CS165 – Computer Security

Understanding low-level program execution

Oct 5, 2021

Agenda

- Compilation Workflow ✓
- x86 Execution Model
  - Basic Execution ✓
  - Memory Operation ✓
  - Control Flow ← NEXT
  - Memory Organization

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

"For"→ "While"→ "Do-While"

For Version

```
for (Init; Test; Update )

    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

Do-While Version

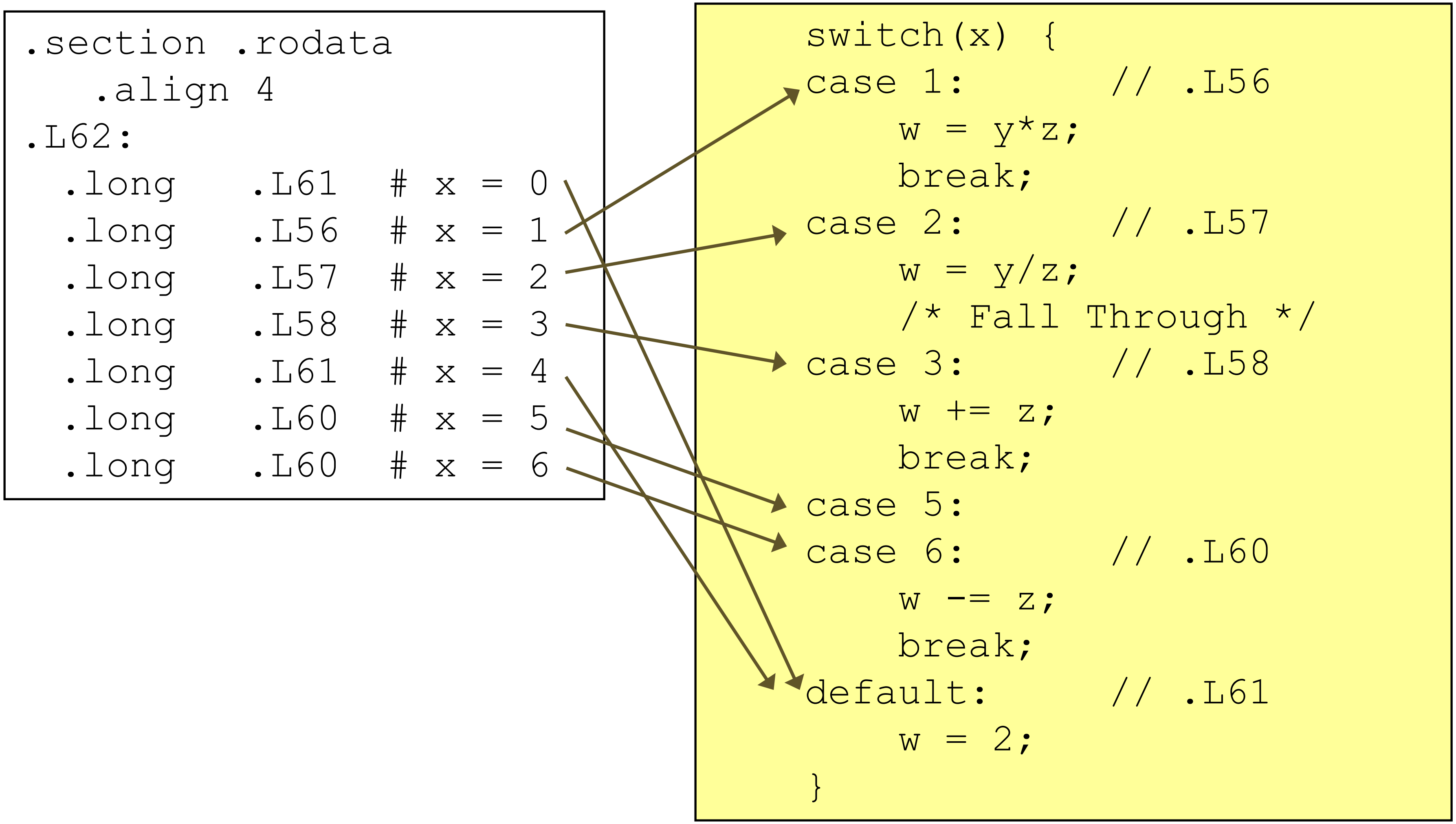
```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version (close to assembly)

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

Jump Table

Table Contents



```
.section .rodata
    .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

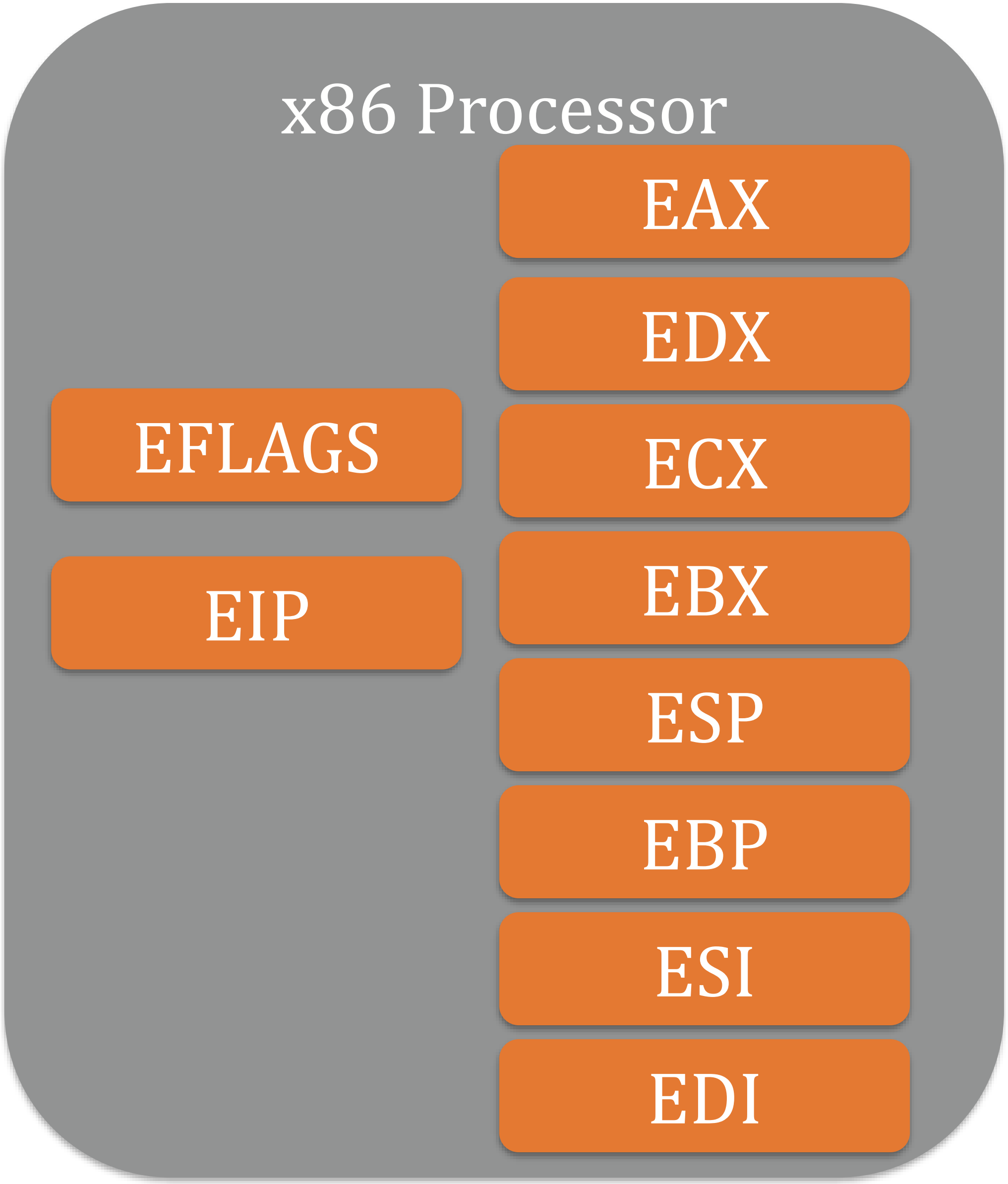
```
switch(x) {
case 1:         // .L56
    w = y*z;
    break;
case 2:         // .L57
    w = y/z;
    /* Fall Through */
case 3:         // .L58
    w += z;
    break;
case 5:
case 6:         // .L60
    w -= z;
    break;
default:        // .L61
    w = 2;
}
```

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

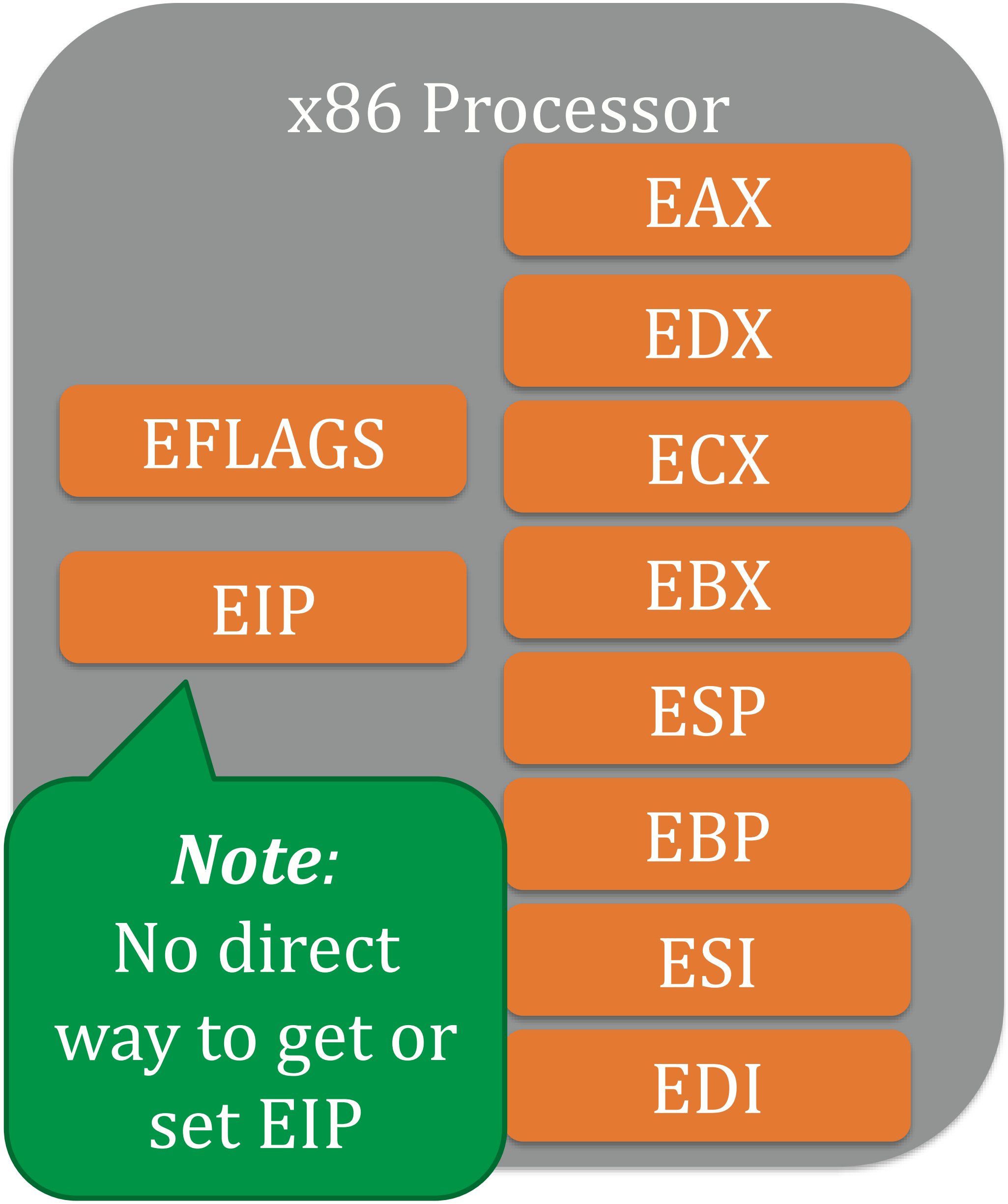


Jumps

- 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



Note:
No direct way to get or set EIP

Implementing "if"

C

```
if(x <= y)
  z = x;
else
  z = y;
```

Assembly is 2 instrs

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

Implementing “if”

C

```
if(x <= y)
    z = x;
else
    z = y;
```

Psuedo-Assembly

1. Computing x – y. Set eflags:
   1. CF =1 if x < y
   2. ZF =1 if x==y

Assembly is 2 instrs

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

Implementing “if”

C

```
if(x <= y)
    z = x;
else
    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 5
3. mov x, z
4. Jump to 6
5. mov y, z
6. <end of if-then-else>

Assembly is 2 instrs

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

If (x > y)

```
cmp 0xc(%ebp), %eax    # x-y
ja addr
```

If (x > y)

```
cmp 0xc(%ebp), %eax    # x-y
ja addr
```

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

If (x > y)

```
cmp 0xc(%ebp), %eax    # x-y
ja addr
```

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

Jump if CF=0 and ZF=0

If (x > y)

```
cmp 0xc(%ebp), %eax    # x-y
ja addr
```

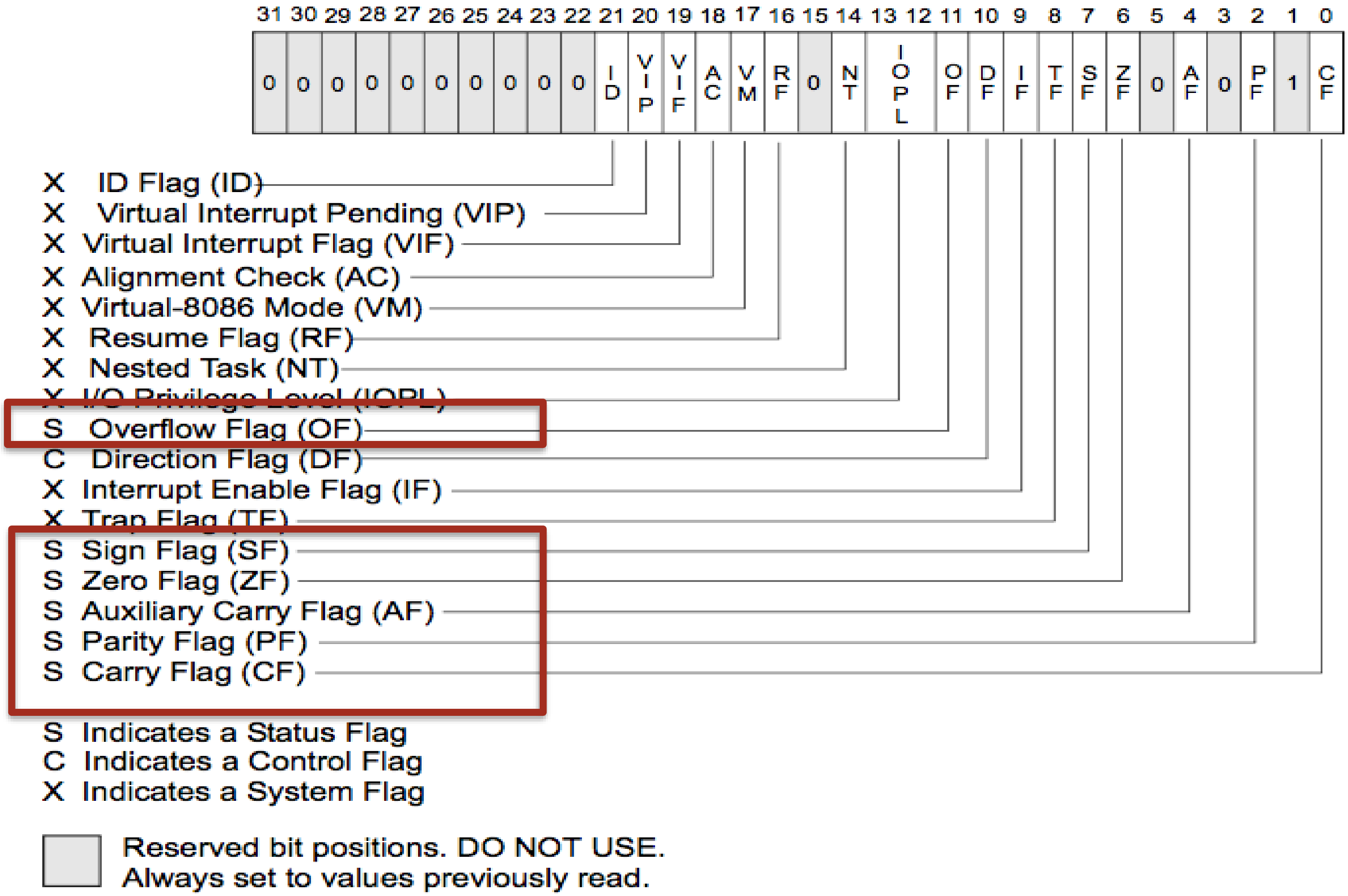
Same as “sub” instruction
r = %eax - M[%ebp+0xc], i.e., x – y

Jump if <u>CF=0</u> and <u>ZF=0</u>

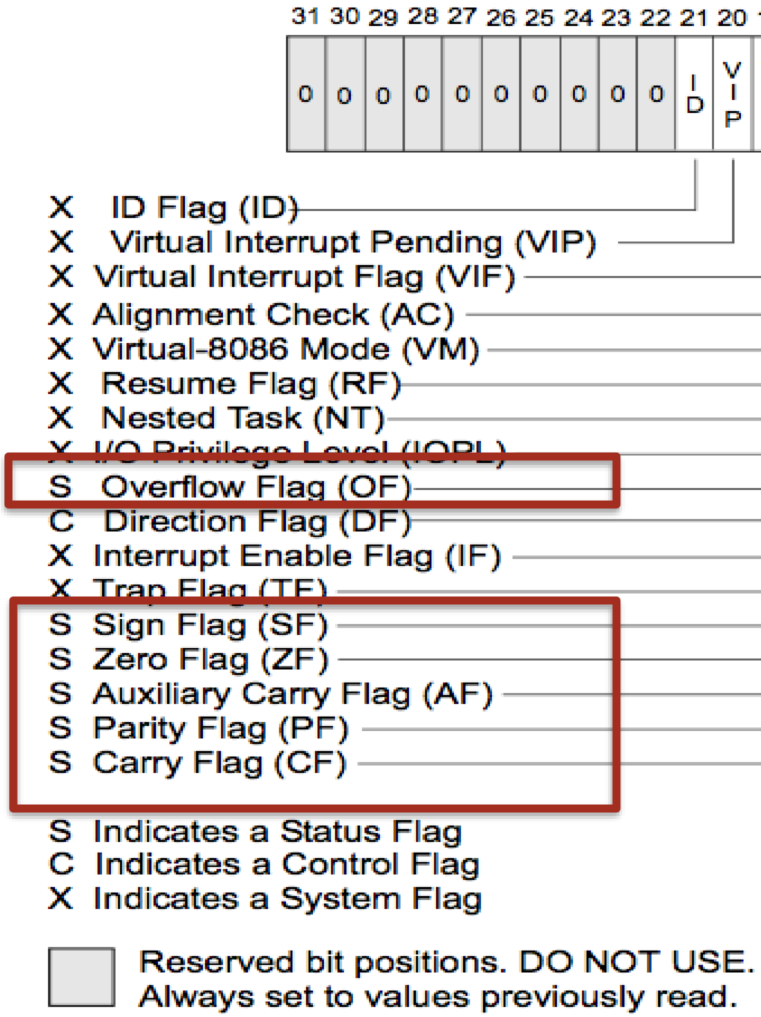
$$(x >= y) \land (x \mathrel{!=} y) \Rightarrow x > y$$

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)


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

From the Intel x86 manual



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

□ Reserved bit positions. DO NOT USE.
Always set to values previously read.

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

From the Intel x86 manual

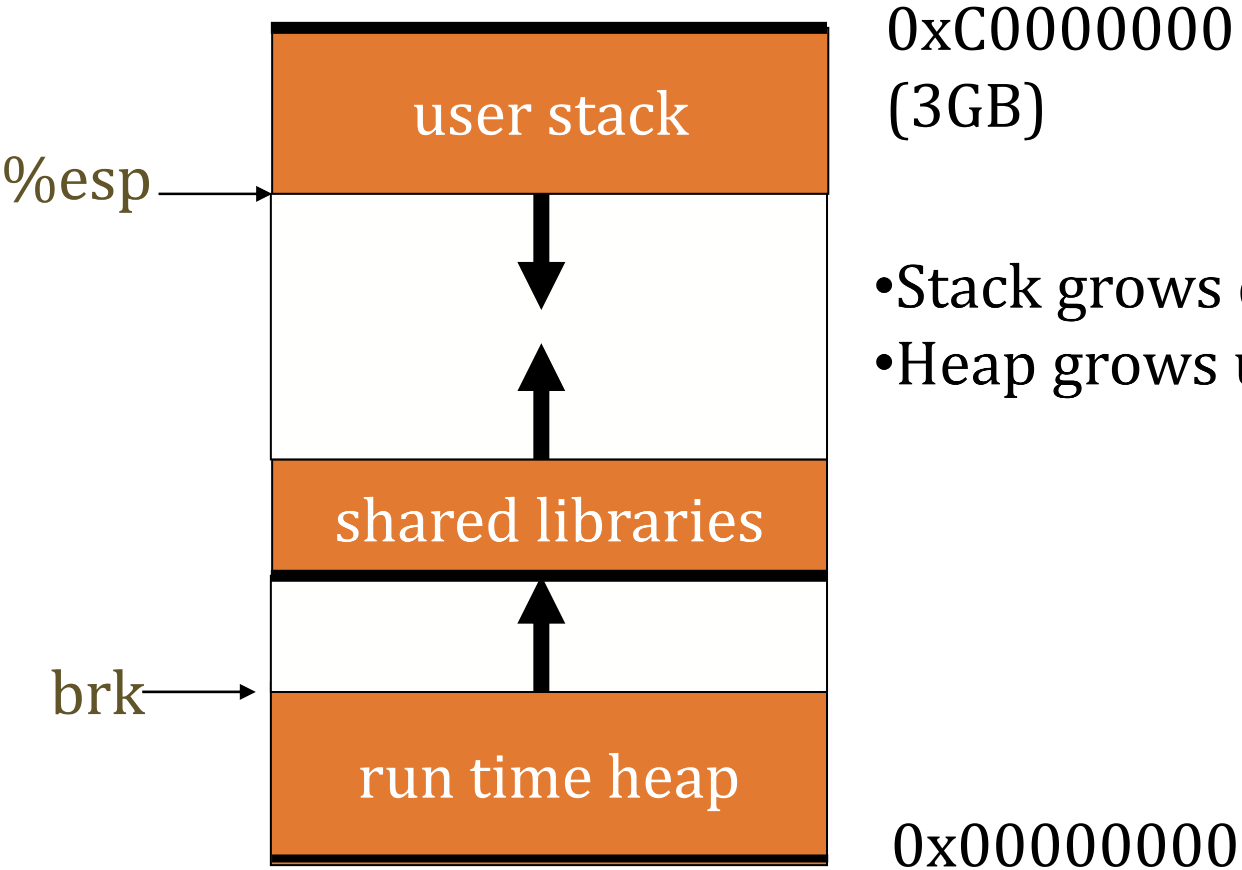
See the x86 manuals available on Intel’s website for more information

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

Agenda

- Compilation Workflow ✓
- x86 Execution Model
  - Basic Execution ✓
  - Memory Operation ✓
  - Control Flow ✓
  - Memory Organization ← NEXT

Memory
Program text
Shared libs
Data
...

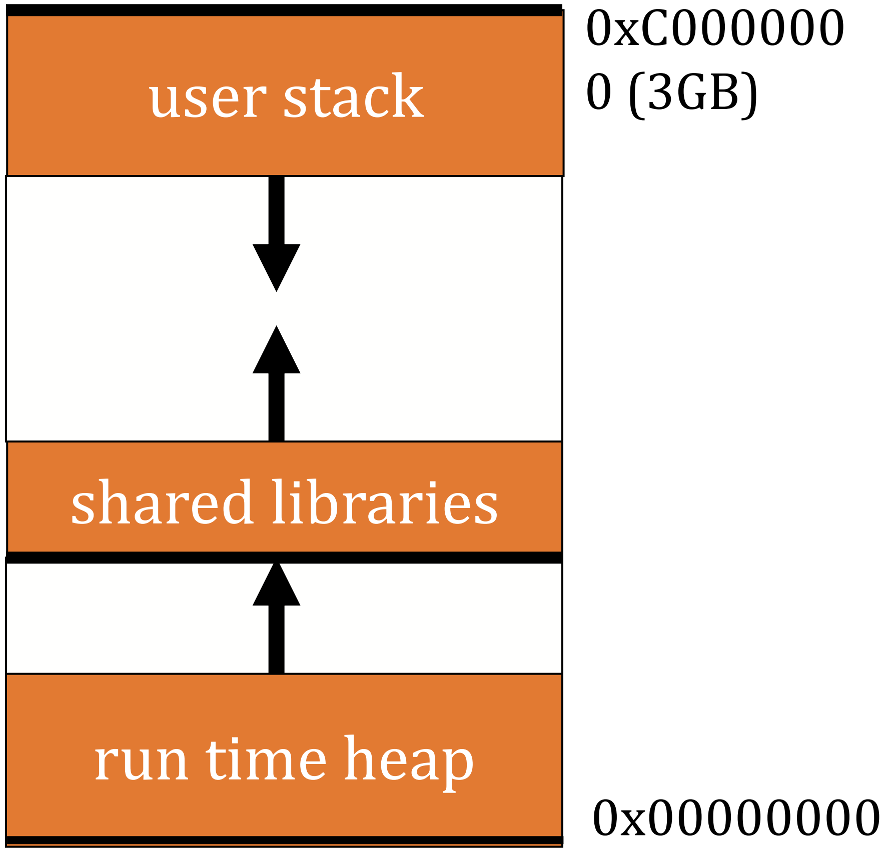


- Stack grows down
- Heap grows up

The Stack grows down towards lower addresses.

Variables

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



Procedures and Stacks

- Procedures are not native to assembly
- Compilers *implement* procedures
  - On the stack
  - Following the call/return stack discipline
  - Work together with x86 instruction call/ret

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

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()
}
```

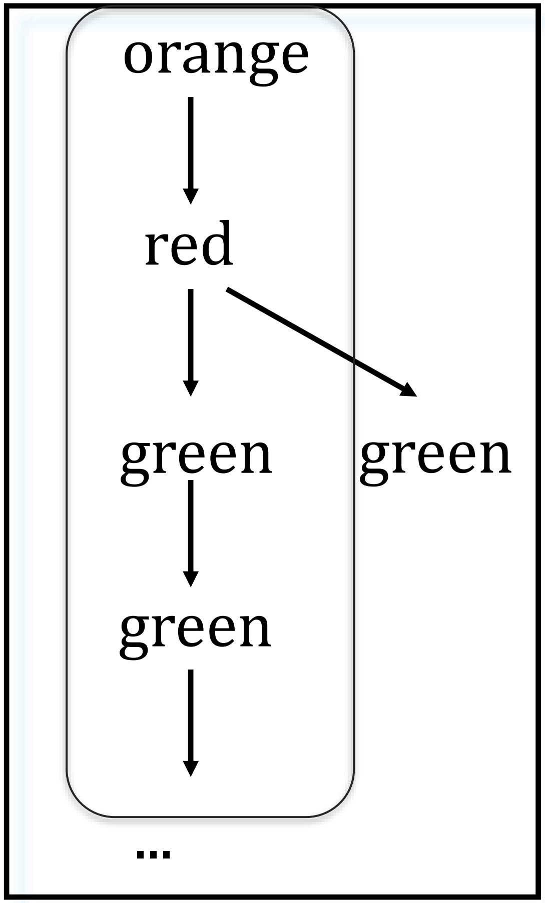
```
green(…)
{
    …
    green()
    …
}
```

```
orange(…)
{
    …
    red()
    …
}
```

```
red(…)
{
    …
    green()
    …
    green()
}
```

```
green(…)
{
    …
    green()
    …
}
```

Function Call Chain



orange

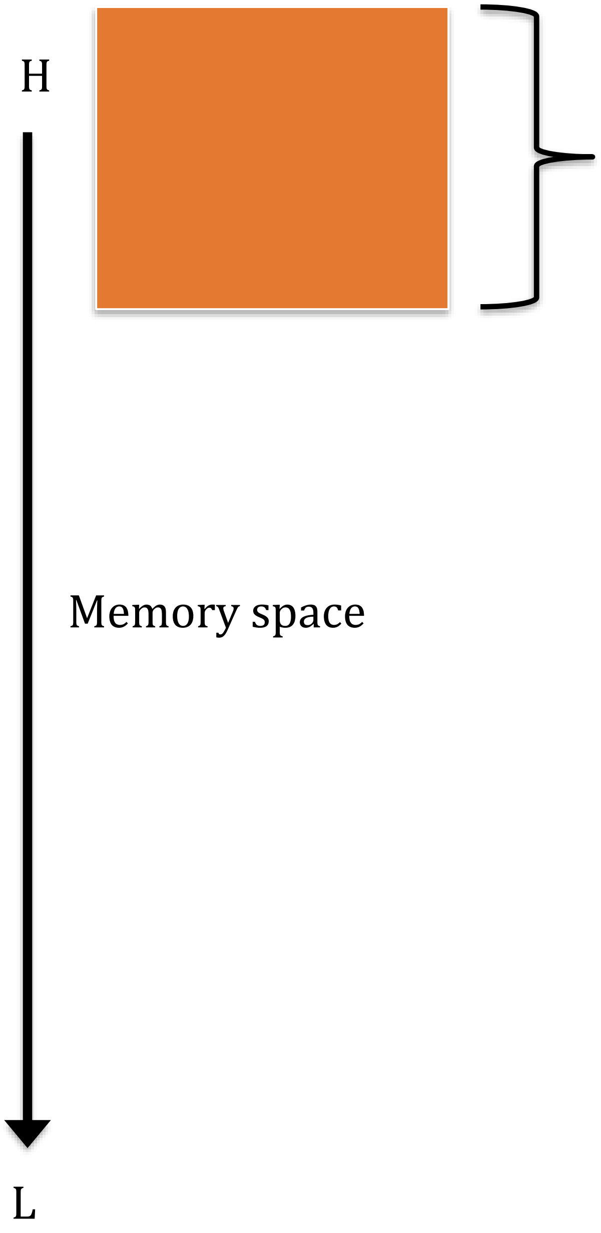
red

green

green

green

…



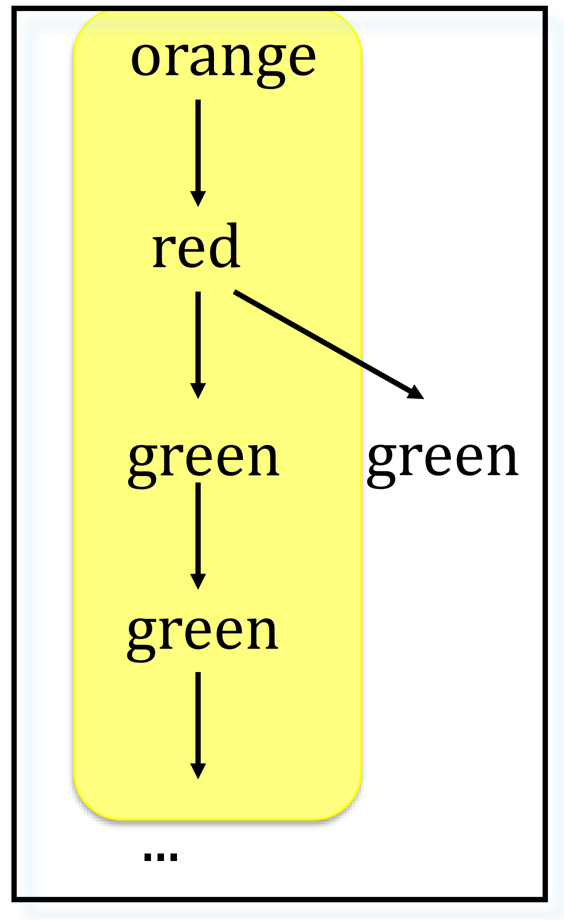
H

Memory space

L

Frame for
- locals
- pushing arguments
- temporary space

Function Call Chain



orange

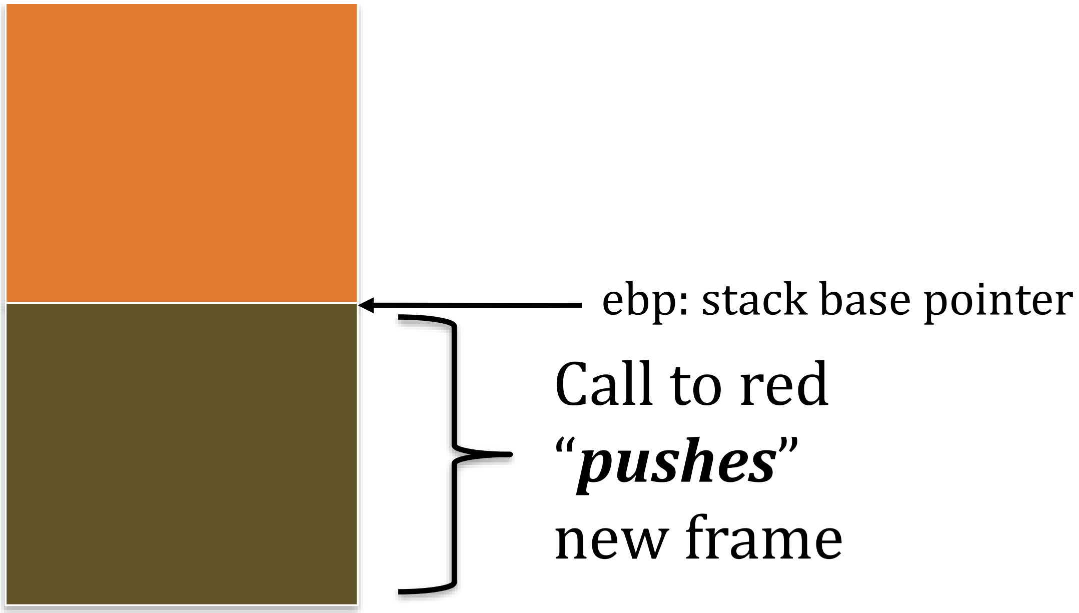
red

green

green

green

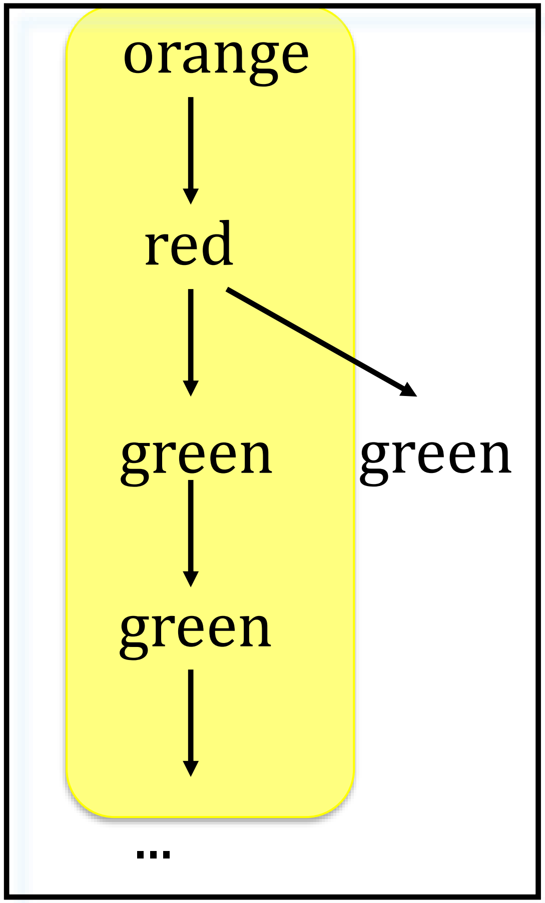
...



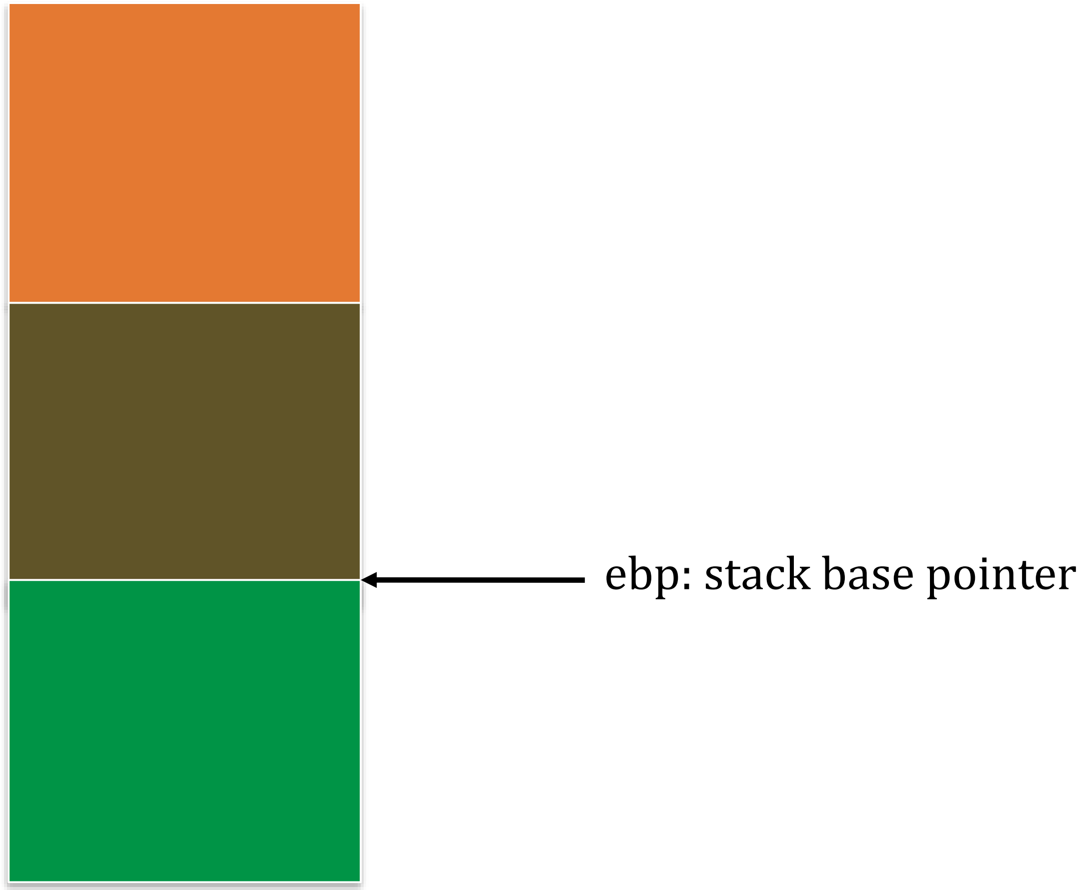
ebp: stack base pointer

Call to red
“***pushes***”
new frame

Function Call Chain

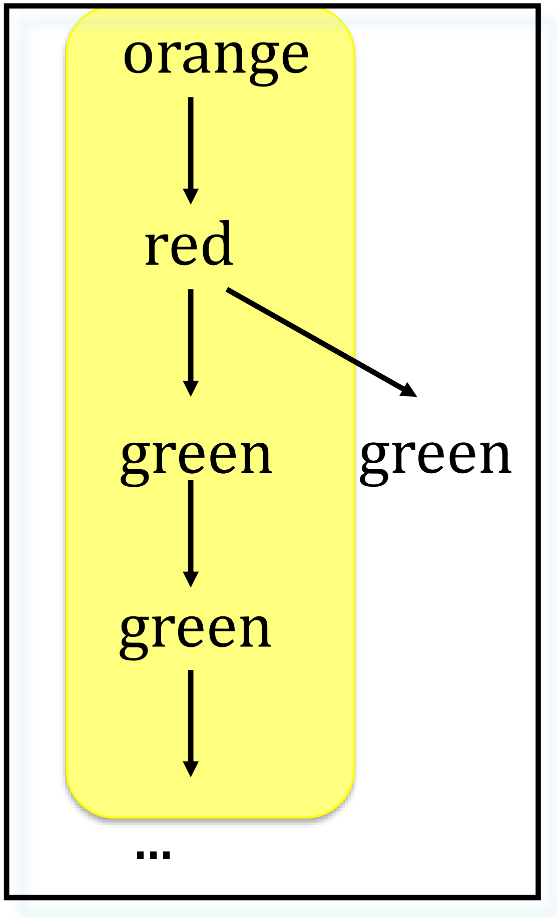


orange
red
green
green
green
...



ebp: stack base pointer

Function Call Chain



orange

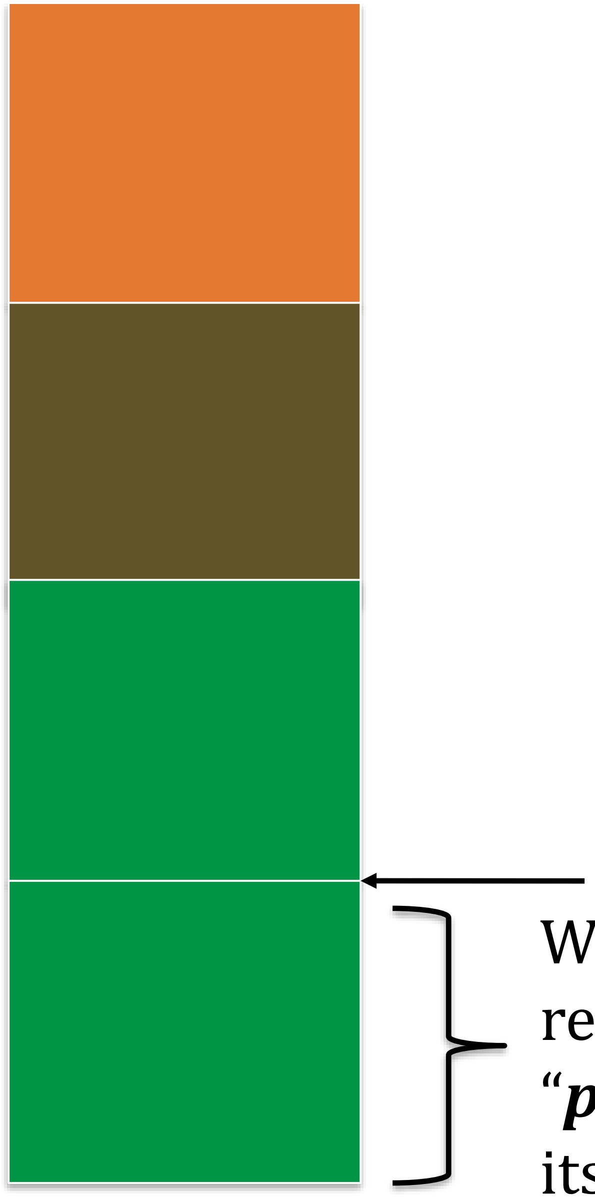
red

green

green

green

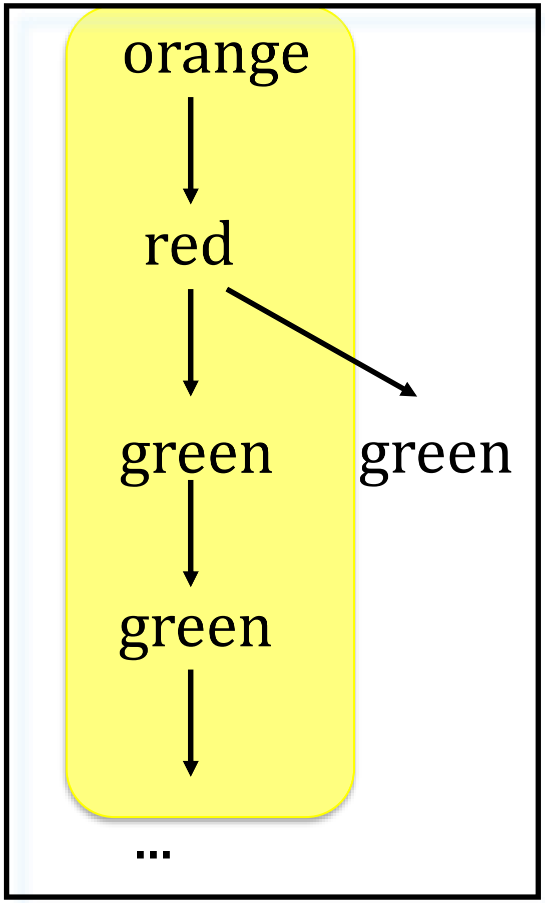
...

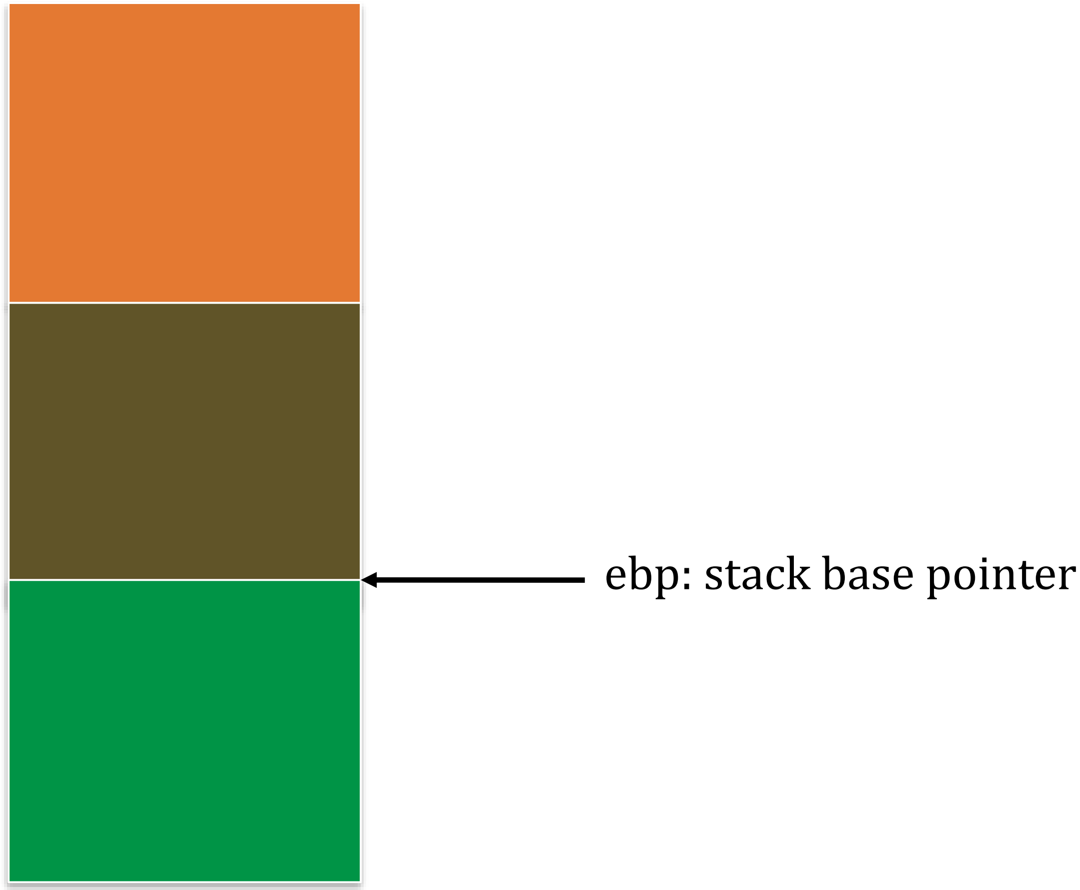


ebp: stack base pointer

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

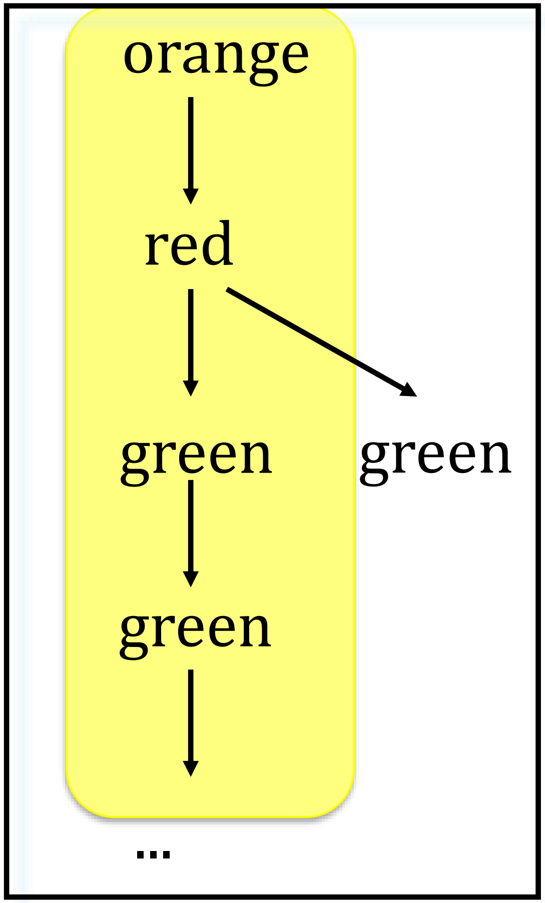
Function Call Chain





ebp: stack base pointer

Function Call Chain



orange

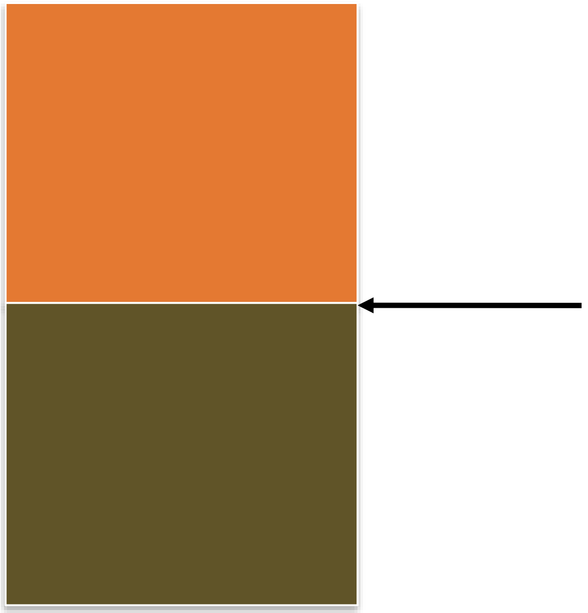
red

green

green

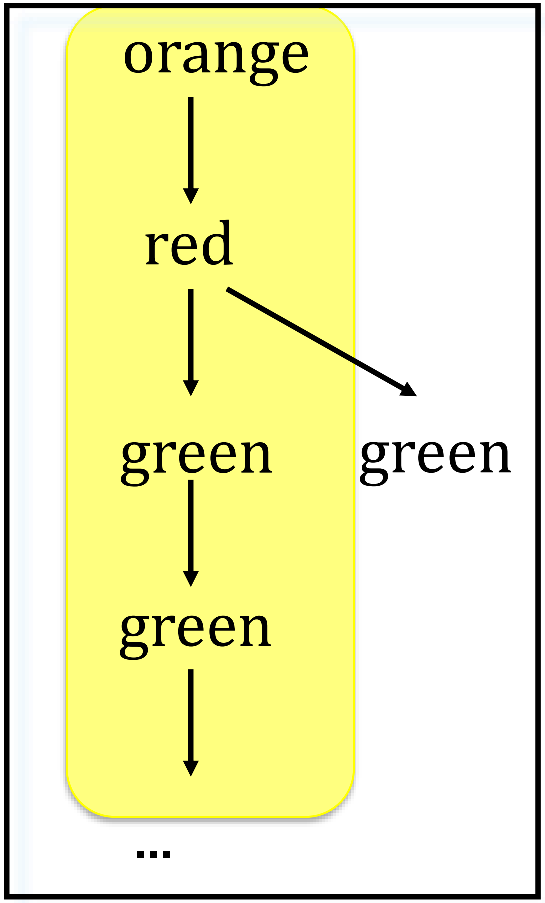
green

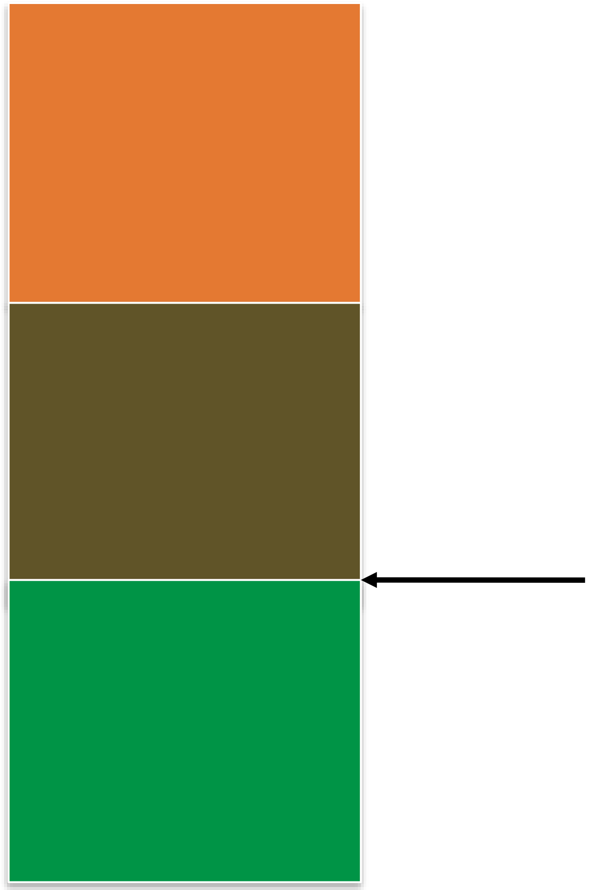
...



ebp: stack base pointer

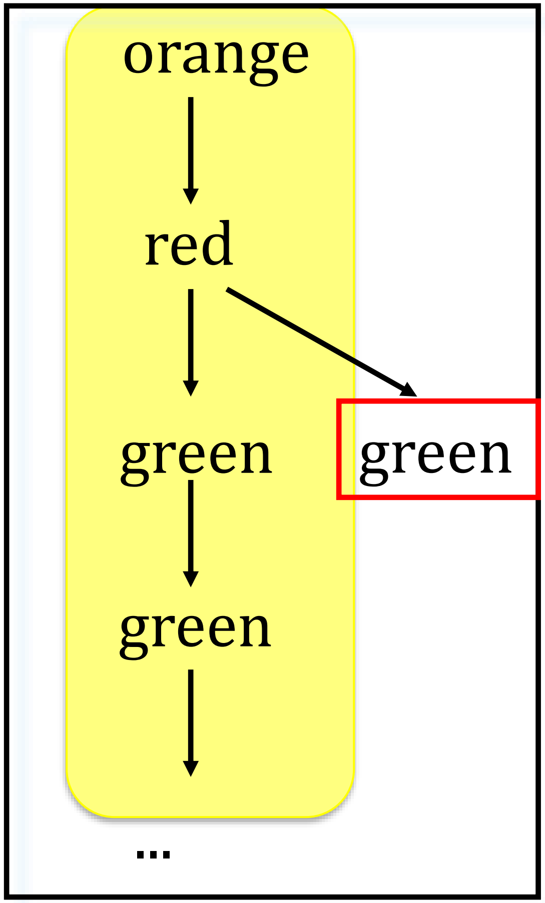
Function Call Chain





ebp: stack base pointer

Function Call Chain



orange

red

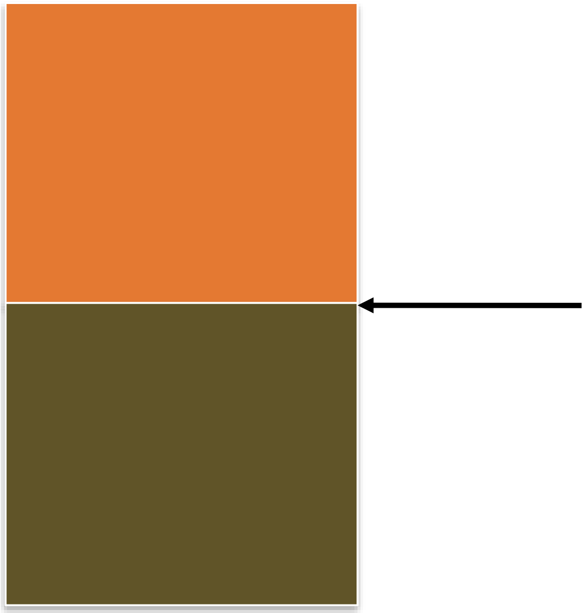
green

green

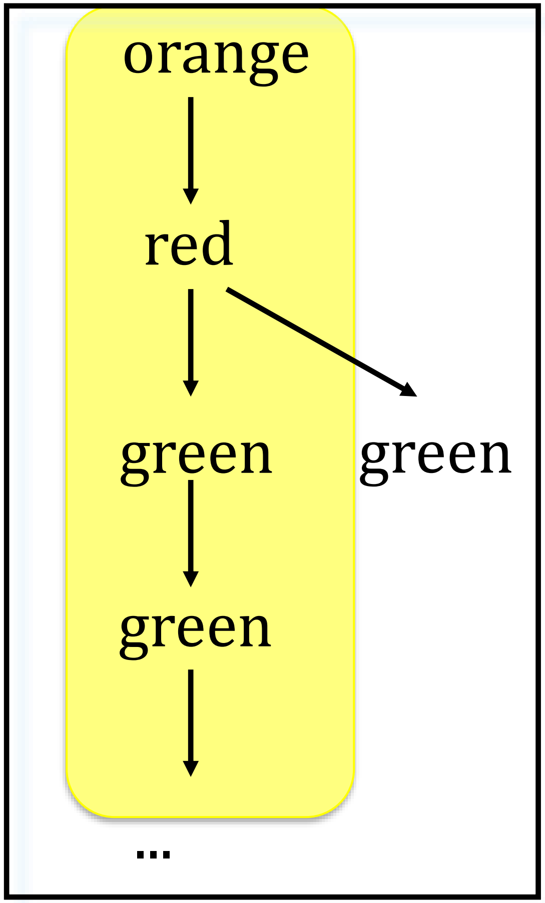
green

...

Function Call Chain



ebp: stack base pointer



orange

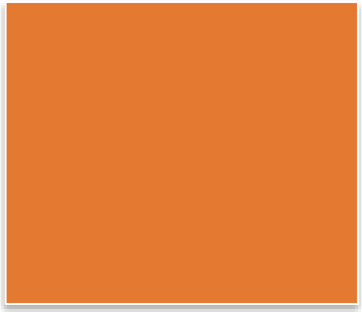
red

green

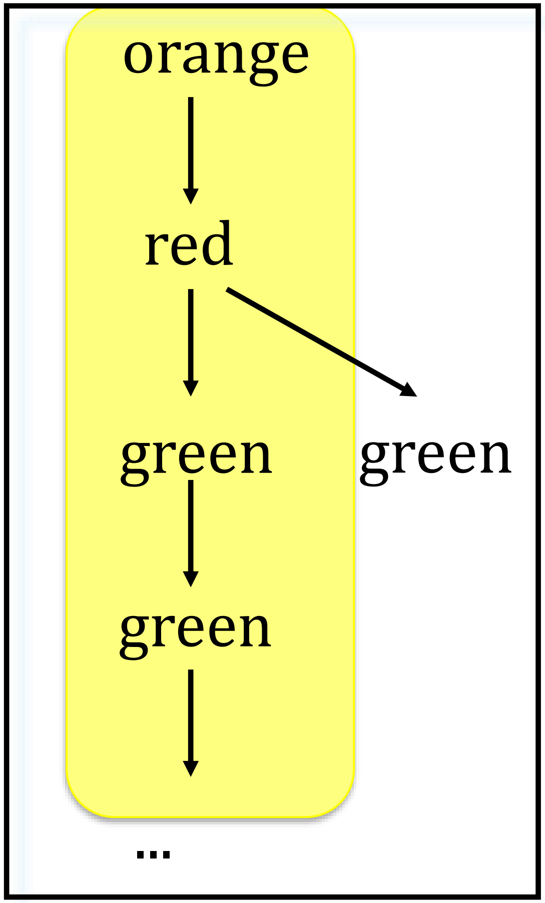
green

green

...



Function Call Chain



orange

red

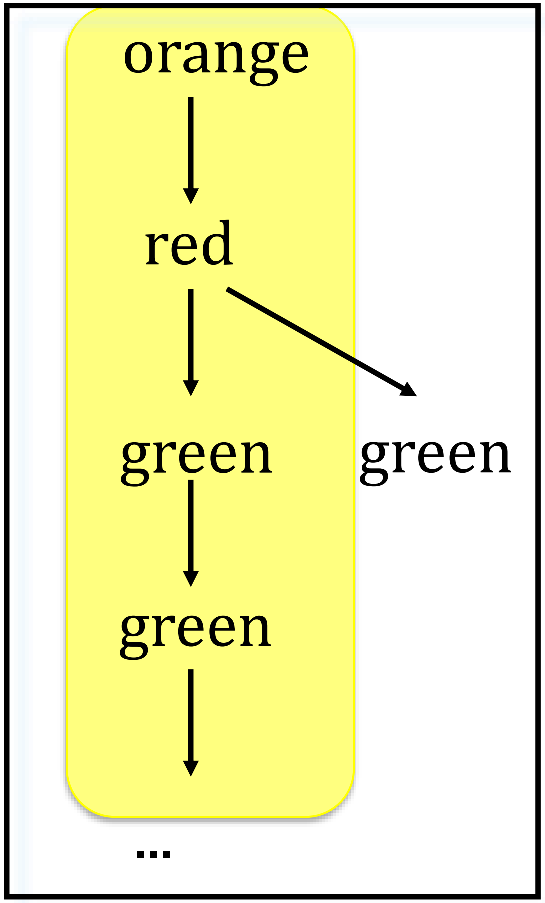
green

green

green

...

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;
}
```

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

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)

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

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

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)

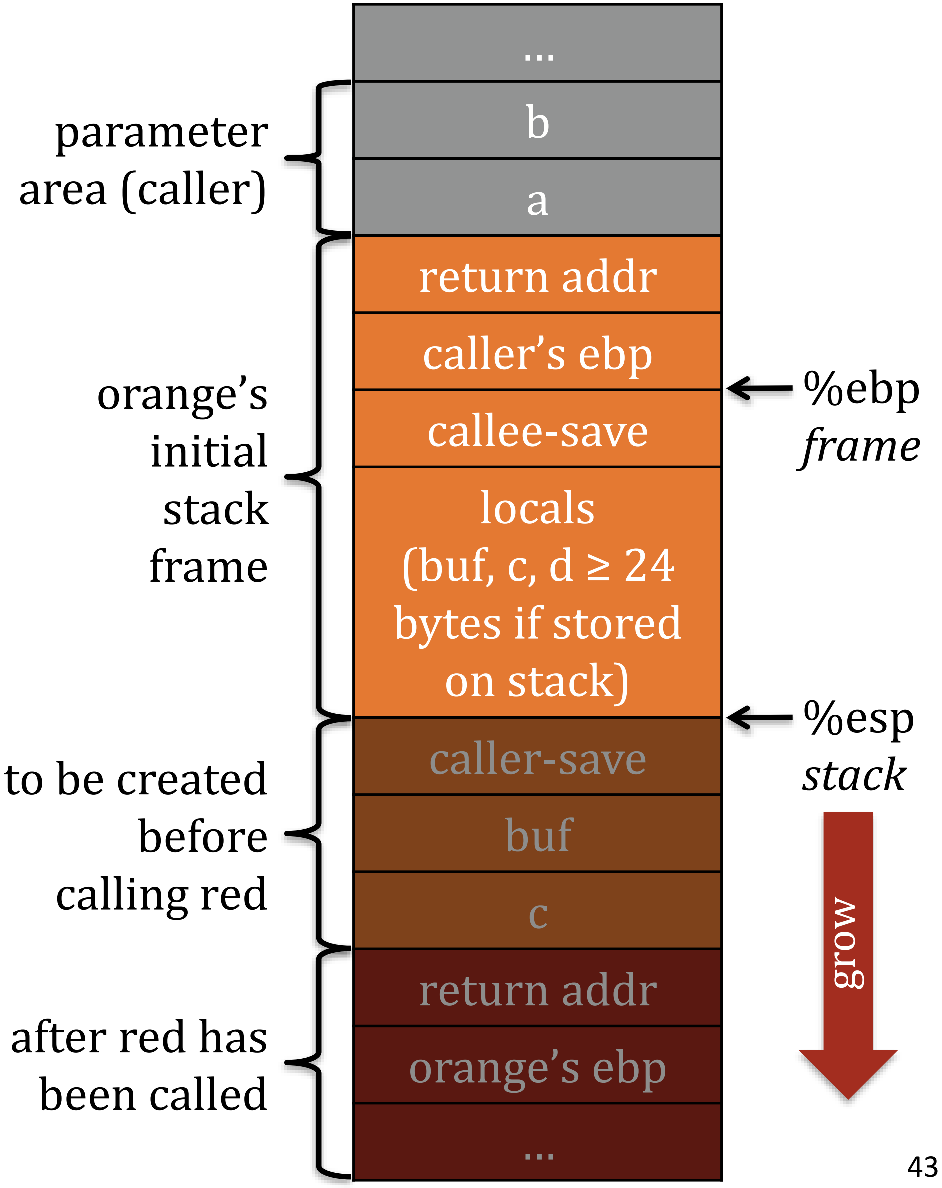
Need space to put arguments for callee

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;
}
```



…

b

a

return addr

caller's ebp

callee-save

locals
(buf, c, d ≥ 24
bytes if stored
on stack)

caller-save

buf

c

return addr

orange's ebp

…

parameter area (caller)

orange's initial stack frame

to be created before calling red

after red has been called

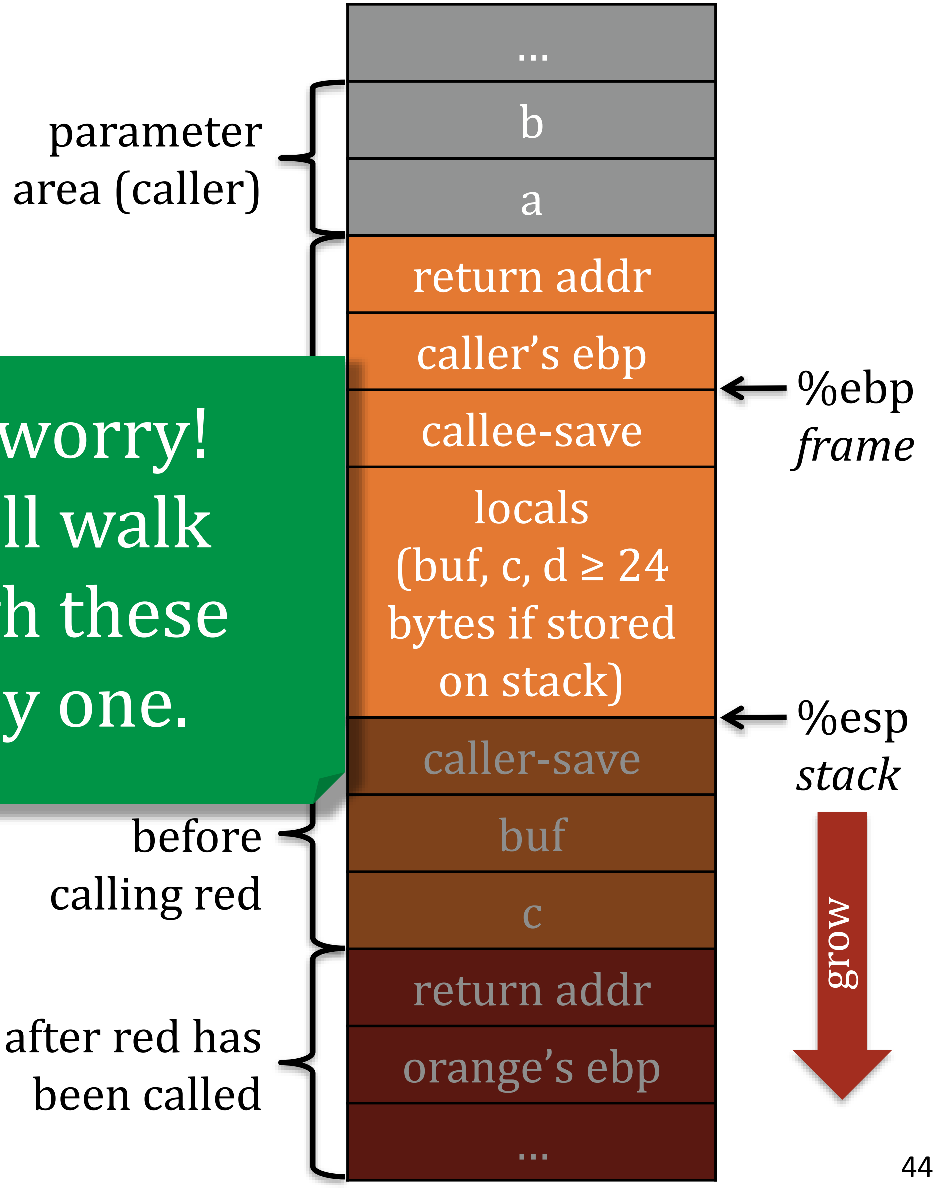
%ebp *frame*

%esp *stack*

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.



Register Saving Conventions

- When procedure **foo** calls **bar**:
  - **foo** is the ***caller***
  - **bar** is the ***callee***
- Can register be used for temporary storage?

```
foo:
    • • •
    movl $15213, %edx
    call bar
    addl %edx, %eax
    • • •
    ret
```

```
bar:
    • • •
    movl 8(%ebp), %edx
    addl $18243, %edx
    • • •
    ret
```

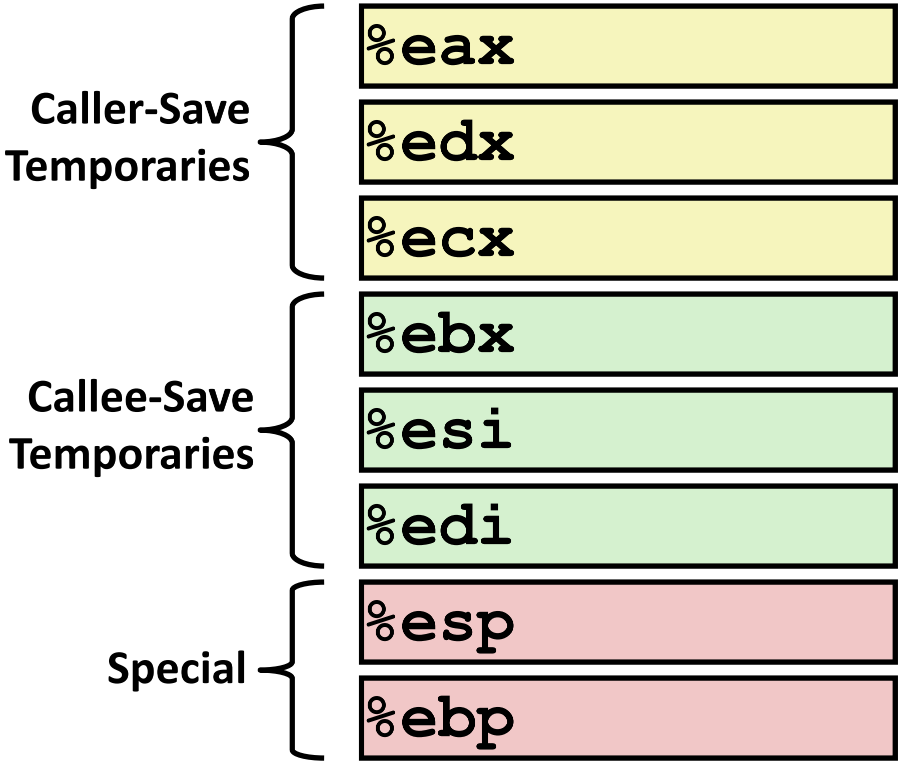
  - Contents of register **%edx** overwritten by **bar**
  - This could be trouble → something should be done!
    - Need some coordination

Register Saving Conventions

- When procedure **`foo`** calls **`bar`**:
  - **`foo`** is the ***caller***
  - **`bar`** is the ***callee***
- Can register be used for temporary storage?
- Conventions
  - ***"Caller Save"***
    - Caller saves temporary values in its frame before the call
  - ***"Callee Save"***
    - Callee saves temporary values in its frame before using

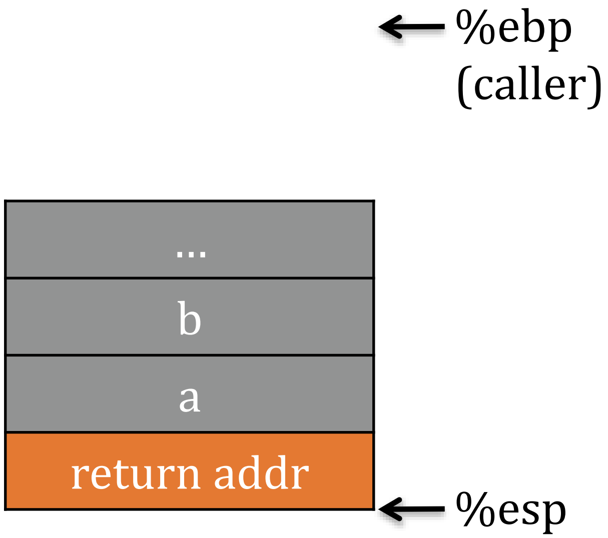
IA32/Linux+Windows Register Usage

- **`%eax`**, **`%edx`**, **`%ecx`**
  - Caller saves prior to call if values are used later
- **`%eax`**
  - also used to store the return value
- **`%ebx`**, **`%esi`**, **`%edi`**
  - Callee saves if wants to use them
- **`%esp`**, **`%ebp`**
  - special form of callee save
  - Restored to original values upon exit from procedure



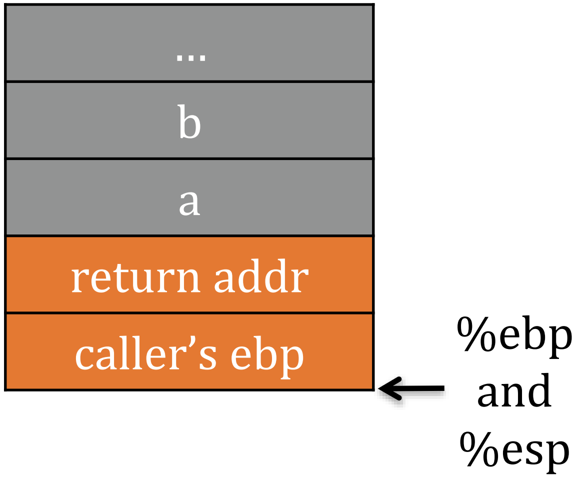
When orange attains control,

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



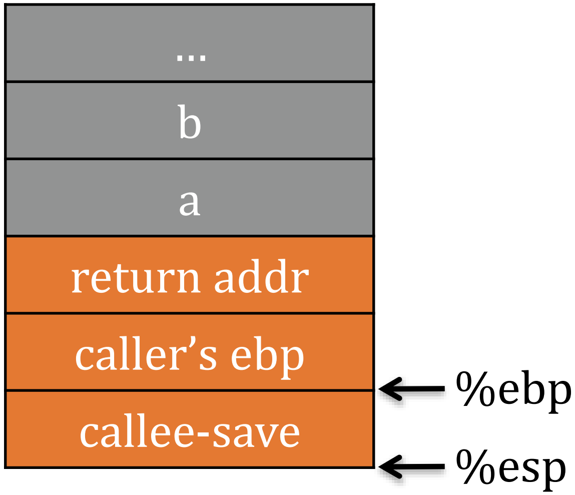
When orange attains control,

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



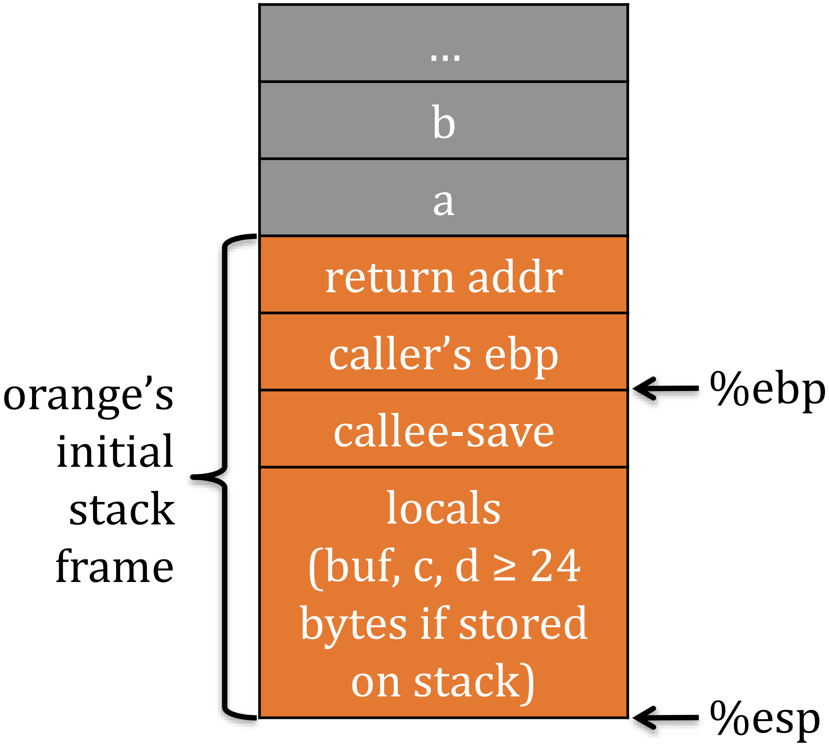
When orange attains control,

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

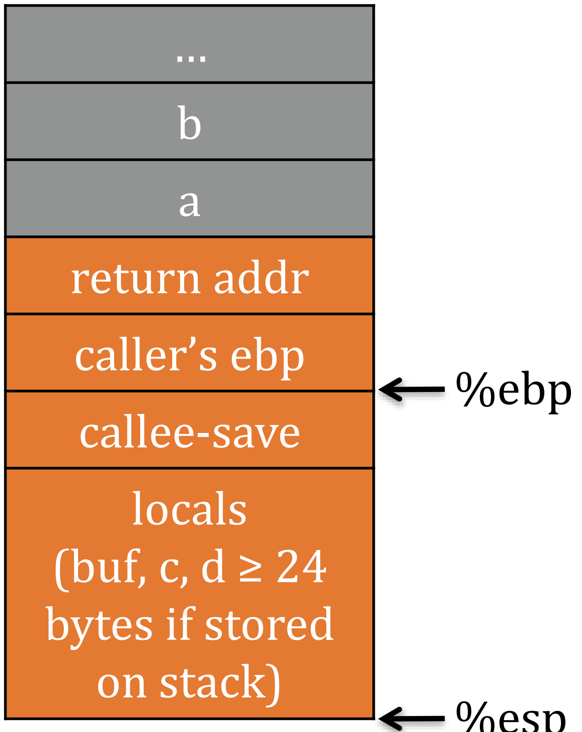


When orange attains control,

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

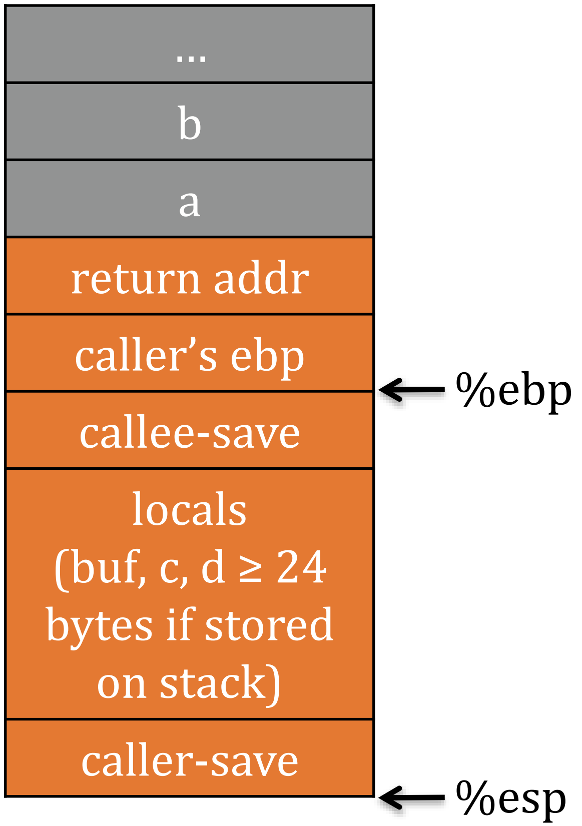


For *caller* orange to call *callee* red,



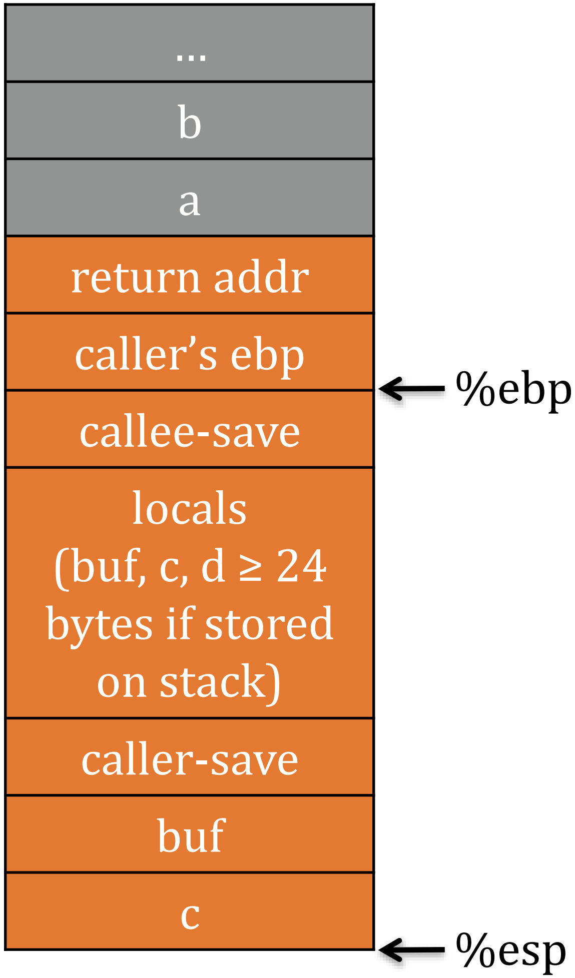
For *caller* orange to call *callee* red,

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



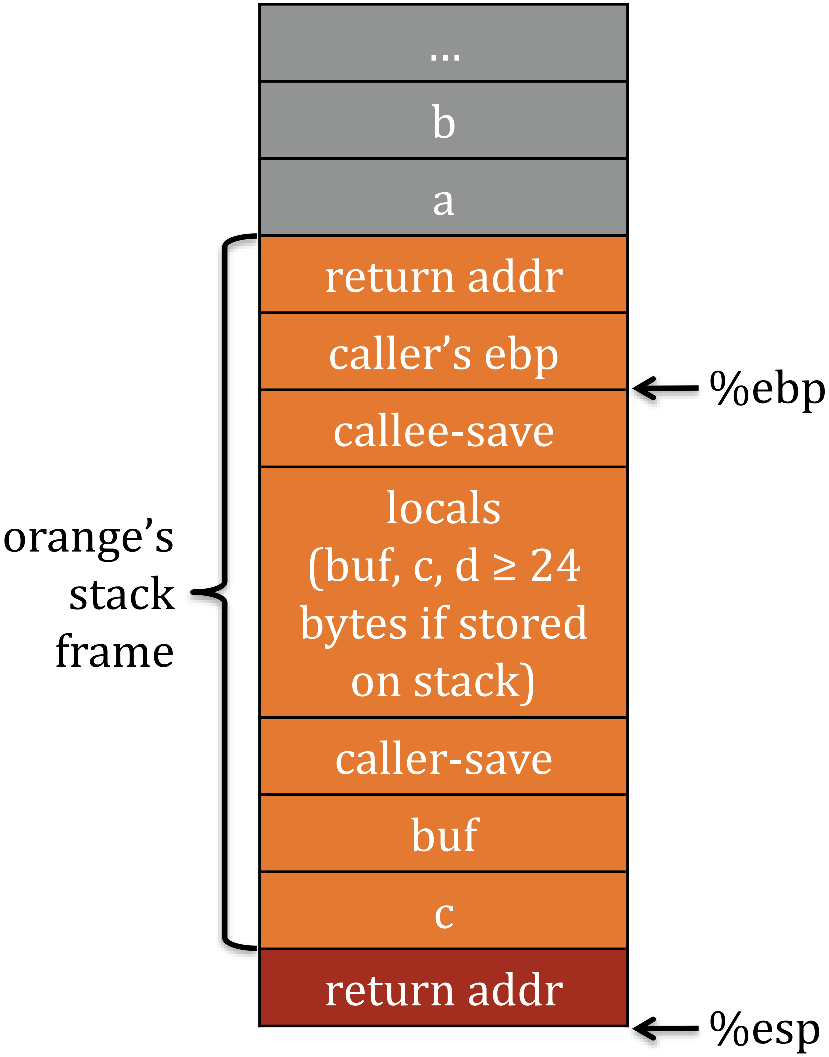
For *caller* orange to call *callee* red,

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 on stack



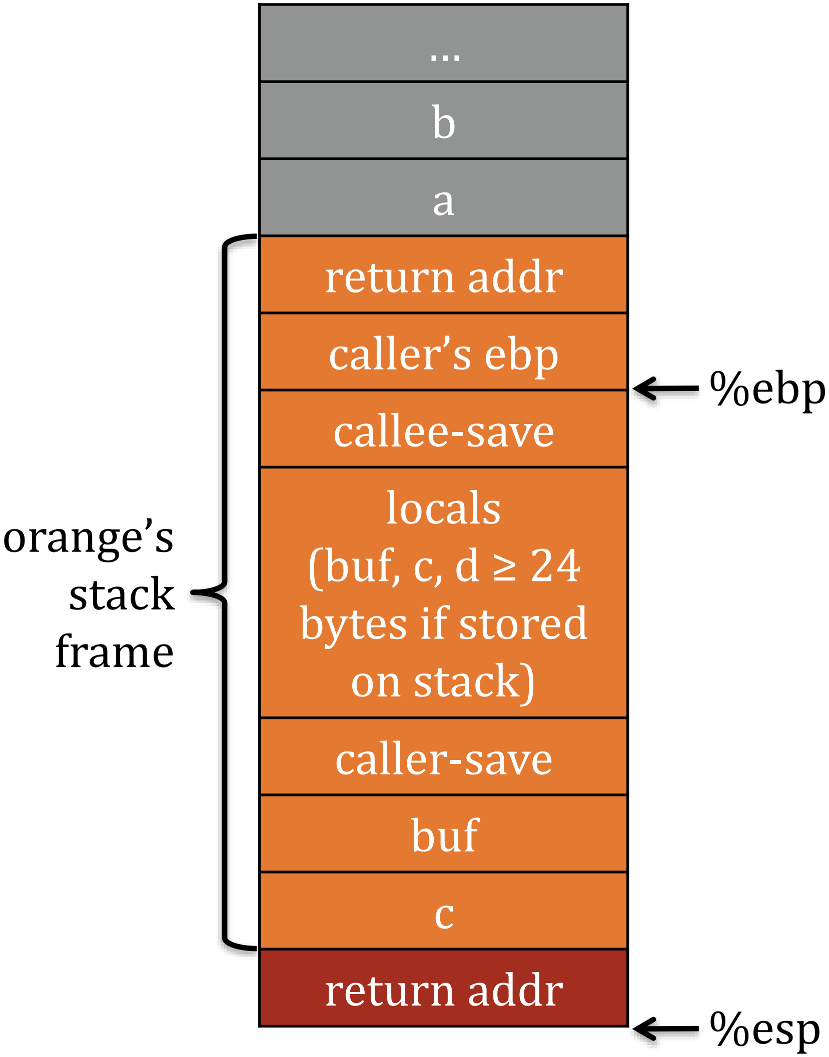
For *caller* orange to call *callee* red,

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 on stack
3. push return address, i.e., the *next* instruction to execute in orange after red returns



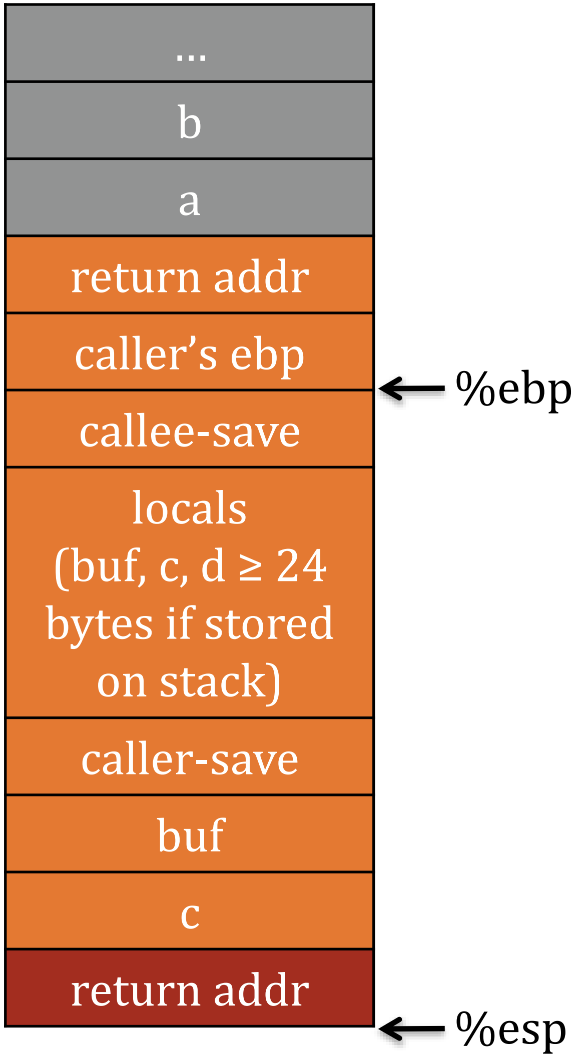
For *caller* orange to call *callee* red,

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 on 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`



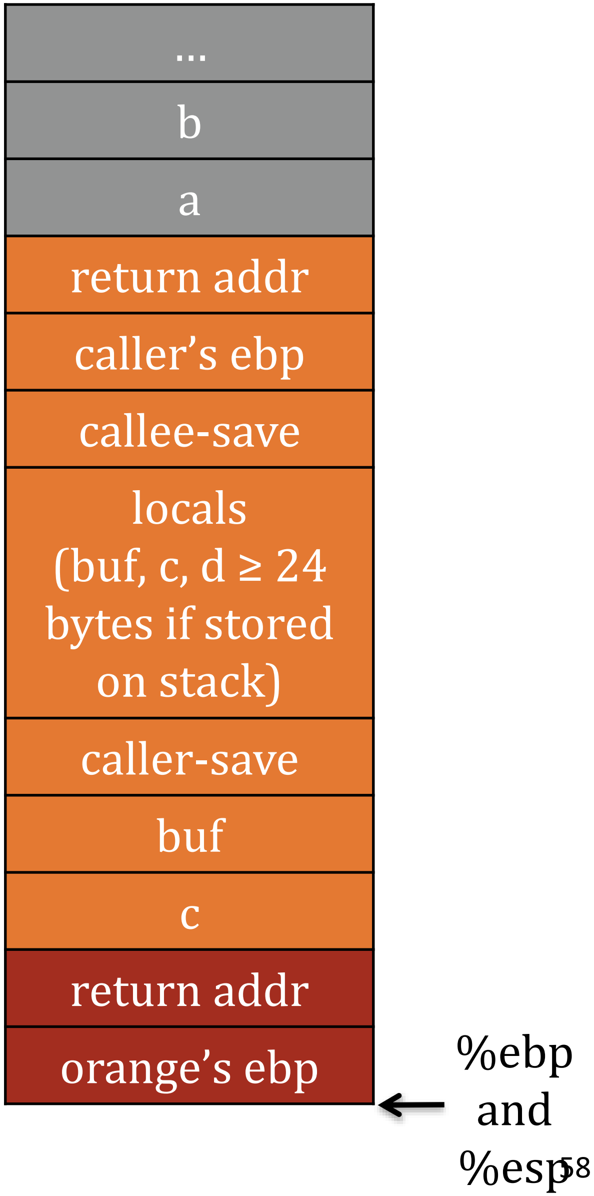
When red attains control,

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



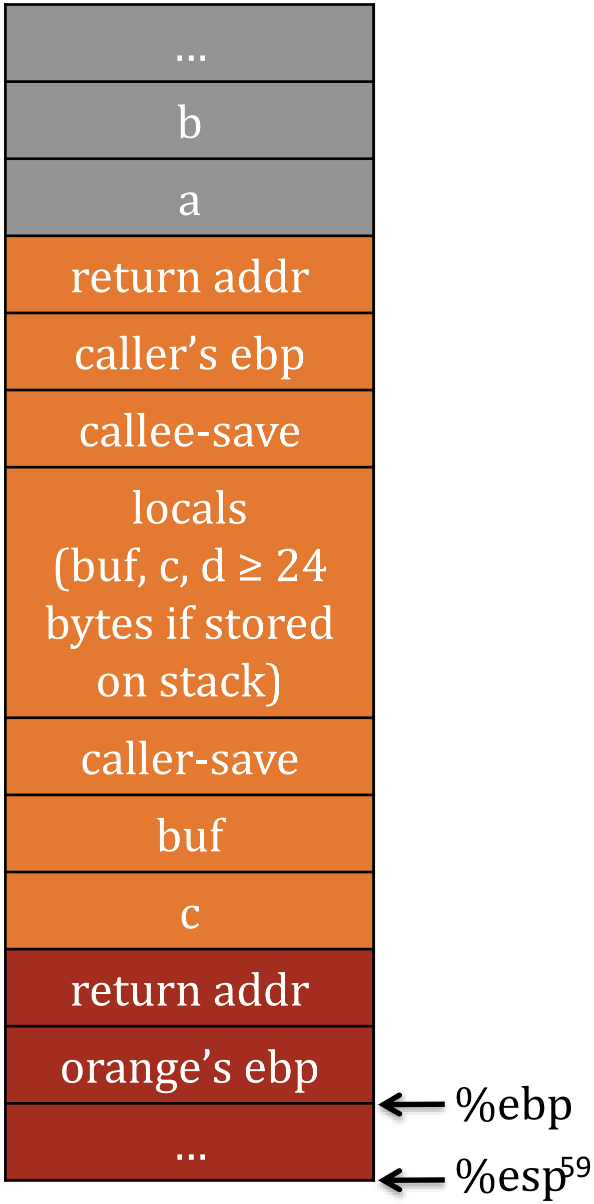
When red attains control,

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



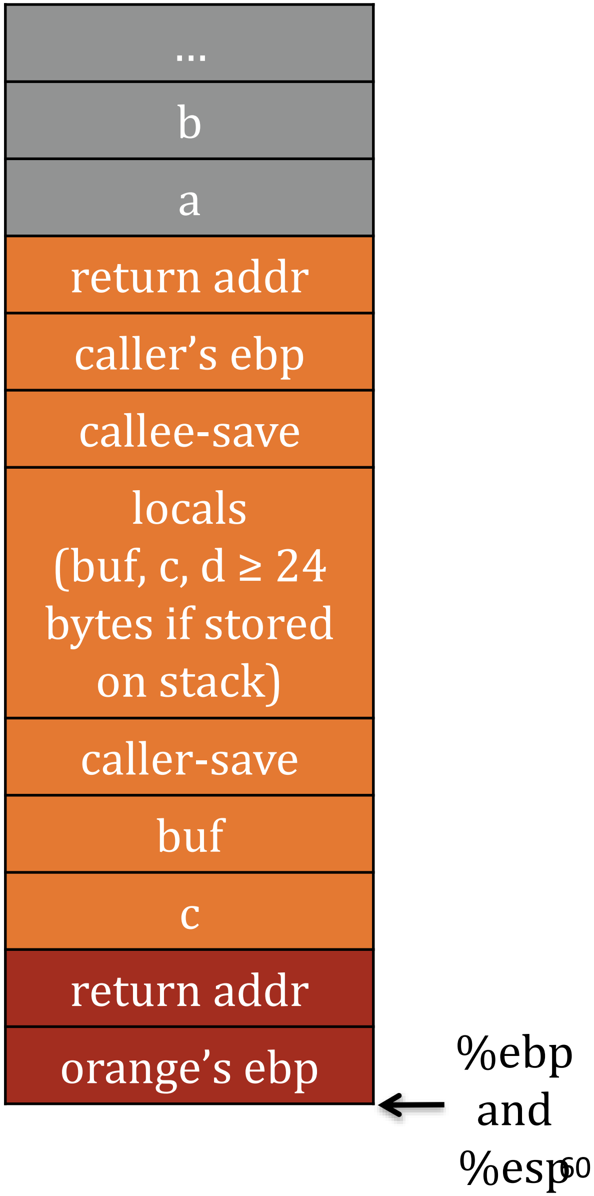
When red attains control,

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



…

b

a

return addr

caller’s ebp

callee-save

locals (buf, c, d ≥ 24 bytes if stored on stack)

caller-save

buf

c

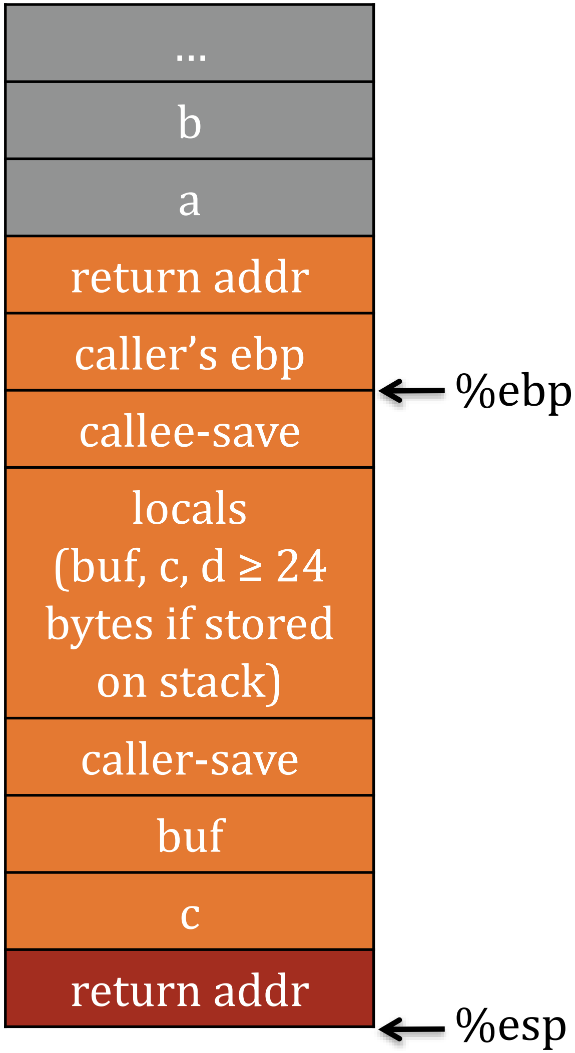
return addr

orange’s ebp

%ebp and %esp

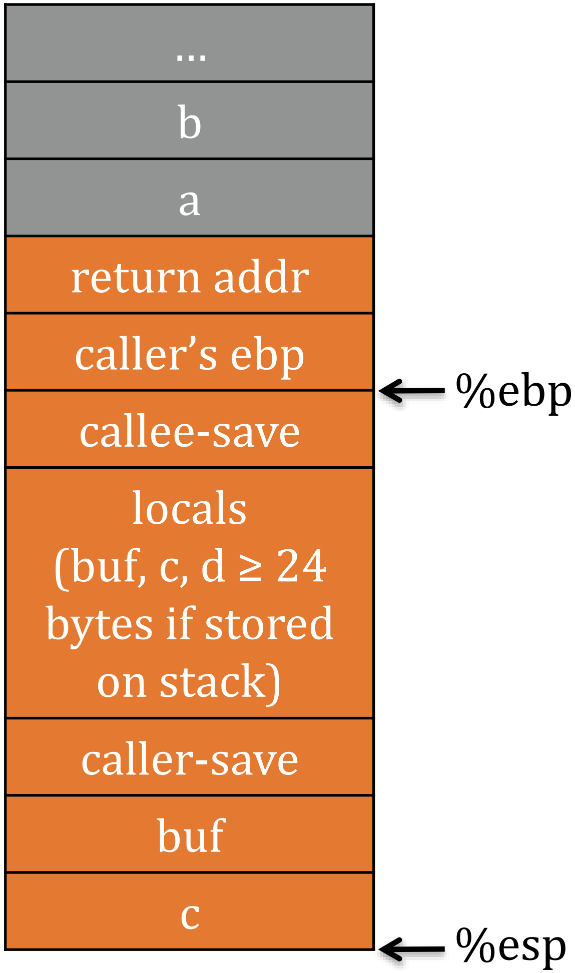
When red attains control,

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`

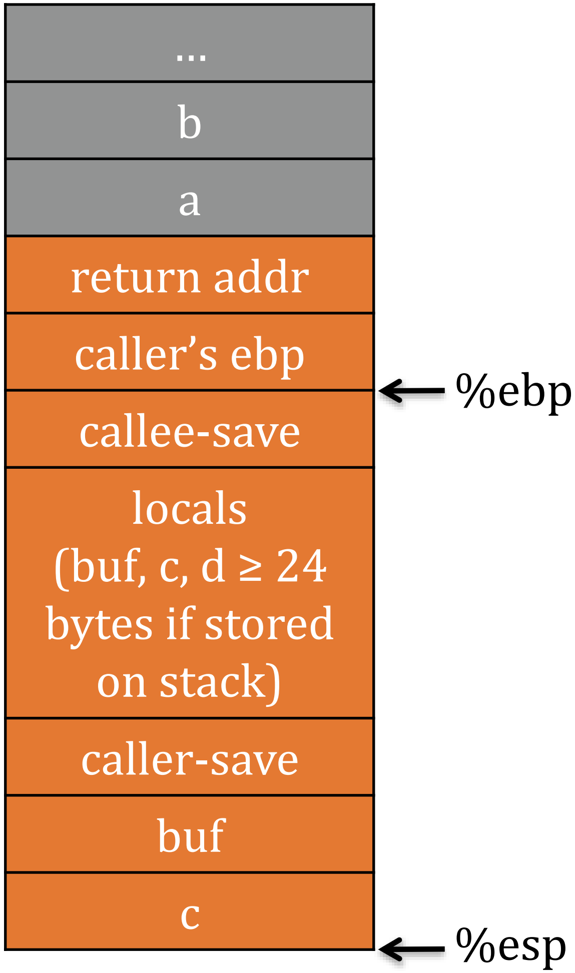


When red attains control,

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 (EIP changed)

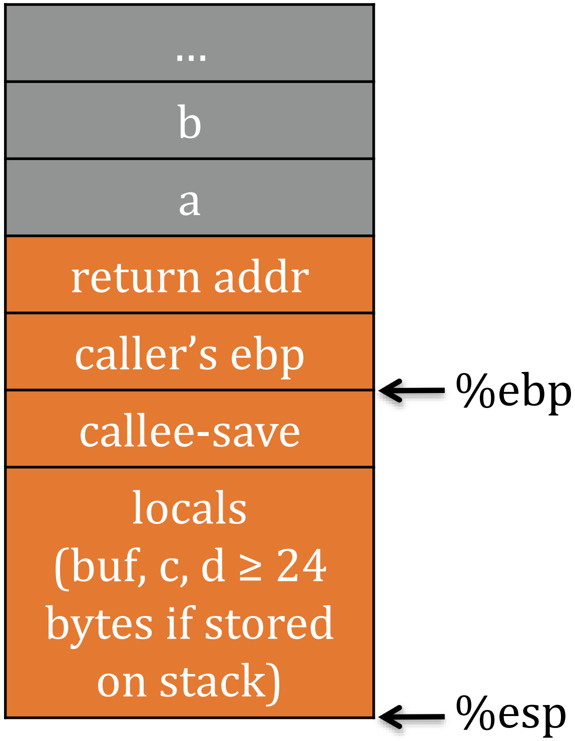


When orange regains control,



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

Quiz

- printf(“%s, %d”, aString, anInteger);

- How are the arguments pushed onto stack?