Registers

Ray Seyfarth

June 29, 2012

Outline

Register basics

- Computer main memory has a latency of about 80 nanoseconds
- A 3.3 GHz CPU uses approximately 0.3 nsecs per cycle
- Memory latency is about 240 cycles
- The Core i7 has 3 levels of cache with different latencies
  - Level 3 about 48 nsec latency or about 150 cycles
  - Level 2 about 10 nsec latency or about 39 cycles
  - Level 1 about 4 nsec latency or about 12 cycles
- There is a need for even faster memory
- This ultra-fast "memory" is the CPU's registers
- Some register-register instructions complete in 1 cycle

x86-64 registers

- CPUs running in x86-64 mode have 16 general purpose registers
- There are also 16 floating point registers (XMM0-XMM15)
- There is also a floating point register stack which we ignore
- The general purpose registers hold 64 bits
- The floating point registers can be either 128 or 256 bits
  - The CPU can use them to do 1 32 bit or 1 64 bit floating point operation in an instruction
  - The CPU can also use these to do packed operations on multiple integer or floating point values in an instruction
  - "Single Instruction Multiple Data" - SIMD
- The CPU has a 64 bit instruction pointer register - `rip`
- There is a 64 bit flags register, `rflags`, holding status values like whether the last comparison was positive, zero or negative

General purpose registers

- These registers evolved from 16 bit CPUs to 32 bit mode and finally 64 bit mode
- Each advance has maintained compatibility with the old instructions
- The old register names still work
- The old collection was 8 registers which were not entirely general purpose
- The 64 bit collection added 8 completely general purpose 64 bit registers named `r8` - `r15`

The 64 bit registers evolved from the original 8

- Software uses the "r" names for 64 bit use, the "e" names for 32 bit use and the original names for 16 bit use
- `rax` - general purpose, accumulator
  - `rax` uses all 64 bits
  - `eax` uses the low 32 bits
  - `ax` uses the low 16 bits
- `rbx`, `ebx`, `bx` - general purpose
- `rcx`, `ecx`, `cx` - general purpose, count register
- `rdx`, `edx`, `dx` - general purpose
- `rdi`, `edi`, `di` - general purpose, destination index
- `rsi`, `esi`, `si` - general purpose, source index
- `rbp`, `ebp`, `bp` - general purpose, stack frame base pointer
- `rsp`, `esp`, `sp` - stack pointer, `rsp` is used to push and pop

The original 8 registers as bytes

- Kept from the 16 bit mode
  - `al` is the low byte of `ax`, `ah` is the high byte
  - `bx` can be used as `bl` and `bh`
  - `cx` can be used as `cl` and `ch`
  - `dx` can be used as `dl` and `dh`
- New to x86-64
  - `dil` for low byte of `rdi`
  - `sil` for low byte of `rsi`
  - `bpl` for low byte of `rbp` (probably useless)
  - `spl` for low byte of `rsp` (probably useless)
- There is no special way to access any “higher” bytes of registers

The 8 new general purpose registers as smaller registers

- Here the naming convention changes
- Appending “d” to a register accesses its low double word - `r8d`
- Appending “w” to a register accesses its low word - `r12w`
- Appending “b” to a register accesses its low byte - `r15b`

Moving a constant into a register

- Moving is fundamental
- `yasm` uses the mnemonic `mov` for all sorts of moves
- The code from `gcc` uses mnemonics like `movq`
- Most instructions use 1, 2 or 4 byte immediate fields
- `mov` can use an 8 byte immediate value

```
mov rax, 0x0123456789abcdef ; can move 8 byte immediates
mov rax, 0
mov eax, 0                  ; the upper half is set to 0
mov r8w, 16                 ; affects only low word
```

- Time to try some `movs` using gdb

Moving a value from memory into a register

```
        segment .data
a       dq      175
b       dq      4097
c       db      1, 2, 3, 4
d       dd      0xffffffff
        segment .code
        mov     rax, a
        mov     rbx, [a]
        mov     rcx, [c]
        mov     edx, [c]
```

- Using simply `a` places the address of `a` into `rax`
- Using `[a]` places the value of `a` into `rbx`
- `mov rcx, [c]` makes `rcx` = 0x01020304ffffffff
- `mov edx, [c]` makes `rdx` = 0x01020304

Moving a value from memory into a register (2)

- The from memory `mov` instruction has a 32 bit immediate field
- This is where the address is placed
- This means using addresses greater than 4 GB requires getting the address into a register rather than using the immediate field
- There is a special 64 bit form, but generally you will not have a 64 bit immediate address
- The register name defines the number of bytes moved
- `mov rax, a` is really a "move constant" instruction
- `mov rax, [a]` is a "move from memory" instruction

A program to add 2 numbers from memory

```
        segment .data
a       dq      175
b       dq      4097
        segment .text
        global  main
main:
        mov     rax, [a]     ; mov a into rax
        add     rax, [b]     ; add b to rax
        xor     eax, eax
        ret
```

- Time to try this with gdb
- You will see that gdb thinks variables are double word integers

Move with sign extend or zero extend

- If you move a double word into a double word register, the upper half is zeroed out
- If you move a 32 bit immediate into a 64 bit register it is sign extended
- Sometimes you might wish to load a smaller value from memory and fill the rest of the register with zeroes
- Or you may wish to sign extend a small value from memory
- For `movsx` and `movzx` you need a size qualifier for the memory operand

```
    movsx  rax, byte [data]     ; move byte, sign extend
    movzx  rbx, word [sum]      ; move word, zero extend
    movsx  rcx, dword [count]   ; move dword, sign extend
```

Moving values from a register into memory

- Simply use the memory reference as the first operand

```
mov    [a], rax        ; move a quad word to a
mov    [b], ebx        ; move a double word to b
mov    [c], r8w        ; move a word to c
mov    [d], r15b       ; move a byte to d
```

Moving data from one register to another

- Use 2 register operands

```
mov     rax, rbx     ; move rbx to rax
mov     eax, ecx     ; move ecx to eax, zero filled
mov     cl, al       ; move al to cl, leave rest of
                     ; unchanged
```