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"
The Stack

- The stack is a conceptual area of main memory (RAM) which is designated by the OS when a program is started.
  - Different OS start it at different addresses by their own convention
- A stack is a Last-In-First-Out (LIFO/FILO) data structure where data is "pushed" on to the top of the stack and "popped" off the top.
- By convention the stack grows toward lower memory addresses. Adding something to the stack means the top of the stack is now at a lower memory address.
The Stack 2

- As already mentioned, RSP points to the top of the stack, the lowest address *which is being used*
  - While data will exist at addresses beyond the top of the stack, it is considered undefined
- The stack keeps track of which functions were called before the current one, it holds local variables and is often used to pass arguments to the next function to be called.
- A firm understanding of what is happening on the stack is *essential* to understanding a program’s operation.
The stack grows down towards low addresses!

This message brought to you by M-308 Gunner from MetalStorm!

http://2.bp.blogspot.com/_OcRaBrP1awY/SAelZhj61tI/AAAAAAAAATw/NudzjUumtRk/s400/luncheon_plates_stacks_DSCN4744.JPG
PUSH - Push Quadword onto the Stack

• For our purposes, it will always be a QWORD (8 bytes).
  – Can either be an “immediate” (Intel’s term for a numeric constant), or the value in a register

• The push instruction automatically decrements the stack pointer, RSP, by 8.

Will always be a QWORD because we will be running the processor in 64bit mode, and the instruction for a 16 bit push is the same as the one for a 32bit push is the same as the one for a 64 bit push. The processor just interprets the size based on the mode it is currently running in (or more accurately the segment, but that’s a story for Intermediate x86-64 ;))
### Registers Before

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>0x00000000`0</td>
</tr>
<tr>
<td>RSP</td>
<td>0x00000000`0 012FF88</td>
</tr>
</tbody>
</table>

### Registers After

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>0x00000000`0</td>
</tr>
<tr>
<td>RSP</td>
<td>0x00000000`0 012FF80</td>
</tr>
</tbody>
</table>

### Stack Before

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000`0 012FF90</td>
<td>0x00000000`0 00000000</td>
</tr>
<tr>
<td>0x00000000`0 012FF88</td>
<td>0x00000000`0 00000001</td>
</tr>
<tr>
<td>0x00000000`0 012FF80</td>
<td>0x00000000`0 00000002</td>
</tr>
<tr>
<td>0x00000000`0 012FF78</td>
<td>undefined</td>
</tr>
</tbody>
</table>

### Stack After

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000`0 012FF90</td>
<td>0x00000000`0 00000003</td>
</tr>
<tr>
<td>0x00000000`0 012FF88</td>
<td>0x00000000`0 00000001</td>
</tr>
<tr>
<td>0x00000000`0 012FF80</td>
<td>0x00000000`0 00000002</td>
</tr>
<tr>
<td>0x00000000`0 012FF78</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Note about the ` address convention

• When writing 64 bit numbers, it can be easy to lose track of whether you have the right number of digits

• WinDbg (which we don’t use in this class, but do in the Intermediate x86-64 class) allows you to write 64 bit numbers with a ` between the two 32 bit halves.

• I think this is helpful to see when a number is > 32 bit or not (because there will be some non-zero value on the left side of the `)

• So in this class I’ll occasionally write 64 bit numbers like 0x12345678`12345678.

• But keep in mind that the only tool which probably supports you entering them like that is WinDbg
POP- Pop a Value from the Stack

- Take a QWORD off the stack, put it in a register, and increment RSP by 8
- (Also has a “pop-into-memory” form which you can look up when you're more advanced and you know how to RTFM :))
### Registers Before

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>unknown</td>
</tr>
<tr>
<td>RSP</td>
<td>0x00000000`0012FF80</td>
</tr>
</tbody>
</table>

### Registers After

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>0x00000000`00000001</td>
</tr>
<tr>
<td>RSP</td>
<td>0x00000000`0012FF80</td>
</tr>
</tbody>
</table>

### Stack Before

- 0x00000000`0012FF90
- 0x00000000`0012FF88
- 0x00000000`0012FF80
- 0x00000000`0012FF78

### Stack After

- 0x00000000`00000001
- 0x00000000`00000002
- 0x00000000`00000003

- undef

- undef

- undef

The instruction `pop RAX` is executed, which pops the value from the stack and puts it into RAX.
Calling Conventions

• How code calls a subroutine is compiler-dependent and configurable. But there are a few conventions.

