ISA Aging: A X86 case study

Bruno Lopes, Rafael Auler, Rodolfo Azevedo, Edson Borin
University of Campinas
Institute of Computing
{blopes,rauler,rodolfo,edson}@ic.unicamp.br

Abstract—

Microprocessor designers such as Intel and AMD implement old instruction sets at their modern processors to ensure backward compatibility with legacy code. In addition to old backward compatibility instructions, new extensions are constantly introduced to add functionalities. In this way, the size of the IA-32 ISA is growing at a fast pace, reaching almost 1300 different instructions in 2013 with the introduction of AVX2 and FMA3 by Haswell. Increasing the size of the ISA impacts both hardware and software: it costs a complex microprocessor front-end design, which requires more silicon area, consumes more energy and demands more hardware debugging efforts; it also hinders software performance, since in IA-32 newer instructions are bigger and take up more space in the instruction cache. In this work, after analyzing x86 code from 3 different Windows versions and its respective contemporary applications plus 3 Linux distributions, from 1995 to 2012, we found that up to 30 classes of instructions get unused with time in these software. Should modern x86 processors sacrifice efficiency to provide strict conformance with old software from 30 years ago? Our results show that many old instructions may be truly retired.

I. Introduction

Nowadays, the importance of code compaction is still high, despite the current large amount of available main memory. This happens because computer systems are designed with an ever increasing gap of speed between processor and main memory, a problem handled by size-constrained intermediate level caches. Failure to fit the program working set in the instruction cache can severely impact performance. Similarly, in shared-memory multiprocessor systems, memory bandwidth reduction is crucial to overall system performance. Nevertheless, thanks to the previous efforts of compacting instruction format in the CISC IA-32 and its variable-length encoding, the x86 processor software base shows good code compaction that leverages the processor with better code organization to address these problems, when compared to RISC code.

Despite good code compaction, old CISC ISAs like the IA-32 [5] suffer from the ISA aging problem: as the interface matures, it is necessary to add new instructions in the already occupied opcode space, and eventually the ISA runs out of space for new opcodes. CISCs handle this problem by increasing instruction length, and even recent RISC ISAs such as ARM [1] may run into this problem as well, in which case they handle opcode space shortage by adding another processor mode – one in which the opcode space is interpreted differently. For instance, modern x86 uses both approaches: it introduces additional instruction prefixes to expand the opcode space and also uses another mode, the IA-32e [5], to interpret instructions differently in the context of 64-bit programs.

When expanding CISC ISAs, increasing instruction length harms code compaction because instructions introduced later have larger sizes and are not necessarily the least used ones. A size-efficient approach for CISC ISA organization would assign the most frequent instructions to the smallest opcode encodings. Nevertheless, in the current approach, as newer extensions begin to be adopted by compilers, larger instructions are used and programs decrease its space efficiency. For example, our experiments detected that the IA-32 ISA AAA instruction (ASCII adjust after addition)[1] is never found in recent software but still occupies a noble area in the opcode space, the category of instructions encoded in 1 byte. In contrast, the `vcmpsd` instruction, introduced with the AVX IA-32 extension [6], needs 5 bytes for opcode and is frequently generated by modern compilers when compiling applications that rely on floating-point computations.

On the other hand, increasing the number of processor operating modes increases the hardware complexity. For example, the ARM Thumb approach requires two different decoders, one for the 32-bit ARM mode and another for the 16-bit Thumb mode. It is also difficult for the compiler to support mode switching, since the use of heuristics [11] is necessary to determine if a program fragment is better represented using the normal instruction set or the reduced instruction set.

Therefore, in general, we can expect that the processor front-end decoder becomes increasingly more complex and more power-demanding as the ISA expands, regardless of whether it uses a CISC or RISC ISA. In order to partially mitigate the problem, Intel microprocessor designers have relinquished the hardware decoding of complex and seldom used instructions and handed them over to a microcode ROM [10]. Even though this simplifies the decoding logic, it only transfers the aforementioned issues to the microcode ROM. In fact, Borin et al. [2] state that an estimation for an Intel low power design concluded that up to 20% of the die area would be used solely by the microcode ROM.

In this paper, we study the IA-32 evolution over time and present data showing how opcode usage of programs released from 1995 to 2012 evolved and how many instructions stopped being used. We suggest that some opcodes are, in fact, never used in recent software, despite using the shortest encodings because of the time when these operations were introduced.

