Introduction to X86 Assembly

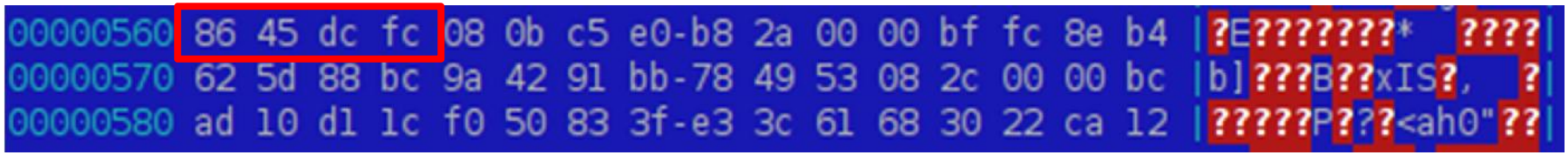
F-Secure

Content

- Numbers and strings
- Registers
- Data movement instructions
- Control flow instructions
- Arithmetic Instructions
- Bitwise operations

Byte order or endianness

- x86 is little endian
  - Least significant byte in the lowest address
  - 86 45 dc fc == 0xfcdc4586



F-Secure

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
```

Registers

Registers

- EAX is 32 bits
- AX is the lower 16 bits of EAX
- AH is the higher 8 bits of AX, and AL the lower 8 bits of AX

31 16	15 8	7 0
EAX		
	AX	
	AH	AL
EBX	BH	BL
ECX	CH	CL
EDX	DH	DL
EDI		
ESI		
ESP		
EBP		

Registers

- EAX: Accumulator
  - Imul and idiv store the result to accumulator
  - stos, lods, xlat, etc move data in and out of accumulator
- EDX: Data/general
  - Multiplication, division etc store the most significant bits to EDX
- EBX: Base index
  - xlat instruction uses EBX as a base
- ECX: Counter
  - Automatically decreased by loop & rep

Registers

- EDI: Destination Index for string operations, e.g. stos, scas
- ESI: Source Index for string operations, e.g. lods
- ESP: Stack pointer for top address of the stack
  - Push, pop, call and ret modify esp indirectly
- EBP: Stack base pointer for base address of the current stack frame
  - Will be explainted in more detail later

Data Movement Instructions

Mov

- Move data
  - mov <reg>,<reg>
  - mov <reg>,<mem>
  - mov <mem>,<reg>
  - mov <reg>,<const>
  - mov <mem>,<const>

- Direct memory-to-memory moves are not possible

- Examples:
  - mov eax, ebx ; copy the value in ebx to eax

Lea

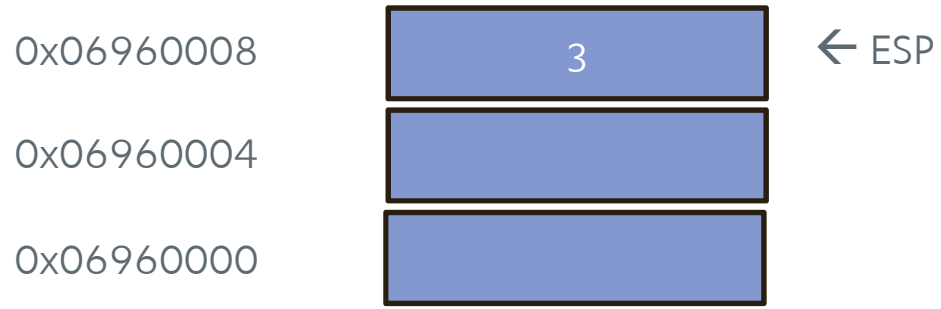
- Place the *address* specified by its second operand into the register specified by its first operand
  - lea <reg32>,<mem>
- Lea is sometimes used to perform simple arithmetic
- Examples:
  - lea edi, [ebx+4*esi] ; value EBX+4*ESI is placed into EDI

Stack

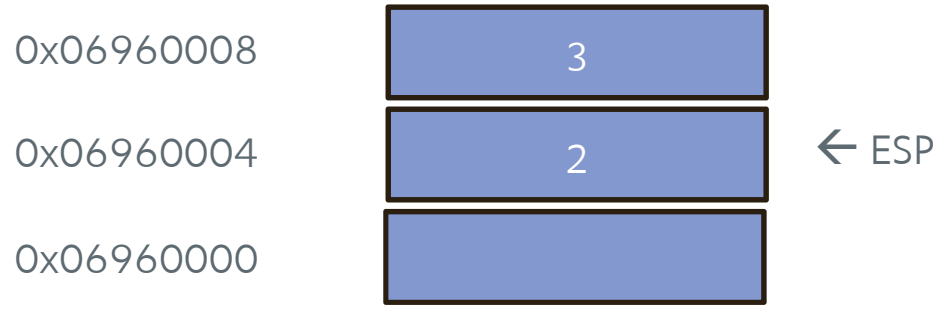
