A Little Bit of Math

Edited from the work of Ray Seyfarth

August 7, 2015

A little bit of math

- So far we have learned how to get values into registers
- And how to place them back into memory
- Just some ordinary arithmetic can help us write slightly more useful programs
- We will only discuss integer math in this lecture.

Negation

- The negate instruction, `neg`, converts a number to its **two's complement.**
- `neg` sets the sign and zero flags
  - Which will be useful when we perform conditional jumps and moves.
- There is only a single operand which is source and destination

```
neg     rax   ; negate the value in rax
neg     eax   ; negate the value of eax and zx the rest
neg     ax    ; negate the value of ax
neg     al    ; negate the value of al
```

Negation

- For memory operands you must include a size prefix
- The sizes are `byte`, `word`, `dword` and `qword`

```
neg     qword [x] ; negate a 8 byte integer at x
neg     dword [x] ; negate a 4 byte integer at x
neg     word [x] ; negate a 2 byte integer at x
neg     byte [x]  ; negate a 1 byte integer at x
```

The add instruction

- The add instruction always has exactly 2 operands
  - The source and, (RHS)
  - the destination (LHS)
- It adds its source value to its destination
- The source can be a
  - immediate
  - register
  - memory location
- The destination can be a
  - register
  - memory location
- Using memory locations for both source and destination is **not allowed**
  - as is the general pattern with the x86-64 instruction set.

The add instruction

- After an ADD instruction executes it sets the following flags:
  - sign flag(SF)
  - zero flag(ZF)
  - overflow flag(OF)
  - there are more, but they are no important in this course.
- There is no special "signed add" versus "unsigned add" since the logic is identical
- There is a special 1 operand increment instruction, `inc`

```
inc rax      ; add one to rax
inc byte [x]; add one to the interger byte at x
```

A program using add

Program has three "variables": **a**=151,**b**=310, and **sum**=0. We want to:

- set **a**=**a**+9
- set **sum**=**a**+**b**+10

A program using add

```
        segment .data
a       dq      151
b       dq      310
sum     dq      0
        segment .text
        global  main
main:
        mov     rax, 9       ; set rax to 9
        add     [a], rax     ; add rax to a
        mov     rax, [b]     ; get b into rax
        add     rax, 10      ; add 10 to rax
        add     rax, [a]     ; add the contents of a
        mov     [sum], rax   ; save the sum in sum
        xor     rax, rax
        ret
```

The subtract instruction

- The `sub` instruction performs integer subtraction
- Like `add` it supports 2 operands
- Only one of the operands can be a memory operand
- There is a "subtract one" instruction, `dec`
- It sets the sign flag, the zero flag and the overflow flag
- There is no special "signed subtract" versus "unsigned subtract" since the logic is identical

A program using `sub`

Program has three "variables": **a**=100,**b**=200, and **diff**=0. We want to:

- set **a**=**a**-10
- set **b**=**b**-10
- set **diff**=**b-a**

A program using sub

```
        segment .data
a       dq      100
b       dq      200
diff    dq      0
        segment .text
        global  main
main:
        mov     rax, 10
        sub     [a], rax     ; subtract 10 from a
        sub     [b], rax     ; subtract 10 from b
        mov     rax, [b]     ; move b into rax
        sub     rax, [a]     ; set rax to b-a
        mov     [diff], rax  ; move the difference to diff
        mov     rax, 0
        ret
```

Multiplication

- Unsigned multiplication is done using the `mul` instruction
- Signed multiplication is done using `imul`
- There is only 1 form for `mul`
  - It uses 1 operand, the source operand
  - The other factor is in `rax`, `eax`, `ax` or `al`
  - The destination is `ax` for byte multiplies
  - Otherwise the product is in `rdx:rax`, `edx:eax`, or `dx:ax`

```
mov     rax, [a]
mul     qword [b]   ; a * b will be in rdx:rax
mov     eax, [c]
mul     dword [d]   ; c * d will be in edx:eax
```

Signed multiplication

- `imul` has a single operand form just like `mul`
- It also has a 2 operand form, source and destination, like `add` and `sub`
- Finally there is a 3 operand form: destination, source and immediate source
- If you need all 128 bits of product, use the single operand form

```
imul    rax, 100        ; multiply rax by 100
imul    r8, [x]         ; multiply r8 by x
imul    r9, r10         ; multiply r9 by r10
imul    r8, r9, 11      ; store r9 * 11 in r8
```

Division

- Division returns a quotient and a remainder
- It also has signed (`idiv`) and unsigned forms (`div`)
- In both forms the dividend is stored in `rdx:rax` or parts thereof
- The quotient is stored in `rax`
- The remainder is stored in `rdx`
- No flags are set

```
mov     rax, [x]        ; x will be the dividend
mov     rdx, 0          ; 0 out rdx, so rdx:rax == rax
idiv    qword [y]             ; divide by y
mov     [quot], rax     ; store the quotient
mov     [rem], rdx      ; store the remainder
```

Conditional move instructions

- There are many variants of conditional move, `cmovCC`, where `CC` is a condition like `l` for less
- These are great for simple conditionals
- You can avoid interrupting the instruction pipeline

Instruction	effect
cmovz	move if zero flag set
cmovnz	move if zero flag not set (not zero)
cmovl	move if result was negative
cmovle	move if result was negative or zero
cmovg	move if result was positive
cmovge	result was positive or zero

Conditional move examples

- Here is some code to compute absolute value of rax

```
mov     rbx, rax     ; save original value
neg     rax          ; negate rax
cmovl   rax, rbx     ; replace rax if negative
```

- The code below loads a number from memory, subtracts 100 and replaces the difference with 0 if the difference is negative

```
mov     rbx, 0       ; set rbx to 0
mov     rax, [x]     ; get x from memory
sub     rax, 100     ; subtract 100 from x
cmovl   rax, rbx     ; set rax to 0 if rax was negative
```

Why use a register?

- Don't use a register if a value is needed for 1 instruction
- Don't worry about it for things which execute infrequently
- Use registers instead of memory for instructions which execute enough to matter
- If you are writing a program for a class and efficiency is not part of the grade, pick the clearest way to write the code
- With so many registers, it can create opportunities for efficiency at the cost of clarity

Print to Console

```
mov  rax, 1          ; write
mov  rdi, 1           ; stdout
mov  rsi, output       ; address of first byte in output
mov  rdx, [length]    ; load length in rdx
syscall
```