[1]AAA was removed in the IA-32e 64 bit mode.

We argue that these lost opportunities are inefficiencies caused by the immutable x86 ISA and its over-conservative role to preserve backward compatibility down to its first release. We intend to discuss what is the future of the x86 ISA and if it is reasonable to pay the price of hardware and software added complexity for backward compatibility, while showing how the bloat of opcode space can make old ISAs like the x86 significantly less attractive.

This paper is organized as follows. Section II gives a brief summary of the x86 ISA, Section III shows the x86 case study, and Section IV concludes the paper.

II. The x86 instruction set

The x86 instruction set family is the collection of all machine instructions derived from the Intel 8086 family of processors. The architecture evolved to support floating point operations, 32-bit and 64-bit addressing modes and SIMD instructions [5]. The ISA is backwards compatible for all processor families, and recent machines are able to run old programs and libraries assembled more than 30 years ago!

Each instruction in the x86 ISA has a variable-length format, and the basic encoding to represent a single instruction is usually determined via the *opcode* and *prefix* fields. Some instructions further require the use of the *ModR/M* field to be decoded. The layout is given in Figure 1.

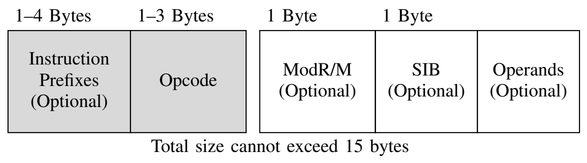


Figure 1: Intel IA-32e and IA-32 instruction formats. Prefix and Opcode fields need to be decoded in order to correctly identify the instructions.

For example, a logic *or* instruction between a 16-bit immediate and a 16-bit value held in memory that is indexed by the register *rcx* may be represented by the assembly text form *orw $12804, (%rcx)*. Table I depicts its equivalent encoding in machine language.

Prefix	Opcode	ModR/M			SIB	Operands
		Mod	Reg/Opc	R/M		
66h	81h	00b	001b	001b	N/A	04h 32h

Table I: x86 instruction encoding example

The ModR/M byte is part of the opcode encoding in this instruction because its subfield Reg/Opc is used as an opcode extension. Hence the instruction has 5 bytes: 3 bytes are used for opcode and prefix and 2-bytes for immediate.

For the purposes of the x86 instruction set analysis, we do not solely use the opcode bytes described in Figure 1 to identify an instruction, but we identify instructions using the combination of the regular opcode plus the necessary bytes to make its identity unique in the representation used throughout

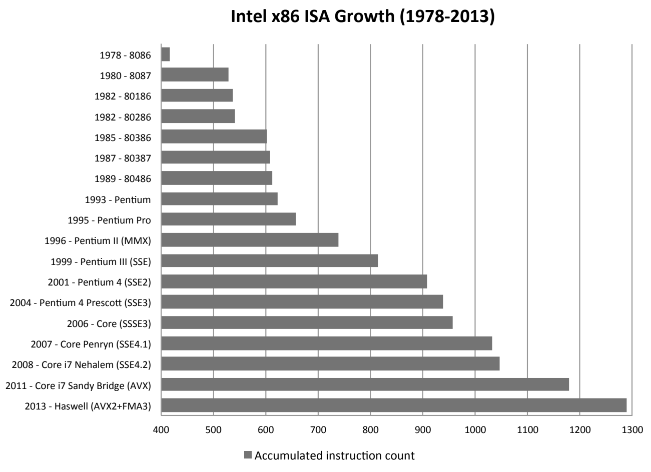


Figure 2: x86 ISA growth over the years

this paper. We chose an identification granularity level similar to a RISC ISA: instructions with different operands using the same main opcode are considered distinct.

The Bochs [9] disassembler library contains an instruction description with this desired granularity level and, for this reason, was adopted in this work. For example `addl_Ed_Id` is the instruction that adds a double word (32-bits) immediate to a general-purpose double word register, while `addl_Ed_Gd` is an instruction that adds the value in a double word general-purpose register to another general-purpose register. In fact the mnemonic `addl` may correspond to 5 different instructions, depending on the operands used. To uniquely identify each one of these instructions, the prefix bytes, the opcode bytes, and, in some cases, the ModR/M byte must be used.

