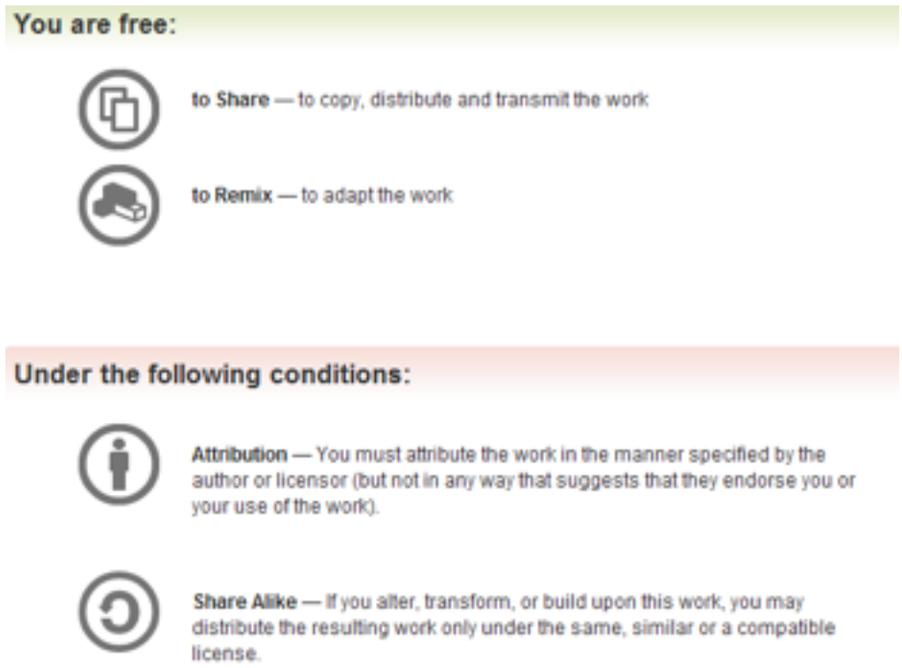
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

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"

Attribution condition: You must indicate that derivative work

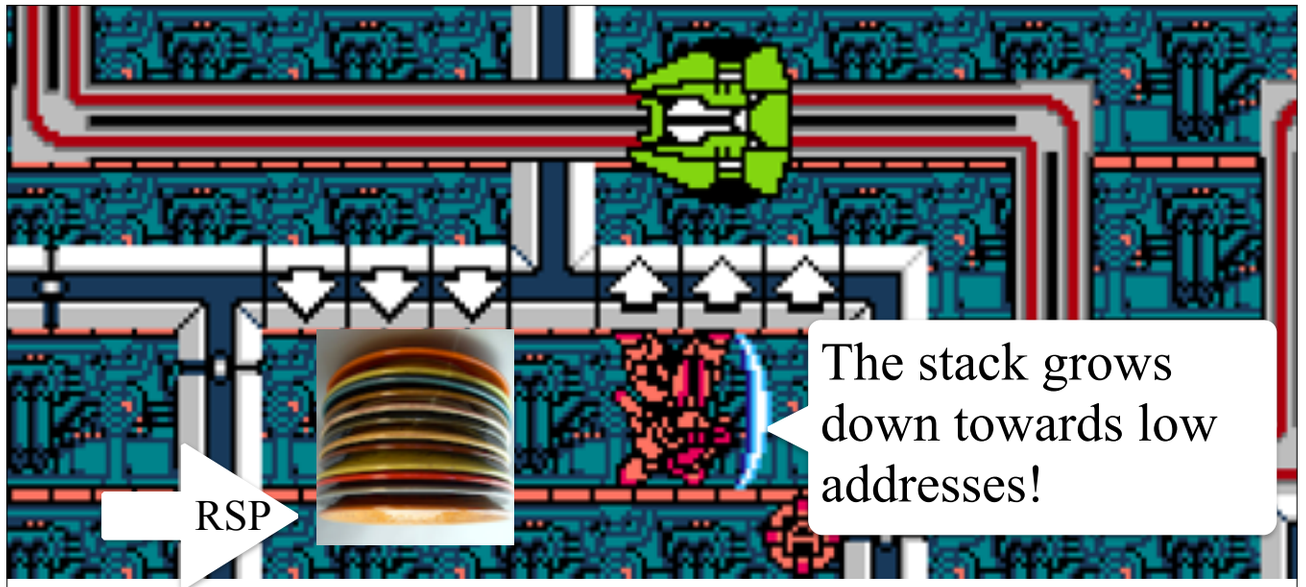
"Is derived from Xeno Kovah's ‘Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html"

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 ***<u>essential</u>*** to understanding a program's operation.



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

http://satoshimatrix.files.wordpress.com/2011/08/metal-storm-u-0009.png

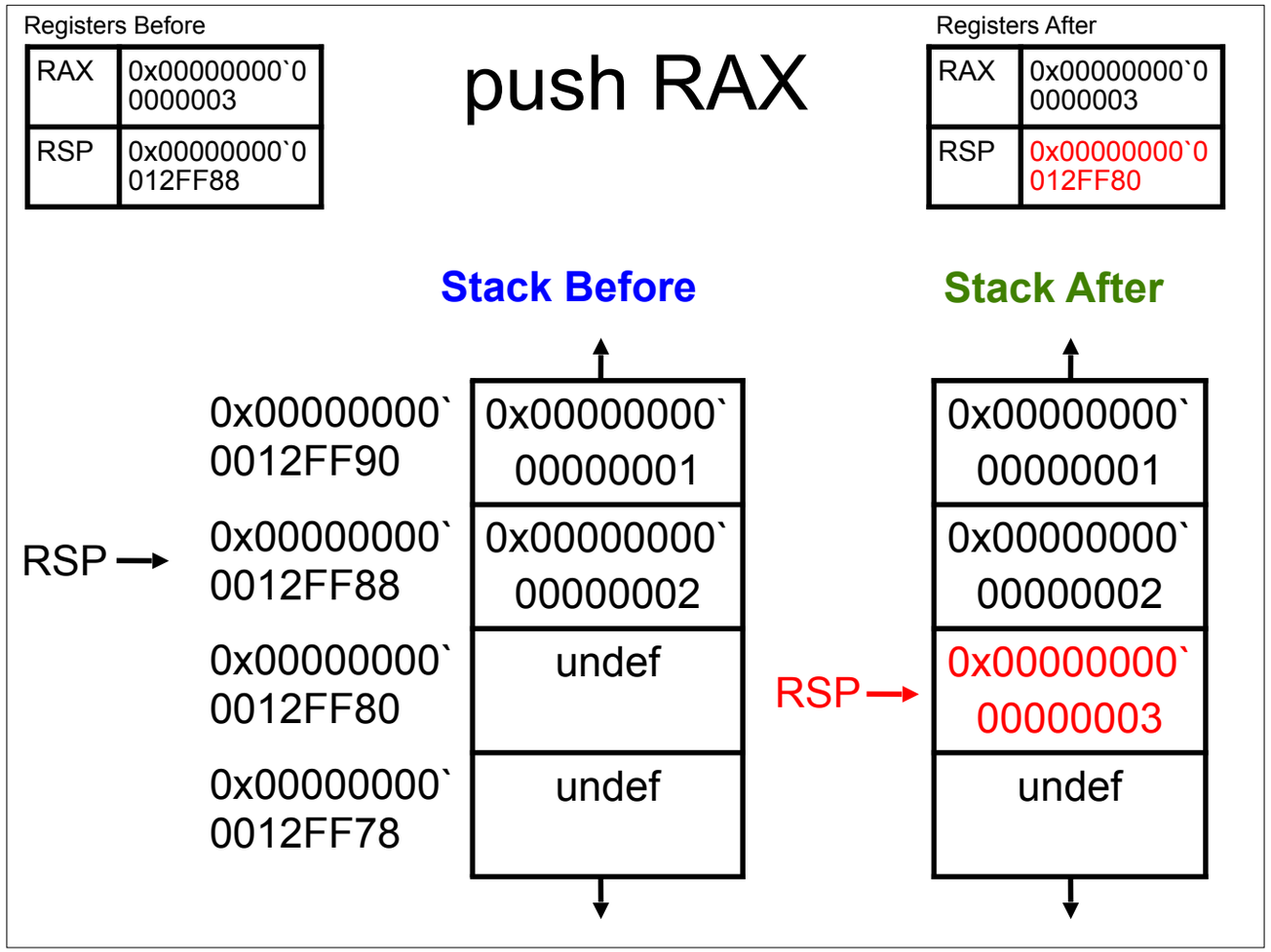
http://2.bp.blogspot.com/_OcRaBrP1awY/SAelZhj61tI/AAAAAAAAATw/NudzjUumtRk/s400/luncheon_plates_stacks_DSCN4744.JPG

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.

Book page 107

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 ;))

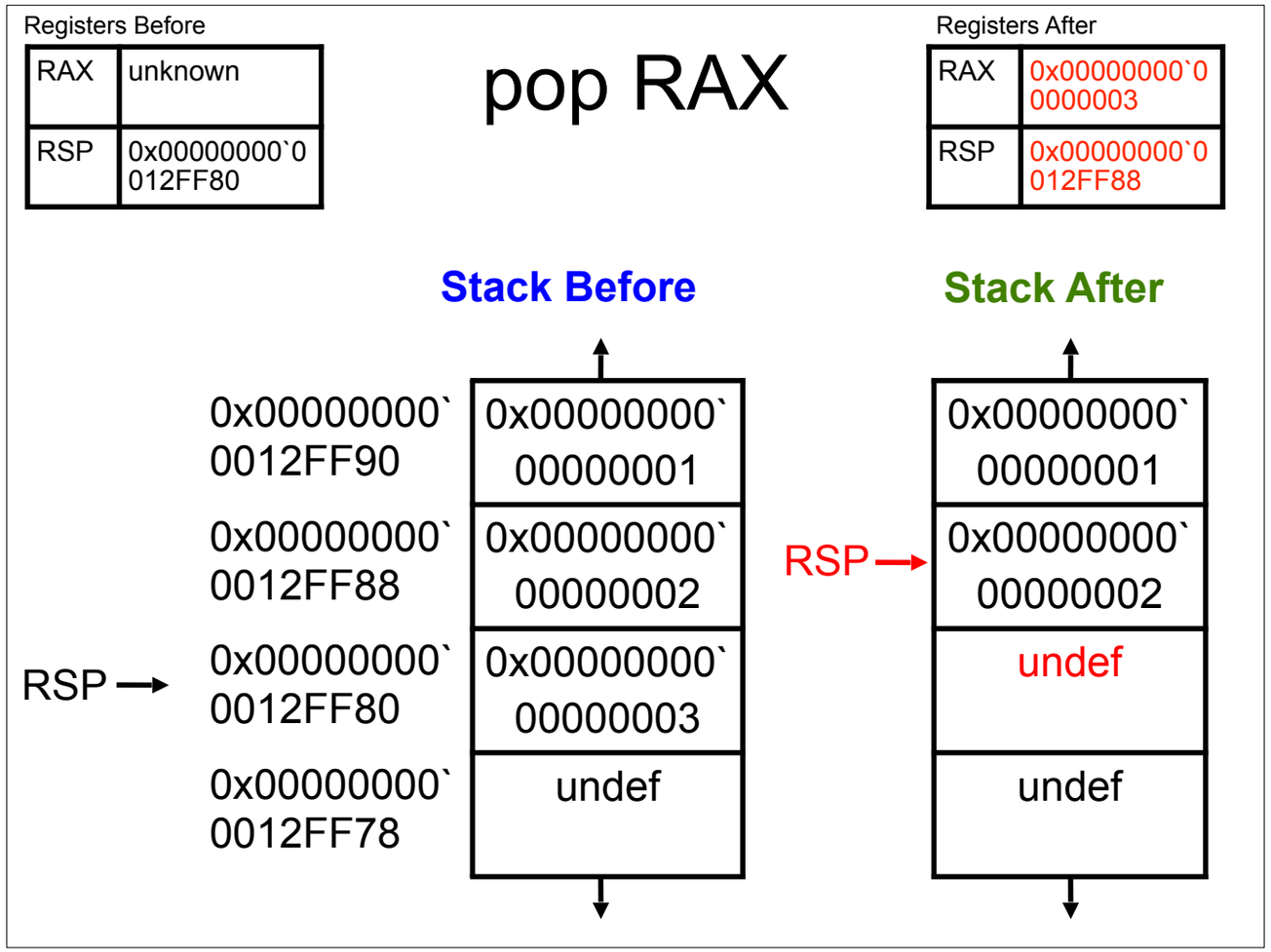

Registers Before
RAX
0x00000000`00000003
RSP
0x00000000`0012FF88
push RAX
Registers After
RAX
0x00000000`00000003
RSP
0x00000000`0012FF80
Stack Before
Stack After
RSP
0x00000000`0012FF90
0x00000000`0012FF88
0x00000000`0012FF80
0x00000000`0012FF78
0x00000000`00000001
0x00000000`00000002
undef
undef
RSP
0x00000000`00000001
0x00000000`00000002
0x00000000`00000003
undef

