# Flare-On 5: Challenge 12 Solution - Suspicious Floppy Disk

**Challenge Author: Nick Harbour**

## Background

This challenge is framed as spy tool. You are told that we found a suspicious floppy disk that we suspect was given to spies to transmit secret messages. The spies were given the password but you weren't. You need to figure out the hidden message.

## Running the floopy disk

I recommend using the Bochs emulator for this challenge. You can start up a minimal x86 VM and add this disk image as a bootable floppy drive. Plus, the bochs image is directly loadable and debuggable with IDA Pro. If you load the floppy correctly with bochs you will see the screen shown in Figure 1 below.

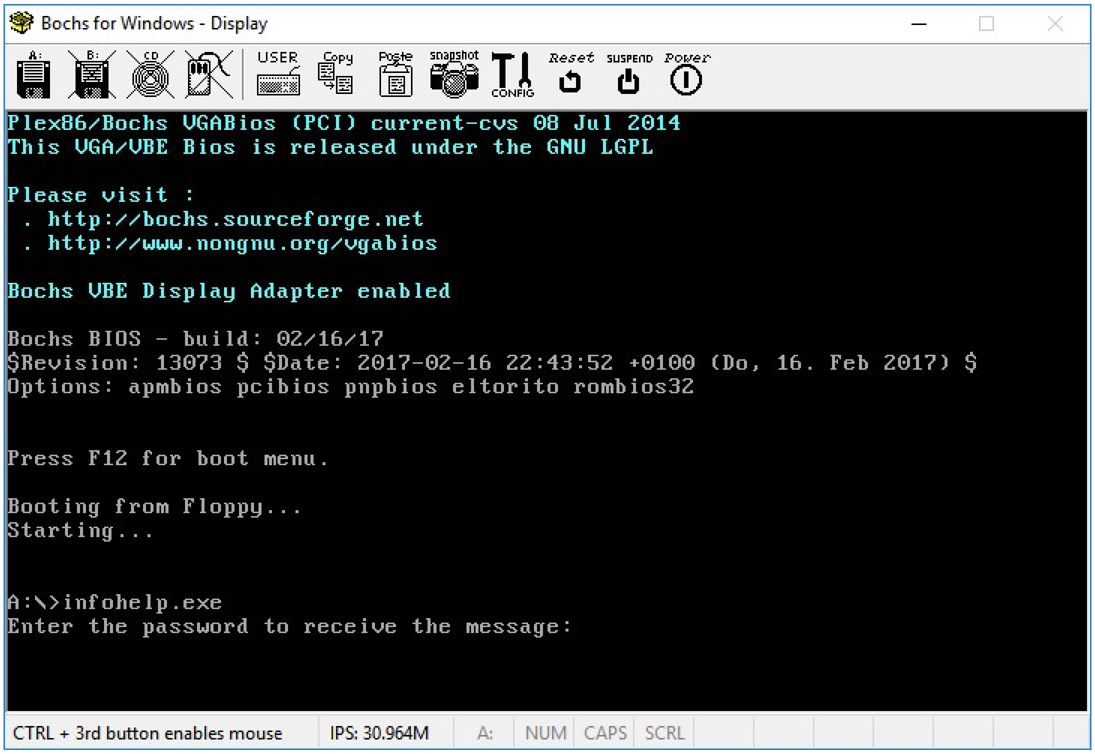

*Figure 1: Bochs Initial Bootup*

Before we attempt to look closer at the contents of the floppy disk we can determine that the program that is launched in the boot sequence that appears to be prompting for the password is named infohelp.exe. If you hit Ctrl-C here it will terminate the batch job (autoexec.bat) and return you to the command prompt. Using the DIR command here will show you the list of files as shown in Figure 2 below.

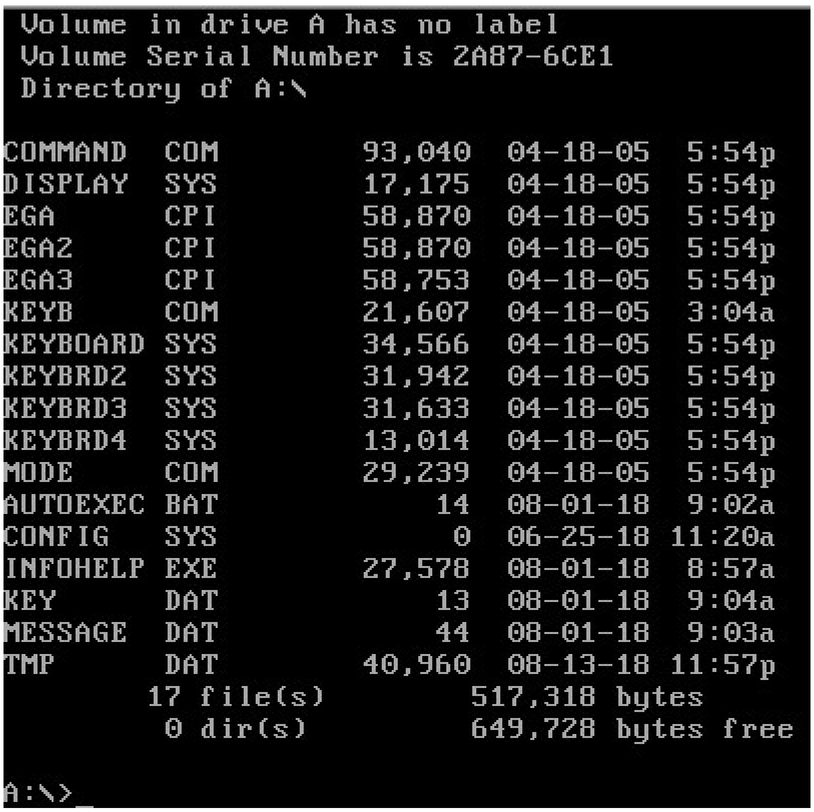


*Figure 2: Floppy disk contents*

You can see from the file timestamps that many of the files on this disk are old. This floppy disk was originally generated with WindowXP using a “format /s” command to create a bootable floppy. All of these files with the 2005 timestamp were placed by the format command and constitute the minimal DOS operating system standard to bootable floppy disks of that era. The files listed with the 2018 timestamps are all related to this challenge. Firstly, the autoexec.bat file which only contains one line, “infohelp.exe”. This is what launches the infohelp program upon boot. The config.sys file is empty. We will examine infohelp.exe further in the next section. The file key.dat contains the string “key goes here". An interesting thing happens with the next file, if you attempt to inspect its contents from the DOS floppy environment. Using the command “type message.dat” draws the text shown in Figure 3 below conspicuously slow.

```
A:\>type message.dat
Welcome to FLARE spy messenger by Nick Harbour.
Your preferred covert communication tool since 1776.
Please wait while I check that password...
Incorrect Password.
You fail it. Please try harder.
This is not the message you are looking for.
A:\>_
```

*Figure 3: Message.dat*

If you examine this file with an external tool or disk editor you will see that the actual file contents are only the last line: “This is not the message you are looking for.” The final file in the floppy is a 40 Kb file named “TMP.DAT” which does not show any printable text when we use the TYPE command against it. Lets examine the infohelp.exe program in more detail to determine how the challenge collects input and how it determines if the input is correct.

## infohelp.exe

This program is a 16-bit DOS program written in C and compiled with the OpenWatcom compiler. IDA Pro does not contain the FLIRT signatures for this compiler, which may hinder your analysis. The first step is usually trying to identify the Main() function. With most C runtime environments, the code at the entry point of the program is not main() but rather a C runtime environment startup function. The main() function is usually one of the last functions called by this code and takes three arguments: argc, argv, and envp in c lingo. At the end of the startup function in this program you will see a call to a function at address 12F4C. The disassembly for this function is provided below in Figure 4.

```
00012F4C
00012F4C
00012F4C ; Attributes: noreturn bp-based frame
00012F4C
00012F4C sub_12F4C proc far      ; CODE XREF: start_0+178↑p
00012F4C inc     bp
00012F4D push    bp
00012F4E mov     bp, sp
00012F50 push    bx
00012F51 push    cx
00012F52 mov     ax, seg dseg
00012F55 mov     ds, ax
00012F57 mov     bx, word_169E2
00012F5B mov     cx, word_169E4
00012F5F mov     ax, word_169E6
00012F62 mov     word_1677C, 2000h
00012F68 push    cs
00012F69 call    near ptr sub_10000
00012F6C db      36h
00012F6C push    cs
00012F6E db      3Eh
00012F6E call    sub_14ED3
00012F6E sub_12F4C endp
```

*Figure 4: Call to Main()*

The function call at line 12F69 represents the call to Main() and the three register mov's starting at 12F57 represent are setting up the argc, argv, and envp arguments for the function. Analyzing the main() function code is the next step.

This function can be daunting because your disassembler may not be efficient at resolving the DOS style pointers. For example, the first printf function call encountered here, which will likely not be labeled as printf as I have shown in Figure 5, does not show the connection to the actual data being printed to the screen. Notice the three pushes on the stack before the printf, the first is a segment register, the second is the number 22h, and the third is a copy of the stack segment selector.

```
00010045 mov     ax, ss
00010047 mov     dx, 22h ; '"'
0001004A push    ax
0001004B push    dx
0001004C push    cs
0001004D call    near ptr _printf
```

*Figure 5: infohelp.exe printf() call*

If we examine the data segment of this program we should see what the value 22h corresponds to. Figure 6 below shows the contents of the data segment and at offset 22h is the beginning of the message “Enter the password to receive the message:”. Similar lookups can be performed for the remainder of the main() function.

```
dseg:0000                          ; Segment type: Pure data
dseg:0000                          dseg            segment para public 'DATA' use16
dseg:0000                                          assume cs:dseg
dseg:0000 01                       unk_15B80       db    1                    ; DATA XREF: start_0+183↑o
dseg:0000                                                                     ; sub_10E9E+B1↑o
dseg:0001 01 01 01 01 01 01+byte_15B81      db 1Fh dup(1), 2 dup(0) ; DATA XREF: sub_11831+150↑o
dseg:0022 45 6E 74 65 72 20+aEnterThePasswordToReceiveTheMessage db 'Enter the password to receive the message:',0
dseg:004D 25 37 39 73 00    a79s            db '%79s',0
dseg:0052 72 62 2B 00       aRb             db 'rb+',0
dseg:0056 6B 65 79 2E 64 61+aKeyDat         db 'key.dat',0
dseg:005E 45 72 72 6F 72 20+aErrorOpeningKeyDat db 'Error opening key.dat',0
dseg:005E 6F 70 65 6E 69 6E+                                                  ; DATA XREF: sub_12AB4:loc_12B40↑o
dseg:0074 45 72 72 6F 72 20+aErrorWritingToKeyDat db 'Error writing to key.dat',0
dseg:008D 72 00             aR              db 'r',0
dseg:008F 6D 65 73 73 61 67+aMessageDat     db 'message.dat',0
dseg:009B 45 72 72 6F 72 20+aErrorOpeningMessageDat db 'Error opening message.dat',0
dseg:00B5 4D 65 73 73 61 67+aMessageS       db 'Message: %s',0Ah,0
```

*Figure 6: Data Segment Contents*

The main() function original source code is provided below in Figure 7.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
	FILE *messagefile;
	FILE *keyfile;
	char messagebuf[512] = {0};
	char input_buf[512] = {0};

    printf("Enter the password to receive the message:");
	scanf("%79s", input_buf);
	keyfile = fopen("key.dat", "rb+");
	if (keyfile == NULL)
	{
		perror("Error opening key.dat");
		return -1;
	}

	if (0 == fwrite(input_buf, sizeof(input_buf), 1, keyfile))
	{
		perror("Error writing to key.dat");
		return -1;
	}
	fclose(keyfile);

	messagefile = fopen("message.dat", "r");
```

```
        if (messagefile == NULL)
        {
                perror("Error opening message.dat");
                return -1;
        }
        fread(messagebuf, 1, 512, messagefile);
        fclose(messagefile);
        printf("Message: %s\n", messagebuf);

        return 0;
}
```

*Figure 7: Main() Source*

This program is nothing spectacular, it simply displays the prompts, reads your input and writes it to the key.dat file. After it writes the key.dat file, it reads the message.dat file and displays it to the screen. If you attempted to interact with this program you would have noticed the same slow text display that happened if you did the "type message.dat" command from the command line as shown in Figure 3. Clearly, this program is not responsible for the slow text display, nor is it responsible for determining the correct password input.

The fact that this was a bootable floppy disk should have raised suspicions about how this challenge was constructed. Another bootkit was introduced in challenge 8, which was not a difficult challenge but was positioned later in the contest to get the players comfortable and setup with bootkit analysis and debugging. This challenge will put those skills to the test. Let's first examine the boot sector of the floppy drive.

## Boot Sector Analysis

The first instruction in the boot sector jumps to the sequence of instructions show in Figure 8 below, which simply copies the current boot sector to memory from address 7C00h (the standard boot sector location) to memory address 600h and jumps to it with a push/ret. This is a common feature of first stage boot sectors since all subsequent boot sectors will also be designed to execute at address 7C00h, it is the responsibility of each boot sector to move itself to another memory location and load and execute the next level boot sector at this address.

```
BOOT_SECTOR:7C3E xor     ax, ax
BOOT_SECTOR:7C40 mov     ss, ax
BOOT_SECTOR:7C42 assume ss:MEMORY
BOOT_SECTOR:7C42 mov     sp, 7BFEh
BOOT_SECTOR:7C45 mov     es, ax
BOOT_SECTOR:7C47 assume es:MEMORY
BOOT_SECTOR:7C47 mov     ds, ax
BOOT_SECTOR:7C49 assume ds:MEMORY
BOOT_SECTOR:7C49 mov     si, 7C00h
BOOT_SECTOR:7C4C mov     di, 600h
BOOT_SECTOR:7C4F mov     cx, 200h
BOOT_SECTOR:7C52 cld
BOOT_SECTOR:7C53 rep movsb
BOOT_SECTOR:7C55 push    662h
BOOT_SECTOR:7C58 retn
```

*Figure 8: Boot Sector Memory Copy*

Following the execution of the boot sector to memory address 662h will show the instructions listed in Figure 9 below.

```
00000662 sub_662 proc near        ; CODE XREF: sub_7C3E+1A↓j
00000662 mov     ax, 230h
00000665 mov     cx, 2201h
00000668 mov     dh, 0
0000066A mov     bx, 800h
0000066D int     13h              ; DISK - READ SECTORS INTO MEMORY
0000066D                          ; AL = number of sectors to read, CH = track, CL = sector
0000066D                          ; DH = head, DL = drive, ES:BX -> buffer to fill
0000066D                          ; Return: CF set on error, AH = status, AL = number of sectors read
0000066F call    bx ; unk_800
00000671 mov     ah, 2
00000673 mov     al, 1
00000675 mov     ch, 0
00000677 mov     cl, 6
00000679 mov     bx, 7C00h
0000067C int     13h              ; DISK - READ SECTORS INTO MEMORY
0000067C                          ; AL = number of sectors to read, CH = track, CL = sector
0000067C                          ; DH = head, DL = drive, ES:BX -> buffer to fill
0000067C                          ; Return: CF set on error, AH = status, AL = number of sectors read
0000067E mov     word_204, 68Eh
00000684 mov     word_206, 0
0000068A push    7C00h
0000068D retn
0000068D sub_662 endp
```

*Figure 9: Boot sector copied stage*

The first int 13h instruction, as documented by IDA Pro here, is used to read sectors from disk into memory. It reads 48 sectors start at head 0, track 34, sector 1 from boot drive. If you examine the structure of the floppy disk you'd find these sectors in allocated space belonging to the file TMP.DAT. it loads this data to memory address 800h then the call instruction at address 66Fh calls into that code. We will examine this next stage of code in the next section but for now we will proceed with examining the remainder of this boot sector.

The remainder of the boot sector contains another disk read operation (via BIOS interrupt 13h). This reads one sector from head 0, track 0, sector 6 to address 7C00h. This represents loading the next stage bootloader to memory and executing it. In this case, since the current boot sector was crafted by the malware author (me) this next operation is to load the original boot sector that was originally built for the floppy disk. If you look at sector 6 in the original floppy image though it will not look like a valid boot sector, though at this stage in the boot cycle a clean boot sector will be loaded into memory. The reason for that will be examined in more detail in the next section.

## Boot Loader Stage 2

The second stage bootloader begins by fetching the available memory value from memory address 413h, this is a built-in value that contains the number of available KB of conventional memory. The program lowers this number by 32 with the instructions on lines 809 and 80C as shown in Figure 10 below.

```
00000800 sub_800 proc near       ; CODE XREF: sub_662+D↑p
00000800 pusha
00000801 push    es
00000802 xor     ebx, ebx
00000805 mov     bx, ds:413h
00000809 sub     bx, 20h ; ' '
0000080C mov     ds:413h, bx
00000810 shl     bx, 6
00000813 mov     word_66F3, bx
00000817 mov     es, bx
00000819 mov     si, 843h
0000081C xor     di, di
0000081E mov     cx, 5EB0h
00000821 cld
00000822 rep movsb
00000824 mov     eax, dword_4C
00000828 mov     es:dword_5E5A, eax
0000082D mov     es:byte_5E6F, dl
00000832 mov     dword_200, eax
00000836 mov     word ptr dword_4C, offset unk_12
0000083C mov     word ptr dword_4C+2, es
00000840 pop     es
00000841 popa
00000842 retn
00000842 sub_800 endp
```
*Figure 10: Beginning of Second Stage Bootloader*

The reason this memory value was decremented is to "reserve" 32kb of upper conventional memory for use by the bootkit code. The instruction on line 810 shifts the available KB value left by 6, which is the same as multiplying the value by 2^6 to produce a segment value for addressing this memory. The new segment value is loaded into the ES segment register on line 817 and offset 843h is loaded into the SI register on line 819. The movsb instruction on line 822 will copy 24,240 bytes of memory from address 819 to this newly reserved area of upper conventional memory.

The instruction at line 824 is loading a 32-bit value from address 4Ch to the eax register. This address is part of the Interrupt Vector table and is the interrupt vector table entry for interrupt 13h, the primary BIOS interface. It then saves a copy of this handler value to memory, along with the DL register (which at this point is still the value of the boot device number). Line 832 saves the int 13h handler to the Interrupt Vector table but in position 80h. With this set, Int 80h would work as an unmodified Int 13h.

Lines 836 and 83C are perhaps the most critical of all lines in this fragment, it sets int interrupt vector table handler for Int 13h to a new value, the first stage in hooking this handler. In the setup used for this solution document, the ES segment was 97C0. After this hook is installed, any BIOS interrupt call will be handled by the function at address 97C0:12 instead of the original BIOS handler. This second stage hook installation routine returns to the main boot sector code on line 842.

## Int 13h Hook Handler

**Error! Reference source not found. Error! Reference source not found.** shows both the beginning of the data copied to upper memory as well as the Int 13h hook handler function at offset 12. The function call at line 97C25 is an unimplemented function that could perform pre-interrupt hook functionality. I ended up performing all the hook handling in this challenge with post interrupt hooking, i.e. performing the original interrupt code then examining and manipulating the data after the original interrupt code. The jmp instruction at line 97C46 will jump to the original interrupt handler in the BIOS. This value was saved to this memory location when the hook was originally installed in the previous function described.

```
debug002:97C00 aNickRules db 'NICK RULES',0
debug002:97C0B db 90h, 90h, 90h, 90h, 90h, 90h, 90h
debug002:97C12
debug002:97C12 ; =============== S U B R O U T I N E
debug002:97C12
debug002:97C12
debug002:97C12 sub_97C12 proc near
debug002:97C12 mov     cs:word_5E66, ax
debug002:97C16 mov     cs:word_5E68, bx
debug002:97C1B mov     cs:word_5E6A, cx
debug002:97C20 mov     cs:word_5E6C, dx
debug002:97C25 call    near ptr unk_7C7E
debug002:97C28 push    eax
debug002:97C2A mov     eax, [esp+4]
debug002:97C30 mov     cs:dword_5E5E, eax
debug002:97C35 mov     ax, cs
debug002:97C37 shl     eax, 10h
debug002:97C3B mov     ax, 4Bh ; 'K'
debug002:97C3E mov     [esp+4], eax
debug002:97C44 pop     eax
debug002:97C46 jmp     cs:dword_5E5A
debug002:97C46 sub_97C12 endp
```

*Figure 11: Int 13h Hook Handler*

In the event of an Int 13h, the jmp on line 97C46 will call the original BIOS int 13h handler. For example, if the operation was to read a sector from the disk, this is what will cause the sector to be read. The instructions including and leading up to line 97C3E will build a return pointer on the stack so that when the original BIOS int 13h handler returns it will return to a function in this upper memory region instead of the location that initiated the interrupt. This is how the program will gain control and manipulate date post-interrupt.

The memory address of the post-interrupt handler code that it builds on the stack is the address immediately following the jmp instruction on line 97C46, in this case it is address 97C4B. This code is shown below in Figure 12.

```
97C4B jb      short loc_7C55
97C4D mov     cs:byte_5E6E, 0
97C53 jmp     short near ptr unk_7C5B
97C55 ; -----------------------------
97C55 mov     cs:byte_5E6E, 1
97C5B sub     esp, 6
97C5F call    near ptr unk_7C7F
97C62 push    eax
97C64 mov     eax, cs:dword_5E5E
97C69 mov     [esp+4], eax
97C6F mov     al, cs:byte_5E6E
97C73 test    al, al
97C75 pop     eax
97C77 jz      short near ptr unk_7C7C
97C79 stc
97C7A jmp     short near ptr unk_7C7D
97C7C ; -----------------------------
97C7C clc
97C7D iret
```

*Figure 12: Post-Interrupt return code*

The first few lines of this code are simply to save the state of the carry flag so that it can be restored prior to returning to the original program. This is necessary because BIOS int 13h uses the carry flag as an error indicator. The call instruction at line 97C5F is where the post-interrupt hook handler is called. In this case IDA Pro lost track of the CS segment and erroneously lists address 7C7F as the function location but it is a relative call instruction and in this run the memory address of the function called at this point is 97C7F.

The post-interrupt handler is lengthy and instead of walking through it line-by-line I will summarize its behavior and provide the fragment of source assembler with comments in Figure 13 below. The post_request_handler function checks the AH register to see if it is 02h, 03h, 0Ah, or 42h which correspond to the Int13h functions it is interested in hooking (disk sector read, write, long read, extended read). If it is a write sector operation and the sector matches the key sector (absolute sector #1221, which is the sector behind the KEY.DAT file on disk) then it will call a function to process the key input which just makes a copy of the key input to memory for later use.

The handler has three special cases for handling reads. First case is when attempting to read the original MBR boot sector (stored at sector number 6). Before returning to the code that was requesting the sector it modifies the sector data in memory by performing a Roll Right by 1 bit on every byte in the sector. This is why the sector on disk did not appear to be a valid boot sector, it was ROL 1 encoded.

The second special case we will discuss for sector read handling is the sectors comprising the second stage bootloader code itself. This second stage bootloader does contain printable strings (such as “NICK RULES” at the start and end) but when we printed it to the console with the TYPE Command earlier it did not show anything. This is because this post-interrupt request handler is wiping the data from memory if those sectors are read from disk.

The third special case is reading the message sector. When we typed the MESSAGE.DAT file on the command line at the beginning of this solution it took a long time to print out text to the screen. That is because the code that is called in the post-request interrupt handler when that sector is read does a lot of work and it takes the computer a long time to work through all the code required to print out those messages. The message sector read handler will be covered in the next section of this document.

```
post_request_handler:
    pusha
    mov     al, byte byte [cs:CALC_ADDR_uppermemory(_data_saved_cf)]
    test    al, al
    jnz     error_occured

    mov     ax, word [cs:CALC_ADDR_uppermemory(_data_saved_ax)]
    cmp     ah, 02h
    je      case_normal_read_sectors
    cmp     ah, 03h
    je      case_normal_write_sectors
    cmp     ah, 0Ah
    je      case_long_read_sectors
    cmp     ah, 42h
    je      case_extended_read_sectors
    jmp     cleanup_and_leave

    case_normal_write_sectors:
        mov     cx, word [cs:CALC_ADDR_uppermemory(_data_saved_cx)]
        mov     dx, word [cs:CALC_ADDR_uppermemory(_data_saved_dx)]
        mov     bx, word [cs:CALC_ADDR_uppermemory(_data_saved_bx)]

        ; bail if the read wasn't the boot drive
        cmp     dl, byte [cs:CALC_ADDR_uppermemory(_data_boot_drive_number)]
        jne     cleanup_and_leave
```

```
    ;*** check for special non-encoded sectors

    ; first check for key sector (1221)
    cmp     cx, 2110h
    jne     not_key_sector
    cmp     dh, 1
    jne     not_key_sector
    cmp     al, 1
    jne     not_key_sector

    ; we got us a key sector request (write)

    call    process_key_input
    jmp     cleanup_and_leave

not_key_sector:
    jmp     cleanup_and_leave

case_normal_read_sectors:
    mov     cx, word [cs:CALC_ADDR_uppermemory(_data_saved_cx)]
    mov     dx, word [cs:CALC_ADDR_uppermemory(_data_saved_dx)]
    mov     bx, word [cs:CALC_ADDR_uppermemory(_data_saved_bx)]

    ; bail if the read wasn't the boot drive
    cmp     dl, byte [cs:CALC_ADDR_uppermemory(_data_boot_drive_number)]
    jne     cleanup_and_leave

    ;*** check for special non-encoded sectors

    ; first check for encoded original boot sector
    cmp     cx, 0006h
    jne     not_original_mbr_sector
    cmp     dh, 00h
    jne     not_original_mbr_sector
    cmp     al, 1
    jne     not_original_mbr_sector

    ; we got us an original boot sector request
    mov     cx, 200h
    call    original_mbr_decode

    jmp     cleanup_and_leave

    not_original_mbr_sector:

    ; check for the message sector (1222)

    cmp     cx, 2111h
    jne     not_message_sector
```

```
    cmp     dh, 1
    jne     not_message_sector
    cmp     al, 1
    jne     not_message_sector

    ; we got us a message sector request

    call    handle_special_message_sector_read
    jmp     cleanup_and_leave

    not_message_sector:

    ; check for second stage bootloader sectors

    push    ax
    push    bx
    push    dx
    movzx   ax, ch
    shl     ax, 1
    shr     dx, 8
    add     ax, dx
    mov     bx, 18
    mul     bx
    movzx   bx, cl
    add     ax, bx
    dec     ax
    cmp     ax, 1224
    jl      not_stage2_sectors
    cmp     ax, 1427
    jge     not_stage2_sectors
    pop     dx
    pop     bx
    pop     ax
    call    handle_stage2_sectors_read
    jmp     cleanup_and_leave

    not_stage2_sectors:
    pop     dx
    pop     bx
    pop     ax

    normal_read_non_special_sectors:

    jmp     cleanup_and_leave

case_long_read_sectors:
    jmp     cleanup_and_leave

case_extended_read_sectors:
```

```
        cmp     dword [si+0Ch], 0
        jne     cleanup_and_leave
        cmp     dword [si+08h], 7
        jne     extended_read_non_special_sectors
        ; special sector read

        jmp     cleanup_and_leave

        extended_read_non_special_sectors:
        jmp     cleanup_and_leave
```
*Figure 13: Post-Interrupt Handler Source*

## Message Sector Read Handler

This is where this challenge starts to go off the rails in terms of difficulty. Posted below in Figure 14 below is the source for the handle_special_message_sector_read function, which is called when sector 1222h (the sector that holds the contents of the MESSAGE.DAT file) is read.

```
handle_special_message_sector_read:
    ; data read from the sector is stored in, ds:bx. lets mess with it!
    push    es
    push    ds
    push    cs
    pop     es
    push    cs
    pop     ds
    call    BigVM
    pop     ds
    pop     es
    test    ax, ax
    jz      end_of_handle_special_message_sector_read
    ;; mess with the read buffer
    push    ds
    push    es
    push    si
    push    di
    push    cx
    mov     cx, over_ovaltine-ovaltine
    call    over_ovaltine
ovaltine:
db  'BE SURE TO DRINK YOUR OVALTINE',0
```

```
over_ovaltine:
    pop     si
    push    cs
    pop     ds
    mov     di, bx
    repne   movsb

    pop     cx
    pop     di
    pop     si
    pop     es
    pop     ds
end_of_handle_special_message_sector_read:
    ret
```
*Figure 14: Special Message Sector Read Handler*

This function calls a single function named BigVM, then if that function returns a non-zero value in AX, it will overwrite the buffer that holds the contents of the sector read from disk with the string "BE SURE TO DRINK YOUR OVALTINE". This is a reference to the hit 1983 movie A Christmas Story. If you entered the correct key, the BigVM function would return true and this is the secret message that would be displayed to the "spy" in this scenario.

## Subleq VM

The BigVM function mentioned in the assembler source in Figure 14 above was loaded at address 97DDA in the Bochs VM instance used for this solution. Reverse engineering this function you will see a series of calls, the bottom of which resides at offset 9D99D in this solution. It is a straightforward function that uses a BP based stack frame. If you participated in last year's challenge then you may recognize the construct represented by this function, which is called repeatedly over a large array by the BigVM function: A SUBLEQ one-instruction set computer mini-VM. Unlike last year's, this one is a 16-bit implementation and is implemented to run entirely in the interrupt handler.

The Subleq architecture is well described online, but to sum it up briefly it is built around only one instruction: "SUBtract and Branch if Less than or EQual to". Each 3 values in the array of 16-bit values represent one "instruction", though branching can branch to any value in the array so it isn't easily disassemblable without some degree of tracing, emulation, or other analysis of the instructions as they are disassembled. To build a working program in this architecture I developed a macro assembler and a large set of macros to build logic and programs up from this humble instruction. The complete macro package is provided in Appendix A: macros.subleq and Appendix B: macros16.subleq. This is the same macro package that was used to build last year's challenge (#10). I am not providing the macro assembler I constructed but if you are interested in building your own subleq or rssb OISC programs it would be a straight-forward program to develop and the macro packages I've developed in these appendices are a significant step toward making these architectures usable for program development.

Figure 15 contains the entire Subleq program source code for the program executed by the subleq VM in the interrupt handler. Understanding the translation of this heavily macro-laden code to individual subleq instructions requires examining the individual macros in Appendix A: macros.subleqand Appendix B: macros16.subleq.

The Subleq program implements another one-instruction set computer architecture mini-vm: RSSB (Reverse Subtract and Skip if Borrow). The next section of this document will examine the RSSB VM instructions.

```
.include "macros16.subleq"

Z:      dd 0        // do not change this
INPUT:  dd 0        // do not change this
OUTPUT: dd 0        // do not change this
INPUT_READY: dd 0   // do not change this
OUTPUT_READY: dd 0  // do not change this

start:

    PUSH_LITERAL(rssb_progy_end-rssb_progy, sp)
    PUSH_PTR_REF(rssb_progy, sp)
    CALL(rssb_vm_run, sp)
    POP_IGNORE(2, sp)

    HALT()

stack:  dd 0,0,0,0,0,0
sp:     dd stack
```

```
FUNCTION(rssb_vm_run, sp)
        DECLARE_FUNCTION_ARGUMENT(_program,-2)
        DECLARE_FUNCTION_ARGUMENT(_length,-3)
        DECLARE_VARIABLE(_ip, 0)
        DECLARE_VARIABLE(_progptr, 0)
        DECLARE_VARIABLE(_currop, 0)
        DECLARE_VARIABLE(_halt_code, -2)
        DECLARE_VARIABLE(_rssb_input, 3)
        DECLARE_VARIABLE(_rssb_output, 4)
        DECLARE_VARIABLE(_rssb_input_ready, 5)
        DECLARE_VARIABLE(_rssb_output_ready, 6)
        DECLARE_VARIABLE(_tmp, 0)
        DECLARE_VARIABLE(_one, 1)
        DECLARE_VARIABLE(_zero, 0)
        DECLARE_VARIABLE(_original_start, 0)

        // save start IP so code can be reentrant
        MOV(_program, _progptr)
        DEREF_SRC_MOV(_progptr, _original_start)

        //  while (program[RSSB_IP] < length && program[program[RSSB_IP]] !=
RSSB_HALT)
        _vmloop_start:
            MOV(_program, _progptr)
            DEREF_SRC_MOV(_progptr, _ip)
            BGEQ(_ip, _length, _vmloop_end)
            ADD(_ip, _progptr)
            DEREF_SRC_MOV(_progptr, _currop)
            BEQ(_currop, _halt_code, _vmloop_end)
            // rssb(program);
            PUSH(_program, sp)
            CALL(rssb_insn, sp)
            POP_IGNORE(1, sp)

            // if (program[RSSB_OUTPUT_READY] == 1)
            MOV(_program, _progptr)
            ADD(_rssb_output_ready, _progptr)
            DEREF_SRC_MOV(_progptr, _tmp)
            BNEQ(_tmp, _one, _past_output)
                DEREF_DST_MOV(_zero, _progptr) // program[RSSB_OUTPUT_READY] = 0;
                MOV(_program, _progptr)
                ADD(_rssb_output, _progptr)
                DEREF_SRC_MOV(_progptr, _tmp)
                OUT(_tmp)
                DEREF_DST_MOV(_zero, _progptr) // program[RSSB_OUTPUT] = 0;
            _past_output:

            // if (program[RSSB_INPUT_READY] == 1)
```

```
            MOV(_program, _progptr)
            ADD(_rssb_input_ready, _progptr)
            DEREF_SRC_MOV(_progptr, _tmp)
            BNEQ(_tmp, _one, _past_input)
                DEREF_DST_MOV(_zero, _progptr) // program[RSSB_INPUT_READY] = 0;
                IN(_tmp)
                MOV(_program, _progptr)
                ADD(_rssb_input, _progptr)
                DEREF_DST_MOV(_tmp, _progptr) // program[RSSB_INPUT] = inputval;
            _past_input:

        JMP(_vmloop_start)
        _vmloop_end:

        // restore original rssb IP start
        MOV(_program, _progptr)
        DEREF_DST_MOV(_original_start, _progptr)

        RET(sp)
END_FUNCTION()

FUNCTION(rssb_insn, sp)
        DECLARE_FUNCTION_ARGUMENT(_program,-2)
        MOV(_program, _progptr)
        DEREF_SRC_MOV(_progptr, _ip)  //  ip = program[0]
        ADD(_ip, _progptr)
        DEREF_SRC_MOV(_progptr, _current_operand) // curop = program[ip]
        //  if (program[ip] == RSSB_NOP)
        BNEQ(_current_operand, _negone, _not_nop)
            MOV(_program, _progptr)
            INC(_ip)
            DEREF_DST_MOV(_ip, _progptr)
            RET(sp)
        _not_nop:
        //  program[RSSB_ACC] = program[program[ip]] - program[RSSB_ACC];
        MOV(_program, _progptr)
        INC(_progptr)
        DEREF_SRC_MOV(_progptr, _acc)

        MOV(_program, _progptr)
        ADD(_current_operand, _progptr)
        DEREF_SRC_MOV(_progptr, _curop_cellval)
        MOV(_curop_cellval, _tmp)
        SUB(_acc, _tmp)
        MOV(_tmp, _acc)
        MOV(_program, _progptr)
        INC(_progptr)
        DEREF_DST_MOV(_acc, _progptr)
```

```
        //  if (program[ip] != RSSB_ZERO) { program[program[ip]] =
program[RSSB_ACC] }
        BEQ(_current_operand, _rssb_zero, _past_rssb_zero)
            MOV(_program, _progptr)
            ADD(_current_operand, _progptr)
            DEREF_DST_MOV(_acc, _progptr)
        _past_rssb_zero:

        DEREF_SRC_MOV(_program, _ip)
        // if (program[RSSB_ACC] < 0) { program[RSSB_IP] += 2 } // skip insn
        BGEZ(_acc, _past_acc_neg)
            INC(_ip)
        _past_acc_neg: // else ip += 1
        INC(_ip)
        DEREF_DST_MOV(_ip, _program)
        RET(sp)

    _acc: dd 0
    _current_operand: dd 0
    _curop_cellval: dd 0
    _progptr: dd 0
    _negone: dd -1
    _rssb_zero: dd 2
    _tmp: dd 0
    _ip: dd 0
END_FUNCTION()

rssb_progy:
.include "rssbprogram_bin.subleq"
rssb_progy_end:
```

*Figure 15: Subleq Program Source Code*

## RSSB VM

The RSSB instruction only uses one operand per instruction unlike Subleq's three operands per instruction. This architecture makes use of an accumulator and only has built in branching to skip a single instruction. Appendix C: rssbmacros.subleq provides an implementation of the macros used to build functioning high-level programs in this obscure architecture. Unlike subleq, much less is code and implementations have been attempted with RSSB. Implementing conditional branching in this architecture drove me to the edge of insanity (see the implementation of the BLZ macro, from whence all other conditional branching is then derived).

The source code for the RSSB Program is provided below in Figure 16. The x86 code in the BigVM function wrote a copy of the key data to the input buffer used by this program. It is surrounded by the words “Magic” and “Box” in memory. Some strings in the program specify letters in the string with single quotes and others use decimal numbers for each digit. This is not an attempt to be cryptic but rather I only added that feature to my assembler after the other strings were already placed in this source code in decimal format. This program performs a computation which is described in the next section of this document and sets the cell labeled “retval” to one if the key matched or not. For performace it first checks to see if the ‘@’ symbol is in the correct place in the string before attempting to check each digit pair in the key. This also was a measure to slightly hinder brute-forcing. Another odd feature you may find in the code is that I injected fake but reasonable looking values after the valid values of the encoded key. These aren’t used by the program at all but if you blindly extracted all they key-looking values from memory you may have accidentally grabbed some of these fake values.

```
.include "rssb_macros.subleq"

IP:     dd start
ACC:    dd 0
ZERO:   dd 0
INPUT:  dd 0
OUTPUT: dd 0
INPUT_READY:    dd 0
OUTPUT_READY:   dd 0

// Welcome to FLARE spy messenger by Nick Harbour.
// Your preferred covert communication tool since 1776.
// Please wait while I check that password...
Prompt: dd 87, 101, 108, 99, 111, 109, 101, 32, 116, 111, 32, 70, 76, 65, 82, 69,
32, 115, 112, 121, 32, 109, 101, 115, 115, 101, 110, 103, 101, 114, 32, 98, 121,
32, 78, 105, 99, 107, 32, 72, 97, 114, 98, 111, 117, 114, 46, 13, 10, 89, 111, 117,
114, 32, 112, 114, 101, 102, 101, 114, 114, 101, 100, 32, 99, 111, 118, 101, 114,
116, 32, 99, 111, 109, 109, 117, 110, 105, 99, 97, 116, 105, 111, 110, 32, 116,
111, 111, 108, 32, 115, 105, 110, 99, 101, 32, 49, 55, 55, 54, 46, 13, 10, 80, 108,
101, 97, 115, 101, 32, 119, 97, 105, 116, 32, 119, 104, 105, 108, 101, 32, 73, 32,
```

```
99, 104, 101, 99, 107, 32, 116, 104, 97, 116, 32, 112, 97, 115, 115, 119, 111, 114,
100, 46, 46, 46, 0


// Password Matched.
// You win it. Congratulations!
Success: dd 13, 10, 80, 97, 115, 115, 119, 111, 114, 100, 32, 77, 97, 116, 99, 104,
101, 100,46, 13, 10, 89, 111, 117, 32, 119, 105, 110, 32, 105, 116, 46, 32, 67,
111, 110, 103, 114, 97, 116, 117, 108, 97, 116, 105, 111, 110, 115, 33, 13, 10, 13,
10, 0

// Incorrect Password.
// You fail it. Please try harder.
Fail: dd 13, 10, 73, 110, 99, 111, 114, 114, 101, 99, 116, 32, 80, 97, 115, 115,
119, 111, 114, 100, 46, 13, 10, 89, 111, 117, 32, 102, 97, 105, 108, 32, 105, 116,
46, 32, 80, 108, 101, 97, 115, 101, 32, 116, 114, 121, 32, 104, 97, 114, 100, 101,
114, 46, 13, 10, 0

dd 'M','a','g','i','c'
special: dd 0
inputbuf: dd
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
end_of_inputbuf:
dd 0,'B','o','x'
stack:  dd 0,0,0,0,0,0,0
sp:     dd stack
retval: dd 0
start:
        PUSH_PTR_REF(Prompt, sp)
        CALL(printstring, sp)
        POP_IGNORE(1,sp)

        PUSH_LITERAL(end_of_inputbuf-inputbuf, sp)
        PUSH_PTR_REF(inputbuf, sp)
        CALL(checkinput,sp)
        GET_FUNCTION_RETURN_VALUE(sp, retval)
        POP_IGNORE(1,sp)

        BNEZ(retval, correct_key)

    incorrect_key:
        PUSH_PTR_REF(Fail, sp)
        CALL(printstring, sp)
        POP_IGNORE(1, sp)
        HALT()
    correct_key:
        PUSH_PTR_REF(Success, sp)
        CALL(printstring, sp)
```

```
        POP_IGNORE(1, sp)
        HALT()


FUNCTION(printstring, sp)
        DECLARE_FUNCTION_ARGUMENT(_string,-2)
        DECLARE_VARIABLE(_char,0)
        LOOP_START(print_loop)
            DEREF_SRC_MOV(_string, _char)
            BNEZ(_char, _print_loop_continue)
            LOOP_BREAK(print_loop)
_print_loop_continue:
            OUT(_char)
            INC(_string)
        LOOP_END(print_loop)
        RET(sp)
END_FUNCTION()

encoded_key: dd 64639, 62223, 62305, 61777, 63622, 62417, 56151, 55765, 57966,
63693, 63849, 55564, 63521, 61825, 63583
end_of_encoded_key:
fake_data: dd 63619, 58017, 62588, 59995, 65019

FUNCTION(checkinput,sp)
        DECLARE_FUNCTION_ARGUMENT(_string,-2)
        DECLARE_FUNCTION_ARGUMENT(_stringsize, -3)
        DECLARE_VARIABLE(_char,0)
        DECLARE_VARIABLE(_char2,0)
        //DECLARE_VARIABLE(_correct_checksum, 53598)
        DECLARE_VARIABLE(_running_checksum, 0)
        DECLARE_VARIABLE(_i, 0)
        DECLARE_VARIABLE(_zero, 0)
        DECLARE_VARIABLE(_tmp, 0)
        DECLARE_VARIABLE(_tmp2, 0)
        DECLARE_VARIABLE(_all_good, 0)
        DECLARE_VARIABLE(_success, 1)
        DECLARE_VARIABLE(_FAILURE, 0)
        DECLARE_VARIABLE(_atsymbol, '@')
        DECLARE_VARIABLE(_neg_105, -105)

        SETZ(_all_good)
        SETZ(_i)
        SETZ(_tmp)
        SETZ(_tmp2)
        SETZ(_running_checksum)
        FORLOOP_VAR_START(_i,_stringsize,checksum_loop)
                MOV(_string,_tmp)
                ADD(_i,_tmp)
                DEREF_SRC_MOV(_tmp,_char)
```

```
            //OUT(_char)
            BEQ(_char,_atsymbol,_hit_the_atsymbol)
            BEQ(_char,_zero,_hit_null_byte)
            ADD(_char,_running_checksum)
            JMP(_checksum_loop_continue)
        _hit_the_atsymbol:
            FORLOOP_BREAK(checksum_loop)
        _hit_null_byte:
            JMP(_return_false)
        _checksum_loop_continue:
    FORLOOP_END(_i,checksum_loop)

    DOUBLE(_i) // 2
    DOUBLE(_i) // 4
    DOUBLE(_i) // 8
    DOUBLE(_i) // 16
    DOUBLE(_i) // 32
    DOUBLE(_i) // 64
    DOUBLE(_i) // 128
    DOUBLE(_i) // 256
    DOUBLE(_i) // 512
    DOUBLE(_i) // 1024
    MOV(_i,_tmp)
    DOUBLE(_i) // 2048
    ADD(_tmp,_i) // 3072

    ADD(_i,_running_checksum)
    //BEQ(_running_checksum,_correct_checksum,_return_true)

    SETZ(_i)
    FORLOOP_START(_i,end_of_encoded_key-encoded_key,_compute_input_pairs_loop)
        //OUT(plus)
        MOV(_string,_tmp)
        MOV(_i,_tmp2)
        ADD(_i,_tmp2)
        ADD(_tmp2,_tmp)
        DEREF_SRC_MOV(_tmp,_char)
        //OUT(_char)
        INC(_tmp)
        DEREF_SRC_MOV(_tmp,_char2)
        //OUT(_char2)
        SUB_LITERAL(32, _char)
        SUB_LITERAL(32,_char2)
        SHL(7,_char2)
        ADD(_char,_char2)
        MOV(_i,_tmp)
        DOUBLE(_tmp) // 2
        DOUBLE(_tmp) // 4
        DOUBLE(_tmp) // 8
```

```
            DOUBLE(_tmp) // 16
            DOUBLE(_tmp) // 32
            ADD(_i,_tmp) // 33

            XOR(_char2,_tmp)
            ADD(_running_checksum,_tmp)

            PTR_REF(encoded_key, _tmp2)
            ADD(_i, _tmp2)
            DEREF_SRC_MOV(_tmp2, _char)
            SUB(_char,_tmp)
            BEZ(_tmp, _good_pair)
        _bad_pair:
            ADD(_i,_all_good)
        _good_pair:
            SUB(_i,_all_good)
        FORLOOP_END(_i,_compute_input_pairs_loop)
        SETZ(special)
        BEQ(_all_good,_neg_105,_return_true)

    _return_false:
        SET_FUNCTION_RETURN_VALUE(_FAILURE)
        JMP(_return)
    _return_true:
        INC(special)
        SET_FUNCTION_RETURN_VALUE(_success)
    _return:
        RET(sp)
END_FUNCTION()
```

*Figure 16: RSSB Program Source Code*

## Key Encoding Scheme

This program, like many Flare-On challenges, does not decode the correct key in memory but rather encodes your input and compares it to the correct key. It does the comparison by maintaining a running subtractive sum of the indices of each matching pair. In the end the sum should be -105 if each pair matched the encoded pair correctly.

The function to compute a value for each two bytes of key input also incorporates the index in the string that those two digits occur. The multiplication value of 3072 was the difficult part of this algorithm to discover from the RSSB code, because due to performance I had to break it down into a series of doublings and additions. Multiplication is complex and expensive in a modern processor but in a ridiculous processor like RSSB running in Subleq built on macros written by an insane Flare Team member, it is apocalyptically slow. The relative simplicity of this key obfuscation technique and the extreme difficulty of reverse engineering it I hope shows some promise for this approach to anti-reverse engineering. Luckily for all the future contestants, will not be revisiting this technique in the coming years of Flare-On.

```
correct_key = "Av0cad0_Love_2018@flare-on.com"

def obfuscate_key(key):
    atplacement = key.index('@')
    pre_domain_portion = key[:atplacement]
    rolling_sum = 0
    for c in pre_domain_portion:
        rolling_sum += ord(c)
    juke = rolling_sum + atplacement * 3072
    print juke
    i = 0
    while i < len(key):
        first = ord(key[i]) - 32
        if len(key) > i + 1:
            second = (ord(key[i+1]) - 32) << 7
        else:
            second = 0
        firstsecond = first+second
        imulti = (i/2) * 33
        firstsecond_i_product = firstsecond ^ imulti
        total = juke + firstsecond_i_product
        print "compute('%c','%c',%x) = (%x+(%x^(%x/2*33=%x)=%x)=%x)" %
(key[i],key[i+1],i, juke, firstsecond, i, imulti, firstsecond_i_product, total),
        print '%d (0x%08X)' % (total, total)
        i += 2
    return
```

*Figure 17: Key Obfuscation Python Script*

Running this in a python shell yields the following output:

```
>>> obfuscate_key(correct_key)
53598
compute('A','v',0) = (d15e+(2b21^(0/2*33=0)=2b21)=fc7f) 64639 (0x0000FC7F)
compute('0','c',2) = (d15e+(2190^(2/2*33=21)=21b1)=f30f) 62223 (0x0000F30F)
```

```
compute('a','d',4) = (d15e+(2241^(4/2*33=42)=2203)=f361) 62305 (0x0000F361)
compute('0','_',6) = (d15e+(1f90^(6/2*33=63)=1ff3)=f151) 61777 (0x0000F151)
compute('L','o',8) = (d15e+(27ac^(8/2*33=84)=2728)=f886) 63622 (0x0000F886)
compute('v','e',a) = (d15e+(22d6^(a/2*33=a5)=2273)=f3d1) 62417 (0x0000F3D1)
compute('_','2',c) = (d15e+(93f^(c/2*33=c6)=9f9)=db57) 56151 (0x0000DB57)
compute('0','1',e) = (d15e+(890^(e/2*33=e7)=877)=d9d5) 55765 (0x0000D9D5)
compute('8','@',10) = (d15e+(1018^(10/2*33=108)=1110)=e26e) 57966 (0x0000E26E)
compute('f','l',12) = (d15e+(2646^(12/2*33=129)=276f)=f8cd) 63693 (0x0000F8CD)
compute('a','r',14) = (d15e+(2941^(14/2*33=14a)=280b)=f969) 63849 (0x0000F969)
compute('e','-',16) = (d15e+(6c5^(16/2*33=16b)=7ae)=d90c) 55564 (0x0000D90C)
compute('o','n',18) = (d15e+(274f^(18/2*33=18c)=26c3)=f821) 63521 (0x0000F821)
compute('.','c',1a) = (d15e+(218e^(1a/2*33=1ad)=2023)=f181) 61825 (0x0000F181)
compute('o','m',1c) = (d15e+(26cf^(1c/2*33=1ce)=2701)=f85f) 63583 (0x0000F85F)
```

Entering the correct key into the challenge binary running on Bochs emulator on a modern system will take several minutes to validate the input, but will yield the following victory screen shown in Figure 18.

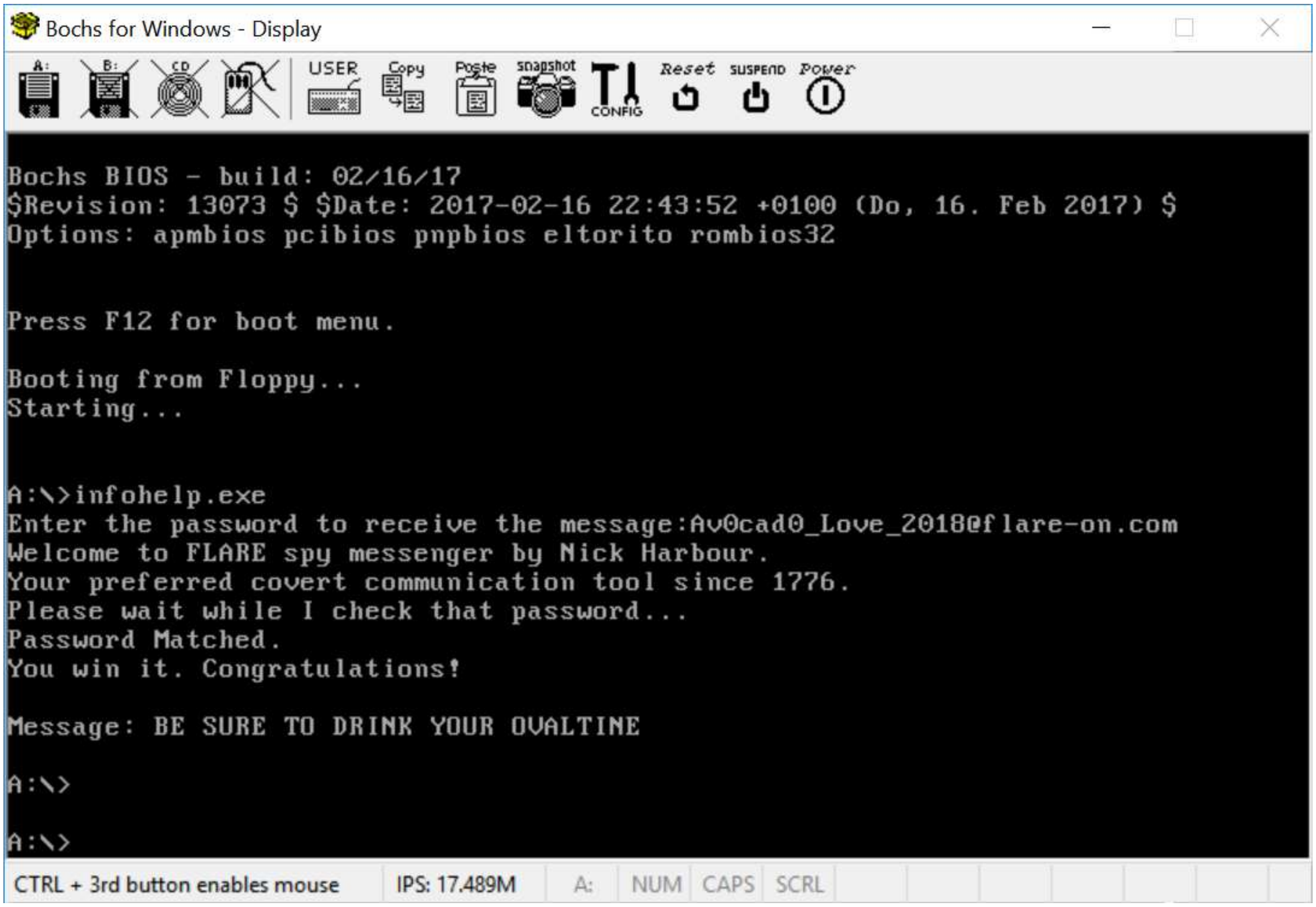


*Figure 18: Sweet Victory*

## Appendix A: macros.subleq

```
// this file contains the generic forms of all subleq abstract instructions that
are abstract from the wordsize
// as such, this file should not be included directly. instead include macros32 or
macros16 depending on the
// target wordsize of the subleq VM you are building

//      subleq a, b, c    ; Mem[b] = Mem[b] - Mem[a]
//                        ; if (Mem[b] ≤ 0) goto c

.macro HALT()   // Terminate the program
{
        subleq Z, Z, -1
}
.macro JMP(%target)   // Jump to %target
{
        subleq Z, Z, %target
}
.macro SETZ(%a) // Sets %a to 0
{
        subleq %a, %a, 0
}
.macro SETVALUE(%value,%b) // Sets %b to the value specified (direct)
{
.start_context
                MOV(_SETVALUE_tmp, %b)
                JMP(_SETVALUE_end)
_SETVALUE_tmp:  dd %value
_SETVALUE_end:
.end_context
}
.macro PTR_REF(%a,%b)   // set %b to the address of %a
{
        SETVALUE(%a,%b)
}
.macro INDIRECT_JMP(%a) // jump to the location stored in %a
{
.start_context
            MOV(%a, _INDIRECT_JMP_tailjump+2)
_INDIRECT_JMP_tailjump: subleq Z, Z, Z
.end_context
}
.macro LEA(%a, %b)   // get the address of %a and store in %b
{
            SETVALUE(%a, %b)
}
.macro DEREF_SRC_MOV(%a,%b) // get the value pointed to by %a, store in %b
```

```
{
.start_context
                MOV(%a,_DEREF_target)
                subleq %b, %b
_DEREF_target:  subleq Z, Z     // First operand will be overwritten
                subleq Z, %b
                subleq Z, Z
.end_context
}
.macro DEREF_DST_MOV(%a,%b) // put the value %a in the location pointed by %b
{
.start_context
                MOV(%b, _DEREF_DST_MOV_x)
                MOV(%b, _DEREF_DST_MOV_x+1)
                MOV(%b, _DEREF_DST_MOV_y+1)
_DEREF_DST_MOV_x:   subleq Z, Z    // will both be overwritten
                subleq %a, Z
_DEREF_DST_MOV_y:   subleq Z, Z   // second will be overwritten
                subleq Z, Z
.end_context
}
.macro ADD(%a,%b) // Add %a to %b, store in %b
{
        subleq %a, Z
        subleq Z, %b
        subleq Z, Z
}
.macro ADD_LITERAL(%value,%b)  // add literal %value to %b, store in %b
{
.start_context
        ADD(_ADD_LITERAL_val, %b)
        JMP(_ADD_LITERAL_end)
_ADD_LITERAL_val:   dd %value
_ADD_LITERAL_end:
.end_context
}
.macro INC(%a) // Increment %a
{
.start_context
            JMP(_INC_add)
_INC_ONE:   dd 1
_INC_add:   ADD(_INC_ONE,%a)
.end_context
}
.macro SUB(%a,%b) // Subtract %a from %b, store in %b
{
        subleq %a, %b
}
.macro SUB_LITERAL(%value, %b) // subtract %value from %b, store in %b
```

```
{
.start_context
        subleq _SUB_LITERAL_val, %b
        JMP(_SUB_LITERAL_end)
_SUB_LITERAL_val:   dd %value
_SUB_LITERAL_end:
.end_context
}
.macro DEC(%a)  // Decrement %a
{
.start_context
            JMP(_DEC_sub)
_DEC_ONE:   dd 1
_DEC_sub:   SUB(_DEC_ONE,%a)
.end_context
}
.macro MOV(%a,%b) // Move %a to %b
{
        MOVJMP(%a,%b,Z)
}
.macro MOVJMP(%a,%b,%c)  // Move %a to %b then jump to %c
{
        subleq %b, %b
        subleq %a, Z
        subleq Z, %b
        subleq Z, Z, %c
}
.macro BEZ(%a,%c) // Branch to %c if %a = Zero
{
.start_context
            subleq %a, Z, _BEZ_true
            subleq Z, Z, _BEZ_false
_BEZ_true:  subleq Z, Z
            subleq Z, %a, %c
_BEZ_false:
.end_context
}
.macro BNEZ(%a,%c) // Branch to %c if %a != 0
{
.start_context
            BEZ(%a,_BNEZ_false)
            JMP(%c)
_BNEZ_false:
.end_context
}
.macro BLEZ(%a,%c)  // Branch to %c if %a <= 0
{
        subleq Z, %a, %c
}
```

```
.macro BGEZ(%a,%c) // Branch to %c if %a >= 0
{
.start_context
            subleq %a, Z, _BGEZ_true
            subleq Z, Z, _BGEZ_false
_BGEZ_true: subleq Z, Z, %c
_BGEZ_false:
.end_context
}
.macro BGZ(%a,%c) // Branch to %c if %a > 0
{
.start_context
        subleq Z, %a, _BGZ_false
        subleq Z, Z, %c
_BGZ_false:
.end_context
}
.macro BLZ(%a,%c) // Branch to %c if %a < 0
{
.start_context
            subleq %a, Z, _BLZ_false
            subleq Z, Z, %c
_BLZ_false: subleq Z, Z
.end_context
}
.macro BEQ(%a,%b,%c)  // Branch to %c if %a = %b
{
.start_context
            MOV(%a,_BEQ_tmp)
            SUB(%b,_BEQ_tmp)
            BNEZ(_BEQ_tmp,_BEQ_false)
            JMP(%c)
_BEQ_tmp:   dd 0
_BEQ_false:
.end_context
}
.macro BNEQ(%a,%b,%c) // Branch to %c if %a != %b
{
.start_context
        MOV(%a,_BNEQ_tmp)
        SUB(%b,_BNEQ_tmp)
        BEZ(_BNEQ_tmp,_BNEQ_false)
        JMP(%c)
_BNEQ_tmp:  dd 0
_BNEQ_false:
.end_context
}
.macro BLEQ(%a,%b,%c) // Branch to %c if %a <= %b
{
```

```
.start_context
        MOV(%a,_BLEQ_tmp)
        SUB(%b,_BLEQ_tmp)
        BGZ(_BLEQ_tmp,_BLEQ_false)
        JMP(%c)
_BLEQ_tmp:  dd 0
_BLEQ_false:
.end_context
}
.macro BGEQ(%a,%b,%c) // Branch to %c if %a >= %b
{
.start_context
        MOV(%a,_BGEQ_tmp)
        SUB(%b,_BGEQ_tmp)
        BLZ(_BGEQ_tmp,_BGEQ_false)
        JMP(%c)
_BGEQ_tmp:  dd 0
_BGEQ_false:
.end_context
}
.macro BL(%a,%b,%c) // Branch to %c if %a < %b
{
.start_context
        MOV(%a,_BL_tmp)
        SUB(%b,_BL_tmp)
        BGEZ(_BL_tmp,_BL_false)
        JMP(%c)
_BL_tmp: dd 0
_BL_false:
.end_context
}
.macro BG(%a,%b,%c) // Branch to %c if %a > %b
{
.start_context
        MOV(%a,_BG_tmp)
        SUB(%b,_BG_tmp)
        BLEZ(_BG_tmp,_BG_false)
        JMP(%c)
_BG_tmp: dd 0
_BG_false:
.end_context
}

.macro DOUBLE(%a)   // double %a, store result in %a
{
        subleq %a, Z
        subleq %a, Z
        subleq %a, %a
        subleq Z, %a
```

```
        subleq Z, Z
}

.macro MUL(%a,%b) // multiply %a by %b and store in %b
{
// loop %b number of times adding %a to a tmp var each time.
// two bodies depending on if b is pos or negative
.start_context
            SETZ(_MUL_i)
            SETZ(_MUL_result)
            BLZ(%b,_MUL_b_neg)
            JMP(_MUL_b_pos)
_MUL_i:     dd 0
_MUL_result: dd 0
_MUL_b_pos:
            BEQ(%b, _MUL_i, _MUL_finished)
            ADD(%a,_MUL_result)
            INC(_MUL_i)
            JMP(_MUL_b_pos)
_MUL_b_neg: BEQ(%b,_MUL_i,_MUL_finished)
            SUB(%a,_MUL_result)
            DEC(_MUL_i)
            JMP(_MUL_b_neg)
_MUL_finished:
            MOV(_MUL_result,%b)
.end_context
}
.macro DIV(%a,%b,%c) // Divide %a by %b and store quotient in %b and remainder in
%c
{
.start_context
            SETZ(_DIV_i)
            SETZ(_DIV_sum)
            BLZ(%a, _DIV_a_neg)
            BLZ(%b, _DIV_a_pos_b_neg)
_DIV_a_pos_b_pos:  // keep adding %b to sum until sum > a
            BG(_DIV_sum, %a, _DIV_sub_finish)
            ADD(%b, _DIV_sum)
            BG(_DIV_sum, %a, _DIV_a_pos_b_pos)
            INC(_DIV_i)
            JMP(_DIV_a_pos_b_pos)
_DIV_a_pos_b_neg:  // keep subtracting %b from sum until sum >= a
            BG(_DIV_sum, %a, _DIV_add_finish)
            SUB(%b, _DIV_sum)
            BG(_DIV_sum, %a, _DIV_a_pos_b_neg)
            DEC(_DIV_i)
            JMP(_DIV_a_pos_b_neg)
_DIV_a_neg: BLZ(%b,_DIV_a_neg_b_neg)
_DIV_a_neg_b_pos:  // keep subtracting %b from sum until sum <= a
```

```
            BL(_DIV_sum, %a, _DIV_add_finish)
            SUB(%b, _DIV_sum)
            BL(_DIV_sum, %a, _DIV_a_neg_b_pos)
            DEC(_DIV_i)
            JMP(_DIV_a_neg_b_pos)
_DIV_a_neg_b_neg:  // keep adding %b to sum until sum <= a
            BL(_DIV_sum, %a, _DIV_sub_finish)
            ADD(%b, _DIV_sum)
            BL(_DIV_sum, %a, _DIV_a_neg_b_neg)
            INC(_DIV_i)
            JMP(_DIV_a_neg_b_neg)
_DIV_i:     dd 0
_DIV_sum:   dd 0
_DIV_tmp:   dd 0
_DIV_sub_finish:
            SUB(%b, _DIV_sum)
            JMP(_DIV_finish)
_DIV_add_finish:
            ADD(%b, _DIV_sum)
_DIV_finish:
            MOV(%a, %c)
            SUB(_DIV_sum, %c)
            MOV(_DIV_i, %b)
.end_context
}
.macro NOT(%a) // invert bits in %a
{
.start_context
        subleq %a, _NOT_tmp
        subleq _NOT_one, _NOT_tmp
        //MOV(_NOT_tmp,%a)
        MOVJMP(_NOT_tmp,%a,_NOT_finished)
_NOT_tmp: dd 0
_NOT_one: dd 1
_NOT_finished:
.end_context
}
.macro GETMSB(%a,%b)  // get the most significant bit of %a, store in %b
{
.start_context
        BLZ(%a,_GETMSB_return1)
_GETMSB_return0:
        MOVJMP(_GETMSB_zero,%b,_GETMSB_finished)
_GETMSB_return1:
        MOVJMP(_GETMSB_one,%b,_GETMSB_finished)
_GETMSB_one: dd 1
_GETMSB_zero: dd 0
_GETMSB_finished:
.end_context
```

```
}
.macro GETLSB_INNER(%a,%b,%wordsize)  // get the least significant bit of %a, store in %b
{
.start_context
        DECLARE_VARIABLE(_GETLSB_tmp,0)
        DECLARE_VARIABLE(_GETLSB_bit,0)
        MOV(%a, _GETLSB_tmp)
        SHL(%wordsize-1,_GETLSB_tmp)
        GETMSB(_GETLSB_tmp,%b)
.end_context
}
.macro SHL1(%a)  // shift the bits in %a left a bit, 0 enters right
{
    DOUBLE(%a)
}
.macro SHR1(%a,%wordmask) // shift the bits in %a right by 1. 0 enters from the left
{
.start_context
        DECLARE_VARIABLE(_SHR1_quotient,2)
        DECLARE_VARIABLE(_SHR1_remainder,0)
        DECLARE_VARIABLE(_SHR1_mask,%wordmask)
        DIV(%a,_SHR1_quotient,_SHR1_remainder)
        AND(_SHR1_mask, _SHR1_quotient)
        MOV(_SHR1_quotient, %a)
.end_context
}
.macro ROL1(%a)  // ROL %a by one bit
{
.start_context
        DECLARE_VARIABLE(_ROL1_tmp,0)
        GETMSB(%a,_ROL1_tmp)
        SHL1(%a)
        ADD(_ROL1_tmp, %a)
.end_context
}
.macro ROR1_INNER(%a,%wordsize)   // ROR %a by one bit. not a primitive function
{
        ROR_INNER(1,%a,%wordsize)
}
.macro SHL(%numbits,%b)   // Shift %b left by LITERAL %numbits amount, store result in %b
{
.start_context
        DECLARE_VARIABLE(_i,0)
        SETZ(_i)
        FORLOOP_START(_i,%numbits,_SHL_loop)
            SHL1(%b)
```

```
        FORLOOP_END(_i,_SHL_loop)
.end_context
}
.macro SHR(%numbits,%b)   // Shift %b right by LITERAL %numbits amount, store
result in %b
{
.start_context
        DECLARE_VARIABLE(_i,0)
        FORLOOP_START(_i,%numbits,_SHR_loop)
            SHR1(%b)
        FORLOOP_END(_i,_SHR_loop)
.end_context
}
.macro ROL(%numbits,%b)   // Roll %b left by LITERAL %numbits amount, store result
in %b
{
.start_context
        DECLARE_VARIABLE(_i,0)
        FORLOOP_START(_i,%numbits,_ROL_loop)
            ROL1(%b)
        FORLOOP_END(_i,_ROL_loop)
.end_context
}
.macro ROR_INNER(%numbits,%b,%wordsize)  // Roll %b right by LITERAL %numbits
amount, store result in %b
{
        ROL(%wordsize-%numbits,%b)
}
.macro FORLOOP_START(%i,%numloops,%loopID) // increment %i each time, max number of
loops %numloops, a unique loopID (in context) that a lebel will be built on
{
// note that this label is in the context of the "calling" code
_FORLOOP_START_LOOPID_%loopID:
.start_context
        JMP(_FORLOOP_START_compare)
_FORLOOP_START_numloops: dd %numloops
_FORLOOP_START_compare:
        BGEQ(%i,_FORLOOP_START_numloops,_FORLOOP_END_LOOPID_%loopID)
.end_context
}
.macro FORLOOP_VAR_START(%i,%numloops_var,%loopID) // same as FORLOOP_START but
%numloops is a variable not a literal
{
_FORLOOP_START_LOOPID_%loopID:
.start_context
        JMP(_FORLOOP_START_compare)
_FORLOOP_START_compare:
        BGEQ(%i,%numloops_var,_FORLOOP_END_LOOPID_%loopID)
.end_context
```

```
}
.macro FORLOOP_BREAK(%loopID)   // early loop terminator
{
        JMP(_FORLOOP_END_LOOPID_%loopID)
}
.macro FORLOOP_END(%i,%loopID)   // use at the end of your loop
{
        INC(%i)
        JMP(_FORLOOP_START_LOOPID_%loopID)
_FORLOOP_END_LOOPID_%loopID:
}
.macro LOOP_START(%loopID)       // Generic loop mechanism
{
_LOOP_START_LOOPID_%loopID:    // in outer context
}
.macro LOOP_BREAK(%loopID)       // generic loop break
{
        JMP(_LOOP_END_LOOPID_%loopID)
}
.macro LOOP_END(%loopID)         // generic loop ender
{
        JMP(_LOOP_START_LOOPID_%loopID)
_LOOP_END_LOOPID_%loopID:      // in outer context
}
.macro BITWISE_JMP(%a,%b,%none_target,%one_target,%both_target) // internal use.
jump to a different location based on if none, one, or both bits are set
{
.start_context
        MOV(%a,_BITWISE_JMP_tmp)
        ADD(%b,_BITWISE_JMP_tmp)
        BEZ(_BITWISE_JMP_tmp,%none_target)
        DEC(_BITWISE_JMP_tmp)
        BEZ(_BITWISE_JMP_tmp,%one_target)
        JMP(%both_target)
_BITWISE_JMP_tmp:   dd 0
.end_context
}
.macro BITWISE_OPERATOR(%a,%b,%none,%onlyone,%both)
{
.start_context
        MOV(%a,_BITWISE_OPERATOR_tmp_a)
        MOV(%b,_BITWISE_OPERATOR_tmp_b)
        SETZ(_BITWISE_OPERATOR_i)
        FORLOOP_START(_BITWISE_OPERATOR_i,32,_BITWISE_OPERATOR_loop)
            GETMSB(_BITWISE_OPERATOR_tmp_a,_BITWISE_OPERATOR_msb_a)
            GETMSB(_BITWISE_OPERATOR_tmp_b,_BITWISE_OPERATOR_msb_b)

BITWISE_JMP(_BITWISE_OPERATOR_msb_a,_BITWISE_OPERATOR_msb_b,_BITWISE_OPERATOR_none,
_BITWISE_OPERATOR_onlyone,_BITWISE_OPERATOR_both)
```

```
_BITWISE_OPERATOR_tmp_a: dd 0
_BITWISE_OPERATOR_tmp_b: dd 0
_BITWISE_OPERATOR_msb_a: dd 0
_BITWISE_OPERATOR_msb_b: dd 0
_BITWISE_OPERATOR_result: dd 0
_BO_ZERO: dd 0    // USE these in outer macro context for convenience
_BO_ONE:  dd 1    // same
_BITWISE_OPERATOR_i:     dd 0
_BITWISE_OPERATOR_none:
            MOVJMP(%none,_BITWISE_OPERATOR_msb_a,_BITWISE_OPERATOR_insert_new_bit)
_BITWISE_OPERATOR_onlyone:

MOVJMP(%onlyone,_BITWISE_OPERATOR_msb_a,_BITWISE_OPERATOR_insert_new_bit)
_BITWISE_OPERATOR_both:
            MOV(%both,_BITWISE_OPERATOR_msb_a)
_BITWISE_OPERATOR_insert_new_bit:
            DOUBLE(_BITWISE_OPERATOR_result)
            ADD(_BITWISE_OPERATOR_msb_a,_BITWISE_OPERATOR_result)
            DOUBLE(_BITWISE_OPERATOR_tmp_a)
            INC(_BITWISE_OPERATOR_tmp_a) // workaround for -INT_MIN = INT_MIN
            DOUBLE(_BITWISE_OPERATOR_tmp_b)
            INC(_BITWISE_OPERATOR_tmp_b)
        FORLOOP_END(_BITWISE_OPERATOR_i,_BITWISE_OPERATOR_loop)
        MOV(_BITWISE_OPERATOR_result,%b)
.end_context
}
.macro AND(%a,%b) // logical and %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ZERO,_BO_ZERO,_BO_ONE)
}
.macro OR(%a,%b) // logical OR %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ZERO,_BO_ONE,_BO_ONE)
}
.macro XOR(%a,%b) // logical XOR %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ZERO,_BO_ONE,_BO_ZERO)
}
.macro XNOR(%a,%b) // logical XNOR %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ONE,_BO_ZERO,_BO_ONE)
}
.macro NOR(%a,%b) // logical NOR %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ONE,_BO_ZERO,_BO_ZERO)
}
.macro NAND(%a,%b) // logical NAND %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ONE,_BO_ONE,_BO_ZERO)
```

```
}
.macro IN(%a)  // receive input and store in %a
{
    SETVALUE(1, INPUT_READY)
    MOV(INPUT,%a)
}
.macro OUT(%a)  // send %a as output
{
    MOV(%a,OUTPUT)
    SETVALUE(1, OUTPUT_READY)
}
.macro DECLARE_VARIABLE(%name,%value)  // declare a variable in your current
context
{
        subleq Z, Z, $+4
%name:  dd %value
}
.macro PUSH(%a,%sp) // push %a on the stack pointed to by %sp
{
        // note that the stacks here work exactly backwards from x86
        // and provide no overrun protection
        DEREF_DST_MOV(%a,%sp)
        INC(%sp)
}
.macro PUSH_PTR_REF(%a,%sp) // push a pointer reference to %a on the stack pointed
to by %sp
{
.start_context
        DECLARE_VARIABLE(_PTR_REF_tmp,0)
        PTR_REF(%a,_PTR_REF_tmp)
        PUSH(_PTR_REF_tmp,%sp)
.end_context
}
.macro PUSH_LITERAL(%value,%sp)  // push a LITERAL value %value on to the stack
pointed to by %sp
{
.start_context
        DECLARE_VARIABLE(_PUSH_LITERAL_tmp,%value)   // this works out to be the
same code as PUSH_PTR_REF.. probably best to use them seperately
        PUSH(_PUSH_LITERAL_tmp,%sp)
.end_context
}
.macro POP(%sp, %b)  // pop a value from the stack pointed to by %sp into %b
{
        // note that the stacks here work exactly backwards from x86
        // and provide no overrun protection
        DEC(%sp)
        DEREF_SRC_MOV(%sp,%b)
}
```

```
.macro POP_IGNORE(%numargs,%sp)   // remove %numargs items from the stack
{
    SUB_LITERAL(%numargs,%sp)
}
.macro CALL(%target, %sp)
{
.start_context
        PUSH_PTR_REF(_CALL_retloc, %sp)
        JMP(%target)
_CALL_retloc:
.end_context
}
.macro SET_FUNCTION_RETURN_VALUE(%a)   // moves %a to frame position -2
{
.start_context
        DECLARE_VARIABLE(_SET_FUNCTION_RETURN_VALUE_tmp,0)
        MOV(_bp,_SET_FUNCTION_RETURN_VALUE_tmp)
        SUB_LITERAL(2,_SET_FUNCTION_RETURN_VALUE_tmp)
        DEREF_DST_MOV(%a,_SET_FUNCTION_RETURN_VALUE_tmp)
.end_context
}
.macro GET_FUNCTION_RETURN_VALUE(%sp, %b)   // get the return value from the
previous function called with stack pointer %sp, store in %b
{
        POP(%sp,%b)
}
.macro RET(%sp)
{
.start_context
                POP(%sp, _RET_tailjump+2)
_RET_tailjump:  subleq Z, Z, Z     // third operand will be overwritten
.end_context
}
.macro FUNCTION(%name,%sp)
{
%name:
.start_context
        MAKE_FRAME_PTR(%sp,_bp)
}
.macro END_FUNCTION()
{
.end_context
}
.macro MAKE_FRAME_PTR(%sp,%bp) // make a stack frame pointer in %bp on the stack
pointed to by %sp
{
        DECLARE_VARIABLE(%bp,0)
        MOV(%sp,%bp)
}
```

```
.macro DECLARE_FUNCTION_ARGUMENT(%name,%bp_offset)  // declare a funciton argument.
meant to be used in a FUNCTION() context. uses _bp  . offsets start at -2 and work
their way lower
{
        DECLARE_VARIABLE(%name,0)   // In outer context
.start_context
        DECLARE_VARIABLE(_DECLARE_FUNCTION_ARGUMENT_bp_offset,%bp_offset)
        DECLARE_VARIABLE(_DECLARE_FUNCTION_ARGUMENT_tmp,0)
        MOV(_DECLARE_FUNCTION_ARGUMENT_bp_offset,_DECLARE_FUNCTION_ARGUMENT_tmp)
        ADD(_bp,_DECLARE_FUNCTION_ARGUMENT_tmp)
        DEREF_SRC_MOV(_DECLARE_FUNCTION_ARGUMENT_tmp,%name)
.end_context
}

.macro DEREF_OFFSET_SRC_MOV(%a, %offset, %b) // move %a[%offset] to %b
{
.start_context
        DECLARE_VARIABLE(_tmpptr, 0)
        MOV(%a, _tmpptr)
        ADD(%offset, _tmpptr)
        DEREF_SRC_MOV(_tmpptr, %b)
.end_context
}

.macro DEREF_OFFSET_DST_MOV(%a, %b, %offset) // move %a to %b[%offset]
{
.start_context
        DECLARE_VARIABLE(_tmpptr, 0)
        MOV(%b, _tmpptr)
        ADD(%offset, _tmpptr)
        DEREF_DST_MOV(%a, _tmpptr)
.end_context
}

// EOF
```

# Appendix B: macros16.subleq

```
.include "macros.subleq"

.macro GETLSB(%a,%b)
{
    GETLSB_INNER(%a,%b,16)
}

.macro ROR(%a,%b)
{
    ROR_INNER(%a,%b,16)
}

.macro ROR1(%a,%b)
{
    ROR1_INNER(%a,%b,16)
}

.macro SHR1(%a)
{
    SHR1_INNER(%a,0x7FFF)
}
```

# Appendix C: rssbmacros.subleq

```
.macro HALT()  // Terminate the program
{
        rssb -2
}

.macro NOP()
{
    rssb -1
}

.macro JMP(%target)
{
.start_context
    rssb    ACC
    rssb    _JMP_TARGET_DELTA
    rssb    ZERO // acc = -acc
    rssb    ZERO // same, but for skipped

    rssb    IP
_JMP_TAIL:
_JMP_TARGET_DELTA: dd %target-_JMP_TAIL
.end_context
}

.macro ADD(%a,%b) // add %a to %b and store result in %b
{
    rssb    ACC
    rssb    %a
    rssb    ZERO // acc = -acc
    rssb    ZERO // same, but for skipped
    rssb    %b
    rssb    ACC // pad if result was zero
    //rssb  ACC
}

.macro SUB(%a,%b) // subtract %a from %b and store result in %b
{
.start_context
_SUB_start:
    rssb    ACC
    rssb    %a
    NOP()
    rssb    ZERO // acc = -acc
    NOP()
    rssb    ZERO // acc = -acc
    NOP()
```

```
    rssb    %b
    NOP() // pad if result was zero
    //rssb  ACC
.end_context

}

.macro DEC(%a) // Subtract 1 from %a, store result in %acc
{
.start_context
    SUB(_DEC_ONE,%a)
    JMP(_DEC_TAIL)
_DEC_ONE: dd 1
_DEC_TAIL:
.end_context
}

.macro INC(%a) // Add 1 to %a, store result in %acc
{
.start_context
    ADD(_INC_ONE,%a)
    JMP(_INC_TAIL)
_INC_ONE: dd 1
_INC_TAIL:
.end_context
}

.macro SETZ(%a)  // set %a to zero
{
    // clear out destination
    SUB(%a,%a)
}

.macro MOV(%a,%b) // move %a to %b
{
    SETZ(%b)

    ADD(%a,%b)

