Rutgers, the State University of New Jersey

Senior Capstone Design, Spring '13

FPGA Design - the Making of an Intel 8086 Microprocessor with Modern Technology

Author:
Elie Rosen

Advisor:
Professor Michael Caggiano

April 29, 2013

Abstract

The Intel 8086 microprocessor was first introduced in 1978. Since then the semiconductor industry has changed vastly from the old chip manufacturing techniques of the time. Today we can fit thousands of Intel 8086 microprocessors in the same size package with use of modern semiconductor techniques such as the ability to design with 22nm feature size and better yield from improved wafer quality. This paper examines how we can still learn from preceding technology with a more modern twist. By utilizing field programmable gate arrays, we can easily implement the same technology from the past and learn about architectures that are still relevant in todays modern processors.

Contents

1 Introduction

It was in the mid 1970s when Intel announced their latest project, the Intel 8086 - a 16-bit microprocessor capable of supporting up to a revolutionary 1 mega*byte* of address space and 64 kilo*bytes* of input/output. Gone were the days of simple computing in only 8-bits of freedom, this was the 70's and 16-bits were here to take over. Along with the increases in accessible memory and larger size ALU computations, Intel introduced a new type of architecture and instruction set known as x86, this new method of computing revolved around the use of registers that stored input and output data which could then have computations performed on them. This improvement has since paved the way for future computing by setting a standard on how the processor receives data and how the data can then be processed in a regular clock cycle. The 8086 supported 80 assembly instructions which also included instructions compatible with the older 8-bit processors so the older programs would still be backwards compatible with the new technology, this capability alone was one of the major reasons that large companies began transitioning to the new architecture which began a substantial drive to future 16-bit iterations of the processor.

The field-programmable gate array (FPGA) has been around since the 1980's, its purpose was to be able to easily create and prototype custom hardware without having to expend vast resources required for designing and manufacturing application-specific integrated circuits (ASICs). The FPGA accomplishes this by using "logic elements" (LE), a term that varies by manufacturer but is essentially the same, which is typically a circuit that consists of a lookup table for performing an array of logic operations, multiplexers and low level logic gates that can be configured in such a way as to create custom complex logic such as adders/subtractors or even be used for more simple XOR and NAND gates. A common structure of a logic element can be found in Figure 1, this one is from the Altera Cyclone III family. The elements are most often configured as a matrix with interconnects for inputs, outputs and configuration paths in between. In the last decade or so, improvements in silicon technologies have allowed FPGA manufacturers to greatly increase the number of logic elements on a chip into the hundreds of thousands and beyond which makes it possible to design hardware of almost the same complexity as modern ASICs.

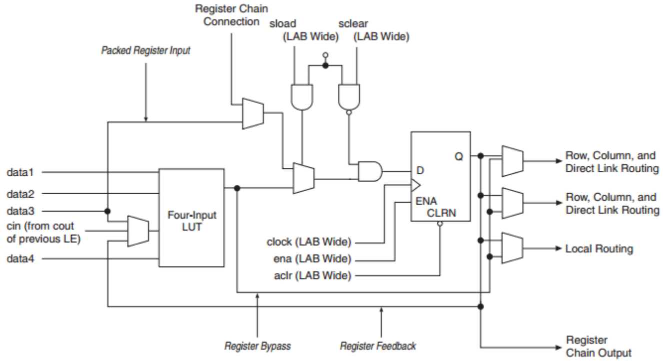


Figure 1: Cyclone III Device Family LEs in Normal Mode [1]

After discovering an independent open source initiative, Zet Processor - a clone of the Intel 8086 hardware description by Zeus Gmez Marmolejo, it was determined that this project would make it possible so that the Intel 8086 microprocessor could be implemented onto a modern FPGA [2]. The project as a whole would encourage study in all areas of focus for a person with primary interests in Computer Architecture, Digital Systems Design, FPGAs, and Assembly programming. This project and study is something that can also be completed within the time frame of a typical academic semester.

2 Implementation Requirements

Through extensive research of various FPGA manufacturers and the different styles of development boards offered from a select number of companies, it was determined that an FPGA that is able to hold more than 9,000 LEs would be required to successfully implement the Intel 8086 microprocessor. As an added necessity, it was important to select a development board that would be able to handle the project inputs and outputs such as PS2 keyboard and VGA output in order to spend more time on the study and not building miscellaneous external hardware. From this research, it was determined that the Altera DE0 development board sufficiently met the project needs with over 15,000 LE's, a VGA port, a PS2 port, buttons, LEDs, switches, and USB interface. Another important addition to the board is its Secure Digital (SD) memory card slot which would allow for external flash memory storage of an operating system, full specifications for the Altera DE0 board can be found in Appendix 7.

Other components for the project include; an LCD monitor to visualize the system status, a PS2 keyboard to interact with the operating system, a computer with Linux to compile the processor and write the operating system to the memory card.

Since the study is sponsored by the Altera University Program, an Altera DE0 development board has been provided as well as an Altera DE0-Nano development boards at no cost. This makes the projects overall required budget $0 since the devices come with the necessary software to program the FPGA and all of the other software packages used are open source.

3 Procedure

The Zet Processor is more than just Verilog code for an Intel 8086 microprocessor, it also contains the necessary BIOS and drivers for treating the SD memory card as a mounted hard disc. The preliminary steps for installing the processor first include compiling the BIOS by running it through the Open Watcom compiler which takes the BIOS code written in C and compiles it into x86 Assembly optimized for 16-bit instructions [3]. We then take the BIOS assembly and convert it into hexadecimal by means of a simple file conversion script, this is now our ROM for placing onto the FPGA. The BIOS then gets placed onto the FPGA through running the DE0_Control_Panel, an example program that comes with the FPGA software, which allows for files to be placed directly onto the on-board flash memory. This is important because as the processor boots, it will automatically look at address 0x00000 for instructions on where to next proceed such as how to mount the operating system and ultimately boot it.

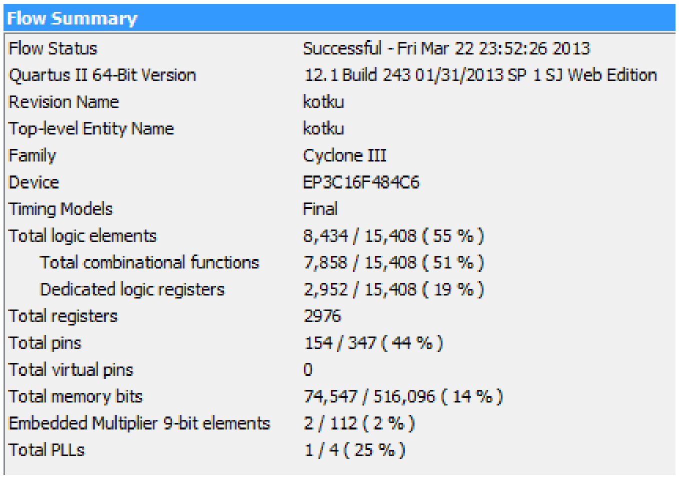
Since the processor runs 16-bit x86 it is necessary to choose a compatible operating system. This leaves a few options including MS-DOS 6.22, FreeDOS 1.1, and Microsoft Windows 3.0. MS-DOS was chosen due to its small file size and easy to work with command line interface, as an added bonus the copyright for MS-DOS has expired which makes it very easy and convenient to find the source code in various places on the Internet. To load the operating system, it was necessary to load the files exactly as described from the downloaded image. The process was accomplished by running `dd if=./msdos.img of=/dev/sdc` on a Linux computer, this instruction mounts the MS-DOS image byte-by-byte onto the SD memory card which is necessary for when the BIOS looks at a specific memory location to start the operating system.

In order to begin the process of loading the Zet Processor hardware onto the FPGA, it is first important to read the documentation and manuals for the Altera DE0 Development Board and accompanying Quartus II software manual.

The process for loading Verilog onto the device is fairly straightforward, that is to compile the Verilog code and debug any miscellaneous warnings and compilation errors and then to utilize the on-board programmer to load the code onto the device. When loading the code it is important to take note of the different ways in which the code can be loaded. If the code is loaded through the Joint Tag Action Group (JTAG) interface, it is important to note that this only temporarily loads the hardware and all progress will be lost after powering down the device. This feature is due to the fact that JTAG is made for testing and only loads values directly into the flip-flops and accompanying hardware but does not save this setup data to the flash memory. Since this method only sets the hardware, it can load the hardware almost instantaneously. The other method for loading hardware is known as Active Serial programming (AS), this method requires that the FPGA device is placed into programming mode which can be done by flipping a switch placed on the development board. AS places the FPGA configuration data into FLASH memory which is read into the device at power up.

4 Results

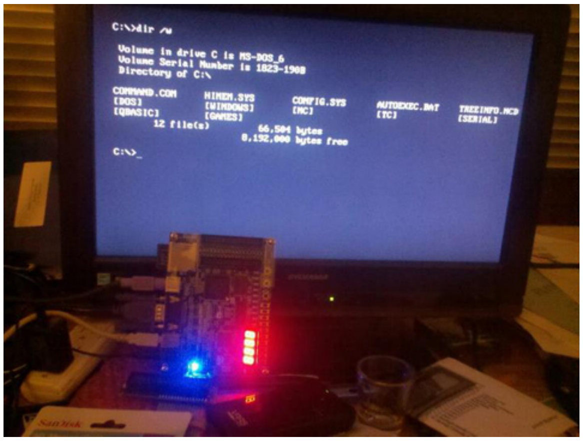
After performing the necessary steps as described by the procedure, the processor is now fully functional and can run an operating system as shown in Figure 3. This was the primary objective for the project and is considered to define the project as an overall success. The next steps were to design presentation materials for demonstration, since MS-DOS includes a program called QBASIC which is a BASIC language interpreter based off of Microsoft's more popular software QuickBASIC, some demonstration programs were written to show off some of the processors capabilities such as the Hello World program and a small game for guessing a random number. The operating system also included some games which could be played through keyboard



Flow Summary	
Flow Status	Successful - Fri Mar 22 23:52:26 2013
Quartus II 64-Bit Version	12.1 Build 243 01/31/2013 SP 1 SJ Web Edition
Revision Name	kotku
Top-level Entity Name	kotku
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	8,434 / 15,408 (55 %)
Total combinational functions	7,858 / 15,408 (51 %)
Dedicated logic registers	2,952 / 15,408 (19 %)
Total registers	2976
Total pins	154 / 347 (44 %)
Total virtual pins	0
Total memory bits	74,547 / 516,096 (14 %)
Embedded Multiplier 9-bit elements	2 / 112 (2 %)
Total PLLs	1 / 4 (25 %)

Figure 2: Compilation results of the Zet Processor from Quartus II software.

input such as Rogue, which is described as a crawling dungeon game with ASCII text based graphics that are easily displayed in terminal where the player has to fight off goblins only disguised as the letter ”G” and collect coins throughout the map [4].



```
C:\>dir /w

 Volume in drive C is MS-DOS_6
 Volume Serial Number is 1823-1908
 Directory of C:\

COMMAND.COM     HIMEM.SYS       CONFIG.SYS      AUTOEXEC.BAT    TREEINFO.NCD
[DOS]           [WINDOWS]       [NC]            [TC]            [SERIAL]
[QBASIC]        [GAMES]
       12 file(s)          66,504 bytes
                        8,192,000 bytes free

C:\>_
```

Figure 3: FPGA loaded with an Intel 8086 microprocessor running MS-DOS 6.22

5 Problems Faced & Troubleshooting

The number one challenge in completing this study was a lack of full documentation and user base for the Zet Processor. It became evident very early on in research that this project would require extensive digging through the source code and custom modifications to get it to work with the Cyclone III device. The last update for the project was over a year ago and the online support forums provided little resource with many of the problems faced when trying to implement the processor. It was also difficult to determine which version of the processor was relevant to the documentation so it was assumed that most of what was found was obsolete or not correct.

Issues in implementing the processor involved problems with the BIOS not compiling due to Open Watcom not being correctly added to the system path, once this was resolved the BIOS compiled correctly. The installation document also does not make clear that it is necessary to convert the BIOS into hexadecimal, when this was done and added into the ROM the processor behaved correctly.

There were also some stability issues with the SD memory, the project site lists a way to mount MS-DOS to the SD card through a program known as Winimage but this failed multiple times and always left the processor hung up at different parts of the boot sequence. This was corrected by loading instead the image byte-by-byte in Linux as mentioned previously, which was not as simple as it sounds since there were also issues with the SD card not being formatted properly. It was finally fixed by formatting the card with all zeros and then the operating system began to work properly.

Lastly, once MS-DOS was finally operational after quite a few hours of debugging it was powered down for the evening and then turned back on in the morning, the result was an error stating that the SD card had become corrupt. This was odd due to the fact that it was operational not but a few hours beforehand. After power cycling the device a few times and with feelings of great defeat that the processor was still not functional, a last ditch effort of removing the SD card and blowing air into the slot and placing the card back into the slot was attempted. The device was then powered back on and worked flawlessly.

6 Conclusion

One of the primary objectives for this study was to gain comprehensive knowledge of the inner working of FPGAs and x86 architecture. By working with the Zet Processor as a working clone of the Intel 8086 hardware it was possible to complete this research within the course of one academic semester. Through this research, it has become clear that a project of this complexity can adequately prepare a person to learn the inner workings of FPGAs. The process of compiling and debugging an early processor gives experience in how to manage a design where hardware may sometimes be limited. It also examines topics in low level device drivers and memory management. Working with the original x86 architecture from the 1970's provided great insight into its relation to the 32-bit and 64-bit versions of x86. It is very clear to see how the industry has changed so much during this time but yet, many of the artifacts from 16-bit architecture are still present in modern versions. Overall, the fact that technology from long ago can still be examined through more modern means such as through FPGAs allows for the technology to continue to be explored and aids in giving great experience in computer architecture design to students interested in pursuing this area of study.

Acknowledgment

The author would like to thank the Altera University Program [5] for providing development boards and necessary software for work on this area of study.

7 Appendix

7.1 Full specifications for Altera DE0 development board:

- FPGA
  - Cyclone III 3C16 FPGA
  - 15,408 LEs
  - 56 M9K Embedded Memory Blocks

    - 504K total RAM bits
    - 56 embedded multipliers
    - 4 PLLs
    - 346 user I/O pins
    - FineLine BGA 484-pin package

- Memory
    - SDRAM
        - One 8-Mbyte Single Data Rate Synchronous Dynamic RAM memory chip
    - Flash memory
        - 4-Mbyte NOR Flash memory
        - Support Byte (8-bits)/Word (16-bits) mode
    - SD card socket
        - Provides both SPI and SD 1-bit mode SD Card access

- Interface
    - Built-in USB Blaster circuit
        - On-board USB Blaster for programming
        - Using the Altera EPM240 CPLD
    - Altera Serial Configuration device
        - Altera EPCS4 serial EEPROM chip
    - Pushbutton switches
        - 3 pushbutton switches
    - Slide switches
        - 10 Slide switches
    - General User Interfaces
        - 10 Green color LEDs
        - 4 seven-segment displays
    - Clock inputs
        - 50-MHz oscillator
    - VGA output
        - Uses a 4-bit resistor-network DAC
        - With 15-pin high-density D-sub connector
        - Supports up to 1280x1024 at 60-Hz refresh rate
    - Serial ports
        - One RS-232 port (Without DB-9 serial connector)
        - One PS/2 port

  - Two 40-pin expansion headers
    - 72 Cyclone III I/O pins, as well as 8 power and ground lines, are brought out to two 40-pin expansion connectors
    - 40-pin header is designed to accept a standard 40-pin ribbon cable used for IDE hard drives

7.2 Available x86 Instructions on the Zet Processor:

Data transfer instructions
mov, push/pop, in/out, lahf/sahf, lds/lea/les, pushf/popf, xchg, xlat
Arithmetic instructions
aaa/aas, aam, aad, daa/das, cbw/cwd, inc, dec, add/adc, sub/sbb, mul/imul, div/idiv, neg, cmp
Bitwise handling instructions
and/or, not, rcl, rcr, rol, ror, sal/shl, sar, shr, test, xor
Control transfer instructions
call, ja/jnbe, jae/jnb/jnc, jb/jnae/jc, jbe/jna, jcxz, je/jz, jg/jnle, jge/jnl, jl/jnge, jle/jng, jne/jnz, jno, jnp/jpo, jns, jmp, jo, jp/jpe, js, loop, loope/loopz, loopne/loopnz, ret
String handling instructions
cmpsb/cmpsw, lodsb/lodswm, movsb/movsw, rep (pref), repe/repz (pref), repne/repnz (pref), scasb/scasw, stosb/stosw
Interrupt instructions
int, into, iret
Microprocessor control instructions
clc, cld, cli, cmc, hlt, nop, stc, std

7.3 Processor BIOS

```
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
// ZET Bios C Helper functions:
// This file contains various functions in C called fromt the zetbios.asm
// module. This module provides support fuctions and special code specific
// to the Zet computer, specifically, special video support and disk support
// for the SD and Flash types of disks.
//
// This code is compatible with the Open Watcom C Compiler.
// Originally modified from the Bochs bios by Zeus Gomez Marmolejo
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------

#include ”zetbios.h”

//--------------------------------------------------------------------------
// Low level assembly functions
```

```
//--------------------------------------------------------------------------------
Bit16u get_CS(void) { __asm { mov  ax, cs } }
Bit16u get_SS(void) { __asm { mov  ax, ss } }

//--------------------------------------------------------------------------------
//  memset of count bytes
//--------------------------------------------------------------------------------
static void memsetb(Bit16u s_segment, Bit16u s_offset, Bit8u value, Bit16u count)
{
    __asm {
                    push ax
                    push cx
                    push es
                    push di
                    mov  cx, count         // count
                    test cx, cx
                    je   memsetb_end
                    mov  ax, s_segment     // segment
                    mov  es, ax
                    mov  ax, s_offset      // offset
                    mov  di, ax
                    mov  al, value         // value
                    cld
                    rep  stosb
     memsetb_end:   pop  di
                    pop  es
                    pop  cx
                    pop  ax
    }
}
//--------------------------------------------------------------------------------
//  memcpy of count bytes
//--------------------------------------------------------------------------------
static void memcpyb(Bit16u d_segment, Bit16u d_offset, Bit16u s_segment, Bit16u s_offset, Bit16u count)
{
    __asm {
                    push ax
                    push cx
                    push es
                    push di
                    push ds
                    push  si
                    mov   cx, count       // count
                    test  cx, cx
                    je    memcpyb_end
                    mov   ax, d_segment   // dest segment
                    mov   es, ax
                    mov   ax, d_offset    // dest offset
                    mov   di, ax
                    mov   ax, s_segment   // ssegment
                    mov   ds, ax
                    mov   ax, s_offset    // soffset
                    mov   si, ax
                    cld
                    rep   movsb
      memcpyb_end:  pop   si
                    pop   ds
                    pop   di
                    pop   es
                    pop   cx
                    pop   ax
    }
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// Low level print functions
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static void wrch(Bit8u character)
{
    __asm {
            push    bx
            mov     ah, 0x0e        // 0x0e command
            mov     al, character
            xor     bx, bx
            int     0x10            // 0x10 intereupt
            pop     bx
    }
}
//--------------------------------------------------------------------------
static void send(Bit16u action, Bit8u  c)
{
    if(action & BIOS_PRINTF_SCREEN) {
        if(c == '\n') wrch('\r');
        wrch(c);
    }
}
//--------------------------------------------------------------------------
static void put_int(Bit16u action, short val, short width, bx_bool neg)
{
    short nval = val / 10;
    if(nval) put_int(action, nval, width - 1, neg);
    else {
        while(--width > 0) send(action, '␣');
        if(neg) send(action, '-');
    }
    send(action, val - (nval * 10) + '0');
}
//--------------------------------------------------------------------------
static void put_uint(Bit16u action, unsigned short val, short width, bx_bool neg)
{
    unsigned short nval = val / 10;
    if(nval) put_uint(action, nval, width - 1, neg);
    else {
        while(--width > 0) send(action, '␣');
        if(neg) send(action, '-');
    }
    send(action, val - (nval * 10) + '0');
}
//--------------------------------------------------------------------------
static void put_luint(Bit16u action, unsigned long val, short width, bx_bool neg)
{
    unsigned long nval = val / 10;
    if(nval) put_luint(action, nval, width - 1, neg);
    else {
        while(--width > 0) send(action, '␣');
        if(neg) send(action, '-');
    }
    send(action, val - (nval * 10) + '0');
}
//--------------------------------------------------------------------------
static void put_str(Bit16u action, Bit16u segment, Bit16u offset)
{
    Bit8u c;
    while(c = read_byte(segment, offset)) {
        send(action, c);
        offset++;
    }
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// bios_printf()  A compact variable argument printf function.
//   Supports %[format_width][length]format
//   where format can be x,X,u,d,s,S,c
//   and the optional length modifier is l (ell)
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static void bios_printf(Bit16u action, Bit8u *s, ...)
{
    Bit8u     c;
    bx_bool   in_format;
    short     i;
    Bit16u   *arg_ptr;
    Bit16u    arg_seg, arg, nibble, hibyte, format_width, hexadd;

    arg_ptr = (Bit16u  *)&s;
    arg_seg = get_SS();

    in_format = 0;
    format_width = 0;

    if((action & BIOS_PRINTF_DEBHALT) == BIOS_PRINTF_DEBHALT)
        bios_printf(BIOS_PRINTF_SCREEN, "FATAL: ");

        while(c = read_byte(get_CS(), (Bit16u)s)) {
        if( c == '%' ) {
            in_format = 1;
            format_width = 0;
        }
        else if(in_format) {
            if( (c >= '0') && (c <= '9') ) {
                format_width = (format_width * 10) + (c - '0');
            }
            else {
                arg_ptr++;              // increment to next arg
                arg = read_word(arg_seg, (Bit16u)arg_ptr);
```

```c
        if(c == 'x' || c == 'X') {
            if(format_width == 0) format_width = 4;
            if(c == 'x') hexadd = 'a';
            else         hexadd = 'A';
            for(i = format_width-1; i >= 0; i--) {
                nibble = (arg >> (4 * i)) & 0x000f;
                send(action, (nibble<=9)? (nibble+'0') : (nibble-10+hexadd));
            }
        }
        else if(c == 'u') {
            put_uint(action, arg, format_width, 0);
        }
        else if(c == 'l') {
            s++;
            c = read_byte(get_CS(), (Bit16u)s);      // is it ld,lx,lu?
            arg_ptr++;                               // increment to next arg
            hibyte = read_word(arg_seg, (Bit16u)arg_ptr);
            if(c == 'd') {
                if(hibyte & 0x8000) put_luint(action, 0L-(((Bit32u) hibyte << 16) | arg), format_width-1, 1);
                else                put_luint(action, ((Bit32u) hibyte << 16) | arg, format_width, 0);
            }
            else if(c == 'u') {
                put_luint(action, ((Bit32u) hibyte << 16) | arg, format_width, 0);
            }
            else if(c == 'x' || c == 'X') {
                if(format_width == 0) format_width = 8;
                if(c == 'x') hexadd = 'a';
                else         hexadd = 'A';
                for(i=format_width-1; i>=0; i--) {
                    nibble = ((((Bit32u) hibyte <<16) | arg) >> (4 * i)) & 0x000f;
                    send(action, (nibble<=9)? (nibble+'0') : (nibble-10+hexadd));
                }
            }
        }
        else if(c == 'd') {
```

```
                    if(arg & 0x8000) put_int(action, -arg, format_width - 1, 1);
                    else             put_int(action, arg, format_width, 0);
                }
                else if(c == 's') {
                    put_str(action, get_CS(), arg);
                }
                else if(c == 'S') {
                    hibyte = arg;
                    arg_ptr++;
                    arg = read_word(arg_seg, (Bit16u)arg_ptr);
                    put_str(action, hibyte, arg);
                }
                else if(c == 'c') {
                    send(action, arg);
                }
                else bios_printf(BIOS_PRINTF_DEBHALT,"bios_printf:␣unknown␣format\n");
                in_format = 0;
            }
        }
        else {
            send(action, c);
        }
        s ++;
    }
    if(action & BIOS_PRINTF_HALT) {  // freeze in a busy loop.
        __asm {
                        cli
            halt2_loop: hlt
                        jmp halt2_loop
        }
    }
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// print_bios_banner -  displays a the bios version
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
#define BIOS_COPYRIGHT_STRING   "(c)␣2009,␣2010␣Zeus␣Gomez␣Marmolejo␣and␣(c)␣2002␣MandrakeSoft␣S.A."
#define BIOS_BANNER             "Zet␣SoC␣BIOS␣-␣build␣date:␣"
```

```
#define BIOS_BUILD_DATE           "31␣Aug␣2010\n"
#define BIOS_VERS                 "␣␣Version:␣v1.1.1:15:g8c8e616\n"
#define BIOS_DATE                 "␣␣Release␣date:␣31␣Aug␣2010\n\n"
void __cdecl print_bios_banner(void)
{
    bios_printf(BIOS_PRINTF_SCREEN,BIOS_BANNER);
    bios_printf(BIOS_PRINTF_SCREEN,BIOS_BUILD_DATE);
    bios_printf(BIOS_PRINTF_SCREEN,BIOS_VERS);
    bios_printf(BIOS_PRINTF_SCREEN,BIOS_DATE);
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// BIOS Boot Specification 1.0.1 compatibility
//
// Very basic support for the BIOS Boot Specification, which allows expansion
// ROMs to register themselves as boot devices, instead of just stealing the
// INT 19h boot vector.
//
// This is a hack: to do it properly requires a proper PnP BIOS and we aren't
// one; we just lie to the option ROMs to make them behave correctly.
// We also don't support letting option ROMs register as bootable disk
// drives (BCVs), only as bootable devices (BEVs).
//
// http://www.phoenix.com/en/Customer+Services/White+Papers-Specs/pc+industry+specifications.htm
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static char drivetypes[][20]={"", "Floppy␣flash␣image", "SD␣card" };
void __cdecl init_boot_vectors(void)
{
    ipl_entry_t e;
    Bit8u       sd_error, switches;
    Bit16u      count = 0;
    Bit16u      hdi, fdi;
    Bit16u      ss = get_SS();

    memsetb(IPL_SEG, IPL_TABLE_OFFSET, 0, IPL_SIZE);  // Clear out the IPL table.

    write_word(IPL_SEG, IPL_BOOTFIRST_OFFSET, 0xFFFF);  // User selected device not set
    sd_error = read_byte(0x40, 0x8d);
    if(sd_error) {
        bios_printf(BIOS_PRINTF_SCREEN,"Error initializing SD card controller (at stage %d)\n", sd_error);

        // Floppy drive
        e.type        = IPL_TYPE_FLOPPY;
        e.flags       = 0;
        e.vector      = 0;
        e.description = 0;
        e.reserved    = 0;
        memcpyb(IPL_SEG, IPL_TABLE_OFFSET + count * sizeof(e), ss, (Bit16u)&e, sizeof(e));
        count++;
    }
    else {            // Get the boot sequence from the switches
        switches = inb(0xf100);
        if(switches) { hdi = 1; fdi = 0; }
        else         { hdi = 0; fdi = 1; }

        e.type = IPL_TYPE_HARDDISK; e.flags = 0; e.vector = 0; e.description = 0; e.reserved = 0;
        memcpyb(IPL_SEG, IPL_TABLE_OFFSET + hdi * sizeof(e), ss, (Bit16u)&e, sizeof(e));

        e.type = IPL_TYPE_FLOPPY; e.flags = 0; e.vector = 0; e.description = 0; e.reserved = 0;
        memcpyb(IPL_SEG, IPL_TABLE_OFFSET + fdi * sizeof(e), ss, (Bit16u)&e, sizeof(e));
        count = 2;
    }
    write_word(IPL_SEG, IPL_COUNT_OFFSET, count);    // Remember how many devices we have
    write_word(IPL_SEG, IPL_SEQUENCE_OFFSET, 1);     // Try to boot first boot device
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// print_boot_failure
//   displays the reason why boot failed
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static void print_boot_failure(Bit16u type, Bit8u reason)
```

```
{
    if(type == 0 || type > 0x03) BX_PANIC("Bad drive type\n");
    printf("Boot failed");
    if(type < 4) {        // Report the reason too
        if(reason == 0)    printf(": not a bootable disk");
        else               printf(": could not read the boot disk");
    }
    printf("\n\n");
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// De-queue the key - Called only by INT16 Key stroke function:
// Takes a key stroke out of the keyboard buffere and returns the value
// If incr is 0, then it just checks for a key in the buffer but does not
// alter the buffer pointers.
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static BOOL __cdecl dequeue_key(Bit8u BASESTK *scan_code, Bit8u BASESTK *
    ascii_code, int incr)
{
    Bit16u buffer_start, buffer_end, buffer_head, buffer_tail;

    buffer_start = read_word(0x0040, 0x0080);
    buffer_end   = read_word(0x0040, 0x0082);
    buffer_head  = read_word(0x0040, 0x001a);
    buffer_tail  = read_word(0x0040, 0x001c);

    if(buffer_head != buffer_tail) {
        *ascii_code  = read_byte(0x0040, buffer_head);
        *scan_code   = read_byte(0x0040, buffer_head+1);
        if(incr) {
            buffer_head += 2;
            if(buffer_head >= buffer_end) buffer_head = buffer_start;
            write_word(0x0040, 0x001a, buffer_head);
        }
        return(1);
    }
    return(0);
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
```

```
// INT16 Support function - Keyboard support routine
// This function checks for if a key has been pressed and is waiting in the
// buffer for processing and returns the appropriate values.
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void __cdecl int16_function(Bit16u rAX, Bit16u rCX, Bit16u rFLAGS)
{
    Bit8u   scan_code, ascii_code;
    Bit8u   shift_flags, led_flags;
    Bit16u  kbd_code;

    shift_flags = read_byte(0x0040, 0x0017);
    led_flags   = read_byte(0x0040, 0x0097);

    switch(GET_AH()) {
        case 0x00:      // read keyboard input
            if(!dequeue_key(&scan_code, &ascii_code, 1)) {          // if
                retirns a 0
                BX_PANIC("KBD: int16h: out of keyboard input\n");   // that
                    means no key strokes waiting
            }
            if(scan_code !=0 && ascii_code == 0xF0) ascii_code = 0;
            else if(ascii_code == 0xE0)             ascii_code = 0;
            kbd_code = (scan_code << 8) | ascii_code;
            SET_AX(kbd_code);
            break;

        case 0x01:      // check keyboard status
            if(dequeue_key(&scan_code, &ascii_code, 0)) {   // We have received
                 a key
                if(scan_code !=0 && ascii_code == 0xF0) ascii_code = 0;
                else if(ascii_code == 0xE0)             ascii_code = 0;
                kbd_code = (scan_code << 8) | ascii_code;
                SET_AX(kbd_code);
                CLEAR_ZF();
            }
            else {              // if dequeue returns 0 then no key is waiting
                SET_ZF();       // Setting the zero flag means no key strokes
                    waiting
            }
            break;

        case 0x02:      // get shift flag status
            shift_flags = read_byte(0x0040, 0x17);
            SET_AL(shift_flags);            // Sets the AL register on the stack
            break;

        case 0x05:    // store key-stroke into buffer
            if(!enqueue_key(GET_CH(), GET_CL())) SET_AL(0x01);
            else                                 SET_AL(0x00);
            break;

        case 0x09: // GET KEYBOARD FUNCTIONALITY
            // bit Bochs Description
            //  7    0   reserved
            //  6    0   INT 16/AH=20h-22h supported (122-key keyboard support)
            //  5    1   INT 16/AH=10h-12h supported (enhanced keyboard support)
