CSO: Recitation 9

Goktug Saatcioglu

10.31.2019

1 Review of Midterm

Done in class.

2 Continuing Assembly

2.1 Multiplication

We have the following multiplication instructions:

- Unsigned
  - Form 1: `mulq s, d`, do $s \times d$ and put the result in the 64-bit operand, i.e. d
  - Form 2: `mulq s`, do $s \times \%rax$ and put the higher order part in `%rdx` and the lower order part in `%rax` (so we get the full 128-bit result)
- Signed
  - Form 1: `imulq s, d`, do $s \times d$ and put the result in the 64-bit operand, i.e. d
  - Form 2: `imulq s`, do $s \times \%rax$ and put the higher order part in `%rdx` and the lower order part in `%rax` (so we get the full 128-bit result)

Exercicse: Why might we need 128-bits if we multiply two 64-bit numbers (hint: think grade-school multiplication)?

2.2 Divison

We have the following division instructions:

- Unsigned
  - `divq s`, do $(\%rdx + \%rax) \div s$ where the dividend is given as `%rdx` for higher-order part and `%rax` for the lower order part, divisor is `s`, quotient is stored in `%rax` and remainder stored in `%rdx` (so we get the full 128-bit result)
- Signed
  - `idivq s`, do $(\%rdx + \%rax) \div s$ where the dividend is given as `%rdx` for higher-order part and `%rax` for the lower order part, divisor is `s`, quotient is stored in `%rax` and remainder stored in `%rdx` (so we get the full 128-bit result)
- Useful instruction for division: `cqto`
  - convert quad word to octal word
  - no operands
  - takes the sign bit from `%rax` and replicates it in all bits of `%rdx`
  - effect: sign extend 64-bit signed `%rax` to 128-bit signed `%rdx:%rax`

3 Assembly Exercise

Try to convert the following C program to assembly. Make sure to note any assumptions you make and what register contains what at any given point.

```
long arith(long x, long y, long z) {
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

One possible solution is given below.

Let's assume the variables `x`, `y`, `z` are located in the registers `%rax`, `%rbx`, `%rcx` respectively. Also, assume the return register is `%rdx`. Then a naive solution could look as follows:

```
arith:
  movq    %rax, %r8     # calculating t1, t1 = x
  imulq   %rbx, %r8     # t1 = x * y
  movq    %rcx, %r9     # calculating t2, t2 = z
  addq    %r8, %r9      # t2 = z + t1
  movq    %rax, %r10    # calculating t3, t3 = x
  addq    $4, %r10      # t3 = x + 4
  movq    %rbx, %r11    # calculating t4, t4 = y
  imulq   $48, %r11     # t4 = y * 48
  movq    %r10, %r12    # calculating t5, t5 = t3
  addq    %r11, %r12    # t5 = t3 + t4
  movq    %r9, %r13     # calculating rval, rval = t2
  imulq   %r12, %r13    # rval = t2 * t5
  movq    %r12, %rdx    # rdx = r12, aka return val
  ret
```

Of course, this implementation is not efficient and can be definitely made better. The idea above was to assume we used every of register available and naively translate each instruction line by line where for each new variable we used a new register. In general, when writing assembly we can begin with a naive version that uses memory liberally, so load and store to memory as many times as we want, and use every register we want. Then, after this implementation we can think about how to optimize our solution and get better assembly.

Let's try to optimize the above assembly code. Notice how variable `t1` is used only once and then never used again, and then variable `t2` is used only once and then never used again and then `ret` is used only once. Let's assign the register `%r8` to hold at any point in time one of these variables. Variable `t3` can actually be calculated using a `leaq` calculation so we do not assign any variables to it. Finally, variable `t4` is only used to calculate the result of variable `t5` so maybe we can somehow only assign these two variables to one register, say `%r9`.

Furthermore, we notice that $y \times 48 = 3y \times 16 = (y+2y) >> 4$ which means that we can write this operation using one `leaq` operation and one `salq` operation. $x + 4$ can be written as `leaq 4(%rcx,%rcx), Dst` where we must choose the destination register carefully. Can you figure out an optimized assembly version of this program? Give it a try and then check out the answer below.

```
arith:
  leaq    (%rax,%rbx), %r8      # r8 = x + y = t1
```

```
addq    %r8, %rcx              # r8 = z + r8 = z + t1 = t2
leaq    (%rbx,%rbx,2), %r9     # r9 = y + 2 * y
salq    $4, %r9                # r9 = r9 >> 4 = 48 = t4
leaq    4(%rcx,%rcx), %r9      # r9 = (x+4) + t4 = t3 + t4 = t5
imulq   %r8, %r9               # r8 = r8 * r9 = t2 * t5 = retval
ret                            # retval is in r8
```

We see that:

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions