An Intel x86 CPU, IBM BIOS, and Hardware Emulator

Steven A. Hanna
hannasa@auburn.edu

J.A. Hamilton, Jr. (Advisor)
hamilton@auburn.edu

November 3, 2011

Abstract

Hardware emulation offers several advantages over native execution: resolving incompatibilities between older applications and new environments, providing features such as snapshots and concurrent execution, and facilitating operating system debugging. An emulator is developed to provide a platform (emulated CPU, BIOS, and misc. hardware) for executing the DOS operating system (specifically FreeDOS[1]) and a subset of DOS-compatible applications in modern operating environments, such as Mac OS X. Techniques to validate this platform are also explored.

1 Introduction

Hardware emulation not only allows software designed for one platform to execute on another, but it also provides several advantages over native execution: machine snapshots, debugging, and virtual disks. This project's primary motivation is to gain a fundamental understanding of the IBM-PC's components and challenges to their emulation, including: expandability, customizability, and potential for optimization. To this end, an application is

[1] http://www.freedos.org

produced which is capable of executing DOS and legacy DOS-compatible applications by emulating the x86 CPU, IO bus, keyboard, mouse, video, PC speaker, hardware timer, etc., and the IBM-compatible BIOS. Care is taken to ensure the application's architecture is modular, allowing new hardware device emulators and software APIs to be added.

2 Why emulation?

2.1 Platform Incompatibilities

Platform incompatibilities, resulting from a change or deprecation in standards, create *portability barriers* for applications. The emulation of hardware and software allows applications designed for one platform to be executed on an otherwise incompatible one (a DOS application on Mac OS X, e.g.) [15].

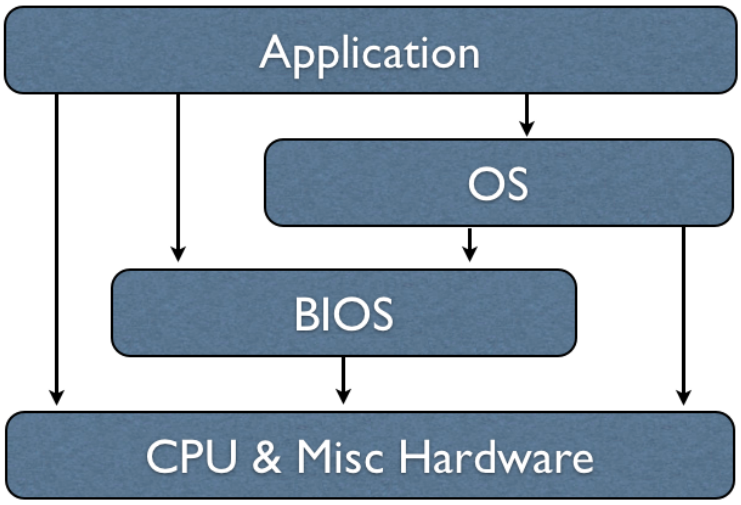

Figure 1: Dependencies

DOS applications, in particular, are susceptible to this issue for the following reasons:

- DOS applications require a DOS-compatible API environment.
- They are composed of instructions requiring the Intel processor's Real Mode.

- Many DOS applications (and DOS) rely upon an IBM-compatible BIOS.
- Either through I/O, or memory-mapped I/O, the underlying hardware becomes a dependency.

Because of these dependencies, a typical DOS application might require specific versions of DOS (system calls), BIOS (service routines), CPU (instructions), and other devices. An emulated environment seeks to mediate these incompatibilities to allow older applications to execute on newer or otherwise incompatible platforms.

2.2 Machine Snapshots

Once the model representing the emulated machine has been constructed, a snapshot of its state at any moment can be preserved. At a later time, this snapshot can be used to revert to the machine's previous state. Multiple snapshots could be made, by the user or on a schedule, and could be used to significantly reduce machine boot time. Machine snapshots can also be used to transfer an emulated machine from one host environment to another. This model is also employed by anti-virus software to detect polymorphic and metamorphic viruses [17].

Snapshots can also facilitate debugging by allowing the developer to record the exact state before an error occurs and jump to it to before testing a possible fix. This feature was used extensively during development to test the emulated BIOS functionality by eliminating the need to complete most of the steps necessary to reproduce the error (test the routine). For example, a bug was discovered in the BIOS routine which reverse-scrolls a text-mode 'window'. To test this routine, the emulator was first booted, and the program which exposed the bug was launched from the command line. Next, the program's menus were navigated to a specific dialog. The cursor was moved to the last item in the scroll view, just before the items were forced to scroll. At this point a machine snapshot was taken. Lastly, the 'Down' key is pressed to invoke the BIOS's scroll routine. For future tests, the machine only needed to be restored from the snapshot, and the 'Down' key pressed. This method significantly reduced debugging time.