            //  4    1   INT 16/AH=0Ah supported
            //  3    0   INT 16/AX=0306h supported
            //  2    0   INT 16/AX=0305h supported
            //  1    0   INT 16/AX=0304h supported
            //  0    0   INT 16/AX=0300h supported
            SET_AL(0x30);

            break;

        case 0x10: // read MF-II keyboard input
            if(!dequeue_key(&scan_code, &ascii_code, 1) ) {
                BX_PANIC("KBD:␣int16h:␣out␣of␣keyboard␣input\n");
            }
            if(scan_code !=0 && ascii_code == 0xF0) ascii_code = 0;
            kbd_code = (scan_code << 8) | ascii_code;
            SET_AX(kbd_code);
            break;

        case 0x11:  // check MF-II keyboard status
            if(!dequeue_key(&scan_code, &ascii_code, 0) ) {
                SET_ZF();
                return;
            }
            if(scan_code !=0 && ascii_code == 0xF0) ascii_code = 0;
            kbd_code = (scan_code << 8) | ascii_code;
            SET_AX(kbd_code);
            CLEAR_ZF();
```

```
            break;

        case 0x12: // get extended keyboard status
            shift_flags = read_byte(0x0040, 0x17);
            SET_AL(shift_flags);
            shift_flags = read_byte(0x0040, 0x18) & 0x73;
            shift_flags |= read_byte(0x0040, 0x96) & 0x0c;
            SET_AL(shift_flags);
            break;

        case 0x92:                  // keyboard capability check called by DOS 5.0+ keyb *
            SET_AL(0x80);           // function int16 ah=0x10-0x12 supported
            break;

        case 0xA2:          // 122 keys capability check called by DOS 5.0+ keyb
            break;          // don't change AH : function int16 ah=0x20-0x22 NOT supported

        case 0x6F:
            if(GET_AL() == 0x08) SET_AL(0x02); // unsupported, aka normal keyboard

        default:
            bios_printf(BIOS_PRINTF_INFO,"KBD:␣unsupported␣int␣16h␣function␣%02x\n", GET_AH());
            break;
    }
}

//--------------------------------------------------------------------------------
// Enqueue Key
//--------------------------------------------------------------------------------
static BOOL enqueue_key(Bit8u scan_code, Bit8u ascii_code)
{
    Bit16u buffer_start, buffer_end, buffer_head, buffer_tail, temp_tail;

    buffer_start = read_word(0x0040, 0x0080);
    buffer_end   = read_word(0x0040, 0x0082);
    buffer_head  = read_word(0x0040, 0x001A);
    buffer_tail  = read_word(0x0040, 0x001C);

    temp_tail = buffer_tail;
    buffer_tail += 2;
    if(buffer_tail >= buffer_end) buffer_tail = buffer_start;
    if(buffer_tail == buffer_head) return(0);   // Buffer over run

    write_byte(0x0040, temp_tail, ascii_code);
    write_byte(0x0040, temp_tail+1, scan_code);
    write_word(0x0040, 0x001C, buffer_tail);
    return(1);
}

//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
// INT09 Support function
//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
void __cdecl int09_function(Bit16u rAX)
{
    Bit8u scancode, asciicode, shift_flags;
    Bit8u mf2_flags, mf2_state;

    scancode = GET_AL();    // DS has been set to F000 before call
    if(scancode == 0) {
        BX_INFO("KBD: int09 handler: AL=0\n");
        return;
    }

    shift_flags = read_byte(0x0040, 0x17);
    mf2_flags   = read_byte(0x0040, 0x18);
    mf2_state   = read_byte(0x0040, 0x96);
    asciicode   = 0;

    switch(scancode) {
        case 0x3a:              // Caps Lock press
            shift_flags ^= 0x40;
            write_byte(0x0040, 0x17, shift_flags);
            mf2_flags |= 0x40;
            write_byte(0x0040, 0x18, mf2_flags);
            break;

        case 0xba:              // Caps Lock release
            mf2_flags &= ~0x40;
            write_byte(0x0040, 0x18, mf2_flags);
            break;
        case 0x2a:              // L Shift press
            shift_flags |= 0x02;
            write_byte(0x0040, 0x17, shift_flags);
            break;

        case 0xaa:              // L Shift release
            shift_flags &= ~0x02;
            write_byte(0x0040, 0x17, shift_flags);
            break;

        case 0x36:              // R Shift press
            shift_flags |= 0x01;
            write_byte(0x0040, 0x17, shift_flags);
            break;

        case 0xb6:              // R Shift release
            shift_flags &= ~0x01;
            write_byte(0x0040, 0x17, shift_flags);
            break;

        case 0x1d:              // Ctrl press
            if((mf2_state & 0x01) == 0) {
                shift_flags |= 0x04;
                write_byte(0x0040, 0x17, shift_flags);
                if(mf2_state & 0x02) {
                    mf2_state |= 0x04;
                    write_byte(0x0040, 0x96, mf2_state);
                }
                else {
                    mf2_flags |= 0x01;
                    write_byte(0x0040, 0x18, mf2_flags);
                }
            }
            break;

        case 0x9d: // Ctrl release
            if((mf2_state & 0x01) == 0) {
                shift_flags &= ~0x04;
                write_byte(0x0040, 0x17, shift_flags);
                if(mf2_state & 0x02) {
                    mf2_state &= ~0x04;
                    write_byte(0x0040, 0x96, mf2_state);
                }
                else {
                    mf2_flags &= ~0x01;
                    write_byte(0x0040, 0x18, mf2_flags);
                }
            }
            break;

        case 0x38: // Alt press
            shift_flags |= 0x08;
            write_byte(0x0040, 0x17, shift_flags);
            if(mf2_state & 0x02) {
                mf2_state |= 0x08;
                write_byte(0x0040, 0x96, mf2_state);
            }
            else {
                mf2_flags |= 0x02;
                write_byte(0x0040, 0x18, mf2_flags);
            }
            break;

        case 0xb8: // Alt release
            shift_flags &= ~0x08;
            write_byte(0x0040, 0x17, shift_flags);
            if(mf2_state & 0x02) {
                mf2_state &= ~0x08;
                write_byte(0x0040, 0x96, mf2_state);
            }
            else {
                mf2_flags &= ~0x02;
                write_byte(0x0040, 0x18, mf2_flags);
            }
            break;

        case 0x45: // Num Lock press
            if((mf2_state & 0x03) == 0) {
                mf2_flags |= 0x20;
                write_byte(0x0040, 0x18, mf2_flags);
                shift_flags ^= 0x20;
                write_byte(0x0040, 0x17, shift_flags);
            }
            break;

        case 0xc5: // Num Lock release
            if((mf2_state & 0x03) == 0) {
                mf2_flags &= ~0x20;
                write_byte(0x0040, 0x18, mf2_flags);
            }
            break;

        case 0x46: // Scroll Lock press
            mf2_flags |= 0x10;
            write_byte(0x0040, 0x18, mf2_flags);
            shift_flags ^= 0x10;
            write_byte(0x0040, 0x17, shift_flags);
            break;

        case 0xc6: // Scroll Lock release
            mf2_flags &= ~0x10;
            write_byte(0x0040, 0x18, mf2_flags);
            break;

        default:
            if(scancode & 0x80) {
                break; // toss key releases ...
            }
            if(scancode > MAX_SCAN_CODE) {
                bios_printf(BIOS_PRINTF_INFO,"KBD:␣int09h_handler():␣unknown␣
                    scancode␣read:␣0x%02x!\n", scancode);
                return;
            }
            if(shift_flags & 0x08) { // ALT
                asciicode = scan_to_scanascii[scancode].alt;
                scancode = scan_to_scanascii[scancode].alt >> 8;
            }
            else if(shift_flags & 0x04) { // CONTROL
                asciicode = scan_to_scanascii[scancode].control;
                scancode = scan_to_scanascii[scancode].control >> 8;
            }
            else if(((mf2_state & 0x02) > 0) && ((scancode >= 0x47) && (
               scancode <= 0x53))) {
                asciicode = 0xe0;    // extended keys handling
                scancode = scan_to_scanascii[scancode].normal >> 8;
            }
            else if(shift_flags & 0x03) { // LSHIFT + RSHIFT
```

```
                // check if lock state should be ignored  because a SHIFT key are pressed
                if(shift_flags & scan_to_scanascii[scancode].lock_flags) {
                    asciicode = scan_to_scanascii[scancode].normal;
                    scancode = scan_to_scanascii[scancode].normal >> 8;
                }
                else {
                    asciicode = scan_to_scanascii[scancode].shift;
                    scancode = scan_to_scanascii[scancode].shift >> 8;
                }
            }
            else {          // check if lock is on
            if(shift_flags & scan_to_scanascii[scancode].lock_flags) {
                asciicode = scan_to_scanascii[scancode].shift;
                scancode = scan_to_scanascii[scancode].shift >> 8;
            }
            else {
                asciicode = scan_to_scanascii[scancode].normal;
                scancode = scan_to_scanascii[scancode].normal >> 8;
            }
        }
        if(scancode==0 && asciicode==0) {
            BX_INFO("KBD:␣int09h_handler():␣scancode␣&␣asciicode␣are␣zero?\n");
        }
        enqueue_key(scancode, asciicode);
        break;
    }
    if((scancode & 0x7f) != 0x1d) mf2_state &= ~0x01;
    mf2_state &= ~0x02;
    write_byte(0x0040, 0x96, mf2_state);
}

//------------------------------------------------------------------------
//------------------------------------------------------------------------
// INT13 Interupt handler function
//------------------------------------------------------------------------
//------------------------------------------------------------------------
#define SET_DISK_RET_STATUS(status) write_byte(0x0040, 0x0074, status)
//------------------------------------------------------------------------
void __cdecl int13_harddisk(rDS, rES, rDI, rSI, rBP, rBX, rDX, rCX, rAX, rIP, rCS, rFLAGS)
Bit16u rDS, rES, rDI, rSI, rBP, rBX, rDX, rCX, rAX, rIP, rCS, rFLAGS;
{
    Bit8u    drive, num_sectors, sector, head, status;
    Bit8u    drive_map, sd_error;
    Bit8u    n_drives;
    Bit16u   max_cylinder, cylinder;
    Bit16u   hd_cylinders;
    Bit8u    hd_heads, hd_sectors;
    Bit8u    sector_count;
    Bit16u   tempbx;
    Bit16u   addr_l, addr_h;
    Bit32u   log_sector;
    Bit8u    tmp;

    SET_IF();   // Turn on IF when Flag Register is popped off the stack

    write_byte(0x0040, 0x008e, 0);  // clear completion flag

    // at this point, DL is >= 0x80 to be passed from the floppy int13h handler code
    // check how many disks first (cmos reg 0x12), return an error if drive not present
    sd_error = read_byte(0x40, 0x8d);
    if(sd_error) drive_map = 0;
    else         drive_map = 1;

    n_drives = 1;

    if(!(drive_map & (1<<(GET_DL()&0x7f)))) {    // allow 0, 1, or 2 disks
        SET_AL(0x01);                            // Set AL register while on the stack frame
        SET_DISK_RET_STATUS(0x01);
        SET_CF();                                // error occurred
        return;
    }

    switch(GET_AH()) {       // AH = Disk command

        case 0x00:                               // disk controller reset
            SET_AH(0x00);                        // Success
            SET_DISK_RET_STATUS(0);              //
            set_diskette_ret_status(0);
            set_diskette_current_cyl(0, 0);      // current cylinder, diskette 1
            set_diskette_current_cyl(1, 0);      // current cylinder, diskette 2
            CLEAR_CF();                          // successful
            break;

        case 0x01:                                  // read disk status
            status = read_byte(0x0040, 0x0074);     //
            SET_AH(status);                         // Return last status
            SET_DISK_RET_STATUS(0);                 //
            if(status) { SET_CF();   }              // set CF if error status read
            else       { CLEAR_CF(); }              //
            break;

        case 0x04:                                  // verify disk sectors
        case 0x02:                                  // read disk sectors
            drive        = GET_DL();                // Get drive number
            hd_cylinders = HD_CYLINDERS;            // get_hd_geometry(drive, &hd_cylinders, &hd_heads, &hd_sectors);
            hd_heads     = HD_HEADS;                // fixed geometry:
            hd_sectors   = HD_SECTORS;              // Hard drive sectors
            num_sectors  =  GET_AL();               // Number of sectors requested
            cylinder     = (GET_CL() & 0x00c0) << 2 | GET_CH();
            sector       = (GET_CL() & 0x3f);
            head         =  GET_DH();

            if((cylinder >= hd_cylinders) || (sector > hd_sectors) || (head >= hd_heads)) {
                SET_AH(0x01);
                SET_DISK_RET_STATUS(1);
                SET_CF();                       // error occurred
                return;
            }
            if(GET_AH() == 0x04 ) {
                SET_AH(0x00);
                SET_DISK_RET_STATUS(0);
                CLEAR_CF();
                return;
            }
            log_sector = ((Bit32u)cylinder) * ((Bit32u)hd_heads) * ((Bit32u)hd_sectors)
                       + ((Bit32u)head) * ((Bit32u)hd_sectors) + ((Bit32u)sector) - 1;

            sector_count = 0;
            tempbx = rBX;

        __asm { sti }  //;; enable higher priority interrupts
        while(1) {
            addr_l = ((Bit16u) log_sector) << 9;
            addr_h =  (Bit16u) (log_sector >> 7);

            __asm {
               mov    es, rES                   // ES: destination segment
               mov    di, tempbx                // DI: destination offset from bx
               cmp    di, 0xfe00                // adjust if there will be an overrun
               jbe    i13_f02_no_adjust

            i13_f02_adjust:
                sub    di, 0x0200               // sub 512 bytes from offset
                mov    ax, es
                add    ax, 0x0020               // add 512 to segment
                mov    es, ax

            i13_f02_no_adjust:
                mov    bx, addr_l
                mov    cx, addr_h

                mov    dx, 0x0100           // SD card IO Port
                mov    ax, 0x51             // CS = 0, command CMD17
                out    dx, ax
                mov    al, ch               // addr[31:24]
                out    dx, al
                mov    al, cl               // addr[23:16]
                out    dx, al
                mov    al, bh               // addr[15:8]
                out    dx, al
                mov    al, bl               // addr[7:0]
                out    dx, al
                mov    al, 0x0ff            // CRC (not used)
                out    dx, al
                out    dx, al               // wait

            i13_f02_read_res_cmd17:
                in     al, dx               // card response
                cmp    al, 0
                jne    i13_f02_read_res_cmd17
            i13_f02_read_tok_cmd17:      // read data token: 0xfe
                in    al, dx
                cmp   al, 0x0fe
                jne   i13_f02_read_tok_cmd17
                mov   cx, 0x100

            i13_f02_read_bytes:
                in    al, dx                    // low byte
                mov   bl, al
                in    al, dx                    // high byte
                mov   bh, al
                mov   word ptr es:[di], bx      // eseg
                add   di, 2
                loop  i13_f02_read_bytes

                mov   ax, 0xffff     //; we are done, retrieve checksum
                out   dx, al         //; Checksum, 1st byte
                out   dx, al         //; Checksum, 2nd byte
                out   dx, al         //; wait
                out   dx, al         //; wait
