CSE320 System Fundamentals II

x86 Assembly Language

YOUNGMIN KWON / TONY MIONE

Announcements

Acknowledgements

Some slides provided by Dr Yoon Seok Yang

Why Assembly Language?

The history of assembly language is closely tied to the development of computers and their architectures. Here's a brief overview of the history of assembly language

- **Early Computers (1940s-1950s):**
  Assembly language emerged with the first electronic computers in the 1940s, such as the ENIAC and UNIVAC. These early computers required programmers to work directly with machine code, which was complex and challenging.
  As computers evolved, assembly languages were developed to provide symbolic representations of machine instructions, making programming more manageable. These early assembly languages were often closely tied to specific computer models.
- **Assembly Languages for Mainframes (1950s-1960s)**:
  IBM played a significant role in the development of early assembly languages, including the IBM 704 Assembly for the IBM 704 computer. IBM's dominance in the mainframe computer market led to the widespread use of assembly languages on their machines.
- **Standardization (1950s-1960s)**:
  The need for standardization became evident as different computer manufacturers developed their own assembly languages. In 1957, the United States Department of Defense introduced the "Federally Standardized Fortran" (FSA) to establish common assembly language standards for mainframes.
- **High-Level Languages (1950s-1960s)**:
  The development of high-level programming languages like Fortran and COBOL in the late 1950s and early 1960s reduced the need for assembly language programming for many tasks. High-level languages provided more abstraction and ease of use.

Why Assembly Language?

- **Microprocessors (1970s)**:
  With the introduction of microprocessors in the early 1970s, assembly language programming gained popularity again. Early microprocessors like the Intel 4004 and 8080 had limited resources, and assembly language allowed programmers to write efficient code for these platforms.
- **Diverse Architectures (1970s-1980s)**:
  The 1970s and 1980s saw a proliferation of microprocessor architectures, each with its own assembly language. Popular assembly languages during this period included Intel x86, Motorola 68000, Zilog Z80, and MOS Technology 6502.
- **RISC vs. CISC (1980s-1990s)**:
  The 1980s and 1990s brought the debate between Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC) architectures. RISC architectures favored simpler and more regular instruction sets, making assembly language programming more approachable.
- **Modern Assembly Languages (2000s-Present)**:
  Assembly languages for modern processors like x86-64, ARM, and others have continued to evolve. These assembly languages are used in various domains, including system programming, embedded systems, and performance-critical applications.
- **Use in Specialized Domains (Present)**:
  Assembly language is still used in specialized domains, such as firmware development, operating system kernels, device drivers, and performance-critical software where precise control over hardware is essential.

Why Assembly Language?

Assembly language serves several important purposes in the field of computer science and programming, making it a valuable tool for various tasks and situations. Here are some reasons why we need assembly language:

- **Low-Level Control**: Assembly language provides a low-level interface to a computer's hardware. It allows programmers to have direct control over the CPU, memory, and other hardware components, which can be necessary for tasks like system programming, device drivers, and firmware development.
- **Efficiency**: Assembly language programs can be highly efficient because they allow fine-grained control over the instructions executed by the CPU. This level of control is crucial for optimizing critical sections of code where performance is a primary concern.
- **Embedded Systems**: Assembly language is often used in embedded systems programming, where resources are limited, and code size and execution speed are critical factors. Assembly allows developers to write compact, efficient code tailored to specific hardware.
- **Reverse Engineering**: Assembly language is essential for reverse engineering binary executables. Security professionals, malware analysts, and software engineers use assembly to analyze and understand the behavior of compiled programs, identify vulnerabilities, and develop patches or workarounds.
- **Operating Systems**: Operating system kernels are often written in assembly language or a combination of assembly and C/C++. Assembly is used to implement the low-level components of an operating system, such as task scheduling, interrupt handling, and memory management.

Why Assembly Language?

