The xbasic VM is a stack machine. Most instructions take their arguments from the stack and return their values to the stack. The implementation uses a variable called TOS that is the value on the top of the stack. This speeds up some stack operations by eliminating a hub access for references to the top element on the stack.

Pushing an element onto the stack is done by

1) Subtract 4 from SP
2) Set SP[0] = TOS
3) Set TOS to the value being pushed

Popping an element from the stack is done by

1) Set TOS to SP[0]
2) Add 4 to SP

All of the arithmetic opcodes expect two operands on the stack. They remove the two operands and replace them with the result of the operation. This includes these opcodes:

ADD, SUB, MUL, DIV, REM, BNOT, BAND, BOR, BXOR, SHL, SHR, LT, LE, EQ, NE, GE, LT

The LIT opcode is followed by a 32 bit literal value which it pushes onto the stack.

The SLIT opcode is followed by a signed 8 bit literal value which it pushes onto the stack.

The LREF opcode is followed by a signed offset byte. It multiplies that offset by 4 and adds it to the current value of FP and then pushes the value at that address onto the stack. "LREF 0" refers to the first function argument, "LREF 1" refers to the second, etc.

The LSET opcode is followed by a signed offset byte. It multiplies that offset by 4 and adds it to the current value of FP and then pops the top element off the stack and stores it into that location.

Source code:

```
def waitpeq(state, mask)
  asm
    lref 1              // get mask
    native 0xa0bc0805   // mov t4, tos
    drop
    lref 0              // get state
    native 0xf03c0a04   // waitpeq tos, t4
    returnx
  end asm
end def
```

Here is what the xbasic compiler generates from this code:

```
waitpeq:
00000020 25 01          FRAME 01
00000022 1f 01          LREF 1
00000024 2a a0 bc 08 05 NATIVE a0bc0805
00000029 28             DROP
0000002a 1f 00          LREF 0
0000002c 2a f0 3c 0a 04 NATIVE f03c0a04
00000031 26             RETURN
00000032 27             RETURNZ
arguments:
  L I 00000000 00000000 state
  L I 00000001 00000000 mask
```

On entry to any function, the return address is in TOS and the function parameters are on the stack. In the case of the waitpeq function, the stack will look like this:

```
SP[1] = value of the mask parameter (second parameter)
SP[0] = value of the state parameter (first parameter)
TOS   = return address
```

You'll notice that every function begins with a FRAME instruction. This sets up the stack frame so that you can use the LREF and LSET instructions to access both function parameters and local variables. What it actually does is this:

1. save the current value of fp (pointer to the current stack frame) in t2
2. set fp to sp to create a new stack frame
3. add the byte that comes after the FRAME opcode (in this case, 1) multiplied by 4 to sp to reserve space for local variables. In this case there are no local variables but we still need a place on the stack to save the old value of fp
4. store the old value of fp (saved in t2 in step 1) in the stack frame
5. through all of this the return address for the function is in TOS

After execution of the FRAME instruction, the stack looks like this:

```
SP[2] and FP[1]  = value of the mask parameter
SP[1] and FP[0]  = value of the state parameter
SP[0] and FP[-1] = old value of FP
TOS              = return address
```

The RETURN instruction undoes all of this except that it expects the return value of the function to be in TOS. At the point where the RETURN instruction executes the stack looks like this:

```
SP[3] and FP[1]  = value of the mask parameter
SP[2] and FP[0]  = value of the state parameter
SP[1] and FP[-1] = old value of FP
SP[0]            = return address
TOS              = return value of function
```

1. Restores the PC from SP[0] which is where it ends up after the function code pushes the PC onto the stack before setting the TOS to the return value
2. Sets the SP to the current value of FP thereby popping the stack frame created by the FRAME instruction off the stack.
3. Restores the FP that was saved during the execution of the FRAME instruction.
4. Throughout all of this the function return value remains in TOS and execution resumes where it left off before the function was called.
5. It is the responsibility of the caller of the function to remove the function arguments from of the stack using the CLEAN instruct. This is to allow for functions with a variable number of arguments.