An Introduction to x86 ASM

Malware Analysis Seminar
Meeting 1
Cody Cutler, Anton Burtsev
Registers

- General purpose
  - EAX, EBX, ECX, EDX
  - ESI, EDI (index registers, but used as general in 32-bit protected mode)
- Stack
  - EBP, ESP
- Instruction pointer
  - EIP
- Flags
  - EFLAGS
- Segment
  - CS, DS, SS, ES, FS, GS
Syntax

- General form:
  - *mnemonic operand(s)*
    - movl %eax, %ebx

- Operands (0-3) may refer:
  - Registers
  - Memory
  - Immediate
Syntax (contd.)

• Register naming (AT&T is UNIX default)
  • **AT&T:** %eax
  • **Intel:** eax

• Source/Destination Ordering:
  • Load EBX with the value of EAX
  • **AT&T:** movl %eax, %ebx
  • **Intel:** mov ebx, eax
Constant value/immediate value format

- Load EAX with the address of the "C" variable boo
  - **AT&T:** movl $-_boo, %eax
  - **Intel:** mov eax, _boo
  - Note that "_" works for static (global) variables only

- Now let's load ebx with 0xd00d:
  - **AT&T:** movl $0xd00d, %ebx
  - **Intel:** mov ebx, d00dh
Operator size specification

- You don't want make GAS to guess this wrong
  - **AT&T**: movw %ax, %bx
  - **Intel**: mov bx, ax
Referencing memory

- 32bit protected mode addressing
  - **AT&T**: immed32(basepointer,indexpointer,indexscale)
  - **Intel**: [basepointer + indexpointer*indexscale + immed32]

- A global C variable
  - **AT&T**: _booga
  - **Intel**: [_booga]

- Addressing what a register points to:
  - **AT&T**: (%eax)
  - **Intel**: [eax]
Referencing memory (contd.)

- Addressing a variable offset by a value in a register:
  - **AT&T:** `_variable(%eax)`
  - **Intel:** `[eax + _variable]`

- Addressing a value in an array of integers (scaling by 4):
  - **AT&T:** `_array(,%eax,4)`
  - **Intel:** `[eax*4 + array]`

- Offsets with immediate value
  - **C code:** `*(p+1)` where p is a char *
  - **AT&T:** `1(%eax)` where eax has the value of p
  - **Intel:** `[eax + 1]`
Referencing memory (contd.)

• Addressing a particular char in an array of 8-character records
  • **EAX** holds the number of the record desired.
  • **EBX** has the wanted char's offset within the record.

• **AT&T:** _array(%ebx,%eax,8)
• **Intel:** [ebx + eax*8 + _array]
Integers:

- Two's compliment:
- Reverse bits, then add one (throw away carry)
  - Original value: 00111000 (+56)
  - Reverse bits: 11000111
  - Add 1: 11001000 (-56)

- Rules of arithmetic are preserved

\[
\begin{align*}
002C & \quad 44 \\
+ \ FFFF & \quad + \ (-1) \\
002B & \quad 43
\end{align*}
\]
Carry and overflow

• Overflow
  • Set if the true result of the operation is too big to fit into the destination for signed arithmetic.

• Carry
  • Set if there is a carry in the msb of an addition or a borrow in the msb of a subtraction.
  • Can be used to detect overflow for unsigned arithmetic.
Extended precision arithmetic

- **ADC**
  
  \[ \text{operand1} = \text{operand1} + \text{carry flag} + \text{operand2} \]

- **SBB**
  
  \[ \text{operand1} = \text{operand1} - \text{carry flag} - \text{operand2} \]

- **Sum of 64-bit integers in EDX:EAX and EBX:ECX**
  
  ```
  add eax, ecx ; add lower 32-bits
  adc edx, ebx ; add upper 32-bits and carry
  ```
Control structures

- Control structures decide what to do based on comparisons of data

- **CMP** instruction
  - subtract operands
  - set EFLAGS

- **EFLAGS** register
  - ZF – zero flag
  - CF – carry flag
  - SF – sign flag
Control structures (contd.)

