Design and Testing of a CPU Emulator

Weiqin Ma, Jyh-Charn (Steve) Liu
Texas A&M University

Alessandro Forin
Microsoft Research

August 2009

Technical Report
MSR-TR-2009-155

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Design and Testing of a CPU Emulator

Weiqin Ma
Department of Computer Science
Texas A&M University
College Station, TX 77843
maweiqin@tamu.edu

Jyh-Charn (Steve) Liu
Department of Computer Science
Texas A&M University
College Station, TX 77843
jcliu@neo.tamu.edu

Alessandro Forin
Microsoft Research
One Microsoft Way
Redmond, WA 98052
sandrof@microsoft.com

Abstract

In this paper, we describe the design and testing process of an x86 CPU model for the Giano simulator. We used a hardware-oracle for test generation and as a reference behavioral specification. We developed an automatic tester that extracts the CPU specifications from the published CPU datasheets, and automatically generates test cases with complete test coverage. The number of required tests is reduced by many orders of magnitude by the separate testing of operand routing from computation. Debugging efficacy is such that the tester is now used concurrently with the development process, not afterwards. The hardware-oracle detects and documents several undocumented or erroneously-specified CPU behaviors which would be difficult to detect by conventional testing methods.

1 INTRODUCTION

The correct implementation of CPU emulators is critical for computer system design, and yet software errors in CPU simulators can remain undetected for years after the software is released. For example, implementation errors for the ADC and SBB instructions in the commercial 8086 emulator [18] were not found until version 4.08. For variable length instruction sets, such as the x86 instructions, the cascading effects of an error can lead to a total crash of the emulator. Bugs in Valgrind [17] could lead to errors in all subsequent instruction executions. Despite its importance, it is extremely costly and technically challenging to implement, debug and test CPU emulators as three separate steps, purely based on software methods. In addition to programming errors, the incomplete and often incorrect documentation of CPU instructions is also an important issue. Specifications are written with the programmer and compiler writer as targets, not for CPU implementation or simulation. Consequently, they do not cover some important details, and leave many behaviors as "undefined" or "implementation specific".

The laborious search and matching process of data sheets for testing and validation is error prone, and slow. When the hardware for the CPU is available, it is much more reliable and accurate to use the actual outputs of the hardware as the ground truth to test and verify the emulator. Using actual hardware as a reference is what we call a *hardware-oracle*.

With the broad applications of CPU emulators for the design of virtual execution environments, a modular emulation architecture is highly desirable, so that instructions can be implemented and verified incrementally and comprehensively. We adopted the system emulator Giano [1] as the integration target for this work. Giano has been used in several product developments, and its source can be downloaded from the Microsoft web site [23] Positioned as a multi-platform system simulation framework, Giano already supports simulation of several instruction sets such as ARM, PowerPC, VAX, and MIPS. It provides real-time support, extensive I/O modules, and an interface to the Xilinx ModelSim hardware simulator. With this work we have now added support for the popular x86 instruction set.

In this paper, we propose a *hardware-oracle* based implementation methodology for CPU emulators. We use a real processor to run each instruction being simulated and check its outputs against that of the simulator. When the hardware-oracle runs on a bare-bone machine, it can support testing of all instructions and operations. When the hardware oracle runs within an application program with user level privileges, some of the exception and interrupt related functions will be restricted.

The hardware-oracle eliminates any guessing of the true behavior of a CPU, and it had already been found highly effective for the rapid and correct implementation of the ARM and MIPS modules in Giano. That earlier work was manual, undocumented, and *ad hoc*. In this paper we

present a complete, automatic testing strategy and tool that leverages the published specifications, in English, from the Intel PDF files.

We design and test our concepts based on the 32-bit x86 architecture. The printed x86 instruction specifications are only used as the starting point for implementation -- they are not needed for subsequent testing when the hardware-oracle can deliver the required data. Our experiments show that the new design methodology drastically improves the productivity in development, debugging, testing, and validation of the simulator. It also helped identifying several CPU behaviors which are not documented in the published CPU manuals.

The rest of this paper is organized as follows. Section 2 describes the overall structure of the emulator. Section 3 described the automated testing framework we developed for testing the new emulator. Section 4 presents a technique for reducing considerably the number of tests needed for full coverage. Section 5 presents the results of our evaluation. Section 6 presents the related work and Section 7 concludes.

2 Emulator Design

A conceptual illustration of an interpretive CPU emulator is shown in Figure 1. Each instruction is handled in its own separate function. The execution flow of our new x86 emulator is also of the type shown in Figure 1. In the execution loop, it fetches an instruction from memory, then decodes and executes the instruction. Before fetching the next instruction, it first checks the trap queue for any outstanding trap or interrupt requests. We use a lookup table for opcode decoding so that the execution can directly jump to the simulation function. We also implemented an instruction cache and its prefetch controller to improve the performance of simulation.

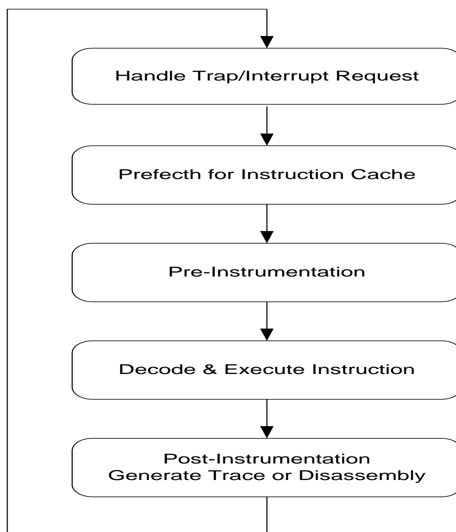

Figure 1: The execution flow of the emulator.

The emulator is realized as a dynamic loadable library (DLL) so that it can be readily embedded into a full system emulator, or a virtual execution environment. We used a parser to automatically extract all instruction types, formats, normal behaviors and their exceptions from the Intel manuals. The instruction specifications are also formalized for subsequent implementation and test case generation.
Real-time support is important for system level simulation. Giano supports real-time execution on a general-purpose host OS. In this emulator, we provide a real-speed calibration module to adjust the running speed of the CPU emulator, in order to match the speeds of CPU, Memory and IO device modules with the expected clock speed.

2.1 Common Macro Templates

Generally speaking, an instruction set architecture consists of a few different types of operations, e.g., arithmetic, Boolean logic, jumps, etc. Within each category different instructions have very similar ways of handling their operands, yet cannot be easily or efficiently implemented as common functions. Using *common macro templates* takes advantage of this structural relationship between instructions and reduces the costs in software implementation, testing and validation.
A macro template mimics, in a way, how the actual hardware is implemented by using common and replicated circuits as much as possible. In addition to the better code reuse in the implementation, macro templates also help with debugging and testing. Using the ground truths provided by the hardware-oracle, we can eliminate software bugs rapidly and efficiently, because finding a bug in a template will fix a number of bugs in the related instructions. Instructions which do not have specific structural patterns need to be implemented and tested individually. This technique is particularly useful in the implementation of complex address and operand classes in x86 instructions.
We have created several macro templates. A first mining rule is to search for instructions that have different types of operands but the same computation logic. For example, the computation logic in the ADD instruction can be represented as DEST $\leftarrow$ DEST + SRC. Many other instructions have this same general structure. Different computations can reuse the same simulated logics for fetching operands and storing the result, only the operation itself needs to be replaced. A second mining rule is searching for operand types, and different instructions which have same type of operands. For example, using the Intel manual's symbolism [9,10,11] the instructions ADD Eb,Gb; ADC Eb,Gb; AND Eb,Gb; and XOR Eb,Gb have different operations but the same operands. A third mining rule is for instructions which similar operands, e.g., INC eAX; INC eCX; INC eDX; INC eBX. Many other rules can

be created to streamline the emulator design. We will discuss some of these rules in the rest of this section.

The operands and addresses of x86 instructions can be 8bit (byte), 16 bit (word), 32 bit (double), or 64 bit wide. They are specified in the *Operand Size*, *Address Size*, and *prefix* fields. The first macro template is the *vertical pattern*, which refers to the set of instructions who have the same type of input and output operands, and only differ in the 'core' operation. Instructions in this category include ADD, ADC, AND, OR, SUB, XOR, SBB, and CMP. Their operands are of identical types, e.g., ADD Eb,Gb, and their opcodes are 0x00, 0x10, 0x20, 0x30, 0x08, 0x18, 0x38. Here, Eb means that the operand can be a memory or register byte, and Gb means a general register byte.

