C FOR EMBEDDED SYSTEMS

(CONT. 1)

LECTURE II
TDDI11 Embedded Software

DEPT. COMPUTER AND INFORMATION SCIENCE (IDA)

LINKÖPINGS UNIVERSITET

OUTLINE

- **Bit manipulation**
- Mixing C and assembly

BOOLEAN AND BINARY OPERATORS

Operation	Boolean Operator	Bitwise Operator
AND	&&	&
OR	||	|
XOR	*unsupported*	^
NOT	!	~

- Boolean operators are primarily used to form conditional expressions (as in an *if* statement)
- Bitwise operators are used to manipulate bits.

BOOLEAN VALUES

- Most implementations of *C* don't provide a Boolean data type.
- Any numeric data type may be used as a Boolean operand.
- Boolean operators yield results of type *int*, with true and false represented by 1 and 0.
- Zero is interpreted as false; any non-zero value is interpreted as true.

BOOLEAN EXPRESSIONS

(5 || !3) && 6

True / False ?

= (true OR (NOT true)) AND true

= (true OR false) AND true

= (true) AND true

= true

= 1

BITWISE OPERATORS

- Bitwise operators operate on individual bit positions within the operands

- The result in any one bit position is entirely independent of all the other bit positions.

INTERPRETING THE BITWISE-AND

m	p	m AND p	interpretation
0	0	0	If bit m of the mask is 0, then bit p is **cleared** to 0.
	1	0	
1	0	p	If bit m of the mask is 1, then bit p is **unchanged**.
	1	1	

INTERPRETING THE BITWISE-OR

m	p	m OR p	interpretation
0	0	0	If bit m of the mask is 0, then bit p is **unchanged**.
	1	1	
1	0	1	If bit m of the mask is 1, then bit p is **set** to 1.
	1	1	

INTERPRETING THE BITWISE-XOR

m	p	m XOR p		interpretation
0	0	0	p	If bit m of the mask is 0, then bit p is **unchanged**.
	1	1		
1	0	1	~p	If bit m of the mask is 1, then bit p is **inverted**.
	1	0		

BITWISE EXPRESSIONS

(5 | ~3) & 6

= (00..0101 OR ~00..0011) AND 00..0110

= (00..0101 OR 11..1100) AND 00..0110

= (11..1101) AND 00..0110

= 00..0100

= 4

TESTING BITS

- A 1 in the bit position of interest is AND'ed with the operand. The result is non-zero if and only if the bit of interest was 1:

```
if  ((bits  &  64) != 0)          /* check to see if bit 6 is set */
```

TESTING BITS, CONT.

- Since any non-zero value is interpreted as *true*, the redundant comparison to zero may be omitted, as in:

```
if  (bits & 64) /* check to see if bit 6 is set */
```

TESTING BITS, CONT.

- The mask (64) is often written in hex (0x0040), but a constant-valued shift expression provides a clearer indication of the bit position being tested:

```
if (bits & (1 << 6)) /* check to see if bit 6 is set */
```

- Almost all compilers will replace such constant-valued expressions by a single constant, so using this form almost never generates any additional code.

TESTING KEYBOARD FLAGS USING BITWISE OPERATORS

```
#define     FALSE       (0)
#define     TRUE        (1)

typedef unsigned char   BOOL ;

typedef struct SHIFTS
{
BOOL right_shift;
BOOL left_shift;
BOOL ctrl;
BOOL alt;
BOOL left_ctrl;
BOOL left_alt;
} SHIFTS;
```

```
BOOL Kybd_Flags_Changed(SHIFTS  *) ;
void Display_Kybd_Flags(SHIFTS  *) ;

void main()
{
SHIFTS  kybd ;
  do
  { /* repeat until both shift keys pressed */
     if (Kybd_Flags_Changed(&kybd))
          Display_Kybd_Flags(&kybd) ;
     } while (!kybd.left_shift ||!kybd.right_shift);
}
```

