Simulation of CPUID of Intel Architecture

Grigory Rechistov* Name Surname†

March 24, 2014

Contents

Abstract

In this paper we give an overview of existing microprocessor features identification facilities. Then we describe our approach to implementation of a software model of Intel IA-32 CPUID instruction. The described solution allows to define all recent CPUs' features, as well as future extensions. Our model was incorporated into the Wind River Simics* simulator framework.

Key words: cpu identification, cpu simulation, CPUID, Simics.

*Moscow Institute of Physics and Technology, grigory.rechistov@phystech.edu
†email@mail.com

1 Introduction

Most of documents defining CPU architectures do not specify every implementation detail; only interfaces, such as register state and instruction set (ISA), are defined. Microarchitectural details are usually left to vendors.

Any more-or-less mature processor architecture will have a set of extensions meant to improve performance, reliability, energy consumption or any other aspect of computer operation. A long evolution happened with backwards compatibility in mind and influenced by strong market competition will lead to a situation when there are quite a few of extensions, sometimes contradicting each other but nevertheless designated as "compatible" architectures.

Software programmers, who wish to use a certain architecture extension in their code, should be able to reliably identify its presence at run time. Therefore, processors provide means of feature identification in forms of instructions and/or registers which, being accessed, yield values that encode extended capabilities: vendor name, model name, machine word width, register count and similar model specific information.

Software simulators serve needs of pre-silicon software development, system firmware debugging, architecture and algorithm exploration etc. As with hardware, in order to be useful, software embodiments of processor specifications must faithfuly declare supported extensions. Yet, building a software model for processor feature identification may turn out to be a non-trivial task, complicated by a bulk and complexity of information that needs to be represented, especially when a number of supported processor models is high.

From our experience with pre-silicon simulation tools, certain types of target software, such as BIOS, firmware and drivers, are quite sensitive to processor identification and will refuse to work or crash if a single bit of it is wrong. At the same time, such software in its early age may be incomplete; a software model has to adapt to mimic features that are absent for it to work. A naïve ad-hoc approach to feature data storage and handling will lead to a code that is very hard to extend support. These circumstances lead us to develop a dedicated framework for simulation of CPUID instruction, described in the paper.

The rest of this paper is organized as follows. In the section 2, we present a survey of approaches to defining model-specific features used in different microprocessor architectures. In the section 3 it is shown that the Intel IA-32 has one of the most complex CPUID instruction. Section 4 describes approaches to CPUID representation and simulation found in several software models. A new specification language for CPUID definition and tools for generation of simulation code are described in section 5. Our experience with its integration into an industrial simulation framework Wind River Simics* are shown in section 6. Section 7 gives final remarks and future work directions.

1.1 Contributions

In this paper we make the following contributions.

1. Comparison of existing approaches to extensions/feature identification of different modern processor architectures. To our knowledge, no such survey has been published yet.

2. Describe design, implementation and evaluation of a structured solution to the simulation of CPUID instruction of Intel IA-32. Analysis of several open source and proprietary simulators has shown that CPUID simulation is traditionally made ad-hoc, resulting in an entanglement of code which is hard to support.

3. Define essential user interface facilities that firmware developers demand from a CPUID simulation. We describe how CPUID should be represented for inspection, and what types of manupulation should be allowed to an end user.

2 Overview of Processor Identification

This section outlines mechanisms for processor identification employed by several modern general purpose microprocessor architectures. We do not attempt to give exhaustive descriptions of instruction sets; nor we intend to give a historical overview of now discontinued designs. The goal is to highlight differences and similarities between systems. Detailed explanations for all mentioned registers and instructions can be found at respective reference manuals cited below.

2.1 MIPS

MIPS CPUs store processor identification within PRid register, which is the 15th register in Coprocessor 0 [8]. It contains 32 bits of information (Fig. 1), part of which is vendor–specific.

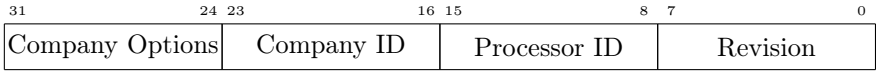

Figure 1: MIPS PRid register fields

MIPS architecture also implements special register that contains information to list capabilities of floating point unit, termed Floating Point Implementation Register (FIR) [14]. It is a read-only 32-bit register (Fig. 2).

