PROGRAM EXECUTION ENVIRONMENT

Girish Bharambe

NVIDIA

WHY LEARN PROGRAM EXECUTION ENVIRONMENT

- Learn details of what is going on "under the hood" of a computer system
  - Being able to understand various abstractions and interfaces that exist between various system software components and Hardware
    - Not just "what" but also "why" and "how"
  - Being able to understand how does computer execute your programs
- Why should you learn these topics?
  - Makes you a better programmer
  - Makes you more effective at debugging
  - Makes you aware of how system is built and that enables you to build new systems as needed
    - Note although specific implementations of systems change, underlying concepts mostly hold or at least form initial intuitions for new ideas

OUTLINE

Life of a program

Abstraction of functions

Mechanisms to implement Functions

Mechanisms for Control Transfer

Mechanism for Local Variables

Mechanism for Parameters

Mechanism for Register Usage

Register Saving Conventions

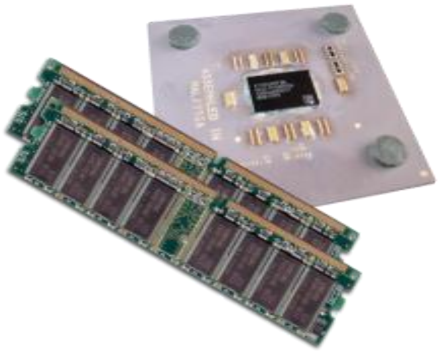
LIFE OF A PROGRAM

LIFE OF A PROGRAM (HIGH LEVEL VIEW)

```
int foo(int p) {
    int var = 5;
    return var + p;
}
```

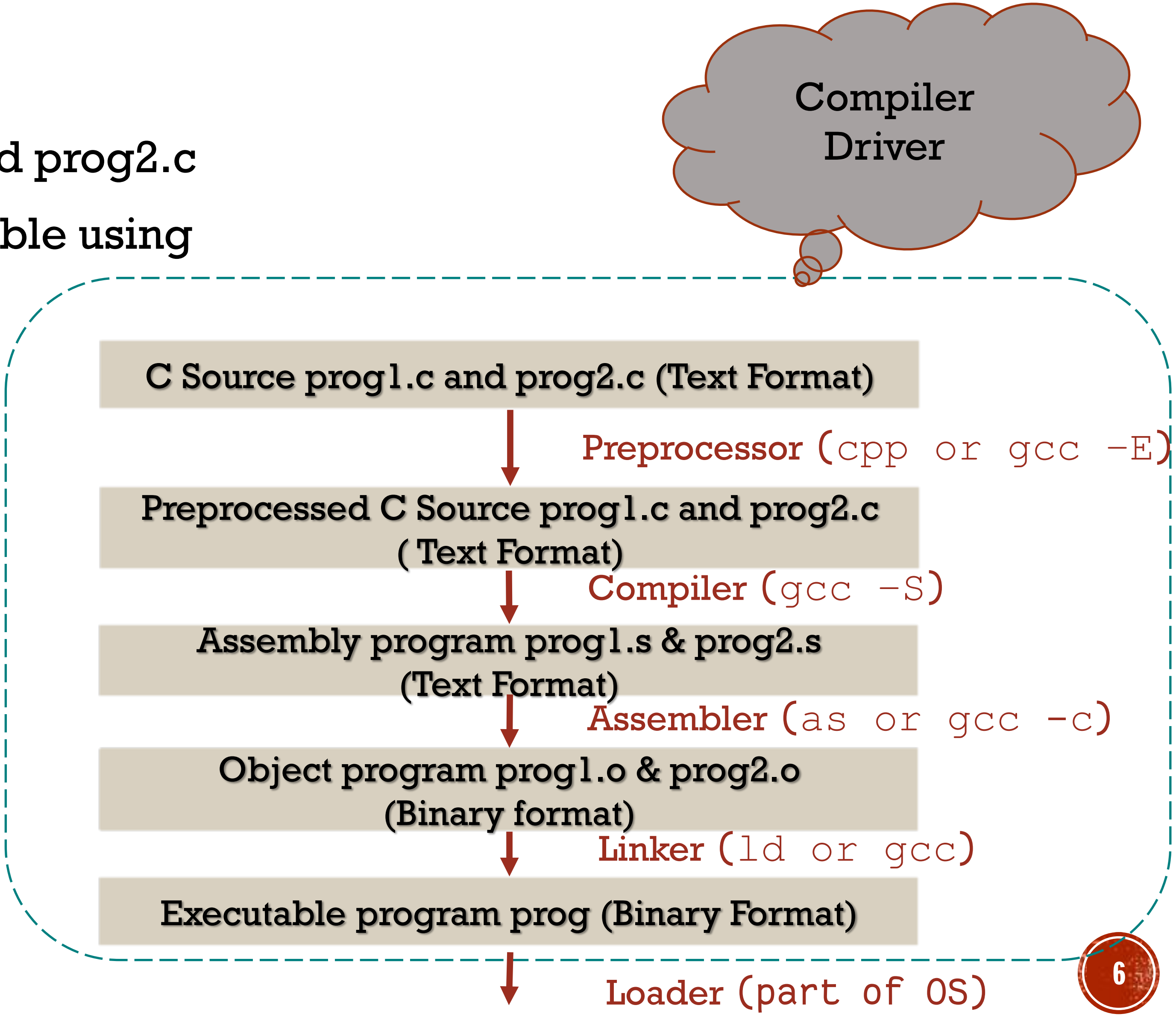
```
foo:
    pushl %ebp
    movl %esp, %ebp
    ...
    addl %edx, %eax
    ret
```

```
1010101
1000100111100101
...
000111010000
11000011
```

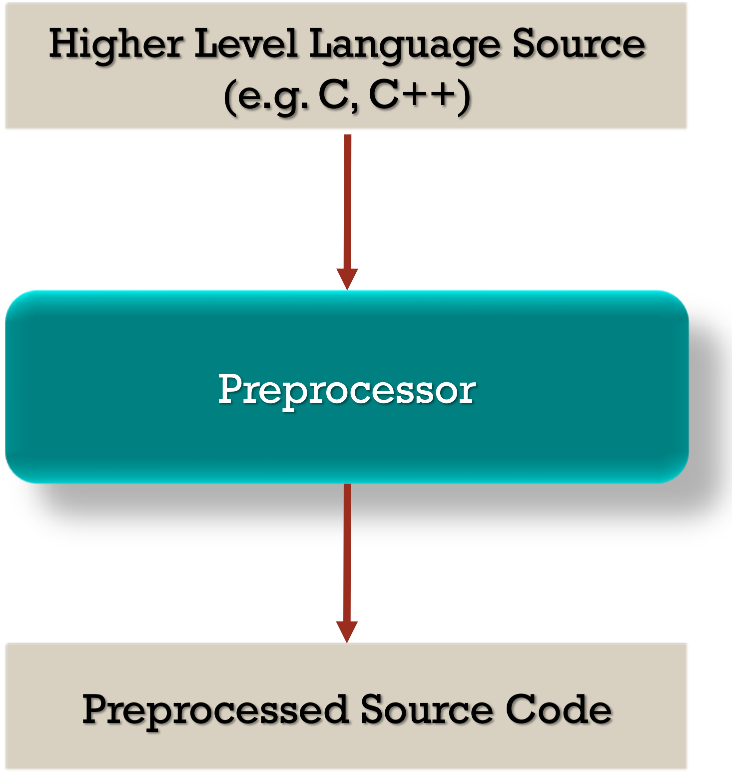


BUILDING AN EXECUTABLE FROM C SOURCE

- Consider source code in prog1.c and prog2.c
- Compile the source file into executable using
  - `gcc prog1.c prog2.c -o prog`
- Run the program using
  - `./prog`

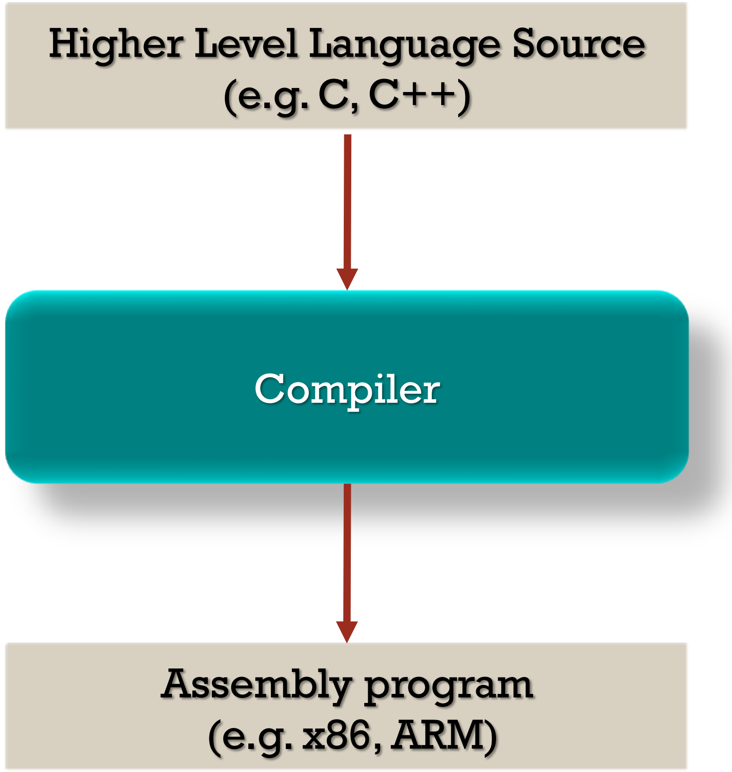


PREPROCESSOR



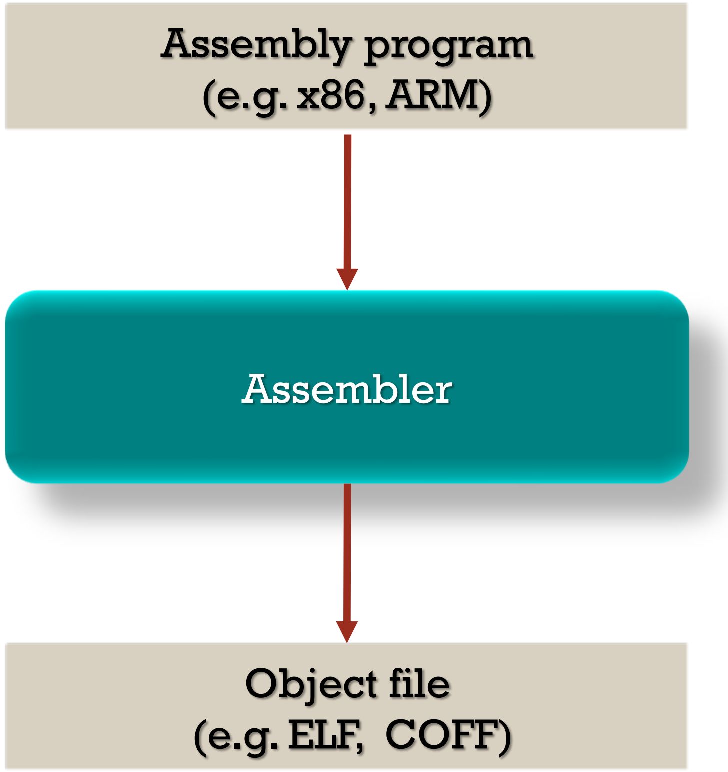
- Processes pre-processor directives like `#define`, `#include`, macro expansions.
- Typically is a separate program invoked before actual compilation.

COMPILER



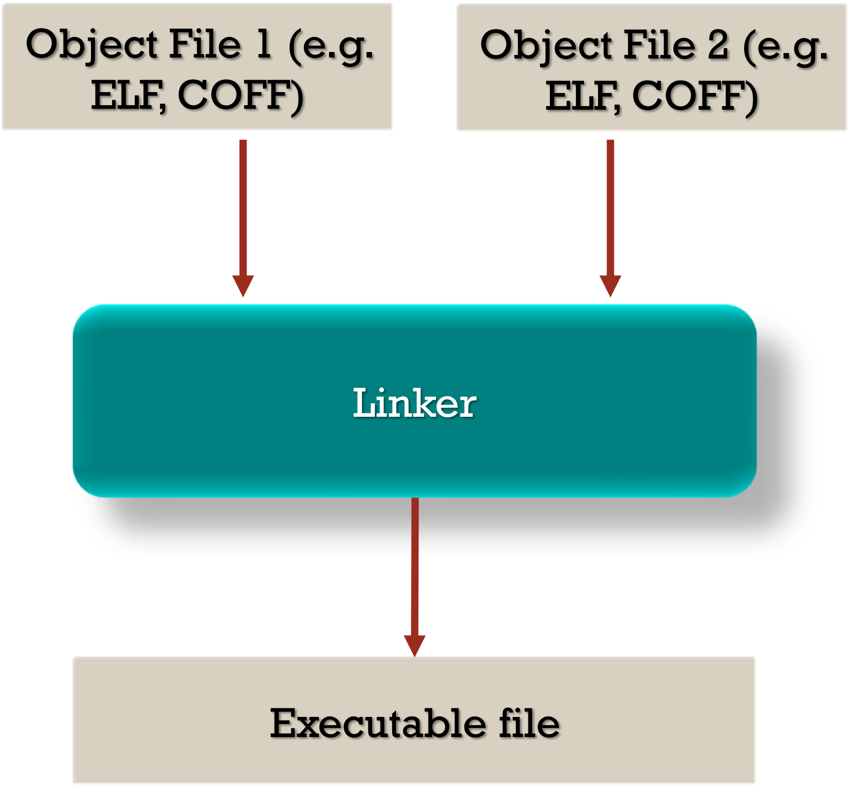
- Translates source written in higher level languages into lower level assembly
- Not just translator but also has optimizations and code generation

ASSEMBLER



- Translate textual assembly program into binary object file
- Assembly languages typically support various directives to write different parts of assembly program e.g. `.text`, `.data` which needs to be handled by assemblers
- Encodes textual machine instructions to corresponding instruction encoding
- Creates object files containing machine instructions and other required information

LINKER

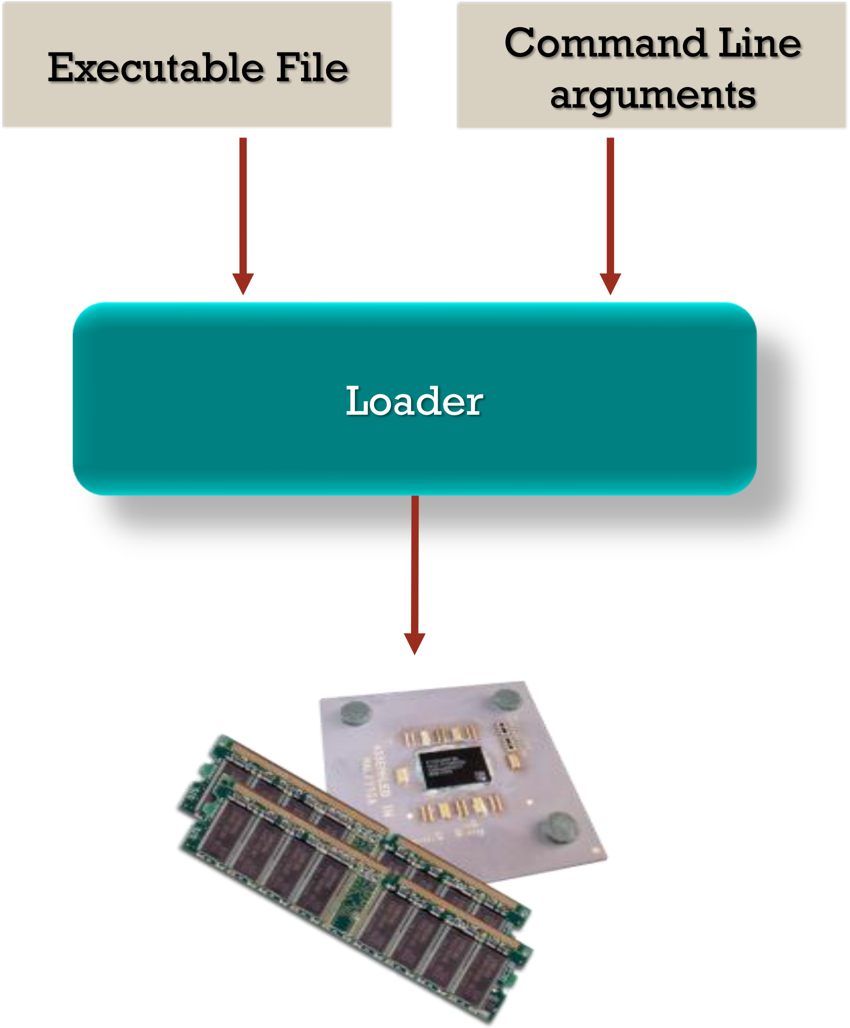


- Combines several object files into a single executable
- Primarily need to resolve extern references from object files to their corresponding definitions from other object file.
- Creates executable binary program that can be loaded by loader.

WHAT IS NEED FOR A LINKER?

- Recall : Linker combines several object files into a single executable
- Why write source code in multiple files?
  - Very uncommon to have large codes to be fully self-contained without any other dependencies.
  - For modularity, better to write large program as collection of smaller source files
- Code Reuse
  - Allows reuse of already written codes
  - Can create library of common utilities
    - Facilitates creation and distribution of pre-compiled libraries
    - IP protection : Allows library author to distribute code without shipping source code
- Separate Compilation
  - When one of the source file changes, no need to recompile all parts of program, can only compile modified source file and relink
- BUT WAIT, are there NO advantages of having source code in single file?
  - Having source in single file provides more visibility to compiler which enables more optimizations
  - Linker will have visibility and can do some optimizations at link time but is limited compared to compiler optimizations

LOADER

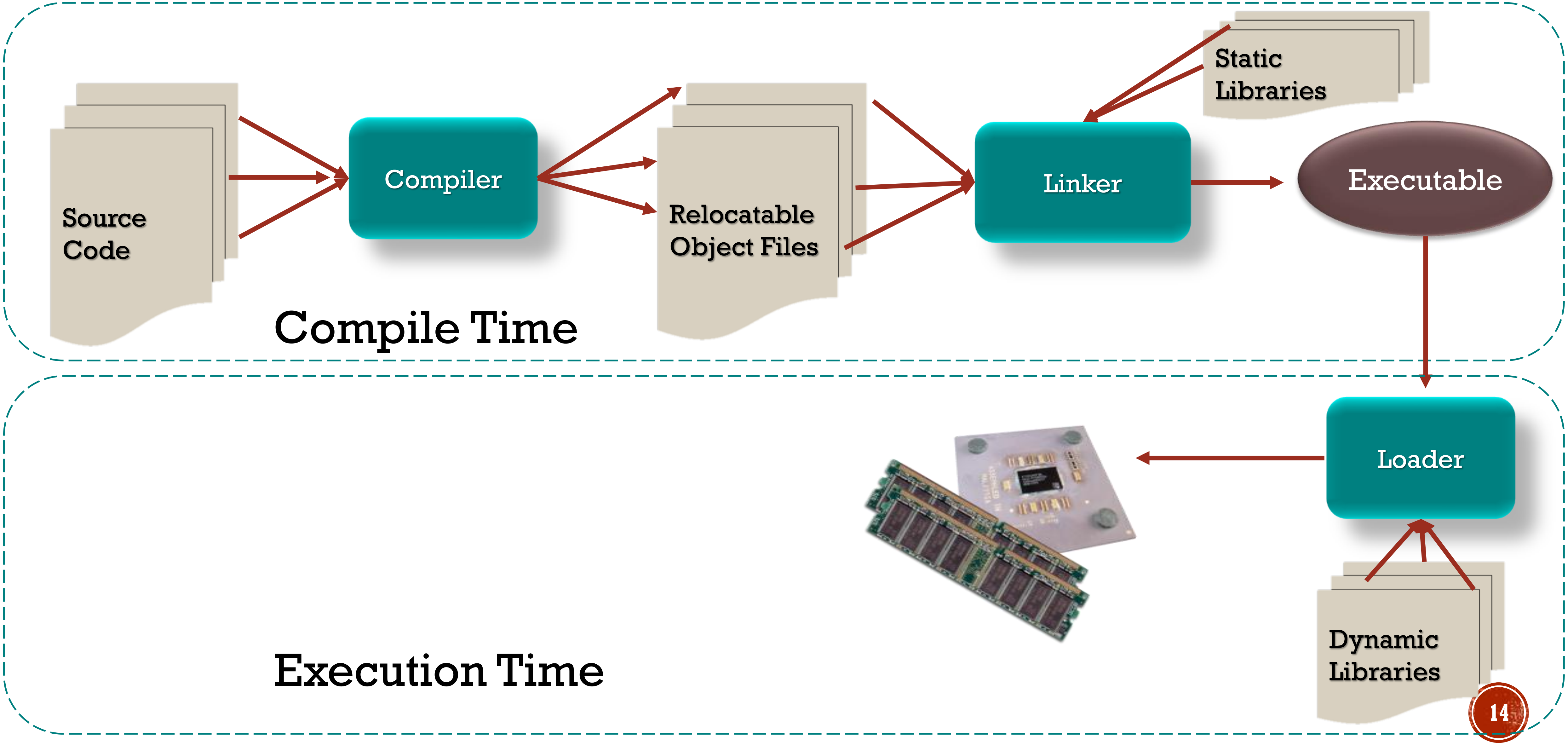


- Part of OS / Kernel
- Reads executable binary from disk and loads it into memory and then starts running program by jumping to first instruction of program

WHAT IS NEED FOR A LOADER?

- In a system that supports running only one program at a time, may be no need for a loader
  - Program can be assembled and linked for a fixed memory address
  - Program has access to entire memory
- With operating system and system running multiple programs, physical memory is shared between various programs including operating systems
- Actual address at which program will be loaded isn't known until execution time

LIFE OF A PROGRAM



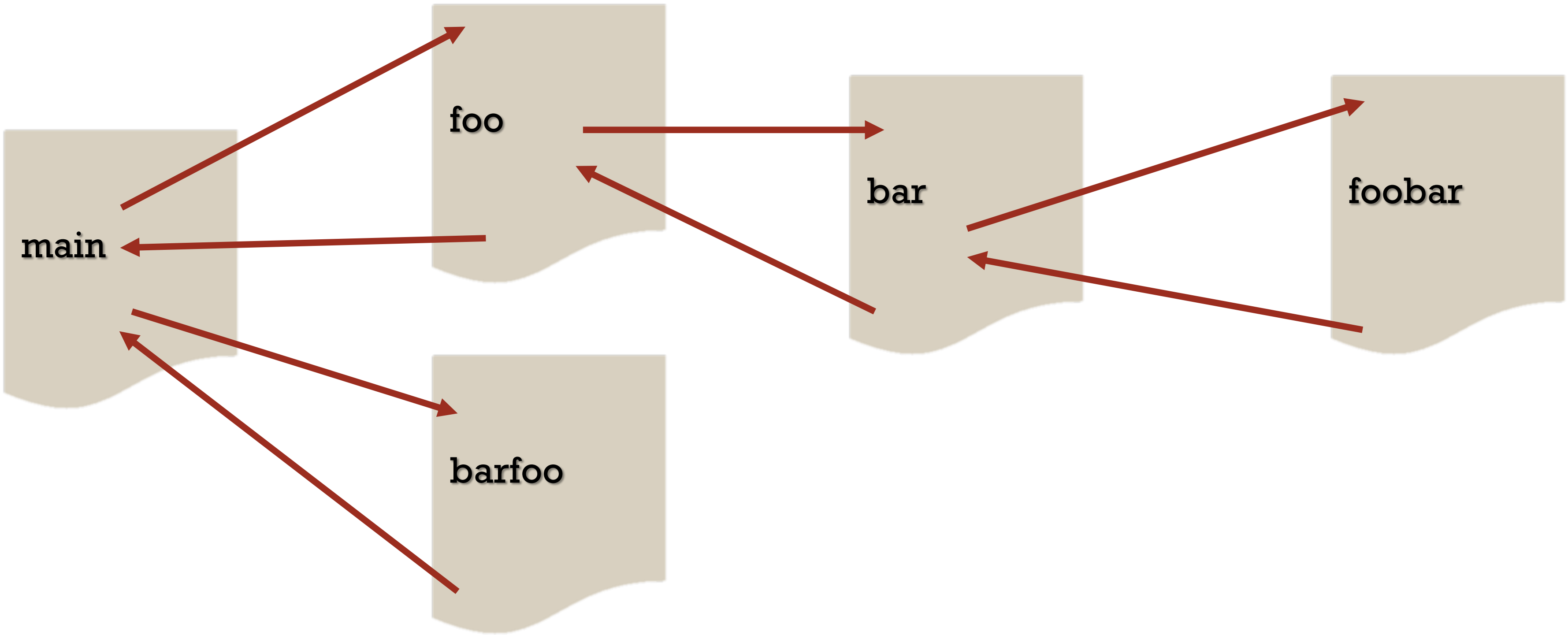
DEMYSTIFYING ABSTRACTION OF FUNCTIONS

FUNCTIONS

- Programming Languages allow encapsulating a block of code that *typically* perform single, related task.
- Functions allow better modularity and provide high degree of code reuse
- A function consists of a
  - Name that is symbol that represents address of function
  - Parameters (aka input arguments) that are used to specify data that is necessary for function to work
  - Return value (aka output argument) that is used to return data back to the caller function
  - Function body that contains collection of instructions that perform operations function is intended to do.
  - Local Variables that are data storage that function uses during processing and is thrown away when function returns.

CHARACTERISTICS OF FUNCTION

- Each function *typically* has a single-entry point
- The calling function (aka *caller*) *typically* is suspended during execution of the called function (aka *callee*)
- Control returns to the caller function when callee function returns i.e. function returns only after all functions it calls have returned



MECHANICS NEEDED TO IMPLEMENT FUNCTION CALL

- Mechanics to transfer control from caller function to callee and return from callee function to caller
- Mechanics to pass function parameters and return values
- Mechanics to allocate memory for local variables and deallocate after function returns
- Mechanics for register usage across caller and callee functions

```
int main(void) {
    ...
    q = foo(p);
    printf("%d", q);
}
```

```
int foo(int m) {
    int var = 5;
    int arr[10];
    ...
    return arr[m];
}
```

MECHANICS FOR CONTROL TRANSFER

- Function call requires processor to start executing instructions from callee and when returned from callee, continue execution of instructions following call instruction
- This mechanism can be implemented using simple branch / jump instruction.
  - X86 processor has `jmp` instruction which jumps execution control to specified address

```
int main() {
    foo(5);
}
int foo(int p) {
    return p + 5;
}
```

```
main:
    ...
    jmp foo;
    ...
foo:
    ...
```

MECHANICS FOR CONTROL TRANSFER (CONT)

- But what about transferring control back to caller?
  - How does callee know where to return to?
- Save address of where to return to before doing the call
- Use this saved address to jump back
- But which address really?
- Where to save it?

RETURN ADDRESS AS IMMEDIATE/LABEL

- Do we really need to save address? Why not simply jmp back to required address?

```c
int main() {
    foo(5);
}
int foo(int p) {
    return p + 5;
}
```

```
main:
                ...
0x804854e:   jmp foo
0x8048553:
                ...
foo:
                ...
         jmp 0x8048553
```

- What are issues with this scheme?
  - `jmp` in foo will always return to same address.
  - Won't work if foo is called multiple times from main or from different function.
  - Return address is different for every invocation of a function

RETURN ADDRESS IN GLOBAL VARIABLE

- Say we create a dummy global variable retAddr_foo that holds return address

```
int main() {
    foo(5);
}
int foo(int p) {
    return p + 5;
}
```

```
.bss
retAddr_foo:
    .space 4
main:
    movl lbl, retAddr_foo
    jmp foo
  lbl:
    ...
foo:
    ...
    jmp *retAddr_foo
```

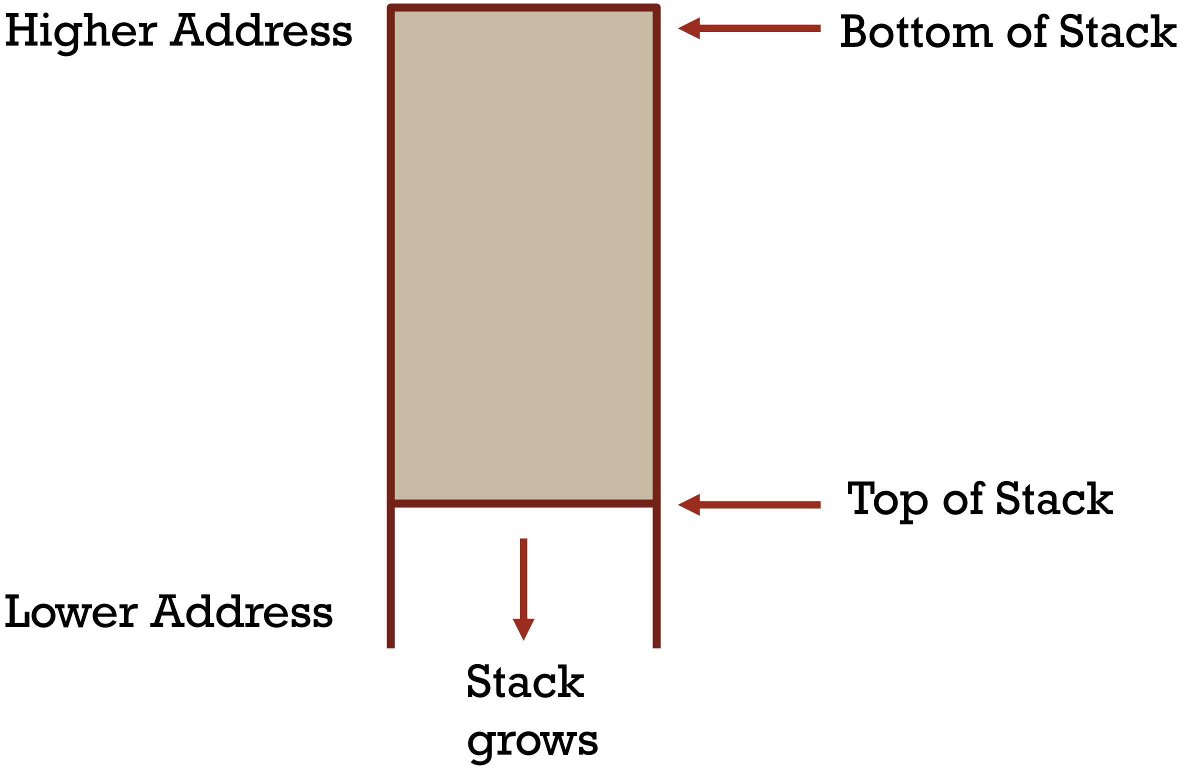
- What are issues with this scheme?
  - Both caller and callee function must agree on name and location of global variable. May not be best interface design.
  - Will this work for all cases?
    - How will recursive functions work in this model?

HANDLING OF RETURN ADDRESS (CONT)

- So far, we know:
  - Both caller and callee **must agree** on location of return address.
  - Storage used for return address is specific to every invocation of the function and hence must be created for every function invocation

STACK

- Many programming languages support stack-based execution for supporting recursion.
- Stack is a region of memory that works in LIFO manner.
- x86 and many other modern processors have native support for stack in HW.



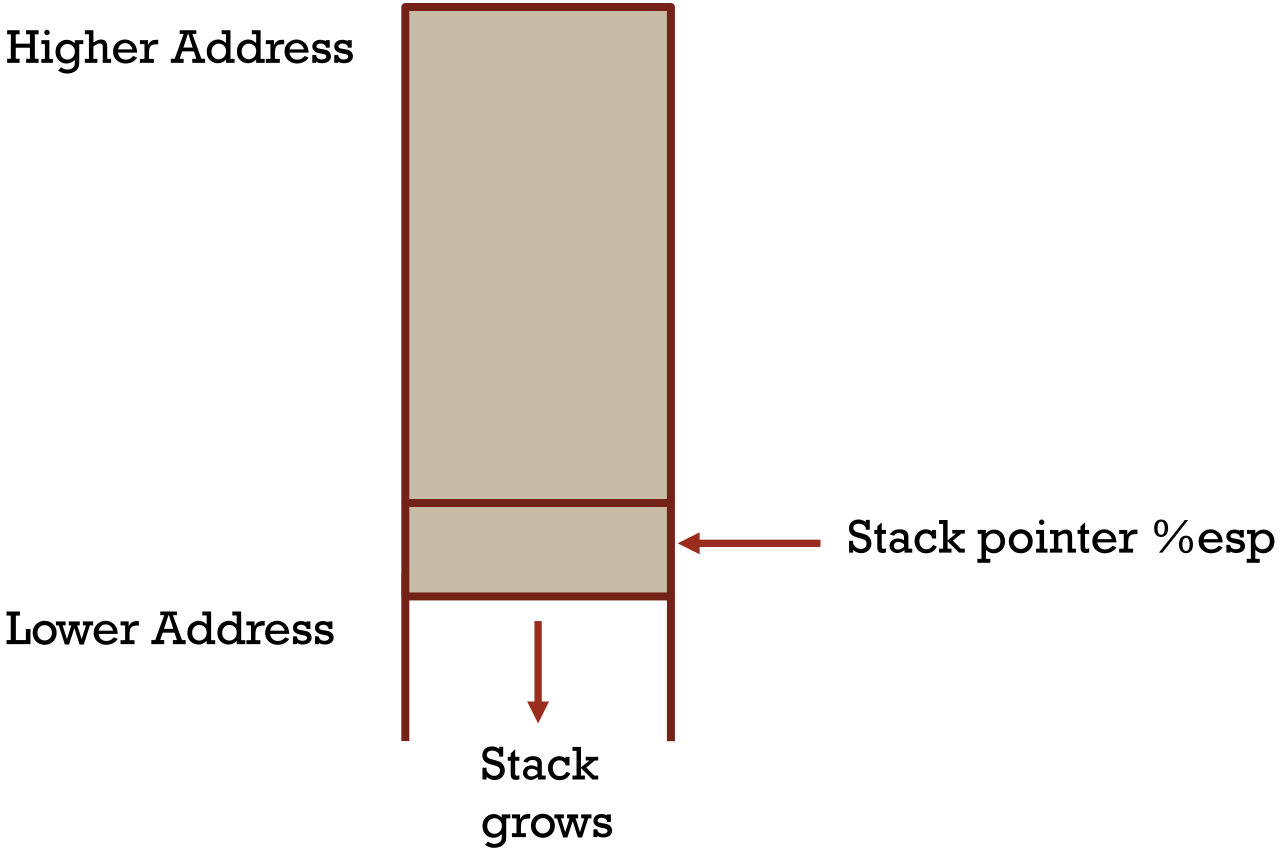
X86 STACK

- HW register `%esp` acts as stack pointer
- Push new data on stack
  - Instruction `pushl src`
  - Store value at address given by `%esp`
  - Decrement `%esp` by 4
  - `pushl %eax` equivalent to

```
subl $4, %esp
movl %eax, (%esp)
```

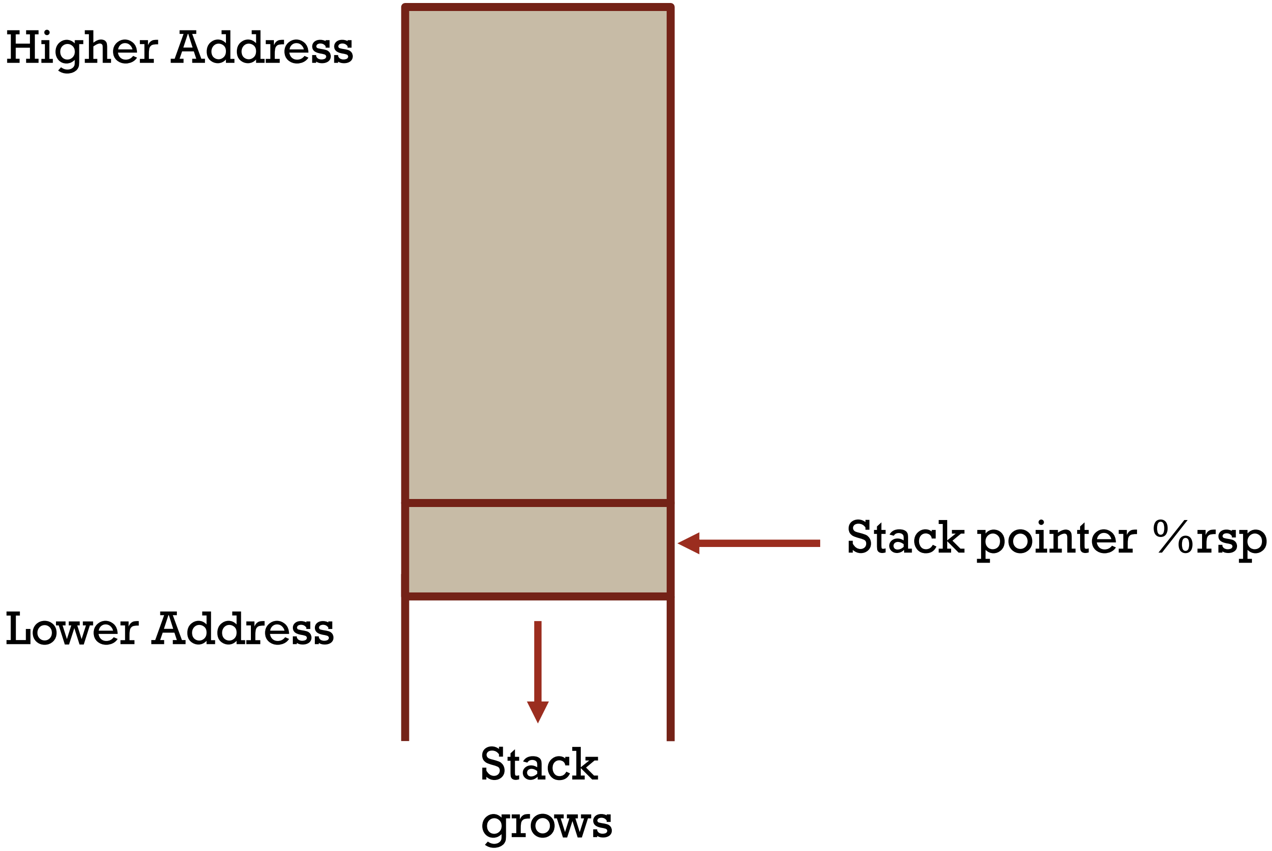
- Pop data from stack
  - Instruction `popl dst`
  - Loads value at address given by `%esp` in `dst`
  - Increments `%esp` by 4
  - `popl %eax` Equivalent to

```
movl %(esp), %eax
addl $4, %esp
```



X86_64 STACK

- HW register `%rsp` acts as stack pointer
- Push new data on stack
  - Instruction `pushq src`
  - Store value at address given by `%rsp`
  - Decrement `%rsp` by 8
- Pop data from stack
  - Instruction `popq dst`
  - Loads value at address given by `%rsp` in `dst`
  - Increments `%rsp` by 8

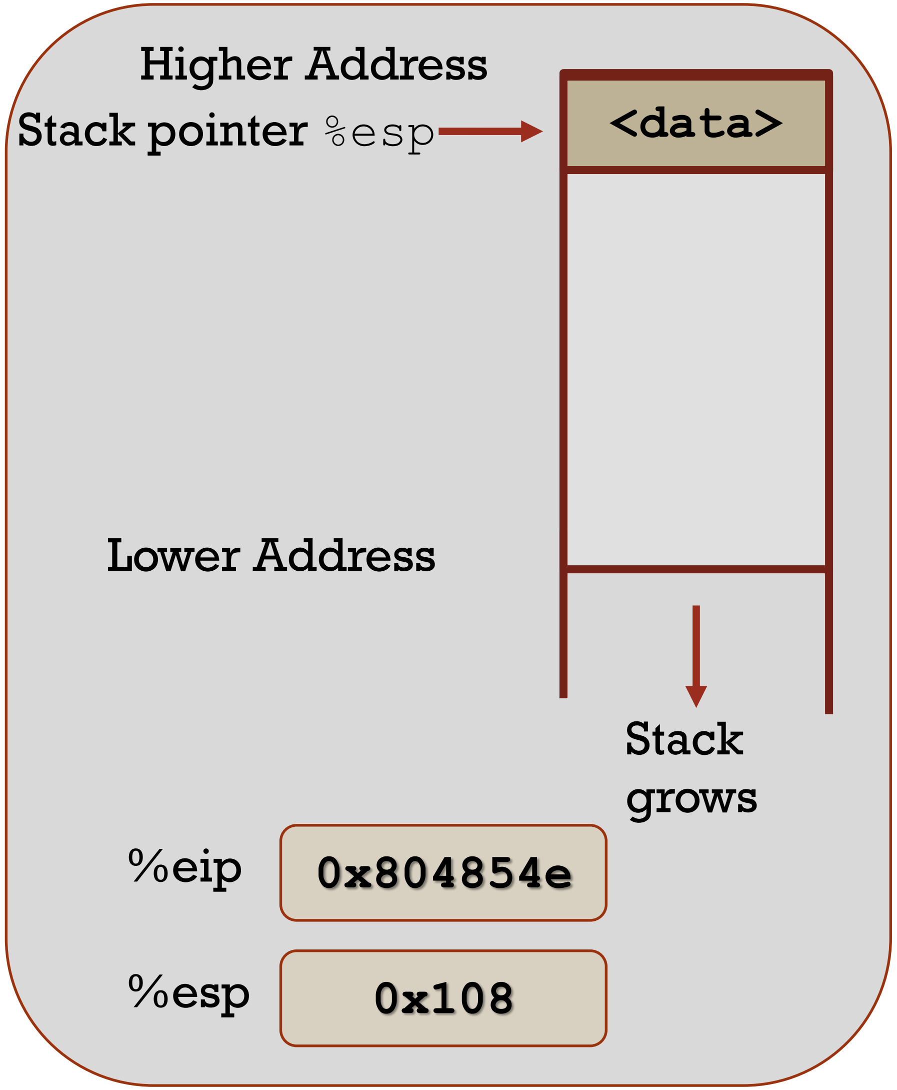


MECHANICS FOR CONTROL TRANSFER IN X86

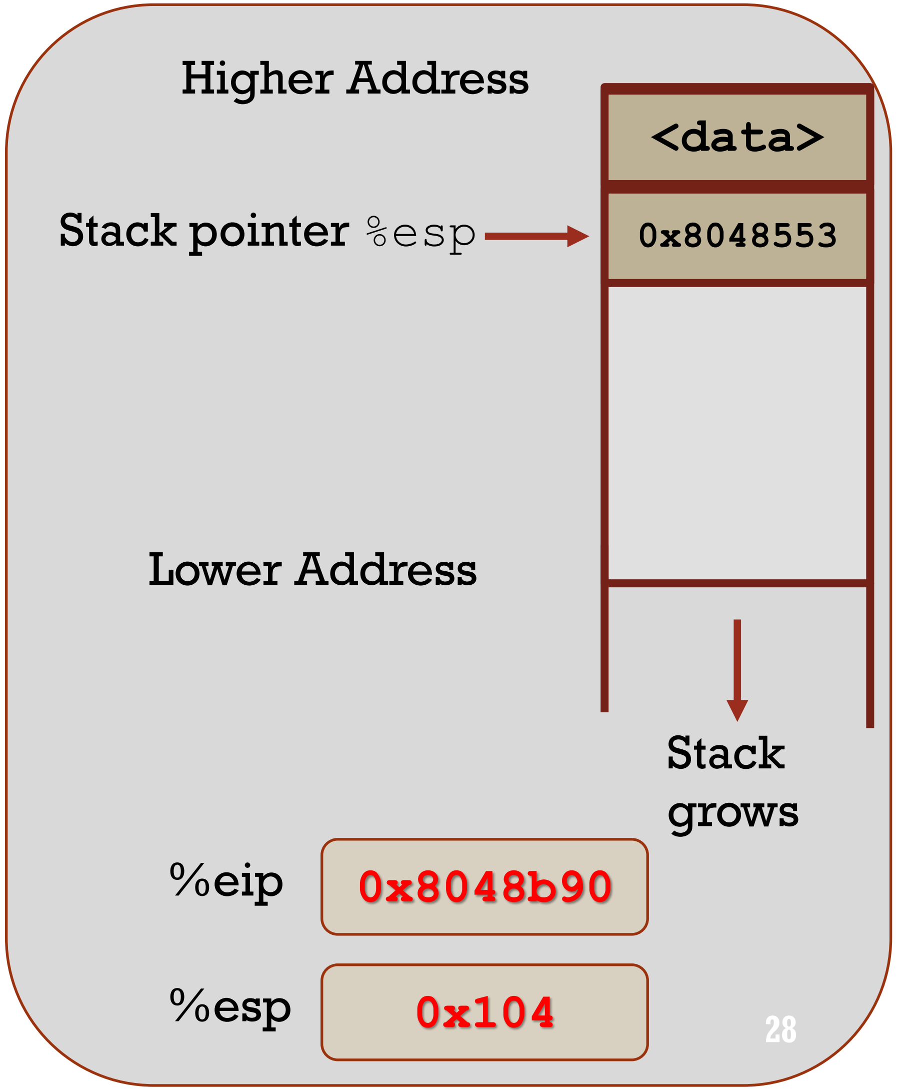
- x86 ISA supports following instructions for implementing function calls and returns

Instruction	Description
call Label	Function call • Pushes return address on the stack and jump to start of the called function
ret	Return from call • Pops address from top of stack and jumps to that location • Proper use requires just before instruction, stack is setup such that stack pointer points to place where call stored return address

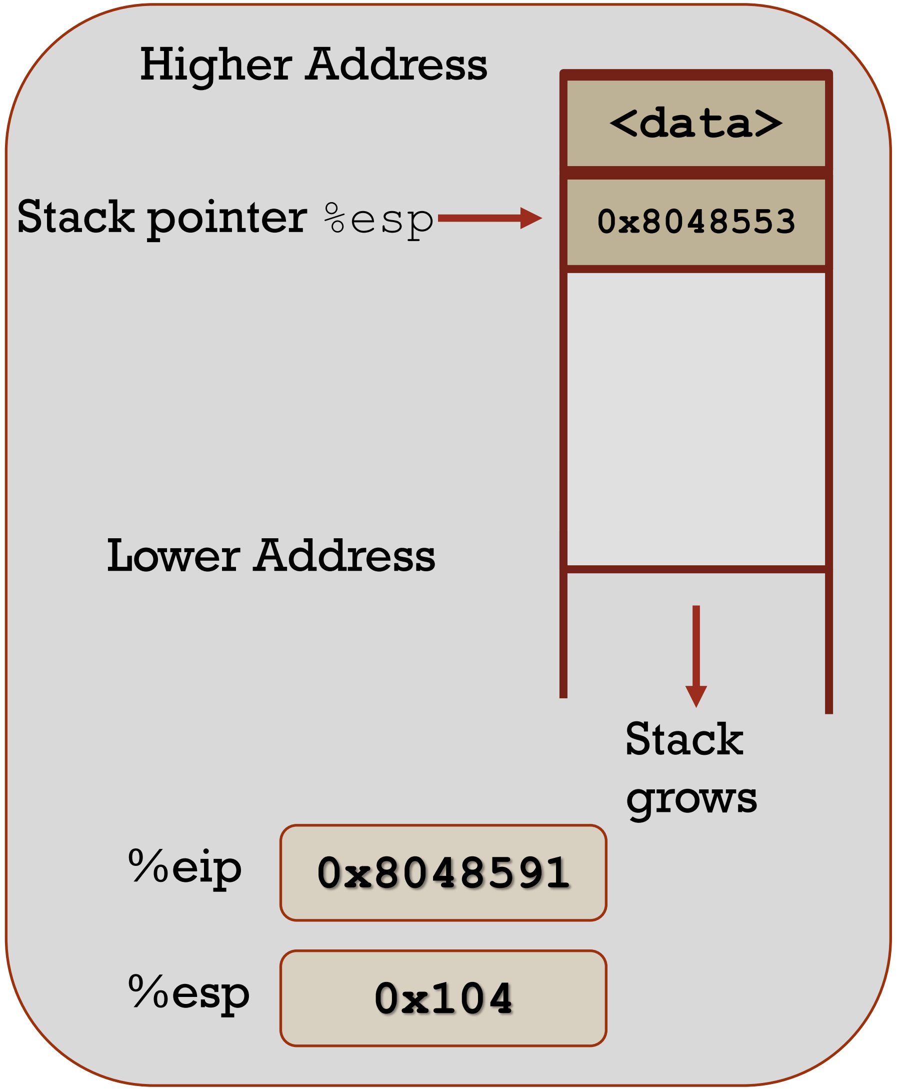
FUNCTION CALL DEMONSTRATION



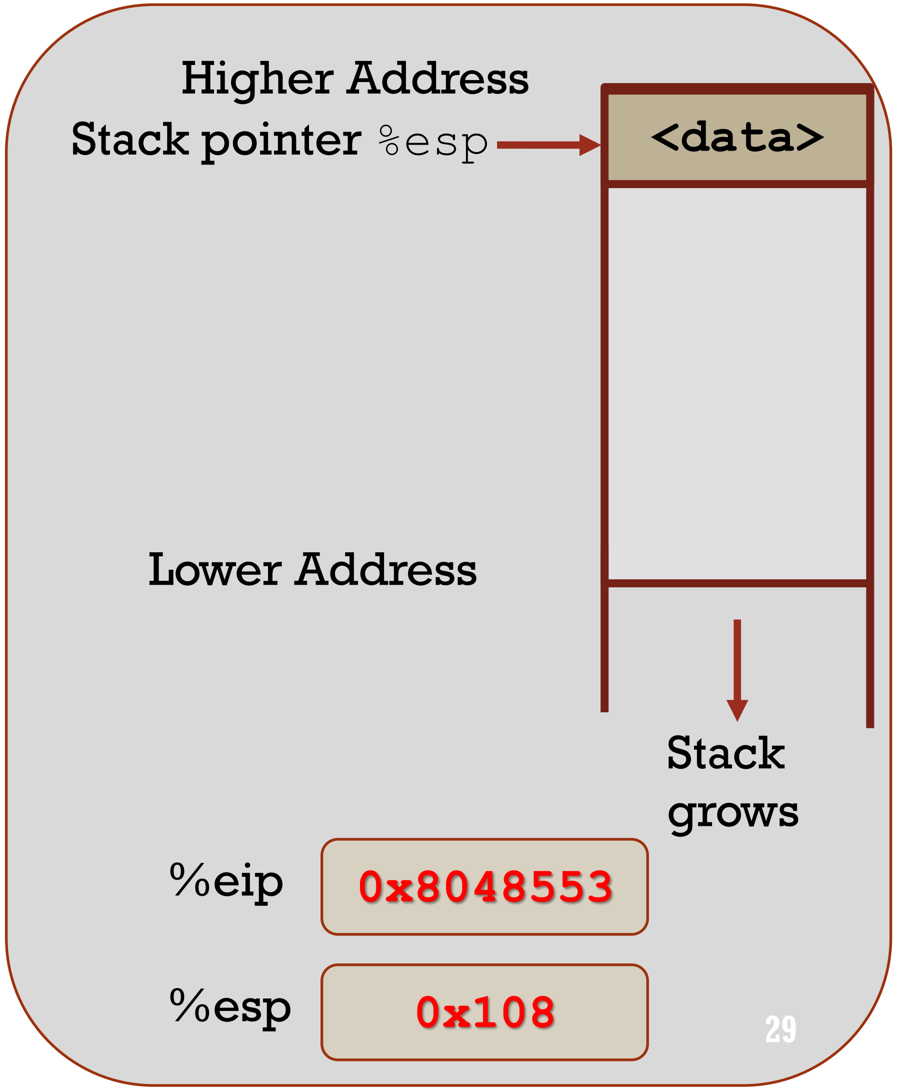

```
0x804854e: call 8048b90
0x8048553: movl $0, %eax
```



FUNCTION RETURN DEMONSTRATION




```
0x8048591: ret
```



MECHANICS FOR LOCAL VARIABLES IN FUNCTION

- A function can defined its own variables often referred as local variables
- Languages like C/C++ support two types of local variables
  - Variables with `auto` storage (default)
  - Variables with `static` storage
- What is the scope of function local variables?
  - Local variables are accessible only in the function
- What is lifetime of function local variables?
  - Local variables with `auto` storage have lifetime of function
  - `static` local variables have lifetime of entire program
- When are `auto` variables allocated?
  - Each invocation of function has its own instantiation of `auto` variables
  - Dynamically allocated, initialized every time function is called
  - Deallocated every time function returns
- When are `static` variables allocated?
  - When program is loaded, details will not be covered in this talk

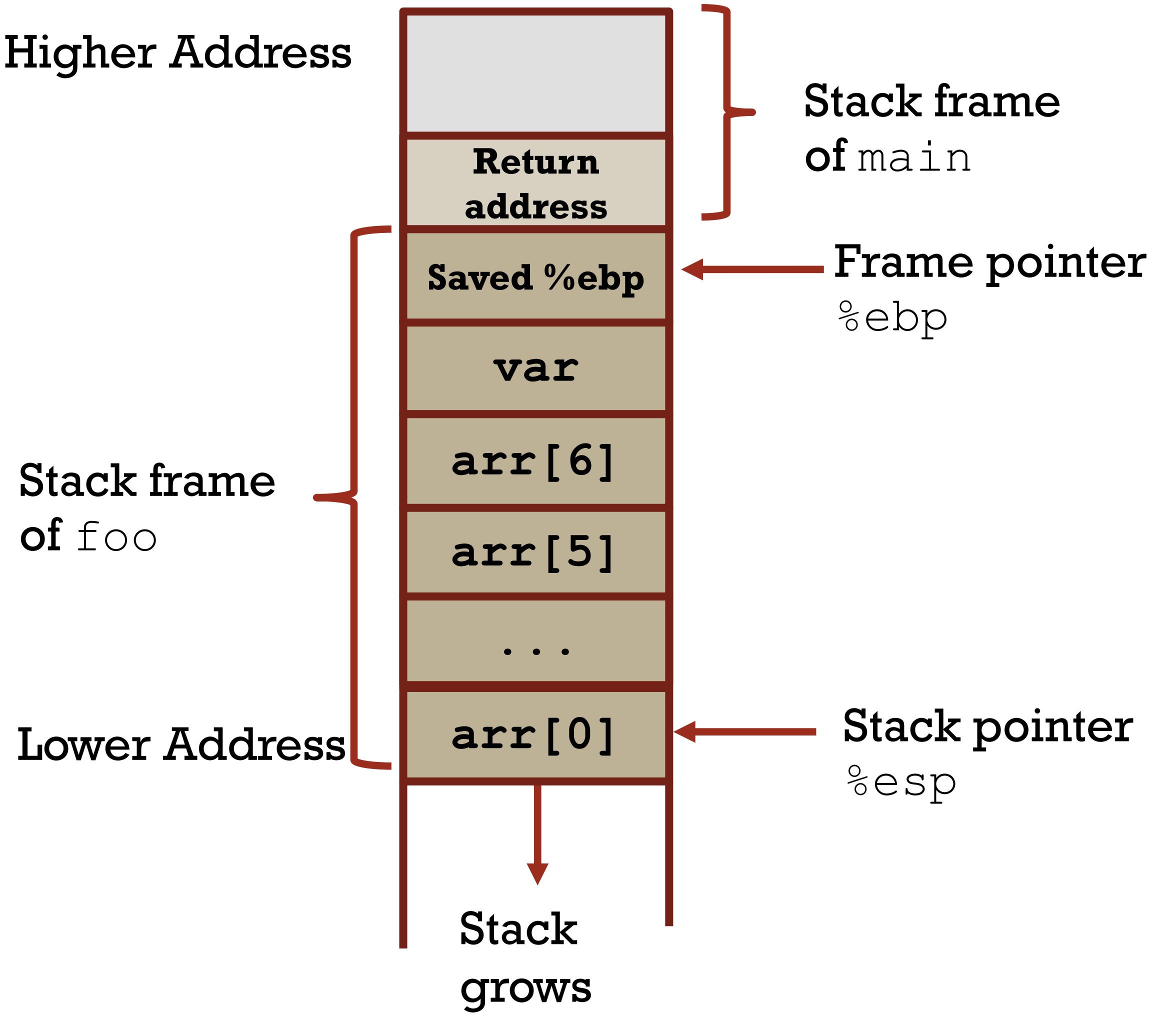
```
int foo(int m) {
    int var = 5;
    int arr[10];
    static int cnt;
    ...
    return arr[m];
}
```

STACK FRAME IN X86

- Portion of stack allocated for a function call is called "*stack frame*".
  - An implementation of "*Activation Records*" for stack-based languages
- Apart from stack pointer, also has frame pointer `%ebp`

```
int main(void) {
    ...
    q = foo();
    printf("%d", q);
}
```

```
int foo() {
    int var = 5;
    int arr[7];
    ...
    return arr[var];
}
```



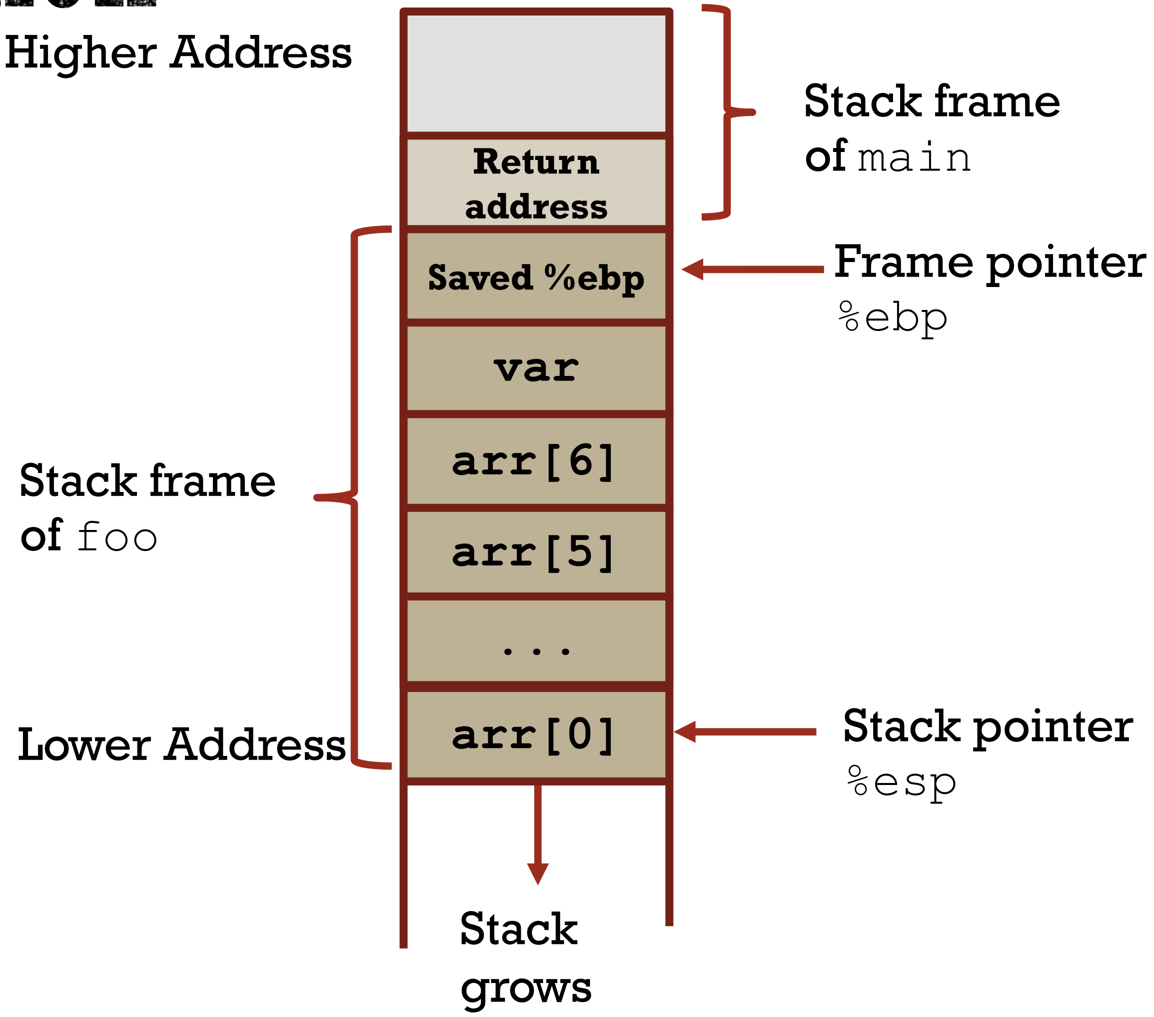
ALLOCATE SPACE IN STACK

- Function `foo` requires 4 + 7 * 4 = 32 bytes of stack for holding `auto` local variables.

```
foo:
    subl $32, %esp
    ...
```

```
int main(void) {
    ...
    q = foo();
    printf("%d", q);
}
```

```
int foo() {
    int var = 5;
    int arr[7];
    ...
    return arr[var];
}
```



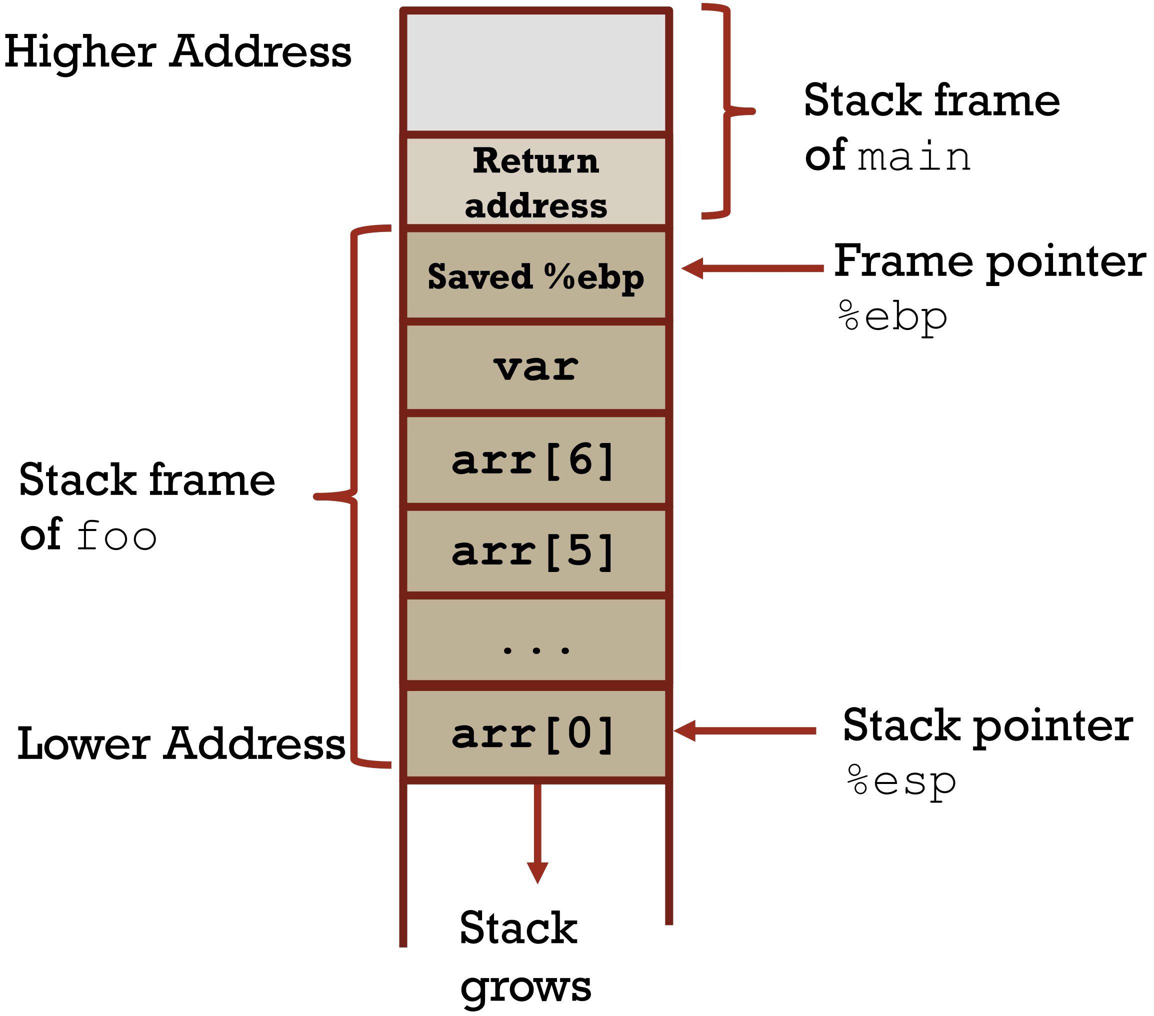
ACCESS AUTO LOCAL VARIABLES IN STACK

- How does program access `var`?
  - What is address of `var`?

```
foo:
    subl  $32, %esp
    ...
    movl    $5, 28(%esp)
```

```
int main(void) {
    ...
    q = foo();
    printf("%d", q);
}
```

```
int foo() {
    int var = 5;
    int arr[7];
    ...
    return arr[var];
}
```



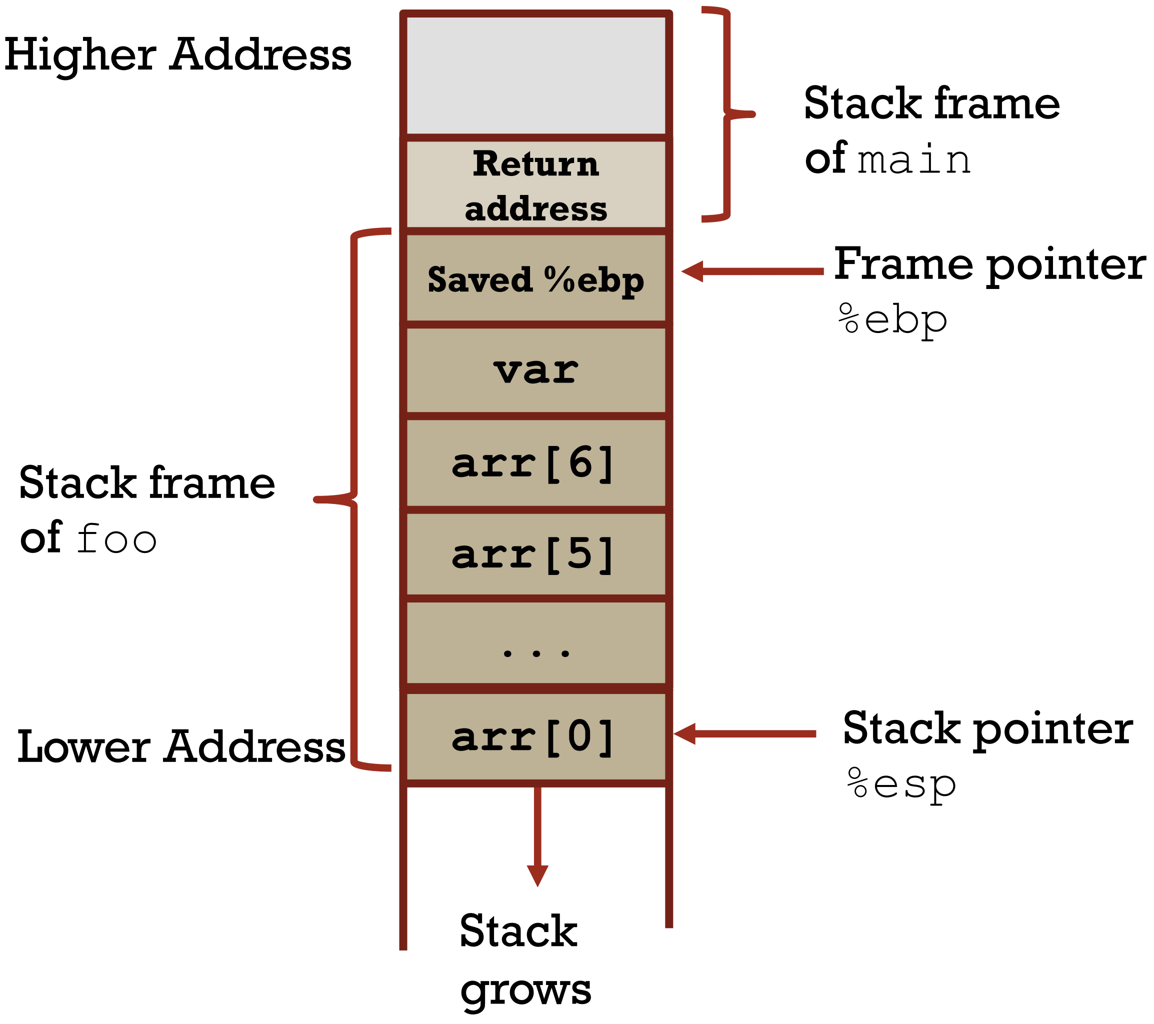
FRAME POINTER

- Why is frame pointer needed?
  - As stack grows, address of variables relative to stack pointer will change
  - Variables always at fixed address from frame pointer, `var` will always be at `%ebp - 4`
  - `var` can be accessed as

    ```
    movl   $5, -4(%ebp)
    ```

```
int main(void) {
    ...
    q = foo();
    printf("%d", q);
}
```

```
int foo() {
    int var = 5;
    int arr[7];
    ...
    return arr[var];
}
```



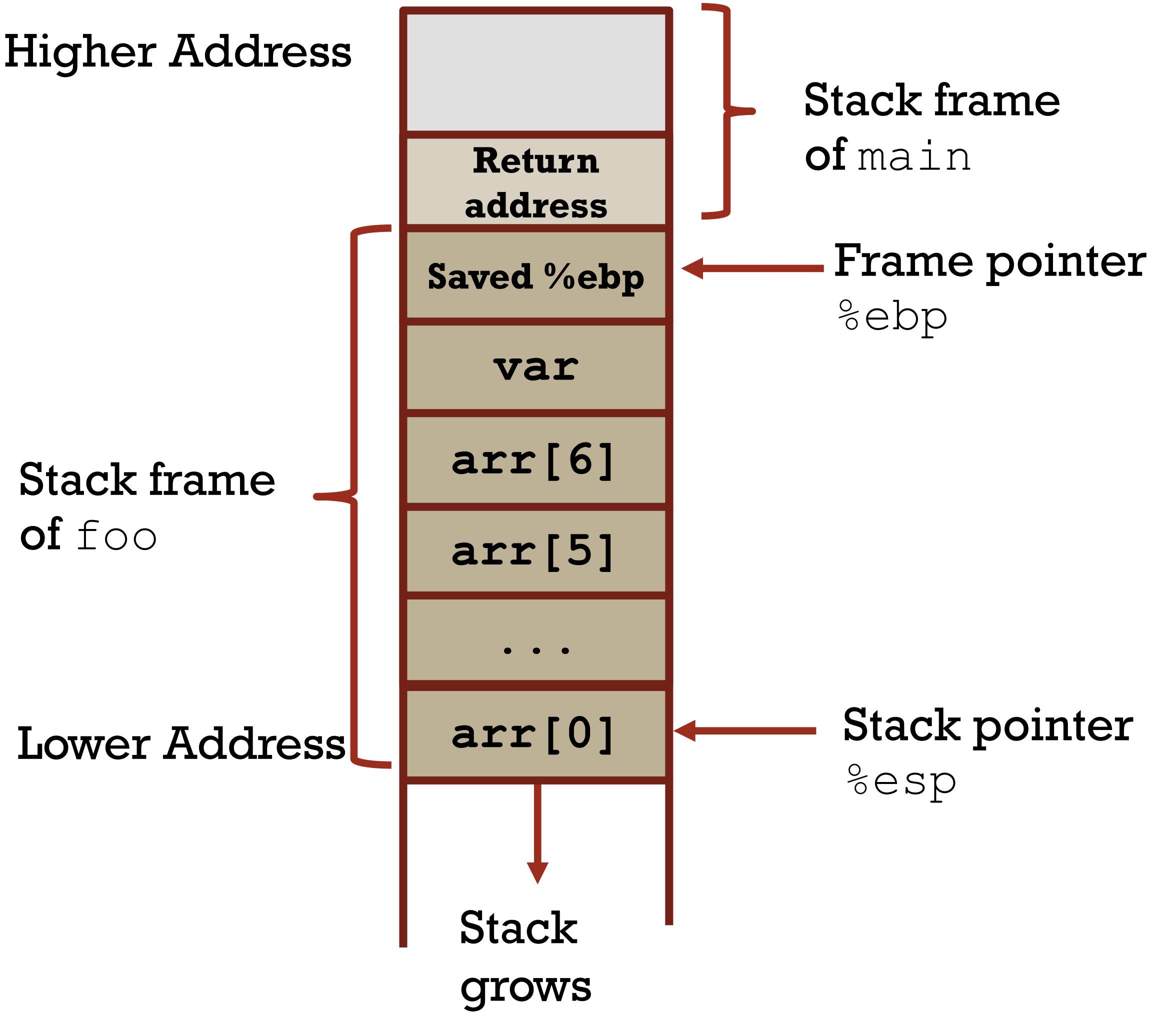
PREPARE STACK FOR RETURN

- What all is needed to return to caller?
  - Deallocate stack allocated for local variables etc.
  - Need to ensure stack pointer is pointing to return address

```
foo:
    ...
    movl %ebp, %esp
    popl %ebp
    ret
```

```
int main(void) {
    ...
    q = foo();
    printf("%d", q);
}
```

```
int foo() {
    int var = 5;
    int arr[7];
    ...
    return arr[var];
}
```



FUNCTION PROLOGUE AND EPILOGUE

Function Prologue	**Function Epilogue**
Lines of code at beginning of function to prepare stack and registers for use within the function	Reverses action of prologue and returns control to caller function
Instruction `enter N, 0` is equivalent to prologue shown below	Instruction `leave` is equivalent movl + popl instruction

```
foo:
 pushl %ebp       ; save %ebp
 movl %esp, %ebp ; update %ebp
 subl $<N>, %esp ; space in stack
 ...
```

```
foo:
    ...
    movl %ebp, %esp ; Update %esp
    popl %ebp       ; restore %ebp
    ret             ; return
```

MECHANICS FOR FUNCTION PARAMETERS AND RETURN VALUES

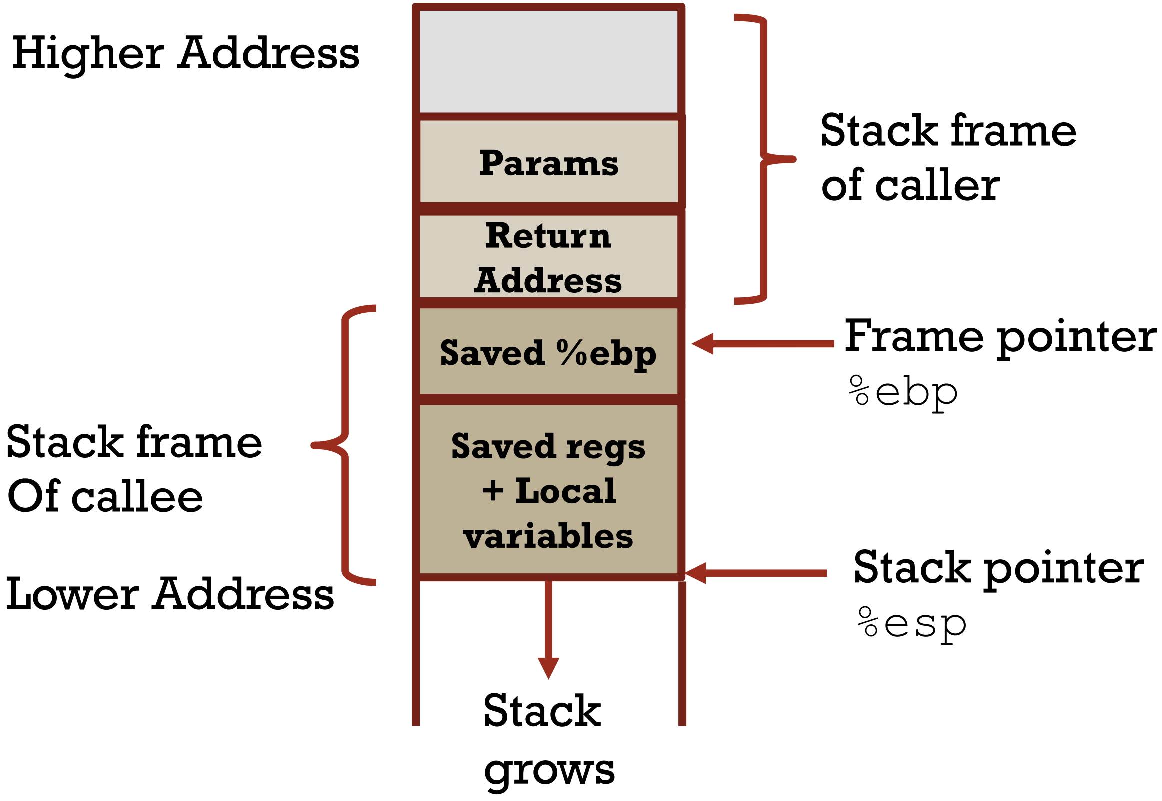
- How does callee know where to find parameters?
- How does caller know where to find return value?
- Need to have some contract between caller and callee function for this.

```
CallerFunction() {
   set up parameters
   call
   read return value
```

```
CalleeFunction() {
   create local vars
   read parameters
   ...
   create return value
   free local vars
   ret
```

STACK FRAME X86 REVISITED

- Caller function allocates parameters in stack frame just before call instruction which will push return address on stack
- This forms yet another contract between caller and callee

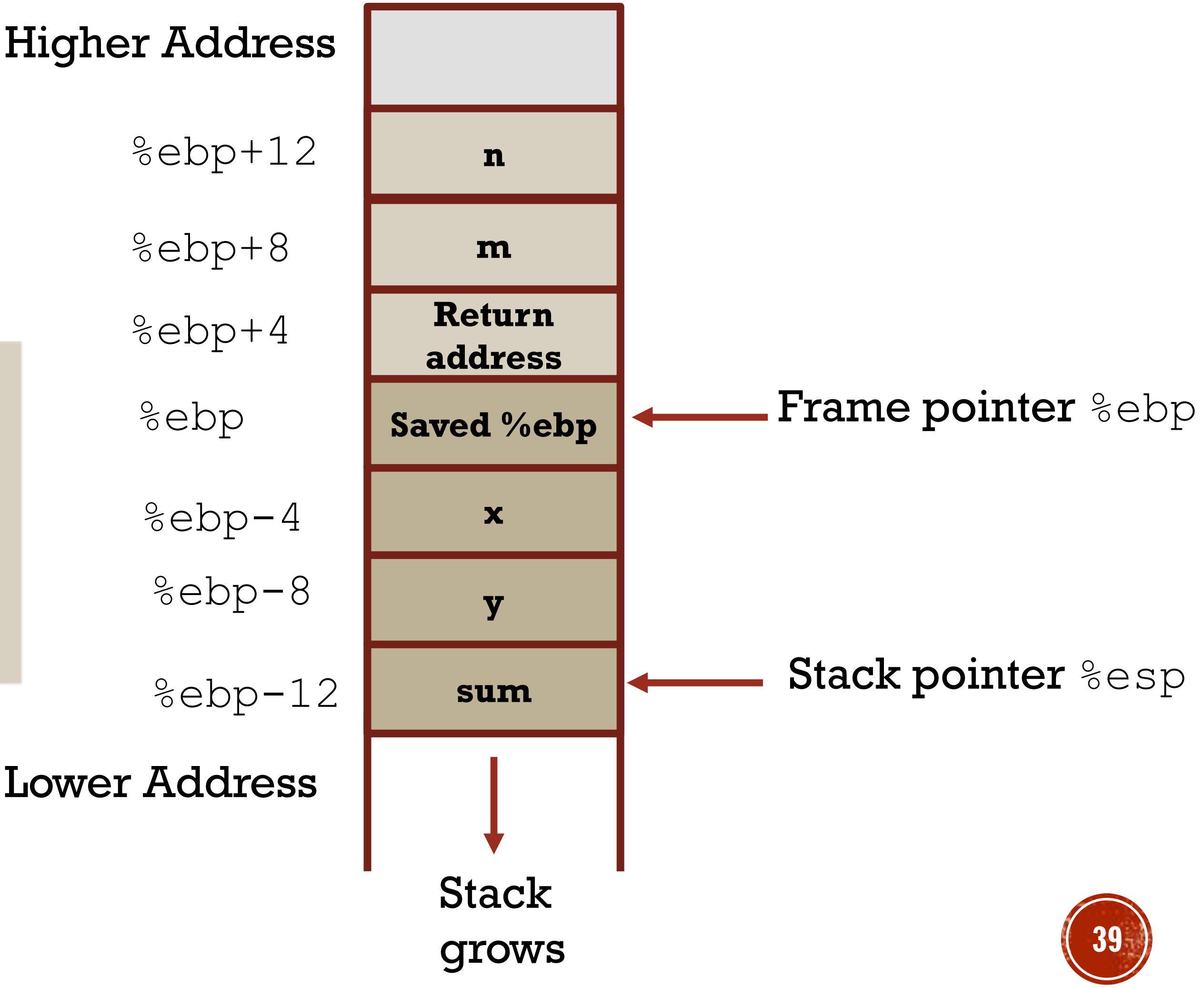


STACK FRAME IN X86 (CONT)

- How to access parameter `m`?
  - `%ebp + 8`

```
int main(void) {
    ...
    r = foo(p, q);
    printf("%d", r);
}
```

```
int foo(int m, int n) {
    int x = m + n;
    int y = 5;
    int sum = x+y;
    ...
    return sum;
}
```

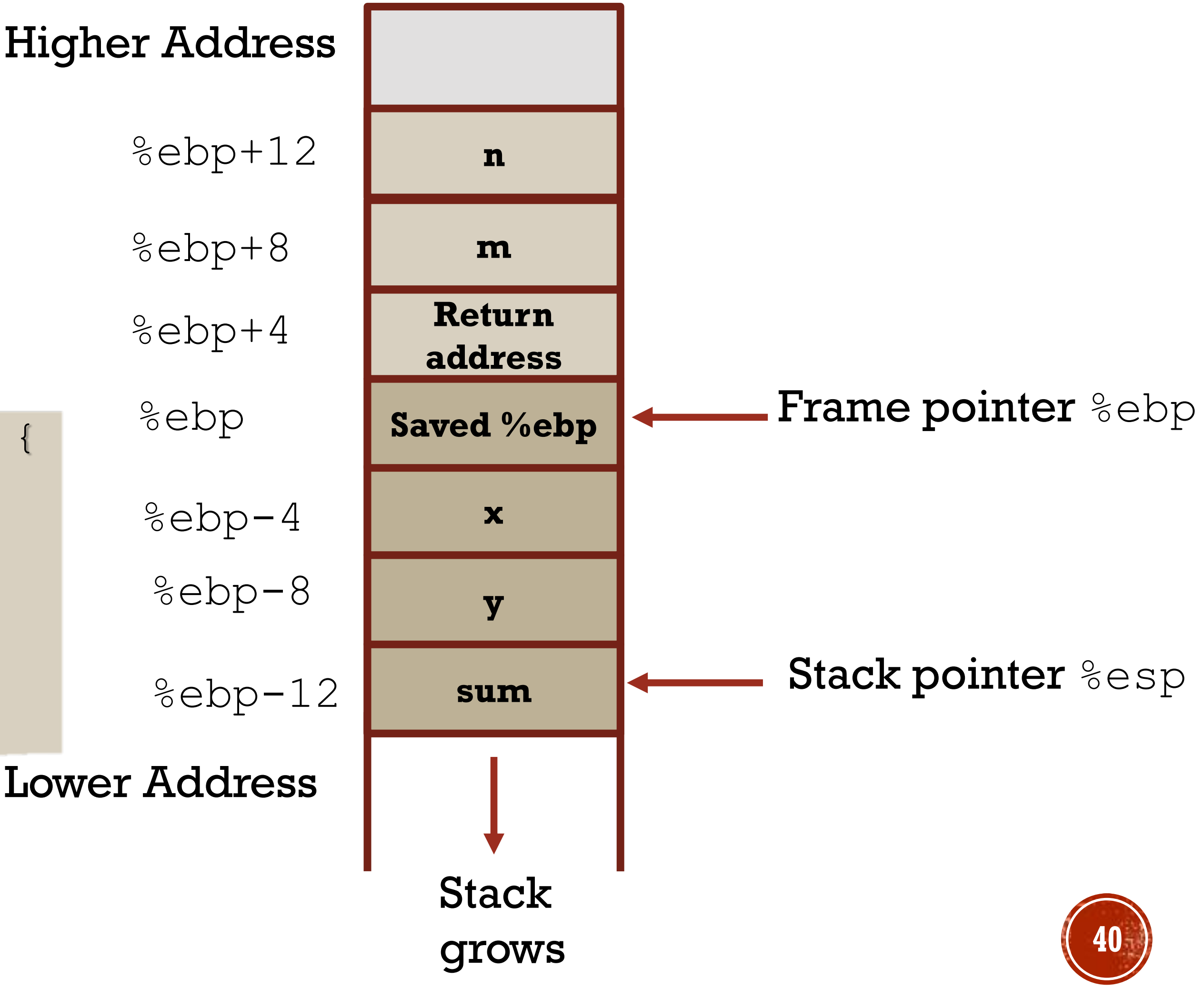


STACK FRAME IN X86 (CONT)

- What should be order of parameters in the stack?
  - Should they be passed from right to left or left to right or some other order?
  - Does this even matter as long as caller and callee agree on order?
    - Varargs functions

```
int main(void) {
    ...
    r = foo(p, q);
    printf("%d", r);
}
```

```
int foo(int m, int n) {
    int x = m + n;
    int y = 5;
    int sum = x+y;
    ...
    return sum;
}
```



HOW TO HANDLE RETURN VALUES?

- Most programming languages support only one return value from a function
- Most of the implementations hence support return value using a register
- x86 uses register `%eax` for return value.
- BUT Wait, what about larger objects say like struct returned by function?
  - x86 requires caller to allocate space and pass a pointer to that as a "hidden" parameter on stack. Callee will write return value to this address.

MECHANICS FOR REGISTER USAGE

- Both caller and callee functions are running on same HW sharing HW state and resources.
- So what happens when both caller and callee need to make use of same HW registers?

```
caller:
    ...
    movl $5, %edx
    call callee
    addl %edx, %eax
```

```
callee:
    ...
    movl $10, %edx
    ...
    ret
```

Overwrites %edx

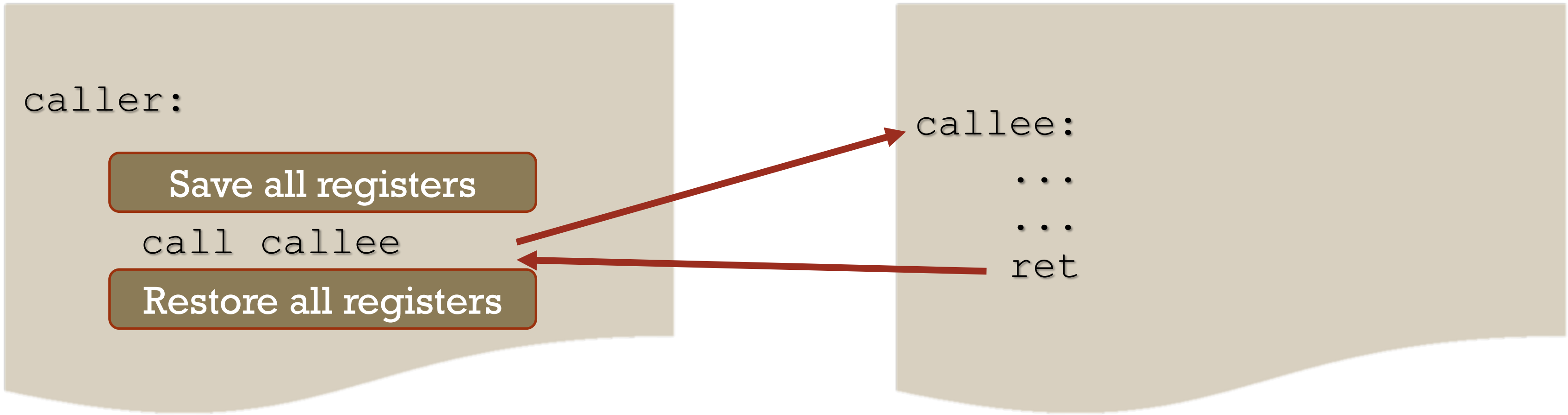
- Somehow `%edx` must be saved so that caller function can use it after call instruction
- Whose responsibility to save `%edx`?
  - Need some coordination between `caller` and `callee`

REGISTER SAVING CONVENTIONS

- Requirement:
  - Caller function should save (and restore) registers it is using before and after call; callee function should save (and restore) registers it is using in function body.
- Possible redundancy?
  - If callee function is using a register that is saved and restored by caller, does callee function really need to save (and restore) it?
  - If caller function is using a register saved and restored by callee, does caller function really need to save (and restore) it?
- Does caller know which registers are used by callee? Does callee know which registers are used by caller?
  - Inter-procedural analysis and optimizations?
  - What to do in Separate compilation?

CONVENTION: CALLER SAVES ALL REGISTERS

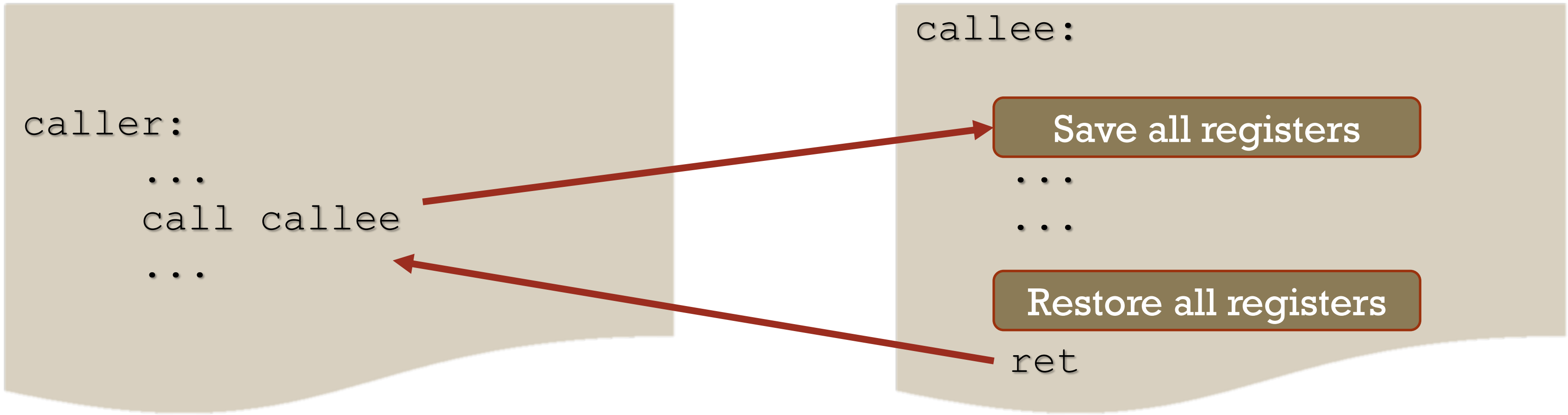
- Caller function saves all registers before calling anything and restores them after call is done



- Every caller function needs to do save and restore at every invocation of call
- May end up saving and restoring registers that are not used by callee function at all
- Does caller really need to save and restore *"ALL"* registers?
  - What if a register is used by caller only before call but is not used after call (i.e. dead register)?

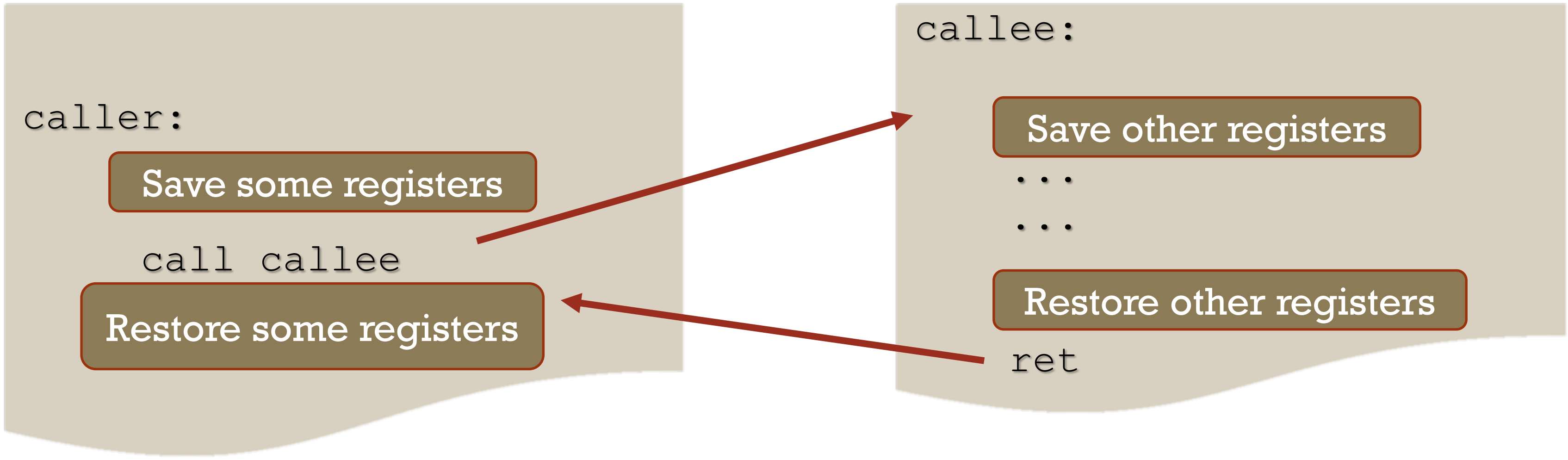
CONVENTION: CALLEE SAVES ALL REGISTERS

- Callee function saves all registers before at start of the function and restores them before returning



- Code to do save, restore is at one place only, in the body of callee function
- May still end up saving and restoring registers that are not used by caller function at all or registers that are "dead"?
- Does callee really need to save and restore *"ALL"* registers?
  - What if a register is not at all used by callee function?

CONVENTION: TRY TO GET BEST OF BOTH WORLDS



REGISTER SAVING CONVENTIONS

- Split available registers into two sets
- *Caller saved Registers*
  - Will be caller's responsibility to save these registers before calling the function and restoring them after the call
  - Callee function can use and modify these registers freely
  - Also known as "*scratch registers*"
- *Callee saved Registers*
  - Will be callee's responsibility to save these registers before using them and restoring them before returning
  - Caller function expects value of these registers will be after function call
  - Also known as *"preserved registers"*
- Yet another agreement between caller and callee functions.

REGISTER SAVING CONVENTIONS FOR X86

- Caller saved:
  - %eax, %ecx, %edx
- Callee saved:
  - %ebx, %esi, %edi
- Special callee saved:
  - %esp, %ebp

CONTRACTS BETWEEN CALLER AND CALLEE

- So far, we've seen following things that caller and callee need to agree upon
  - Where is return address located?
    - x86 stored this on stack as part of call instruction
  - How are parameters passed and in what order?
    - x86 passed them on stack and calling conventions for C passed them from right to left order
  - Where is return value?
    - x86 used `%eax` to return value, with larger structs being handled as hidden parameters
  - Which registers are caller saved and which are callee saved
    - x86 had following conventions
      - Caller saved: `%eax, %ecx, %edx`
      - Callee saved : `%ebx, %esi, %edi`
      - Special callee saved: `%esp, %ebp`
- Such contracts between caller and callee functions are referred as *"Calling Conventions"*

CALLING CONVENTIONS

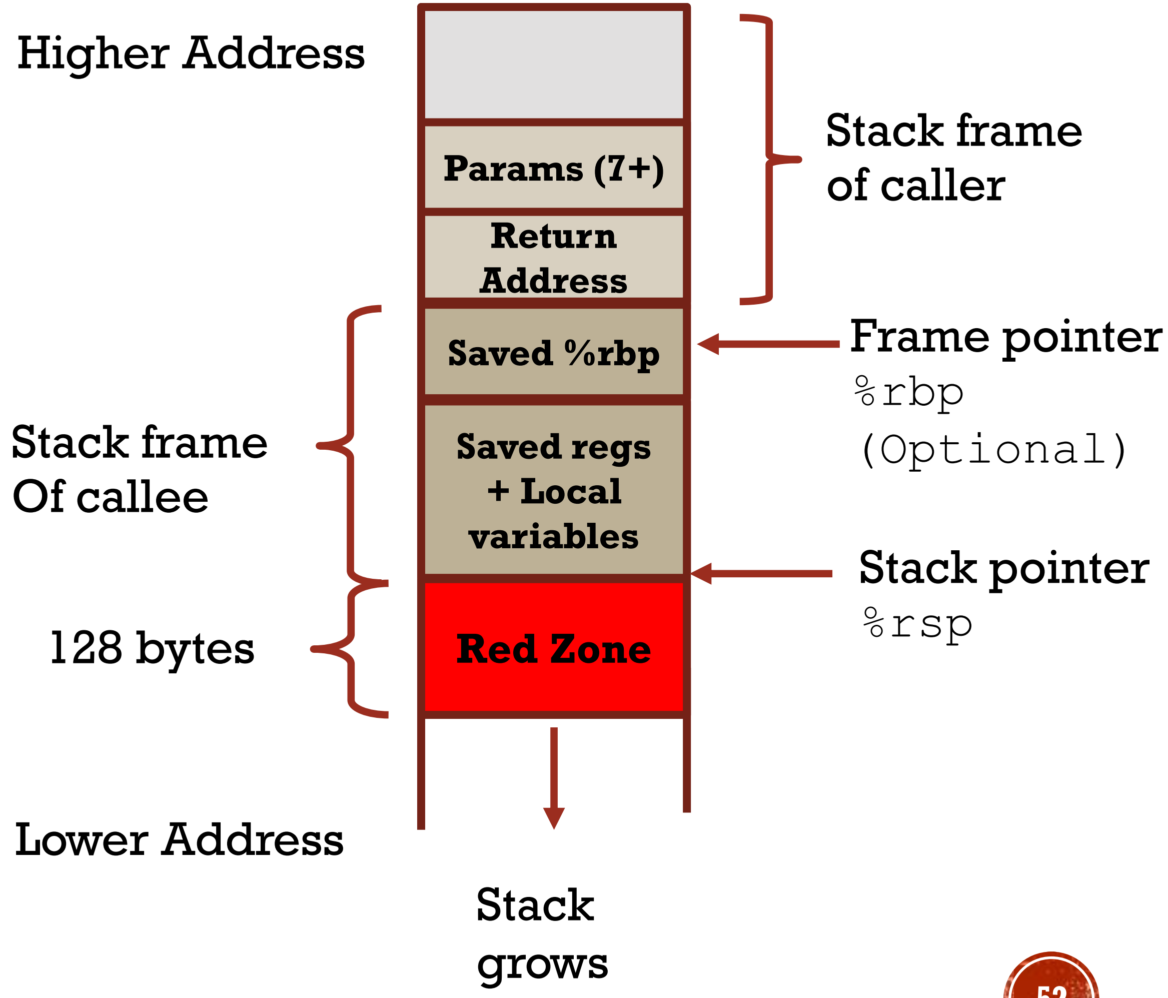
- Who decides calling conventions?
  - Application Binary Interface (ABI)
  - Every HW and Operating System specify ABI to be followed for that platform.
    - Contains more than just calling conventions.
    - Compiler implementation for a platform conforms to the ABI of that platform.
- Is there only one calling convention specified by ABI?
  - What are issues of having only one calling convention?
  - Platforms support different calling conventions as well mostly for performance reasons e.g. Windows supports `__fastcall` that passes first two arguments in registers `%ecx`, `%edx`, supports bunch of more `stdcall`, `safecall` etc.
- Can calling conventions be relaxed by compiler as optimization?
  - What if all callers of a function are known?
    - `static` functions?
  - What if compiler knows it is whole program compilation?
- What constructs in C language make it hard for compiler to apply optimization related to relaxing calling conventions?
  - Extern function calls (separate compilation)
  - Function pointers

X86_64 CALLING CONVENTIONS

- x86 had only 8 general purpose registers.
- x86_64 extends them to 64bits (prefix "r" instead of "e"), also adds another 8 general purpose registers r8..r15
- Below details are for System V AMD64 ABI.
  - Linux follows this, Windows follows different ABI
- x86_64 calling convention makes use of more available registers and allows passing up to 6 function parameters in registers and remaining are passed via stack.
  - `%rdi, %rsi, %rdx, %rcx, %r8, %r9` hold first 6 parameters of a function.
- AMD64 ABI makes frame pointer optional i.e. `%rbp` can be used as general purpose.
  - References are done relative to stack pointer `%rsp`
- `%rax` continues to be return value register

X86_64 STACK FRAME FOR AMD64 ABI

- Remember first 6 parameters are not in stack
- Red Zone:
  - AMD64 ABI defines 128 bytes area beyond stack pointer as "Red Zone"
  - Optimization for leaf functions
  - Leaf functions can use red zone for scratch data without adjusting stack pointer
    - Saves two instructions as don't need to decrement `%rsp` and restore it
  - Red zone will NOT be clobbered by interrupt and signal handlers but will be clobbered by function calls hence useful only in leaf functions



WHAT ELSE FOR FUNCTION CALLS?

- Function Overloading
- Templates
- Virtual Functions
- Exception Handling and stack unwinding

WHAT OTHER TOPICS IN PROG EXECUTION ENV

- Object File formats
- Linker
  - Symbol Resolution
  - Merging object files
- Libraries
  - Static Libraries
  - Shared/Dynamic Libraries
    - Position Independent Code
- Loader
  - Process
  - Virtual Memory
  - C Runtime