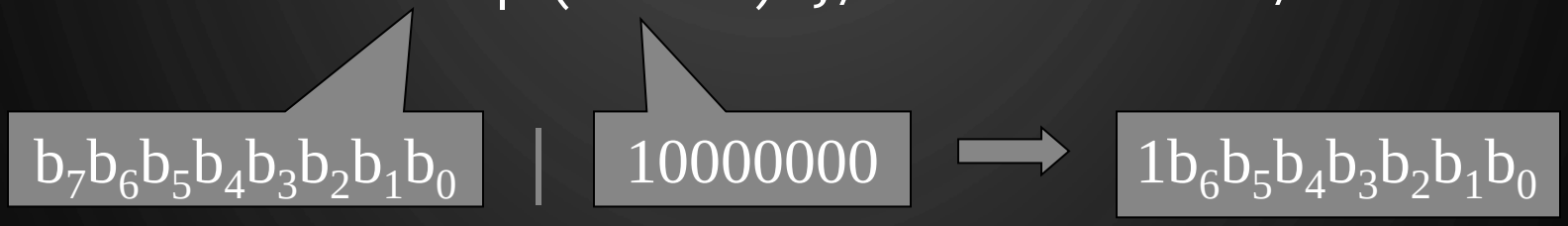
CONTINUED …

```
typedef unsigned int WORD16 ;

BOOL Kybd_Flags_Changed(SHIFTS  *kybd)
{
       static WORD16 old_flags = 0xFFFF ;
       WORD16 new_flags ;

       dosmemget(0x417, sizeof(new_flags), &new_flags) ;
       if (new_flags == old_flags) return FALSE ;
       old_flags = new_flags ;

       kybd->right_shift          = (new_flags & (1 << 0)) != 0 ;
       kybd->left_shift           = (new_flags & (1 << 1)) != 0 ;
       kybd->ctrl                 = (new_flags & (1 << 2)) != 0 ;
       kybd->alt                  = (new_flags & (1 << 3)) != 0 ;
       kybd->left_alt             = (new_flags & (1 << 9)) != 0 ;
       kybd->left_ctrl            = (new_flags & (1 << 8)) != 0 ;

       return TRUE ;
}
```

SETTING BITS

- Setting a bit to 1 is easily accomplished with the bitwise-OR operator:

```
bits = bits | (1 << 7) ;/* sets bit 7 */
```



- This would usually be written more succinctly as:

```
bits |= (1 << 7) ;   /* sets bit 7 */
```

SETTING BITS, CONT.

- Note that we don't *add* (+) the bit to the operand! That only works if the current value of the target bit in the operand is known to be 0.

- Although the phrase *"set a bit to 1"* suggests that the bit was originally 0, most of the time the current value of the bit is actually unknown.

CLEARING BITS

- Clearing a bit to 0 is accomplished with the bitwise-AND operator:

```
bits &= ~(1 << 7) ;       /* clears bit 7 */
```

(1 << 7) ⟹ **10000000**

~(1 << 7) ⟹ **01111111**

- Note that we don't *subtract* the bit from the operand!

CLEARING BITS, CONT.

When clearing bits, you have to be careful that the mask is as wide as the operand. For example, if *bits* is changed to a 32-bit data type, the right-hand side of the assignment must also be changed, as in:

```
bits &= ~(1L << 7);/* clears bit 7 */

bits &= ~((long int)1 << 7);/*clears bit 7 */
```

INVERTING BITS

- Inverting a bit (also known as toggling) is accomplished with the bitwise-XOR operator as in:

```
bits  ^=  (1 << 6); /* flips bit 6 */
```

- Although *adding* 1 would invert the target bit, it may also propagate a carry that would modify more significant bits in the operand.

EXTRACTING BITS

Extract minutes from time

time

time >> 5

(time >> 5) & 0x3F

minutes = (time >> 5) & 0x3F

INSERTING BITS

Updates minutes in time

oldtime

Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
Hours	Old Minutes	Seconds ÷ 2

newtime = oldtime & ~(0x3F << 5)

Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
Hours	000000	Seconds ÷ 2

newtime |= (newmins & 0x3F) << 5

Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
Hours	New Minutes	Seconds ÷ 2

OUTLINE

- Bit manipulation
- **Mixing C and assembly**

INLINE ASSEMBLY: COMPILER DEPENDENT

- DJGPP (C development environment for Intel 80386)
  - `__asm__ __volatile__ {push ebp  mov ebp, esp}`
  - __asm__ instructs the compiler to treat the parameters of the statement as pure assembly and to pass them to the assembler as written
  - __volatile__ optional statement which instructs the compiler not to move opcodes around
- Microsoft C