- **Real-Time Systems**: For real-time systems, where precise timing and responsiveness are crucial, assembly language can be used to write code with predictable and deterministic behavior. This is common in applications like robotics and aerospace.
- **Hardware Interaction**: Assembly is essential for programming and controlling hardware devices directly, such as graphics cards, sound cards, and input/output ports. It allows for low-level interfacing and communication with hardware components.
- **Education**: Learning assembly language helps programmers gain a deeper understanding of computer architecture and how high-level languages translate into machine code. It's often a fundamental part of computer science and engineering curricula.
- **Legacy Code**: Some legacy systems and applications are written in assembly language. Knowledge of assembly is required for maintaining and updating such systems.
- **Debugging**: Assembly language is valuable for debugging and diagnosing issues in low-level code, as it allows programmers to inspect the exact instructions and memory contents during program execution.
- **Specialized Algorithms**: Certain algorithms, like cryptographic algorithms or signal processing routines, require assembly language for optimal performance due to their computational intensity.

Why Assembly Language?

Assembly language is a low-level programming language that is specific to a particular **computer architecture or microprocessor**. The instructions and registers available in assembly language can vary significantly depending on the architecture you are working with. Therefore, I'll provide you with a general overview of assembly language concepts, but keep in mind that you'll need to consult the documentation for the specific architecture you're working with to get detailed information.

```
------------------------ hello.s
    .file   "hello.c"
    .section    .rodata
.LC0:
    .string "hello world"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
    .section    .note.GNU-stack,"",@progbits
```

```
------------------------ hello.c
#include <stdio.h>
int main()
{
    printf("hello world\n");
    return 0;
}
```

Generating an Assembly File from C

```
gcc –S –c –O0 hello.c
```

- `-S`: generate an assembly file (`hello.s`)
- `-c`: do not link
- `-O0`: no optimization [will make the generated assembler code easier to follow]

Assembly Instructions

- **Arithmetic and Logic Instructions**: These instructions perform various arithmetic and logical operations on data stored in registers or memory. Common instructions include ADD (addition), SUB (subtraction), MUL (multiplication), DIV (division), AND (logical AND), OR (logical OR), XOR (logical XOR), CMP (compare), and more.
- **Data Movement Instructions**: These instructions move data between registers and memory. Common instructions include MOV (move), LOAD (load data from memory), and STORE (store data to memory).
- **Control Flow Instructions**: These instructions control the flow of program execution. Common instructions include JMP (jump), CALL (call a subroutine), RET (return from a subroutine), JZ (jump if zero), JNZ (jump if not zero), and more.
- **Conditional Instructions**: These instructions allow you to perform actions based on conditions. They often work in conjunction with control flow instructions. Common conditional instructions include JE (jump if equal), JNE (jump if not equal), JL (jump if less than), JG (jump if greater than), and so on.
- **Stack Instructions**: These instructions manipulate the stack, which is used for function calls and managing local variables. Common stack instructions include PUSH (push data onto the stack) and POP (pop data from the stack).
- **String Instructions**: Some architectures provide specific instructions for working with strings of characters or bytes. These include instructions like MOVS (move string), CMPS (compare strings), and more.

Machine Registers

Registers are **small, fast storage locations within the CPU** that are used for performing operations and storing intermediate data. The number and types of registers can vary between different CPU architectures, but here are some common types of registers you might encounter.

- **General-Purpose Registers**: These registers are used for general data manipulation. They may be used to store integers, memory addresses, or other data. Examples include EAX, EBX, ECX, EDX on the x86 architecture.
- **Index Registers**: These registers are often used as pointers or offsets for accessing data in memory. Examples include ESI and EDI on x86.
- **Segment Registers**: These registers are used for managing memory segments. On x86, common segment registers include CS (Code Segment), DS (Data Segment), ES (Extra Segment), and SS (Stack Segment).