                out   dx, ax         //; CS = 1 (disable SD)

            i13_f02_done:            //;; store real DI register back to temp bx
                mov  tempbx, di
            }
            sector_count++;
            log_sector++;
            num_sectors--;
            if(num_sectors) continue;
            else            break;
        }
        SET_AH(0x00);                       // Indicate success
        SET_DISK_RET_STATUS(0);             // Set status
        SET_AL(sector_count);               // return sector count done
        CLEAR_CF();                         // successful
        break;

   case 0x03:                              // write disk sectors
        drive        = GET_DL();            // get_hd_geometry(drive, &hd_cylinders, &hd_heads, &hd_sectors);
        hd_cylinders = HD_CYLINDERS;        // fixed geometry:
        hd_heads     = HD_HEADS;
```

```
    hd_sectors   = HD_SECTORS;

    num_sectors = GET_AL();
    cylinder    = GET_CH();
    cylinder   |= (((Bit16u) GET_CL()) << 2) & 0x300;
    sector      = (GET_CL() & 0x3f);
    head        = GET_DH();

    if((cylinder >= hd_cylinders) || (sector > hd_sectors) || (head >=
       hd_heads)) {
        SET_AH(0x01);
        SET_DISK_RET_STATUS(1);
        SET_CF();                       // error occurred
        return;
    }
    log_sector = ((Bit32u)cylinder) * ((Bit32u)hd_heads) * ((Bit32u)
       hd_sectors)
            + ((Bit32u)head) * ((Bit32u)hd_sectors) + ((Bit32u)
                sector) - 1;

    sector_count = 0;
    tempbx = rBX;

    __asm { sti }  //;; enable higher priority interrupts
    while(1) {
        addr_l = ((Bit16u) log_sector)<< 9;
        addr_h =  (Bit16u)(log_sector >> 7);

        __asm {
                mov   es, rES          //;; ES: source segment
                mov   si, tempbx       //;; SI: source offset from
                    temp bx
                cmp   si, 0xfe00       //;; adjust if there will be an
                     overrun
                jbe   i13_f03_no_adjust

        i13_f03_adjust:
                sub   si, 0x0200     //; sub 512 bytes from offset
                mov   ax, es
                add   ax, 0x0020     //; add 512 to segment
                mov   es, ax

        i13_f03_no_adjust:
                mov     bx, addr_l
                mov     cx, addr_h

                mov     dx, 0x0100      //; SD card Port
                mov     ax, 0x58        //; CS = 0, SD card command CMD24
                out     dx, ax
                mov     al, ch          //; addr[31:24]
                out     dx, al
                mov     al, cl          //; addr[23:16]
                out     dx, al
                mov     al, bh          //; addr[15:8]
                out     dx, al
                mov     al, bl          //; addr[7:0]
                out     dx, al
                mov     al, 0xff        //; CRC (not used)
                out     dx, al
                out     dx, al          //; wait

        i13_f03_read_res_cmd24:
                in      al, dx          //; command response
                cmp     al, 0
                jne     i13_f03_read_res_cmd24
                mov     al, 0xff        //; wait
                out     dx, al
                mov     al, 0xfe        //; start of block: token 0xfe
                out     dx, al
                mov     cx, 0x100

        i13_f03_write_bytes:
                mov     ax, word ptr es:[si]        // eseg
                out     dx, al
                mov     al, ah
                out     dx, al
                add     si, 2
                loop    i13_f03_write_bytes

                mov     al, 0xff            //; send dummy checksum
                out     dx, al
                out     dx, al

                in      al, dx              //; data response
                and     al, 0x0f
                cmp     al, 0x05
```

```
                        je      i13_f03_good_write
                        hlt                         //; problem writing

                i13_f03_good_write:                 //; write finished?
                        in      al, dx
                        cmp     al, 0
                        je      i13_f03_good_write

                        mov     ax, 0xffff          //; goodbye mr. writer!
                        out     dx, al              //; wait
                        out     dx, al              //; wait
                        out     dx, ax              //; CS = 1 (disable SD)

                i13_f03_done:   //;; store real SI register back to temp bx
                        mov   tempbx, si
                }

            sector_count++;
            log_sector++;
            num_sectors--;
            if(num_sectors) continue;
            else            break;
        }
        SET_AH(0x00);                   // Return success
        SET_DISK_RET_STATUS(0);         // Set Status
        SET_AL(sector_count);           // Return sectors done
        CLEAR_CF();                     // successful
        break;

        case 0x08:                              // Get Current Drive Parameters
            drive        = GET_DL();            // same as get_hd_geometry(drive, &hd_cylinders, &hd_heads, &hd_sectors);
            hd_cylinders = HD_CYLINDERS;    // fixed geometry:
            hd_heads     = HD_HEADS;
            hd_sectors   = HD_SECTORS;
            max_cylinder = hd_cylinders - 2; // 0 based
            SET_AL(0x00);
            tmp = (Bit8u)(max_cylinder & 0xff);
            SET_CH(tmp);
            tmp = (Bit8u)(((max_cylinder >> 2) & 0xc0) | (hd_sectors & 0x3f));
            SET_CL(tmp);
            tmp = (hd_heads - 1);
            SET_DH(tmp);
            SET_DL(n_drives);        // returns 0, 1, or 2 hard drives
            SET_AH(0x00);
            SET_DISK_RET_STATUS(0);
            CLEAR_CF();              // successful
            break;

        case 0x09:           // initialize drive parameters
        case 0x0c:           // seek to specified cylinder
        case 0x0d:           // alternate disk reset
        case 0x10:           // check drive ready
        case 0x11:           // recalibrate
            SET_AH(0x00);
            SET_DISK_RET_STATUS(0);
            CLEAR_CF();                      // successful
            break;

        case 0x14:                           // controller internal diagnostic
            SET_AH(0x00);                    // Status
            SET_DISK_RET_STATUS(0);
            CLEAR_CF();                      // successful
            SET_AL(0x00);                    // Probably not needed
            break;

        case 0x15:                               // read disk drive size
            drive         = GET_DL();            // same as get_hd_geometry(drive, &hd_cylinders, &hd_heads, &hd_sectors);
            hd_cylinders = HD_CYLINDERS;     // fixed geometry:
            hd_heads      = HD_HEADS;
            hd_sectors    = HD_SECTORS;

            __asm {
                    mov   al, hd_heads             //;; al = heads
                    mov   bl, hd_sectors           //;; bl = sectors
                    mul   bl                       //;; ax = al * bl = heads * sectors
                    mov   bx, hd_cylinders         //;; bx = cylinders
                    dec   bx                       //;; bx = cylinders - 1
                    mul   bx                       //;; dx:ax = bx*ax = (cylinders-1) * (heads * sectors)
                    mov   ss:rCX, dx               //;; BIOS wants 32bit result in CX:DX
                    mov   ss:rDX, ax               //;; which will be returned on the stack
            }

            SET_AH(0x03);              // hard disk accessible
            SET_DISK_RET_STATUS(0); // ??? should this be 0
            CLEAR_CF();                // successful
            break;

        default:
            BX_INFO("int13_harddisk:␣function␣%02xh␣unsupported,␣returns␣fail\n", GET_AH());
            SET_AH(0x01); // defaults to invalid function in AH or invalid parameter
            SET_DISK_RET_STATUS(GET_AH());
            SET_CF();      // error occurred
            break;
    }
}

//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
//  Transfer Sector drive
//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
static void transf_sect_drive_a(Bit16u s_segment, Bit16u s_offset)
{
    __asm {
                push  ax
                push  bx
                push  cx
                push  dx
                push  di
                push  ds

                mov  ax, s_segment        // segment
                mov  ds, ax
                mov  bx, s_offset         // offset
                cmp  bx, 0xfe00           // adjust if there will be an overrun
                jbe  transf_no_adjust

                sub   bx, 0x0200          // sub 512 bytes from offset
                mov   ax, ds
                add   ax, 0x0020          // add 512 to segment
                mov   ds, ax
    transf_no_adjust:
                mov  dx, 0xe000
                mov  cx, 256
                xor  di, di
    one_sect:   in   ax, dx             // read word from flash
                mov  ds:[bx+di], ax     // write word
                inc  dx
                inc  dx
                inc  di
                inc  di
                loop one_sect
                pop  ds
                pop  di
                pop  dx
                pop  cx
                pop  bx
                pop  ax
    }
}

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
// The principle of this routine is to copy directly from flash to the ram disk
// Using the same call that is used to read the flash disk. This routine is
// called from The assembly section during post. It is commented out here
// Because it was also commented out in the original zet bios and I tried
// uncommenting it there and building the old way and it did not work. It does
// not work here either. I have not been able to debug it. Maybe someone can
// figure it out. It would be nice to have, but it is not working right now.
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
void MakeRamdisk(void)
{
/*
    Bit16u Sector, base_count;
    outb(EMS_ENABLE_REG, EMS_ENABLE_VAL);                 // Turn on EMS from 0
        xB0000 - 0xBFFFF
    for(Sector = 0; Sector < SECTOR_COUNT; Sector++) {   // Configure the sector
         address
        outw(FLASH_PAGE_REG, Sector);                     // Select the Flash
            Disk Sector
        base_count = GetRamdiskSector(Sector);  // Select the Flash Page and get the address within the page of the Sector
        transf_sect_drive_a(EMS_SECTOR_OFFSET, base_count);     // We now have the correct page of flash selected and the sector is always in the same place so just pass the place to copy it too
    }
*/
}
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
// The RAM Disk is stored at 0x110000 to 0x277FFF in the SDRAM
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
static Bit16u GetRamdiskSector(Bit16u Sector)
{
    Bit16u Page;
    // The bits above the upper five bits tells us which memory location
    // The lower five bits tells us where in the 16K Page the Sector is
    Page = RAM_DISK_BASE + (Sector >> 5);
    outb(EMS_PAGE1_REG, Page);          // Set the first 16K
    return((Sector & 0x001F) << 9); // Return the memory location within the sector
}

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
// INT13 Diskette service function
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
void __cdecl int13_diskette_function(rDS, rES, rDI, rSI, rBP, rBX, rDX, rCX, rAX, rIP, rCS, rFLAGS)
Bit16u rDS, rES, rDI, rSI, rBP, rBX, rDX, rCX, rAX, rIP, rCS, rFLAGS;
{
    Bit8u  drive, num_sectors, track, sector, head;
    Bit8u  drive_type, num_floppies;
    Bit16u last_addr, base_address, base_count;
    Bit16u log_sector, j, RamAddress;

    SET_IF();    // Turn on IF when Flag Register is popped off the stack
    switch(GET_AH()) {

        case 0x00:                    // Disk controller reset
```

```
            drive = GET_DL();       // Was here but that meant that drive was not set for other cases
            set_diskette_ret_status(0);
            set_diskette_current_cyl(drive, 0); // Current cylinder
            SET_AH(0);                          // disk operation status (see ~INT 13,STATUS~)
            CLEAR_CF();                         // CF = 0 if Successful
            break;

        case 0x01:                              // Disk status
            set_diskette_ret_status(0);
            SET_AL(0);                          // no error
            CLEAR_CF();                         // CF = 0 if Successful
            break;

        case 0x02:                      // Read Diskette Sectors
            num_sectors = GET_AL();     // number of sectors to read (1-128 dec.)
            track       = GET_CH();     // track/cylinder number (0-1023 dec., see below)
            sector      = GET_CL();     // CL = sector number (1-17 dec.)
            head        = GET_DH();     // head number (0-15 dec.)
            drive       = GET_DL();     // drive number (0=A:, 1=2nd floppy, 80h=drive 0, 81h=drive 1)

            if((drive > 1) || (head > 1) || (sector == 0) || (num_sectors == 0) || (num_sectors > 72)) {
                BX_INFO("int13_diskette: read/write/verify: parameter out of range\n");
                set_diskette_ret_status(1);
                SET_AH(1);
                SET_AL(0);          // No sectors have been read
                SET_CF();           // An error occurred
                return;
            }

            log_sector  = track * 36 + head * 18 + sector - 1;  // Calculate the first sector we are going to read
            if(drive == DRIVE_A) {      // This is the Flash Based Drive
                for(j = 0; j < num_sectors; j++) {
                    outw(FLASH_PAGE_REG, log_sector + j);       // We now have the correct page of flash selected
                    transf_sect_drive_a(rES, (rBX + (j << 9))); // now just pass the place to copy it too, j<<9 is the same thing as multiplying by 512
                }                                               // a good optimizing compiler probably does this for you anyway
            }
            else {                      // This is the SDRAM based drive
                base_address = (rES << 4) + rBX;          // Base Address is upper 12 bits of segment + offset
                base_count   = (num_sectors * 512);       // Number of bytes to be transfered
                last_addr = base_address + base_count -1; // Compute the last address is in the same segment
                if(last_addr < base_address) {            // If the last address is less than the base then there must have been an overflow above !
                    BX_INFO("int13_diskette - 03: 64K boundary overrun\n");
                    SET_AH(0x09);
                    set_diskette_ret_status(0x09);
                    SET_AL(0x00);                                   // No sectors have been read
                    SET_CF();                                       // An error occurred
                    return;
                }
                for(j = 0; j < num_sectors; j++) {
                    BX_INFO("int13_diskette - 02: Accessing ramdisk\n");
                    RamAddress = GetRamdiskSector(log_sector + j);  // Pass in the sector which will set the right RAM page and give back the ram address
                    base_count = base_address + (j << 9);
                    memcpyb(last_addr, base_count, EMS_SECTOR_OFFSET,
                        RamAddress, SECTOR_SIZE);  // Copy the sector
                }
            }
            set_diskette_current_cyl(drive, track); // ??? should track be new val from return_status[3] ?
            SET_AH(0);       // AH = 0, sucess AL = number of sectors read (same value as passed)
