Compilers:
From Programming to Execution

David Brumley
Carnegie Mellon University

You will find

at least one error

on each set of slides. :)

To answer the question

"Is this program safe?"

We need to know

"What will executing this program do?"

What will executing this program do?

```
#include <stdio.h>
void answer(char *name, int x){
  printf(“%s, the answer is: %d\n”,
            name, x);
}

void main(int argc, char *argv[]){
  int x;
  x = 40 + 2;
  answer(argv[1], x);
}
```

42.c

```
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
         name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 40 + 2;
  answer(argv[1], x);
}
```

Compilation

David

0011010
1101010
1000101

The *compiler* and *machine* determines the semantics

David, the answer is 42

"Compiled Code"

Source Language → **Compilation** → Target Language

Input → Target Language → Output

“Interpreted Code”

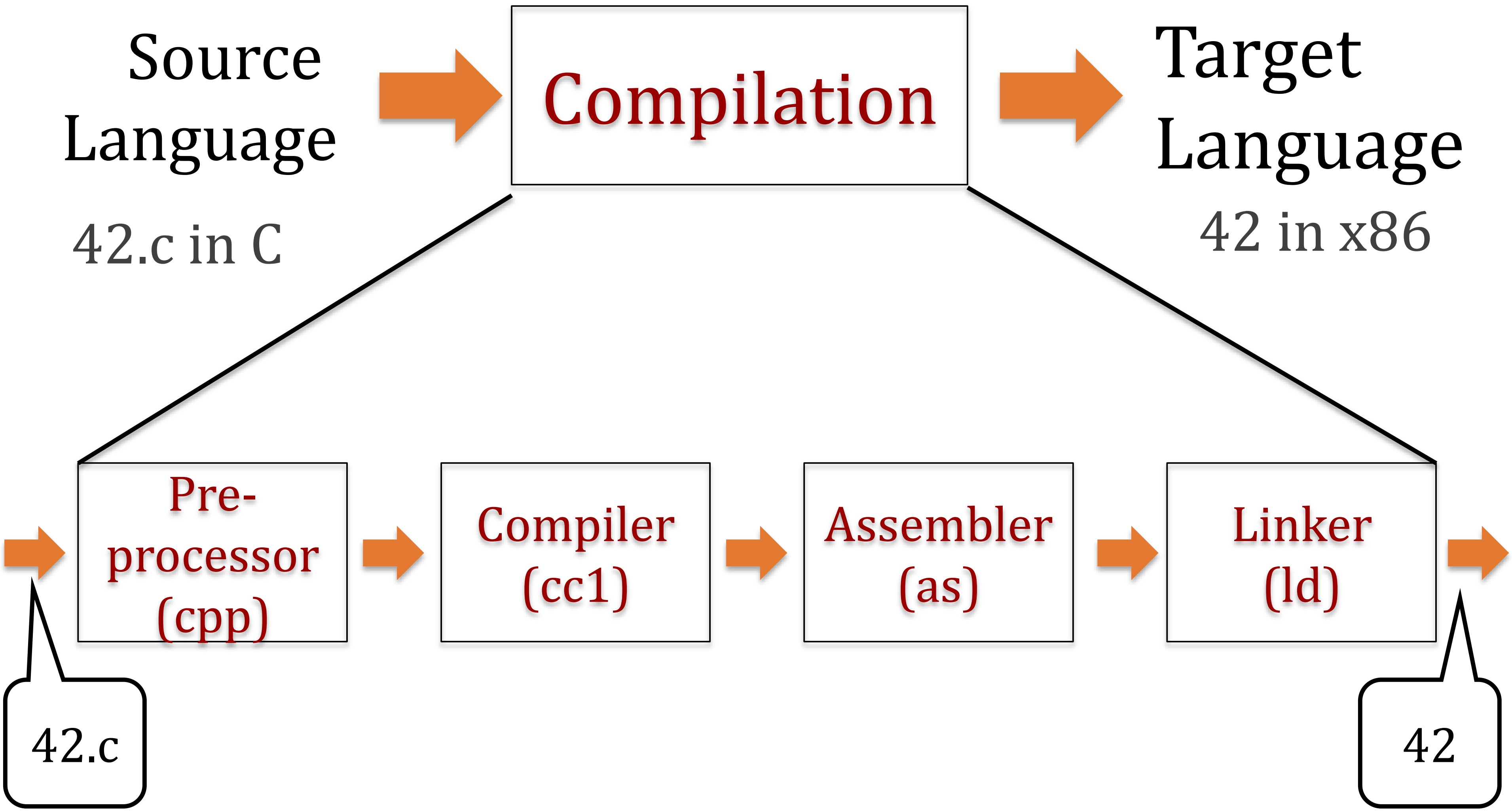
Today: Overview of Compilation

1. How is C code translated to executable code?
2. What is the machine model for executing code?

Key Concepts

- Compilation workflow
- x86 execution model
- Endian
- Registers
- Stack
- Heap
- Stack frames

Compilation Workflow


Source Language
42.c in C
Compilation
Target Language
42 in x86
Pre-processor (cpp)
Compiler (cc1)
Assembler (as)
Linker (ld)
42.c
42

Pre-processor (cpp) → Compiler (cc1) → Assembler (as) → Linker (ld) →

$ cpp

```
#include <stdio.h>
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
              name, x);
}
...
```

#include expansion

#define substitution

→ Pre-processor (cpp) → **Compiler (cc1)** → Assembler (as) → Linker (ld) →

$ gcc -S

```
#include <stdio.h>
void answer(char *name, int x){
  printf(“%s, the answer is: %d\n”,
                name, x);
}
...
```

Creates Assembly

gcc –S 42.c outputs 42.s

```
_answer:
Leh_func_begin1:
        pushq   %rbp
Ltmp0:
        movq    %rsp, %rbp
Ltmp1:
        subq    $16, %rsp
Ltmp2:
        movl    %esi, %eax
        movq    %rdi, -8(%rbp)
        movl    %eax, -12(%rbp)
        movq    -8(%rbp), %rax
        ....
```

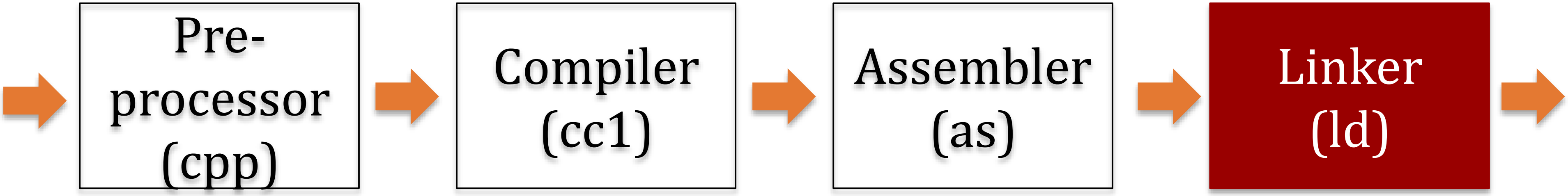
Pre-processor (cpp) → Compiler (cc1) → **Assembler (as)** → Linker (ld) →

$ as <options>

```
_answer:
Leh_func_begin1:
        pushq   %rbp
Ltmp0:
        movq    %rsp, %rbp
Ltmp1:
        subq    $16, %rsp
Ltmp2:
        movl    %esi, %eax
        movq    %rdi, -8(%rbp)
        movl    %eax, -12(%rbp)
        movq    -8(%rbp), %rax
        ....
```

42.s

Creates object code



$ ld <options>

```
01011001010101011010101010
10101010101010101111111100
00110101011010101001010111
01011110101001011000001010
10111101
```

42.o

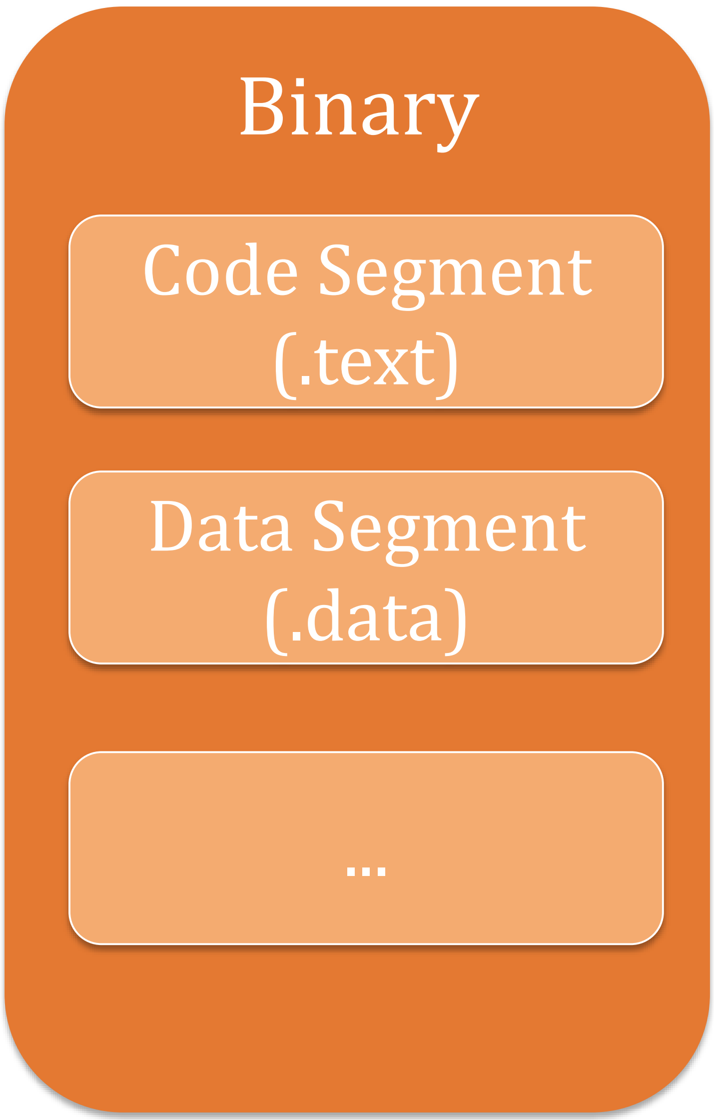
Links with other files and libraries to produce an exe

Disassembling

- Today: using objdump (part of binutils)
  – objdump –D <exe>

- If you compile with "-g", you will see more information
  – objdump –D –S

- Later: Disassembly



The program ***binary***
(aka executable)

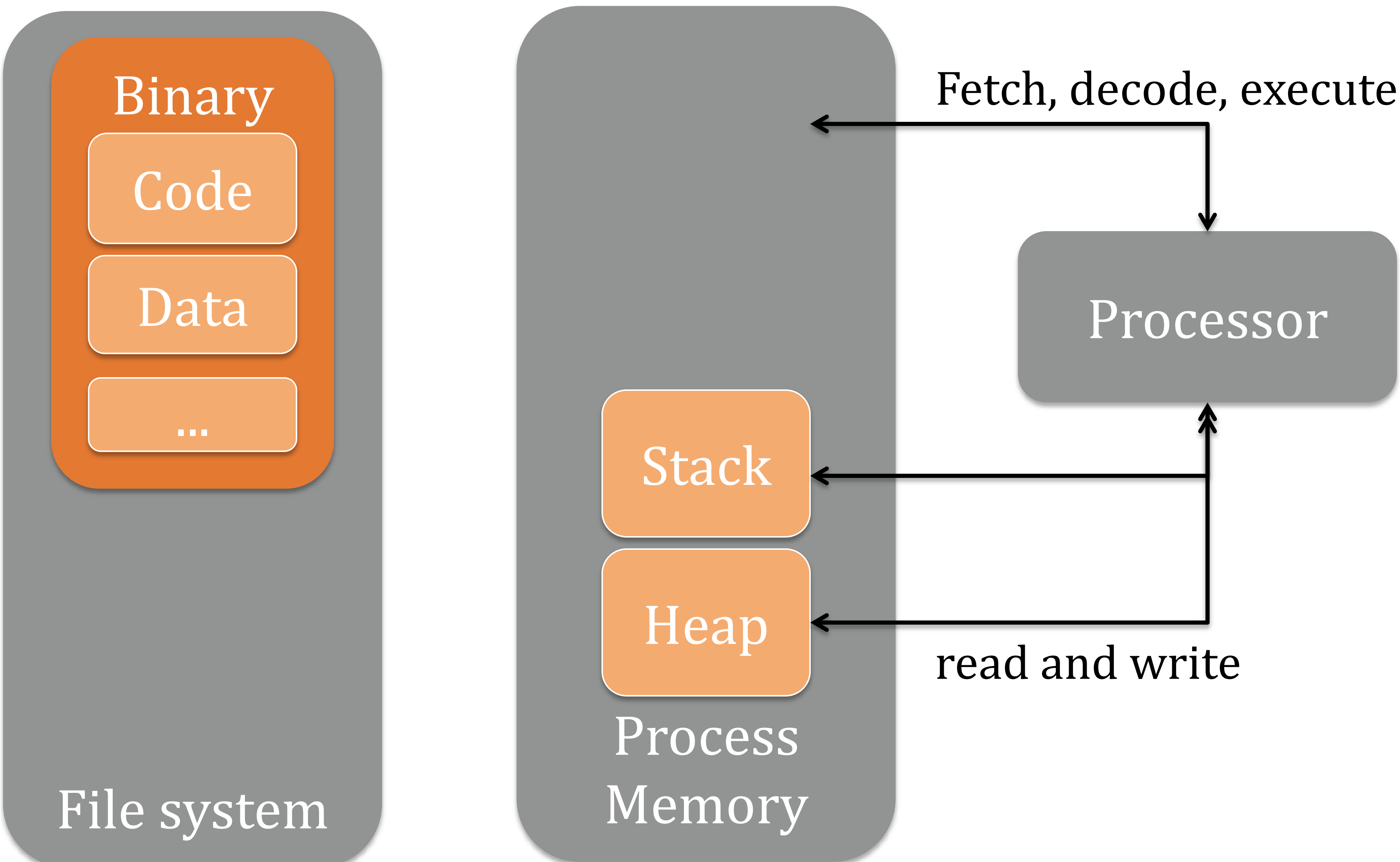
Final executable consists of several *segments*

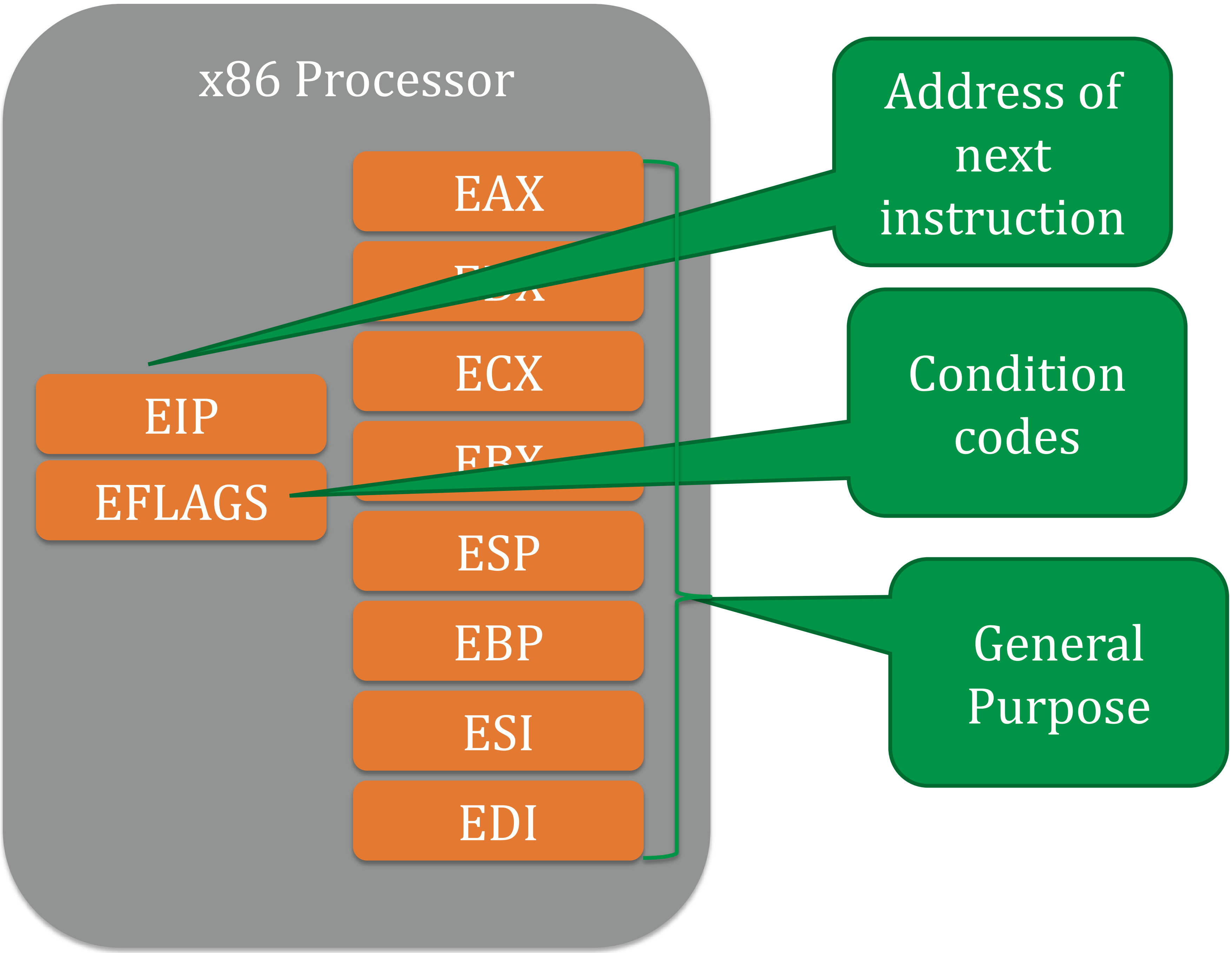
- Text for code written
- Read-only data for constants such as “hello world” and globals
- ...

$ readelf –S <file>

Basic Execution Model

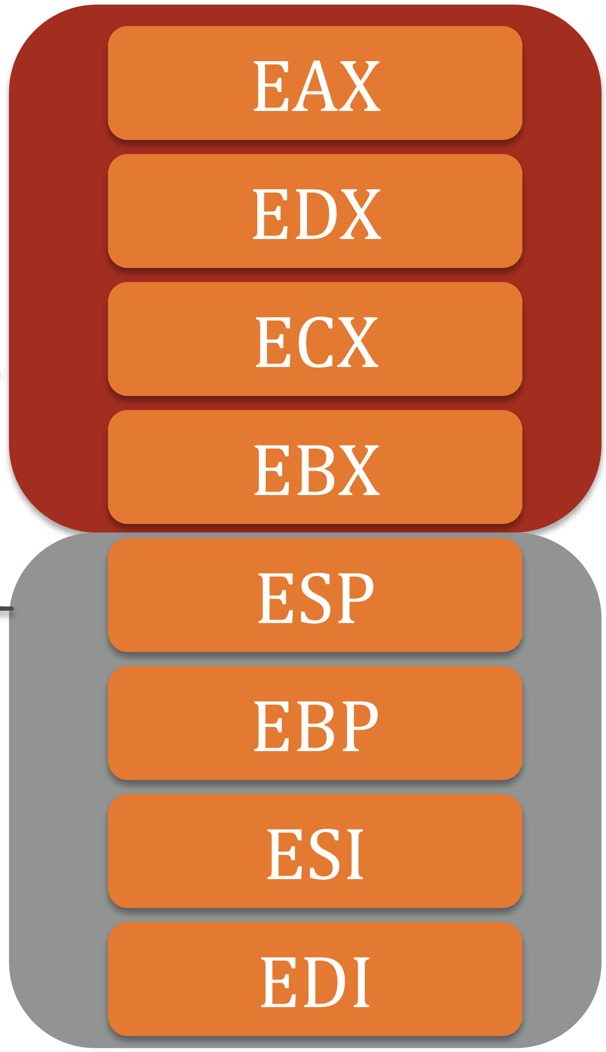
Basic Execution



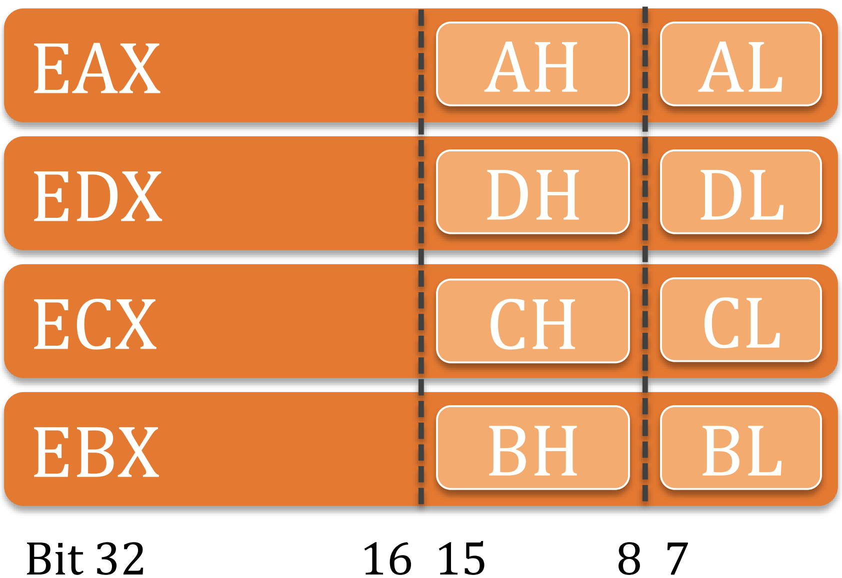

x86 Processor
EAX
ECX
ESP
EBP
ESI
EDI
EIP
EFLAGS
Address of next instruction
Condition codes
General Purpose

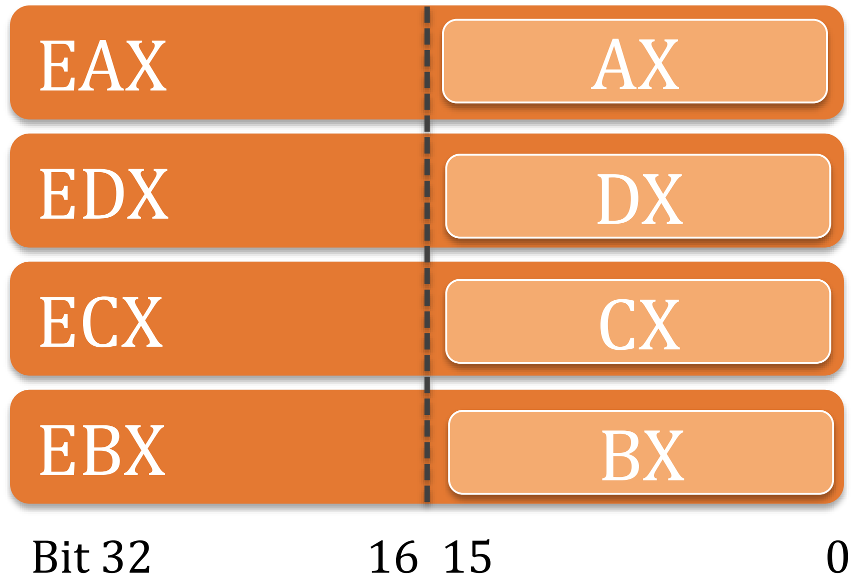
Registers have up to 4 addressing modes

1. Lower 8 bits
2. Mid 8 bits
3. Lower 16 bits
4. Full register



EAX, EDX, ECX, and EBX

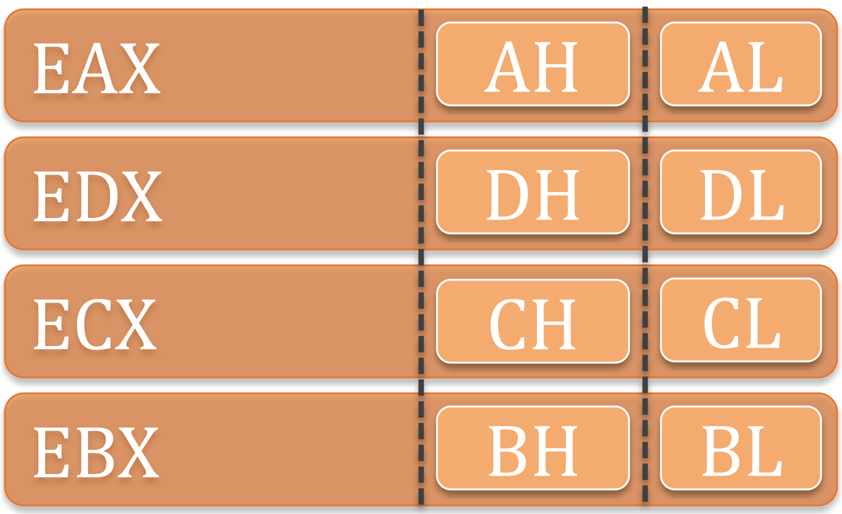


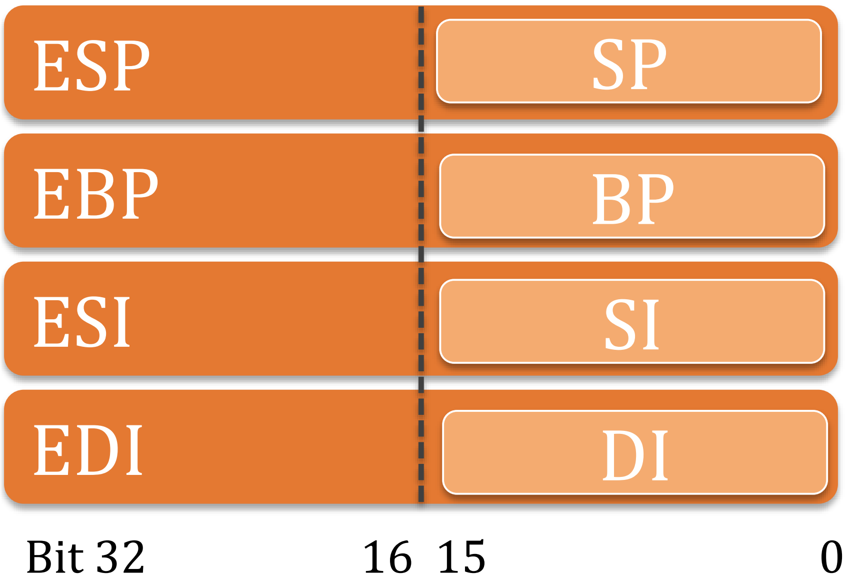


- 32 bit registers (three letters)
- Lower bits (bits 0-7) (two letters with L suffix)
- Mid-bits (bits 8-15) (two letters with H suffix)

- Lower 16 bits (bits 0-15) (2 letters with X suffix)

ESP, EBP, ESI, and EDI





- Lower 16 bits (bits 0-15) (2 letters)

Basic Ops and AT&T vs Intel Syntax

source first

destination first

Meaning	**AT&T**	**Intel**
ebx = eax	movl %eax, %ebx	mov ebx, eax
eax = eax + ebx	addl %ebx, %eax	add eax, ebx
ecx = ecx << 2	shl $2, %ecx	shl ecx, 2

- AT&T is <u>at odds</u> with assignment order. It is the default for objdump, and traditionally used for UNIX.
- Intel order <u>*mirrors*</u> assignment. Windows traditionally uses Intel, as is available via the objdump '-M intel' command line option

Memory Operations

x86: Byte Addressable

It's convention: lower address at the bottom

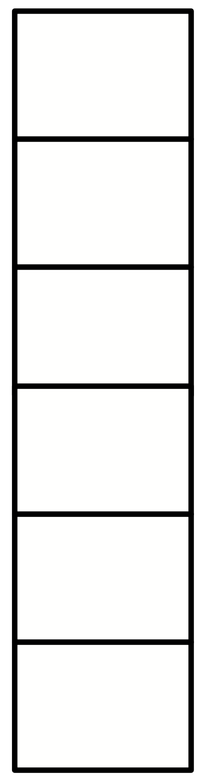
I can fetch bytes at any address

Address 3 holds 1 byte

Address 2 holds 1 byte

Address 1 holds 1 byte

Address 0 holds 1 byte



Memory is just like using an array!

Alternative: **Word addressable**

Example: For 32-bit word size, it's valid to fetch 4 bytes from Mem[0], but not Mem[6] since 6 is not a multiple of 4.

x86: Addressing bytes

Addresses are indicated by operands that have a bracket "[]" or paren "()", for Intel vs. AT&T, resp.

What does mov dl, [al] do?

Moves 0xcc into dl

Register	Value
eax	0x3
edx	0x0
ebx	0x5

	Addr
0xff	6
0xee	
0xdd	
0xcc	
0xbb	
0xaa	
0x00	0

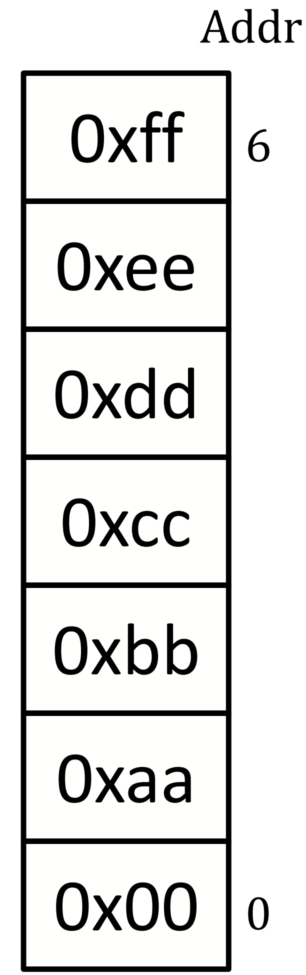
x86: Addressing bytes

Addresses are indicated by operands that have a bracket "[]" or paren "()", for Intel vs. AT&T, resp.

What does
mov edx , [eax]
do?

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Which 4 bytes get moved, and which is the LSB in edx?

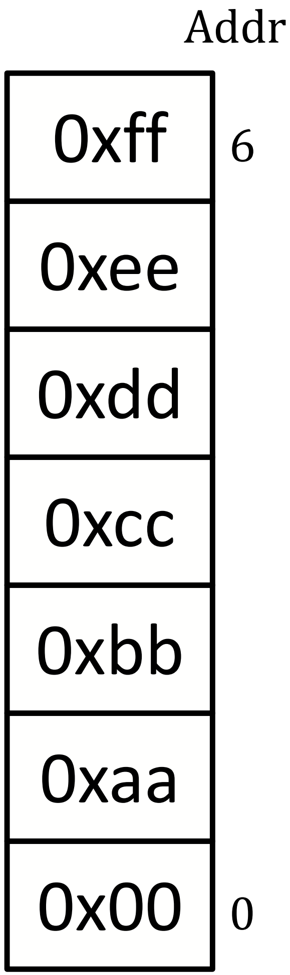


Endianess

- *Endianess*: Order of individually addressable units
- *Little Endian*: Least significant byte first

so address *a* goes in littlest byte (e.g., AL), *a+1* in the next (e.g., AH), etc.

Register	Value
eax	0x3
edx	0xcc
ebx	0x5



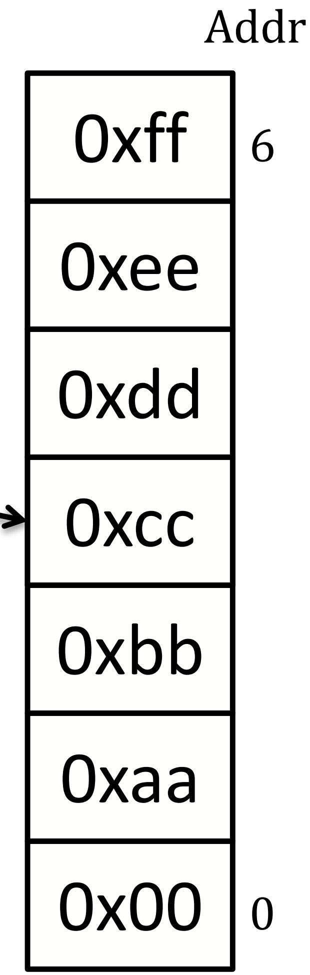
mov edx, [eax]

EDX

Register	Value
eax	
edx	
ebx	

Bit 0

EDX = 0xffeeddcc!



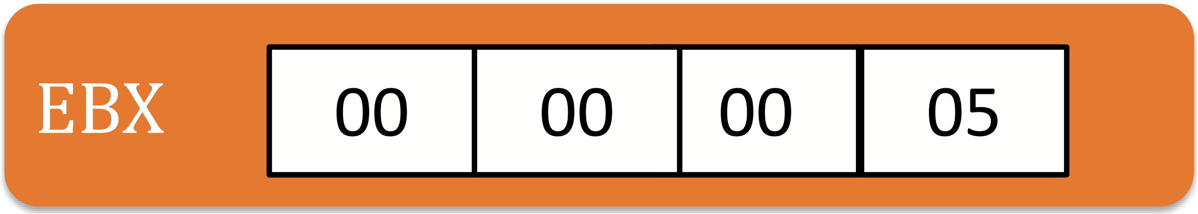
Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

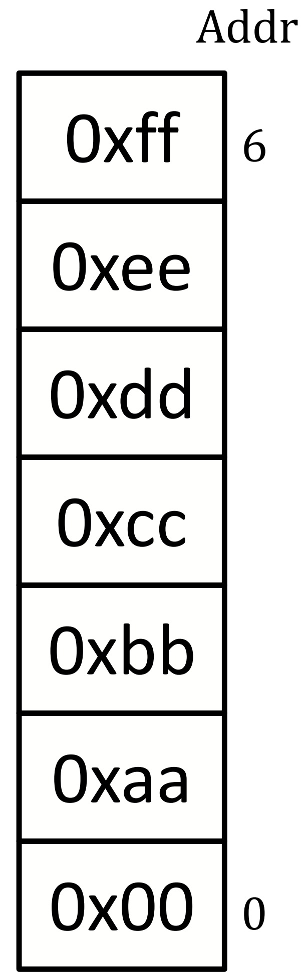
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov [eax], ebx



Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0



Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

There are other ways to address memory than just [register].

These are called *Addressing Modes.*

An ***Addressing Mode*** specifies how to calculate the effective memory address of an operand by using information from registers and constants contained with the instruction or elsewhere.

Motivation: Addressing Buffers

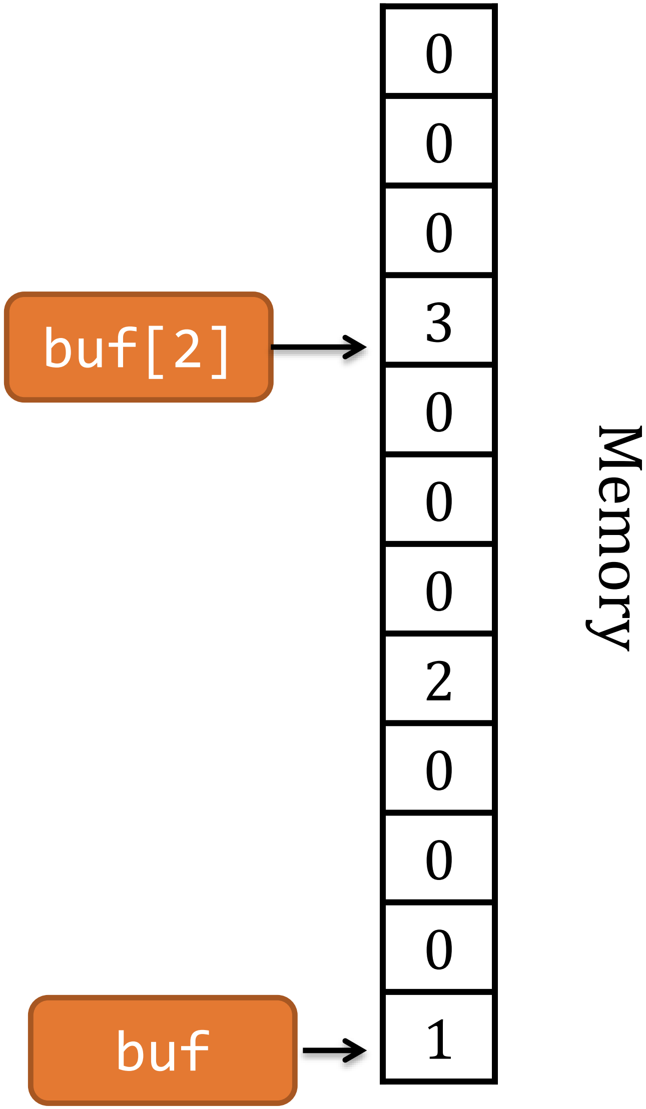
```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

Motivation: Addressing Buffers

```
typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);
```

What is x? what memory cell does it ref?



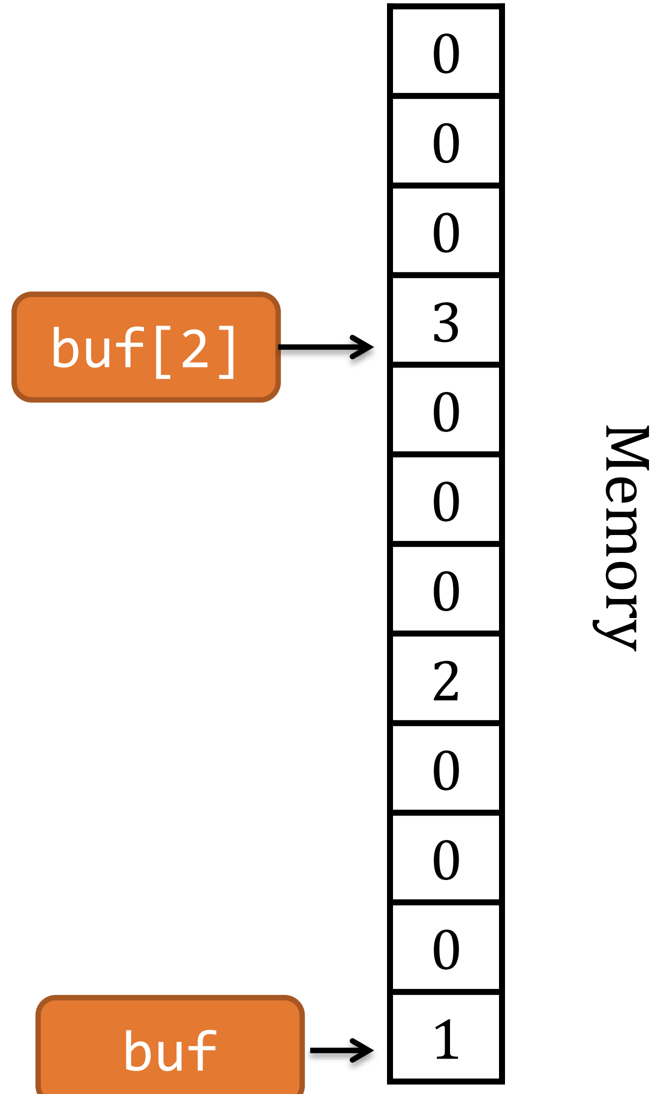
Motivation: Addressing Buffers

```
typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);
y = *( (uint32_t *) (ptr+8));
```

Equivalent

(addr_t) (ptr + 8) = (uint32_t *) buf+2



Motivation: Addressing Buffers

```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

Say at imm $+r_1$

Constant *scaling* factor s, typically 1, 2, 4, or 8

Say in Register r_2

$$imm + r_1 + s*r_2$$

AT&T: imm (r_1, r_2, s)

Intel: $r_1 + r_2*s + imm$

AT&T Addressing Modes for Common Codes

Form	Meaning on memory M
imm (r)	M[r + imm]
imm (r_1, r_2)	M[r_1 + r_2 + imm]
imm (r_1, r_2, s)	M[r_1 + r_2*s + imm]
imm	M[imm]

Referencing Memory

Loading a value from memory: mov

```
<eax> = *buf;
```

```
mov  -0x38(%ebp),%eax (I)
mov  eax, [ebp-0x38] (A)
```

Loading an address: lea

```
<eax> = buf;
```

```
lea  -0x38(%ebp),%eax (I)
lea  eax, [ebp-0x38] (A)
```

Suppose I want to access address 0xdeadbeef directly

Loads the address

```
lea eax, 0xdeadbeef (I)
```

Deref the address

```
mov  eax, 0xdeadbeef (I)
```

Note missing $. This distinguishes the address from the value

Control Flow

Assembly is “Spaghetti Code”

Nice C Abstractions

- if-then-else
- while
- for loops
- do-while

Assembly

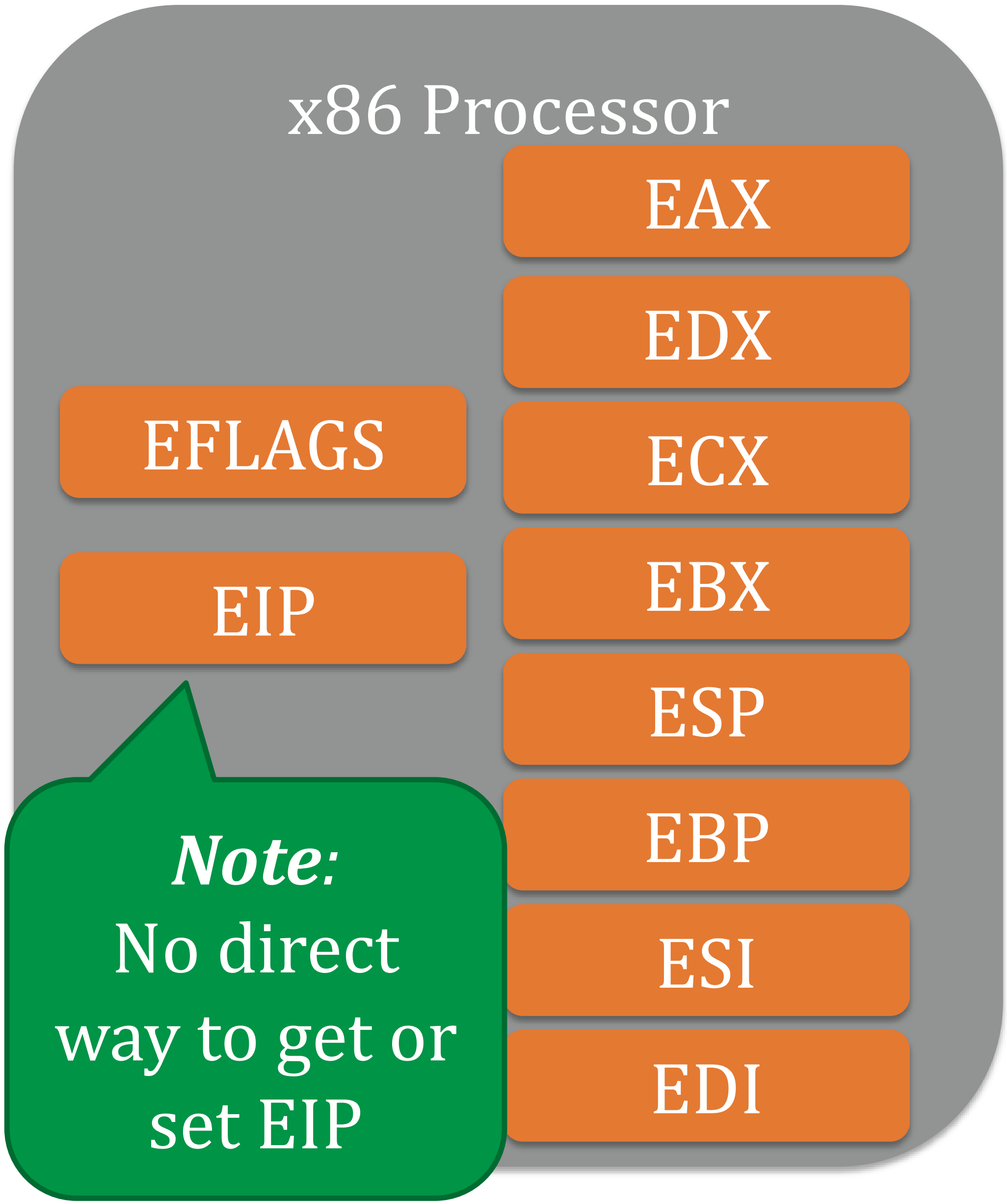
- Jump
  - Direct: jmp addr
  - Indirect: jmp reg
- Branch
  - Test EFLAG
  - if(EFLAG SET) goto line

Jumps

- jmp 0x45, called a ***direct jump***
- jmp *eax , called an ***indirect jump***

Branches

- `if (EFLAG) jmp x`
  Use one of the 32 EFLAG bits to determine if jump taken



Implementing “if”

C

```
1. if(x <= y)
2.    z = x;
3. else
4.    z = y;
```

Psuedo-Assembly

1. Computing x – y. Set eflags:
   1. CF =1 if x < y
   2. ZF =1 if x==y
2. Test EFLAGS. If both CF and ZF **not** set, branch to E
3. mov x, z

Jump to F

mov y, z

<end of if-then-else>

Assembly is 2 instrs

1. Set eflag to conditional
2. Test eflag and branch

If (x > y)

%eax holds x and 0xc(%ebp) holds y

```
cmp 0xc(%ebp), %eax
ja addr
```

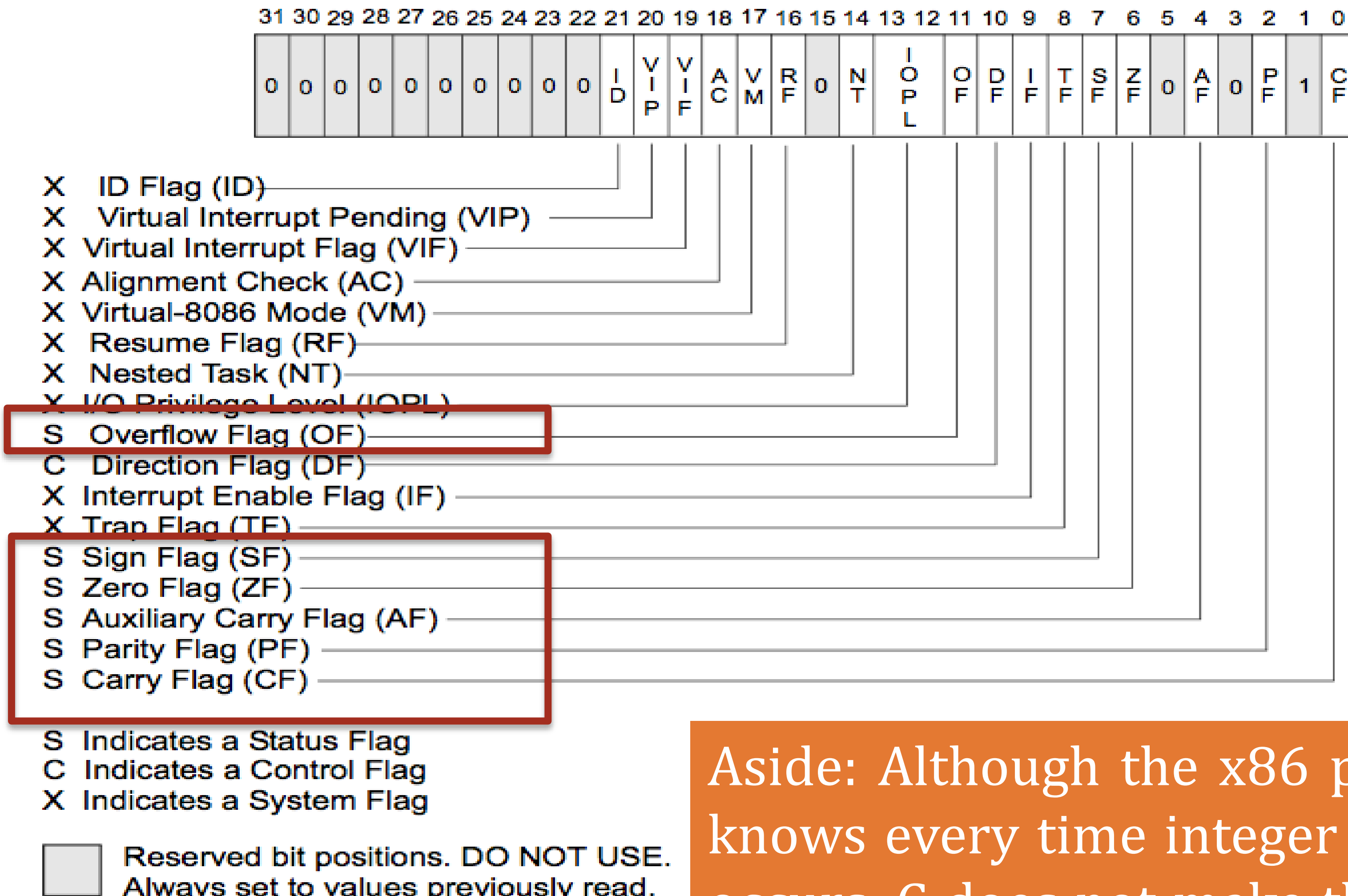
Same as "sub" instruction
r = %eax - M[%ebp+0xc], i.e., x – y

Jump if CF=0 and ZF=0

$(x >= y) \wedge (x \,!= y) \Rightarrow x > y$

Setting EFLAGS

- Instructions may set an eflag, e.g.,
- "cmp" and arithmetic instructions most common
  - Was there a carry (CF Flag set)
  - Was the result zero (ZF Flag set)
  - What was the parity of the result (PF flag)
  - Did overflow occur (OF Flag)
  - Is the result signed (SF Flag)



Aside: Although the x86 processor knows every time integer overflow occurs, C does not make this result visible.

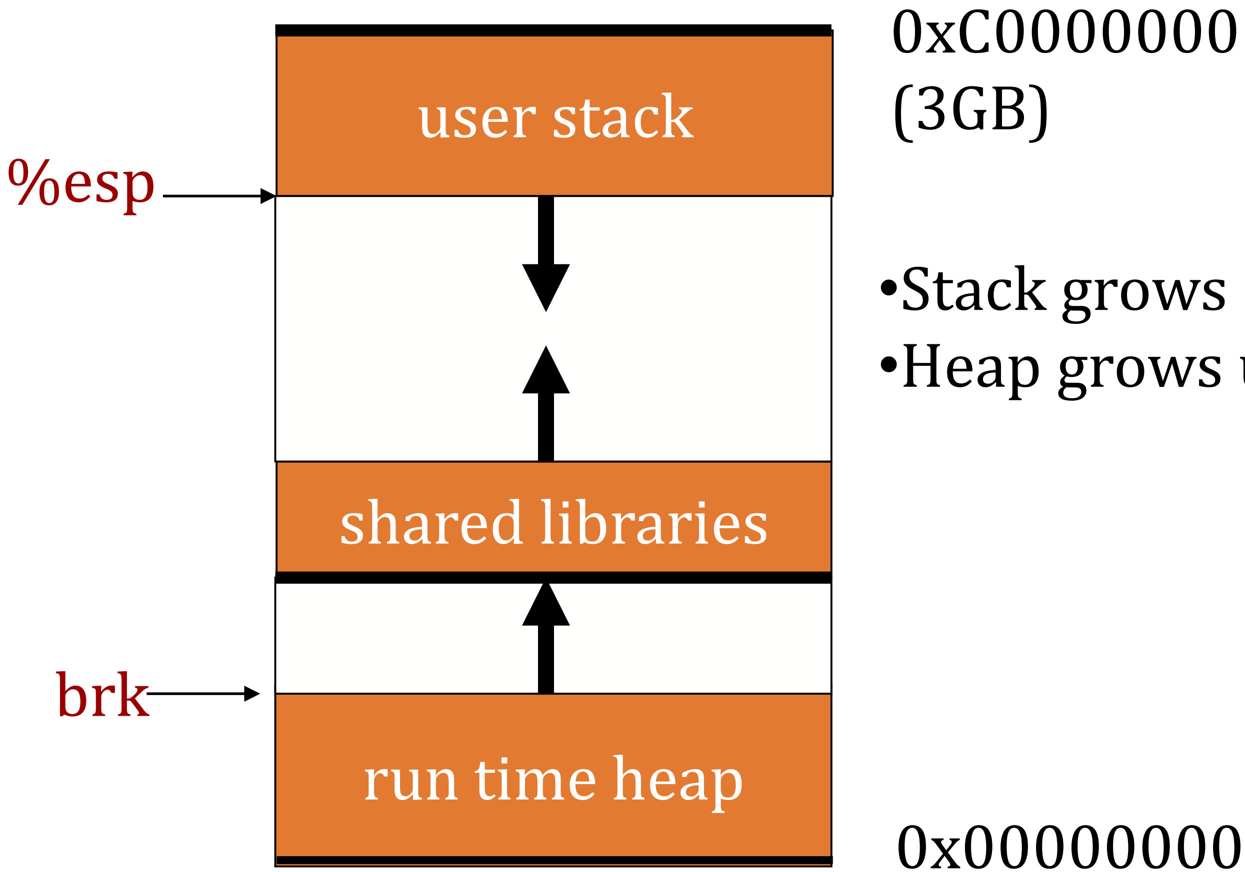
From the Intel x86 manual

See the x86 manuals available on Intel's website for more information

Instr.	Description	Condition
JO	Jump if overflow	OF == 1
JNO	Jump if not overflow	OF == 0
JS	Jump if sign	SF == 1
JZ	Jump if zero	ZF == 1
JE	Jump if equal	ZF == 1
JL	Jump if less than	SF <> OF
JLE	Jump if less than or equal	ZF ==1 or SF <> OF
JB	Jump if below	CF == 1
JP	Jump if parity	PF == 1

Memory Organization

Memory
Program text
Shared libs
Data
...

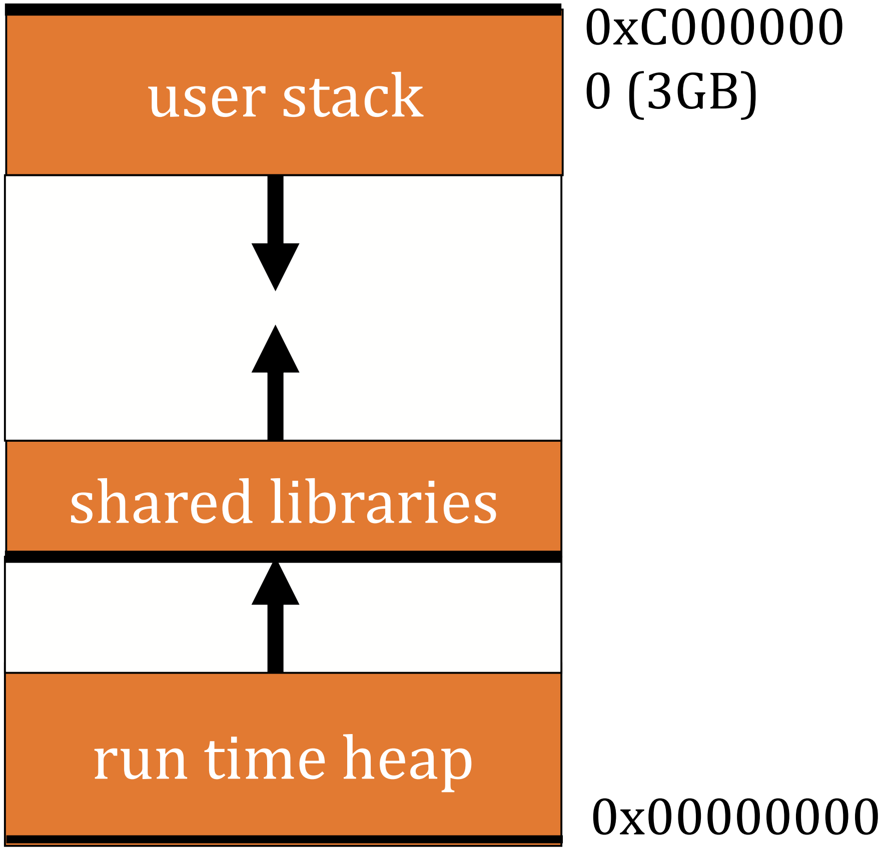


- Stack grows down
- Heap grows up

The Stack grows down towards lower addresses.

Variables

- On the stack
  - Local variables
  - Lifetime: stack frame
- On the heap
  - Dynamically allocated via new/malloc/etc.
  - Lifetime: until freed



Procedures

- Procedures are not native to assembly
- Compilers *implement* procedures
  - On the stack
  - Following the call/return stack discipline

Procedures/Functions

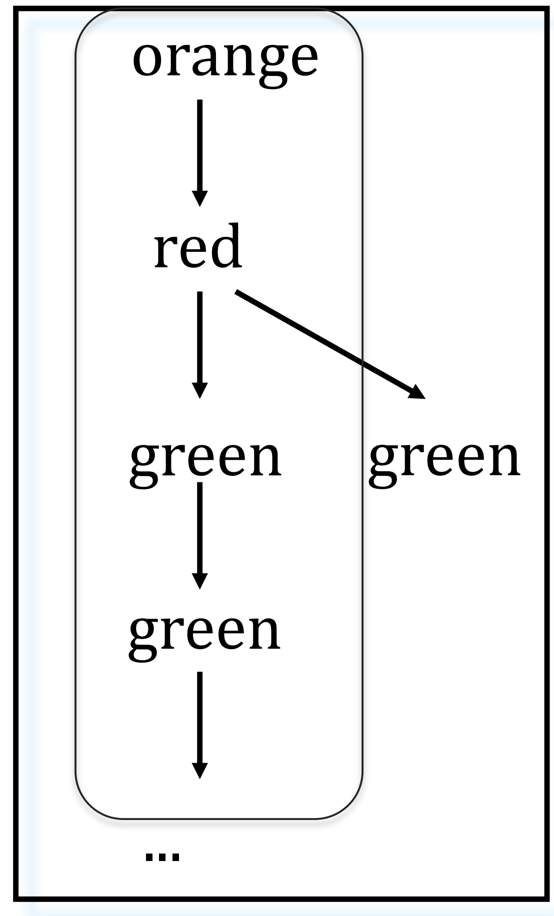
- We need to address several issues:
  1. How to allocate space for local variables
  2. How to pass parameters
  3. How to pass return values
  4. How to share 8 registers with an infinite number of local variables
- A stack frame provides space for these values
  - Each procedure invocation has its own stack frame
  - Stack discipline is LIFO
    - If procedure A calls B, B's frame must exit before A's

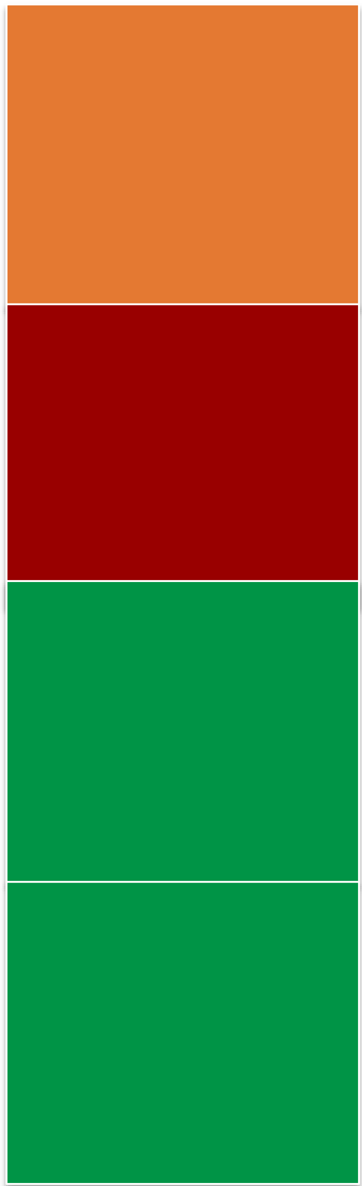
```
orange(...)
{
    ...
    red()
    ...
}
```

```
red(...)
{
    ...
    green()
    ...
    green()
}
```

```
green(...)
{
    ...
    green()
    ...
}
```

Function Call Chain





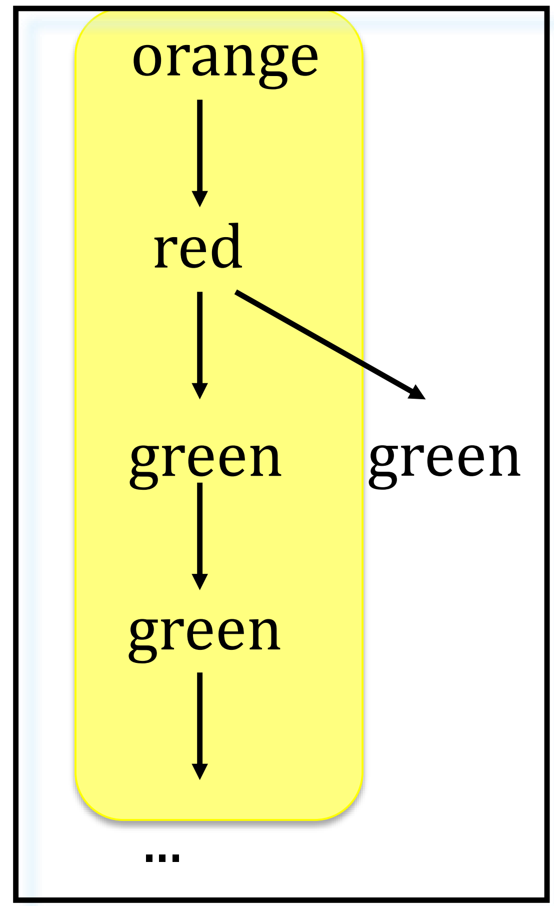
Frame for

- locals
- pushing parameters
- temporary space

Call to red "***pushes***" new frame

When green returns it "***pops***" its frame

Function Call Chain



On the stack

```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
     c = a;
  else
     c = b;
  d = red(c, buf);
  return d;
}
```

Need to access arguments

Need space to store local vars (buf, c, and d)

Need space to put arguments for callee

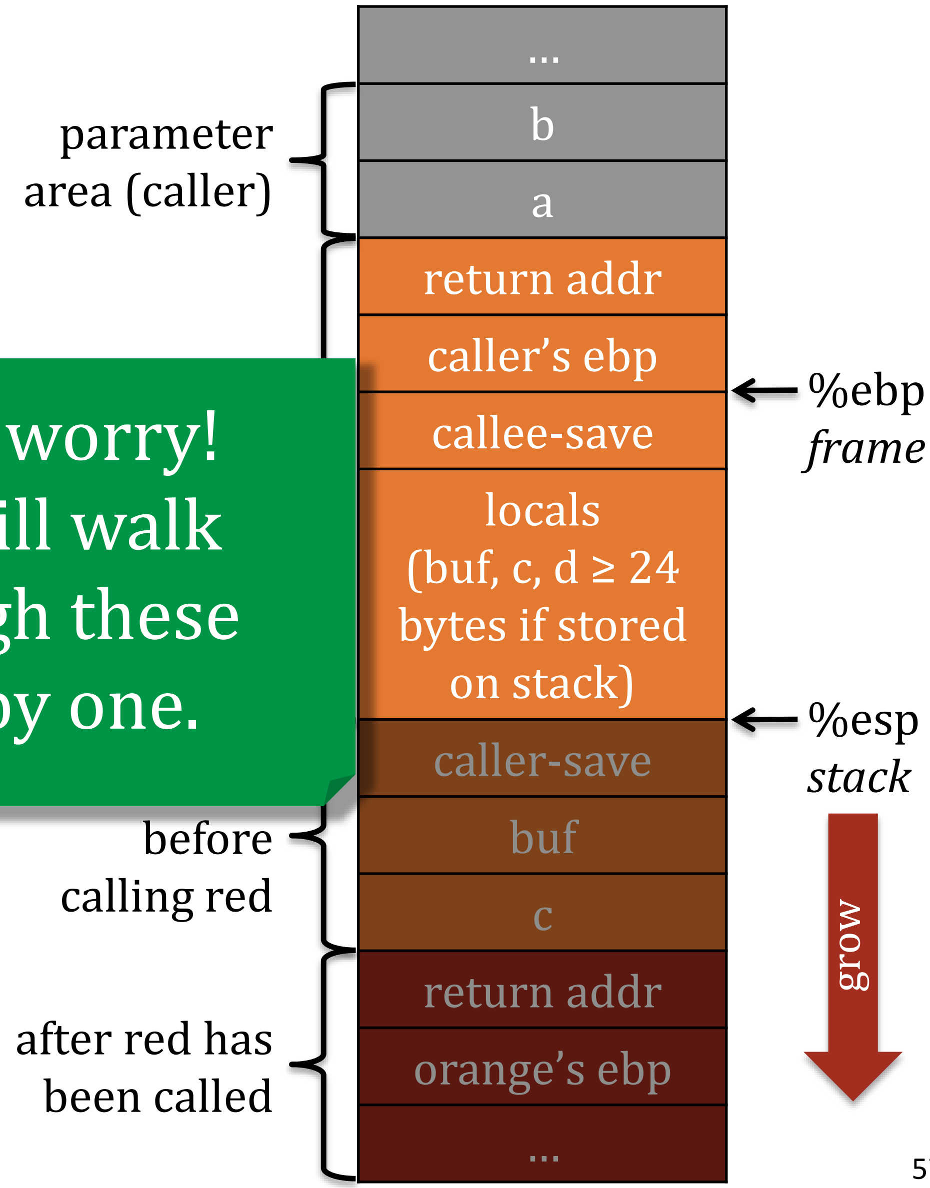
Need a way for callee to return values

Calling convention determines the above features

cdecl – the default for Linux & gcc

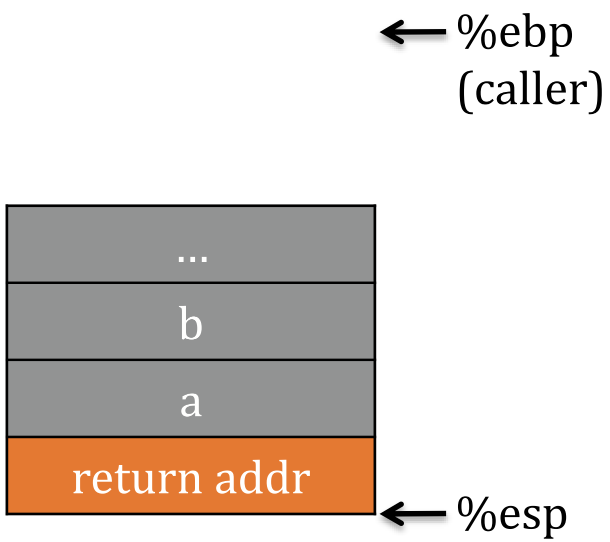
```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
     c = a;
  else
     c = b;
  d = red(c, buf);
  return d;
}
```

Don't worry!
We will walk
through these
one by one.



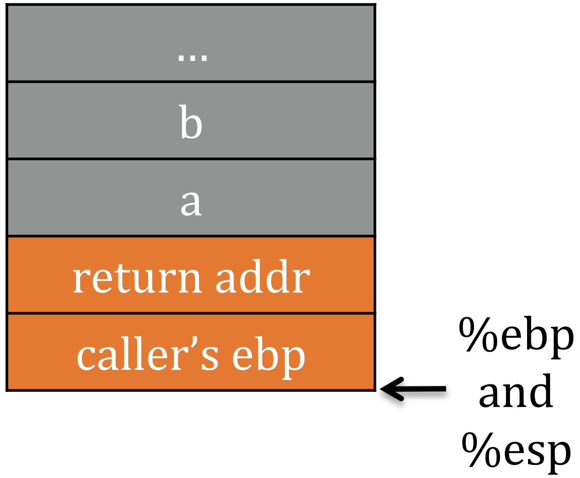
When orange attains control,

1. return address has already been pushed onto stack by caller



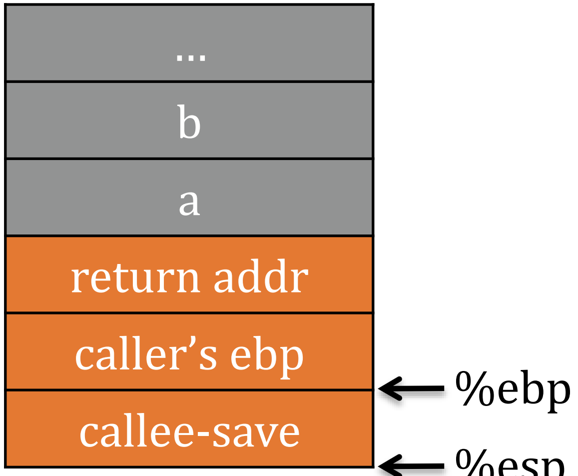
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8



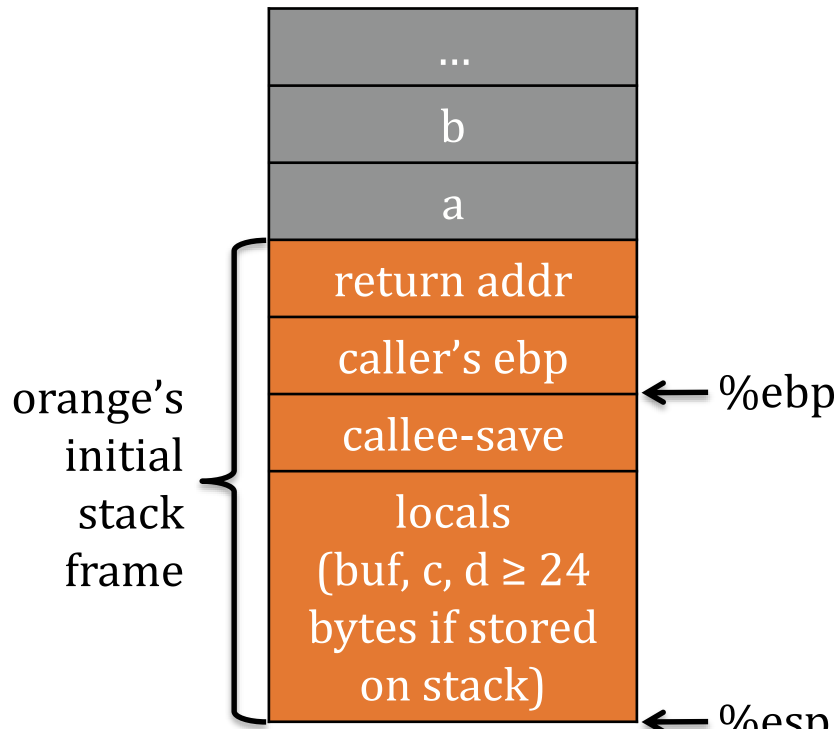
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic

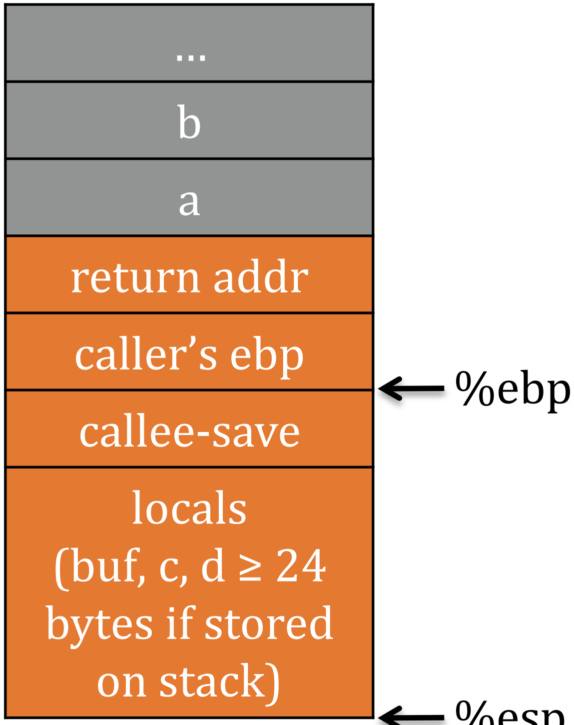


When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
4. allocate space for locals
   - subtracting from esp
   - "live" variables in registers, which on contention, can be "***spilled***" to stack space

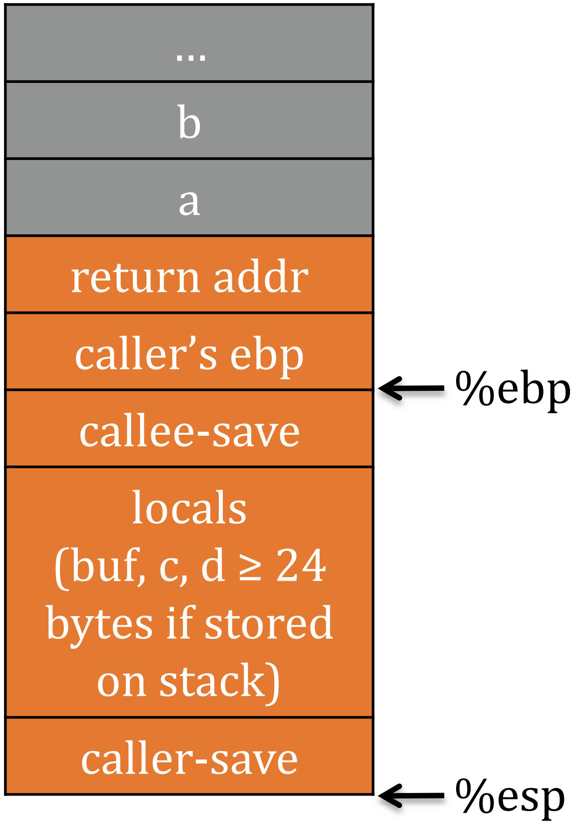


For *caller* orange to call *callee* red,



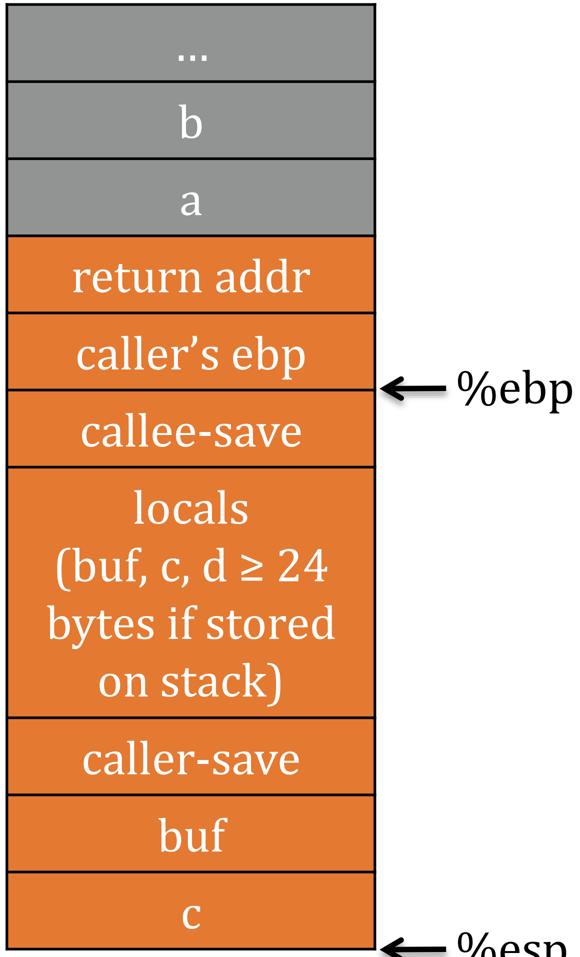
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx



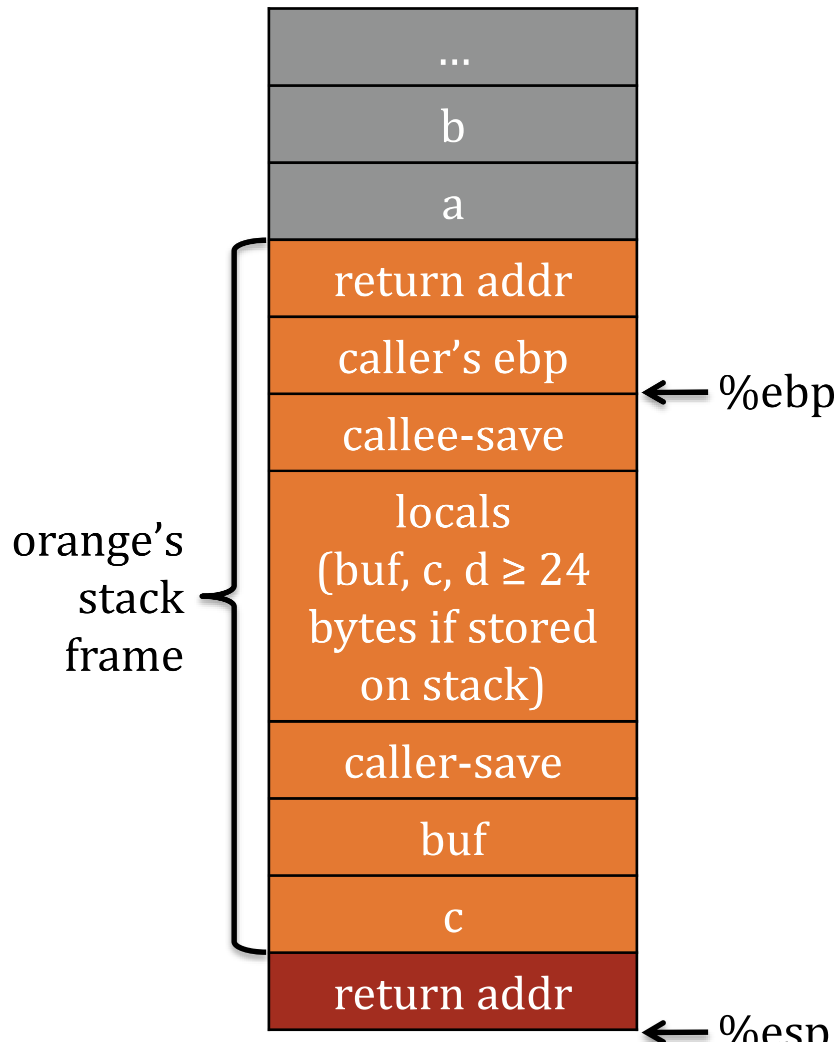
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack



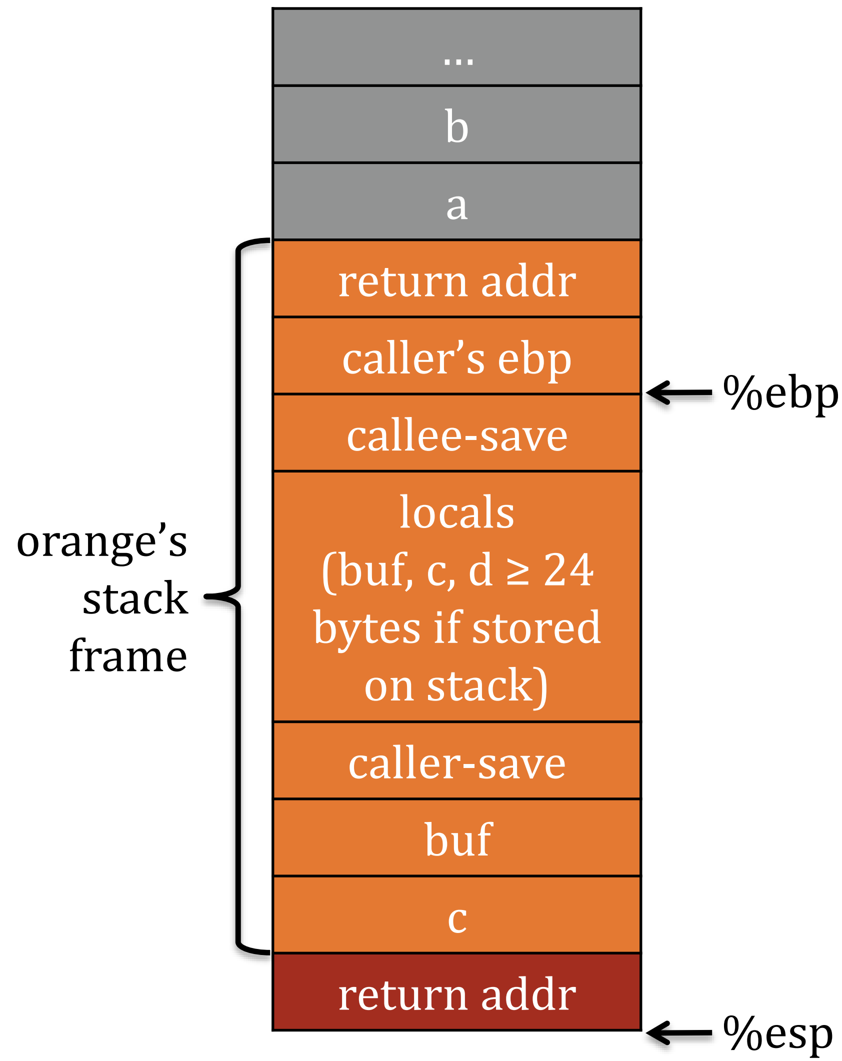
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in orange after red returns



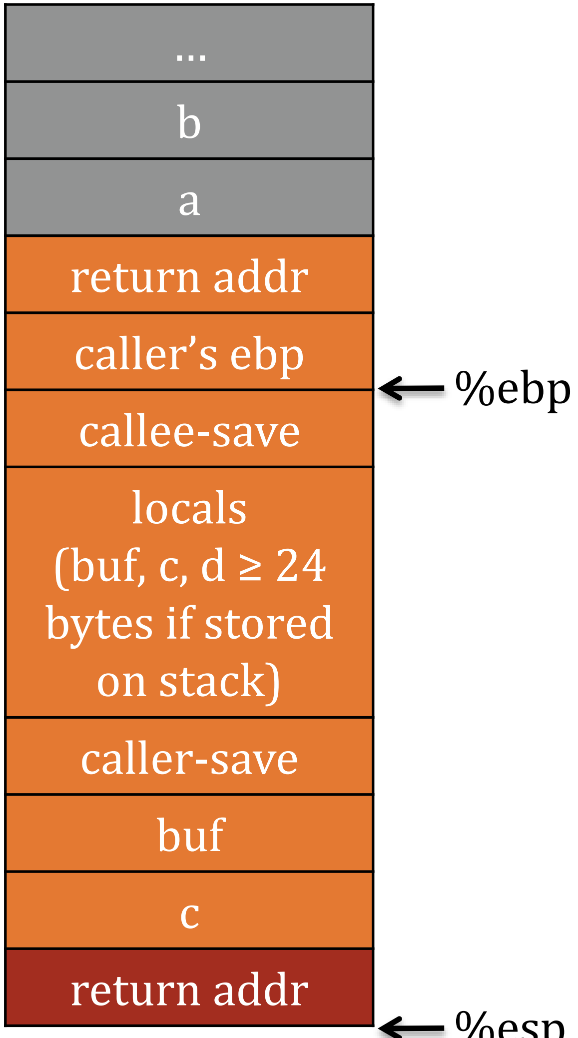
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in orange after red returns
4. transfer control to red
   - usually happens together with step 3 using `call`



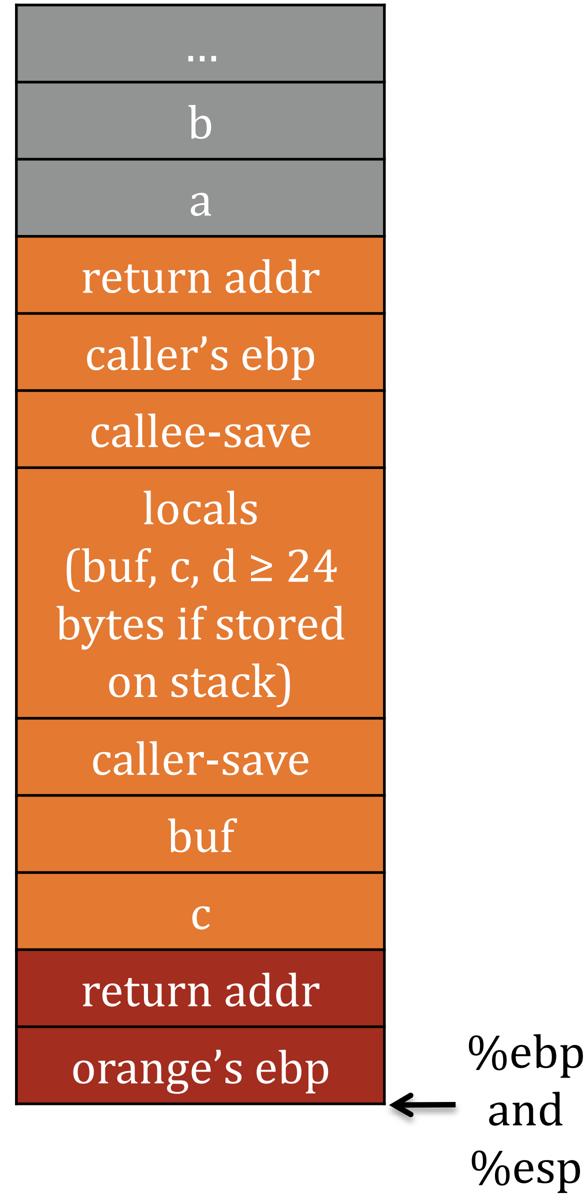
When red attains control,

1. return address has already been pushed onto stack by orange



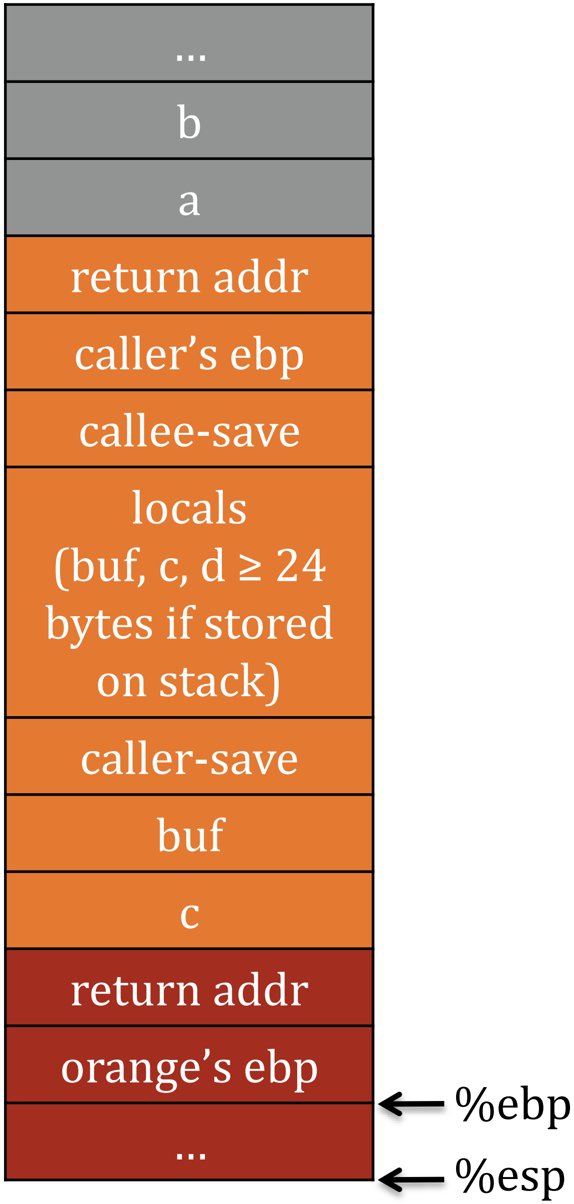
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer



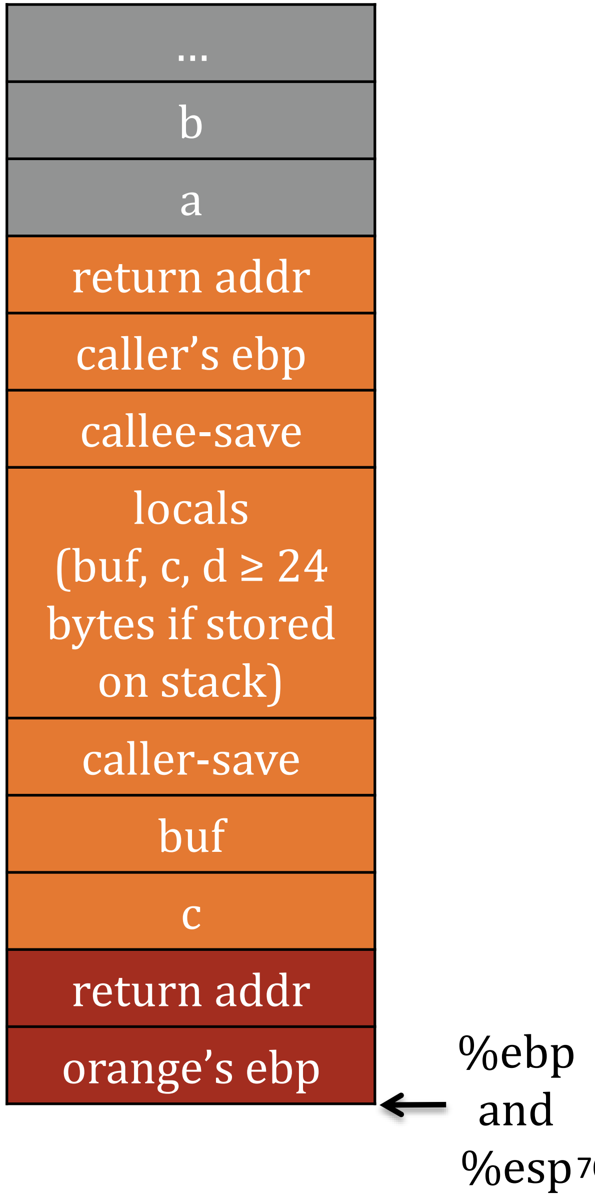
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...



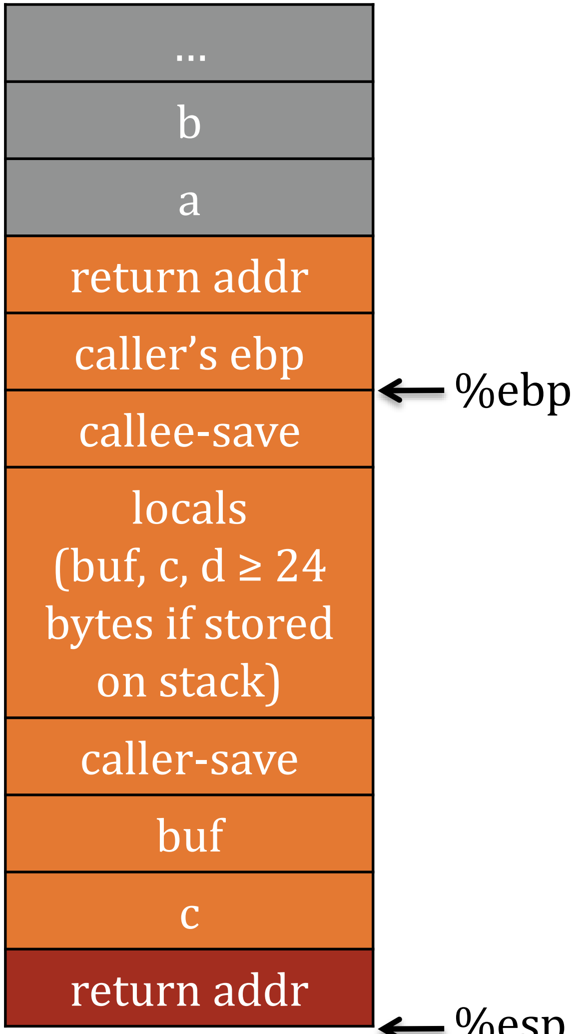
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers



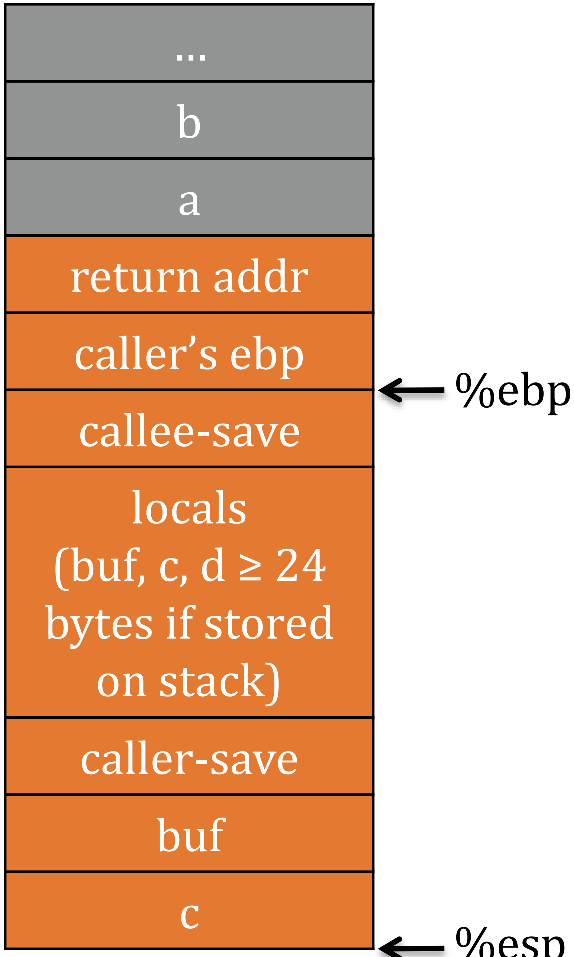
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange's frame pointer
   - `pop %ebp`

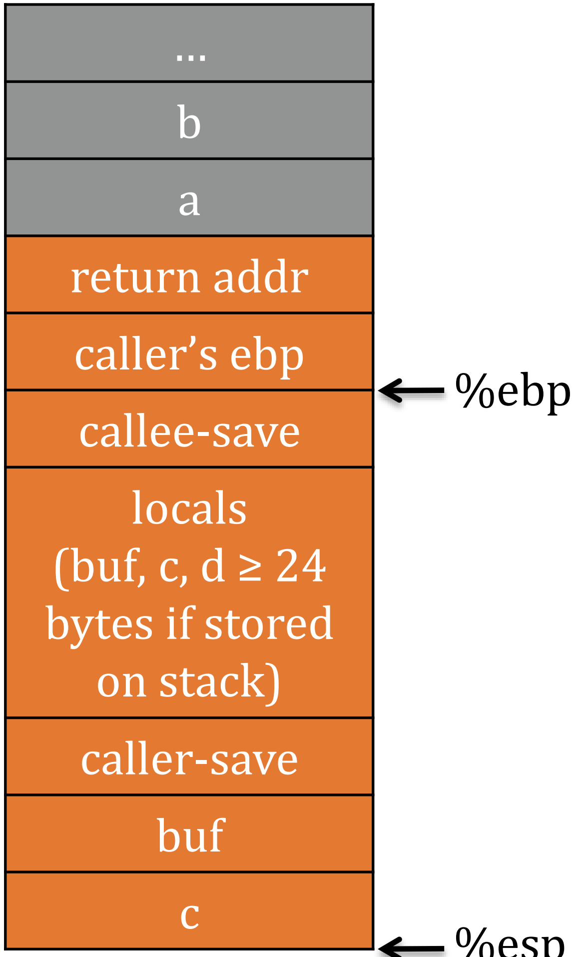


When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange's frame pointer
   - `pop %ebp`
8. return control to orange
   - `ret`
   - pops return address from stack and jumps there

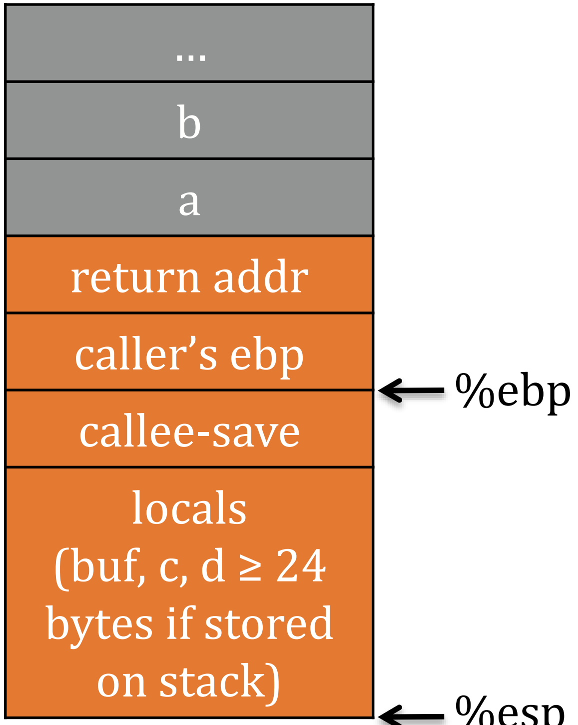


When orange regains control,



When orange regains control,

1. clean up arguments to red
   - adding to esp
2. restore any caller-save registers
   - pops
3. ...



Terminology

- ***Function Prologue*** – instructions to set up stack space and save callee saved registers
  - Typical sequence:
    push ebp
    ebp = esp
    esp = esp - <frame space>
- ***Function Epilogue***- instructions to clean up stack space and restore callee saved registers
  - Typical Sequence:
    leave // esp = ebp, pop ebp
    ret // pop and jump to ret addr

cdecl – One Convention

Action	Notes
caller saves: eax, edx, ecx	push (old), or mov if esp already adjusted
arguments pushed right-to-left	
linkage data starts new frame	call pushes return addr
callee saves: ebx, esi, edi, ebp, esp	ebp often used to deref args and local vars
return value	pass back using eax
argument cleanup	caller's responsibility

Q&A

- Why do we need calling conventions?

- Does the callee always have to save callee-saved registers?

- How do you think varargs works (va_start, va_arg, etc)?

```
void myprintf(const char *fmt, ...){}
```

Today's Key Concepts

- Compiler workflow
- Register to register moves
  - Register mnemonics
- Register/memory
  - mov and addressing modes for common codes
- Control flow
  - EFLAGS
- Program Memory Organization
  - Stack grows down
- Functions
  - Pass arguments, callee and caller saved, stack frame

For more information

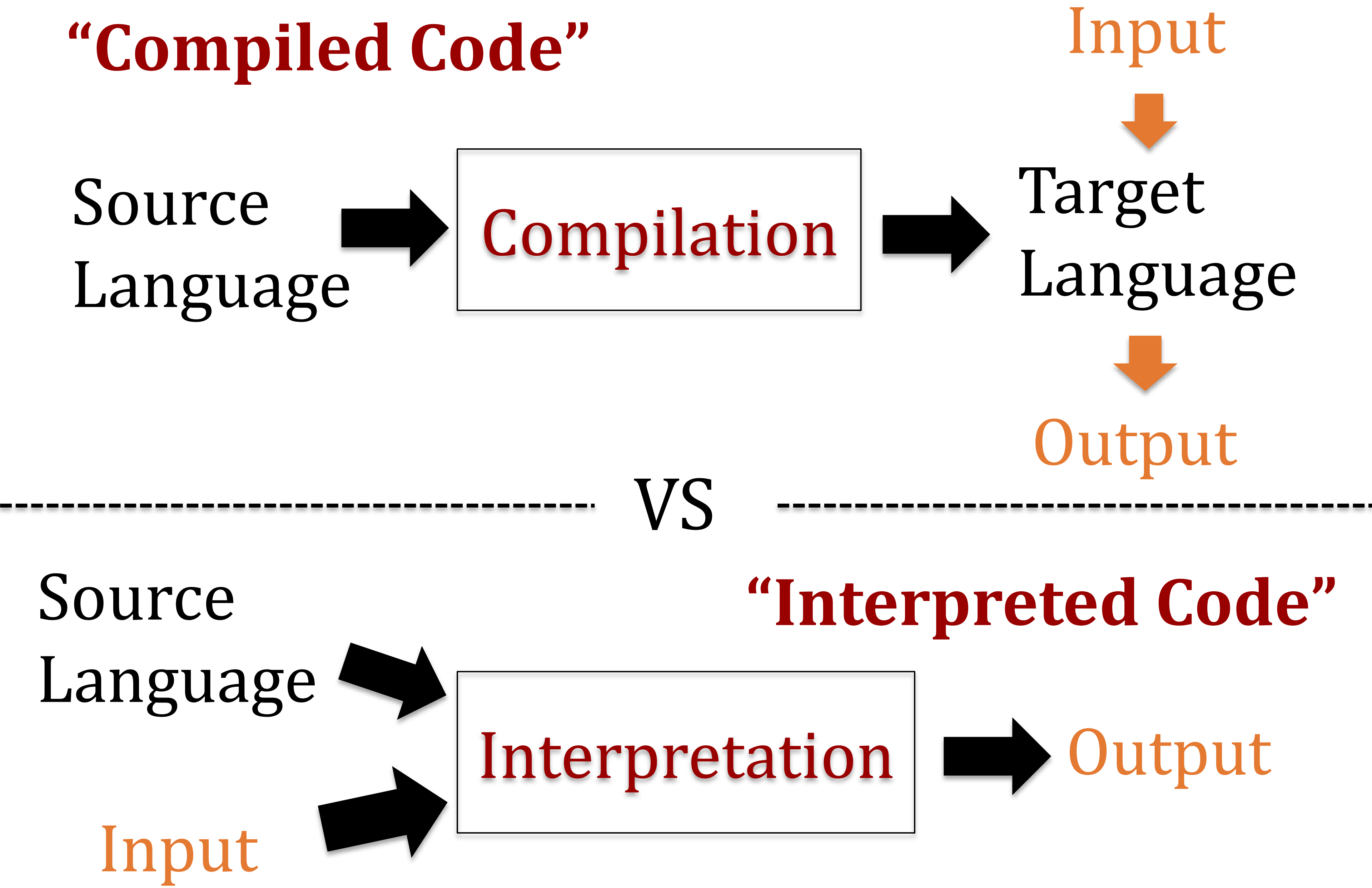
- Overall machine model:
  Computer Systems, a Programmer's Perspective
  by Bryant and O'Hallaron
- Calling Conventions:
  - http://en.wikipedia.org/wiki/X86_calling_conventions


Questions?

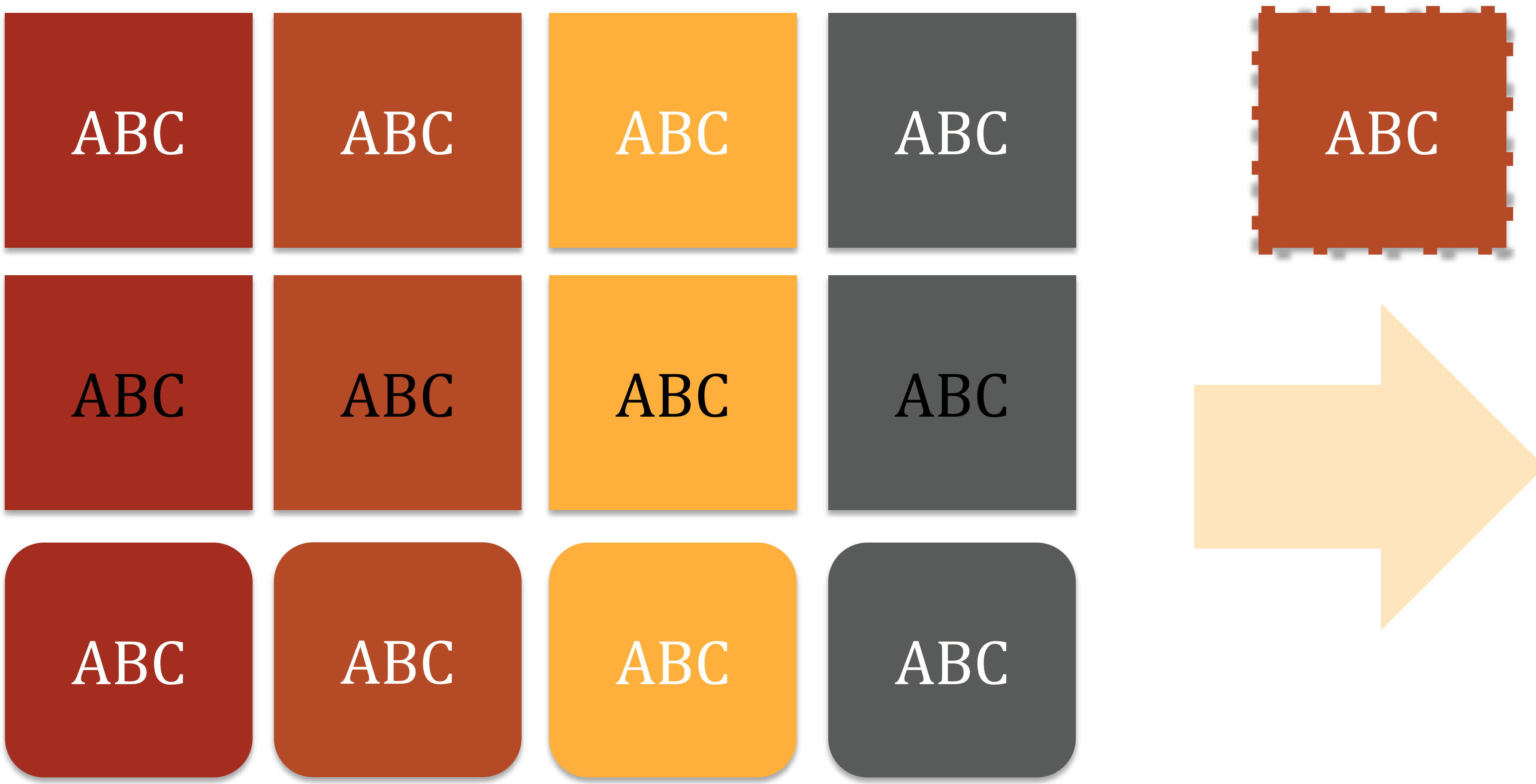
END

Backup slides here.

- Titled cherries because they are for the pickin. (credit due to maverick for wit)


"Compiled Code"
Input
Source Language
Compilation
Target Language
Output
VS
Source Language
"Interpreted Code"
Interpretation
Output
Input

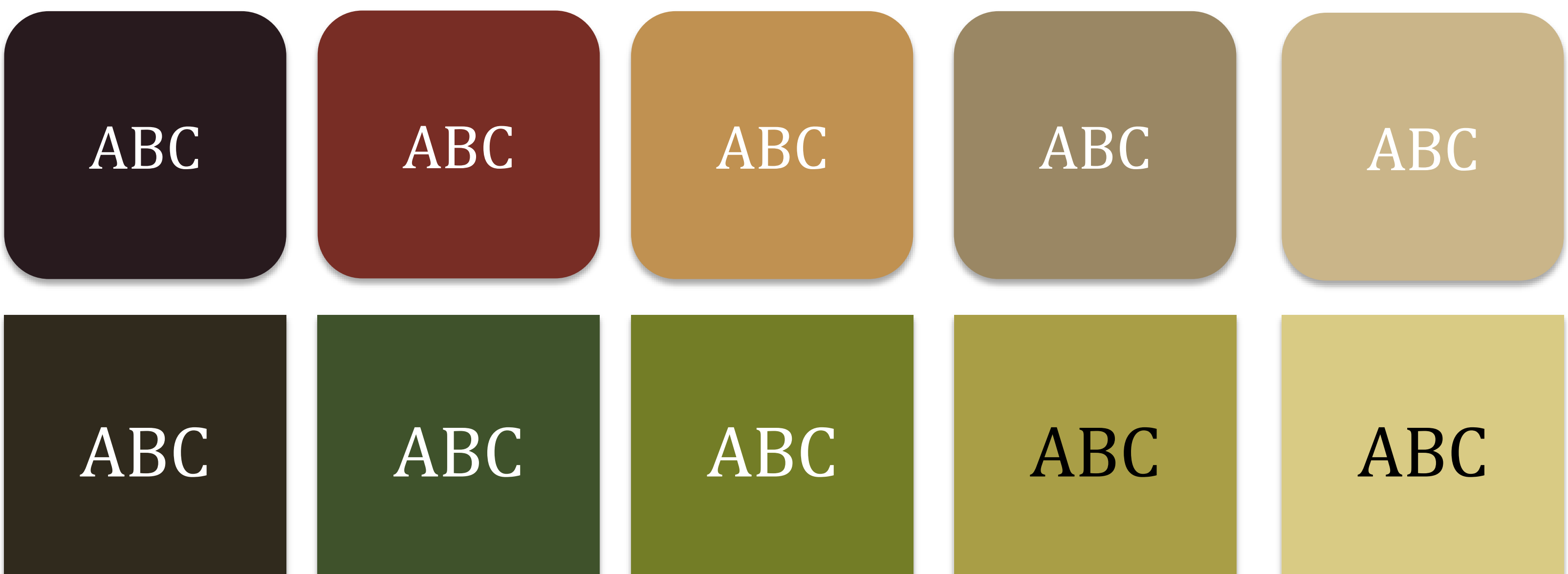
Stencils



Other Colors from Adobe Kuler

Don’t use these unless absolutely necessary.
We are not making skittles, so there is no rainbow of colors necessary.

Mac application for Adobe Kuler:
http://www.lithoglyph.com/mondrianum/
http://kuler.adobe.com/



To answer the question

“Is this program safe?”

We need to know

“What will executing this program do?”

Understanding the compiler and machine semantics are key.