Low level attacks
Assembly (part 1)

Mario Alviano

University of Calabria, Italy

A.Y. 2017/2018

What is assembly language?

- The CPU manages arithmetical, logical, and control activities
- The CPU follows machine language instructions
- Machine language instructions are strings in $\{0, 1\}^*$
- Assembly is almost one-to-one to machine language

Why studying an assembly language?

To understand the following:

- How programs interface with OS, processor, and BIOS
- How data is represented in memory and other external devices
- How the processor accesses and executes instruction
- How instructions access and process data
- How a program accesses external devices

Setup

Download and install NASM

`http://www.nasm.us/`

Example

Try hello.asm

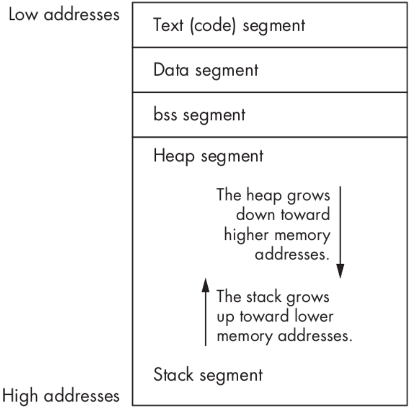
- Assemble: `nasm -f elf hello.asm`
- Link: `ld -m elf_i386 -o hello hello.o`
- Run: `./hello`

Basic syntax

Three sections:

- `section .text`
  - Actual code to be executed
  - Entry point declared by `global _start`
- `section .data`
  - Global initialized variables
- `section .bss`
  - Global unitialized variables

Memory segments



- Text: assembly code
- Data: global initialized variables
- BSS: global unitialized variables
- Heap: dynamically allocated memory
- Stack: local (and temporary) memory

Statements (1)

Three types:

- Executable instructions or instructions
  - Consist of an operation code and up to 3 arguments
  - Each instruction generates one machine language instruction
- Assembler directives or pseudo-ops
  - Used by the assembler
  - Do not generate machine language instructions
- Macros
  - Text substitution

Statements (2)

Syntax

```
[label] mnemonic [operands] [;comment]
```

Examples of assembly language statements

- Increment the value of variable count

  ```
  inc count
  ```

- Move value 0 into variable count

  ```
  mov count, 0
  ```

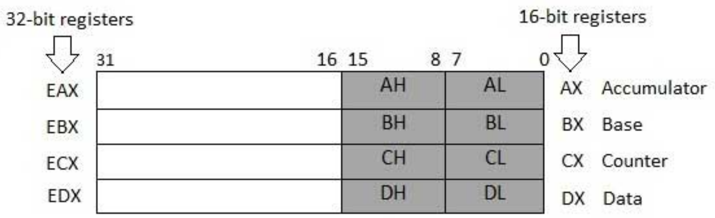
- Add the value stored in register ebx to the value stored in register eax

  ```
  add eax, ebx
  ```

Registers of an x86 processor

- General registers
  - Data registers
  - Pointer registers
  - Index registers
- Control registers
- Segment registers

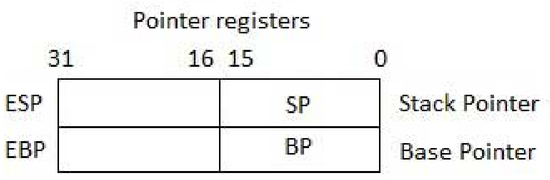
Data registers



- Four 32-bit data registers
- Used for arithmetic, logical and other operations
- Can be also used as 16-bit or 8-bit data registers

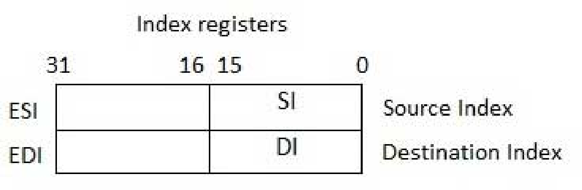
AX, BX, CX, DX use bits 0-15

Pointer registers



- Three 32-bit pointer registers
  - ESP: address of current top stack element
  - EBP: address of the stack frame
- Can be also used as 16-bit pointer registers

Index registers



- Two 32-bit index registers
- Used for addressing memory
- Can be also used as 16-bit pointer registers

Control registers

- EIP: 32-bit instruction pointer register
  - Address of the next instruction to be executed
  - Can be also used as 16-bit IP register

Flag:					O	D	I	T	S	Z		A		P		C
Bit no:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- EFLAGS: 32-bit flags register
  - Overflow Flag (OF): 1 if the last arith. op. overflowed
  - Direction Flag (DF): left-to-right (0) or right-to-left (1) processing of strings
  - Interrupt Flag (IF): ignore (0) or process (1) external interrupts
  - Trap Flag (TF): 1 for single-step execution (to debug)
  - Sign Flag (SF): 0 if the last arith. op. gave a positive result
  - Zero Flag (ZF): 1 if the last arith. op. gave 0
  - Auxiliary Carry Flag (AF): the carry from bit 3 to bit 4 in the last arith. op.
  - Parity Flag (PF): parity bit of the last arith. op.
  - Carry Flag (CF): the carry of the high-order bit in the last arith. op.

Segment registers

Registers pointing to starting addresses of memory segments

- Code Segment (CS)
- Data Segment (DS)
- Stack Segment (SS)
- Extra Segments (ES, FS, GS)

Example

Try `9starts.asm`, focusing on the use of registers.