}

.macro PTR_REF(%a,%b)  // set %b to the address of %a
{
        SETVALUE(%a,%b)
}

.macro DEREF_SRC_MOV(%a,%b) // get the value pointed to by %a, store in %b
{
.start_context
```

```
    SETZ(%b)
    MOV(%a,_DEREF_SRC_MOV_TAIL_ADD+1)
_DEREF_SRC_MOV_TAIL_ADD:
    // expanded ADD macro
    rssb    ACC
    rssb    %a // << will be overwritten
    rssb    ZERO // acc = -acc
    rssb    ZERO // same, but for skipped
    rssb    %b
    rssb    ACC // pad if result was zero
.end_context
}

.macro DEREF_DST_MOV(%a,%b) // put the value %a in the location pointed to by %b
{
.start_context
    MOV(%b,_DEREF_DST_MOV_SETZ_SUB+1)
    MOV(%b,_DEREF_DST_MOV_SETZ_SUB+5)
_DEREF_DST_MOV_SETZ_SUB:
    // expanded SUB(x,x)
    rssb    ACC
    rssb    %b
    rssb    ZERO
    rssb    ZERO
    rssb    ZERO
    rssb    %b
    rssb    ACC

    MOV(%b,_DEREF_DST_MOV_TAIL_ADD+4)
_DEREF_DST_MOV_TAIL_ADD:
    // expanded ADD macro
    rssb    ACC
    rssb    %a
    rssb    ZERO // acc = -acc
    rssb    ZERO // same, but for skipped
    rssb    %b // << will be overwritten
    rssb    ACC // pad if result was zero
.end_context

}

.macro INDIRECT_JMP(%a) // jump to the location stored in %a
{
.start_context
    SETZ(_INDIRECT_JMP_NEW_TARGET)
    MOV(%a, _INDIRECT_JMP_NEW_TARGET)
    SUB(_INDIRECT_JMP_TAIL_ADDR,_INDIRECT_JMP_NEW_TARGET)
_INDIRECT_JMP_tailjump:
    rssb    ACC
```

```
    rssb    _INDIRECT_JMP_NEW_TARGET
    rssb    ZERO // acc = -acc
    rssb    ZERO // same, but for skipped
    rssb    IP
_INDIRECT_JMP_TAIL:
_INDIRECT_JMP_TAIL_ADDR: dd _INDIRECT_JMP_TAIL
_INDIRECT_JMP_NEW_TARGET: dd 0
.end_context
}

.macro BLZ(%a,%target) // Branch to %target if %a < 0
{
.start_context
    SETZ(_BLZ_WILL_BRANCH)
    SETZ(_BLZ_TMP)
    rssb    ACC  // clear ACC
    rssb    %a
    rssb    ACC  // skipped if a < 0, so ACC is either negative or 0 after this
    rssb    _BLZ_WILL_BRANCH  // will be positive number if branching, 0 if not
    MOV(_BLZ_WILL_BRANCH,_BLZ_TMP)
    MOV(_BLZ_NOT_BRANCHING_TARGET_DELTA_ORIGINAL,_BLZ_NOT_BRANCHING_TARGET_DELTA)
    MOV(_BLZ_BRANCHING_TARGET_DELTA_ORIGINAL,_BLZ_BRANCHING_TARGET_DELTA)
    DEC(_BLZ_TMP)
    JMP(_BLZ_OVER_VARS)
_BLZ_TARGET: dd %target-_BLZ_TAIL
_BLZ_ONE: dd 1
_BLZ_NEGONE: dd -1
_BLZ_TMP: dd 0
_BLZ_INNER_JUMP_VAL: dd 0
_BLZ_WILL_BRANCH: dd 0
_BLZ_BRANCHING_TARGET_DELTA_ORIGINAL: dd %target-_BLZ_TAIL
_BLZ_NOT_BRANCHING_TARGET_DELTA_ORIGINAL: dd _BLZ_TAIL - _BLZ_NOT_BRANCHING_TAIL
_BLZ_NOT_BRANCHING_TARGET_DELTA: dd 0 // gets set from a MOV at the beginning of
the macro
_BLZ_BRANCHING_TARGET_DELTA: dd 0 // gets set from a MOV each time the macro is
executed
_BLZ_OVER_VARS:
    rssb    ACC
    rssb    _BLZ_TMP   // tmp should be -1 if not branching or 0+ if branching
    rssb    _BLZ_TMP   // skipped if not branching. should normalize branching
values to 0
    INC(_BLZ_TMP)
    // at this point _BLZ_TMP should be 0 if not branching and 1 if branching)

    // I think what we want to do is repeatedly add _BLZ_TMP to determine which
tail to jump to
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
```

```
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    ADD(_BLZ_TMP,_BLZ_INNER_JUMP_VAL)
    // perform jump to $+INNER_JUMP_VAL
    rssb    ACC
    rssb    _BLZ_INNER_JUMP_VAL
    rssb    ZERO
    rssb    ZERO
    rssb    IP  // should jump forward _BLZ_INNER_JUMP_VAL instructions

_BLZ_NOT_BRANCHING:
    //SETZ(_BLZ_INNER_JUMP_VAL)
    rssb    ACC
    rssb    _BLZ_INNER_JUMP_VAL
    rssb    -1
    rssb    ZERO // acc = -acc
    rssb    -1
    rssb    ZERO // acc = -acc
    rssb    -1
    rssb    _BLZ_INNER_JUMP_VAL
    rssb    -1 // pad if result was zero
    // expanded JMP macro since we don't want this function to break if we change
the original JMP macro at all
    rssb    ACC
    rssb    _BLZ_NOT_BRANCHING_TARGET_DELTA
    rssb    ZERO // acc = -acc
    rssb    ZERO // same, but for skipped
    rssb    IP
_BLZ_NOT_BRANCHING_TAIL:

_BLZ_BRANCHING:
    //SETZ(_BLZ_INNER_JUMP_VAL)
    rssb    ACC
    rssb    _BLZ_INNER_JUMP_VAL
    rssb    -1
    rssb    ZERO // acc = -acc
    rssb    -1
    rssb    ZERO // acc = -acc
    rssb    -1
    rssb    _BLZ_INNER_JUMP_VAL
    rssb    -1
```

```
        // expanded JMP macro since we don't want this function to break if we
change the original JMP macro at all
    rssb    ACC
    rssb    _BLZ_BRANCHING_TARGET_DELTA
    rssb    ZERO // acc = -acc
    rssb    ZERO // same, but for skipped
    rssb    IP
_BLZ_TAIL:
.end_context
}


.macro BEZ(%a,%c) // Branch to %c if %a = Zero
{
.start_context
    MOV(%a,_BEZ_TMP)
    BLZ(_BEZ_TMP,_BEZ_NOT_BRANCHING)
    DEC(_BEZ_TMP)
    BLZ(_BEZ_TMP,_BEZ_BRANCHING)
    JMP(_BEZ_NOT_BRANCHING)
_BEZ_BRANCHING:
    JMP(%c)
_BEZ_NOT_BRANCHING:
    JMP(_BEZ_TAIL)
_BEZ_TMP: dd 0
_BEZ_TAIL:
.end_context
}

.macro BNEZ(%a,%c) // Branch to %c if %a != 0
{
.start_context
    BEZ(%a,_BNEZ_TAIL)
    JMP(%c)
_BNEZ_TAIL:
.end_context
}
.macro BLEZ(%a,%c)  // Branch to %c if %a <= 0
{
    BEZ(%a,%c)
    BLZ(%a,%c)
}
.macro BGEZ(%a,%c) // Branch to %c if %a >= 0
{
.start_context
    BLZ(%a,_BGEZ_TAIL)
    JMP(%c)
_BGEZ_TAIL:
.end_context
```

```
}
.macro BGZ(%a,%c) // Branch to %c if %a > 0
{
.start_context
    BLZ(%a,_BGZ_TAIL)
    BEZ(%a,_BGZ_TAIL)
    JMP(%c)
_BGZ_TAIL:
.end_context
}

.macro BEQ(%a,%b,%c)  // Branch to %c if %a = %b
{
.start_context
    MOV(%a,_BEQ_TMP)
    SUB(%b,_BEQ_TMP)
    BEZ(_BEQ_TMP,%c)
    JMP(_BEQ_TAIL)
_BEQ_TMP: dd 0
_BEQ_TAIL:
.end_context
}
.macro BNEQ(%a,%b,%c) // Branch to %c if %a != %b
{
.start_context
    MOV(%a,_BNEQ_TMP)
    SUB(%b,_BNEQ_TMP)
    BNEZ(%a,%c)
    JMP(_BNEQ_TAIL)
_BNEQ_TMP: dd 0
_BNEQ_TAIL:
.end_context
}
.macro BLEQ(%a,%b,%c) // Branch to %c if %a <= %b
{
.start_context
    MOV(%a,_BLEQ_TMP)
    SUB(%b,_BLEQ_TMP)
    BLEZ(_BLEQ_TMP,%c)
    JMP(_BLEQ_TAIL)
_BLEQ_TMP: dd 0
_BLEQ_TAIL:
.end_context
}
.macro BGEQ(%a,%b,%c) // Branch to %c if %a >= %b
{
.start_context
    MOV(%a,_BGEQ_TMP)
    SUB(%b,_BGEQ_TMP)
```

```
    BGEZ(_BGEQ_TMP,%c)
    JMP(_BGEQ_TAIL)
_BGEQ_TMP: dd 0
_BGEQ_TAIL:
.end_context
}
.macro BL(%a,%b,%c) // Branch to %c if %a < %b
{
.start_context
    MOV(%a,_BL_TMP)
    SUB(%b,_BL_TMP)
    BLZ(_BL_TMP,%c)
    JMP(_BL_TAIL)
_BL_TMP: dd 0
_BL_TAIL:
.end_context
}
.macro BG(%a,%b,%c) // Branch to %c if %a > %b
{
.start_context
    MOV(%a,_BG_TMP)
    SUB(%b,_BG_TMP)
    BGZ(_BG_TMP,%c)
    JMP(_BG_TAIL)
_BG_TMP: dd 0
_BG_TAIL:
.end_context
}

.macro MOVJMP(%a,%b,%c) // Move %a to %b then jump to %c
{
    MOV(%a,%b)
    JMP(%c)
}

.macro NOT(%a) // invert bits in %a
{
.start_context
        MOV(%a,_NOT_tmp)
        SUB(%a,_NOT_tmp)
        SUB(%a,_NOT_tmp)
        MOVJMP(_NOT_tmp,%a,_NOT_TAIL)
_NOT_tmp: dd 0
_NOT_TAIL:
.end_context
}

.macro MUL(%a,%b) // multiply %a by %b and store in %b
{
```

```
// loop %b number of times adding %a to a tmp var each time.
// two bodies depending on if b is pos or negative
.start_context
            SETZ(_MUL_i)
            SETZ(_MUL_result)
            BLZ(%b,_MUL_b_neg)
            JMP(_MUL_b_pos)
_MUL_i:     dd 0
_MUL_result: dd 0
_MUL_b_pos:
            BEQ(%b, _MUL_i, _MUL_finished)
            ADD(%a,_MUL_result)
            INC(_MUL_i)
            JMP(_MUL_b_pos)
_MUL_b_neg: BEQ(%b,_MUL_i,_MUL_finished)
            SUB(%a,_MUL_result)
            DEC(_MUL_i)
            JMP(_MUL_b_neg)
_MUL_finished:
            MOV(_MUL_result,%b)
.end_context
}

.macro DIV(%a,%b,%c) // Divide %a by %b and store quotient in %b and remainder in
%c
{
.start_context
            SETZ(_DIV_i)
            SETZ(_DIV_sum)
            BLZ(%a, _DIV_a_neg)
            BLZ(%b, _DIV_a_pos_b_neg)
_DIV_a_pos_b_pos:  // keep adding %b to sum until sum > a
            BG(_DIV_sum, %a, _DIV_sub_finish)
            ADD(%b, _DIV_sum)
            BG(_DIV_sum, %a, _DIV_a_pos_b_pos)
            INC(_DIV_i)
            JMP(_DIV_a_pos_b_pos)
_DIV_a_pos_b_neg:  // keep subtracting %b from sum until sum >= a
            BG(_DIV_sum, %a, _DIV_add_finish)
            SUB(%b, _DIV_sum)
            BG(_DIV_sum, %a, _DIV_a_pos_b_neg)
            DEC(_DIV_i)
            JMP(_DIV_a_pos_b_neg)
_DIV_a_neg: BLZ(%b,_DIV_a_neg_b_neg)
_DIV_a_neg_b_pos:  // keep subtracting %b from sum until sum <= a
            BL(_DIV_sum, %a, _DIV_add_finish)
            SUB(%b, _DIV_sum)
            BL(_DIV_sum, %a, _DIV_a_neg_b_pos)
            DEC(_DIV_i)
```

```
            JMP(_DIV_a_neg_b_pos)
_DIV_a_neg_b_neg:  // keep adding %b to sum until sum <= a
            BL(_DIV_sum, %a, _DIV_sub_finish)
            ADD(%b, _DIV_sum)
            BL(_DIV_sum, %a, _DIV_a_neg_b_neg)
            INC(_DIV_i)
            JMP(_DIV_a_neg_b_neg)
_DIV_i:     dd 0
_DIV_sum:   dd 0
_DIV_tmp:   dd 0
_DIV_sub_finish:
            SUB(%b, _DIV_sum)
            JMP(_DIV_finish)
_DIV_add_finish:
            ADD(%b, _DIV_sum)
_DIV_finish:
            MOV(%a, %c)
            SUB(_DIV_sum, %c)
            MOV(_DIV_i, %b)
.end_context
}