Within the vertical pattern, there are four different sub-patterns to represent six types of operands for the instructions ADD, ADC, AND, OR, SUB, XOR, SBB, and CMP. These six types of operands include (Eb,Gb), (Gb,Eb), (Ev,Gv) (Gv,Ev), (AL,Ib), and (rAX,Iz). The symbols, Ev (Gv), represents the addressing method and the operand type. For example, E represent that the operand is a general register or a memory address specified by a following ModR/M byte. As such, the ADD instruction can be one of the six different variations: ADD Eb,Gb, ADD Gb,Eb, ADD Ev,Gv, ADD Gv,Ev, ADD AL,Ib, or ADD rAX,Iz. By making each of these four operand types into a macro, we can use just six operand templates under the vertical pattern to implement a total of 6x8=48 instructions. Furthermore, there are a total of four combinations for the six types of operands. By implementing and testing these types in groups, we save a significant amount of effort.

A second macro template is called the *horizontal pattern* for opcodes whose numerical values increase linearly, and where the opcode value also implies the order of registers in the operand fields. For instance, the opcodes and registers of the instructions INC and PUSH are listed in Table 1.

Using this ordering relationship to organize registers in an array, we can directly locate the register operand for each opcode. This pattern is used for implementation of the instructions PUSH, XCHG, MOV, DEC, and POP.

Table 1: Opcodes of the INC and PUSH instructions.

0x40 INC eAX	0x41 INC eCX	0x42 INC eDX	0x43 INC eBX	0x44 INC eSP	0x45 INC eBP	0x46 INC eSI	0x47 INC eDI

0x50 PUSH rAX	0x51 PUSH rCX	0x52 PUSH rDX	0x53 PUSH rBX	0x54 PUSH rSP	0x55 PUSH rBP	0x56 PUSH rSI	0x57 PUSH rDI

Some other one-byte opcode instructions that have been implemented using macro templates are listed in Table 2. The first column of the table is the template name, the second column is the instruction element that can be implemented using the macro template, and the third column is the total instruction counts for the macro template. As of the writing of this report, 84.8% (195/230) of the realized instructions have been implemented using macro templates. Other instructions not covered by the above discussions are implemented individually, although some of them still have un-exploited common patterns. For instance, the TEST instruction with immediate operands: TEST AL,Ib and TEST rAX,Iz. These and others will be consolidated in the follow up phase of the project.

Table 2 Instructions that use code templates

Template ID	Instruction elements	Instructions
1	ADD Eb,Gb	8
2	ADD Ev,Gv	8
3	ADD Gb,Eb	8
4	ADD Gv,Ev	8
5	ADD AL,Ib	8
6	ADD rAX,Iz	8
7	INC eAX	8
8	DEC eAX	8
9	PUSH rAX	8
10	POP rAX	8
11	JO Jb	16
12	XCHG rCX,rAX	7
13	MOV AL,Ib	8
14	MOV rAX,Iv	8
15	Group#1 Eb,Ib	8
16	Groou#1 Ev,Iz	8
17	Groou#1 Eb,Ib	8
18	Gropu#1 Ev,Ib	8

2.2 Minimal CRT for Stand-alone Execution

When the emulator is used for architectural experimentation it is desirable to run simple programs with a minimal overhead. Simulators such as SimpleScalar [24] directly interpret the binary program as if it was a Unix program, and by realizing the Unix syscall interface they can provide services such as printf() and file I/O. To run the application, the simulator loads the code in simulated memory, allocates the memory space of a stack, and invokes the entry point specified in the executable image. If the image uses shared libraries, it is again the simulator that

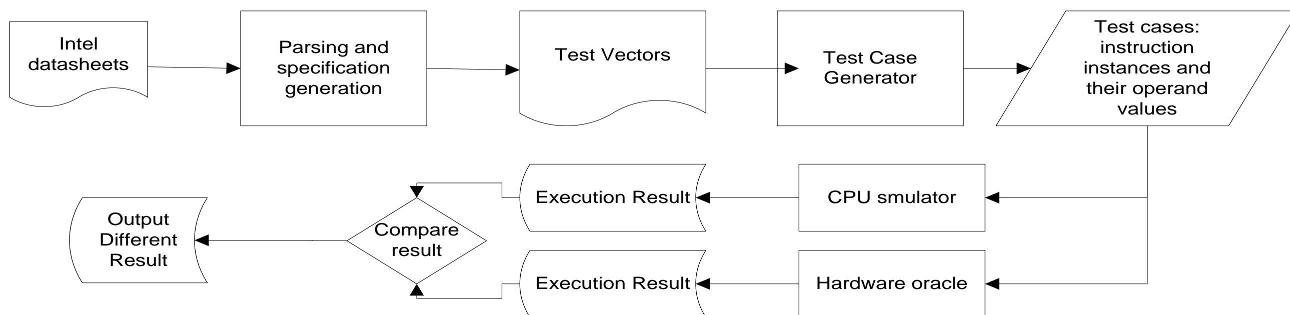

Figure 2: The conceptual flow when testing with a hardware-oracle.

loads them. In a hardware, full-system simulator such as Giano all we have are CPU, memory, busses, and I/O devices. How to directly run very simple applications on the CPU emulator is therefore a challenge. To achieve this goal, we designed a Minimal C Run Time Library (CRT) for the emulator to run a C program without providing any complicated system environment.

The minimal CRT consists of two simple functions. The start() function initializes the stack pointer using a user-provided memory array, invokes the program's main() function, and when this returns it terminates the program by executing a HLT instruction. Note that on a full-system implementation the HLT instruction does not terminate the CPU but transfers the execution control back to the emulator. The putchar() function actually is an empty function, but by using Giano's built-in tracing functionality every time execution reaches putchar() we can look at the EAX register and print it as a character on the screen. We make use of code injection techniques for building the minimal programs as memory dump images. The system is therefore unaware of executable file formats. A simple tool injects the executable binary with jump code at the beginning position to transfer control to the actual entry point of the program. The minimal CRT consists of 14 lines of C code that compile to 19 bytes of code.

In a typical implement/debug/test setup, the minimal CRT would be used in the debugging and testing phases. Simple programs are generated and run by the test harness, and the outputs are verified. As we shall illustrate in the following sections, both debugging and testing can be made much more efficient using a hardware- oracle based approach.

2.3 Real-time support

Real-time support is important for real-time system simulation and research. It is also useful in the general case, as it provides a more faithful execution and interaction with external devices and other systems. In this emulator we provide a real-time module, which can calibrate the frequency of the CPU emulator. We found that the following simple scheme is efficient and sufficiently accurate:

- Every *M* thousands of clock ticks spin idle for *D* microseconds;
- Every *N* millions of clock ticks check the actual frequency against the target frequency and adjust the delay *D*.

In this way, the effective delay on each clock period is *D/M*, which can be as small as a few picoseconds and therefore would be impossible to realize on a per-period basis. The inevitable control overhead is also spread over a larger number of cycles. Checking the actual frequency involves reading the current time, which on many Operating Systems is an expensive operation compared to incrementing a counter and performing an integer division. As can be seen in Figure 3, the adjustment of the delay value *D* is based on a first-order low-pass filter, which seems sufficient to provide quick convergence without excessive fluctuations. Again, the cost of this computation is amortized over *N* million cycles. This feedback loop not only matches the target simulation clock speed to the host computer's clock speed, but it also takes care, at least in part, of the fluctuations in execution speed due to multiprogramming on a general-purpose and non-Real-Time commodity Operating System. The result is that the CPU module executes at the target clock speed while the user can continue with other activities.

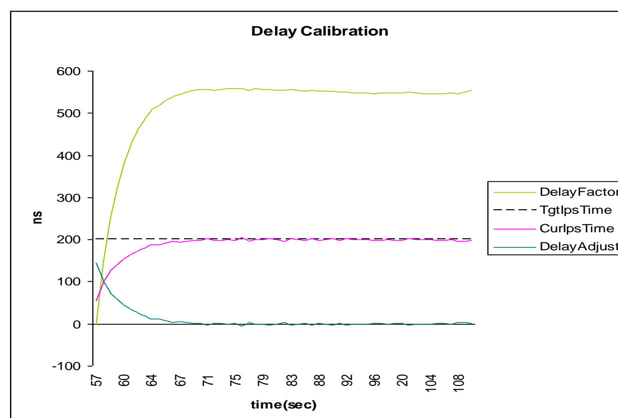

Figure 3: Delay adaptation to match the actual clock speed of the simulator to the target clock speed.