To easily refer to the necessary bits required to uniquely identify our definition of x86 instructions and to avoid confusion with the x86 *Opcode* byte, these bits will be henceforth referred as the `operation code`. For example when it is said that an operation code has 5 bytes, it means that the minimum number of bytes necessary to uniquely identify the instruction and its particular operand addressing mode is a 5 byte combination of prefix, opcode and, if necessary, a ModR/M byte.

Figure 2 presents an overview of the increasing number of x86 instructions over the years. The 16-bit 8086 processor, released in 1978 and the first in the family, had little more than 400 instructions – a number that has grown twice by 1999 with the release of Pentium III and the introduction of the SSE multimedia processing instructions [8]. By today, x86 has about 1200 instructions and will grow to 1300 with the release of the new Haswell architecture [7]. The overall introduction of multimedia instructions from MMX to AVX[2] has about the same number of x86 instructions accumulated from 8086 up to Pentium Pro and therefore has the complexity and size of

[2]This instruction count does not include all the re-encoded versions of all SSE instructions which are present in AVX, but only instructions with new functionality.

Release	Operating System	Additional Software
1996-1997	Slackware Linux 3	Netscape 4.0.1, StarOffice 3.1
2007-2008	Ubuntu 8.10	Firefox 3.0.3, OpenOffice 2.4
2011-2012	Ubuntu 12.04	Firefox 11, LibreOffice 3.5
1995-1996	Windows 95	I.E. 3, Office 95
2001-2004	Windows XP SP2	I.E. 6, Office 2003
2010-2012	Windows 7 SP1	I.E. 8, Office 2010

Table II: Software systems analyzed, each with its own virtual machine

an entire new instruction set by itself.

To expand the instruction set to support new instructions without breaking backward compatibility, vendors create new operation codes to hold new functionality. The growth of the x86 ISA is followed by a subsequent increase in the average instruction size. Figure 3 shows that for each Intel x86 ISA variant released over the years, the average operation code size increases – operands are ignored. For example, 308 new instructions were introduced between MMX and SSE 4.2 as shown by Table 2, and to represent this additional amount of instructions, more than 1 operation code byte is needed. Therefore the average number of operation code bytes has grown from 2.7 to 4 bytes (Figure 3).

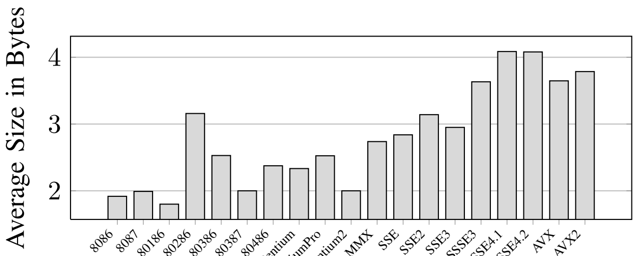


Figure 3: Average instruction operation code size for each x86 feature

III. Legacy Code Analysis

This section presents a study of instructions that stopped being used over time. We measured and generated all of the x86 static analysis of this section and for Section II using two different disassembler tools: Agner's *object file converter* [3] tool and the disassembler library present in `Bochs` virtual machine [9]. Both tools are open source under the GPL version 2 license [4] and are interchangeably used as libraries to a higher level tool designed for these measurements.

We organized a number of virtual machines containing a complete 32-bit x86 software environment of a specific year. Table II shows the age of the software systems analyzed and their software contents. For example, our first Windows-based environment uses the Windows 95 operating system, the Internet Explorer 3 browser, and the Office 95 productivity suite to represent how x86 software from 1995 to 1996 used the IA-32 instruction set. To improve coverage, we include common software used by people in home or office. We studied static frequencies of x86 instructions of different types and their evolution in time both in Windows and Linux desktops.

The static analysis uses a crawler that analyzes the entire virtual machine disk for executable files. When x86 instructions are found, their type is catalogued using a disassembler library and the frequency count for the types found are updated. Both per-program and per-virtual machines static frequency histograms were extracted to show how different x86 operation codes are being used in programs over time. The static analysis is broad because single instructions are catalogued even though they may never execute.

A. Total Opcodes Recorded in All Disks

The total number of unused operation codes in all disks is 505. This means that considering all the 1646 x86 prefix plus operation code combinations, about 30% of them were never found in any virtual machine disk that we scanned. From this count, we excluded 149 combinations that use the `0x48` prefix, which requires the 64-bit mode, because our analysis focus on 32-bits virtual machines.