Machine Registers

- **Instruction Pointer (IP)**: This register stores the address of the next instruction to be executed.
- **Flags Register**: This register contains various condition flags that are set or cleared based on the results of arithmetic and logical operations. Common flags include the zero flag (ZF), carry flag (CF), sign flag (SF), and overflow flag (OF).
- **Stack Pointer (SP) and Base Pointer (BP)**: These registers are used for stack manipulation. SP points to the top of the stack, while BP is often used as a base address for accessing local variables in a subroutine.

x86 Assembly

Two different Syntaxes

- Intel Syntax: op dst, src
  - `movl eax, 1`
  - `addl eax, ebx`

- AT&T (GAS) Syntax: op src, dst
  - `movl 1, %eax`
  - `addl %eax, %ebx`

[Intel Architecture Reference Manuals]

Assembler Directives for Sections

`.text`

- Instructions/program code are placed here

`.data`

- Initialized read/write data are defined here

`.section .rodata`

- Initialized read only data are defined here

`.comm symbol, length, alignment`

- Uninitialized data are allocated in the bss section

`.local name`

- Makes a name a local symbol
- `.lcomm = .local + .comm`

BSS (Block Started by Symbol) section is a special section in memory or an object file where uninitialized data is stored. BSS is commonly used for variables that are declared in the source code but do not have an initial value assigned. These variables are typically initialized to zero or null by default, depending on their data type.

More Assembler Directives

`.ascii "string" …`
- Define strings without the terminal zero

`.string "string" …`
- Define null-terminated strings

`.byte, .int, .long, .quad`
- Define integer numbers

`.double, .float`
- Define floating point numbers

`.align`
- Pad the location counter to a particular storage boundary.

`.size`
- Set the size associated with a symbol

AT&T Assembly Format

General format:

- operation source, destination
- e.g. `movb $0x05, %al`

Operation Suffixes

- Instructions are suffixed with
  `b`: byte, `s`: short (2 byte int or 4 byte float), `w`: word (2 byte), `l`: long(4 byte int or 8 byte float), `q`: quad (8 byte), `t`: ten byte (10 byte float)

Prefixes

- `%` for registers, `$` for constant numbers (literals/immediates)

Literals

Integers
- `24, 0b1010, 0x4a, 074`

Floating point numbers
- `0.1, 1.2e3`

Strings
- `"abc\n"`

Characters
- `'a', '\n'`

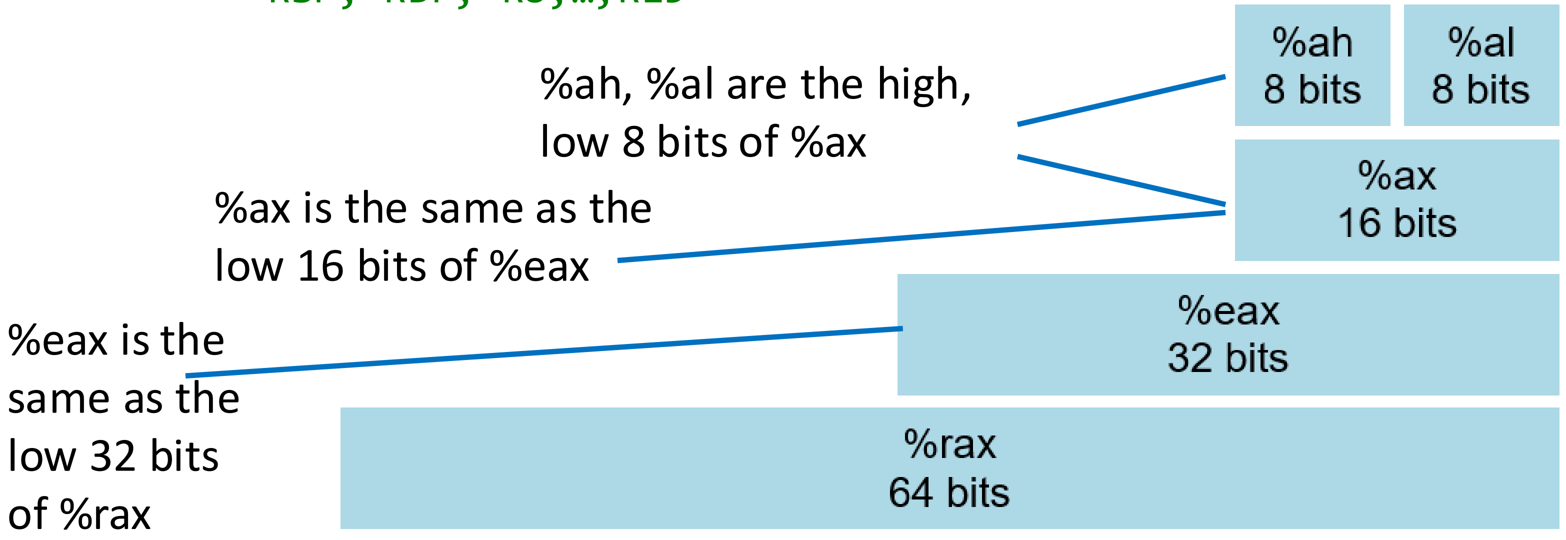
Registers

```
8 bit:  AH, AL, BH, BL, CH, CL, DH, DL, R8B,…,R15B

16 bit: AX, BX, CX, DX, SI, DI, SP, BP, R8W,…,R15W

32 bit: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP,
        R8D,…,R15D

64 bit: RAX, RBX, RCX, RDX, RSI, RDI,
        RSP, RBP, R8,…,R15
```