- **Unsigned:** \( \text{cmp } \text{vleft}, \text{vright} \iff \text{vleft} - \text{vright} \)
  - \( \text{vleft} = \text{vright} \): ZF (1), CF (0)
  - \( \text{vleft} > \text{vright} \): ZF (0), CF (0) – no borrow
  - \( \text{vleft} < \text{vright} \): ZF (0), CF (1) – borrow

- **Signed:** \( \text{cmp } \text{vleft}, \text{vright} \iff \text{vleft} - \text{vright} \)
  - \( \text{vleft} = \text{vright} \): ZF (1), CF (0)
  - \( \text{vleft} > \text{vright} \): ZF (0), SF = CF
  - \( \text{vleft} < \text{vright} \): ZF (0), SF != CF
Branch instructions

• JMP
  • Short:
    - One byte instruction!
    - But jumps only 128 bytes up or down
  • Near:
    - Jump anywhere in a segment
    - 2-byte displacement: jump 32000 bytes
    - 4-byte displacement: jump anywhere in 32-bit mode
  • Far:
    - Jump across segments
Examples

if ( EAX == 0 )
    cmp eax, 0  ; set flags (ZF set if eax - 0 = 0)
    EBX = 1;
else
    mov ebx, 2  ; ELSE part of IF
    EBX = 2;
    jmp next    ; jump over THEN part of IF
thenblock:
    mov ebx, 1  ; THEN part of IF
next:
Comparison instructions

- JE branches if vleft = vright
- JNE branches if vleft != vright
- JL, JNGE branches if vleft < vright
- JLE, JNG branches if vleft <= vright
- JG, JNLE branches if vleft > vright
- JGE, JNL branches if vleft >= vright
Loops

• LOOP
  • Decrements ECX, if ECX != 0 branches to label

• LOOPE, LOOPZ
  • Decrements ECX (FLAGS register is not modified), if ECX != 0 and ZF = 1, branches

• LOOPNE, LOOPNZ
  • Decrements ECX (FLAGS unchanged), if ECX != 0 and ZF = 0, branches
Loop example

```c
sum = 0;
for ( i=10; i >0; i-- )
    sum += i;
```

```assembly
mov eax, 0   ; eax is sum
mov ecx, 10 ; ecx is i

loop_start:
    add eax, ecx
    loop loop_start
```

Stack

- **SS**
  - Specifies stack segment (usually same as data)

- **ESP**
  - Contains the address of the data that would be removed from the stack

- **PUSH/POP**
  - Insert/remove data on the stack
  - Subtract/add 4 to ESP
Call/return

• **CALL**
  - Makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack

• **RET**
  - Pops off an address and jumps to that address
Calling conventions

- Goal: reentrant programs
  - Conventions differ from compiler, optimizations, etc.
- Call/return are used for function invocations
- Parameters passed on the stack
  - Pushed onto the stack before the CALL instruction
### Stack bottom pointer

<table>
<thead>
<tr>
<th>ESP + 4</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>Return address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESP + 8</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP + 4</td>
<td>Return address</td>
</tr>
<tr>
<td>ESP</td>
<td>subprogram data</td>
</tr>
</tbody>
</table>

Initially parameter is:
- [ESP + 4]

Later as the function pushes things on the stack it changes, e.g.:
- [ESP + 8]
- Use dedicated register **EBP**
Prologue/epilogue

subprogram_label:
  push ebp          ; save original EBP value on stack
  mov ebp, esp     ; new EBP = ESP
; subprogram code
  pop ebp          ; restore original EBP value
  ret

• Example invocation

  push dword 1      ; pass 1 as parameter
  call fun
  add esp, 4       ; remove parameter from stack
Local variables

- Stored right after the saved EBP value in the stack
- Allocated by subtracting the number of bytes required from ESP

subprogram_label:
  push ebp  ; save original EBP value on stack
  mov ebp, esp  ; new EBP = ESP
  sub esp, LOCAL_BYTES ; = # bytes needed by locals
; subprogram code
  mov esp, ebp  ; deallocate locals
  pop ebp  ; restore original EBP value
  ret
Enter/leave

• ENTER
  • prologue code

• LEAVE
  • Epilogue

subprogram_label:
  enter LOCAL_BYTES, 0 ; = # bytes needed by locals
; subprogram code
  leave
  ret
Examples