2.3 Debugging Operating Systems

Operating systems' close relationship with hardware make them difficult to debug. To be useful, a debugger must read the machine's state (CPU, memory, etc.) and somehow record this, either for immediate usage (video display) or for later reference (to disk). For the debugger to accomplish these tasks, the debugger must reside in the machine's memory and execute on the machine's CPU, etc. It can be difficult for the debugger to do this (be useful) without interfering with the software being debugged.

Emulation significantly mediates these issues by allowing the debugger to exist *external* to the machine being debugged. In such a situation, the debugger could "watch" execution without interfering, could at any point not only stop execution, but also could "freeze" the hardware (similar to a snapshot). This is a new level of debugging. Even more, if the debugger *knew* something about the OS being debugged, it could provide much more useful information than CPU state and memory contents. The QEMU emulator [2], for example, has several options for booting Linux: "use 'bzImage' as kernel image", "use 'cmdline' as kernel command line", "and use 'file' as initial ram disk" and can "freeze CPU at startup".

2.4 Virtual Disks and Hardware Commoditization

Another benefit of emulation is that executing software is not coupled to a physical machine, but an emulated one. To facilitate this, storage media made available to the emulated environment need not be actual physical media, but could be represented by "images" of physical media represented as files, either on disk or in memory. This physical media abstraction allows the emulator to arbitrate reads and writes to and from the storage media. For example, writes could be documented, diverted, ignored, or require user permission to be made. This also allows physical media which are not readily available, such as floppy disks, to continue to be used in the form of disk images. This decoupling of software and hardware allows multiple operating systems to execute concurrently on one physical machine, commoditizing hardware [1].

3 Application User Interface

The majority of the user's interaction with this application is accomplished through a single application window. The emulator displays video output and receives keyboard and mouse input through this single window on the user's desktop. When the user launches the application, he has the opportunity to specify disk image files which are exposed to the emulated environment as storage devices (floppy or hard disks). All key presses and mouse events within this window are captured and delivered to the emulated environment. Closing this window terminates the emulated environment.

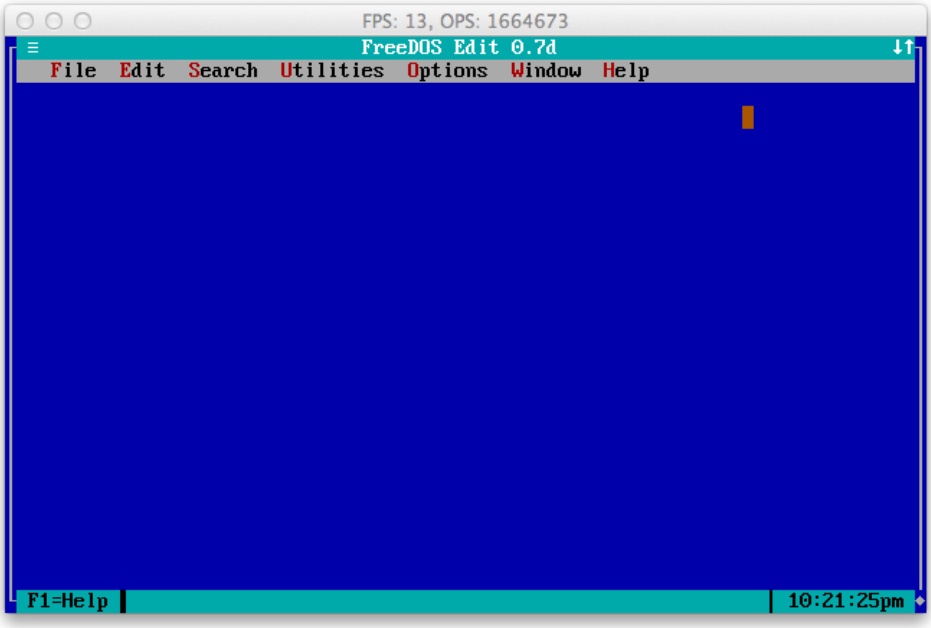

Figure 2: User interface with mouse cursor

3.1 Debug Console

A debug console facilitates inspecting, manipulating, and debugging the emulated environment. The user can inspect memory with the `d` command, display the CPU's registers with `r`, insert a break point with `g`, pause, resume, and step execution with `b`, `c`, and `n`, snapshot/restore the emulated machine from disk with `w` and `l`. `v` displays the interrupt vector table (IVT),

`!` dumps the contents of memory to disk, `i` advances the CPU until a software interrupt occurs, and `q` exits the emulator. The presence of a debug console *outside* of the emulated environment has proved indispensable in the development process.

4 Implementation Details

The application is mostly implemented in C, with some x86 assembly, and has an architecture focused on modularity. Each of the emulated devices is represented as a module, independent of the others. Some software, such as a portion of the BIOS, is also emulated. The Simple DirectMedia Layer (SDL)[2] is an open-source, cross-platform library that is used to provide keyboard/mouse input, video/audio output (as well as timing, threading, and mutexes).

4.1 Languages

The primary language of implementation is C. Speed, extensive support for data manipulation, static typing, and pointer support make C an ideal development language for this project.

Some of the BIOS is implemented in x86 assembly and is executed on the emulated CPU. Rather than as an optimization, this is done to demonstrate the versatility of the emulated environment. As an example, the BIOS keyboard interrupt service routine (ISR) was originally implemented in C (external), but was later rewritten in assembly to execute within the emulated environment (internal).

4.2 Architecture

A major design goal is simple and modular architecture (see Figure 3). To attain this, intermodule dependencies are minimized. A high-level architecture similar to the figure was maintained, and it was consulted throughout the development process to evaluate a change's impact. While this requirement facilitates component reuse, etc., in some cases the insertion of interfaces results in a slight performance regression. This can occur because intermodule

[2]http://www.libsdl.org

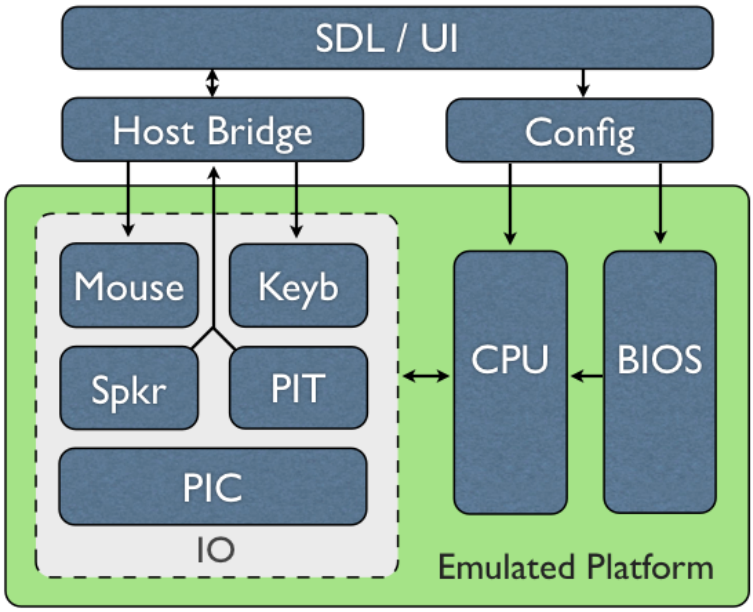

Figure 3: Application Architecture

communication may (by design) need to be intermediated by a third module to reduce coupling and better represent the target platform.

This modular interface is demonstrated in the IO module, which provides a uniform interface for emulated devices to interact with the CPU. During initialization, devices register specific IO ports with the IO module. When an emulated CPU instruction reads or writes (using `in` or `out`) to an IO port, the CPU module delegates these requests to the IO module. Devices can also raise hardware interrupts which are aggregated in the IO module and delivered to the CPU. The IO module functions as an intermediary between the CPU and devices. Another example is the Host Bridge, which translates user interface events between the host environment and the emulated devices.

4.2.1 Threads of Execution

The emulator consists of five threads of execution, two of which serve diagnostic purposes and are not necessary for emulation. The first, the SDL event thread, launches the other threads, receives SDL events (key presses and mouse movements, e.g.), and makes them available to the emulated environment through the Host Bridge module. This thread also draws the screen.

The CPU thread iterates the CPU's instruction loop. The PIT thread is used to raise interrupts (`int 0x08`) when its Counter 0 expires. It produces approximately one interrupt every 54 ms. The console and performance threads are the two diagnostic threads. The console thread provides a debugging interface for the user, allowing the interface to function independent of the rest of the application, so that it can remain responsive to the user. The performance thread calculates the average frames per second rendered and instructions per second executed. These numbers are displayed in the title bar of the application window.

5 Emulated Devices

Because applications make use of devices which are not present in the host environment, several devices must be emulated and bridged with the host environment. As seen in Figure 3, the emulated devices are independent of one another and either interact with the CPU via the IO module, or the host environment (user), via the Host Bridge. The user mostly interacts with the keyboard, mouse, and speaker modules, while the CPU, PIT, floppy, and PIC modules are internal. A description of each emulated device follows:

5.1 Intel x86 Processor

The CPU module emulates over 300 instructions [13], executing in the Intel x86 processor's Real Mode [12]. It fetches instructions from memory, decodes, executes them, and maintains the CPU's state. It also supports the segment override, repeat, operand size override instruction prefixes (to support 32-bit operands), and includes partial support for the address-size override prefix (to support 32-bit addressing).