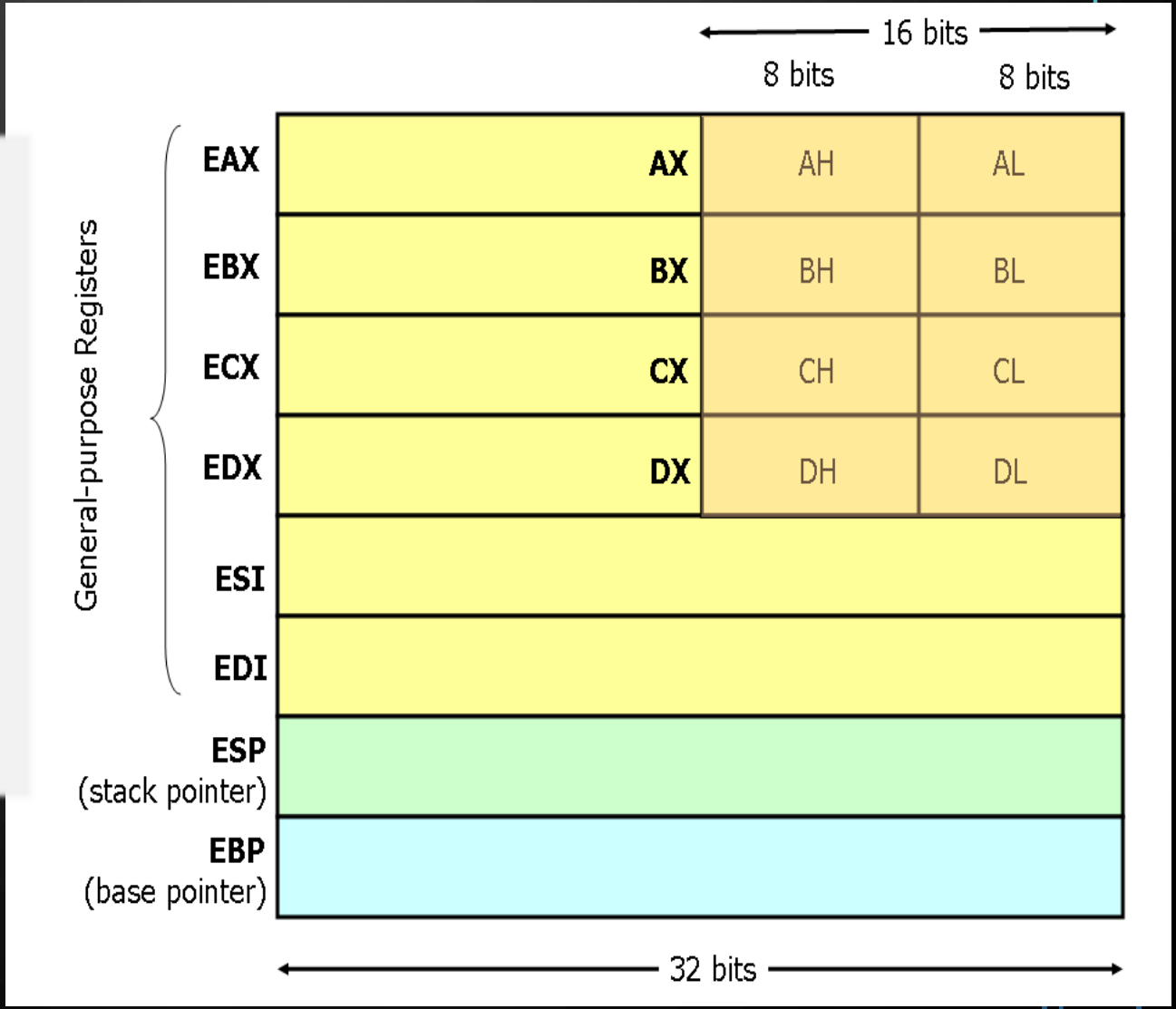
```
__asm push ebp
__asm mov ebp, esp


__asm {
 push ebp
 mov ebp, esp
}
```

X86 ASSEMBLER

- Intel syntax
  - Used by NASM assembler (open-source assembler for 16, 32 and 64-bit X86)
  - Opcode *Destination* Source (order as in "Y = X" in C)
  - Hexadecimal (resp. decimal, octal, binary) constants end with "h" (resp. d, o, b) as in 10h (resp. 16d, 20o, 10000b)
  - Operand determine size: byte (8bits), word (16bits), dword (32bits), qword (64bits)
  - Memory addressing like section:[base + index*scale + offset]

X86 ASSEMBLER

```
PUSH    EBP
MOV     EBP, ESP
MOV     ECX, 0
MOV     EAX, [EBP]
MOV     EBX, [EBP + 32]
MUL     EBX
ADD     [ESI + 4], EAX
```



C-ASSEMBLY INTERFACING

- EBP is base pointer (helps us to point to things in stack)
- ESP is current stack pointer
- PUSH send contents to top of stack
- POP retrieve contents from top of stack

WRITE C FUNCTION IN ASSEMBLY

```
print(const char* str, int size);
```

```
print:
 PUSH EBP ; save previous stack frame
 MOV EBP, ESP ; save current stack frame
 MOV ECX,[EBP+8] ; read parameter 'str'
 MOV EDX,[EBP+12] ; read parameter 'size'

; do function stuff here

 MOV ESP, EBP ; restore stack frame
 POP EBP ; restore previous frame
 RET
```

CALLING CONVENTION

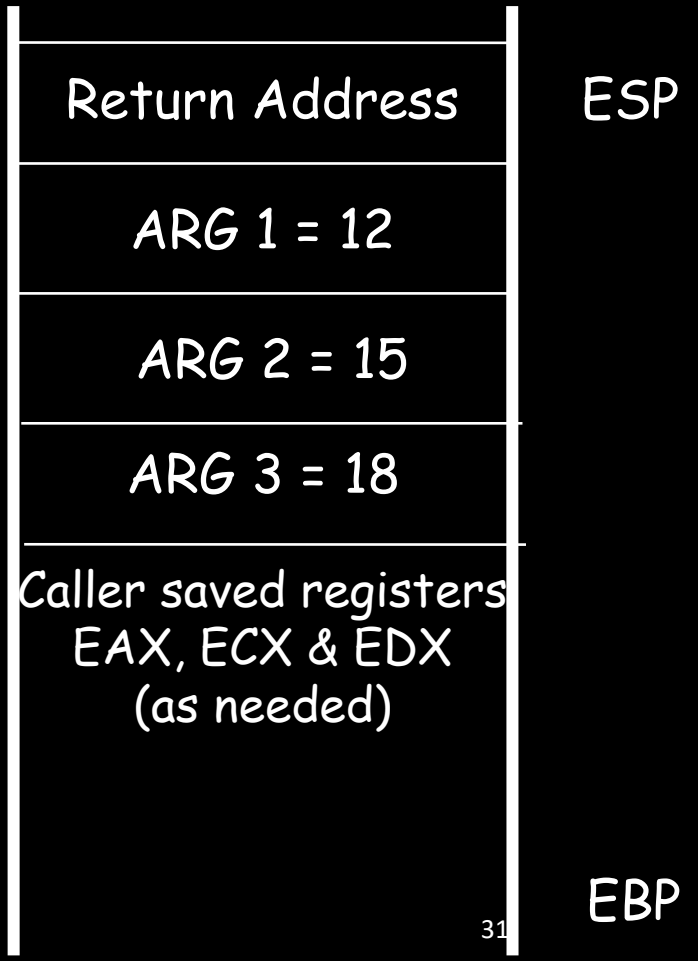
- We want to serve different procedures one after another
- When a caller calls a procedure, the program must follow some steps
  - Put parameters in a place in memory so that the callee (i.e., the called procedure) may access them
  - Transfer control to the callee
  - Acquire space on memory to store local variables, if needed
  - Do computation
  - Return control to point of origin in caller
- Calling convention is the set of rules that compilers and programmers must follow to achieve the above

CALLER (BEFORE IT CALLS)

- Example,
  - Caller is the main function
  - It will call a function called foo
  - a = foo(12, 15, 18)

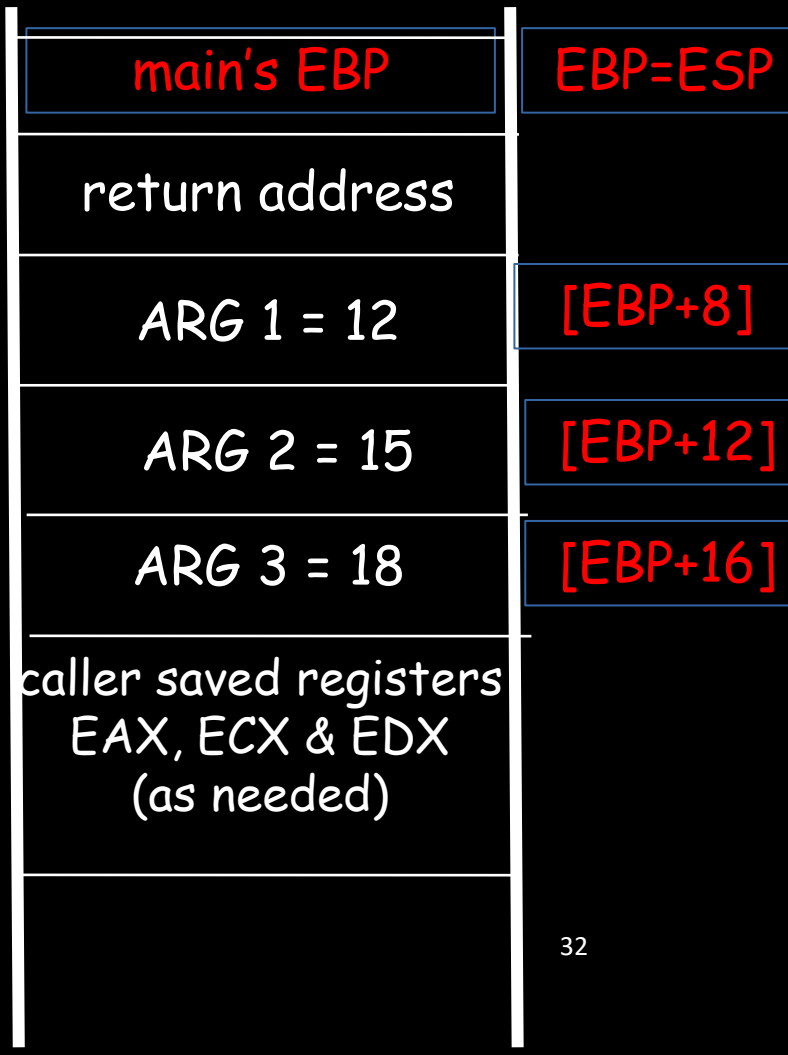
JUST BEFORE THE CALL

- Main (caller) is using ESP and EBP
- First, main pushes, EAX, ECX and EDX (only if they need to be preserved)
- Next, it pushes the arguments (last argument first) i.e., for "a = foo(12, 15, 18)":
  - push dword 18
  - push dword 15
  - push dword 12
- Then, the return address (contents of register EIP, i.e., the program counter) is pushed
- Finally, control transferred to foo



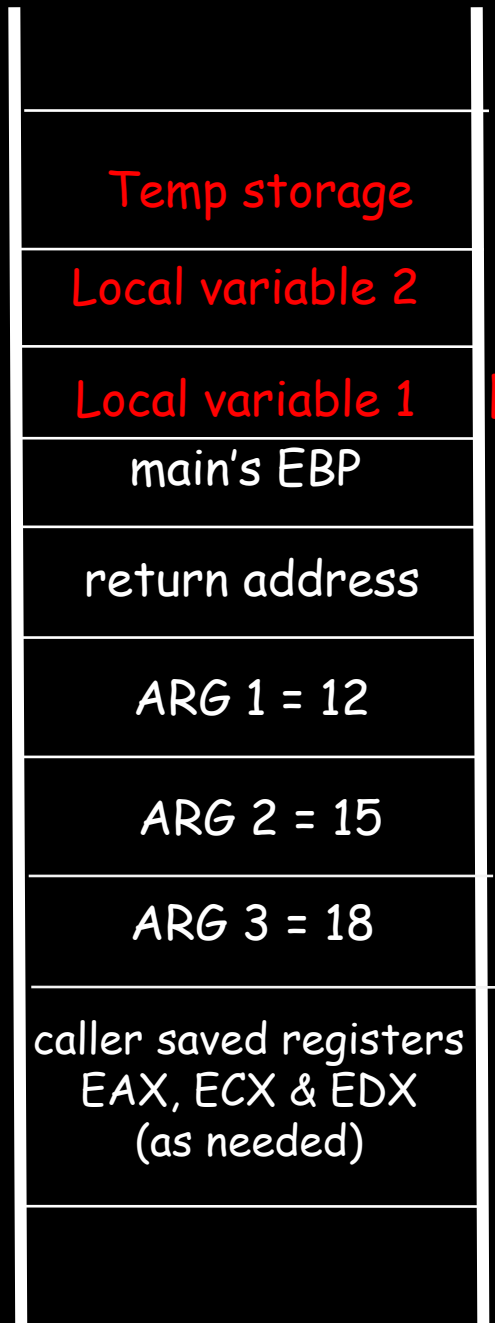
JUST AFTER THE CALL

- First, foo (callee) must setup its own stack frame. EBP register was pointing to somewhere in main's stack frame. This must be preserved. So, we push it.
- Then, the content of ESP is copied to EBP. ESP is freed to do other things and EBP is now the base pointer for foo. So, we can point to things in foo's stack with an offset from EBP
- The above two steps are:
  - push EBP
  - mov EBP, ESP
- 4B for main's EBP and 4B for return address. That's why 8B offset to first argument.



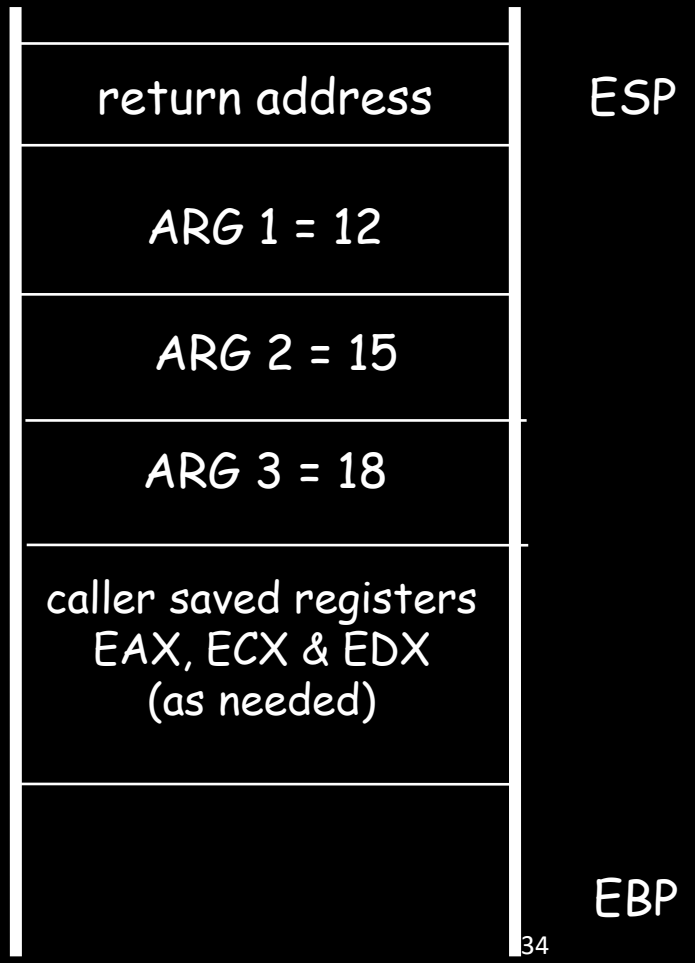
CALLEE (AFTER IT WAS CALLED)

- Foo must allocate space for temporary variables that cannot be stored in registers as well as other storage memory
- If, e.g, foo has 2 variables of type int (4B), plus it needs 12B. Total = 20 bytes.
- Allocate 20B in the stack, by adjusting ESP
  - sub esp, 20
- Finally, foo must preserve EBX, ESI and EDI if they are being used
- Now, foo can be executed. ESP can go up and down but the EBP will remain same.
- Maybe other functions are called but always EBP is restored.



CALLEE (BEFORE IT RETURNS)

- Store the computed (return) value in EAX
- Restore values of EBX, ESI and EDI, if needed.
- Now, release area used for local storage and temporary registers spillover. Then, pop the return address of main to EBP
  - mov ESP, EBP
  - pop EBP
- Finally, just return
  - ret



CALLER (AFTER RETURNING)

- Control returns to caller (main)
- The arguments passed to foo are not needed any more, so adjust ESP
  - `add ESP , 12`
- Save the EAX (returned value) in appropriate memory location
- Main pops, EAX, ECX and EDX (only if they were preserved before the call)

NEXT LECTURE

- Pointers
- Structures
- Unions
- Endianness
- Bitfield