3 Automatic Test Generation

Two key issues in the automated test framework (*test harness*) are the automated *test vector* generation, and the automated *test case* generation based on the test vectors. A test vector of an instruction defines the different fields of the instruction, their ranges, lengths and other details, so that they can be used by the test case generator to generate a series of test cases. Each test case represents an instruction with one distinct combination of argument values. Each test case consists of a set of *test runs* to assess the correctness of the CPU emulator by operating the instruction on selected operand values. Because there is no simple expression which can describe the relationship between an opcode and its operands, the only available information is the published manuals of the CPU, and the physical device itself. As a result, we developed an automated test vector generator based on the parsing of the PDF files for the Intel manuals. The test vectors are used by a table-driven, automated test case generator to generate all testing cases for both the emulator and the hardware-oracle. Each automatically generated pair {instruction, input processor state} is run by the hardware-oracle, which generates the output processor state. The triple {instruction, input processor state, output processor state} is sent as a test case to the CPU emulator for execution, to test the correctness of the CPU emulator by comparing their outputs. Note that in addition to a pass/fail result, the emulator will provide the developer with its own output processor state. The quad {instruction, input processor state, output processor state, error processor state} provides the necessary information for efficiently pinpointing the error. In case of a poorly or erroneously documented instruction, the set of quads completely tabulate the required behavior.

3.1 Test Vector Generation

The Intel manuals define the instruction formats using such symbol as *sign, word, mod, reg, r/m, sreg,* etc., see Figure 4. Both the test vector and test cases of an instruction are generated in a deterministic fashion, following these formats. We designed a parser which takes operand tables in the *instruction format and encodings* part of the x86 manual [10,11,12] as its input and generates test vectors for most instructions. For a small number of instructions, e.g., BOUND, CPUID etc, whose formats were not fully documented in the manual, it is necessary to manually locate their technical descriptions from other manual volumes to create their test vectors case by case. All test vectors are collected in a computer-readable file, which can then be edited as necessary to cope with errors and omissions in the specifications.

Prefix	Opcode	ModR/M	SIB	Displacement	Immediate

Figure 4: Basic structure for an x86 instruction [11]

A test vector for an instruction consists of several records. Consider the instruction ADC as an example. The Intel manual[11] reports seven possible encodings, one of them is as follows:

Register to memory 0001 000w: mod reg r/m

The parser generates the test vector for this specific encoding as follows:

name ADC *// instruction name (and optional reference index)*
inst 2 16 0 *// 2 bytes, opcode 0x16, 2nd byte initialized as 0x00*
w 0 0 *// both byte and word forms, per byte 0, bit 0*
reg 1 3 *// the reg-ister is encoded in first byte, from 3rd bit*
md 1 6 *// mod encoded starting at 6th bit of the first byte*
rm 1 0 *// register/memory at first byte, starting 0th bit*
end *// end of instruction*

Each test vector represents the general format of an instruction, its initial values, and also fields whose values need to be enumerated for the complete testing of the instruction. Each unique combination of these field values represents one distinct mode of the instruction, or a test case. For example, we could generate an instruction "0x11 0x90" by replacing the "w" bit with 1, "mod" bits with b"10", "R/M" with "000", and "reg" with b"010". We will then have to provide specific values for the indicated registers or memory location. An overview of the automated design and testing system for CPU emulators is shown in Figure 2.

3.2 Test Case Generation

Test cases are *executable* instructions, instantiated from the abstract instruction specifications defined in the test vector. For each test case we also need to generate different operand values to evaluate the functional correctness of the CPU emulator under different numerical values. It is relatively easier to handle register based operands, but it is more complicated to deal with memory based operands. One major challenge is the correct handling of all the addressing modes defined in the MODRM byte, which may be further expanded to include two additional bytes: the SIB byte (0,1 byte length) and Displacement byte (0,1,2, byte length according to the operand size). Some instructions may also include immediate operand values of 1, 2, or 4-byte long. Furthermore, the addressing modes are somewhat irregular, and include a number of special cases. Consequently, even the simple task of computing the test instruction length is not as trivial as, say, with a RISC architecture.

The memory address of an operand is calculated from the 1 to 3 byte values of the MODRM-SIB-displacement fields. Because the test must be actually executed in the hardware-oracle environment, it is impossible to exercise all possible combinations of the three fields. For instance, we cannot modify the memory ranges for the code of the hardware-oracle itself. As a compromise, we adopted the following technique to test selected address ranges as follows. We first select a memory address MA as the expected memory location for an operand. Then test if the operation "[Base

Register] + [Index Register] *[Scale*Register] + displacement" (performed in the CPU emulator) does give the value MA for a set of MODRM-SIB-displacement values which are known to produce MA on the hardware-oracle. The test is repeated for a few times to run through different combinations of the three fields, but with the same MA. As an illustration, in the first type of tests we can set one of the three fields as zero, and let the two remain fields added to the value of MA. In the second case, two of them are zero, and one field is made equal to MA. Finally, none of the three fields is zero, and their sum is made equal to MA. In reality, there are cases where it is demonstrably impossible to produce a given MA with a required set of registers. Consider the case, for instance, where [Base Register] and [Index Register] are the same. To cover these cases we allow for backtracking, e.g. small adjustments around the desired MA value. The problem is exacerbated by implementation restrictions, such as maintaining a valid stack pointer value in test cases that can generate a trap.

To compare memory related operations, we must use the same addresses in both hardware-oracle and the CPU emulator. We employ some simple strategies to generate the same logical memory addresses for both. This is relatively simple in the Windows XP environment, because the malloc() starting addresses are fixed for all instantiations of the hardware-oracle process. It is slightly harder to do the same for Windows Vista, because of its address randomization scheme, where the range of randomization is the order of a few MB. As a result, we malloc() sufficiently large address spaces for both CPU emulator and the hardware-oracle, and then identify sufficiently large, overlapped address subspace for both.

Brute force testing of a CPU emulator is an impractical idea. For a single 32-bit operand one would need to enumerate 4 billion distinct values to scan through all possible values. For two operands we need to test all 2^{64} combinations. An x86 instruction can have up to 3 operands (2^{96}), each of which could additionally be a register or memory. Special flags often affect the results of an operation, but we also need to test that they do not. Full enumeration of the operand values is clearly impractical for real world applications. As an effective yet practical testing strategy, we propose to use boundary cases to generate operand values for their testing.

For all practical purposes of emulator testing, it is unnecessary to enumerate all values. We use the following two techniques to maximize the test coverage under the constraint of testing time. Experiences suggest that errors are always revealed by the boundary cases, and very rarely by the "common" values. Random number generators are not guaranteed to generate all boundary values, and they do not support scanning of the entire set anyways. Instead, we use a 32-bit generator for boundary cases that only uses about a 100 distinct values, thereby reducing considerably the number of tests generated. Similar considerations apply to the other machine data types, pointers, and special fields.

The second consideration is that fetching of operands is independent of computation on those operands. For instance, the carry-chain adder circuit for the ADD instruction (or its software counterpart) will operate identically whether the operands come from the EAX, EDX register pair or from any other pair of registers. Therefore in testing we can separately test that

1- the ADD instruction correctly computes its result, and
2- the ADD instruction correctly fetches its source operands and deposits the result to the correct destination.

The two aforementioned techniques lead to a drastic reduction in the number of tests. Consider the ADD instruction, such as "ADD r32A,r32B". The source operand may be chosen from one of the 8 general purpose, 32-bit registers. The destination is r32A and we ignore the flags and anything else. A complete test generator would need to generate $2^{32} \times 2^{32} \times 8 \times 8 = 2^{70}$ distinct patterns, without considering any operands in memory. Using the first reduction technique alone, we need roughly $2^7 \times 2^7 \times 8 \times 8 = 2^{20}$ test patterns. Using the second technique alone, we need $(2^{32} \times 2^{32}) + (8 \times 8) \sim= 2^{64}$ patterns. When the two techniques are combined together, we get $(2^7 \times 2^7) + (8 \times 8) \sim= 2^{14}$ patterns. The second technique is especially valuable in a complex instruction set such as the x86, with so many addressing modes, 725 different modes in all.

For the symbols with short bit counts, such as the one-bit "w" field, we generate all the possible values. For the 8 bit and 16 bit and 32 bit integer values, we use an algorithm to generate boundary values for operands, which are represented by symbols in the test vector, e.g., w, reg, mod, etc, and their 8-, 16-, and 32-bit integer values. A total of three different boundary value patterns are generated for each N-bit operand. In the first type of patterns, we interplay the 0x0 and 0xf every four bits for the operand. In the second type of patterns, we shift a single bit of "1" from the lowest significant bit (LSB) to the most significant bit (MSB). In the third type of boundary value patterns, we shift a single bit "0" from the LSB to the MSB.

It is relatively straightforward to run an instruction on the CPU emulator, because the emulator has full control of the execution flows and state changes. On the other hand, it is necessary for the hardware-oracle to save the processor state values before the instruction being tested can be executed, and other additional routines for safe execution of the hardware-oracle in the user mode. More details on those house-keeping steps will be discussed shortly.

