Assembly Introduction

Systems Programming

Michael Sonntag
Michael Roland
Institute of Networks and Security

Programming environment

- ~~**Some Linux distribution**~~
  - ~~CentOS: https://www.centos.org/~~
  - ~~Ubuntu: http://www.ubuntu.com/~~
  - ~~Debian: https://www.debian.org/~~
  - ~~…~~
- ~~**GNU Binutils**~~
  - ~~Collection of binary creation/analysis/manipulation tools~~
  - ~~http://www.gnu.org/software/binutils/~~
  - ~~**GNU Assembler (as)**~~
  - ~~**GNU Linker (ld)**~~

➔ **Technically the above is correct and possible.**

BUT: We strongly advise you to use the VM that we provided as our ready-made development environment!

Terminology

- **Assembly language**
  - Low-level programming language
  - Symbolic representation of machine code (*mnemonics*)
- **Machine code**
  - Actual numbers (bits/bytes) a CPU interprets as commands
  - We are not concerned with this – Disassemblers are (very) reliable
- **Assembler / to assemble**
  - Software utility translating programs written in assembly language into machine code (object code)
- **Linker / to link**
  - Software combining modules (object code) into an executable
  - Static linking: linking at compile time
  - Dynamic linking: linking at run time
    - Needs Dynamic Link Library/Shared Object (DLL / .so) at runtime

Example: Shortest program (="Suicide")

exit.s

```
          #PURPOSE: Simple program that exits and returns a
          #         status code back to the Linux kernel
          #INPUT:   none
          #OUTPUT:  returns a status code.  This can be viewed
          #         by typing
          #         echo $?
          #         after running the program
          #VARIABLES:
          #         %rax holds the system call number
          #         %rdi holds the return status

          .section .data
          .section .text
          .globl _start

_start:
          movq $60, %rax     # this is the linux kernel command
                             # number (system call) for exiting
                             # a program
          movq $0, %rdi      # this is the status number we will
                             # return to the operating system.
                             # Change this around and it will
                             # return different things to
                             # echo $?
          syscall            # this wakes up the kernel to run
                             # the exit command
```

Assemble, link & run

- First step: **assembling**
  - □ `as exit.s -o exit.o --gstabs+`
- Second step: **linking**
  - □ `ld exit.o -o exit`
- Third step: **running**
  - □ `./exit`
- Fourth step: view **exit code**
  - □ `echo $?`
  - □ Prints the return value of the last executed program

- "`--gstabs+`": add debug information

- Note: 32 Bit programs run in a 64 Bit environment need "tweaking"
  - □ "`--32`" / "`-m elf_i386`" must be added for assembler / linker
  - □ Otherwise it will crash almost immediately!
  - □ All our programs are 64 Bit, however

Syntax (1)

- **Assembler directives**
  - Begin with “.”
  - Are commands for the assembler program (=the software utility)
  - Directives are **not** translated to code
    - Will end up in the executable file as a specification for the OS what to do with the following bytes/where to load them/...
  - **`.section .data`**
    - Tells the assembler that the memory storage section begins
  - **`.section .text`**
    - Tells the assembler that the program instruction section begins
  - **`.globl`**
    - Defines global symbol needed by the linker
  - **`.quad`** (64 bits), **`.long`** (32 bits), **`.int`** (16 bits!), **`.byte`** (8 bits) and **`.ascii`** (8 bits) for specifying the size of data items

Syntax (2)

- **Labels**
  - End with ":"
  - Symbolic locations: provide a name for an address
    - We don't know in advance which specific address. The linker will later on assign it one and replace all occurrences with the selected one
  - **`_start:`**
    - Tells the assembler where the program starts
    - This name is **not changeable**! It must **always** be **exactly** like this!