- The stack is LIFO – Last In, First Out
- Push to add data to the stack, pop to read from it
- In x86, ESP points to the top of the stack
- The stack grows “down” – pushing items subtracts from the ESP, popping adds
- Understanding how the stack is being used is very important in reversing
  - The stack is used to pass arguments to functions and to track control flow when calls are made

Stack

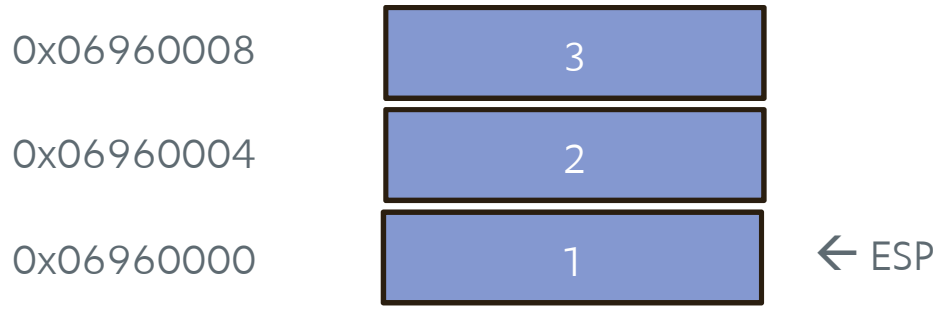
1. push 0x3



2. push 0x2

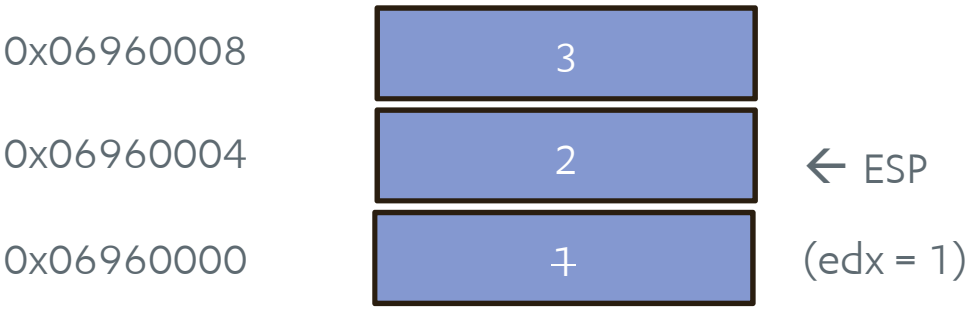


3. push 0x1

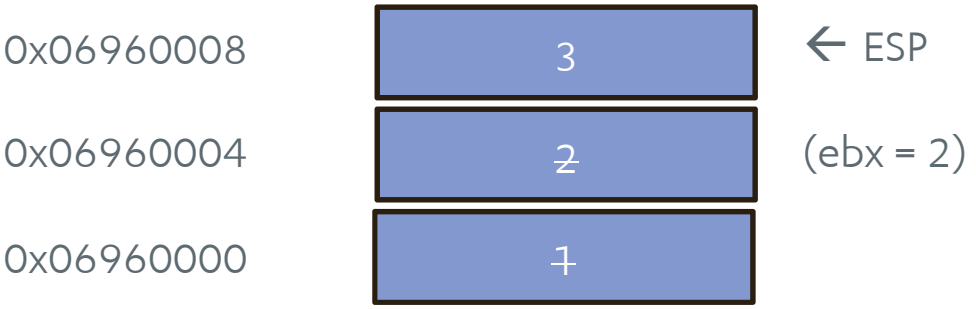


Stack

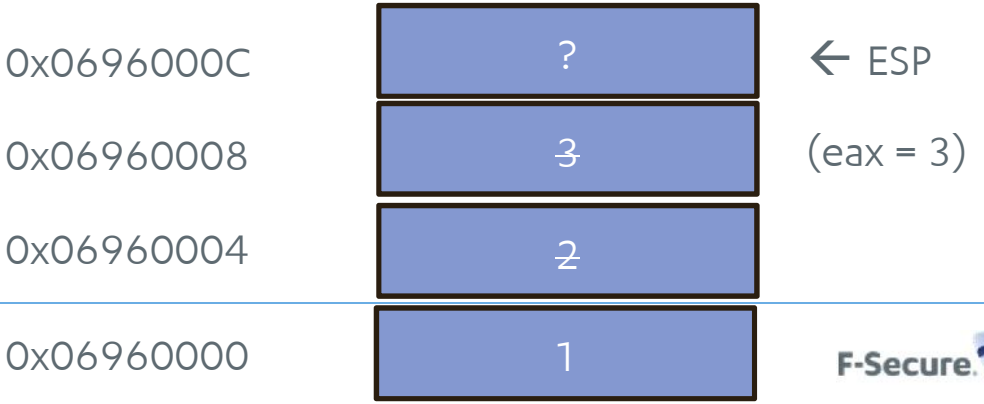
4. pop edx



5. pop ebx



6. pop eax



Push

- Push: place the operand to stack
  - push <reg32>
  - push <mem>
  - push <con32>

- Push first decrements ESP by 4, then places the operand into the address where ESP points to

- Examples:
  - push eax ; push eax to stack

Pop

- Pop: remove the 4-byte element from the top of the stack, and put it into the specified register or memory location
  - pop <reg32>
  - pop <mem>
- Pop first moves the data, then increments ESP by four
- Examples:
  - pop eax ; pop the top element of the stack to EAX

Stosb

- Stores the byte in AL at [EDI] and increments/decrements EDI
- Whether EDI is incremented or decremented depends on the direction flag which can be set with instruction and cleared with instruction
  - cld = clear direction flag = go forward, increment EDI
  - std = set direction flag = go backwards, decrement EDI
- REP prefix may be used to repeat the instruction ECX times
- Stosw = store the word in AX to [EDI]
- Stosd = store the doubleword in EAX to [EDI]

Loadsb

- Load the byte at [ESI] into AL and increments/decrements ESI

- Whether EDI is incremented or decremented depends on the direction flag which can be set with instruction and cleared with instruction
  - cld = clear direction flag = go forward, increment ESI