3.3 Test Run Control

Each generated test case is represented by a data structure shown in simplified form in Figure 4. It includes the instruction being tested, and input and output states for registers and memories. The test execution engine maintains a JTAG style pipe channel for communication between the CPU emulator and the hardware-oracle. The

test data is sent over the pipe, so that the emulator can safely run test cases and compare their outcomes.

```
typedef struct {
    GPREG i386_gpr[NREGS];
    FPREG i386_fpr[NREGS];
    UINT16 i386_fcr;   // FPU control register
    UINT16 i386_fcs;   // FPU status register
    UINT16 i386_ftag;   // FPU tag register
    UINT64 i386_fip;   // FPU instruction pointer (errors)
    UINT64 i386_dip;   // FPU data pointer (errors)
    SREG i386_seg[6];  // segment registers
    SDESC i386_desc[6];  // cached segment descriptors
    SDESC i386_desc2[4]; // more segment descriptors
    UINT32 i386_flags;  // EFLAGS
    UINT32 i386_spr[16];  // special purpose registers
    UINT32 i386_cia;   // current instruction address
} REGSTATE, *PREGSTATE;

typedef struct {
    UINT32 base_address;
    UINT32 count;
    UINT32 values[NMEM];
} MEMSTATE, *PMEMSTATE;

typedef struct {
    UINT32 test_code;
    UINT8 instruction[IBYTES];
    REGSTATE input;
    REGSTATE results;
    MEMSTATE meminput[2];
    MEMSTATE memresults[2];
} TEST_ITEM, *PTEST_ITEM;
```

Figure 5: The data structure of a test case

Every test case generated by the test generator must be run by the hardware-oracle and by the CPU emulator. When the hardware- oracle works in user mode under the control of the operating system, test cases are run by the *test execution engine*, which maintains an execution environment similar for both hardware- oracle and the CPU emulator. The core code for the execution engine is shown in Figure 5, it includes CPU state backup, instruction execution, execution result preservation, CPU state restore. The code is self-modified at run time to realize the individual instruction, and to cope with execution-dependent addresses for global variables.

CPU state backup uses PUSHAD and PUSFD for preserving general and flag registers. Also, the stack pointer (ESP) is saved into a global variable. Similar to this aforementioned state preservation step, multiple stacks are created by using global variables and changes of the stack point register will preserve different system states whenever necessary. A block of memory space at the array *testcode[],* called *NOP block,* reserved for running the instruction under test is otherwise filled with NOP instructions by the test program. Next, the test program loads registers and (affected) memories of the CPU emulator and the hardware-oracle based on the test case. Then, the instruction under test is placed at the beginning of the NOP block. The execution EIP then transfers to the address of *testcode*.

```
#define PROLOGUE 28
#define EPILOGUE 29
UINT8 testcode[PROLOGUE+IBYTES+EPILOGUE] = {
 0x9c,       /* pushfd */
 0x60,       /* pushad */
#define SAVED_ESP_OFF1 4
 0x89, 0x25, 1,2,3,4,   /* mov dword ptr ds:[_saved_esp
04030201],esp */
#define EFLAGS_OFF1   (SAVED_ESP_OFF1+4 +2)
 0xFF, 0x35, 1,2,3,4,   /* push  dword ptr [_eflags] */
 0x9D,       /* popfd */
#define TEST_ESP_OFF1  (EFLAGS_OFF1+4 +1 +2)
 0x8B, 0x25, 1,2,3,4,   /* mov  esp,dword ptr [_test_esp] */
 0x61,       /* popad */
#define TEST_ESP_VALUE_OFF1 (TEST_ESP_OFF1+4
+1 +2)
 0x8B, 0x25, 1,2,3,4,   /* mov  esp,dword ptr
[_test_esp_value] */
 /* PROLOGUE ENDS HERE */
#define TEST_INST_OFFSET
(TEST_ESP_VALUE_OFF1+4)
 0x90, 0x90, 0x90, 0x90,  /* IBYTES(5*4=20) NOPS */
 0x90, 0x90, 0x90, 0x90,
 0x90, 0x90, 0x90, 0x90,
 0x90, 0x90, 0x90, 0x90,
 0x90, 0x90, 0x90, 0x90,
 /* EPILOGUE STARTS HERE */
#define EPILOGUE_START  (PROLOGUE+IBYTES)
#define TEST_ESP_OFF2  (EPILOGUE_START+2)
 0x89, 0x25, 1,2,3,4,   /* mov  dword ptr [_test_esp],esp */
#define RESULT_ESP_OFF1  (TEST_ESP_OFF2+4 +2)
 0x8B, 0x25, 1,2,3,4,   /* mov  esp,dword ptr [_result_esp]
*/
 0x60,       /* pushad */
#define EFLAGS_PTR_OFF1  (RESULT_ESP_OFF1+4
+1 +2)
 0x8B, 0x25, 1,2,3,4,   /* mov  esp,dword ptr [_eflags_ptr]
*/
 0x9C,       /* pushfd */
#define SAVED_ESP_OFF2  (EFLAGS_PTR_OFF1+4 +1
+2)
 0x8B,0x25, 1,2,3,4,   /* mov esp,dword ptr ds:[_saved_esp
04030201] */
 0x61,       /* popad */
 0x9d,       /* popfd */
 0xc3       /* ret */
};
```

Figure 5: The core function in the hardware-oracle execution engine.

After the instruction is executed, the CPU will run a variable number of NOP instructions. Next, we need to preserve the resulting stack pointer, general registers, and eflag register into some global variables. Finally, we need to restore the CPU states and returns the execution control back to the test execution engine. The remainder of the engine code is written in C, and is responsible for setting up the desired test memory state before *testcode* runs and for recovering the modified memory state afterwards.

The CPU emulator and the hardware-oracle communicate with each other through a JTAG style pipe. The pipe transfers a structure of a test case which includes the input values for one instruction and the corresponding execution results from the hardware-oracle to the CPU emulator for comparison, after it executes the same instruction with the same set of inputs. Although the CPU hardware may contain some design or implementations which do not conform to the published specifications, we still take the hardware-oracle as the ground truth, because the CPU emulator is meant to correctly reflect all behaviors of the actual hardware device. The basic architecture for complete testing of an instruction is shown in Figure 6.

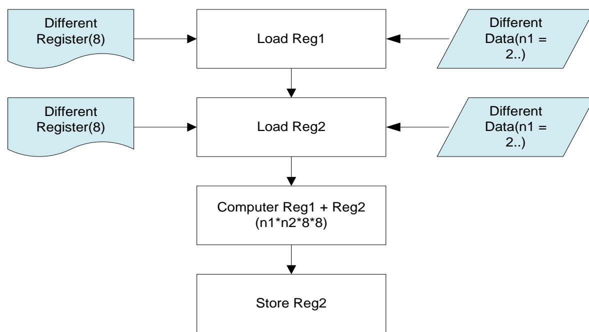

Figure 6: Brute force testing of an instruction

4 Routing-based Testing

Brute force, exhaustive testing of an instruction would enumerate all possible encoding options and operand values. For example, for a 32-bit register, we need to test 4,294,967,296 distinct values for a single register and 18,446,744,073,709,551,616 values for two registers. There are very few cases where a complete enumeration is required, for all other cases we propose the following *routing based test* scheme: We split the instruction into two stages -- input/output stage and instruction computation stage. We presume the two stages independent and test them separately.

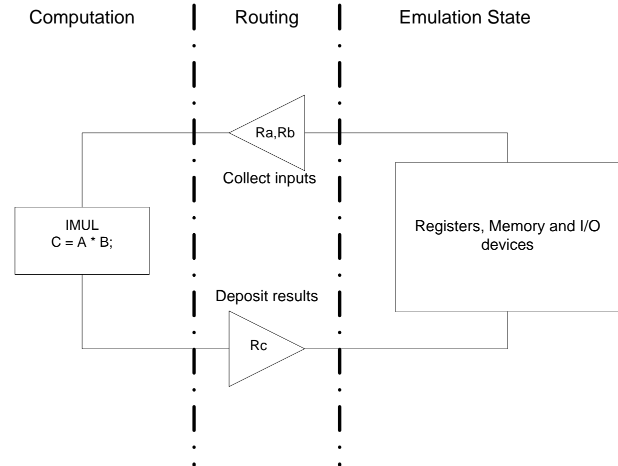

Figure 7: The routing test model.

With reference to Figure 7, the Routing stage is the I/O stage that fetches the operands and deposits the results into the Emulation State. The Computation stage performs the instruction-specific computation. We can think of data being circulated out of the Emulation State and back into it by a software “routing” network. Testing simply tries to find any errors in this routing network.

