Enumerating x86-64 – It's Not as Easy as Counting

William Mahoney
University of Nebraska at Omaha
PKI 281-E
6001 Dodge Street
Omaha, Nebraska 68182
wmahoney@unomaha.edu

J. Todd McDonald
University of South Alabama
1121 Shelby Hall
150 Jaguar Drive
Mobile, AL 36688
jtmcdonald@southalabama.edu

ABSTRACT

In our work for software watermarking, we have been examining the possibility of executable steganography, hiding intel/AMD x86-64 instructions within the operands of other instructions. Early thoughts about this concept revolved around creating a database of sorts to reflect which x86-64 instructions had large enough operand fields to hold the hidden payload. It was assumed that this database would be easily constructed, but it turns out to be a surprisingly difficult endeavor. Even the question of "how many are there?" is challenging to answer. Different CPUs support different instruction sets, different instruction decoders or reverse assemblers give different results for the same combination of bytes, and even the number of distinct mnemonics for instructions is blurry. In the process of attempting to construct the x86-64 database we encountered several stumbling blocks along the way and we report on the stumbling blocks here. This white paper is not a traditional research paper, with background, other relevant prior work. Rather, it describes our attempts to answer what we thought was a fairly simple question: for various numbers of bytes, just how many legal x86-64 instructions exist?

CCS Concepts

• Computer systems organization → Architectures → Serial architectures → Complex instruction set computing

Keywords

Instruction set, instruction decoding, mnemonics

1. INTRODUCTION

In the process of working on our executable steganography efforts [1] we desired to construct a database of x86-64 instructions and what we called their "cover numbers". Our intent is to hide short executable instructions inside the operands of longer x86-64 instructions in such a way that there would be a hidden payload or watermark inside the code, and that this watermark not be visible normally by reverse engineering tools. The "cover number" of an instruction was defined as the number of bytes that the instruction is capable of hiding. For instance, an x86-64 instruction with a 64-bit operand would be capable of secretly encoding eight bytes in the operand, so it's cover number would be eight. Our early thoughts for the project included some kind of searchable database, where, if I need an instruction with a cover number of at least three, the database will tell me all potential instructions.

This research is supported by the National Science Foundation under the Secure and Trusted Computing (SaTC) grants CNS-1811560 and 1811578. The project is a collaborative effort between the University of Nebraska at Omaha (UNO) and the University of South Alabama (USA).

Later it was determined that this database of cover numbers is not as useful as a database of which instructions are available for various numbers of bytes; rather than looking up an instruction to see what it might be capable of hiding, the better approach is to determine what operations require only one byte, or two, or three, … In this way the author of the code which will be hidden can select operations based on the number of opcode bytes.

Although the intel/AMD 64-bit instruction set is large, it would seem that there would be a relatively simple / programmatically easy way to generate all of the instructions and from there, or in the process, to determine the number of bytes for each. Come to find out this task is surprisingly difficult. Even as recently as 2016 one could assume that "a formal semantics of the current 64-bit x86 ISA … would be readily available, but one would be wrong. In fact, the only published description of the x86-64 ISA is the Intel manual with over 3,800 pages written in an ad-hoc combination of English and pseudo-code" [2].

So of course, when the assumption is that something should be simple, often it is not, and this is only discovered after "jumping in head first". A result of this "jumping in" is reported here. Our paper is less of a research tome and more of a running commentary on how we approached the problem and what results we had (or did not have!) along the way.

Section two presents terminology and states the problem in more detail. Section three describes our approach to exhaustively searching a list of x86 instructions. Are the results correct? Surprisingly this is not an easy question to answer. The reasons are given in section four, as well as some thoughts about future changes which could be made to shed some illumination on the answer.

2. THE PROBLEM

In a nutshell our question is: *how many valid byte combinations correspond to legal x86-64 architecture instructions of a certain length?* Can they be enumerated, and if so how? Specifically, due to our steganography work we are interested in instructions whose size is six or fewer bytes.

2.1 Considerations

On the surface the issue of constructing our valid instruction list seems an easy problem. But consider:

- The number of potential x86-64 instructions is huge, as the hardware limit is the number of bytes that the CPU is willing to fetch for one instruction. On x86-64 this is 15 bytes [3] (pp 208) and as a result there are 2^{15*8} potential instructions.
- Certain prefix bytes can be added in advance of the instruction, some of them causing extended behavior and some of them having no effect whatsoever. As a result, a simple instruction such as an addition of two registers can

have many variations in the byte encoding and yet all perform an identical function.

- Some CPUs include additional features such as Multimedia Extensions (MMX) and some do not. The number of valid instructions is thus CPU model dependent. When we say "legal instructions" this needs to be accommodated.

To clarify, when we refer to "instructions" we are describing all possible forms of the instruction. For example, "MOV" is one operation mnemonic but there are many potential encodings, depending on the desired source, destination, and size of the operands. We thus need to be clear that when we use the term "instructions" we are describing byte sequences and not mnemonics. When we refer to "MOV instructions" or just "instructions" we are referring to all possible "MOV" operations, or all operations in general, respectively.