One iteration of the CPU (for the execution of a single instruction) performs the following steps: First, if the CPU is halting (due to the `HLT` instruction), the CPU thread will wait for the IO module to receive an IRQ from a device module. Next, if the Interrupt flag is set, the CPU module checks the IO module for the presence of a pending hardware interrupt and executes the appropriate ISR to handle it [16]. If the Trap flag is set, `int 0x01` is executed before each instruction. Then, the instruction is fetched and decoding begins. The first step of instruction decoding is the parsing of instruction prefixes (previously mentioned). If the CPU module is in a

repeating state (`REP`, `REPNZ`, `REPE`, etc.), this is handled. The operands of the instruction are then resolved to registers, memory locations, or immediate values. Finally, the instruction is executed and the results are written to the destination operands and/or flags. The CPU thread repeatedly performs these steps while the machine is running.

5.2 VGA-compatible graphics adapter

Video output is provided via an emulated VGA-compatible [11] graphics adapter. Applications, BIOS, and the operating system output text to the adapter via memory-mapped IO. VGA `mode 0x3` (16-color text) has been implemented, and the BIOS video routines and graphics adapter module have been parameterized to facilitate the addition of other modes. By default, `Mode 0x3` consists of 25 rows and 80 columns of 8x16-pixel characters. The screen state is stored in a linear frame buffer at physical memory offset `0xb8000` (real-mode segment `0xb800`). Characters are represented sequentially, in left-right, top-bottom order in this buffer by two bytes each. The first byte indicates the ASCII character code, with the second byte indicating its foreground and background colors [7]. The default frame buffer is 4000[3] bytes in size. The BIOS's video routines (scroll screen, output character, etc.) manipulate this memory area, which the emulated adapter interprets to draw the screen.

5.3 Keyboard and Pointing Device (PS/2 Mouse)

Keyboard input is made available by the emulated 8042 keyboard controller [6]. Keyboard input is captured from the host environment and delivered to the Host Bridge, which forwards it to the Keyboard module. This module then raises an interrupt to the IO module, which leads the CPU to execute the BIOS's corresponding interrupt service routine. This routine reads the keyboard data and command information from IO ports `0x60` and `0x64`, respectively.

Applications generally access the mouse through a third-party mouse driver. FreeDOS includes the CuteMouse[4] driver, which requests mouse movement and button state information from the BIOS. Using the SDL library, mouse events are captured from the host environment and delivered to

[3] 25 rows * 80 characters/row * 2 bytes/character = 4000 bytes

[4] http://cutemouse.sourceforge.net

the Host Bridge, which forwards them to the Mouse module. When a mouse event occurs, the emulated mouse device records the relative axis movement and button state and raises `int 0x74` to trigger the BIOS. The BIOS then reads the event data via IO ports `0x0x70-0x72` and forwards it to the mouse driver, if present. This module is responsible for communicating mouse events to the driver, allowing it to interface with applications and draw the mouse cursor.

5.4 Programmable Interval Timer

The programmable interval timer (PIT) has three counters and is driven by a 1.193182 MHz oscillator [10]. The counters are individually initialized with a count value. When a counter reaches the end of its period, the counter's output pin flashes high and the count is reset, effectively performing frequency division. Counter 0 is initialized with the value `0xffff` (65535) and generates `IRQ 0`, Counter 1 is used to refresh RAM, and Counter 2 is connected to the PC speaker. The PIT module maintains the state of the three counters and notifies the Host Bridge when Counter 2's value has been initialized, indicating the PC speaker's frequency has changed. The PIT can be programmed to operate in other modes, but these have not been encountered during testing and have not been implemented. The PIT module maintains the PIT thread which is responsible for raising `int 0x08` at the end of Counter 0's period (approximately 55 ms[5]).

5.5 PC Speaker

The PC speaker generates tones and can be programmed by software. The PIT's Counter 2 is connected to the PC speaker, but is gated by the two least significant bits of `port 0x61` (the keyboard controller). Consequently, the speaker's frequency is controlled by the value of Counter 2 of the PIT, and it is enabled/disabled depending upon the value at `port 0x61`[6]. The speaker module relays speaker on/off commands to the Host Bridge, which forwards them to the user interface, allowing it to emulate the PC speaker on the host platform.

Support for audio output is provided by the SDL library, which requires that the square-wave output waveform be manually generated. Initially, a

[5] $65535/1.193182MHz * 1000ms/s \approx 55ms$

[6] http://wiki.osdev.org/PC_Speaker

sine-wave was generated, but it sounded too "smooth" compared to actual PC speaker tones. This is because Counter 2 drives the PC speaker with what is effectively a square-wave.

