

x86 Assembly Programming

Azlan Mukhtar

- **x86 Basic Architecture**

- Introduction
- Endianness
- Registers and Flags
- Memory Addressing
- Stacks
- Signed number representations

- **Assembly Programming**

- Data movement
- Arithmetic operation
- Control structure
- Bitwise operation
- Data/Array manipulations
- Assembler

Introduction to x86

- Started with 8086 in 1978
- Continued with 8088, 80186, 80286, 386, 486, Pentium, 686... Intel Core
- CISC architecture
- 32-bit is called x86-32 or IA-32
- 64-bit is called x86-64, AMD64, Intel 64

Intel 80386

- Introduced in 1986
- Has a 32-bit word length
- Has eight general-purpose registers
- Supports paging and virtual memory
- Addresses up to 4GiB of memory
- Base for current x86 32 bit architecture

Byte Order/Endianness

- Little Endian
 - On memory: 78 56 34 12
 - Actual value: 0x12345678
- Big Endian
 - On Memory: 12 34 56 78
 - Actual value: 0x12345678

String Styles

- ASCIIZ – C-style

| | | | | | |
|----|----|----|----|----|----|
| H | e | l | l | o | |
| 48 | 65 | 6C | 6C | 6F | 00 |

- Null-terminated Unicode

| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| H | | e | | l | | l | | o | | | |
| 48 | 00 | 65 | 00 | 6C | 00 | 6C | 00 | 6F | 00 | 00 | 00 |

- Pascal

| | | | | | |
|----|----|----|----|----|----|
| | H | e | l | l | o |
| 05 | 48 | 65 | 6C | 6C | 6F |

- Delphi

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| | | H | e | l | l | o | | |
| 05 | 00 | 00 | 00 | 48 | 65 | 6C | 6C | 6F |

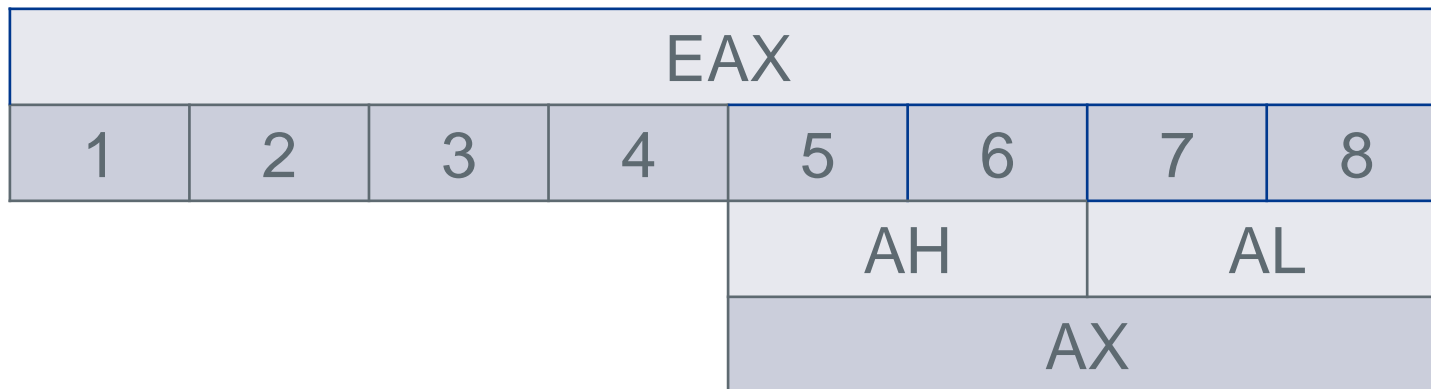
Data Register Layout

General-Purpose Registers

| 31 | 16 | 15 | 8 | 7 | 0 | 16-bit | 32-bit |
|----|----|----|----|---|---|--------|--------|
| | AH | | AL | | | AX | EAX |
| | BH | | BL | | | BX | EBX |
| | CH | | CL | | | CX | ECX |
| | DH | | DL | | | DX | EDX |
| | BP | | | | | | EBP |
| | SI | | | | | | ESI |
| | DI | | | | | | EDI |
| | SP | | | | | | ESP |

Register Layout - Example

- EAX = 12345678h
- AX = 5678h
- AH = 56h
- AL = 78h



* h is for hexadecimal notation

Data Registers

| Register | Description | Usage |
|--------------------|----------------------|--------------------------------|
| AL / AH / AX/ EAX | Accumulator Register | Arithmetic operations |
| BL / BH / BX / EBX | Base register | General data storage, index |
| CL / CH / CX / ECX | Counter register | Loop constructs |
| DL / DH / DX / EDX | Data register | Arithmetic |

Data Register - Example

- Accumulator
- `ADD EAX, EBX` ; add the value in EBX and EAX, and store the
; result into EAX
- `SUB EAX, EBX` ; substitute the value in EBX from EAX, and
; store the result into EAX
- `IMUL EAX, EBX` ; multiply EAX and EBX, and store the result
; into EAX
- More about this later

Address Registers

| Register | Description | Usage |
|----------|---------------------|--|
| IP / EIP | Instruction Pointer | Program execution counter |
| SP / ESP | Stack Pointer | ESP will hold an offset to top of stacks memory location |
| BP / EBP | Base Pointer | Stack frame |
| SI / ESI | Source Index | String operation |
| DI / EDI | Destination Index | String operation |

Segment Registers

| Register | Description | Usage |
|-------------|----------------|--------------|
| CS | Code Segment | Program code |
| DS | Data Segment | Program data |
| SS | Stack Segment | Stack |
| ES / FS /GS | Other Segments | Other uses |

Segment Registers

- Modern Operating System such as Windows, MacOSX, *nix are using protected mode flat model and segment registers are less relevant. CS, DS, SS, and ES are pointing to the same location, which is 0 offset.
- Windows NT is utilizing FS register to store Process Environment Block and Thread Information Block (TIB)
 - https://en.wikipedia.org/wiki/Process_Environment_Block
 - http://en.wikipedia.org/wiki/Win32_Thread_Information_Block

EFLAGS Register (bit 0 – 15)

| | |
|---|--------------|
| S | Status Flag |
| C | Control Flag |
| X | System Flag |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|------|----|----|----|----|----|----|----|---|----|---|----|---|----|
| 0 | NT | IOPL | | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | 1 | CF |

| | | | | | | | | | | | | | | | |
|---|-----------------------|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| X | Nested Task | | | | | | | | | | | | | | |
| X | IO Privilege Level | | | | | | | | | | | | | | |
| S | Overflow Flag | | | | | | | | | | | | | | |
| C | Direction Flag | | | | | | | | | | | | | | |
| X | Interrupt Enable Flag | | | | | | | | | | | | | | |
| X | Trap Flag | | | | | | | | | | | | | | |
| S | Sign Flag | | | | | | | | | | | | | | |
| S | Zero Flag | | | | | | | | | | | | | | |
| S | Auxiliary Carry Flag | | | | | | | | | | | | | | |
| S | Parity Flag | | | | | | | | | | | | | | |
| S | Carry Flag | | | | | | | | | | | | | | |

EFLAGS Register (bit 16 – 31)

| | |
|---|--------------|
| S | Status Flag |
| C | Control Flag |
| X | System Flag |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|----|----|----|----|----|----|----|----|----|----|----|-----|-----|----|----|----|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ID | VIP | VIF | AC | VM | RF |

| | |
|---|---------------------------|
| X | ID Flag |
| X | Virtual Interrupt Pending |
| X | Virtual Interrupt Flag |
| X | Alignment Check |
| X | Virtual 8086 Mode |
| X | Resume Flag |

EFLAGS Register – Status Flags

- ZF - Zero flag - Set if the result is zero; cleared otherwise.
- SF - Sign flag - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
- CF - Carry flag - Set if an arithmetic operation generates a carry or a borrow out of the most significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.
- OF - Overflow flag - Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.
- AF - Adjust flag - Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic. (Rarely Used)
- PF - Parity flag - Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise. (Rarely Used)

EFLAGS Registers – DF FLAG

- The direction flag (DF, located in bit 10 of the EFLAGS register) controls string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).
- The STD and CLD instructions set and clear the DF flag, respectively.

EFLAGS Register – Flags Changing Instructions

- Instruction that affecting flags
 - ADD/SUB – Modifies Flags: AF CF OF PF SF ZF
 - CMP- Modifies Flags: AF CF OF PF SF ZF
 - TEST - Modifies Flags: CF OF PF SF ZF (AF undefined)
 - CLD/STD – Modifies Flags: DF

Signed number representations

– Two's Complement

- 8 bit two's complement

| Binary Value | Hex Value | Two's Complement | Unsigned |
|--------------|-----------|------------------|----------|
| 00000000 | 0x00 | 0 | 0 |
| 00000001 | 0x01 | 1 | 1 |
| ... | ... | ... | .. |
| 01111110 | 0x7E | 126 | 126 |
| 01111111 | 0x7F | 127 | 127 |
| 10000000 | 0x80 | -128 | 128 |
| 10000001 | 0x81 | -127 | 129 |
| 10000010 | 0x82 | -126 | 130 |
| ... | ... | ... | ... |
| 11111110 | 0xFE | -2 | 254 |
| 11111111 | 0xFF | -1 | 255 |

Two's Complement – How to convert

- Method one

| Steps | Example 1 | Example 2 |
|---|-----------------|------------------|
| 1. Starting from the right, find the first '1' | 010100 1 | 0101 1 00 |
| 2. Invert all of the bits to the left of that one | 11010111 | 1010100 |

- Method two

| Steps | 00000001 |
|---|----------|
| 1. Invert all the bits through the number | 11111110 |
| 2. Add one | 11111111 |

- Method three
 - Use x86 instruction for Two's Complement Negation, NEG
 - Example – NEG EAX

Two's Complement - examples

Example 1:

```
mov al, 0x05
```

```
add al, 0xFE ; 0xFE = -2 or 0xFE = 254
```

```
⇒ al = 3
```

Example 2:

```
mov al, 0x02
```

```
add al, 0xFA ; 0xFA = -6 or 0xFA = 250
```

```
⇒ al = 0xFC ; 0xFC = -4 or 0xFC = 252
```

Basic x86 Assembly instructions

- Data Transfers
- Arithmetic
- Logic
- Jumps
- Misc

Reading the x86 Manual (RTxM)

MOVZX - Move with Zero-Extend

| | | |
|-----------|-----------------|---|
| 0F B6 / r | MOVZX r16,r/m8 | Move byte to word with zero-extension |
| 0F B6 / r | MOVZX r32,r/m8 | Move byte to doubleword, zero-extension |
| 0F B7 / r | MOVZX r32,r/m16 | Move word to doubleword, zero-extension |

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

| Operands | Bytes | Clocks |
|----------|--------------|--------|
| reg, reg | 3 | 3 NP |
| reg, mem | 3+d(0,1,2,4) | 3 NP |

(Note: destination reg is 16 or 32-bits; source is 8 or 16 bits)

Flags

None.

Reading the x86 Manual – cont.

| | |
|-----|--|
| r | Any general register |
| r8 | One of the byte general-purpose registers AL, CL, DL, BL, AH, CH, DH, or BH. |
| r16 | One of the word general-purpose registers AX, CX, DX, BX, SP, BP, SI, or DI. |
| r32 | One of the doubleword general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI. |
| m | A 16- or 32-bit operand in memory. |
| m8 | A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions and the XLAT instruction. |
| m16 | A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions. |

....

And so on, please refer the docs.

Data Transfer Instructions

- MOV - Move data between general-purpose registers; move data between memory and general purpose or segment registers; move immediates to general-purpose registers
- MOVSX - Move and sign extend
- MOVZX - Move and zero extend
- PUSH - Push onto stack
- POP - Pop off of stack
- XCHG - Exchange
- BSWAP - Byte swap - converts between 32 bit little endian and big endian values
- PUSHA/PUSHAD - Push general-purpose registers onto stack
- POPA/POPAD - Pop general-purpose registers from stack
- CWD/CDQ - Convert word to doubleword/Convert doubleword to quadword
- CBW/CWDE - Convert byte to word/Convert word to doubleword in EAX register

Binary Arithmetic Instructions

- ADD - Integer add
- ADC - Add with carry
- SUB - Subtract
- SBB - Subtract with borrow
- IMUL - Signed multiply
- MUL - Unsigned multiply
- IDIV - Signed divide
- DIV - Unsigned divide
- INC - Increment
- DEC - Decrement
- CMP - Compare – (Similar to SUB but the result is not stored (discarded) and the flags are changed)
- NEG - Negate

Logical Instructions

- AND - Perform bitwise logical AND
- OR - Perform bitwise logical OR
- XOR - Perform bitwise logical exclusive OR
- NOT - Perform bitwise logical NOT

Shift and Rotate Instructions

- SAR - Shift arithmetic right
- SHR - Shift logical right
- SAL/SHL - Shift arithmetic left/Shift logical left
- SHRD - Shift right double
- SHLD - Shift left double
- ROR - Rotate right
- ROL - Rotate left
- RCR - Rotate through carry right
- RCL - Rotate through carry left

Control Transfer Instructions

- JMP - Jump
- JE/JZ - Jump if equal/Jump if zero
- JNE/JNZ - Jump if not equal/Jump if not zero
- JA/JNBE - Jump if above/Jump if not below or equal
- JAE/JNB - Jump if above or equal/Jump if not below
- JB/JNAE - Jump if below/Jump if not above or equal
- JBE/JNA - Jump if below or equal/Jump if not above
- JG/JNLE - Jump if greater/Jump if not less or equal
- JGE/JNL - Jump if greater or equal/Jump if not less
- JL/JNGE - Jump if less/Jump if not greater or equal
- JLE/JNG - Jump if less or equal/Jump if not greater
- JC - Jump if carry

Control Transfer Instructions - continue

- JNC - Jump if not carry
- JO - Jump if overflow
- JNO - Jump if not overflow
- JS - Jump if sign (negative)
- JNS - Jump if not sign (non-negative)
- JPO/JNP - Jump if parity odd/Jump if not parity
- JPE/JP - Jump if parity even/Jump if parity
- JCXZ/JECXZ - Jump register CX zero/Jump register ECX zero
- LOOP - Loop with ECX counter
- LOOPZ/LOOPE - Loop with ECX and zero/Loop with ECX and equal
- LOOPNZ/LOOPNE - Loop with ECX and not zero/Loop with ECX and not equal

Control Transfer Instructions - continue

- CALL Call procedure
- RET Return
- IRET Return from interrupt
- INT Software interrupt
- ENTER High-level procedure entry
- LEAVE High-level procedure exit

String Instructions

- MOVS/MOVS_B Move string/Move byte string
- MOVS/MOVS_W Move string/Move word string
- MOVS/MOVS_D Move string/Move doubleword string
- CMPS/CMPS_B Compare string/Compare byte string
- CMPS/CMPS_W Compare string/Compare word string
- CMPS/CMPS_D Compare string/Compare doubleword string
- SCAS/SCAS_B Scan string/Scan byte string
- SCAS/SCAS_W Scan string/Scan word string
- SCAS/SCAS_D Scan string/Scan doubleword string

String Instructions - Continue

- LODS/LODSB Load string/Load byte string
- LODS/LODSW Load string/Load word string
- LODS/LODSD Load string/Load doubleword string
- STOS/STOSB Store string/Store byte string
- STOS/STOSW Store string/Store word string
- STOS/STOSD Store string/Store doubleword string
- REP Repeat while ECX not zero
- REPE/REPZ Repeat while equal/Repeat while zero
- REPNE/REPNZ Repeat while not equal/Repeat while not zero

Miscellaneous Instructions

- LEA Load effective address
- NOP No operation

Examples of Mnemonics

| mnemonic argument1, argument2, argument3 | |
|--|--------------------------------|
| MOV EAX, 1 | Move 1 to EAX |
| ADD EDX, 5 | Add 5 to EDX |
| SUB EBX, 2 | Subtract 2 from EBX |
| AND ECX, 0 | Bit-wise AND 0 to ECX |
| XOR EDX, 4 | Bit-wise eXclusive OR 4 to EDX |
| SHL ECX, 6 | Shift ECX left by six |
| ROR EBX, 3 | Bit-wise rotate EBX right by 3 |
| INC ECX | Increment ECX |

Addressing Memory



- `mov eax, [ebx]`
; Move the 4 bytes in memory at the address contained in EBX into EAX
- `mov [var], ebx`
; Move the contents of EBX into the 4 bytes at memory address var.
(Note, var is a 32-bit constant).
- `mov eax, [esi-4]`
; Move 4 bytes at memory address ESI + (-4) into EAX
- `mov [esi+eax], cl`
; Move the contents of CL into the byte at address ESI+EAX
- `mov edx, [esi+4*ebx]`
; Move the 4 bytes of data at address ESI+4*EBX into EDX

Addressing Memory – With Size Directive

- `mov BYTE PTR [ebx], 2`
; Move 2 into the single byte at the address stored in EBX.
- `mov WORD PTR [ebx], 2`
; Move the 16-bit integer representation of 2 into the 2 bytes starting at the address in EBX.
- `mov DWORD PTR [ebx], 2`
; Move the 32-bit integer representation of 2 into the 4 bytes starting at the address in EBX.