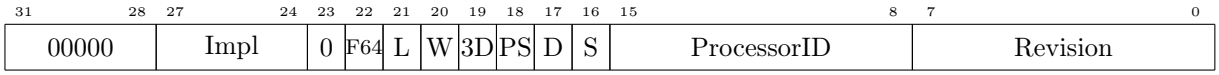

Figure 2: MIPS FIR register fields

TODO Check this

`http://hwdb.mipt.cc/MIPS_PRId_register`

`http://code.google.com/p/phantomuserland/source/browse/trunk/phantom/dev/mips/cpuid.c?r=1094`

`http://www.imgtec.com/mips/mips32-architecture.asp`

2.2 ARM

The popular RISC architecture ARM provides two different ways to identified cores. This ways described in [2].

The first way is to read info form the Register 0 of the System Control Coprocessor also called CPUID base register [3]. As with MIPS, it holds 32 bits of information and look a bit differ for ARM7 core family (Fig. 3) and ARM9 and later core families (Fig. fig:arm-cpuid-v9).

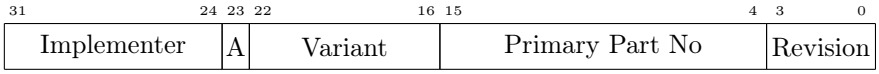

Figure 3: ARM7 Core Family CPUID base register fields

Examples of values [11]: Intel (XScale) PXA272 – 0x69054117, Qualcomm MSM7200A — 0x4117b362.

The other way that ARM core can be identified is through a TAP ID [2] that used ton configure debug software and indicates the information about an ARM core.

2.3 IBM System z

IBM System z10 [6] provides `STSI` and `STIDP` instructions to obtain processor information.

The `STSI` instruction used to determine processor model and model capacity identifier for the base configuration and for any additional changes through update actions (Fig. 5).

The `STIDP` instruction provides information about the processor type, serial number, and logical partition identifier (Fig. 6).

2.4 PowerPC

PowerPC [7] offers a 32-bit PVR register that contains just version and revision numbers.

For ISA extensions a mechanism of APU (application processor units) is employed. MSR (machine state register) is used to store information of APUs available.

TODO See also

`http://lxr.linux.no/#linux+v3.13.5/arch/powerpc/kernel/cputable.c#L2245` — identify_cpu function