- **Registers**
  - Prefixed with "`%`"

- **Immediate values**
  - Prefixed with "**`$`**"

Syntax (3)

- **Instruction suffix**
  - Suffix determines the amount of data
  - “b” → byte (8 bits)
  - “w” → word (16 bits)
  - “l” → long int (32 bits)
  - “q” → quadword (64 bits)
  - For example “**`movl $1, %eax`**” copies the number 1 as a **long** into register EAX
    - Same instruction, different size of operands:
      `movb $1, %al  -  movw $1, %ax  -  movq $1, %rax`
    - Note: Suffix **must match** size of register used (if any)!
- **Endianness:** High-byte at higher or lower address?
  - IA-32, X86-64:
    Little endian → High-byte at high-address

1001	*1002*	*1003*	*1004*
0x0A	0x0B	0x0C	0x0D

1001: 0x0D0C0B0A (l)
1001: 0x0B0A (w)
1002: 0x0C0B (w)

- **Comments**
  - Lines beginning with “**#**”

Example: Find maximum in list of numbers (1)

- **Given**: list of integer numbers
- **Task**: find largest number in list and return it
- **Storage requirements**: how much/which memory do we need?
  - □ List of numbers, terminated by special value (**E**nd **O**f **L**ist)
  - □ Index variable: position of currently examined element
  - □ Current value
  - □ Maximum value found so far
- **Algorithm**
  1. Initialize index with 0
  2. Load number in list at current index as current value
  3. Save current value as maximum value
  4. While current value is not EOL
     1. Increase index
     2. Load number in list at current index as current value
     3. If current value is larger than maximum, save current value as new maximum
  5. Return maximum value and exit

Example: Find maximum in list of numbers (2)

- Linear execution of program code; loop using branches & goto (comparison & jump instructions)
- **Pseudo code**

```
1.  index = 0
2.  curVal = numList[index]
3.  maxVal = curVal
4.  IF curVal == 0 THEN GOTO 10
5.  index = index + 1
6.  curVal = numList[index]
7.  IF curVal <= maxVal THEN GOTO 4
8.  maxVal = curVal
9.  GOTO 4
10. RETURN maxVal
11. EXIT
```

Example: Find maximum in list of numbers (3)

maximum.s

```
             #PURPOSE:  This program finds the maximum number of a
             #          set of data items.
             #
             #VARIABLES: The registers have the following uses:
             # %rdx - Holds the index of the data item being examined
             # %rdi - Largest data item found
             # %rax - Current data item
             #
             # The following memory locations are used:
             # data_items - contains the item data.  A 0 is used
             #              to terminate the data

             .section .data

   data_items:                             #These are the data items
             .quad 3,67,34,222,45,75,54,34,44,33,22,11,66,0

             .section .text

             .globl _start

   _start:
1.           movq $0, %rdx                           # move 0 into the index register
2.           movq data_items(,%rdx,8), %rax          # load the first byte of data
3.           movq %rax, %rdi                         # since this is the first item, %rax is
                                                     # the biggest
```

Example: Find maximum in list of numbers (4)

```
    start_loop:                           # start loop
4.          cmpq $0, %rax                 # check to see if we've hit the end
            je loop_exit
5.          incq %rdx                     # load next value
6.          movq data_items(,%rdx,8), %rax
7.          cmpq %rdi, %rax               # compare values
            jle start_loop                # jump to loop beginning if the new
                                          # one isn't bigger
8.          movq %rax, %rdi               # move the value as the largest
9.          jmp start_loop                # jump to loop beginning

    loop_exit:
10.         # %rdi is the status code for the exit system call
            # and it already contains the maximum number
11.     movq $60, %rax                    # 60 is the exit() syscall
        syscall
```

Data access methods - Details

- Immediate
  - □ `movq $12, %rax`
  - □ Load the number 12 into RAX
    - ● Note: limited to 32 Bit; sign-extension used for 64 Bit registers!
- Register
  - □ `movq %rbx, %rax`
  - □ Copy the value of RBX into RAX
- Direct
  - □ `movq 12, %rax`
  - □ Load value from memory address **12** into RAX
    - ● If you want to load the **address itself** and not the **value at it**, then use LEAQ (see later – allows memory calculation – or general arithmetic ☺)
- Indirect
  - □ `movq (%rbx), %rax`
  - □ Load the value from memory (address is in RBX) into RAX