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 :))

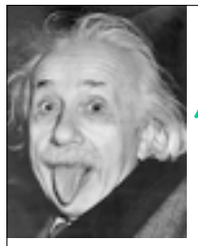

Registers Before
RAX
unknown
RSP
0x00000000`0012FF80
pop RAX
Registers After
RAX
0x00000000`00000003
RSP
0x00000000`0012FF88
Stack Before
Stack After
0x00000000`0012FF90
0x00000000`00000001
0x00000000`0012FF88
0x00000000`00000002
RSP
0x00000000`0012FF80
0x00000000`00000003
undef
0x00000000`0012FF78
undef
RSP
0x00000000`00000001
0x00000000`00000002
undef
undef

Calling Conventions

- How code calls a subroutine is compiler-dependent and configurable. But there are a few conventions.
- More info at
  - http://en.wikipedia.org/wiki/X86_calling_conventions
  - 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/_bngNvVpYNjI/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

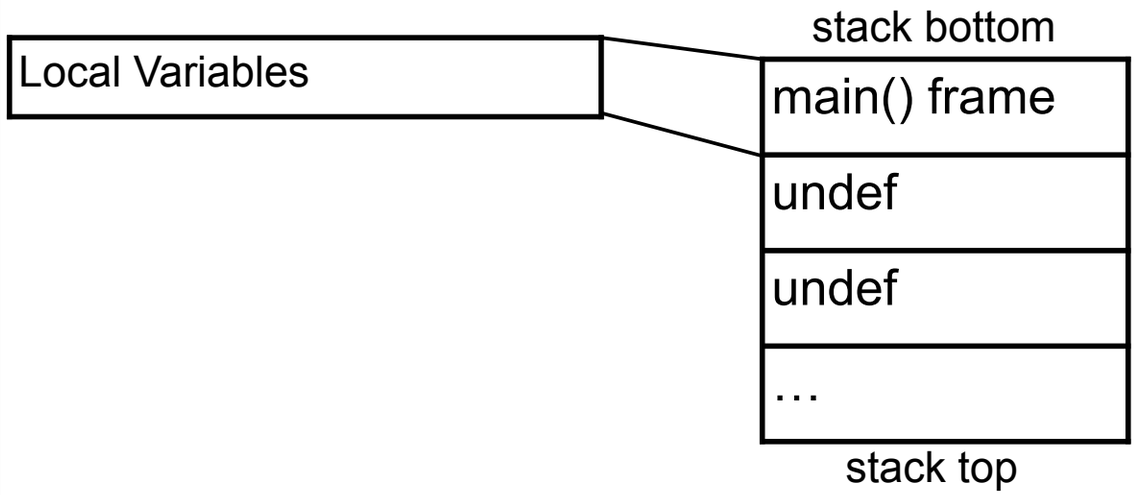
http://24.media.tumblr.com/tumblr_m9p96hnQVc1qgy3iwo1_500.gif