`http://cache.freescale.com/files/archives/doc/support_info/PPCPVR.pdf`

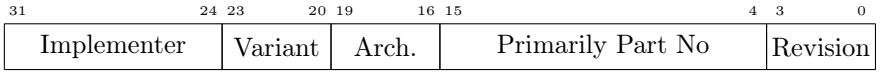

Figure 4: ARM9 and later Core Family CPUID base register fields

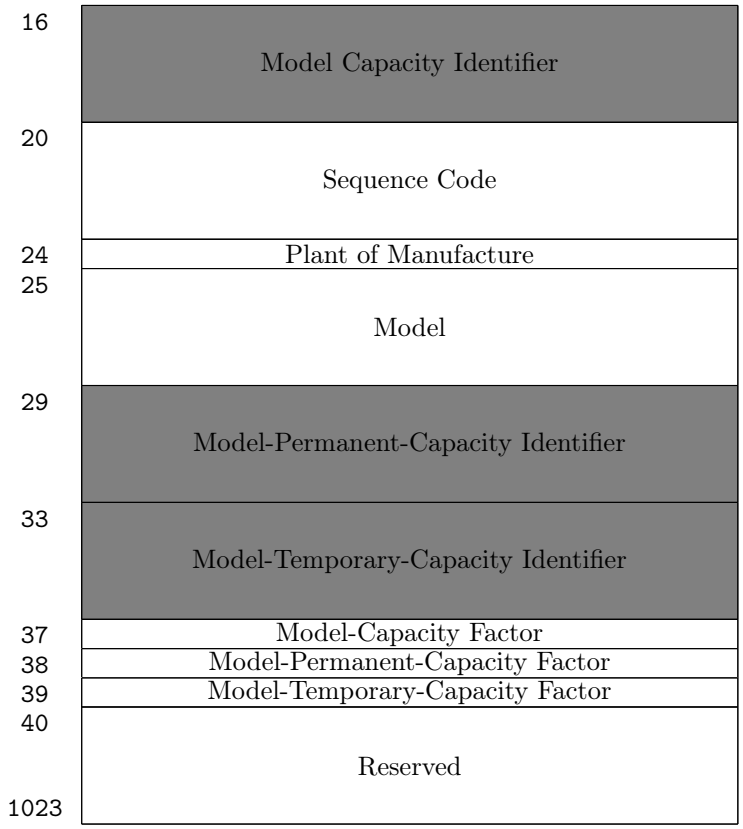

Figure 5: `STSI` output

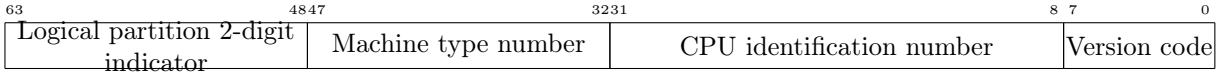

Figure 6: `STIDP` output

2.5 SPARC

The SPARC v9 standard [17] leaves quite a number of details implementation specific. To distinct between version a 64-bit register VER is defined (Fig. 7).

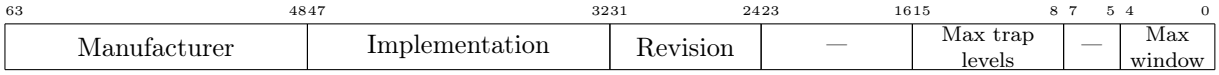

Figure 7: SPARC v9 VER register fields

2.6 Intel IA-64 (Itanium)

Intel IA-64, also known as Itanium™ [10] was conceived after the original Intel 80x86 series and was meant to supplant it. A set of CPUID registers is used for identification purposes. The set's design somewhat resembles IA-32 CPUID instruction (discussed shortly after) — its register numbers loosely resemble leaves of the latter. At the moment of this writing, all announced IA-64 systems offered up to five CPUID registers. To have room for feature expansion, bits 0–7 of CPUID[3] store the actual size of the register set (limiting it to 256 entries).

A CPUID table for an Itanium 2 system is given on Fig. 8. Total capacity of it can be estimated as $5 \times 64 = 320$ bits.

```
Leaf                 Value
-----------------------
  0  0x49656e69756e6547
0x1           0x6c65746e
0x2                    0
0x3           0x20010104
0x4                  0x5
```

Figure 8: Contents of CPUID registers for an Intel Itanium 9100 (code name Montvale) system using `ggg-cpuid` [16]

2.7 Intel IA-32 and Intel 64

The common IBM PC architecture, starting from Intel Pentium™and its clones, provides CPUID instruction [9, 1]. The 64-bit extension of IA-32, known as Intel 64 or AMD64, did not change CPUID and we will make no further distinction between them throughout this paper. and we will make no distinction between them. CPUID takes two input operands in 32-bit registers EAX and ECX (called *leaf* and *subleaf*) and puts the result to four 32-bit registers, namely EAX, EBX, ECX and EDX.

Since its inception it has been extended numerous number of times.TODO Expand

On Figure 9 an output of all defined leaves and subleaves for a modern Intel processor (of microarchitecture Intel Sandy Bridge) is shown. The resulting table contains 24 tuples of 4 values of 32 bit width each, total of more than 3 kbit of information. Essentially a number of features can be deduced from it.

- Processor producer brand (leaf 0) and SKU naming (leaves 0x80000002–0x80000004).
- Availability of ISA extensions such as 64 bit mode, SSE2/3/4.1/4.2, AVX, MOVBE etc.
- Cache configuration of all layers, both in legacy format (leaf 2) and in current format (leaf 4 with subleaves).
- Multi-processor configuration knowledge, such as availability of Intel HyperThreading, relative position inside CPU package (topology at leaf 0xb), presence of APIC (interrupt controller).
- Numerous parameters for the implementation, such as addresses widths (leaf 0x80000008), availability of dynamic frequency scaling, supported debugging facilities etc.