Using a routing based test scheme greatly reduced the number of instructions we need to test, and by many orders of magnitude. The routing test is based on the observation that each instruction implementation (e.g. for code coverage) can be split in three parts:

(1) collect inputs, either from a register R(*r*) of a memory location M(*m*)

(2) compute the results

(3) send results to the destination, either R(s) or M(n)

The Routing stage is parts 1 and 3, Computation is 2, and registers R() and memory M() live in Emulation State (along with the I/O peripherals). All "Routing errors" will happen in 1,3, for instance in selecting

(a) a register instead of memory, or vice versa

(b) R(x) instead of R(y),

(c) M(x) instead of M(y),

(d) the wrong size for R(r) or M(m),

(e) the wrong arithmetic sign,

(f) anything else that would get a value different from the intended one.

Intuitively, (a) represents a misunderstanding of the specifications which leads to using a register instead of memory (say *mod*==3 vs. *mod*==0,1,2); (b) is the case of a typo that picks the wrong register number (like swapping *r/m* for *reg*); (c) is the case where, say, fetching the wrong

immediate value produces the wrong effective address (d) is a mis-declaration of a "UINT16 v0" instead of "UINT32 v0" that somehow escapes the compiler; (e) is a typo INT versus UINT, or the following cut-and-paste error that lead to the wrong sign-extension:

```
UINT32 v = REG_EAX;
if (v & 0x8000)
        REG_EDX = 0xffffffff; else REG_EDX = 0;
```

(f) is a catch-all category for any other coding mistake in 1 and 3.

The Computing part of an instruction is anything that does not directly affect the Emulation State.

The pseudo-code in Figure 8 shows the brute force testing strategy, exemplified here for testing the IMUL instruction. In the algorithm, first we construct the instruction and possible encoding formats. This is illustrated in the line 1,2,3 of Figure 8. Furthermore, we construct different values of all the operands (lines 6 and 7).

```
1 forall SRC1 do
2   forall SRC2 do
3     forall DEST do
4       forall V32_A do
5         forall V32_B do
6           SRC1=V32_A
7           SRC2=V32_B
8           test IMUL DEST,SRC1,SRC2
```

Figure 8: Brute force testing strategy.

In the routing strategy, we split the testing of encoding for the instruction and the computation for the instruction. As shown in figure 9, we use fixed values for the operands when we test the encoding of the instruction. In this case "3"is a fixed number for both operands.

```
1 forall SRC1 do
2   forall SRC2 do
3     forall DEST do
4       SRC1=3
5       SRC2=3
6       test IMUL DEST,SRC1,SRC2
```

Figure 9: Encoding tests in the routing strategy.

To test the computation stage of an instruction we use the pseudo-code shown in Figure 10. Here we use different values but a fixed encoding -- "eax,ebx,ec" are the fixed registers for all tests.

```
1 forall V32_A do
2   forall V32_B do
3     eax=V32_A
4     ebx=V32_B
5     test IMUL ecx,eax,ebx
```

Figure 10: Computation tests in the routing strategy.

The reduction in test case numbers stems from the elimination of the inner loop in Figure 8. Now we run the code in Figure 9 *followed* by the code in Figure 10. Rather than generating a number of test cases equal to the product MxN, we only generate the sum N+M.

5 EVALUATION AND DISCUSSION

We have applied the common code template technique in the implementation of our x86 emulator, and applied the hardware-oracle based testing strategy during debugging and testing. In this section, we evaluate our technique by showing some results and report some key observations.

Currently 16-bit mode is only tested using prefix overrides and privileged instructions are not tested. Our algorithm supports 8bit, 16bit and 32 bit instructions, and currently does not support 64bit instructions. However, our algorithm could be easily extended to 64bit. There are more than 1,000 instructions in the x86 instruction set. We only implemented the Intel 486 instructions. Currently the MMX, floating point and 64 bit instruction are not implemented.

Table 3. Some simple examples of discrepancies between published specs and the actual CPU behavior

Instruction	Published Spec	Actual behaviors
AAA	OF,SF,ZF and PF flags are undefined	OF,SF,ZF,PF are affected
AAD	OF,AF and CF are undefined	OF,AF and CF are affected
AAM	OF,AF and CF are undefined	OF,AF and CF are affected
AAS	OF,SF,ZF and PF are undefined	OF,SF,ZF,PF are affected
DAA	OF is undefined	OF is affected
AAS	OF is undefined	OF is affected
AND	AF is undefined	AF is affected
TEST	AF is undefined	AF is affected
OR	AF is undefined	AF is affected
XOR	AF is undefined	AF is affected

It is impossible to quantify precisely the effectiveness of the testing strategy without using a prohibitive, double-blind development process. Using the automated tester we have found at least one, and quite often more than one error in the implementation of every single one of our emulated instructions. Indeed, the debugging efficacy is so high that the automated tester is now used concurrently with the development process, not as a separate phase. We quickly abandoned our initial testing strategy of booting a BIOS or a 32-bit OS because too ineffective. Indeed, Windows CE was running on the simulator when we were only about 40% code complete, and with many errors that were found later.

5.1 When the Specs Are Wrong

Through the course of the design and testing of the CPU emulator we observed gaps both in design and testing. By design gap we mean that there is a difference between what the Intel materials say the processor should do, and what the hardware-oracle actually does. By test gap we mean that there are some incomplete specifications that do affect the processor behavior and are visible to software.

The test generation tool combined with the hardware-oracle helped us identify many discrepancies between the CPU manuals and the physical devices. They would be otherwise very difficult to detect by traditional techniques. For instance, when the document says a flag register is undefined, it is natural to assume that the flag register is not affected, but the assumption does not always hold. In several experiments the adjust flag (AF) was actually set in the CPU while it is stated as "undefined" in the manual for several instructions. Table 3 lists the first few instruction, in alphabetical order, where we found discrepancies. Notice that any known and detectable discrepancy between emulator and actual CPUs can be used to generate "red pill" tests by malware to avoid detection. We tested the instructions reported in Table 3 on the Bochs and QEMU emulators. Bochs emulates only some of the undocumented instructions correctly, and QEMU does not emulate any of them.

We infer the correct behavior based on the execution results of the hardware-oracle. Oftentimes an "undefined" flag is simply set or cleared. However, sometime it is very difficult to guess the correct behavior of an instruction. For example, the computation of the parity flag in the AAA and AAS instructions is quite mysterious. Usually, the "parity" is computed on the low 8 bits of the result (the AL register), such that the total number of bits set is odd. The AAA instruction operates in this way too, except the parity is computed before bits 4-7 of the result are masked. The AAS instruction is even more strange, If the input is untouched AAS computes parity as usual. Otherwise, it uses a variation of the parity table as shown in Figure 11. In figure 11, an asterisk denotes that this is an opposite value to the usual parity value.

1	0	0	1	*1	*0	1	0	0	1	*0	*1	1	0	*1	*0
0	1	1	0	*0	*1	0	1	1	0	0	1	*1	*0	1	0
0	1	1	0	*0	*1	0	1	1	0	*1	*0	0	1	*0	*1
1	0	0	1	*1	*0	1	0	0	1	*0	*1	1	0	*1	*0
0	1	1	0	*0	*1	0	1	1	0	*1	*0	0	1	*0	*1
1	0	0	1	*1	*0	1	0	0	1	1	0	*0	*1	0	1
1	0	0	1	*1	*0	1	0	0	1	*0	*1	1	0	*1	*0
0	1	1	0	*0	*1	0	1	1	0	0	1	*1	*0	1	0
0	1	1	0	*0	*1	0	1	1	0	*1	*0	0	1	*0	*1
1	0	0	1	*1	*0	1	0	0	1	1	0	*0	*1	0	1
1	0	0	1	*1	*0	1	0	0	1	*0	*1	1	0	*1	*0
0	1	1	0	*0	*1	0	1	1	0	*1	*0	0	1	*0	*1
1	0	0	1	*1	*0	1	0	0	1	*0	*1	1	0	*1	*0
0	1	1	0	*0	*1	0	1	1	0	0	1	*1	*0	1	0
0	1	1	0	*0	*1	0	1	1	0	*1	*0	0	1	*0	*1
1	0	0	1	*1	*0	1	0	0	1	1	0	*0	*1	0	1

Figure 11: Parity Matrix used by the AAS instruction.

We also observed that the specifications we extracted from the Intel manuals and that we used for generating instruction were sometimes incorrect. We show some examples of specifications which we found have errors in Table 4. The three columns show the mnemonic of instruction, the published specification in the Intel Manuals, and the corrected specification. From Table 4, we observe that usually this happens in the representation of MODRM bytes. In other cases, an instruction encoding is over-generalized and generates undefined instructions (e.g. instructions that generate the #UD trap). We use the correct version for generating test cases in our experiments.