Microsoft Macro Assembler

- Why MASM
 - Free (comes with Visual C++ Express/Community)
 - Visual Studio IDE for Coding and Debugging

Microsoft Macro Assembler (MASM)

```
.386
.model flat, stdcall
option casemap :none    ; case sensitive

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc

includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
szTitle db 'Win32', 0
szMessage db 'Hello World', 0

.code
start:
push 0                ; uType = MB_OK
push offset szTitle   ; LPCSTR lpCaption
push offset szMessage ; LPCSTR lpText
push 0                ; hWnd = HWND_DESKTOP
call MessageBoxA
push 0
call ExitProcess

end start
```

Netwide Assembler (NASM) - Alternative

```
; Assemble: nasm -f win32 hello_nasm.asm
; Link: link hello_nasm.obj user32.lib /entry:main
; (using Visual C++'s linker)

extern _MessageBoxA@16

segment .text
global _main

_main:
    push dword 0      ; uType = MB_OK
    push szTitle      ; LPCSTR lpCaption
    push szMessage     ; LPCSTR lpText
    push dword 0      ; hWnd = HWND_DESKTOP
    call _MessageBoxA@16
    ret

segment .data
szMessage: db 'Hello World',0
szTitle: db 'Program Title',0
```


FASM – Arithmetic – Alternative

```
format PE Console 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
start:
xor    eax, eax
add    eax, 8
mov    ecx, 2
sub    eax, ecx
mul    ecx
push   eax
push   szResult
call   [printf]
add    esp, 8
ret

section '.data' data readable writeable
szResult    db "The result is: %d",10,0

section '.idata' import data readable
library msvcrt,'msvcrt.dll'

import msvcrt,\
printf,'printf'
```

Procedure & Calling Conventions

- Procedure Call
- Global & Local Variables
- CDECL
- STDCALL
- FASTCALL

Procedure Call

- Procedural programming is derived from structured programming, based upon the concept of the procedure call
- Procedures, also known as routines, subroutines, methods, or functions
- x86 architecture supports procedure call
- The processor supports procedure calls in the following two different ways:
- CALL and RET instructions.
- ENTER and LEAVE instructions, in conjunction with the CALL and RET instructions
- The procedure stack, commonly referred to simply as “the stack”, will save the state of the calling procedure, **pass parameters** to the called procedure, and **store local variables** for the currently executing procedure

Global And Local Variable

```
// global_var.c

#include <stdio.h>

int global_var1 = 5;

int main ()
{
    int local_var1 = 1;
    int local_var2 = 2;
    int result = 0;
    result = global_var1 + local_var1 + local_var2;

    return 0;
}
```

Compilation cmd line:
cl global_var.c /Faglobal_var.asm

CDECL

// C/C++ codes

```
_cdecl int MyFunction1(int a,  
int b)  
{  
    return a + b;  
}
```

```
x = MyFunction1(2, 3);
```

; x86 asm codes

```
_MyFunction1:  
push ebp  
mov ebp, esp  
mov eax, [ebp + 8]  
mov edx, [ebp + 12]  
add eax, edx  
pop ebp  
ret
```

```
Start:  
push 3  
push 2  
call _MyFunction1  
add esp, 8      ; clean up
```

STDCALL

// C/C++ codes

```
_stdcall int  
MyFunction2(int a, int b)  
{  
    return a + b;  
}  
  
x = MyFunction2(2, 3);
```

; x86 asm codes

```
:_MyFunction@8  
push ebp  
mov ebp, esp  
mov eax, [ebp + 8]  
mov edx, [ebp + 12]  
add eax, edx  
pop ebp  
ret 8 ; clean up
```

```
Start:  
push 3  
push 2  
call _MyFunction@8
```

FASTCALL

// C/C++ codes

```
_fastcall int  
MyFunction3(int a, int b)  
{  
    return a + b;  
}  
  
x = MyFunction3(2, 3);
```

; x86 asm codes

```
:@MyFunction3@8  
push ebp  
add eax, edx  
;a is in eax, b is in edx  
pop ebp  
ret
```

Start:

```
mov eax, 2  
mov edx, 3  
call @MyFunction3@8
```

References

Intel® 64 and IA-32 Architectures Software Developer Manuals

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

80x86 Instruction Set (simplified)

<http://www.penguin.cz/~literakl/intel/intel.html>

80x86 Instruction Set cheatsheet

<http://www.jegerlehner.ch/intel/>

x86 Assembly

http://en.wikibooks.org/wiki/X86_Assembly

http://en.wikibooks.org/wiki/X86_Disassembly

<http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>