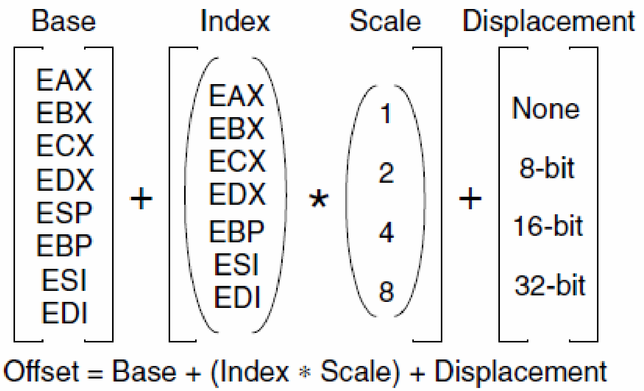
Data access methods - Details

- Base Pointer
  - □ `movq 4(%rbx), %rax`
  - □ Similar to the indirect example, but add the constant offset 4 to the address stored in RBX before accessing the memory
  - □ 64 Bit specialty: `32-Bit-displacement(%rip)` also possible
- Indexed
  - □ `movq data_items(,%rdi,8), %rax`
  - □ Start at address `data_items`, add 8 * content of RDI to that address, and finally load the memory content of this address into the register RAX
  - □ With additional offset
    - ● `movq data_items(%rbx,%rdi,8), %rax`
    - ● start at address `data_items`, add the content of RBX, add 8 * content of RDI to that address, and finally load the memory content into RAX
  - □ Attention: size-suffix (“q”) should generally match multiplicator (“8”)
    - ● It need not; but this is then very likely a programming error!

Data access – Generalized version

- General form of indexed memory address references
  - □ **`DISPLACEMENT(%BASE, %INDEX, SCALE)`**

    `FINAL ADDRESS = DISPLACEMENT + %BASE + (%INDEX * SCALE)`
- Most general form
  - □ Others are specializations of it
    - ● Direct: Displacement only
    - ● Indirect: Base only
    - ● Base pointer: Displacement and Base



- Elements
  - □ Displacement: fixed (**at assembly time**!) number

    None, 1 Byte, 2 Bytes, or 4 Bytes long (**not** 8 Bytes!)
    - □ Note: you can specify multiple numbers; assembler will add them up for your so they **MUST** be fixed (=at compile time) numbers!
  - □ Base: one of RAX, RBX, RCX, RDX, RSP, RBP, RSI, and RDI, R8-R15
  - □ Index: one of RAX, RBX, RCX, RDX, RBP, RSI, and RDI, R8-R15
  - □ Scale: (1), 2, 4, or 8