5.6 Floppy Drive Controller and PIC

The floppy drive [9] and programmable interrupt controller (PIC) [8] modules provide almost no perceivable functionality, except that they act as "bit bucket" place-holders. For example, the floppy module receives the `Motor off` command (common in boot loaders) on port `0x3f2`, but simply ignores it. This PIC's command port is `0x20`, and it's data port is `0x21`. Some applications use the PIC to mask interrupts, or map IRQs, but because of the emulated nature of the environment, it has not been necessary to implement most of the PIC's functionality.

6 BIOS

The IBM-PC's BIOS is stored in ROM and mapped into main memory. It provides access to the varying hardware components through a mostly uniform software interface of software interrupts and shared memory [5] [4]. This BIOS is composed of two kinds of interrupt service routines: internal and external. Internal routines are assembled, loaded into the emulator's memory, and executed on the emulated CPU. External routines are part of the emulator; they emulate the routine's expected functionality. The BIOS module emulates the functionality of most BIOS routines by manipulating memory, the CPU, or other devices.

This module provides services for video (`int 0x10`), disk access (`int 0x13`), serial communications (`int 0x14`), mouse driver and power off (`int 0x15`), parallel communications (`int 0x17`), booting (`int 0x19`), and real-time clock date/time access (`int 0x1a`). The clock ticks (`int 0x08`), keyboard (`int 0x09` and `0x16`), mouse (`int 0x74`), equipment list (`int 0x11`), and memory size (`int 0x12`) routines are internal, executing on the emulated CPU. A limited DOS-compatible API is also provided.

6.1 Internal Routines

6.1.1 Counting Clock Ticks

When the PIT's Counter 0 expires, it raises interrupt `0x08`. It is executed approximately 18.206 times per second, each time incrementing the 32-bit value at absolute memory address `0x46c` [5]. This ISR is hooked by some applications to measure time. Other applications reference the value at `0x46c` for a similar purpose. For more precise time measurements, the value of Counter 0 itself can be read directly from the PIT. Both Borland and Watcom's implementations of `delay` were found to use the more precise method, while `EDIT.COM` relies upon the value at `0x46c` for detecting when to redraw the time.

6.1.2 Equipment List and Memory Size

The BIOS maintains a list of present equipment and current configuration. This list is queried by executing `int 0x11`. Currently, the routine responds by indicating the presence of a floppy disk, PS/2 mouse, and 80x25 color text-mode. `int 0x12` returns the value at absolute address `0x413`, indicating the number of kilobytes of memory.

6.1.3 Keyboard and Mouse

The keyboard interrupt service routine is located at `int 0x16` and provides three services: wait for and return key press, check for key press, and get keyboard "shift" flags. Each of these is performed by reading the Keyboard controller via IO. The "wait for key press" service uses the `HLT` instruction rather than polling. This significantly reduces CPU utilization while waiting for user input. The mouse ISR (`int 0x74`) is executed when the Mouse module receives an event from the Host Bridge. It reads the event information (button state, x movement, and y movement) and passes it to the mouse driver, if installed.

6.2 External Routines

The emulated BIOS routines execute outside the emulated environment. These routines are triggered by pushing the ISR number to the stack and writing the 16-bit register `AX` to port `0xffff`. The CPU module detects this

rare instruction sequence and executes the corresponding routine. Each of the emulated ISRs contains entries in the Interrupt Vector Table (IVT) which point to a set of instructions which performs the task above. This level of indirection is necessary so that these emulated routines can be hooked by third-party applications and continue to function.

6.2.1 Video and Disk Services

The BIOS video services are accessible via `int 0x10` and allow applications to get and set the text-mode cursor shape and position. They provide facilities to scroll all or part of the console both "up" and "down". A character and its color attribute can be read from a given screen position, characters can be output to the screen, and the video mode can be queried and set.

Disk services are initialized at the start of the emulator. Any disk images specified at the command-line with either `-fda [path]` for floppy disk, or `-hda [path]` for hard disk are opened and made available through the BIOS. Through `int 0x13` the BIOS's disk services allows software to reset the disk system, query a drive's parameters and type, and read sectors from a disk or write to it. All disk image access is performed via these routines.

6.2.2 Mouse Driver

The BIOS provides the mouse software driver with a uniform method of accessing the PS/2 mouse via `int 0x15`. Through this interface the mouse driver can enable or disable the mouse, set the desired level of counts/mm, and provide the address of the driver's event handler, to be called by interrupt service routine `0x74` when a mouse event occurs.

6.2.3 Booting and Power Off

At boot, `int 0x19` is triggered which initializes the BIOS. This routine sets the video mode to 0x03 (80x25, 16-colors of text) and loads the non-emulated BIOS routines into memory. Next, the BIOS data area is configured. This includes setting the number of kilobytes of memory, the `COM` and `LPT` IO addresses, keyboard type, PS/2 control flags, number of screen rows and columns, and current display page number. The screen is then cleared and BIOS information is printed. Installed devices are then searched for a valid boot sector. If one is found, it is loaded to absolute memory location `0x7c00` and executed. Otherwise, an error is displayed. The supported method for

power off is via APM shutdown. This is triggered via `int 0x15; AX=0x5307, BX=0x0001, CX=0x0003` [5] and terminates the emulator.