Leaf	Subleaf	EAX	EBX	ECX	EDX
0	0	0xd	0x756e6547	0x6c65746e	0x49656e69
0x1	0	0x206d6	0x3200800	0x1fbee3ff	0xbfebfbff
0x2	0	0x76035a01	0xf0b2ff	0	0xca0000
0x4	0	0x3c004121	0x1c0003f	0x3f	0
0x4	0x1	0x3c004122	0x1c0003f	0x3f	0
0x4	0x2	0x3c004143	0x1c0003f	0x1ff	0
0x4	0x3	0x3c07c163	0x4c0003f	0x3fff	0x6
0x5	0	0x40	0x40	0x3	0x21120
0x6	0	0x77	0x2	0x9	0
0x9	0	0x1	0	0	0
0xa	0	0x7300403	0	0	0x603
0xb	0	0x1	0x2	0x100	0x3
0xb	0x1	0x5	0x10	0x201	0x3
0xd	0	0x7	0x340	0x340	0
0xd	0x1	0x1	0	0	0
0xd	0x2	0x100	0x240	0	0
0x80000000	0	0x80000008	0	0	0
0x80000001	0	0	0	0x1	0x2c100800
0x80000002	0	0x20202020	0x49202020	0x6c65746e	0x20295228
0x80000003	0	0x6e6f6558	0x20295228	0x20555043	0x322d3545
0x80000004	0	0x20303936	0x20402030	0x30392e32	0x7a4847
0x80000006	0	0	0	0x1006040	0
0x80000007	0	0	0	0	0x100
0x80000008	0	0x302e	0	0	0

Figure 9: Output of CPUID instruction obtained for Intel® Xeon® CPU E5-2690 using `ggg-cpuid` [16]

2.8 Comparison of Processor Identification

Based on the presented data several conclusions can be made.

- CPU identification facilities differ greatly between architectures. They may be represented by instructions, registers, or groups of both.
- The common thing that is usually conveyed through a CPUID is a vendor identification. The next on popularity is indication of ISA extensions. Lastly, for multiprocessor systems, values for enumerating cores, threads or processors is often given.
- The "verbosity" of CPUID facilities depends on whether there are requirements to run the same binary code on hardware from multiple vendors and/or of different generations. If, in order to perform efficiently, software must "know" a list of supported instruction extensions and other types of model specific configuration, there has to be a documented way to obtain such knowledge.
- Conversely, CPUs provided by a single vendor and/or designed for specific software usually provide less means of self-identification. Most microcontrollers for embedded applications hardly provide even an idea of CPUID , compared to general purpose processors, because software is often written to be run on a specific HW; binaries are not meant to be moved to some other incarnation of the same architecture.

3 What is so Hard about IA-32 CPUID

We now concentrate solely on the Intel IA-32 architecture and its CPUID . The previous section demonstrates that this instruction generates significantly more data than any other identification mechanism of the rest of surveyed systems. Indeed, a complete definition of CPUID in [9] takes about 40 pages.

3.1 Subleaves, Dynamic Fields and Topology

Let's go through all the complications of CPUID that have to be modeled in a software simulator.

Elements adressing. An "address" of a particular extension field inside a CPUID table comprises of four values.

- Leaf number. It's the input value of EAX for CPUID invocation. There are two continuos numerical ranges for leaf values: the first starting from zero and the second starting from 0x80000000 (see Fig. 9).
- Subleaf number. An input value of ECX for CPUID invocation. Only a few leaves actually have a non-zero number of subleaves; for example, leaf 4 uses them to distinct between cache levels. For the rest, the subleaf is ignored and the instruction's output does not depend on it.
- Output register. Execution of CPUID always changes four registers, effectively producing a 128-bit value. The register name is specified to choose a 32 bit range inside it.
- Bit range. A field can occupy from one to 32 bits of an output register.

In order to fully specify placement of a CPUID field, all these values must be known. For example, the following notation is used in documentation to denote presence of MMX technology: `CPUID.01H:EDX.MMX[bit 23]`.

Dynamic fields. Firmware can affect output of CPUID by enabling and disabling architectural extensions. Let's have some examples.

1. Disabling multiple cores and Intel HyperThreading technology affects the CPUID.1:EDX.HTT[28].
2. Changing bit 18 of register CR4 affects CPUID.1:ECX.OSXSAVE[27].
3. A number of bits in the IA32_MISC_ENABLE model specific register affect CPUID . Thus, its bit 16 affects indication of EIST field, bit 34 disables Execute Disable feature and its indication in CPUID etc.

These facts mean that one cannot just use a constant array to represent the table in the simulation — some bits will be dynamic and have to be calculated every time an instruction is run.