  - std = set direction flag = go backwards, decrement ESI

- REP prefix may be used to repeat the instruction ECX times

- Lodsw = load word to AX
- Lodsd = load doubleword to EAX

Control Flow Instructions

Call & ret

- Call first pushes the return address (i.e. address of the next instruction) and then performs an unconditional jump
  - call <label>

- Ret first pops the return address from the stack and then performs an unconditional jump
  - ret <const>
  - The constant is optional. It can be used to further increase the stack pointer to, for example, clean-up the arguments from the stack

Jmp

- Unconditional jump to given address
  - jmp <label>
  - jmp <const>
  - jmp <reg>
  - jmp <mem>

- The address may be specified as an offset, or as a relative jump

Cmp

- Compare the values of the two specified operands and sets the relevant flags
  - cmp <reg>,<reg>
  - cmp <reg>,<mem>
  - cmp <mem>,<reg>
  - cmp <reg>,<const>

- Equivalent to the sub instruction, except the result of the subtraction is discarded

- Example

  cmp dword ptr [some_variable], 10

  jeq is_ten

Test

- Performs a `mental' bitwise AND of its two operands, and affects the flags as if the operation had taken place, but does not store the result of the operation anywhere

- Is commonly used to test whether something is zero or not

- Example:

  test eax, eax

  jz eax_was_zero

Jcondition

- Jump if condition is satisfied. Condition check is based on flag values
  - je <label> (jump when equal)
  - jne <label> (jump when not equal)
  - jz <label> (jump when last result was zero)
  - jg <label> (jump when greater than)
  - jge <label> (jump when greater than or equal to)
  - jl <label> (jump when less than)
  - jle <label> (jump when less than or equal to)

- Example:

  cmp eax, 3

  je eax_is_three

Loop

- Decrements its counter register (ECX) by one, and if the counter does not become zero as a result of this operation, it jumps to the given label

- Example:

```
mov ecx, 10

ten_times:

    ; do some stuff here

    loop ten_times
```

Arithmetic Instructions

Add

- Add the two operands together and store the result to the first one
  - add <reg>,<reg>
  - add <reg>,<mem>
  - add <mem>,<reg>
  - add <reg>,<con>
  - add <mem>,<con>

- Example:
  - add eax, ebx ; eax = eax + ebx

Sub

- Substracts the second operand from the first, store the result to first
  - sub <reg>,<reg>
  - sub <reg>,<mem>
  - sub <mem>,<reg>
  - sub <reg>,<con>
  - sub <mem>,<con>

- Example:
  - sub eax, ebx ; eax = eax - ebx

Imul

- Signed integer multiplication
  - imul <reg32>
  - imul <mem>
  - imul <reg32>,<reg32>
  - imul <reg32>,<mem>
  - imul <reg32>,<reg32>,<con>
  - imul <reg32>,<mem>,<con>

- Examples:
  - imul ebx ; edx:eax = eax * ebx (edx = high 32 bits, eax = low 32 bits)
  - imul ebx, edx ; ebx = ebx * edx
  - imul esi, edi, 42 ; esi = edi * 42

Idiv

- Signed integer division. Divides the contents of the 64 bit integer EDX:EAX by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX.
  - idiv <reg32>
  - idiv <mem>

- Examples:
  - idiv ebx ; eax = edx:eax/ebx (quotient), edx = edx:eax % ebx (remainder)

Bitwise Operations

Xor, and, or

- Perform the logical operation, place the result in the first operand
  - xor/and/or <reg>,<reg>
  - xor/and/or <reg>,<mem>
  - xor/and/or <mem>,<reg>
  - xor/and/or <reg>,<con>
  - xor/and/or <mem>,<con>

Shl, shr

- Shl= shift left, shr = shift right

- Shl and shr perform a logical shift operation on the given source/destination (first) operand. The vacated bits are filled with zero.

- Shl can be used to multiple with powers of two. Example:
  - shl eax, 4 ; multiply eax by 16 (2^4)

- Shr can be used to perform integer division with power of two. Example:
  - shr eax, 8 ; divide by 256 (2 ^8)

Ror, rol

- Rol = rotate left, ror = rotate right, lol = laughing out loud
- Rol and ror perform a bitwise rotation operation on the given source/destination (first) operand

- Example:
  - Rol al, 1 ; AL is shifted left by 1 and the original top bit of AL moves round into the low bit

Further Reading

- x86 Assembly Guide:
  http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

- NASM Appendix A: Intel x86 Instruction Reference:
  http://www.posix.nl/linuxassembly/nasmdochtml/nasmdoca.html