6.2.4 Date and Time Access

Access to the real-time clock, or in this case, the host environment's current date and time, is provided via `int 0x1a`. It makes available the current hour, minute, second, and daylight savings flag, in BCD format. The date can also be retrieved as day, month, year, and century, also in BCD [4]. When these routines execute, the host operating system is queried and its values are formatted and returned. Any attempt to set the host environment's real-time clock fails silently.

7 Validation

The accuracy by which an emulator replicates its target platform is key to determining its utility. As the largest, most complicated, and relied upon component of the application, the CPU module was chosen for validation. Although the ability to boot to the `C:\` prompt and edit documents with a mouse in a windowed environment might appear to be validation enough, more formal methods are necessary. Unfortunately, according to a study at Intel, methods for validating the CPU are not trivial:

> However, it was clear from the start that we couldn't formally verify the entire design–that was (and still is) way beyond the state of the art for todays tools. [...] An especially effective method for testing the interactions between sequences of instructions is the Random Instruction Test (RIT) environment. It is not mathematically feasible to even test just the possible combinations of a single instruction with all possible operands and processor states. Add to this the possibility of virtually limitless combinations of instruction sequences and it becomes clear that a systematic and exhaustive test strategy is wholly impractical. A practical and effective alternative is to construct an RIT environment.[3]

7.1 Generating Random Instructions

Following the method outlined at Intel by Bentley and Gray, an RIT tool was constructed to dynamically generate a sequence of 24,000 instructions. This tool is able to generate RITs for over 50 instructions (these 50 mnemonics are assembled into approximately 230 unique machine opcodes) and their many possible parameter types. To generate the RIT sequence, an instruction mnemonic is chosen at random. Next, a legal combination parameters is chosen from instruction's possible parameter list. The generic parameter types are then resolved to their final values; immediate values are randomized and memory addresses are chosen. Each of the twenty-four 16-bit effective memory address types are supported, along with random segment override prefixes. This complete instruction is then added to the random instruction stream.

7.2 Executing the RIT

Once the instructions have been generated, they must be executed on a test machine and their results must be collected. A DOS application was designed to take advantage of the x86 processor's support for instruction trapping. First, the application installs itself as the trap handler routine (`int 0x1`). Next, an output file is opened and the TRAP flag is set to `1`. Execution continues into the RIT stream created by the RIT tool, with the previously installed trap handler interrupting execution after each test instruction. The handler appends the CPU state to the file before allowing the next instruction to proceed. The last few instructions disable the instruction trapping, close the file, and exit to DOS. In total, 24,000 instructions generate slightly more than 1 MB of data and execute in less than one minute on the emulated CPU.

7.3 Results

The RIT results from two machines can then be parsed and compared, highlighting any instructions (and their resulting states) which led to inconsistencies. This method relies upon two machines, one being authoritative, while the other is tested. These inconsistencies were generally found to be caused by instruction implementation bugs or instructions which, by design, leave flags in "undefined" states. One method found to mitigate this problem is

to insert instructions with well-defined flag post-conditions, such as `and` or `sub`, either after instructions such as `mult` (several undefined flags), or before instructions which rely upon flags. This is done to "isolate" anomalies, returning the two machines to a known, and hopefully identical state. Although the "false-positive" flag states were distracting, this method led to the identifying and fixing of bugs in the implementations of 9 instructions.

8 Challenges

There were several specific challenges identified. The first of these is incorrect or inconsistent documentation. It is not enough to implement behavior that satisfies the specification. Sometimes, the specification is incorrect, or perhaps over time the de facto standard has deviated. Because of this, *actual*, rather than *specified*, behavior must be replicated. For example, in the Intel Instruction Set Reference the `aaa` instruction operation listing is incomplete, and the `ror` instruction description is incorrect.

The family of `REP` instruction prefixes was also difficult to implement in a way that replicated actual behavior. Its functionality was rewritten several times – adding more compatibility each time. The last rewrite was precipitated by the implementation of hardware interrupts (DOS's DEBUG.COM requires `int 1` and `int 3`). This allowed the repeated instruction to be trapped before *each* iteration, rather than only before the first iteration.

The SDL library requires that all SDL library calls be made from the SDL event thread (other than thread synchronization calls). This requirement affected the application's architecture by forcing threads to route their SDL call "requests" to the SDL event thread. It is even more of an inconvenience when the call must be synchronized, requiring the calling thread to block while the SDL thread performs the required action.