Table 4. Some simple examples of discrepancies between published specs and actual CPU encodings

Instruction Mnemonics	Published Spec	Actual Spec
CMPXCHG8B	0000 1111 : 1100 0111 : mod **reg** r/m	0000 1111 : 1100 0111 : mod **111** r/m
IMUL	1111 011w : mod 101 **reg**	1111 011w : mod 101 **r/m**
MUL	1111 011w : mod 100 **reg**	1111 011w : mod 100 **r/m**
CMOVcc	0000 1111: 0100 tttn : mod **mem** r/m	0000 1111: 0100 tttn : mod **reg** r/m

We also found a few repeated specifications. While this might not be a problem for a human reader, for a test generator that generates many billions of tests this is a total waste of time. For instance, for the instruction "MOV- Move to/from Debug registers", there are six repeated specifications, which could use just one specification. Similarly for the "MOV – Move to/from Segment Register" instructions.

Despite their comprehensive details, some instructions were left out entirely from the Intel manuals. For example, we found that opcode x82 is missing in the instruction set specification. The surrounding opcodes x80, x81, and x83 all realize the same group of arithmetic operations that include ADD, OR, CMP, etc. We inferred that x82 would operate similarly, and we estimated the functionality of the entire group of opcodes with the following general encoding specification:

1”000 00sw : mod 000 r/m : immediate data.

From this specification, the special case 0x82 means that “s” is 1 and “w” is 0. Since “w” is 0, we could further infer that operand size for this instruction is 8 bits. And since “s” is 1 we could infer that the 8-bit immediate data will sign-extend to fill the 16bit or 32bit destination. The hardware-oracle confirmed our guesswork.

Table 5. Code reduction due to Code Templates

template name	instruction example	code size (LOC)	instruction refer number	original code size (LOC)	new code size (LOC)	reduced code size (LOC)
JustLikeOpcd0	ADD Eb,Gb	33	8	256	41	215
JustLikeOpcd1	ADD Ev,Gv	63	8	496	71	425
JustLikeOpcd2	ADD Gb,Eb	33	8	256	41	215
JustLikeOpcd3	ADD Gv,Ev	63	8	496	71	425
JustLikeOpcd4	ADD AL,Ib	10	8	72	18	54
JustLikeOpcd5	ADD rAX,Iz	24	8	184	32	152
JustLikeOpcd6	PUSH ES	43	6	252	49	203
JustLikeOpcd64	INC eAX	21	16	320	37	283
JustLikeOpcd112	JO rel8	15	16	224	31	193
JustLikeOpcd128	ADD r/m8, uimm8	27	8	208	35	173
JustLikeOpcd129	ADD r/m32, uimm32	61	8	480	69	411
JustLikeOpcd130	ADD r/m8, imm8	27	8	208	35	173
JustLikeOpcd131	ADD r/m32, imm8	61	8	480	69	411
JustLikeOpcd192	ROL	129	9	1152	138	1014
JustLikeOpcd196	LES Gz,Mp	40	5	195	45	150
JustLikeOpcd2_64	CMOVO Gv,Ev	48	16	752	64	688
JustLikeOpcd2_128	JO Jz	22	16	336	38	298
JustLikeOpcd2_144	SETO Eb	29	16	448	45	403
JustLikeOpcd2_163	BT Ev,Gv	64	4	252	68	184
TOTAL		813	184	7067	997	6070

5.2 Effect of Code Templates

To measure the effect of using code templates we report the reduction in code size in Table 5. The first column shows the template name. The second column shows an example of instructions which uses this template. The third column shows the code size of the template, in lines of code (LOC). The fourth column shows the number of instructions that reused the template. The fifth and sixth columns show the LOC needed for instruction emulation, without and with the template. The seventh column shows the LOC reduction. We currently have 19 code templates, used by 195 instructions or about 80% of our currently supported instructions. The total code reduction is 6,070 lines of code.

5.3 Comparing Testing Strategies

We performed some experiments to compare our routing based testing strategy against the brute force strategy. To keep things within a reasonable time limit, we used our smart test value generators in both cases. We picked the first 81 test vectors in our specs and measured the time and number of test case generated. These test vectors include the most commonly used instructions such as ADC, ADD, AND, CMP, DEC, IMUL, INC, MOV, MUL, OR, PUSH, POP, SBB, SUB, XCHG, XOR. The test time for the brute force and routing test strategies are reported in Table 6. The routing test strategy is, on average, more than 400 times faster, reducing the overall testing time from over 3 hours down to 25 seconds.

Table 6. Test Strategy Comparison, Time

Test vectors	Test time (milliseconds)		Speedup
	Brute force	Routing	
81	11,231,456	25,797	435

Some test cases generate exceptions, either in the oracle, in the emulator, or both. Handling an exception is very expensive time-wise, therefore it is important to evaluate the number of test cases as well, since they do not all cost the same. As shown in Table 7, the routing test strategy only generates about 0.25% of the test case generated by the brute force approach.

Table 7. Test Strategy Comparison, Test Counts

Test vectors	Test cases		Reduction
	Brute force	Routing	
81	1.966E+09	4.918E+06	99.75%

The test strategies do not make too much of a difference when measuring the throughput, e.g. the number of tests generated and executed (by both oracle and emulator) per unit time. Routing generates and completes about 190 tests per millisecond, while brute force generates and completes about 175 tests per millisecond, with a difference of just 7.8%. We conclude that the speedup obtained from the routing strategy derives overwhelmingly from the reduced number of test cases.

The detailed data for the test time comparison is shown in Figure 12. The speedup of the routing strategy versus brute force is shown in Figure 13. There are some low points both in Figure 12 and in Figure 13. This is because much fewer test cases are generated when one operand is fixed such as the AX register. Figure 14 and Figure 15 show the detailed data for the test counts. Brute force generates far more test cases, because its complexity is O(MxN) while routing has complexity O(M+N). We also see some common trends within a test vector family (same instruction mnemonics). The test case count will increase when the operands are a fixed operand, a register operand or the operands include MODRM bytes.

Table 8 summarizes the results from testing, at the time of this writing[1]. The first columns reports the total number of test vectors generated from the specifications. A number of these test vectors cannot run on the user mode oracle; the second and third columns report the counts of runnable and not runnable test vectors, respectively. Vectors are not run because they are duplicated, or they only really work in privileged mode, or the oracle machine does not support the instruction (like MOVBE), or we have not recreated enough state on the target machine (like the full global descriptor table), or they are too disruptive to the processor state and generate non-recoverable exceptions (like loading an invalid selector in the DS segment), or we simply have not yet added the code to create a controllable execution (like an arbitrary far CALL, for instance). The fifth column reports the total number of test cases generated and run to completion. The sixth column reports the count of tests that generate exceptions, which we catch and then verify that they are the same on both oracle and emulator. The seventh and eight columns report the count of tests passing and failing, respectively. There are a number of reasons for these failures, some we can improve upon in the future and others are more fundamental. The bit string instructions use a pointer and a full 32-bit offset, in the boundary cases they reach all over the oracle's process address space. POPFD can set the trap bit, which takes effect immediately in the next instruction. On the oracle, the OS clears the flag but on the emulator we only execute one instruction and we see the flag still set. The RDTSC instruction will never return the same counter value on two separate machines.

[1] The emulator is in active development, these results are only preliminary and change constantly.

RELATED WORK

The virtualizability of 250 instructions of the Intel Pentium architecture was analyzed by Robin and Irvine [26]. Bochs[8] is a full system emulator using interpretation techniques. Similar to Bochs, our emulator also uses interpretation techniques; however, our simulator focuses on the real-time simulation. Dynamic binary translation was used in QEMU for CPU emulator [13]. Ptlsim is another full system emulator [28]. Demand emulation was used to track tainted data [31]. Time warped time was used for network emulation [32]. Operating System level virtualization trade both isolation and efficiency [29][30]. Xen [44] developed the hypervisor based virtual machine based on hardware extensions.

Virtual machine emulators have become pervasively utilized tools in computer security for malware analysis. Many researchers built tools based on whole system emulators. The analyzed applications can be isolated in the emulated environment for observations. For example, Bochs [14] was used to analyze packed malware [35] and data lifetime [38]. Many researchers also developed platforms [36][37][39][40][41][42] for analyzing binaries based on the full system emulator QEMU. Some researchers also developed malware analysis platform [45] [46] based on the Xen hypervisor [44] .