            CLEAR_CF();      // success
            break;

        case 0x08:                      // read diskette drive parameters
            drive = GET_DL();          //BX_DEBUG_INT13_FL("floppy f08\n");
            if(drive > 1) {
                BX_INFO("int13_diskette - 08: drive >1\n");
                SET_AX(0);
                SET_BX(0);
                SET_CX(0);
                SET_DX(0);
                SET_WORD(rES, 0);
                SET_WORD(rDI, 0);
                SET_DL(num_floppies);
                SET_CF();
                return;
            }
            drive_type = 0x44;         /// inb_cmos(0x10);
            num_floppies = 0;
            if(drive_type & 0xf0) num_floppies++;
            if(drive_type & 0x0f) num_floppies++;
            if(drive == 0) drive_type >>= 4;
            else           drive_type &= 0x0f;
            SET_BH(0);
            SET_BL(drive_type);        // CMOS Drive type
            SET_AH(0);
            SET_AL(0);
            SET_DL(num_floppies);
            switch(drive_type) {
                case 0:                        // none
                    SET_CX(0x00);              // N/A
                    SET_DH(0x00);              // max head #
                    break;

                case 1:                        // 360KB, 5.25"
                    SET_CX(0x2709);            // 40 tracks, 9 sectors
                    SET_DH(0x01);              // max head #
                    break;

                case 2:                        // 1.2MB, 5.25"
                    SET_CX(0x4f0f);            // 80 tracks, 15 sectors
                    SET_DH(0x01);              // max head #
                    break;

                case 3:                        // 720KB, 3.5"
                    SET_CX(0x4f09);            // 80 tracks, 9 sectors
                    SET_DH(0x01);              // max head #
```

```
                    break;

                case 4:                              // 1.44MB, 3.5"
                    SET_CX(0x4f12);                  // 80 tracks, 18 sectors
                    SET_DH(0x01);                    // max head #
                    break;

                case 5:                              // 2.88MB, 3.5"
                    SET_CX(0x4f24);                  // 80 tracks, 36 sectors
                    SET_DH(0x01);                    // max head #
                    break;

                case 6:                              // 160k, 5.25"
                    SET_CX(0x2708);                  // 40 tracks, 8 sectors
                    SET_DH(0x00);                    // max head #
                    break;

                case 7:                              // 180k, 5.25"
                    SET_CX(0x2709);                  // 40 tracks, 9 sectors
                    SET_DH(0x00);                    // max head #
                    break;

                case 8:                              // 320k, 5.25"
                    SET_CX(0x2708);                  // 40 tracks, 8 sectors
                    SET_DH(0x01);                    // max head #
                    break;

                default:                             // Somthing went wrong
                    BX_PANIC("floppy: int13: bad floppy type\n");
                    break;
            }
            SET_WORD(rDI, 0xefc7);  // This table is hard coded into the bios at this location
            SET_WORD(rES, 0xf000);  // This is done for compatibility purposes
            CLEAR_CF();             // success, disk status not changed upon success
            break;

        case 0x15:                  // read diskette drive type
            drive = GET_DL();       // BX_DEBUG_INT13_FL("floppy f15\n");
            if(drive > 1) {
                BX_INFO("int13_diskette - 15: drive >1\n");
                SET_AH(0);          // only 2 drives supported
                SET_CF();            // set_diskette_ret_status here ???
                return;
            }
            drive_type = 0x44;             // inb_cmos(0x10);
            if(drive == 0) drive_type >>= 4;
            else           drive_type &= 0x0f;
            if(drive_type == 0) SET_AH(0); // drive not present
            else                SET_AH(1); // drive present, does not support change line
            CLEAR_CF();                     // successful
            break;

        case 0x03:                       // Write disk sector
            num_sectors = GET_AL();      // number of sectors to write (1-128 dec.)
            track       = GET_CH();      // track/cylinder number (0-1023 dec.)
            sector      = GET_CL();      // sector number (1-17 dec., see below)
            head        = GET_DH();      // DH = head number (0-15 dec.)
            drive       = GET_DL();      // drive number (0=A:, 1=2nd floppy, 80h=drive 0, 81h=drive 1)

            if(drive == DRIVE_B) {       // Writing only works on Drive B
                if((drive > 1) || (head > 1) || (sector == 0) || (num_sectors == 0) || (num_sectors > 72)) {
                    BX_INFO("int13_diskette: read/write/verify: parameter out of range\n");
                    SET_AH(0x01);
                    set_diskette_ret_status(1);
                    SET_AL(0x00);                           // No sectors have been read
                    SET_CF();                               // An error occurred
                    return;
                }
                base_address = (rES << 4) + rBX;           // Base Address is upper 12 bits of segment + offset
                base_count   = (num_sectors * 512);        // Number of bytes to be transfered
                last_addr = base_address + base_count -1;  // Compute the last address is in the same segment
                if(last_addr < base_address) {             // If the last address is less than the base then there must have been an overflow above !
                    BX_INFO("int13_diskette - 03: 64K boundary overrun\n");
```

```
                        SET_AH(0x09);
                        set_diskette_ret_status(0x09);
                        SET_AL(0x00);                                   // No sectors have been read
                        SET_CF();                                       // An error occurred
                        return;
                    }
                    log_sector   = track * 36 + head * 18 + sector - 1;    // Calculate the first sector we are going to read

                    // This is the SDRAM based drive
                    for(j = 0; j < num_sectors; j++) {
                        RamAddress = GetRamdiskSector(log_sector + j);    // Pass in the sector which will set the right RAM page and give back the ram address
                        base_count = base_address + (j << 9);
                        memcpyb(EMS_SECTOR_OFFSET, RamAddress, rES, base_count, SECTOR_SIZE);          // Copy the sector
                    }
                    set_diskette_current_cyl(drive, track);    // ??? should track be new val from return_status[3] ?
                    SET_AH(0x00); // success - AL = number of sectors read (same value as passed)
                    CLEAR_CF();    // success
                    break;
                }
        default:      // If not B Drive, then Fall Through to error message
            BX_INFO("int13_diskette:␣unsupported␣AH=%02x\n", GET_AH());
            SET_AH(0x01); // signal error
            set_diskette_ret_status(1);
            SET_CF();
            break;
    }
}
//---------------------------------------------------------------------------
static void set_diskette_ret_status(Bit8u value)
{
    write_byte(0x0040, 0x0041, value);
}
//---------------------------------------------------------------------------
static void set_diskette_current_cyl(Bit8u drive, Bit8u cyl)
{
    if(drive > 1) drive = 1;     // Temporary hack: for MSDOS
    write_byte(0x0040, 0x0094 + drive, cyl);
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// Get boot vector - only called by INT19 Support Function
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static Bit8u get_boot_vector(Bit16u i, ipl_entry_t BASESTK *e)
{
    Bit16u count;
    Bit16u ss = get_SS();
    count = read_word(IPL_SEG, IPL_COUNT_OFFSET); // Get the count of boot devices, and refuse to overrun the array
    if(i >= count) return(0);                     // OK to read this device
    memcpyb(ss, (Bit16u)e, IPL_SEG, IPL_TABLE_OFFSET + i * sizeof(*e), sizeof(*e));
    return(1);
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// print_boot_device - displays the boot device  - only called by INT19 Support Function
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static void print_boot_device(ipl_entry_t BASESTK *e)
{
    Bit16u type;
    char description[33];
    Bit16u ss = get_SS();
    type = e->type;

    if(type == IPL_TYPE_BEV) type = 0x04; // NIC appears as type 0x80
    if(type == 0 || type > 0x04) BX_PANIC("Bad drive type\n");

    bios_printf(BIOS_PRINTF_SCREEN, "Booting device: %s", drivetypes[type]);

    if(type == 4 && e->description != 0) {     // print product string if BEV, first 32 bytes are significant
        memcpyb(ss, (Bit16u)&description, (Bit16u)(e->description >> 16), (Bit16u)(e->description & 0xffff), 32);
```

```
          description[32] = 0; // terminate string
          bios_printf(BIOS_PRINTF_SCREEN, "␣[%S]", ss, description);
      }
      bios_printf(BIOS_PRINTF_SCREEN, "\n\n");
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// INT19 Support Function
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
void __cdecl int19_function(void)
{
      Bit16u bootdev;
      Bit8u  bootdrv;
      Bit16u bootseg;
      Bit16u bootip;
      Bit16u status;
      ipl_entry_t e;

      // Here we assume that BX_ELTORITO_BOOT is defined, so
      //   CMOS regs 0x3D and 0x38 contain the boot sequence:
      //     CMOS reg 0x3D & 0x0f : 1st boot device
      //     CMOS reg 0x3D & 0xf0 : 2nd boot device
      //     CMOS reg 0x38 & 0xf0 : 3rd boot device
      //   boot device codes:
      //     0x00 : not defined
      //     0x01 : first floppy
      //     0x02 : first harddrive
      //     0x03 : first cdrom
      //     0x04 - 0x0f : PnP expansion ROMs (e.g. Etherboot)
      //     else : boot failure

      bootdev  = read_word(IPL_SEG, IPL_SEQUENCE_OFFSET);    // Read user selected device
      bootdev -= 1;         // Translate from CMOS runes to an IPL table offset by subtracting 1

      if(get_boot_vector(bootdev, &e) == 0) {      // Read the boot device from the IPL table
          printf("Invalid␣boot␣device␣(0x%x)\n", bootdev);
          return;
      }
```

```
    // Do the loading, and set up vector as a far pointer to the boot
    // address, and bootdrv as the boot drive
    print_boot_device(&e);

    switch(e.type) {
        case IPL_TYPE_FLOPPY:    // FDD
        case IPL_TYPE_HARDDISK:  // HDD
            bootdrv = (e.type == IPL_TYPE_HARDDISK) ? 0x80 : 0x00;
            bootseg = 0x07c0;
            status = 0;

            __asm {                     // This little routine loads the DOS
                push ax                 // boot sector from disk into the boot location
                push bx                 // Save the working registers
                push cx
                push dx
                mov  dl, bootdrv        // This is the boot drive
                mov  ax, bootseg        // This is the boot segment
                mov  es, ax             // Load segment into ES
                xor  bx, bx             // Offset is zero
                mov  ah, 0x02           // Disk function 2, read diskette sector
                mov  al, 0x01           // Read 1 sector
                mov  ch, 0x00           // From track 0
                mov  cl, 0x01           // and sector 1
                mov  dh, 0x00           // using head 0
                int  0x13               // Call the read sector bios function
                jnc  int19_load_done    // If Carry flag is clear, then status is good
                mov  ax, 0x0001         // If not then set status flag to bad
                mov  status, ax         // Store it
            int19_load_done:            // Exit the function
                pop  dx                 // By popping our regs
                pop  cx
                pop  bx
                pop  ax
            }

            if(status != 0) {                       // Indicates we had a disk error
                print_boot_failure(e.type, 1);      // show "could not read the boot disk"
                return;
            }

            if(read_word(bootseg, 0x01fe)!= 0xaa55) {    // this is the magic number
                print_boot_failure(e.type, 0);           // "not a bootable disk"
                return;
            }

            bootip   = (bootseg & 0x0fff) << 4;         // Canonicalize bootseg:bootip
            bootseg &= 0xf000;                           // For the right place to jump to
            break;

        default:                                         // if here then the disk is no good
            return;
    }

    BX_INFO("Booting␣from␣%x:%x\n", bootseg, bootip);          // Debugging info

    __asm {                  // This routine Jumps to the boot vector we just loaded
        pushf                // iret pops ip, then cs, then flags, so push them in the opposite order.
        mov  ax, bootseg     // Here is the return segment to jump to
        push ax              // push it so it will get popped when we iret
        mov  ax, bootip      // Jump to the start
        push ax              // again push it for later
        mov  ax, 0xaa55      // Set the magic number in ax and the boot drive in dl.
        mov  dl, bootdrv     // Set the boot drive number
        xor  bx, bx          // Clear BX register
        mov  ds, bx          // Data segment  DS = 0
        mov  es, bx          // Also set ES to 0
        mov  bp, bx          // Base pointer = 0
        iret                 // Now Go!
    }

}
```

```
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// BOOT HALT
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
void __cdecl boot_halt(void)
{
    printf("No more devices to boot - System halted.\n");
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// INT 1A Support function - Time-of-day Service Entry Point
// Input:   AH = 00
// Output:
//          AL = midnight flag, 1 if 24 hours passed since reset
//          CX = high order word of tick count
//          DX = low order word of tick count
//  - incremented approximately 18.206 times per second
//  - at midnight CX:DX is zero
//  - this function can be called in a program to assure the date is
//  updated after midnight; this will avoid the passing two midnights
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
void __cdecl int1a_function(rAX, rCX, rDX, rDI, rSI, rBP, rBX, rDS, rIP, rCS, rFLAGS)
Bit16u rAX, rCX, rDX, rDI, rSI, rBP, rBX, rDS, rIP, rCS, rFLAGS;
{
    Bit16u  ticks_low;
    Bit16u  ticks_high;
    Bit8u   midnight_flag;

    __asm { sti }
    switch(GET_AL()) {
        case 0:             // get current clock count
            __asm { cli }
            ticks_low      = read_word(0x0040, 0x006C);
            ticks_high     = read_word(0x0040, 0x006E);
            midnight_flag = read_byte(0x0040, 0x0070);

            SET_CX(ticks_high);
            SET_DX(ticks_low);
            SET_AL(midnight_flag);

            write_byte(0x0040, 0x0070, 0);  // reset flag
            __asm { sti }
            CLEAR_CF();         // OK  AH already 0
            break;

        default:
            SET_CF(); // Unsupported
    }
}


//---------------------------------------------------------------------------
//  End
//---------------------------------------------------------------------------
```