Providing the PIT with accurate *and* efficient counting was not a simple task. The counter value changes faster than 1 Mhz, and applications generally only rely upon it as a short-term, high-precision timing mechanism. It is not efficient (nor necessary) to keep a constantly updated counter value. When software initializes a counter, it is effectively prescribing its period. If the emulator records the moment initialization occurrs, because of the linear nature (decremented 1 Mhz) of the counter value, any future value can be calculated. Also, the counter's period can be used by the PIT thread to generate interrupts, rather than polling. The only limitation is a diminishing

accuracy over time, though this is acceptable based on the typical short-term usage of a counter.

This counter mechanism created another challenge when resuming a suspended emulator instance. If software running on the machine was executing a `delay` (waiting for a certain counter value before continuing), unless the past and present counter values were reconciled, the delay period would be indeterminate. This is solved by calculating the elapsed time between each counter being set and machine suspension. When the machine is resumed, this duration is subtracted from the current time, and the resulting time stamp is used as the new initialization time – essentially closing the time gap between the machine being suspended and resumed.

9 Conclusion

Adequate functionality is emulated to execute moderately robust, yet otherwise incompatible applications in modern operating environments. Emulation's support for features such as machine snapshots, OS debugging, and virtual disks provides an enhanced user experience, and introduces a new paradigm of computing. An average CPU performance of 8-10 million instructions per second was achieved on a laptop computer, more than enough to emulate hardware designed to run DOS. By abstracting the user interface facilities, the SDL library greatly enhances the portability and development speed of this emulated platform, freeing the developer to focus on the problem domain. A design priority, succinct architecture allows for the addition of new components, as well as the modification and reuse of existing ones. Examples include the reuse of the CPU component or the addition of another emulated device. Together, these emulated devices and software create an environment capable of providing a user experience comparable to that of the original target platform.

10 Future Work

There are three main categories of future work: features, optimizations, and code maintenance. The purpose of code maintenance might be to produce a more legible, modular code base, or to port build-scripts to other platforms. Features provide enhanced functionality, while optimizations provide more

efficient functionality. Possible features and optimizations are discussed:

10.1 Features

10.1.1 Audio and Video Modes

Potential features include support for more video modes and support for an emulated sound card. Currently, video mode 3 (text with color, 80 columns, and 25 rows) is supported. The addition of common graphics modes would allow more applications (such as games) to function.

10.1.2 Mouse Integration

Although mouse input is functional, work could be done to make this interaction feel more natural to the user. One way to do this is to make the emulated mouse match the "sensitivity" of the native one by determining the mouse driver's cursor location and comparing it to where the system cursor "should" be. Then it could slightly adjust the information sent to the driver to mediate these differences – creating much smoother mouse movement. An exaggeration of this method might allow the mouse to seamlessly pass from the native environment to the emulated one without the need to "capture" it. This method would not work well with emulated applications that use the mouse without displaying a cursor to the user.

10.1.3 CPU and PIC

No floating-point instructions are currently implemented. There is also no support for 32-bit Protected Mode. This would require support for paging, multitasking, and access protection. The CPU is implemented such that most operand address resolution occurs in one place. This should facilitate the addition of paging and memory access protection.

The primary role of the PIC is to mediate device interrupts to the CPU. This includes masking interrupts such that they are not handled and mapping interrupts to interrupt service routines [8]. In testing, these features have not been encountered, but some guest operating systems may make use of them.

10.2 Optimizations

Amdahl's Law suggests the greatest opportunity for optimization is in two areas: the CPU instruction loop and drawing the screen. The screen is redrawn tens of times each second, and the CPU loop is iterated more than one million times per second. These optimizations could serve either to reduce host environment resource utilization (power, CPU, e.g.) or to allow for greater emulation performance with the same utilization.

With regard to the CPU loop, one common optimization is to delay calculating the value of the flags until they are required[7]. Another CPU optimization would be to optimize the implementation of commonly emulated instructions, perhaps even modifying the emulation path to handle these instructions earlier.

A more complicated optimization is to perform just-in-time (JIT) compilation, translating the original x86 code to the machine language of the underlying hardware. Java's HotSpot and Microsoft's .NET CLR technologies provide this optimization. This JIT translation reduces application portability and must be implemented for each possible host environment. Because implementing a JIT translator for even a single environment is a significant undertaking, it was not within the scope of the project.

11 Miscellaneous Software

While DOS is waiting for user interaction (such as a key press), it repeatedly executes `int 0x28` or `int 0x2A`, depending upon the context. These ISRs can be hooked to allow other resident applications to execute while the CPU is idle. Two programs were written to take advantage of this: `dosidle.com` and `netidle.com` which hook [14] and `HLT` both of these ISRs, allowing the number of instructions executed per second to fall from almost two million to as low as four thousand, without sacrificing responsiveness.

