebx`
- `%rcx`, `%ecx`
- `%rdx`, `%edx`
- `%rsi`, `%esi`
- `%rdi`, `%edi`
- `%rsp`, `%esp`
- `%rbp`, `%ebp`
- `%r8`, `%r8d`
- `%r9`, `%r9d`
- `%r10`, `%r10d`
- `%r11`, `%r11d`
- `%r12`, `%r12d`
- `%r13`, `%r13d`
- `%r14`, `%r14d`
- `%r15`, `%r15d`

5 Moving Data

- `movq Source, Dest`
- Operand types:
  - Immediate: Constant integer data
    - Example: `$0x400`, `$-533`
    - Like C constants, but prefixed with `$`
  - Register: One of 16 integer registers
    - Example: `%rax`, `%r13`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - Memory: 8 consecutiev bytes of memory at address given by register
    - Example: `(%rax)`
    - There are various other address modes.

5.1 movq Operand Combinations

- Source: Immediate
  - Destination: Register = `movq $0x4,%rax`, C analog = `temp = 0x4;`
  - Destination: Memory = `movq $-147,(%rax)`, C analog = `*p = -147;`
- Source: Register
  - Destination: Register = `movq %rax,%rdx`, C analog = `temp2 = temp1;`
  - Destination: Memory = `movq %rax,(%rdx)`, C analog = `*p = temp;`
- Source: Memory
  - Destination: Register = `movq (%rax),%rdx`, C analog = `temp = *p;`
- Notice how there is No memory-to-memory instruction. Can you think why? One reason is that there is no memory to memory instruction in C. Consider the following: `*p = *x`. This may seem like copying the contents of one memory location to another memory location it is actually two instrucitons. One we must retrieve the contents of a memory location and put it in a register. Then we must take these contents and put it in the new memory location.
- The `q` in `movq` stands for quad-word. A quad-word is 8 bytes, a double-word is 4 bytes and a word is 2 bytes. This corresponds to pointer, long, int and short in C correspondingly. A byte is a char. We have corresponding suffixes in assembly: `l`, `w` and `b`.

6 Special Type of mov

- `movz S,R` $\rightarrow$ `R = ZeroExtend(S)`
  - `movzbw` (zero extend byte to word)
  - `movzbl` (zero extend byte to double word)
  - `movzbq` (zero extend byte to quad word)
  - `movzwl` (zero extend word to double word)
  - `movzwq` (zero extend word to quad word)
- `movs S,R` $\rightarrow$ `R = SignExtend(S)`

  - `movsbw` (sign extend byte to word)
  - `movsbl` (sign extend byte to double word)
  - `movsbq` (sign extend byte to quad word)
  - `movswl` (sign extend word to double word)
  - `movswq` (sign extend word to quad word)
  - `movslq` (sign extend double word to quad word)
- `S`: memory or register `R`: register

7 Simple Memory Addressing Modes

- Normal: `(R)` → `Mem[Reg[R]]`
  - Register `R` specifies memory address to read from/write to
  - `movq (%rcx),%rax`
  - Useful for normal memory accesses.
- Displacement: `D(R)` → `Mem[Reg[R]+D]`
  - Register `R` specifies start of memory region
  - Constant displacement `D` specifies offset in bytes
  - `movq 8(%rbp),%rdx`
  - Useful for accessing struct fields. Why? We know the fixed size of the struct and its fields have fixed offset from the base address. So constant displacement access will let us access a field.

8 General Memory Addressing Modes

- Most general form: `D(Rb, Ri, S)`
  - `D` is some constant displacement, aka offset, cannot be larger than 4 bytes
  - `Rb` is base register (any of the 16 registers)
  - `Ri` is index register (any of the 16 registers except `%rsp`)
  - S is scale $(1, 2, 4, 8)$ (Why only these scale values?)
- So we get: `Mem[Reg[Rb] + S*Reg[Ri] + D]`
- Special cases:
  - `(Rb,Ri)` = `Mem[Reg[Rb] + Reg[Ri]]`
  - `D(Rb,Ri)` = `Mem[Reg[Rb] + Reg[Ri] + D]`
  - `(Rb,Ri,S)` = `Mem[Reg[Rb] + S*Reg[Ri]]`
- Example: Assume `%rdx` = `0xf000` and `%rcx` = `0x0100`
  - `0x8(%rdx)` = `0xf000 + 0x8` = `0xf008`. What kind of calculation can this be? Getting the next part (field) of some structure is one use.
  - `(%rdx,%rcx)` = `0xf000 + 0x100` = `0xf100` What kind of calculation can this be? We can access some memory location we know is at some fixed offset from a base address such as a local variable on the stack.
  - `(%rdx,%rcx,4)` = `0xf000 + 4*0x100` = `0xf400` What kind of calculation can this be? Accessing the i-th element of an integer array where `%rcx` corresponds to i and 4 is the size of an integer. `%rdx` is the base address.

