Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014-2015
xeno@legbacore.com
All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/

Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64' class, available at http://OpenSecurityTraining.info/IntroX86-64.html"
SingleLocalVariable.c

Adding a single local variable

```c
//SingleLocalVariable.c:
int func(){
    int i = 0xbeef;
    return i;
}
int main(){
    return func();
}
```

```
// Assembly code for func:
0000000140001000 sub     rsp,18h
0000000140001004 mov     dword ptr [rsp],0BEEFh
0000000140001008 mov     eax,dword ptr [rsp]
000000014000100E add     rsp,18h
0000000140001012 ret

// Assembly code for main:
0000000140001020 sub     rsp,28h
0000000140001024 call     func (0140001000h)
0000000140001029 add     rsp,28h
000000014000102D ret
```
Don’t forget to change the next project to the StartUp project as you’re moving between labs!
Based on the asm, we can infer the stack looks like this at line 000000014000100B in func()

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000'0012FEB8</td>
<td>return address = 00000001'40001029</td>
</tr>
<tr>
<td>...</td>
<td>undef</td>
</tr>
<tr>
<td>00000000'0012FE78</td>
<td>undef</td>
</tr>
<tr>
<td>00000000'0012FE70</td>
<td>undef 0000BEEF</td>
</tr>
</tbody>
</table>
```

Because the asm only wrote a “dword ptr” (4 bytes) worth of memory at this location, so the top 4 bytes are undefined.
SingleLocalVariable.c takeaways

- Local variables lead to an allocation of space on the stack, within the function where the variable is scoped to
- In VS (when optimization is turned off), there is an over-allocation of space for local variables
  - 0x18 reserved for only 0x4 (int) worth of data

```c
//SingleLocalVariable.c:
int func(){
    int i = 0xbeef;
    return i;
}
int main(){
    return func();
}

// Disassembly:
func:
    sub    rsp,18h
    mov    dword ptr [rsp],0BEEFH
    mov    eax,dword ptr [rsp]
    add    rsp,18h
    ret

main:
    sub    rsp,28h
    call   func (0140001000h)
    add    rsp,28h
    ret
```
ArrayLocalVariable.c
Adding and accessing an array local variable

//ArrayLocalVariable.c:
short main(){
    int a;
    short b[6];
    long long c;
    a = 0x100d;
    c = 0xd00d;
    b[1] = (short)a;
    return b[4];
}
IMUL - Signed Multiply

- FYI, Visual Studio seems to have a predilection for imul over mul (unsigned multiply). You'll see it showing up in places you expect mul.
  - I haven't been able to get it to generate the latter for simple examples.
- Three forms. One, two, or three operands
  - imul r/mX             rdx:rax = rax * r/mX
  - imul reg, r/mX        reg = reg * r/mX
  - imul reg, r/mX, immediate  reg = r/mX * immediate
- Three operands? Possibly the only “basic” instruction (meaning non-added-on-instruction-set(MMX, AVX, AEX, VMX, etc)) of its kind? (see link in notes)

http://www.microsoft.com/msj/0698/hood0698.aspx - “There’s even a form of the IMUL instruction that takes three operands. To my knowledge, this is the only instruction in the Intel opcode set with this distinction.”

I found that quote while trying to find a way to make visual studio naturally emit a MUL instruction. Also, while things containing an r/mX can encode a mnemonic which looks like is has more operands, the information is still contained in a the normal combo of one or two bytes
## IMUL - examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Initial Operation</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imul ecx</code></td>
<td>edx eax</td>
<td>r/mX(ecx)</td>
<td>0x0 0x44000000 0x4</td>
</tr>
<tr>
<td><code>imul rax, rcx</code></td>
<td>rax r/mX(rcx)</td>
<td></td>
<td>0x20 0x4</td>
</tr>
<tr>
<td><code>imul rax, rcx, 0x6</code></td>
<td>rax r/mX(rcx)</td>
<td></td>
<td>0x18 0x4</td>
</tr>
<tr>
<td><code>imul rax, rcx</code></td>
<td>rax r/mX(rcx)</td>
<td></td>
<td>0x80 0x4</td>
</tr>
</tbody>
</table>

- **Initial Operation**: The initial operation is performed on the operands before the multiplication is applied.
- **Operation**: The operation is performed on the operands.
- **Result**: The result of the operation is shown for each instruction.
MOVZX - Move with zero extend
MOVZX - Move with sign extend

• Used to move small values (from smaller types) into larger registers (holding larger types)
• Support same r->r, r->m, m->r, i->m, i->r forms as normal MOV
• “Zero extend” means the CPU unconditionally fills the high order bits of the larger register with zeros
• “Sign extend” means the CPU fills the high order bits of the destination larger register with whatever the sign bit is set to on the small value
MOVZX/MOVSX - examples

- mov eax, 0xF00DFACE
- movzx rbx, eax
  now rbx = 0x00000000`F00DFACE
- movsx rbx, eax
  now rbx = 0xFFFFFFFF`F00DFACE, because the sign bit (most significant bit) of 0xF00DFACE is 1
Based on the asm, we can infer the stack looks like this at line 0000000140001049 of main:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000`0012FEB8</td>
<td>return address = 000000014000131d</td>
</tr>
<tr>
<td>...</td>
<td>undef</td>
</tr>
<tr>
<td>0000000`0012FEAC</td>
<td>undef</td>
</tr>
<tr>
<td>0000000`0012FEAA</td>
<td>short b[5] = undef</td>
</tr>
<tr>
<td>0000000`0012FEA8</td>
<td>short b[4] = maths!</td>
</tr>
<tr>
<td>0000000`0012FEA6</td>
<td>short b[3] = undef</td>
</tr>
<tr>
<td>0000000`0012FEA4</td>
<td>short b[2] = undef</td>
</tr>
<tr>
<td>0000000`0012FEA2</td>
<td>short b[1] = maths!</td>
</tr>
<tr>
<td>0000000`0012FEA0</td>
<td>short b[0] = undef</td>
</tr>
<tr>
<td>0000000`0012FE98</td>
<td>long long c = 0xd00d</td>
</tr>
<tr>
<td>0000000`0012FE90</td>
<td>int a = 0x100d</td>
</tr>
</tbody>
</table>

Note how there's wasted space by storing a 4 byte value ("int a") in an 8 byte space.

I'm cheating here and not using only 0x8-sized entries, for clarity of array indices' address.
ArrayLocalVariable.c takeaways

- Local variables need not be stored on the stack in the same order they are defined in the high level language.
- (In VS unoptimized code) Array access is typically done by multiplying the size of the array element (2 bytes for a short in this case), times the index that is desired to be access (indices 1 and 4 in this case).
- Moving a small value to a large register will result in a zero extend. Addition using signed values could result in a sign extend, if the arithmetic is done in a larger register.