Topology-variable elements. In multiprocessor systems different processor cores inside the same CPU package will have CPUID returning different values, because certain fields indicate relative position of them (topology). For example, the 8-bit of APIC ID of a core is returned in CPUID.1.EBX[24:31]. This means that for simulation we cannot store a single table for all cores.

Finally, it should be noted that, besides EAX, EBX, ECX and EDX, one more register may be affected by CPUID execution, namely IA32_BIOS_SIGN_ID MSR.

3.2 Why CPUID is so complex

One can find roots of having such entangled (and nonsurprisingly slow) instruction by studying the history of IA-32 architecture.

Influence of multiple competing vendors. Until recently, there were numerous CPU vendors that offered processors compatible with Intel architecture, including IBM, Cyrix, VIA, Centaur, Transmeta etc. By 2014, considerably fewer companies remain — in particular Intel itself and AMD. A coordination committee to define how new IA-32 extensions are indicated have not existed. Uncontrolled competition lead to the numerous small and subtle but essential differences in implementation. Until (and for some time after) CPUID was introduced, a robust identification of IA-32 processor brand and model required surprisingly intricate methods [5]. At present times, things are somewhat better documented, but still are not controlled in a centralized manner.

Long history that lead to extremely long list of extensions. With 40 years of backwards-compatible development the IA-32 architecture collected a number of additions. Consider a list of flags which a modern GNU/Linux operating system shows for a CPUID of the laptop this paper is being written on (Fig. 10). It should be noted that only a part of information from CPUID is actually present on this list.

```
flags :  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon
pebs bts nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx
est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer xsave
avx lahf_lm arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority ept vpid
```

Figure 10: Part of output of `cat /proc/cpuinfo` command on GNU/Linux on a recent Intel IA-32 CPU

4 Existing Approaches to CPUID Simulation

Every software simulator, emulator or virtual machine of a recent (Pentium or later) IA-32 system must guarantee CPUID operation of certain accuracy. If such tool is used for system firmware development, which is even more sensitive to identification information, the following requirements should be.

- Be accurate TODO
- Be configurable TODO

4.1 Bochs

Bochs [13] TODO

4.2 Xen

Xen [15] TODO *

4.3 Qemu

Qemu [4] TODO

4.4 Simics

Wind River Simics [12] TODO

5 CPUID Definition and Generation

The approach to be described

5.1 Starting Considerations

The following considerations have been taken into account when designing the specification.

- Have as much regular format as possible.
- Use a natural unit of CPUID configuration state — a field.
- Move computation tasks from run-time to the translator
- Users often think and operate in terms of 32 bit register values which are more compact and therefore convenient.

5.2 Specification of CPUID

Fields definition. This is a common part of all models. Here, the exact position, name, and properties of all fields are specified.

Types of the fields:

1. const —
2. field —
3. fuse —
4. func —

Model Definition. For every CPU model a separate file section formulates what actual values for fields/fuses/func's the model should have.

At model definition section, actual values are set up for all fields defined in the fields section.

A (shortened) example of an input specification is given at Fig. 11.

```
[FIELDS]
# name    leaf sleaf  reg  range   type flags  defval       comment
  sse3,      1, None, ecx,     0, field,    0,      0,      Availability of SSE3
cache_descr, 1, None, eax,  0:31,  fuse,    0,   0x62,      Cache descriptor
apic_id,     1, None, ebx, 24:31,  func,    0, get_apic_id, APIC ID

[CPUID]
# class, field,      value
Atom,    sse3,       1
Atom,    logical_no, atom_apic_id
Atom,    4:1,        0x11111111:0x22222222:0x33333333:0x44444444
```

Figure 11: Examples of fields and model definitions. Lines starting with "#" are ignored

5.3 Translation to Code

We implemented a translator called CPUIDGEN that accepts a file with the specification outlined above and generates source code files, which are included into the build. Fig. 12 outlines the process of integration of the new functionality with the rest of the simulation framework.