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

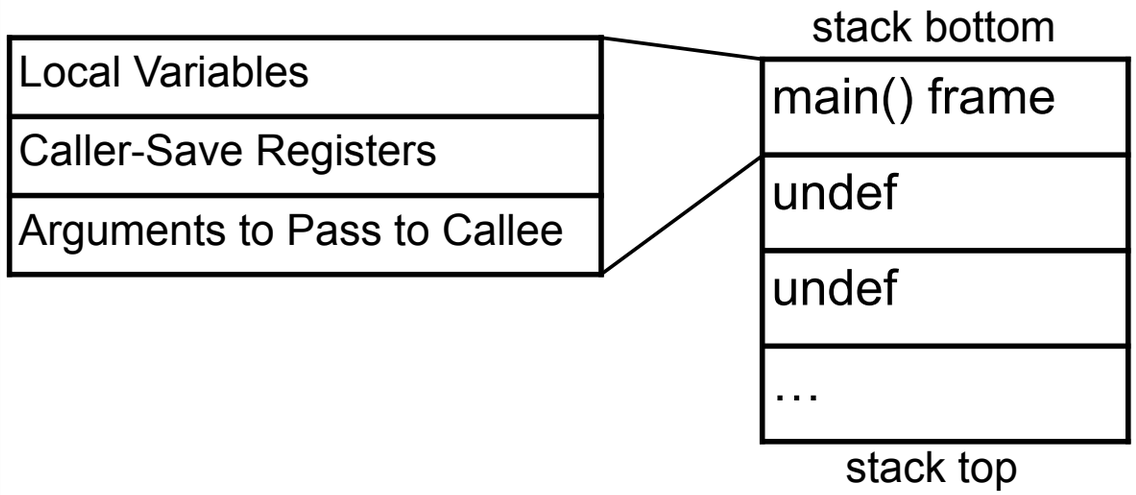
General Stack Frame Operation

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).



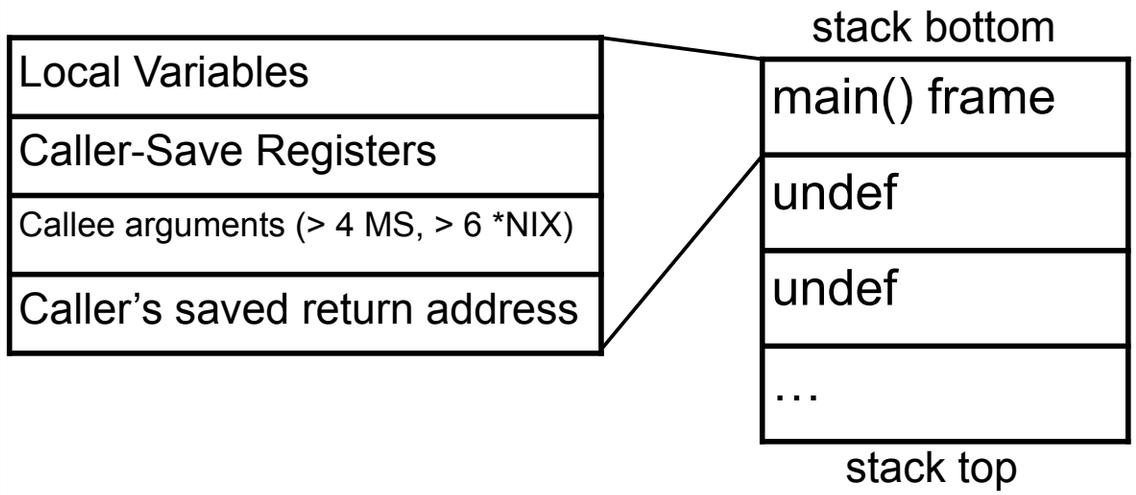
General Stack Frame Operation 2

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.



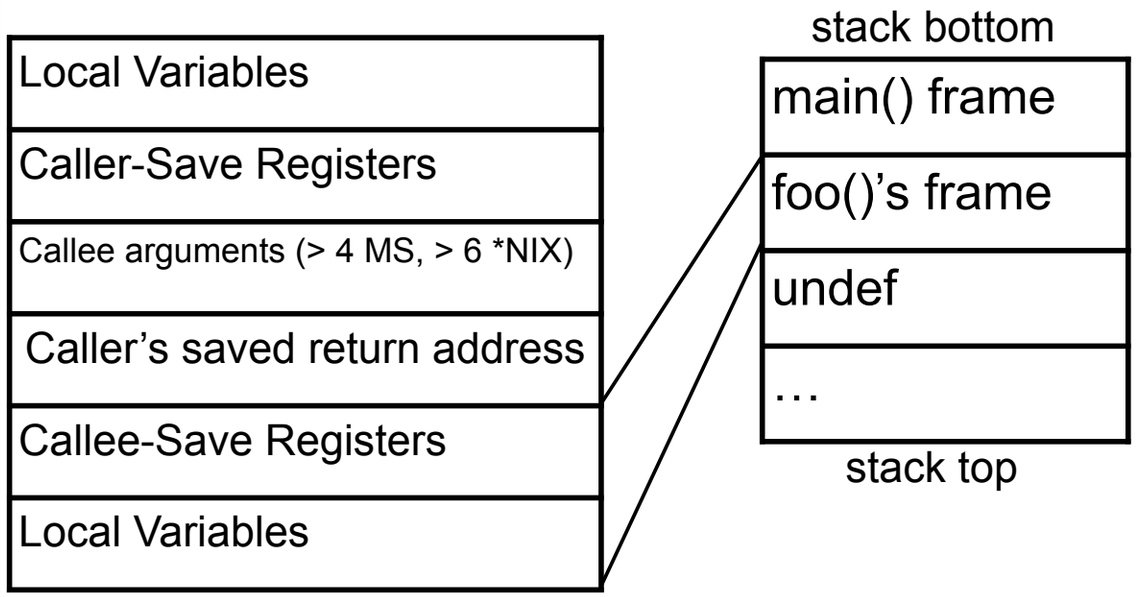
General Stack Frame Operation 3

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.



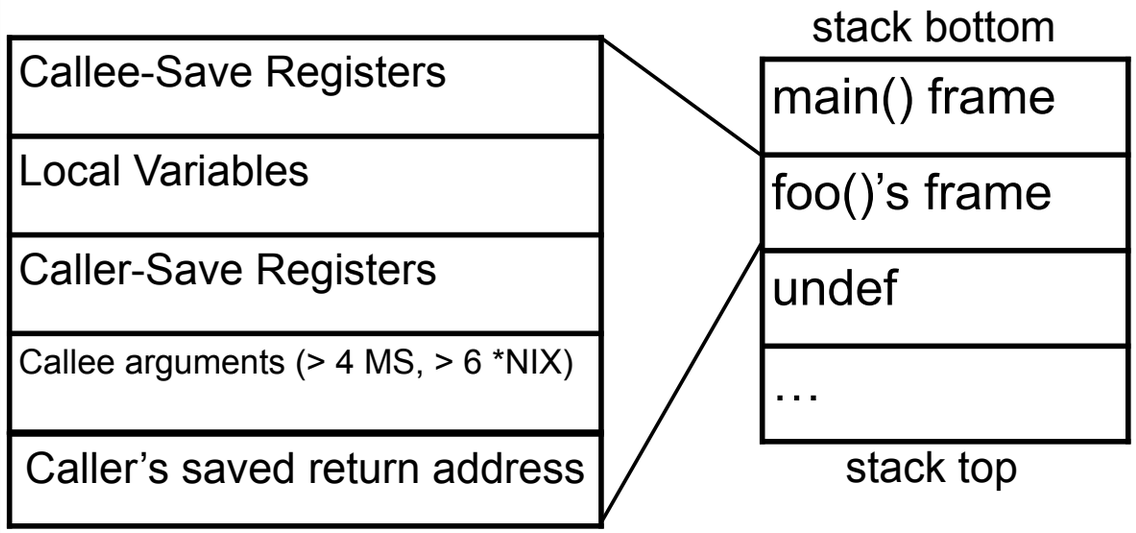
General Stack Frame Operation 4

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.



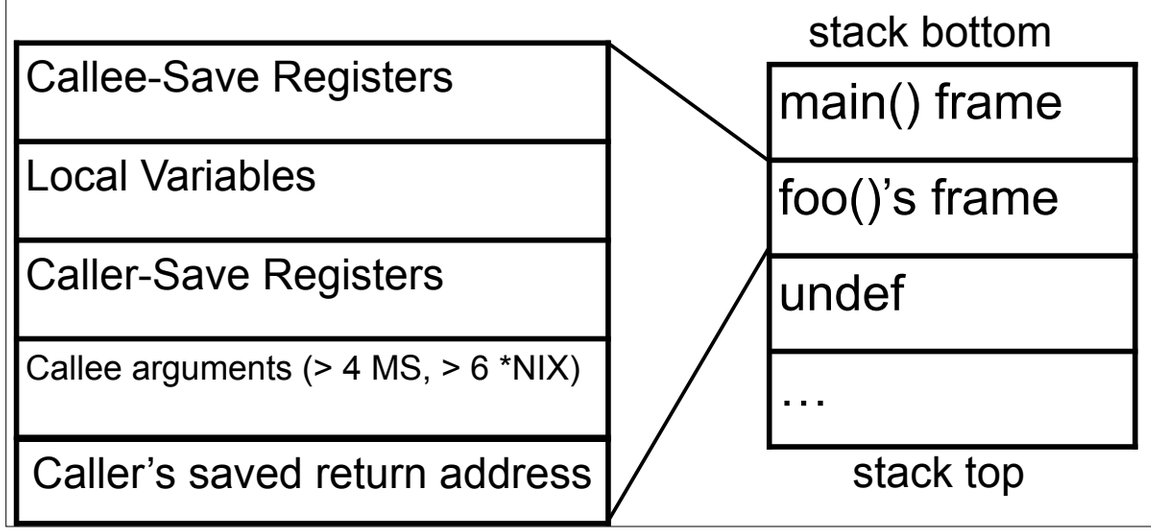
General Stack Frame Operation 6

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.)



Instructions we now know (3)

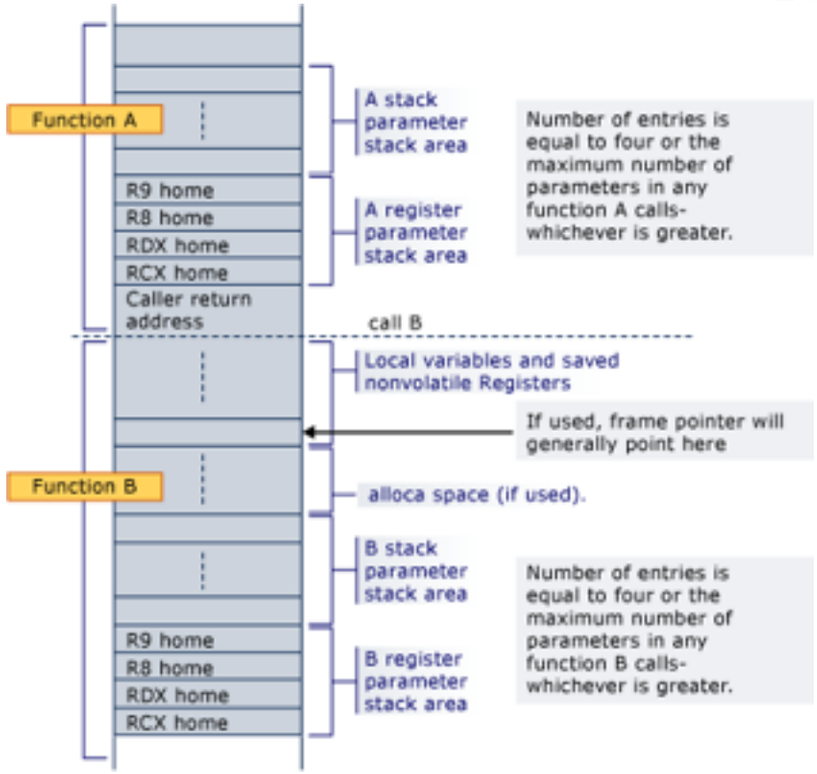
- NOP
- PUSH/POP

Superfluous Curiosities

http://msdn.microsoft.com/en-us/library/ew5tede7.aspx

High mem

Bottom of stack



Low mem

Top of stack

"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."

Caller-save registers just mixed in with the local variables?

```
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;
}
```

```
000000000040052d <main>:
 40052d:	55                   	push   rbp
 40052e:	48 89 e5             	mov    rbp,rsp
 400531:	41 57                	push   r15
 400533:	41 56                	push   r14
 400535:	41 55                	push   r13
 400537:	41 54                	push   r12
 400539:	53                   	push   rbx
 40053a:	48 83 ec 68          	sub    rsp,0x68
 40053e:	41 bc 01 00 00 00    	mov    r12d,0x1
 400544:	41 bd 02 00 00 00    	mov    r13d,0x2
 40054a:	41 be 03 00 00 00    	mov    r14d,0x3
 400550:	c7 45 cc 04 00 00 00 	mov    DWORD PTR [rbp-0x34],0x4
 400557:	c7 45 c8 05 00 00 00 	mov    DWORD PTR [rbp-0x38],0x5
 40055e:	41 bf 06 00 00 00    	mov    r15d,0x6
 400564:	41 b9 07 00 00 00    	mov    r9d,0x7
 40056a:	41 b8 08 00 00 00    	mov    r8d,0x8
 400570:	41 bb 09 00 00 00    	mov    r11d,0x9
 400576:	41 ba 0a 00 00 00    	mov    r10d,0xa
 40057c:	bf 0b 00 00 00       	mov    edi,0xb
 400581:	be 0c 00 00 00       	mov    esi,0xc
 400586:	b9 0d 00 00 00       	mov    ecx,0xd
 40058b:	ba 0e 00 00 00       	mov    edx,0xe
 400590:	b8 0f 00 00 00       	mov    eax,0xf
 400595:	bb 10 00 00 00       	mov    ebx,0x10
 40059a:	89 5c 24 50          	mov    DWORD PTR [rsp+0x50],ebx
 40059e:	89 44 24 48          	mov    DWORD PTR [rsp+0x48],eax
 4005a2:	89 d0                	mov    eax,edx
 4005a4:	89 44 24 40          	mov    DWORD PTR [rsp+0x40],eax
 4005a8:	89 c8                	mov    eax,ecx
 4005aa:	89 44 24 38          	mov    DWORD PTR [rsp+0x38],eax
 4005ae:	89 f0                	mov    eax,esi
 4005b0:	89 44 24 30          	mov    DWORD PTR [rsp+0x30],eax
 4005b4:	89 f8                	mov    eax,edi
 4005b6:	89 44 24 28          	mov    DWORD PTR [rsp+0x28],eax
 4005ba:	44 89 d0             	mov    eax,r10d
 4005bd:	89 44 24 20          	mov    DWORD PTR [rsp+0x20],eax
 4005c1:	44 89 d8             	mov    eax,r11d
 4005c4:	89 44 24 18          	mov    DWORD PTR [rsp+0x18],eax
 4005c8:	44 89 c0             	mov    eax,r8d
 4005cb:	89 44 24 10          	mov    DWORD PTR [rsp+0x10],eax
 4005cf:	44 89 c8             	mov    eax,r9d
 4005d2:	89 44 24 08          	mov    DWORD PTR [rsp+0x8],eax
 4005d6:	44 89 3c 24          	mov    DWORD PTR [rsp],r15d
 4005da:	44 8b 4d c8          	mov    r9d,DWORD PTR [rbp-0x38]
 4005de:	44 8b 45 cc          	mov    r8d,DWORD PTR [rbp-0x34]
 4005e2:	44 89 f1             	mov    ecx,r14d
 4005e5:	44 89 ea             	mov    edx,r13d
 4005e8:	44 89 e6             	mov    esi,r12d
 4005eb:	bf a8 06 40 00       	mov    edi,0x4006a8
 4005f0:	b8 00 00 00 00       	mov    eax,0x0
 4005f5:	e8 16 fe ff ff       	call   400410 <printf@plt>
 4005fa:	41 83 c4 01          	add    r12d,0x1
 4005fe:	b8 00 00 00 00       	mov    eax,0x0
 400603:	48 83 c4 68          	add    rsp,0x68
 400607:	5b                   	pop    rbx
 400608:	41 5c                	pop    r12
 40060a:	41 5d                	pop    r13
 40060c:	41 5e                	pop    r14
 40060e:	41 5f                	pop    r15
 400610:	5d                   	pop    rbp
 400611:	c3                   	ret
```

Caller-save registers pre-local variables?

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

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

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

```
once:
000000013F771000  mov         dword ptr [rsp+8],ecx
000000013F771004  sub         rsp,28h
000000013F771008  mov         edx,dword ptr [rsp+30h]
000000013F77100C  lea         rcx,[3F7CE000h]
000000013F771013  call        000000013F771220
000000013F771018  add         rsp,28h
000000013F77101C  ret
twice:
000000013F771030  mov         dword ptr [rsp+8],ecx //<-caller-save
000000013F771034  sub         rsp,28h
000000013F771038  mov         ecx,dword ptr [rsp+30h]
000000013F77103C  call        000000013F771000
000000013F771041  mov         edx,dword ptr [rsp+30h]
000000013F771045  lea         rcx,[3F7CE004h]
000000013F77104C  call        000000013F771220
000000013F771051  add         rsp,28h
000000013F771055  ret
main:
000000013F771060  sub         rsp,28h
000000013F771064  mov         ecx,0F00Dh
000000013F771069  call        000000013F771030
000000013F77106E  xor         eax,eax
000000013F771070  add         rsp,28h
000000013F771074  ret
```