Malware authors have developed techniques to find the difference between emulated environment and the real hardware environment. Then malware will behave differently in the emulation environment. Rutkowska proposes to use just one instruction to detect a virtual machine[22]. Ormandy [19] presents server implementation

defects in emulators. Quist et al. [20] propose to use local data table to detect a virtual machine. Raffetseder et al. proposed several attacks to detect emulators [21]. Also test oracle was used to generate red-pill for detection of CPU emulator [43] . The closest related work about emulator testing is [17]. This paper proposes an automatic technique to generate *random* instructions and uses real machine as test oracle to find a large class of differences. Our method share the same principal approach, but using a routing based test generation, however, our methods systematically generate the instructions according to the test vectors and guarantee test coverage. Emulation technique is used to repair the state difference between emulator and reference system[47]. Routing test was used in hardware testing to validate hardware architecture [27]. We use the idea of routing test to quickly generate test cases and reduce the complexity.

7. Conclusions

In this paper, we show that the hardware-oracle based testing-debugging architecture is highly effective in supporting the development of CPU emulators. The hardware-oracle is lightweight and precise. It can run either as a separate program, or directly embedded into the emulator. We developed an automatic tester that extracts the CPU specifications directly from the published CPU datasheets, and automatically generates test cases with complete test coverage. The number of required tests is reduced by many orders of magnitude by the separation of operand routing from computation. With the same test coverage, our routing based test generation strategy is more than 400 times faster than a brute force approach, reducing the test time for a sample of test vectors from more than 3 hours to less than 25 seconds. Experimental results show that our design significantly reduces the development time over the post-implementation testing methodology. The hardware-oracle detects and documents several undocumented CPU behaviors which would be very hard to detect by conventional testing methods. The proposed design and testing process is versatile, and can be applied to any CPU emulator.

We used a number of code patterns realized as macros in the implementation. These patterns are useful to save code, to fix many bugs with fewer code changes, and to better understand and document the architecture. By reusing the common patterns, the CPU emulator could conceivably be translated into Verilog and implemented on an FPGA.

References

[1] A. Forin, B. Neekzad, and N. L. Lynch. Giano: The twoheaded system simulator. Technical report vol. msr-tr-2006-130, Microsoft Research, One Microsoft Way, Redmond,WA 98052, 2006.

[2] S.Mohan and J. Helander. Temporal analysis for adapting concurrent applications to embedded systems. In ECRTS, 2008.

[3] Phoenix Compiler. http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx

[4] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, Danilo Bruschi, Testing CPU emulators, Proceedings of the eighteenth international symposium on Software testing and analysis 2009

[5] S. Bhansali, W. Chen, S. De Jong, A. Edwards, andM. Drinic. Framework for instruction-level tracingand analysis of programs. In Second InternationalConference on Virtual Execution Environments VEE,2006

[6] Patrice Godefroid, Michael Y. Levin, David Molnar, Automated Whitebox Fuzz Testing, Proceedings of NDSS'2008.

[7] David A Molnar, David Wagner. "Catchconv: Symbolic execution and run-time type inference for integer conversion errors". Technical report, University of California Berkeley, 2007-23, February, 2007

[8] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. Linux Journal 1996

[9] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture (2006)

[10] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2B: Instruction Set Reference, N-Z (2006)

[11] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide, Part 2 (2006)

[12] Nicholas Nethercote, Julian Seward, Valgrind: A Program Supervision Framework, Electronic Notes in Theoretical Computer Science, Volume 89, Issue 2, October 2003

[13] F Bellard, QEMU, a fast and portable dynamic translator, Proceedings of the USENIX Annual Technical Program 2005

[14] KP Lawton, Bochs: A portable pc emulator for unix/x, Linux Journal, 1996

[15] CK Luk, R Cohn, R Muth, H Patil, A Klauser, Pin: Building customized program analysis tools with dynamic instrumentation , PLDI’05

[16] 8086 Microprocessor Emulator, http://www.emu8086.com/

[17] L. Martingnoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, D., “ Testing CPU emulators”, Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA), Chicago, Illinois, U.S.A. (July 2009), ACM

[18] P. Ferrie, “Attacks on Virtual Machine Emulators”, Tech. report, Symantec Advanced Threat Research, 2006

[19] T. Ormandy, “An Empirical Study into the Security Exposure to Host of Hostile Virtualized Environments”, In Proc. of CanSecWest Applied Security Conference, 2007

[20] D. Quist, and V. Smith, “Detecting the Presence of Virtual Machines Using the Local Data Table”, http://www. offensive computing.net/files/active/0/vm.pdf

[21] T. Raffetseder, C. Kruegel, and E. Kirda. “Detecting System Emulators”, Proc. of Information Security Conference (ISC 2007). Springer-Verlag, 2007

[22] J. Rutkowska. “Red Pill. . . or how to detect VMM using (almost) one CPU instruction”, http://invisiblethings.org/ papers/redpill.html

[23] http:// research.microsoft.com/en-us/projects/giano/ default.aspx.

[24] Burger, D., Austin, T. M. The SimpleScalar Tool Set, Version 2.0.Technical Report 1342, June 1997, University of Wisconsin-Madison.

[25] M. Pezze and M. Young, Software Testing and Analysis: Process,Principles and Techniques. Wiley, April 200.

[26] John Scott Robin , Cynthia E. Irvine, Analysis of the Intel Pentium's ability to support a secure virtual machine monitor, Proceedings of the 9th conference on USENIX Security Symposium, p.10-10, August 14-17, 2000, Denver, Colorado

[27] Aktouf, C., Robach, C., and Marinescu, A. 1995. A Routing Testing of a VLSI Massively Parallel Machine Based on IEEE

1149.1. In Proceedings of the IEEE international Test Conference on Driving Down the Cost of Test (October 21 - 25, 1995).

[28] M. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in ISPASS '07, Apr. 2007.

[29] Stephen Soltesz, Herbert P?tzl, Marc E. Fiuczynski, Andy Bavier,and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In Proceedings of the EuroSys conference, pages 275–287, 2007.

[30] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In Proceedings of the USENIX Large Installation Systems Administration Conference, 2004.

[31] Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S. 2006. Practical taint-based protection using demand emulation. SIGOPS Oper. Syst. Rev. 40, 4 (Oct. 2006), 29-41.

[32] Gupta, D., Yocum, K., McNett, M., Snoeren, A. C., Vahdat, A., and Voelker, G. M. To infinity and beyond: time warped network emulation. In SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles (2005).

[33] RUTKOWSKA, J. Red Pill. . . or how to detect VMM using (almost) one CPU instruction. http://invisiblethings.org/papers/redpill.html.

[34] Danny Quist and Val Smith "Detecting the Presence of Virtual Machines Using the Local Data Table" http://www.offensivecomputing.net/files/active/0/vm.pdf

[35] L. Böhne. Pandora's Bochs: Automatic unpacking of malware. Diploma thesis, RWTH Aachen University, Jan.2008.

[36] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In Proceedings of the 4th International Conference on Information Systems Security, Hyderabad, India, Dec.2008.

[37] BitBlaze Online. https://aerie.cs.berkeley.edu/.

[38] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In 13th USENIX Security Symposium,San Diego, CA, USA, Aug. 2004.

[39] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In 15th EICAR Conference, pages 180–192, Hamburg, Germany, May 2006.

[40] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In 23rd Annual Computer Security Applications Conference (ACSAC), pages 431–441, Miami Beach, FL, USA, Dec. 2007.

[41] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In ACM Conference on Computer and Communication Security (CCS), Alexandria, VA, USA, Oct. 2007.

[42] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In 5th ACM Workshop on Recurring Malcode (WORM), Oct. 2007.

[43] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia and Danilo Brusch, "A fistful of red-pills: How to automatically generate procedures to detect CPU emulators", In the Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT), Montreal, Canada, August 10, 2009.

[44] Paul Barham , Boris Dragovic , Keir Fraser , Steven Hand , Tim Harris , Alex Ho , Rolf Neugebauer , Ian Pratt , Andrew Warfield, Xen and the art of virtualization, Proceedings of the nineteenth ACM symposium on Operating systems principles, October 19-22, 2003, Bolton Landing, NY, USA.

[45] Lionel Litty, H. Andrés Lagar-Cavilla and David Lie. Hypervisor Support for Identifying Covertly Executing Binaries In the 17th USENIX Security Symposium. Pages 243-258. July 2008.

[46] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In 15th ACM Conference on Computer and Communications Security (CCS), pages 51–62, Alexandria, VA, USA, Oct. 2008.

[47] Min Gyung Kang, Heng Yin, Steve Hanna, Steve McCamant, and Dawn Song, "Emulating Emulation-Resistant Malware", In Proceedings of the 2nd Workshop on Virtual Machine Security, November 2009.