Type	Number of unused operation codes			
	3 Bytes	4 Bytes	5 Bytes	6 Bytes
AVX	3	61	5	0
SSE	74	238	7	1
Other	40	76	0	0
Total	117	375	12	1

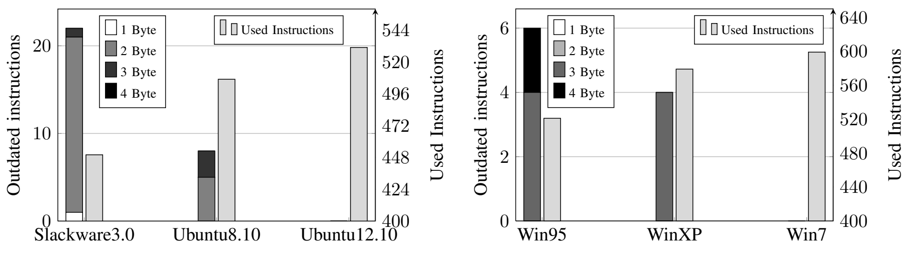
Table III: Number of unused operation codes by size. There were no unused 1 and 2 bytes operation codes.

Table III shows the number of unused instruction operation codes by size. This table also shows the number of these instructions that belongs to vector extensions, because albeit unused, there is a high chance these operation codes may be used in the future – they are still in adoption. SSE category includes all Intel SSE extensions and AMD SIMD extensions. AVX considers AVX and AVX2 extensions. The others include the MMX extension.

B. Aging Effect

The aging analysis shows whether an instruction appears in the disk or not with respect to a given year. In this analysis, vector extensions were not considered because they belong to a large category of instructions that are still in adoption, and we need to present a separated analysis for them. We also skipped privileged and 64-bit only instructions.

Figure 4a shows a two y-axis graph whose *Used Instructions* ordinate axis depicts the number of different operation codes used in Linux systems through time. As expected, the number



(a) Linux. (b) Windows.

Figure 4: Static instruction usage patterns for Windows and Linux systems.

of used operation codes increases because software is absorbing new instructions. Examples of instructions that were not used before but started to appear in the disk at this time scale include: `vmclear` and `vmptrld` virtualization instructions, `xadd` exchange and add instructions, which had its usage increased thanks to the rise of multiprocessing systems, `xchg`, for the same reason, and several `cmov`, conditional move, variants, which were first introduced in Pentium Pro.

When the crawler sees an operation code in a given year, for example, 2004, and no longer can find it in any other subsequent year (2007 up to 2012), we mark this operation code as unused or outdated. The other y-axis of Figure 4a shows the number of outdated operation codes in time and also their size, and Figure 4b shows the same study to Windows systems. In Figure 4a, the Slackware bar shows that 20 2-byte operation codes were last seen in 1996. This means that future software releases stopped using these instructions. Not surprisingly, some outdated operation codes discovered by our static analysis were also deprecated in Intel IA-32e 64-bit mode, including `les`, load far pointer using ES, `push` and `pop` using ES or CS. They were all outdated starting with Ubuntu 8.

As the chart shows, Slackware Linux was the last release to use an 1-byte operation code instruction, which is the `les` instruction. In a scenario where the operation code space could be reused, it is specially important to pick a 1-byte instruction, because we may use this opcode as an escape code to encode 256 new 2-byte instructions. It is also possible to use escape codes to encode 3-byte or bigger instructions, but the benefits for code compaction are reduced.

C. Vector Extensions

The IA-32 ISA recently had almost all extensions focused in adding vector instructions that explore data parallelism. The first extensions to address floating-point calculations were 8087 and 80387, but their operand addressing is stack-based, a rather old and inconvenient addressing method for modern compilers. Newer vector instructions, starting with the MMX extension, have register operands, which allow the compiler to easily control register usage with established register allocation algorithms, and are able to perform multiple floating-point calculations on the same cycle. For these reasons, vector instructions naturally supersede the old x87 instructions.

Our analysis indicates that the older IA-x87 floating-point extensions are still widely present in modern software, showing no signs that it can be outdated. It is possible to see how newer extensions like SSE and SSE2 began to be adopted. It is possible to conclude that IA-x87 extensions did not stop being used over time, albeit being functionally superseded by newer extensions. In fact, it has increased its participation on the total instruction count over time in Windows systems. For this reason, the ISA is forced to be redundant: it is possible to add two floating point data using either IA-x87 or vector extensions, a suboptimal operation code organization.