.macro DECLARE_VARIABLE(%name,%value) // declare a variable in your current context
{
    JMP(%name+1)
%name:  dd %value
}

.macro SETVALUE(%value,%b) // sets %b to the value specified (direct)
{
.start_context
    MOVJMP(_SETVALUE_TMP,%b,_SETVALUE_TAIL)
_SETVALUE_TMP:  dd %value
_SETVALUE_TAIL:
.end_context
}

.macro LEA(%a, %b)  // get the address of %a and store in %b
{
            SETVALUE(%a, %b)
}

.macro ADD_LITERAL(%value,%b)  // add literal %value to %b, store in %b
{
.start_context
        ADD(_ADD_LITERAL_val, %b)
        JMP(_ADD_LITERAL_end)
_ADD_LITERAL_val:   dd %value
_ADD_LITERAL_end:
```

```
.end_context
}

.macro SUB_LITERAL(%value, %b) // subtract %value from %b, store in %b
{
.start_context
        SUB(_SUB_LITERAL_val, %b)
        JMP(_SUB_LITERAL_end)
_SUB_LITERAL_val:   dd %value
_SUB_LITERAL_end:
.end_context
}

.macro DOUBLE(%a) // double %a, store result in %a
{
.start_context
    MOV(%a,_DOUBLE_TMP)
    ADD(%a,_DOUBLE_TMP)
    MOVJMP(_DOUBLE_TMP,%a,_DOUBLE_TAIL)
_DOUBLE_TMP: dd 0
_DOUBLE_TAIL:
.end_context
}

.macro GETMSB(%a,%b)  // get the most significant bit of %a, store in %b
{
.start_context
        BLZ(%a,_GETMSB_return1)
_GETMSB_return0:
        MOVJMP(_GETMSB_zero,%b,_GETMSB_finished)
_GETMSB_return1:
        MOVJMP(_GETMSB_one,%b,_GETMSB_finished)
_GETMSB_one: dd 1
_GETMSB_zero: dd 0
_GETMSB_finished:
.end_context
}

.macro GETLSB_INNER(%a,%b,%wordsize)  // get the least significant bit of %a, store
in %b
{
.start_context
        DECLARE_VARIABLE(_GETLSB_tmp,0)
        DECLARE_VARIABLE(_GETLSB_bit,0)
        MOV(%a, _GETLSB_tmp)
        SHL(%wordsize-1,_GETLSB_tmp)
        GETMSB(_GETLSB_tmp,%b)
.end_context
}
```

```
.macro SHL1(%a)  // shift the bits in %a left a bit, 0 enters right
{
    DOUBLE(%a)
}

.macro SHL(%numbits,%b)   // Shift %b left by LITERAL %numbits amount, store result
in %b
{
.start_context
        DECLARE_VARIABLE(_i,0)
        SETZ(_i)
        FORLOOP_START(_i,%numbits,_SHL_loop)
            SHL1(%b)
        FORLOOP_END(_i,_SHL_loop)
.end_context
}

.macro SHR1(%a,%wordmask) // shift the bits in %a right by 1. 0 enters from the
left
{
.start_context
        DECLARE_VARIABLE(_SHR1_quotient,2)
        DECLARE_VARIABLE(_SHR1_remainder,0)
        DECLARE_VARIABLE(_SHR1_mask,%wordmask)
        DIV(%a,_SHR1_quotient,_SHR1_remainder)
        AND(_SHR1_mask, _SHR1_quotient)
        MOV(_SHR1_quotient, %a)
.end_context
}

.macro SHR(%numbits,%b)   // Shift %b right by LITERAL %numbits amount, store
result in %b
{
.start_context
        DECLARE_VARIABLE(_i,0)
        FORLOOP_START(_i,%numbits,_SHR_loop)
            SHR1(%b)
        FORLOOP_END(_i,_SHR_loop)
.end_context
}

.macro FORLOOP_START(%i,%numloops,%loopID) // increment %i each time, max number of
loops %numloops, a unique loopID (in context) that a lebel will be built on
{
// note that this label is in the context of the "calling" code
_FORLOOP_START_LOOPID_%loopID:
.start_context
        JMP(_FORLOOP_START_compare)
```

```
_FORLOOP_START_numloops: dd %numloops
_FORLOOP_START_compare:
        BGEQ(%i,_FORLOOP_START_numloops,_FORLOOP_END_LOOPID_%loopID)
.end_context
}
.macro FORLOOP_VAR_START(%i,%numloops_var,%loopID) // same as FORLOOP_START but
%numloops is a variable not a literal
{
_FORLOOP_START_LOOPID_%loopID:
.start_context
        JMP(_FORLOOP_START_compare)
_FORLOOP_START_compare:
        BGEQ(%i,%numloops_var,_FORLOOP_END_LOOPID_%loopID)
.end_context
}
.macro FORLOOP_BREAK(%loopID)  // early loop terminator
{
        JMP(_FORLOOP_END_LOOPID_%loopID)
}
.macro FORLOOP_END(%i,%loopID)  // use at the end of your loop
{
        INC(%i)
        JMP(_FORLOOP_START_LOOPID_%loopID)
_FORLOOP_END_LOOPID_%loopID:
}
.macro LOOP_START(%loopID)      // Generic loop mechanism
{
_LOOP_START_LOOPID_%loopID:    // in outer context
}
.macro LOOP_BREAK(%loopID)      // generic loop break
{
        JMP(_LOOP_END_LOOPID_%loopID)
}
.macro LOOP_END(%loopID)        // generic loop ender
{
        JMP(_LOOP_START_LOOPID_%loopID)
_LOOP_END_LOOPID_%loopID:      // in outer context
}
.macro BITWISE_JMP(%a,%b,%none_target,%one_target,%both_target) // internal use.
jump to a different location based on if none, one, or both bits are set
{
.start_context
        MOV(%a,_BITWISE_JMP_tmp)
        ADD(%b,_BITWISE_JMP_tmp)
        BEZ(_BITWISE_JMP_tmp,%none_target)
        DEC(_BITWISE_JMP_tmp)
        BEZ(_BITWISE_JMP_tmp,%one_target)
        JMP(%both_target)
_BITWISE_JMP_tmp:   dd 0
```

```
.end_context
}
.macro BITWISE_OPERATOR(%a,%b,%none,%onlyone,%both)
{
.start_context
        MOV(%a,_BITWISE_OPERATOR_tmp_a)
        MOV(%b,_BITWISE_OPERATOR_tmp_b)
        SETZ(_BITWISE_OPERATOR_i)
        FORLOOP_START(_BITWISE_OPERATOR_i,16,_BITWISE_OPERATOR_loop)
            GETMSB(_BITWISE_OPERATOR_tmp_a,_BITWISE_OPERATOR_msb_a)
            GETMSB(_BITWISE_OPERATOR_tmp_b,_BITWISE_OPERATOR_msb_b)

BITWISE_JMP(_BITWISE_OPERATOR_msb_a,_BITWISE_OPERATOR_msb_b,_BITWISE_OPERATOR_none,
_BITWISE_OPERATOR_onlyone,_BITWISE_OPERATOR_both)
_BITWISE_OPERATOR_tmp_a: dd 0
_BITWISE_OPERATOR_tmp_b: dd 0
_BITWISE_OPERATOR_msb_a: dd 0
_BITWISE_OPERATOR_msb_b: dd 0
_BITWISE_OPERATOR_result: dd 0
_BO_ZERO: dd 0   // USE these in outer macro context for convenience
_BO_ONE:  dd 1   // same
_BITWISE_OPERATOR_i:    dd 0
_BITWISE_OPERATOR_none:
            MOVJMP(%none,_BITWISE_OPERATOR_msb_a,_BITWISE_OPERATOR_insert_new_bit)
_BITWISE_OPERATOR_onlyone:

MOVJMP(%onlyone,_BITWISE_OPERATOR_msb_a,_BITWISE_OPERATOR_insert_new_bit)
_BITWISE_OPERATOR_both:
            MOV(%both,_BITWISE_OPERATOR_msb_a)
_BITWISE_OPERATOR_insert_new_bit:
            DOUBLE(_BITWISE_OPERATOR_result)
            ADD(_BITWISE_OPERATOR_msb_a,_BITWISE_OPERATOR_result)
            DOUBLE(_BITWISE_OPERATOR_tmp_a)
            INC(_BITWISE_OPERATOR_tmp_a) // workaround for -INT_MIN = INT_MIN
            DOUBLE(_BITWISE_OPERATOR_tmp_b)
            INC(_BITWISE_OPERATOR_tmp_b)
        FORLOOP_END(_BITWISE_OPERATOR_i,_BITWISE_OPERATOR_loop)
        MOV(_BITWISE_OPERATOR_result,%b)
.end_context
}
.macro AND(%a,%b) // logical and %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ZERO,_BO_ZERO,_BO_ONE)
}
.macro OR(%a,%b) // logical OR %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ZERO,_BO_ONE,_BO_ONE)
}
.macro XOR(%a,%b) // logical XOR %a and %b, store result in %b
```

```
{
        BITWISE_OPERATOR(%a,%b,_BO_ZERO,_BO_ONE,_BO_ZERO)
}
.macro XNOR(%a,%b) // logical XNOR %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ONE,_BO_ZERO,_BO_ONE)
}
.macro NOR(%a,%b) // logical NOR %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ONE,_BO_ZERO,_BO_ZERO)
}
.macro NAND(%a,%b) // logical NAND %a and %b, store result in %b
{
        BITWISE_OPERATOR(%a,%b,_BO_ONE,_BO_ONE,_BO_ZERO)
}

.macro IN(%a)  // receive input and store in %a
{
    SETVALUE(1, INPUT_READY)
    MOV(INPUT,%a)
}
.macro OUT(%a)  // send %a as output
{
    MOV(%a,OUTPUT)
    SETVALUE(1, OUTPUT_READY)
}
.macro PUSH(%a,%sp) // push %a on the stack pointed to by %sp
{
    // note that the stacks here work exactly backwards from x86
    // and provide no overrun protection
    DEREF_DST_MOV(%a,%sp)
    INC(%sp)
}
.macro PUSH_PTR_REF(%a,%sp) // push a pointer reference to %a on the stack pointed
to by %sp
{
.start_context
    DECLARE_VARIABLE(_PTR_REF_tmp,0)
    PTR_REF(%a,_PTR_REF_tmp)
    PUSH(_PTR_REF_tmp,%sp)
.end_context
}
.macro PUSH_LITERAL(%value,%sp)  // push a LITERAL value %value on to the stack
pointed to by %sp
{
.start_context
    DECLARE_VARIABLE(_PUSH_LITERAL_tmp,%value)   // this works out to be the same
code as PUSH_PTR_REF.. probably best to use them seperately
    PUSH(_PUSH_LITERAL_tmp,%sp)
```

```
.end_context
}
.macro POP(%sp, %b)  // pop a value from the stack pointed to by %sp into %b
{
    // note that the stacks here work exactly backwards from x86
    // and provide no overrun protection
    DEC(%sp)
    DEREF_SRC_MOV(%sp,%b)
}
.macro POP_IGNORE(%numargs,%sp)  // remove %numargs items from the stack
{
    SUB_LITERAL(%numargs,%sp)
}
.macro CALL(%target, %sp)
{
.start_context
    PUSH_PTR_REF(_CALL_retloc, %sp)
    JMP(%target)
_CALL_retloc:
.end_context
}
.macro SET_FUNCTION_RETURN_VALUE(%a)  // moves %a to frame position -2
{
.start_context
    DECLARE_VARIABLE(_SET_FUNCTION_RETURN_VALUE_tmp,0)
    MOV(_bp,_SET_FUNCTION_RETURN_VALUE_tmp)
    SUB_LITERAL(2,_SET_FUNCTION_RETURN_VALUE_tmp)
    DEREF_DST_MOV(%a,_SET_FUNCTION_RETURN_VALUE_tmp)
.end_context
}
.macro GET_FUNCTION_RETURN_VALUE(%sp, %b)  // get the return value from the
previous function called with stack pointer %sp, store in %b
{
    POP(%sp,%b)
}
.macro RET(%sp)
{
.start_context
    POP(%sp, _RET_ptr)
    INDIRECT_JMP(_RET_ptr)
_RET_ptr:   dd 0
.end_context
}
.macro FUNCTION(%name,%sp)
{
%name:
.start_context
    MAKE_FRAME_PTR(%sp,_bp)
}
```

```
.macro END_FUNCTION()
{
.end_context
}
.macro MAKE_FRAME_PTR(%sp,%bp) // make a stack frame pointer in %bp on the stack
pointed to by %sp
{
    DECLARE_VARIABLE(%bp,0)
    MOV(%sp,%bp)
}
.macro DECLARE_FUNCTION_ARGUMENT(%name,%bp_offset)  // declare a funciton argument.
meant to be used in a FUNCTION() context. uses _bp  . offsets start at -2 and work
their way lower
{
    DECLARE_VARIABLE(%name,0)    // In outer context
.start_context
    DECLARE_VARIABLE(_DECLARE_FUNCTION_ARGUMENT_bp_offset,%bp_offset)
    DECLARE_VARIABLE(_DECLARE_FUNCTION_ARGUMENT_tmp,0)
    MOV(_DECLARE_FUNCTION_ARGUMENT_bp_offset,_DECLARE_FUNCTION_ARGUMENT_tmp)
    ADD(_bp,_DECLARE_FUNCTION_ARGUMENT_tmp)
    DEREF_SRC_MOV(_DECLARE_FUNCTION_ARGUMENT_tmp,%name)
.end_context
}
```