User Id : weizhe.goh@digipen.edu | Started : 2019.11.12 12:18:26 | Score :

DigiPen CS Vadim Surov CS100 Presentation #15

Assembler - Basics

This presentation introduces and shows basics of assembler

★ 1 Intro

- Most programming nowdays is done using so-called “high-level” languages (such as C/C++ or JavaScript)
- These languages deliberately “hide” from a programmer many details concerning HOW his problem actually will be solved by the underlying computing machinery
- Key point: high-level languages let programmers focus attention on the problem to be solved, and not spend effort thinking about details of “how” a particular piece of electrical machinery is going to carry out the pieces of a desired computation
- Programmers don’t have to know very much about how a digital computer actually works

★ 2 Intro

- For understanding how computers work, we need familiarity with the computer’s own language (called “machine language”)
- It’s "low-level" language (very detailed)
- It is specific to a machine’s “architecture”
- Here’s what a program-fragment looks like:

  A1BC9304 080305C0 930408A3 C0940408
- It means: z = x + y;

★ 3 Intro

- It is extremely difficult, tedious (and error-prone) for humans to read and write “raw” machine language
- So, human readable machine language, called assembly language or assembler, is invented
- There are two key ideas behind assembler:
  - mnemonic opcodes: we employ abbreviations of English language words to denote operations
  - symbolic addresses: we invent “meaningful” names for memory storage locations we need
- These make machine-language understandable to humans – if they know their machine’s design

★ 4 Intro

- Advantages using assembler:
  - Greater control. It gives you greater control over how certain functions are implemented at the assembly language level of the final program.
  - Optimization. Programmers can use assembly language code to implement the most performance-sensitive parts of their program's algorithms, code that is apt to be more efficient than what might otherwise be generated by the compiler.
  - Access to processor specific instructions.

★ 5 Tools

- as, the GNU Assembler
  - It's a family of assemblers.
  - If you use the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture.
  - Each version has much in common with the others, including object file formats, most assembler directives and assembler syntax.

★ 6 Tools

- MyTA uses online GNU assembler GCC 9.1.0 which is the 64-bit version of the x86 instruction set (x86-64):

Run

```
    .data
str: .ascii "Hello World!"
    .text
    .global main
main:
    push  %rbx #For alignment

    mov   $str, %rdi
    call  puts

    mov  $0, %eax  #return 0;
    pop   %rbx
    ret
```

```
Hello World!
```

- Press Run button to see the output

★ 7 Tools

- Now output using printf

Run

```
    .data
format: .ascii "%d\n"
    .text
    .global main
main:
    push   %rbx #For alignment

    mov    $format, %rdi
    mov    $2, %rsi
    xor    %eax, %eax #Clear AL
    call   printf

    xor   %eax, %eax #return 0;
    pop   %rbx
    ret
```

```
2
```

★ 8 Tools

- Try the following code using online compiler http://jdoodle.com. Select "Assembler (GCC)" compiler.

```
    .data
format: .ascii "%d\n"
    .text
    .global main
main:
    push   %rbx # For alignment

    mov    $0, %eax # The starting point
    mov    $100, %ebx # The ending point

_loop:
    push   %rax  # Saving
    push   %rbx

    # Output
    mov    $format, %rdi
    mov    %rax, %rsi
    xor    %eax, %eax # Clear AL
    call   printf

    pop    %rbx  # Restoring
    pop    %rax

    # Check against the ending value
    cmpl   %eax, %ebx
    je     _end

    # Increment the current value
    inc    %eax
    jmp    _loop
_end:

    xor    %eax, %eax #return 0;
    pop    %rbx
    ret
```

- What is the output?

★ 9 Program structure

- ASCII-character text
- 2 segments: .data and .text
- Program consists of series of ‘statements’
- Each program-statement fits on one line
- Program-statements all have same layout
- Design in 1950s was for IBM punch-cards

★ 10 Program structure

- Each ‘statement’ was comprised of four ‘fields’
- Fields appear in a prescribed left-to-right order
- These four fields were named (in order):
  - the ‘label’ field
  - the ‘opcode’ field
  - the ‘operand’ field
  - the ‘comment’ field
- In many cases some fields could be left blank
  - Extreme case (very useful): whole line is blank!