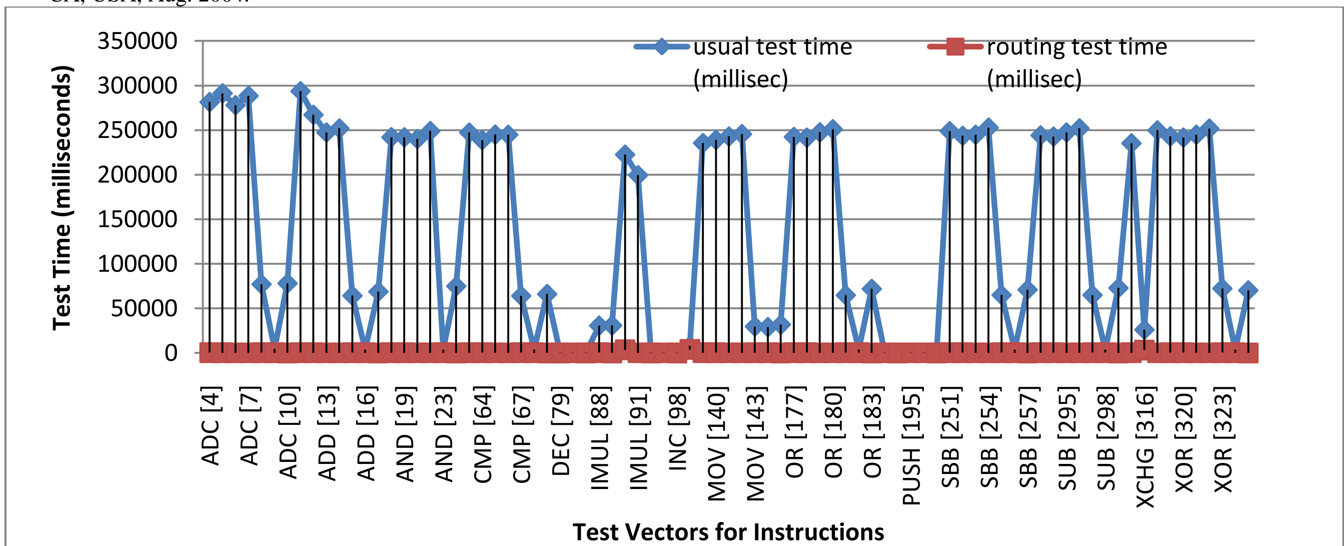

Figure 12: Completion time for selected Test Vectors

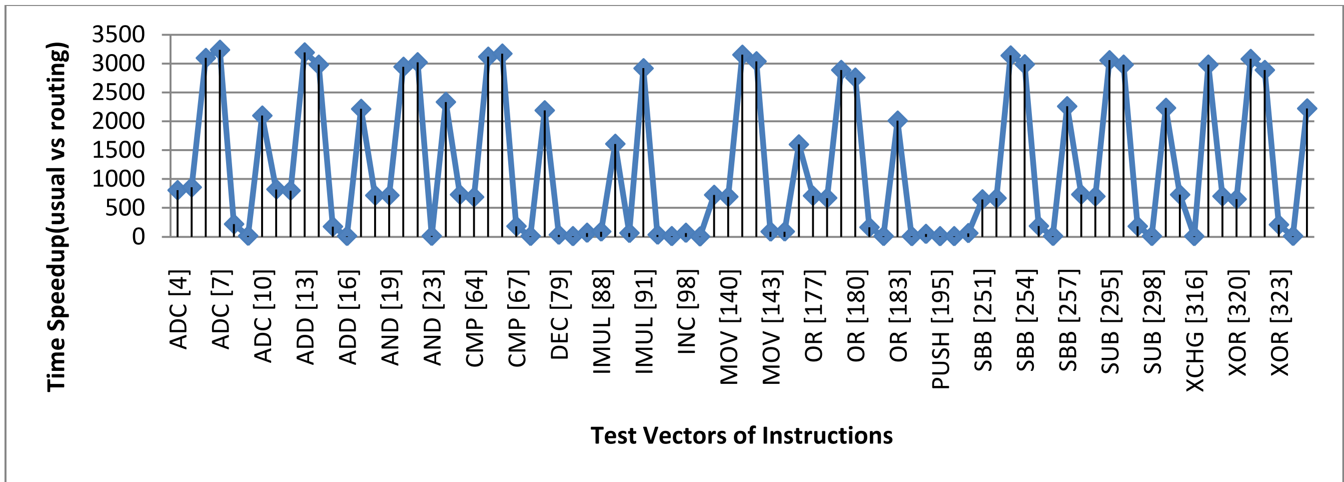

Figure 13: Speedup in test time completion

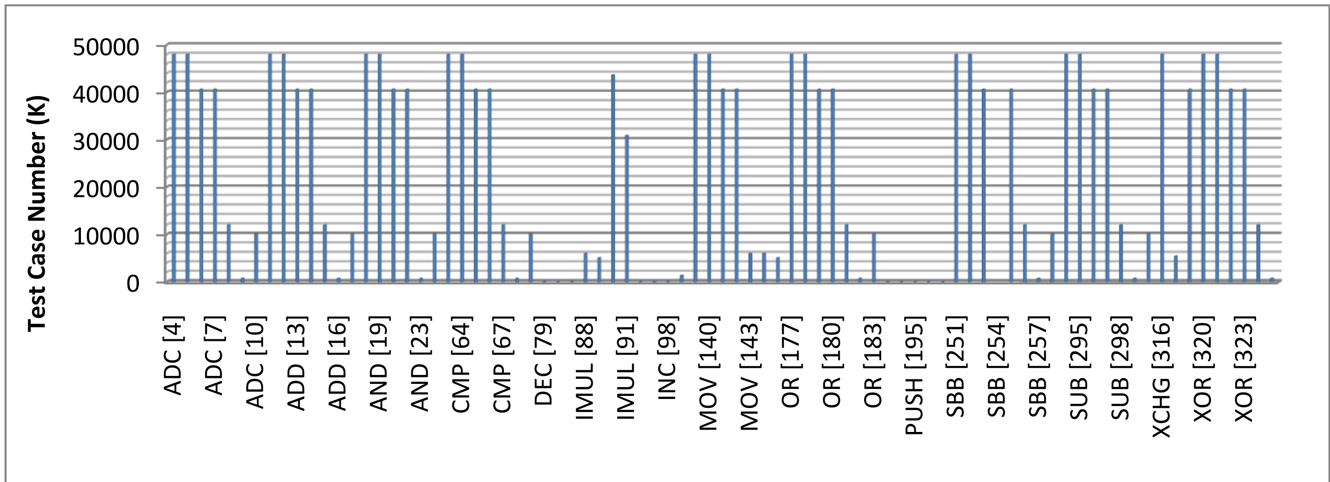

Figure 14: Counts of generated tests of brute force test

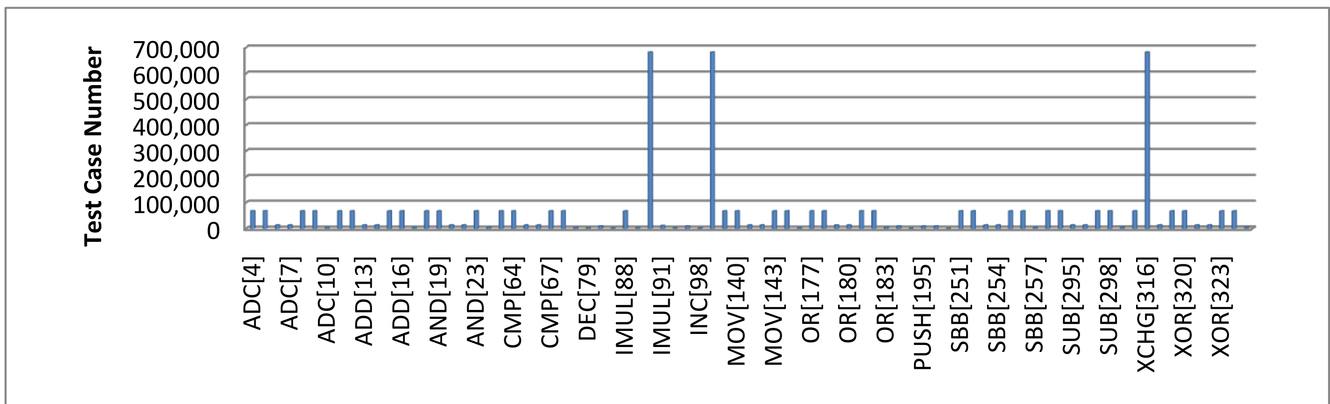

Figure 14: Counts of generated tests of routing test

Table 8 Test execution results

Vectors	Runnable	Not Run.	%Run	Tests	Traps	Pass	Fail	%Fail
328	269	59	82.01	24,669,044,880	97,832,937	24,649,159,730	19,885,150	0.0806