Function calls and parameter passing.

A simple C program is shown below. The program's main function calls on a function F twice and passes two parameters to it.

```
int a,b,c;
int F(int V1, int V2) {
      return V1 & V2;
}
int main()
{
      b = 1;
      c = 3;
      a = F(b,c);  // Call number 1
      a = F(2,15); // Call number 2
      return 0;
}
```

Looking at the function calls, you might ask the following questions:

How does the main program actually send values to F?
How does F know where to return to when it is finished?

The answers to both of these questions involve the stack.

Parameter passing

There are a number of ways that part of a program can send data to other parts. Both parts could share a set of (global) memory locations. Values could be passed by register or alternatively values could be passed via the stack.

Suppose in the case of function F above, the compiler decided that F would always receive its parameters via memory locations 1000 and 1004. If another part of the program needs to call on F, it first places the values to be operated on in those memory locations and then calls F. On the surface this seems fine, however, this mechanism may not work in the case of recursive or nested calls to the same function.

OK, so how about using the registers to pass values around? As it happens compilers will actually do this if there are enough general purpose registers available (The 64bit x86 compilers generate now pass the first 4 parameters via processor registers). In the general case however where there are lots of parameters and few processor registers, this approach is unworkable.

The last option is to use the stack. Just prior to calling a function, a program will save (push) copies of the function parameters to the stack. The called function knows then expects that the variables are at a certain offset relative to the current value of the stack pointer. This is the default approach taken by most C-compilers.

The relevant sections of the compiler output (gcc) for this program are shown below.

```
Section of main
Line  Address
1     0x804833d   movl   $0x1,0x8049564   # b = 1;
2     0x8048347   movl   $0x3,0x8049568   # c = 3;

The function call :           a = F(b,c);
3     0x8048351   mov    0x8049568,%eax   # eax = c
4     0x8048356   mov    0x8049564,%edx   # edx = b
5     0x804835c   push   %eax             # Push Value of c on to stack
6     0x804835d   push   %edx             # Push Value of b on to stack
7     0x804835e   call   0x8048324        # call F
8     0x8048363   add    $0x8,%esp        # free up stack space consumed by b,c
9     0x8048366   mov    %eax,0x8049560   # store result to a


int F(int V1, int V2)
Address
            # Standard gcc function entry code i.e. the opening '{'
10    0x8048324   push   %ebp             # save base pointer register
11    0x8048325   mov    %esp,%ebp        # copy stack pointer to base pointer

12    0x8048327   mov    0xc(%ebp),%eax   # read parameter from stack
13    0x804832a   and    0x8(%ebp),%eax   # and with other, result in eax (V1 & V2)
      # Function returns the result in eax

      # Standard gcc function exit code i.e. the closing '}'
14    0x804832d   pop    %ebp       # restore base pointer
15    0x804832e   ret                   # go back to caller
```

Note the call to F on line 7. Just prior to the call, the registers eax and edx are pushed (placed) on the stack. Register eax contains a copy of the data in the variable 'c' while edx contains a copy of 'b'. Inside F, the first thing that happens is that the base pointer register (ebp) is saved to the stack. Why do this? When Intel designed the 8086 processor it chose to include hardware support for stack based parameters. This support came originally in the form of the bp register (the ebp register is a 32 bit equivalent of the older 16 bit bp register). Instructions that fetch data from memory can use a register as a pointer. In addition to this, a constant value can be added to this pointer so that data can be read relative to some location pointed to by the register. When performing such an instruction, the processor must calculate the actual address involved by adding the constant to the pointer register. This takes a certain length of time. At the time of original 8086 Intel was unable to provide “hardware acceleration” for address calculations for all registers in the processor due to cost. Instead it provided hardware address calculation support for only some of the registers : bp, si and di (ebp, esi and edi in the 32 bit world). The si and di registers were primarily designed for optimizing data processing of large contiguous blocks of data (e.g. Strings). The bp register was therefore left as the favoured register for data movements to and from a stack frame. On entry to a function the general approach is a follows:

Save the ebp register because it is going to be changed.

Copy the esp (stack pointer) into ebp so ebp now points to the stack.

Read data from the stack by using pointers of the form ebp+N where N is some constant offset value.

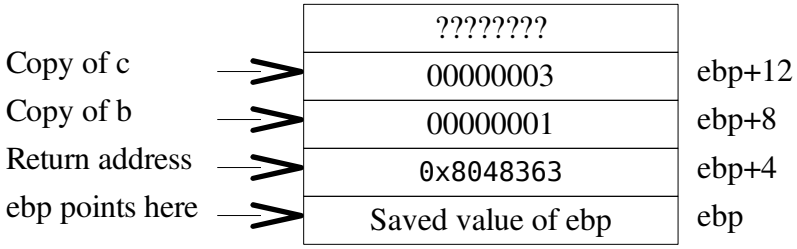
You can see this take place on lines 10 and 11. On lines 12 and 13, data is read from the stack. Taking a closer look at line 12:

```
mov    0xc(%ebp),%eax
```

This instruction means:

"Take the contents of memory address (ebp+12) and place it in eax."

The contents of the stack just prior to execution of line 12 are shown below:



As can be seen, ebp points to the last value pushed on the stack (the original value of ebp is there). The next value on the stack is the address of the location the program must return to after completing function F. Next comes a copy of the value of b and finally a copy of the value of c. The C calling convention pushes values on to the stack in the order of rightmost first (hence c being pushed first, followed by b). The gcc c-compiler generates code for function calls and for accessing parameters within functions that adheres to this calling convention. The offsets of parameters relative to the top of the stack are known when the compiler generates machine code so parameters are in no danger of being mixed up or being misplaced.

You may notice that the stack seems upside down in the above. The convention for 80x86 processors is that the stack is an upside-down stack. The stack pointer register (esp) points to the last value placed on the stack. Whenever a value is pushed on to the stack using the push instruction or as a result of a call instruction (where the return address is pushed) the stack pointer is decremented by the appropriate amount (4 bytes for 32 bit pushes). The stack therefore grows from high memory down towards lower memory. When values are taken from the stack using a pop instruction or a ret instruction, the stack pointer is incremented i.e. the stack shrinks back upwards.

The function F performs the calculation 1&3, the result ending up in eax.

On line 14, the register ebp is restored to its original state (it value is popped from the stack)

On line 15, the ret(urn) instruction is executed. This instruction pops the instruction pointer (eip) from the stack. This is equivalent to a jump back to line 8.

On line 8. the program adjusts esp such that the space used by the two parameters (b and c) is reclaimed.

Finally on line 9, the function's return value is stored back to the variable a. More technically speaking, the value in eax is copied to the memory location pointed to by the symbol 'a'. The symbol 'a' in this case is equivalent to the number 0x8049560.

Note: The C- calling convention on 80x86 machines returns single values from functions via the eax register.