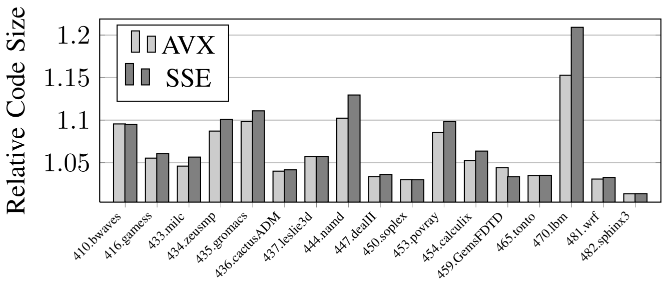


Figure 5: Percentage of the code size growth of SPEC floating point programs when compiled with SSE and AVX relative to a IA-x87 baseline.

Figure 5 shows that for 7 SPEC CPU2006 floating-point programs, the usage of vector extensions yields larger executable binaries than the x87 ones. In this analysis, we compiled the programs using *gcc* version *4.7* with the *-O2* optimization flag and the *-march=corei7-avx* architecture tuning. To generate x87 instructions, we used the *-mfpmath=387* option, while to generate the SSE ones, we used the *-mfpmath=sse* option, and finally we added the *-mavx* option when testing the AVX encoding for SSE instructions. No vectorization optimizations were used, but the compiler was tuned to use AVX for floating point arithmetic.

Notice that the AVX introduced not only a new set of 256-bit vector instructions, but also new encodings for all previous SSE instructions. A program compiled using AVX will have SSE instructions encoded using the new operation codes provided by AVX.

Figure 3 points out that vector extension instructions may have 1 to 2 extra bytes for operation code in comparison with the old IA-x87 extension. The increased code size may explain why compilers still generate old IA-x87 instructions in favor of newer SSE or AVX instructions when there is no data parallelism. It is also known that IA-x87 is still nowadays the default floating point instruction choice for widely used compilers, such as gcc.

This suggests that compilers may explore old encodings because they have better compaction rates. On the other hand, many other `operation codes` are being deprecated, leaving, in terms of compaction rate, valuable encodings unused. Furthermore, it is also important to note that merely removing unused instructions and leaving their `operation codes` unused will be enough to be profitable, since it reduces hardware complexity.

IV. Conclusions

In this paper, we analyze how some instructions from x86, an old but very popular CISC ISA, stopped being used with time. The experimental analysis used a file system crawler to search several disk images of systems from different years for programs to disassemble. In Linux systems, the total number of instructions that disappeared in the time frame from the 1996 Slackware to the last Ubuntu release from 2012 was 30, and for Windows this number was 10, identifying many opcodes that are no longer used.

Despite the opcode space abuse in x86 due to its age and the constant introduction of newer extensions, there is a significant amount of instructions becoming obsolete. It should be profitable to redesign an x86 without old instructions because it would have a simpler hardware. If the old instructions encodings are left unused, they could be further modified to bear different operations, assigning the most frequently used ones to the smallest encodings and therefore foster modern software efficiency. Even though it is not clear whether this backward compatibility disruption in favor of ISA evolution would negatively impact systems that depend on legacy code, in practice, our Windows and Linux-based benchmarks show that many instructions were definitely retired by the software industry.

References

[1] ARM Limited. ARMv7-M Architecture Reference Manual. July 2012.

[2] E. Borin, M. Breternitz, Y. Wu, and G. Araujo. Clustering-based microcode compression. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 189 –196, oct. 2006.

[3] A. Fog. *Instructions for objconv*, 2011. Version 2.11.

[4] Free Software Foundation. General Public License. http://www.gnu.org/licenses/gpl.html. Accessed July 2012.

[5] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, Volume 2: Instruction Set Reference edition.

[6] Intel Corporation. Intel AVX: NewFrontiers in Performance Improvements and Energy Efficiency, 2008. White paper.

[7] Intel Corporation. Haswell new instruction descriptions now available. http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available, 2012. Accessed July 2012.

[8] A. Kilmovitski. Using SSE and SSE2: Misconceptions and reality. Intel Developer Update Magazine, 2001.

[9] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996(29es), Sept. 1996.

[10] J. Shen. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill Companies,Incorporated, 2004.

[11] A. Shrivastava, P. Biswas, A. Halambi, N. Dutt, and A. Nicolau. Compilation framework for code size reduction using reduced bit-width ISAs (rISAs). *ACM TODAES*, 11(1):123–146, Jan. 2006.