• More info at
  – http://www.programmersheaven.com/2/Calling-conventions

• Calling convention (as well as other assembly generation particulars) can be used as a first order heuristic of what compiler was used to generate the code

http://i.ytimg.com/vi/qonpElvRu8Q/hqdefault.jpg
http://4.bp.blogspot.com/_bngNvYNjl/TQBABjDToQI/AAAAAAAABJk/Z25l1apkbmY/s320/HomerSimpsonTowel.gif
Calling Conventions
Microsoft x86-64

• First 4 parameters (from left to right) are put into RCX, RDX, R8, R9 respectively
• Remaining parameters > 4 are “pushed” onto the stack (right-most param “pushed” first)
• NO use of frame pointers (if you know 32 bit calling conventions. If not, ignore this bullet)
• RAX or RDX:RAX returns the result for primitive data types
• Caller is responsible for cleaning up the stack

Colon notation means the full value is represented by the concatenation of the two values.
If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444
http://2.bp.blogspot.com/-RA88gtJkvMM/UUYB6qSrLiI/AAAAAAAAADw/NU7nOMw8kCE/s1600/Eisntein5.jpg
Calling Conventions
System V AMD64 ABI (GCC)

• First 6 parameters (from left to right) are put into RDI, RSI, RDX, RCX, R8, R9 respectively
• Remaining parameters “pushed” onto the stack (right-most param “pushed” first)
• Use of frame pointers in unoptimized code, but not in optimized code (if you know 32 bit calling conventions. If not, ignore this bullet for now)
• RAX or RDX:RAX returns the result for primitive data types
• Caller is responsible for cleaning up the stack

TODO: calling convention identification reinforcement goes here, or at end of deck, or after we’ve seen some asm?

Give students an example randomized C call, and the equivalent templated, randomized, x86 code and ask them to pick which calling convention it uses
General Stack Frame Operation

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).
General Stack Frame Operation 2

When main() decides to call a subroutine, main() becomes “the caller”. We will assume main() has some registers it would like to remain the same, so it will save them. We will also assume that the callee function takes some input arguments.

Local Variables
Caller-Save Registers
Arguments to Pass to Callee

main() frame
undef
undef
...

stack bottom
stack top
When `main()` actually issues the CALL instruction, the return address gets saved onto the stack, and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.

**Diagram:**

```
stack bottom

Local Variables
Caller-Save Registers
Callee arguments (> 4 MS, > 6 *NIX)
Caller's saved return address

main() frame
undef
undef
...

stack top
```
Next, we’ll assume the the callee foo() would like to use all the registers, and must therefore save the callee-save registers. Then it will allocate space for its local variables.

Local Variables
Caller-Save Registers
Callee arguments (> 4 MS, > 6 *NIX)
Caller’s saved return address
Callee-Save Registers
Local Variables
At this point, foo() decides it wants to call bar(). It is still the callee-of-main(), but it will now be the caller-of-bar. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well.
General Stack Frame Layout

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.) But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omitting frame pointers, changing calling convention to pass arguments on stack instead of in registers, etc.)

<table>
<thead>
<tr>
<th>CALLEE-SAVE REGISTERS</th>
<th>CALLER'S SAVED RETURN ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL VARIABLES</td>
<td></td>
</tr>
<tr>
<td>CALLEE-SAVE REGISTERS</td>
<td></td>
</tr>
<tr>
<td>CALLEE ARGUMENTS (&gt; 4 MS, &gt; 6 *NIX)</td>
<td></td>
</tr>
</tbody>
</table>

- stack bottom
  - main() frame
  - foo()'s frame
  - undefined
  - ...
- stack top
Instructions we now know (3)

- NOP
- PUSH/POP
Superfluous Curiosities
“The stack will always be maintained 16-byte aligned, except within the prolog (for example, after the return address is pushed), and except where indicated in Function Types for a certain class of frame functions.”
int main()
{
    register int a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7, h = 8, i = 9, j = 10, k = 11, l = 12, m = 13, n = 14, o = 15, p = 16;
    printf("%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p);
    a++;  
    return 0;
}
void once(int a)
{
    printf("%x\n", a);
}

void twice(int a)
{
    once(a);
    printf("%x\n", a);
}

int main()
{
    twice(0xF00D);
    return 0;
}