★ 11 Segments

- The .data segment
  - contains any global or static variables which have a pre-defined value and can be modified
- The .text segment
  - contains executable instructions

★ 12 Label

- A label is a ‘symbol’ followed by a colon (‘:’)
- The programmer invents his own ‘symbols’
- Symbols can use letters and digits, plus a very small number of ‘special’ characters (‘.’, ‘_’, ‘$’)
  - A ‘symbol’ is allowed to be of arbitrarily length
- Ex: _end:

★ 13 Opcode

- Opcodes are predefined symbols that are recognized by the GNU assembler
- There are two categories of ‘opcodes’ (called ‘instructions’ and ‘directives’)
- ‘Instructions’ represent operations that the CPU is able to perform (e.g., ‘add’, ‘inc’)
  - Each ‘instruction’ gets translated by compiler into a machine-language statement that will be fetched and executed by the CPU when the program runs (i.e., at ‘runtime’)
- ‘Directives’ are commands that guide the work of the assembler (e.g., ‘.global’, ‘.int’)
  - With GNU assembly language, they are easy to distinguish: directives begin with ‘.’

★ 14 Opcode

- An ‘official’ list of the instruction codes can be found in Intel’s programmer manuals:
  - http://developer.intel.com
- But it’s three volumes, nearly 1000 pages (it describes ‘everything’ about Intel Pentiums)
- An ‘unofficial’ list of (most) Intel instruction codes can fit on one sheet:
  - https://drive.google.com/open?id=15Cx2W2Ecbrty44kRpCGS7IhXljdrOUNH
- Compare it with 32-bit version:
  - https://drive.google.com/open?id=1CaS0CbVHZZ4i-KBbF4QW45eGxQuYpEeb

★ 15 Opcode

- CPU Instructions usually operate on data-items
- Only certain sizes of data are supported:
  - byte: one byte consists of 8 bits
  - word: consists of two bytes (16 bits)
  - long: uses four bytes (32 bits)
  - quadword: uses eight bytes (64 bits)
- With AT&T’s syntax, an instruction’s name also incorporates its effective data-size (as a suffix b/w/l/q)
- With Intel syntax, data-size usually isn’t explicit, but is inferred by context (i.e., from operands)

★ 16 Operand

- Operands can be of several types:
  - a CPU register may hold the datum
  - a memory location may hold the datum
- An instruction can have ‘built-in’ data
- Frequently there are multiple data-items
- Sometimes there are no data-items
- An instruction’s operands usually are ‘explicit’, but in a few cases they also could be ‘implicit’

★ 17 Operand

- Some instruction that have two operands:
  - mov %rbx, %rcx
  - add $4, %rsp
- Some instructions that have one operand:
  - incl %eax
  - push $fmt
- An instruction that lacks explicit operands:
  - ret

★ 18 Registers

- The table below lists the commonly used registers (sixteen general-purpose plus two special)

Register	Conventional use	Low 32-bits	Low 16-bits	Low 8-bits
%rax	Return value, callee-owned	%eax	%ax	%al
%rdi	1st argument, callee-owned	%edi	%di	%dil
%rsi	2nd argument, callee-owned	%esi	%si	%sil
%rdx	3rd argument, callee-owned	%edx	%dx	%dl
%rcx	4th argument, callee-owned	%ecx	%cx	%cl
%r8	5th argument, callee-owned	%r8d	%r8w	%r8b
%r9	6th argument, callee-owned	%r9d	%r9w	%r9b
%r10	Scratch/temporary, callee-owned	%r10d	%r10w	%r10b
%r11	Scratch/temporary, callee-owned	%r11d	%r11w	%r11b
%rsp	Stack pointer, caller-owned	%esp	%sp	%spl
%rbx	Local variable, caller-owned	%ebx	%bx	%bl
%rbp	Local variable, caller-owned	%ebp	%bp	%bpl
%r12	Local variable, caller-owned	%r12d	%r12w	%r12b
%r13	Local variable, caller-owned	%r13d	%r13w	%r13b
%r14	Local variable, caller-owned	%r14d	%r14w	%r14b
%r15	Local variable, caller-owned	%r15d	%r15w	%r15b
%rip	Instruction pointer			
%eflags	Status/condition code bits			

★ 19 Comments

- An assembly language program often can be hard for a human being to understand
- Even a program's author may not be able to recall his programming idea after awhile
- So programmer 'comments' can be vital
- A comments begin with the '#' character
  - Also can use c-style block comment /* */
- The assembler disregards all comments (but they will appear in program listings)

★ 20 Constants

- A binary integer is '0b' or '0B' followed by zero or more of the binary digits: 0b11111111.
- An octal integer is '0' followed by zero or more of the octal digits: 01234567
- A decimal integer starts with a non-zero digit followed by zero or more digits: 123456789, -10
  - A hexadecimal integer is '0x' or '0X' followed by one or more hexadecimal digits: 0xFF
- A string is written between double-quotes: "Hello world!\n" (\n is escape character for new line, like in c)

★ 21 Output

- Following code is equivalent to
  printf("x is %d\n", 42);

Run

```
    .data
x:    .long  42
str: .asciz "x is %d\n"
    .text
    .global main
main:
    push  %rbx /* For alignment */

    # Output
    mov   $str, %rdi
    mov   x, %rsi
    call  printf

    xor   %eax, %eax #return 0;
    pop   %rbx
    ret
```

```
x is 42
```

- Use this example to output result of calculations in assignments and for output temp values during debugging.