Linux system calls (1)

- Put the system call number in the EAX register
- Store arguments in EBX, ECX, EDX, ESI, EDI, EBP
  - If there are more than 6 arguments, store the address of the first argument in EBX
- Trigger the interrupt 0x80
- The result is returned in EAX

Linux system calls (2)

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

All system calls are listed in...

`/usr/include/asm/unistd.h`

Example

Try `read_number.asm`, focusing on the system calls.

Addressing modes (1)

- Instructions may have up to 3 operands
- First operand is generally the destination
- Several addressing modes
  - Register addressing: use of register values
  - Immediate addressing: use of constants (with type specifier)
  - Memory addressing: e.g., use of square brakets

Type Specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Addressing modes (2)

`mov destination, source`

- `mov register, register`
- `mov register, immediate`
- `mov register, memory`
- `mov memory, register`
- `mov memory, immediate`

Example

Try `mov.asm`, focusing on the different forms of the `mov` instruction.

Variables

- Use D* to declare initialized global variables
- Use RES* to reserve space for unitialized global variables
- * is one of the following:
  - B: byte
  - W: word
  - D: double word
  - Q: quadword
  - T: ten bytes
- `times` can be used to repeat several times the same initialization
  - e.g., `starts times 9 db '*'` allocates 9 bytes with value '*********'

Constants

- `constant_name equu expression`
  Cannot be redefined
- `%assign constant_name expression`
  Can be redefined
- `%define constant_name string`
  Can be redefined

Example

Try `constants.asm`, focusing the definition of constants.

Arithmetic instructions (1)

- `inc destination`
- `dec destination`
- `add destination, source`
- `sub destination, source`

At least one operand must be different from memory address

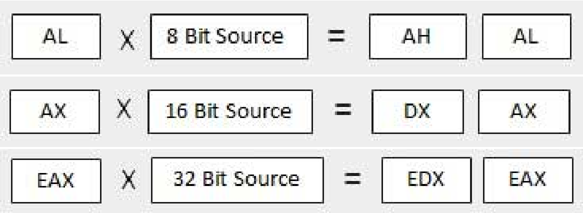
Example

Try `arith1.asm`

Arithmetic instructions (2)

- `mul multiplier` (unsigned integers, or natural numbers)
- `imul multiplier` (signed integers, or integers)

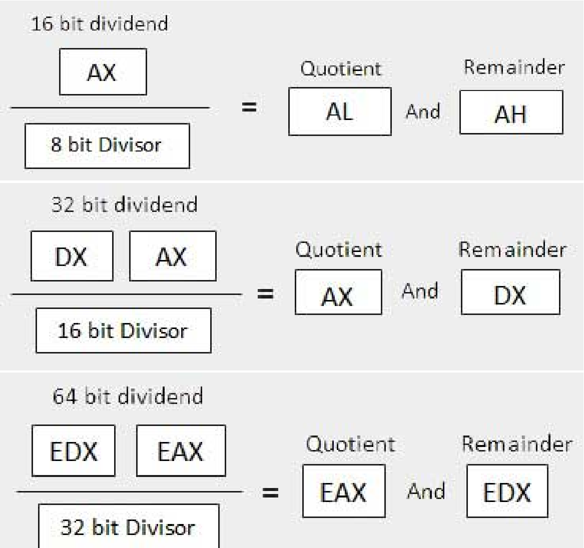
Some operands are implicit depending on the size of the multiplier



Example

Try `arith2.asm`

- `div divisor` (unsigned integers, or natural numbers)
- `idiv divisor` (signed integers, or integers)



Logical instructions

Bitwise logical operations, storing the result in `operand1`:

- `and operand1, operand2`
- `or operand1, operand2`
- `xor operand1, operand2`
- `not operand1`

Bitwise AND, just setting flags (e.g., ZF is set to 1 if the AND is 0)

- `test operand1, operand2`

Unconditional jump

- `jmp label`
  Set IP to the address of the given label

```
MOV   AX, 00     ; Initializing AX to 0
MOV   BX, 00     ; Initializing BX to 0
MOV   CX, 01     ; Initializing CX to 1
L20:
ADD   AX, 01     ; Increment AX
ADD   BX, AX     ; Add AX to BX
SHL   CX, 1      ; shift left CX, this in turn doubles the CX value
JMP   L20        ; repeats the statements
```

Conditional jump

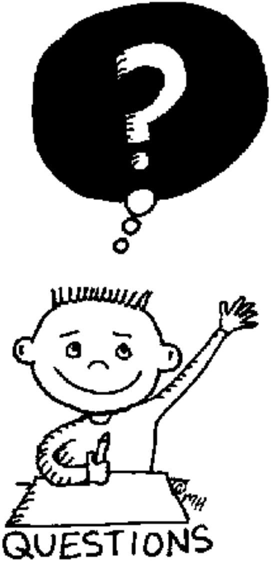
- `j<condition> label`

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater/Equal or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

- Often preceded by `cmp operand1, operand2`
- It is like `sub`, but `operand1` is not changed
- Only flags are affected

```
INC     EDX
CMP     EDX, 10 ; Compares whether the counter has reached 10
JLE     LP1     ; If it is less than or equal to 10, then jump to LP1
```

Example: Try `jumps.asm`



END OF THE LECTURE