7.4 Hardware Description for FPGA RAM

```
// --------------------------------------------------------------------
// --------------------------------------------------------------------
// Module:        WB_Flash.v
// Description:  Wishbone Flash RAM core.
// --------------------------------------------------------------------
// --------------------------------------------------------------------
module WB_Flash(
    input                    wb_clk_i,          // Wishbone slave interface
    input                    wb_rst_i,
    input                    [15:0]  wb_dat_i,
    output                   [15:0]  wb_dat_o,
    input                    wb_we_i,
    input          [ 1:0]    wb_adr_i,          // Wishbone address lines
    input          [ 1:0]    wb_sel_i,
    input                    wb_stb_i,
    input                    wb_cyc_i,
    output reg               wb_ack_o,

    output        [21:0]  flash_addr_,          // Pad signals
    input         [15:0]  flash_data_,
    output                flash_we_n_,
    output                flash_oe_n_,
    output                flash_ce_n_,
    output                flash_rst_n_
  );

  assign flash_rst_n_ = 1'b1;
  assign flash_we_n_  = 1'b1;
  assign flash_oe_n_  = !op;
  assign flash_ce_n_  = !op;
  assign flash_addr_  = address;
  assign wb_dat_o     = flash_data_;
  wire   op           = wb_stb_i & wb_cyc_i;
  wire   wr_command   = op       & wb_we_i;            // Wishbone write access Singal

  always @(posedge wb_clk_i or posedge wb_rst_i) begin         // Synchrounous
    if(wb_rst_i) wb_ack_o <= 1'b0;
    else         wb_ack_o <= op & ~wb_ack_o; // one clock delay on acknowledge output
  end

 // --------------------------------------------------------------------
 // Register addresses and defaults
 // --------------------------------------------------------------------
 ‘define FLASH_ALO   2'h1    // Write only - Lower 16 bits of address lines
 ‘define FLASH_AHI   2'h2    // Write only - Upper  6 bits of address lines
  reg  [21:0] address;
  always @(posedge wb_clk_i or posedge wb_rst_i) begin         // Synchrounous
        if(wb_rst_i) begin
        address <= 22'h000000;       // Interupt Enable default
    end
    else if(wr_command) begin                        // If a write was requested
        case(wb_adr_i)                               // Determine which register was writen to
            ‘FLASH_ALO: address[15: 0] <= wb_dat_i;
            ‘FLASH_AHI: address[21:16] <= wb_dat_i[5:0];
            default:    ;                                           // Default
        endcase                                                     // End of case
    end
end  // Synchrounous always

// --------------------------------------------------------------------
endmodule
// --------------------------------------------------------------------
// End of WB Module
// --------------------------------------------------------------------
```