- `0x80(,%rdx,2)` = `2*0xf000 + 0x80` = `0x1e080`. What kind of calculation can this be? It's actually not immediately clear. One possibility could be a jump instruction (using goto) or a function call where we know where the next function is. Another possibility is that we are accessing something again at a fixed offset and this is how we calculated the memory of whatever is being accessed.

9 Address Computation Instruction

- `leaq Src,Dst`
  - `Src` is address mode expression
  - Set `Dst` to address calculated for `Src`
- Uses:
  - Computing addresses without a memory access, e.g. translation of `p = &x[i]`;
  - Computing arithmetic expressions of the form `x + k*y`, `k = 1,2,4,8`
- Example:

```
long m12(long x)
{
  return x*12;
}
```

gets converted to

```
leaq (%rdi,%rdi,2), %rax     # t = x+x*2
salq $2, %rax                # return t<<2
```

10 Arithmetic and Logic Operations

- Two operand instructions:
  - `addq Src,Dest` = `Dest = Dest + Src`
  - `subq Src,Dest` = `Dest = Dest - Src`
  - `imulq Src,Dest` = `Dest = Dest * Src`
  - `salq Src,Dest` = `Dest = Dest << Src` (also called `shlq`)
  - `sarq Src,Dest` = `Dest = Dest >> Src` (arithetmic)
  - `shrq Src,Dest` = `Dest = Dest >> Src` (logical)
  - `xorq Src,Des` = `Dest = Dest ^ Src`
  - `andq Src,Dest` = `Dest = Dest & Src`
  - `orq Src,Des` = `Dest = Dest` Src—
- Argument order matter!
- There is no distinction between signed and unsigned int. Why is this so? At the machine level, with 2's complement everything can be done with normal arithmetic.
- One operand instructions:
  - `incq Dest` = `Dest = Dest + 1`
  - `decq Dest` = `Dest = Dest - 1`
  - `negq Dest` = `Dest = - Dest`
  - `notq Dest` = `Dest = ~Dest`

11 Exercise 0

What is the difference between `leaq -0x18(%rbp), %rbx` and `movq -0x4(%rbx), %rax`? `leaq` sets `Dst` to the address computed by the `Src` expression but it does not actually go to the memory to retrieve the contents at that address. So `leaq` actually loads the result of `-0x18(%rbp)=-0x18+R[%rbp]` into `%rbx`. On the other hand, `movq` will actually retrieve the contents of the address computed by the `Src` expression. So, in this case, we get `Mem[-0x18(%rbp)]=Mem[-0x18+R[%rbp]]` and load it into `%rbx`. As a side note, `leaq` stands for load effective address `lea` and `q` is the usual quad.

12 Exercise 1

Let's input the following C program into IntelliJ.

```c
#include <stdio.h>

void swap (long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

int main() {
    long x = 5;
    long y = 10;

    printf("x is %ld and y is %ld\n", x, y);
    swap(&x, &y);
    printf("now x is %ld and y is %ld\n", x, y);

    return 0;
}
```

It's a basic C program that swaps two integers. Now, we wish to take a look at the generated assembly code. Let's begin by setting a breakpoint at the function call to swap. Then run the debugger. Once the debugger hits the break point, switch to the gdb or lldb window and type `disassemble`. Upon hitting enter you will see a disassembly of your code. Can you describe what's happening? What direction is the stack growing in? What parts are the swap happening? If you are having trouble at first, that is fine! Reading assembly and understanding it are two seperate things. You should also try adding more breakpoints. Notice how after you step through, if you run the `disassemble` command again, the arrow on the left of the output will have moved. The debugger is telling you where in the program you are currently at. Use that to your advantage. Since outputs may change from machine to machine, no solution is given here. But if you followed along during recitation, then you should be able solve this problem locally at home too.