%ah, %al are the high, low 8 bits of %ax
%ax is the same as the low 16 bits of %eax
%eax is the same as the low 32 bits of %rax
%ah 8 bits
%al 8 bits
%ax 16 bits
%eax 32 bits
%rax 64 bits

Addressing Operand

Syntax

`segment:displacement(base register, offset register, scalar multiplier)`

- base register + offset register * scalar multiplier + displacement (ignoring segment)
- Either or both of numeric parameters can be omitted
- Either of the register parameters can be omitted

Example

- `movl -8(%rbp, %rsi, 4), %eax #load [rbp + rsi * 4 - 8] into eax`
- `movl -8(%rbp), %eax          #load [rbp - 8] into eax`
- `leaq 8(%rbx, %rcx, 2), %rax  #load rcx * 2 + rbx + 8 into rax`

Move and Stack Manipulation Instructions

mov src, dst

- At least one of src or dst must be a register
- `e.g. movl $0, %eax`

push src

- `e.g. pushl %eax`
- `eqv: subq $4, %rsp; movl %eax (%rsp)`

pop dst

- `e.g. popq %rax`
- `eqv: movq (%rsp) %rax; addq $8, %rsp`

leave

-
  ```
  movq %rbp, %rsp
  popq %rbp
  ```

Arithmetic Instructions

add src, dst

- `e.g. addq $2, %rax     # rax = rax + 2`

sub src, dst

- `e.g. subq %rbx, %rax   # rax = rax – rbx`

Multiplies the low word(16-bit) of the AX (16-bit accumulator) register by the low word of the BX (16-bit general purpose) register, and the result is stored in the 32-bit signed product in DX:AX register pair (DX: 16-bit general purpose register)

mul arg

- `e.g. mulw %bx     # bx * ax -> dx (high 16bits), ax (lower 16bits)`

div arg

Divides the 64-bit value formed by the EDX:EAX register pair by the value in the EBX register. The quotient is stored in the EAX register, and the remainder is stored in the EDX register.

-
  ```
  e.g. divl %ebx
  # (edx * 2^32 + eax) / ebx -> eax,
  # (edx * 2^32 + eax) mod ebx -> edx
  ```

Logical Instructions

and src, dst

- `e.g. andl $0xf, %eax # eax &= 0xf`

or src, dst

- `e.g. orl $0xf, %eax  # eax |= 0xf`

not dst

- `e.g. notq %rax       # rax = ~rax`

xor src, dst

- `e.g. xorw %ax, %ax   # ax = ax xor ax`

Flags

ZF (zero flag)

- Set if the result is 0

SF (sign flag)

- Set if the MSB of the result is 1

OF (overflow flag)

- Set when overflow occurred (8 + 8 $\rightarrow$ 16 in 4bit)
- Positive num op Positive num $\rightarrow$ Negative num
- Negative num op Negative num $\rightarrow$ Positive num

Compare and Branch Instructions

cmp arg1 arg2

-
  ```
  cmpq $2, %rax
  # ZF = iff %rax - 2 == 0
  # SF = iff MSB of %rax - 2 == 1
  # OF = iff overflow occurs
  ```

test arg1 arg2

-
  ```
  testq $5, %rax
  # ZF = iff %rax & 5 == 0
  # SF = iff MSB of %rax & 5 == 1
  ```

Perform a bitwise logical AND operation between the first operand and the second operand.