★ 22 Move

- Following code is equivalent to
  x=24;

Run

```
    .data
x:    .long  0
str: .asciz "x is %d\n"
    .text
    .global main
main:
    push  %rbx /* For alignment */

    movl $24, x

    # Output
    mov   $str, %rdi
    mov   x, %rsi
    call  printf

    xor   %eax, %eax #return 0;
    pop   %rbx
    ret
```

```
x is 24
```

- The mov instruction copies the data item referred to by its first operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its second operand (i.e. a register or memory).
- While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

★ 23 Add

- Following code is equivalent to
  printf("x+y=%d\n", x+y);

Run

```
    .data
x:    .long  42
y:    .long  10
str: .asciz "x+y=%d\n"
    .text
    .global main
main:
    push  %rbx /* For alignment */

    movq  x, %rax
    addq  y, %rax

    # Output
    mov   $str, %rdi
    movq  %rax, %rsi
    call  printf

    xor   %eax, %eax #return 0;
    pop   %rbx
    ret
```

```
x+y=52
```

- The **add** a b instruction adds together its two operands a and b, storing the result in its second operand b. a can be register, memory or constant, b is a register or memory.

★ 24 Sub

- Following code is equivalent to
  printf("x-y=%d\n", x-y);
  when x and y one byte long.

Run

```
    .data
x:    .long  42
y:    .long  10
str: .asciz "x-y=%d\n"
    .text
    .global main
main:
    push  %rbx /* For alignment */

    xorq  %rax, %rax
    movb  x, %al
    movb  y, %ah
    subb  %ah, %al
    xor   %ah, %ah

    # Output
    mov   $str, %rdi
    movq  %rax, %rsi
    call  printf

    xor   %eax, %eax #return 0;
    pop   %rbx
    ret
```

```
x-y=32
```

★ 25 Mul

- Following code is equivalent to
  printf("x*y=%hu\n", x*y);
  when x and y two byte long.

Run

```
     .data
x:    .long  42
y:    .long  10
str: .asciz "x-y=%d\n"
     .text
     .global main
main:
     push  %rbx /* For alignment */

     xor     %eax, %eax
     movw   x, %ax
     movw   y, %bx
     imulw   %bx, %ax

     # Output
     mov   $str, %rdi
     movq  %rax, %rsi
     call  printf

     xor   %eax, %eax #return 0;
     pop   %rbx
     ret
```

```
x-y=420
```

★ 26 Address

- Following code is equivalent to
  printf("%d\n", &x);

Run

```
     .data
x:    .long  42
str: .asciz "Address of x is %d\n"
     .text
     .global main
main:
     push  %rbx /* For alignment */

     # Output
     mov   $str, %rdi
     mov   $x, %rsi
     call  printf

     xor   %eax, %eax #return 0;
     pop   %rbx
     ret
```

```
Address of x is 6295600
```

★ 27 Array

- Following code is equivalent to
  printf("%d\n", a[1]);

Run

```
    .data
a:    .long  1, 2, 3, 4
str: .asciz "%d\n"
    .text
    .global main
main:
    push  %rbx /* For alignment */

    movq  a+4, %rax

    # Output
    mov   $str, %rdi
    movq  %rax, %rsi
    call  printf

    xor   %eax, %eax #return 0;
    pop   %rbx
    ret
```

```
2
```

- Try to output second character in str

★ 28 Endianness

- Modify the following code to identify the endianness of the processor.

Run

```
    .data
long:    .byte  1, 2, 3, 4
str: .asciz "%d\n"
    .text
    .global main
main:
    push  %rbx /* For alignment */

    # Output
    mov    $str, %rdi
    movq long, %rsi
    call     printf

    xor    %eax, %eax #return 0;
    pop    %rbx
    ret
```

```
67305985
```

★ 100 References

Manual – The GNU Assembler manual
x86-64 – Quick reference

By signing this document you fully agree that all information provided therein is complete and true in all respects.

Responder sign: