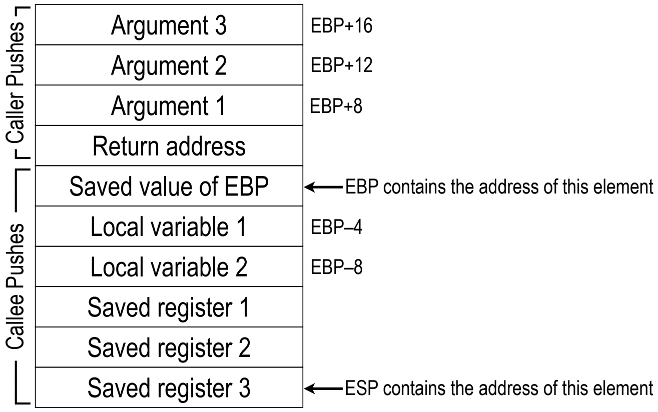
ACTIVITY 11



Creating a Stack Frame	Terminating a Stack Frame
At the call site (i.e., inside the calling function)... 1. Push arguments onto the stack 2. Call the subroutine (CALL pushes the return address) **Inside the function being called...** 3. Push EBP (it will be used to retrieve the arguments) 4. Set EBP equal to ESP 5. Decrement ESP to allocate stack storage for locals 6. Save register values by pushing them on the stack	**Inside the function being called...** 1. If the function returns a value, put it in EAX 2. Pop register values off the stack 3. Set ESP equal to EBP to remove local variables 4. Pop EBP 5. Return (RET pushes the return address); if using the *STDCALL calling convention,* remove arguments by supplying an immediate operand to the RET instruction **Back in the calling function...** 6. If using the *C calling convention,* remove arguments

1. *(Review)* The following code demonstrates a horrible abuse of the `push`, `pop`, `call` and `ret` instructions: Manipulating the values on the stack results in extremely unintuitive interprocedural control flow. Trace through the program, keeping track of the stack contents. What does it output?

```
main PROC              ; 1
    push OFFSET x      ; 2
    call foo           ; 3
    mov eax, 1         ; 4
    call WriteDec      ; 5
    ret                ; 6
x:  mov eax, 2         ; 7
    call WriteDec      ; 8
    exit               ; 9
main ENDP              ; 10
```

```
foo PROC               ; 11
    mov eax, 3         ; 12
    call WriteDec      ; 13
    push OFFSET bar    ; 14
    ret                ; 15
foo ENDP               ; 16

bar PROC               ; 17
    pop eax            ; 18
    call eax           ; 19
    mov eax, 4         ; 20
    call WriteDec      ; 21
    ret                ; 22
bar ENDP               ; 23
```

```
void main() {
    divide(20, 5);
}

int divide(int n1, int n2) {
    int quotient = n1 / n2;       // quotient and remainder are local variables
    int remainder = n1 % n2;
    return quotient;
}
```

1. Translate the above code into assembly, using the STDCALL calling convention. The *divide* procedure should take two signed integer arguments (passed on the stack), store their quotient and remainder in local variables, and then return the value of the *quotient* variable. The *main* procedure should call *divide(20, 5),* which should return 4. (Note: Storing the values in local variables is pointless; it's for illustration only.)

```
include Irvine32.inc

.code
main PROC




    exit
main ENDP

divide PROC




divide ENDP
```

2. Identify the prologue and epilogue in the *divide* procedure.

3. What lines of assembly would change if the *divide* procedure used the C calling convention instead?