7.5 Hardware Description for Loading Static ROM

```
// --------------------------------------------------------------------------
// --------------------------------------------------------------------------
// Module:      BIOSROM.v
// Description: Wishbone Compatible BIOS ROM core using megafunction ROM
// The following is to get rid of the warning about not initializing the ROM
// altera message_off 10030
// --------------------------------------------------------------------------
// --------------------------------------------------------------------------
module BIOSROM(
    input               wb_clk_i,       // Wishbone slave interface
    input               wb_rst_i,
    input        [15:0] wb_dat_i,
    output       [15:0] wb_dat_o,
    input        [19:1] wb_adr_i,
    input               wb_we_i,
    input               wb_tga_i,
    input               wb_stb_i,
    input               wb_cyc_i,
    input        [ 1:0] wb_sel_i,
    output reg          wb_ack_o
);

wire ack_o = wb_stb_i & wb_cyc_i;
always @(posedge wb_clk_i) wb_ack_o <= ack_o;

reg  [15:0] rom[0:127];         // Instantiate the ROM
initial $readmemh(”zetbios_de0.dat”, rom);

wire   [ 6:0] rom_addr = wb_adr_i[7:1];
wire   [15:0] rom_dat  = rom[rom_addr];
assign        wb_dat_o = rom_dat;

// --------------------------------------------------------------------------
endmodule
// --------------------------------------------------------------------------
```

References

[1] Altera Corporation. (2012, Aug.) Cyclone III device handbook. [Online]. Available: http://www.altera.com/literature/hb/cyc3/cyclone3_handbook.pdf

[2] Z. G. Marmolejo. (2011, Feb.) Zet Processor - Source Code. [Online]. Available: https://github.com/marmolejo/zet

[3] Open Watcom. (2010, Jun.) Welcome to Open Watcom. [Online]. Available: http://www.openwatcom.org/index.php/Main_Page

[4] M. Toy and G. Wichman, “Rogue (video game),” 1980.

[5] Altera Corporation. (2012, Aug.) Altera University Program - Learning Through Innovation. [Online]. Available: http://www.altera.com/education/univ/unv-index.html