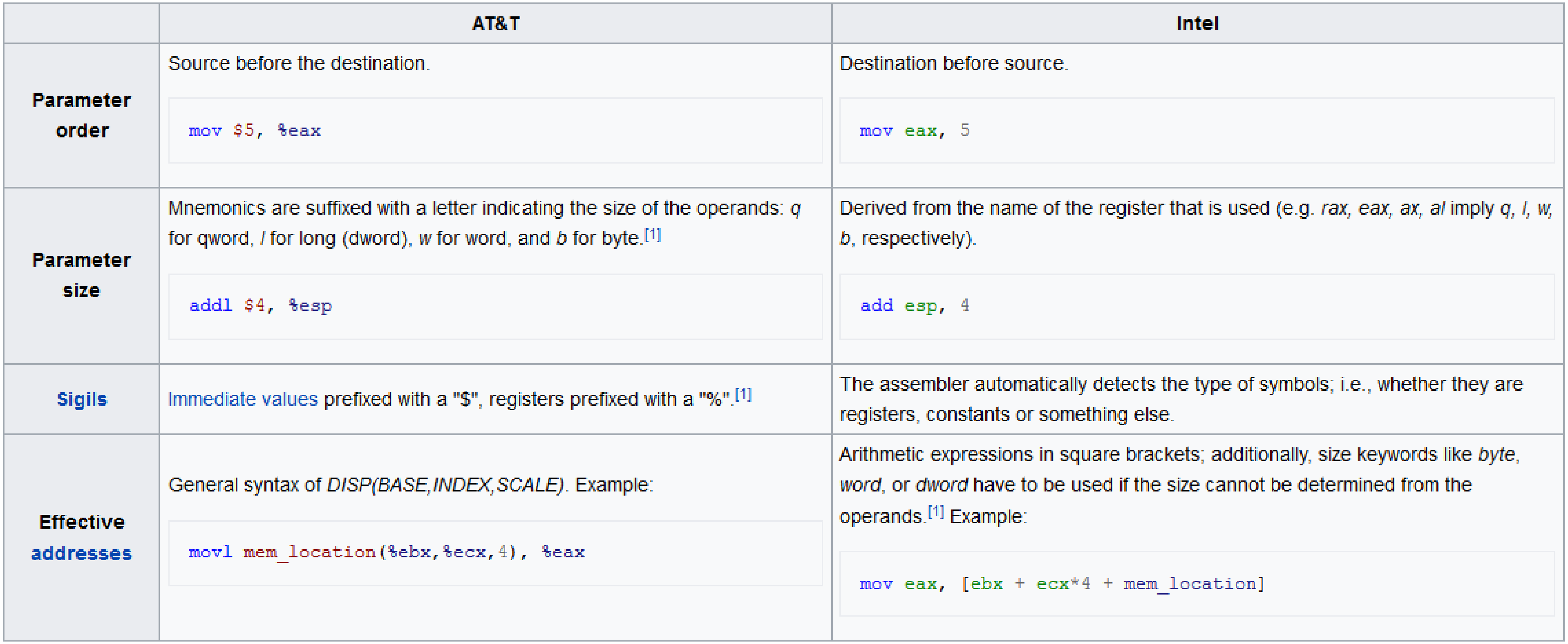
Data access methods - LEA

- What is the difference between
  `movq data_items(%rbx,%rdi,8), %rax`
  and
  `leaq data_items(%rbx,%rdi,8), %rax`
- Translation to “normal” code:

  -
    ```
    movq %rdi,%rax              # Get content of register RDI
    shlq $3,%rax                # Multiply it by 8
    addq %rbx,%rax              # Add content of register RBX
    addq $data_items,%rax       # Add immediate value/offset
    movq (%rax),%rax            # Retrieve memory content
    ```
- But what is now LEA?
  - Simple omit the last line above!
  - We just calculate the address but no not perform the indirect memory access to retrieve the content
- Trick: How could you efficiently implement `int x=y*4+z+17`?
  - Nobody forces us to actually use the “address” for a memory access…

AT&T syntax vs. Intel Syntax

- In this course, we use **AT&T syntax**
- These are the most important differences:



	AT&T	Intel
Parameter order	Source before the destination. `mov $5, %eax`	Destination before source. `mov eax, 5`
Parameter size	Mnemonics are suffixed with a letter indicating the size of the operands: *q* for qword, *l* for long (dword), *w* for word, and *b* for byte.[1] `addl $4, %esp`	Derived from the name of the register that is used (e.g. *rax*, *eax*, *ax*, *al* imply *q*, *l*, *w*, *b*, respectively). `add esp, 4`
Sigils	Immediate values prefixed with a "$", registers prefixed with a "%".[1]	The assembler automatically detects the type of symbols; i.e., whether they are registers, constants or something else.
Effective addresses	General syntax of *DISP(BASE,INDEX,SCALE)*. Example: `movl mem_location(%ebx,%ecx,4), %eax`	Arithmetic expressions in square brackets; additionally, size keywords like *byte*, *word*, or *dword* have to be used if the size cannot be determined from the operands.[1] Example: `mov eax, [ebx + ecx*4 + mem_location]`

https://en.wikipedia.org/wiki/X86_assembly_language

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Slides by: Michael Sonntag, adaptions by Michael Roland
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)