`dumpivt.com` prints a portion of the interrupt vector table (IVT) to the screen. `reboot.com` and `shutdown.com` were written to test the ability to reboot and shutdown the machine from software. `readfont16.com` was written to read the 8x16-pixel font from a third-party BIOS and write it to disk. Every BIOS tested returned the same character data. `kbbios.com` and `kbraw.com` were written to read keyboard input from the bios, and from

[7] Often, flag values are overwritten by subsequent instructions before they are read.

the keyboard controller, respectively. Their output was compared to that of physical machines to test the emulated environment.

12 Prominent Emulators

There are several notable open-source and commercial PC emulators. bochs (`http://bochs.sourceforge.net/`) is an open-source emulator, providing support for several versions of the x86 ISA and several basic devices. bochs also provides a custom BIOS implementation. QEMU [2] (`wiki.qemu.org/Main_Page`) is a similar project with a focus on dynamic code translation (with cross architecture support) and virtualization (assisted by Intel's VT-x[12] or AMD-V) for "near native performances". Borrowing heavily form qemu, VirtualBox (`virtualbox.org/`) is Sun's platform virtualization application. Although it is open-source, it is most similar to commercial products offered by VMWare (`vmware.com/solutions/desktop/`), Microsoft (`microsoft.com/windows/virtual-pc/`), and Parallels (`parallels.com/`). DOSBox (`dosbox.com/`) is an open-source x86 emulator with a custom implementation of DOS. Although they vary in their methods and their target audience, these applications all seek to emulate the environment of the familiar IBM-compatible PC.

References

[1] Amazon.com. Amazon web services: The economics of the aws cloud vs. owned it infrastructure, December 2009. `http://d36cz9buwru1tt.cloudfront.net/The_Economics_of_the_AWS_Cloud_vs_Owned_IT_Infrastructure.pdf`.

[2] Bellard, F. Qemu, a fast and portable dynamic translator. *Translator* (2005), 41–46.

[3] Bentley, B., and Gray, R. Validating the intel pentium 4 processor, intel technology journal q1. Tech. rep., Intel Corporation, 2001. `ftp://download.intel.com/technology/itj/q12001/pdf/art_3.pdf`.

[4] Brey, B. B. *The Intel Microprocessors: Architecture, Programming, and Interfacing*, sixth ed. Prentice Hall, 2003, pp. 833–854.

[5] Brown, R. Ralf brown's interrupt list. http://www.cs.cmu.edu/~ralf/files.html, August 2010.

[6] Chapweske, A. The PS/2 Keyboard Interface. http://www.computer-engineering.org/ps2keyboard/, April 2003.

[7] Duntemann, J. *Assembly Language Step-by-Step: Programming with DOS and Linux*, second ed. Wiley Computer Publishing, 2000, pp. 182–188.

[8] Intel. 8259A PROGRAMMABLE INTERRUPT CONTROLLER (8259A/8259A-2). Tech. Rep. 231468-003, Intel Corporation, December 1988. http://pdos.csail.mit.edu/6.828/2005/readings/hardware/8259A.pdf.

[9] Intel. 82077AA CHMOS SINGLE-CHIP FLOPPY DISK CONTROLLER. Tech. Rep. 290166-007, Intel Corporation, May 1994. http://www.osdever.net/documents/82077AA_FloppyControllerDatasheet.pdf?the_id=41.

[10] Intel. 82C54 CHMOS PROGRAMMABLE INTERVAL TIMER. Tech. Rep. 231244-006, Intel Corporation, October 1994. http://www.sharpmz.org/download/82c54.pdf.

[11] Intel. VGA BIOS OEM Reference Guide. Tech. rep., Intel Corporation, July 1999. http://versalogic.com/support/downloads/pdf/69030bg.pdf.

[12] Intel. Intel 64 and ia-32 architectures software developer's manual: Vol. 1 basic architecture. Tech. Rep. 253665-039US, Intel Corporation, May 2011. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html.

[13] Intel. Intel 64 and ia-32 architectures software developer's manual: Vol. 2 (2a & 2b) instruction set reference. Tech. Rep. 325383-039US, Intel Corporation, May 2011. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-2b-instruction-set-a-z-manual.html.

[14] Irvine, K. R. *Assembly Language for Intel-Based Computers*, fourth ed. Prentice Hall, 2003, pp. 601–608.

[15] Rothenberg, J. *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation*. Council on Library and Information Resources, January 1998. `http://www.clir.org/pubs/reports/rothenberg/contents.html`.

[16] Shanley, T. *Protected Mode Software Architecture*. Mindshare, Inc., 1996, ch. 12.

[17] Szor, P., and Ferrie, P. Hunting for metamorphic. Tech. rep., Symantec Corporation, 2003. `http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf`.