Assembly Language Programming III

x86-64 Architecture

CSCI 2050U - Computer Architecture

Randy J. Fortier
@randy_fortier

Outline

- Moving data between registers and memory
- Arithmetic operations
    - Addition
    - Subtraction
    - Multiplication
    - Division
    - Shift
    - Rotation

Basic and Arithmetic Instructions

CSCI 2050U - Computer Architecture

Basic Instructions

- Each corresponds to a single instruction actually executed by the CPU
- Examples

```
mov rax, [number]
```

  - copies a quadword from memory to the register RAX (also called the accumulator)

```
add rax, 24
```

  - adds the quadword representation of `24` to the number already in `RAX`, replacing the number in `RAX`

```
sub rax, 24
```

  - subtracts the quadword representation of `24` from the number already in `RAX`, replacing the number in `RAX`

Parts of an Instruction

- Instruction's object code begins with the opcode, usually one byte
  - Example, `A1` for `mov rax, [number]`
- Immediate operands are constants embedded in the object code
  - Example, `0000009E` for `add rax, 158`
- Addresses are assembly-time; must be fixed when program is linked and loaded
  - Example, `00000004` for `mov sum, rax`

Operand Types

- Immediate mode (e.g. `mov rax, anyLabel`)
  - Constant assembled into the instruction
- Register mode (e.g. `mov rax, rbx`)
  - A code for a register is assembled into the instruction
- Memory references (e.g. `mov rax, [number]`)
  - Several different modes

Memory References

- Direct – at a memory location whose address is built into the instruction
  - Usually recognized by a data segment label
  - e.g., `mov [sum], rax`

    (here `rax` is a register operand)
- Register indirect – at a memory location whose address is in a register
  - Usually recognized by a register name in brackets,
  - e.g., `mov qword [rbx], 10`

    (here 10 is an immediate operand)

Memory References - Examples

- `[rbp]` - base register only
- `[rbx + rdi * 4]` - base + index * scale
- `[rbp + rax]` - scale is 1 (bytes)
- `[rax - 8]` – offset by -8
- `[rax + rdi * 8 + 4]` - all four components
- `[rax + offset]` - uses the address of the variable 'offset' as the offset
- ...more...

Multiplication - `MUL` and `IMUL`

- Multiplication is different for unsigned (`MUL`) and signed (`IMUL`) numbers
  - e.g. `mul rcx ; multiply RAX * RCX` (`RCX` - 2nd operand)
    - Of course, the operand can be memory or a register, as well
    - The second operand is explicit (`rcx` in this case)
    - The first operand is implicit:

Size	1st operand	Result
byte	AL	AX
word	AX	DX:AX
dword	EAX	EDX:EAX
qword	RAX	RDX:RAX

Multiplication - `MUL` and `IMUL`

- An example:

```
mov rdx, 0
mov rax, 12
mov rcx, 4
mul rcx
; rdx should be zero
; rax should be 48
```

Division - `DIV` and `IDIV`

- Division is different for unsigned (`DIV`) and signed (`IDIV`) numbers
  - e.g. `div rcx        ; divide RDX:RAX / RCX` (`RCX` - 2nd operand)
    - Of course, the operand can be memory or a register, as well
    - The second operand is explicit (`rcx` in this case)
    - The first operand is implicit:

Size	1st operand	Quotient	Remainder
byte	AX	AL	AH
word	DX:AX	AX	DX
dword	EDX:EAX	EAX	EDX
qword	RDX:RAX	RAX	RDX

Division - `DIV` and `IDIV`

- An example:

```
mov rdx, 0
mov rax, 12
mov rcx, 4
div rcx
; rdx should be zero (since 12 % 4 == 0)
; rax should be 3 (since 12 / 4 == 3)
```

Shifting and Rotating

- Move all bits left (or right) 3 positions:
  - `shl rax, 3` (or `shr rax, 3`)
- Move all bits left (or right) 3 positions (sign is preserved) (arithmetic shift):
  - `sal rax, 3` (or `sar rax, 3`)
  - `sal` is functionally identical to `shl`
- There are also rotations
  - Instead of dropping the bit on the left (or right), it is wrapped around
  - `rol rax, 3` (or `ror rax, 3`)

Wrap-Up

- Moving data between registers and memory
- Arithmetic operations
    - Addition
    - Subtraction
    - Multiplication
    - Division
    - Shift
    - Rotation

What is Next?

- Comparisons
- Unconditional jumps
- Conditional jumps
- Implementing conditionals
- Implementing loops