```c
//ArrayLocalVariable.c:
short main() {
    int a;
    short b[6];
    long long c;
    a = 0x100d;
    c = 0xd00d;
    b[1] = (short)a;
    return b[4];
}
```
//StructLocalVariable.c:
typedef struct mystruct{
    int a;
    short b[6];
    long long c;
} mystruct_t;

short main(){
    mystruct_t foo;
    foo.a = 0x100d;
    foo.c = 0xd00d;
    foo.b[1] = foo.a;
    return foo.b[4];
}
Based on the asm, we can infer the stack looks like this:

```
Return address = 000000014000131d

00000000`0012FEB8 | undefined
...                |
00000000`0012FEA8 | undefined
00000000`0012FEA0 | foo.c = 0xd00d
00000000`0012FE9E | foo.b[5] = undefined
00000000`0012FE9C | foo.b[4] = maths!
00000000`0012FE9A | foo.b[3] = undefined
00000000`0012FE98 | foo.b[2] = undefined
00000000`0012FE96 | foo.b[1] = maths!
00000000`0012FE94 | foo.b[0] = undefined
00000000`0012FE90 | foo.a = 0x100d
```

Note how there's no wasted space this time since the "int a" 4 byte value is next to the 6 short 2 byte values, which all added up just happen to be 4 shy of 16 bytes total. Hence why c is accessed with "[rsp+10h]"
StructLocalVariable.c

- Fields in a struct must be stored in the same order they are defined in the high level language. And they will appear with the first field at the lowest address, and all subsequent fields higher.

```
//StructLocalVariable.c:
typedef struct mystruct{
    int a;
    short b[6];
    long long c;
} mystruct_t;

short main(){
    mystruct_t foo;
    foo.a = 0x100d;
    foo.c = 0xd00d;
    foo.b[1] = foo.a;
    return foo.b[4];
}
```
Instructions we now know (11)

- NOP
- PUSH/POP
- CALL/RET
- MOV
- ADD/SUB
- IMUL
- MOVZX/MOVXSX