Compare and Branch Instructions

```
cmp $4, %eax
je label
```

The cmp instruction is typically used to set the condition flags based on the result of the comparison, without actually storing the result in a register. The condition flags can then be used for conditional branching using instructions like je (jump if equal), jg (jump if greater), and so on.

1. Subtract $4 from %eax.
2. Jump if ZF=1: The result of the subtraction is not stored anywhere; instead, the condition flags (such as the zero flag and sign flag) are set based on the result of the subtraction.

Compare and Branch Instructions

JE, JZ, JNE, JNZ, JG, JGE, JL, JLE

- `jne label`
  `# jump if ZF == 0`
- `jg  label`
  `# jump if SF == OF and ZF == 0`
  - `cmp -4, 4 => OF=1, SF=1 on 4bit machines`
  - `cmp 2, -1 => OF=1, SF=0 on 4bit machines`

Call Instructions

call label

- `e.g. call 0x1234`
- Equivalent to

```
pushq %rip
movq 0x1234, %rip
```

ret

- `e.g. ret`
- Equivalent to

```
popq %rip
```

```
---------------------------gcd.c
#include <stdio.h>
long gcd(long x, long y)
{
    while (x != y) {
        if (x > y)
            x -= y;
        else
            y -= x;
    }
    return x;
}
------------------------------gcd.s
    .text
    .globl  gcd
    .type   gcd, @function
gcd:
    pushq   %rbp                #
    movq    %rsp, %rbp          #,
    movq    %rdi, -8(%rbp)      # x, x
    movq    %rsi, -16(%rbp)     # y, y
    jmp .L2
```

```
.L4:
    movq    -8(%rbp), %rax      # x, tmp61
    cmpq    -16(%rbp), %rax     # y, tmp61
    jle .L3                     #,
    movq    -16(%rbp), %rax     # y, tmp62
    subq    %rax, -8(%rbp)      # tmp62, x
    jmp .L2                     #
.L3:
    movq    -8(%rbp), %rax      # x, tmp63
    subq    %rax, -16(%rbp)     # tmp63, y
.L2:
    movq    -8(%rbp), %rax      # x, tmp64
    cmpq    -16(%rbp), %rax     # y, tmp64
    jne .L4                     #,
    movq    -8(%rbp), %rax      # x, D.2060
    popq    %rbp                #
    ret
```

```
-----------------------------gcd.c
long print()
{
    long a = 24, b = 30;
    long c = gcd(a, b);
    printf("gcd(%ld, %ld) = %ld\n",
           a, b, c);
    return 0;
}
```

```
-----------------------------gcd.s
    .section    .rodata
.LC0:
    .string "gcd(%ld, %ld) = %ld\n"
    .text
    .globl  print
    .type   print, @function
```

```
print:
    pushq   %rbp                #
    movq    %rsp, %rbp          #,
    subq    $32, %rsp           #,
    movq    $24, -24(%rbp)      #, a
    movq    $30, -16(%rbp)      #, b
    movq    -16(%rbp), %rdx     # b, tmp62
    movq    -24(%rbp), %rax     # a, tmp63
    movq    %rdx, %rsi          # tmp62,
    movq    %rax, %rdi          # tmp63,
    call    gcd                 #
    movq    %rax, -8(%rbp)      # tmp64, c
    movl    $.LC0, %eax         #, D.2054
    movq    -8(%rbp), %rcx      # c, tmp65
    movq    -16(%rbp), %rdx     # b, tmp66
    movq    -24(%rbp), %rsi     # a, tmp67
    movq    %rax, %rdi          # D.2054,
    movl    $0, %eax            #,
    call    printf              #
    movl    $0, %eax            #, D.2055
    leave
    ret
```

Questions?