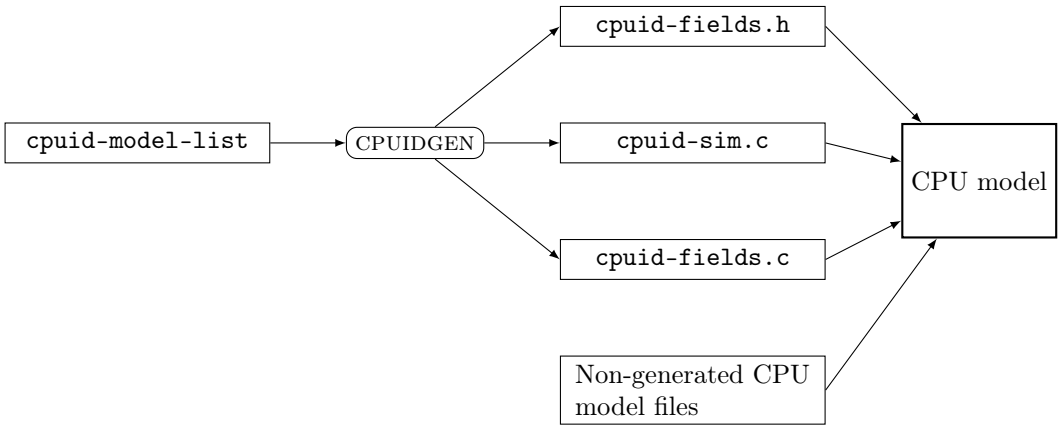

Figure 12: Steps of building for a CPU model

The build consists of these steps.

1. CPUIDGEN reads cpuid-model-list and produces three files: a C header with definitions of all structures and function prototypes, a C source for the main simulation function, `cpuid_simulate()`, and C source for auxiliary functions.
2. New sources are included, built and linked with the rest of the CPU model files.

5.4 Generated Code

The `cpuid_simulate()` is a C function that consists of two levels of `switch` operators — the outer level is for leaf selection, and the inner — for subleaf selection, if any subleaves are present. When leaf/subleaf combination is filtered out, all relavant CPUID fields are combined to create four 32-bit values of output registers. This allows to keep algorithmical complexitiy of the algorithm to be $O(1)$ of number of leaves and subleaves, and $O(N)$ of number of fields. The previous implementation performed a linear search for matches against input leaf/subleaf, which, of course, is less efficient. The translator shifts this search to the compilation time, thus saving run-time.

6 Evaluation

6.1 Compatibility with existing code

"legacy"

6.2 Overriding Facilities

TODO the picture of CPUID dump with overridden fields highlighted.

6.3 Extensions

TODO Field flags, such as "hidden".

7 Conclusions

In this paper we described our approach to simulation of a single but complex processor instruction CPUID .

Model specific registers. Besides CPUID , processors of Intel architecture may have a quite extensive of model specific registers (MSRs), which can be read/written with RDMSR/WRMSR instructions, and also contain bits of processor identification. The simulation of MSRs is also an important aspect, but it is out of this paper focus. Other architectures may have similar facilities, e.g. special purpose registers (SPRs) in PowerPC.

Acknowledgements

Thanks to my parents for raising an awesome me.

References

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, 2012.

[2] ARM Limited. *Application Note 99. Core Type & Revision Identification*, November 2003.

[3] ARM Limited. *CPUID Base Register*, 2008.

[4] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *FREENIX Track: 2005 USENIX Annual Technical Conference*, 2005.

[5] Robert R. Collins. CPUID algorithm wars. *Dr. Dobbs*, November 1996.

[6] Per Fremstad, Wolfgang Fries, Marian Gasparovic, Parwez Hamid, Brian Hatfield, Dick Jorna, Fernando Nogal, and Karl-Erik Stenfors. *IBM System z10 Enterprise Class Technical Guide*. IBM, November 2009.

[7] IBM. *PowerPC® Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors Version 3.0*, July 2005.

[8] Imagination Technologies. *MIPS32 Architecture*, 2014.

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volumes 1–3*, 2012.

[10] Intel Corporation. *Intel® Itanium® Architecture Software Developer's Manual. Volumes 1–4*, 2010.

[11] intruders. ARM CPUID - processor model detection, 2009.

[12] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35:50–58, February 2002.

[13] Darek Mihoka and Stanislav Shwartsman. Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. *ISCA-35 Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2008.

[14] MIPS Technologies, Inc. *MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture*, March 2011.

[15] Ian Pratt. Xen and the art of virtualization, 2006.

[16] Grigory Rechistov. A set of CPU identification tools for Intel IA-32 and IA-64 systems, 2014.

[17] D.L. Weaver, T. Germond, and SPARC International. *The SPARC architecture manual: version 9*. PTR Prentice Hall, 1994.