In the above issues list, the instruction prefixes in particular make this a thorny issue. To explain why the instruction set is so complex requires a bit of x86 history and an overview of the resulting layout of instructions.

2.2 x86 History

Why is the x86-64 instruction set so hard to describe? For historical reasons. Intel (and AMD) have long attempted to maintain backwards compatibility, stretching back as far as 1978. In the process, the various warts and blemishes continue to be replicated over the years. The original 8086 and 8088 16-bit CPUs were follow-ons to the popular 8-bit CPUs developed by intel, the 8080 and the almost identical 8085 [4]. When the 32-bit architecture was created the attempt was to make it compatible with the 16-bit 8086, which was in turn mostly designed to be a better version of the original 8080. This compatibility has caused aspects of the modern day x86-64 to reflect items from 40 years ago, including, for instance, the ability to access bits 8-15 of certain general-purpose registers.

What follows is a very abridged version of the history of the x86 CPU. The descriptions include direct excerpts from [5] (pp 2.1-2.6) except where italicized. Briefly:

1978 – The 8086 has 16-bit registers and a 16-bit external data bus, with 20-bit addressing giving a 1-MByte address space. The 8086/8088 introduced segmentation to the IA-32 architecture.

1982 – The Intel 286 processor introduced protected mode operation into the IA-32 architecture. Protected mode uses the segment register content as selectors or pointers into descriptor tables.

1985 – The Intel386 processor was the first 32-bit processor in the IA-32 architecture family. It introduced 32-bit registers for use both to hold operands and for addressing. The lower half of each 32-bit Intel386 register retains the properties of the 16-bit registers of earlier generations, permitting backward compatibility.

1989 – The Intel486™ processor added more parallel execution capability by expanding the Intel386 processor's instruction decode and execution units into five pipelined stages.

1993 – The introduction of the Intel Pentium processor added a second execution pipeline … A subsequent stepping of the Pentium family introduced Intel MMX technology … uses the single-instruction, multiple-data (SIMD) execution model to perform parallel computations on packed integer data contained in 64-bit registers.

1995-1999 – The P6 family of processors … includes the Pentium Pro, Pentium II and Pentium II Xeon, Celeron, Pentium III and Pentium III Xeon. *(Most of the changes in this period are internal architecture enhancements, but the P-III introduces the SSE instructions.)*

2000-2007 – The Intel Pentium 4 processor introduced Streaming SIMD Extensions 2 (SSE2) … The Intel Pentium 4 processor 3.40 GHz, supporting Hyper-Threading Technology introduced Streaming SIMD Extensions 3 (SSE3). The 64-bit Intel Xeon processor 3.60 GHz … was used to introduce Intel 64 architecture. The Intel Xeon processor 5200, 5400, and 7400 series … improves the performance of Intel® Advanced Digital Media Boost and SSE4.

2008 – The first generation of Intel Atom processors … Support for instruction set extensions up to and including Supplemental Streaming SIMD Extensions 3 (SSSE3). The Intel Core i7 processor 900 series support for SSE4.2 and SSE4.1 instruction sets.

2010 – Intel Core processor family spans Intel Core i7, i5 and i3 processors … Range of instruction set support up to AESNI, PCLMULQDQ, SSE4.2 and SSE4.1.

2013 – Intel Atom Processor C2xxx, E3xxx, S1xxx series … supports instruction set extensions up to and including SSE4.2, AESNI, and PCLMULQDQ.

Today – Beginning with the Pentium II and Pentium with Intel MMX technology processor families, six extensions have been introduced into the Intel 64 and IA-32 architectures to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE extensions, SSE2 extensions, SSE3 extensions, Supplemental Streaming SIMD Extensions 3, and SSE4. Each of these extensions provides a group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements.

The point to take away from this historical perspective is that starting in 1978, intel has attempted to maintain backwards compatibility from generation to generation starting from the 8086 CPU, which in turn was designed as an upgrade from the 8080 and 80805 8-bit CPUs. Along the way, we see the introduction of 32- and then 64-bit instructions as well as six SIMD instruction sets, all of which need to be "bolted on" to the set of operation codes from 1978. To put this in perspective, the next section provides an indication of the current state of x86-64 instruction mnemonics.

2.3 What Do the Doc's Say?

In general, our problem statement is to list valid sequences of bytes and the instructions that each sequence corresponds to. But to get a general idea of the magnitude of the problem, it is worth digressing for a moment to discuss the number of valid mnemonics for the instructions instead of the byte sequences. This will serve to demonstrate that the number of sequences is dependent on the CPU model and which features are supported.

As a starting point, Table 1 corresponds to the number of instruction mnemonics, by category. For each table row, the presence of a set of numbers corresponds to the different sub-groups of instructions within the major category. For example, the "General Purpose" instructions include 32 "Data Transfer", 14 "Binary Arithmetic", and so on. The total in the group is displayed. All instruction counts in this table are for intel and are from [5] (pp 5.1-5.36).

In addition to those in Table 1, CPUs after about 2010 include the Advanced Vector Extensions, with about 243 instructions. We say "about" because quite a number of these are the same as previous SIMD instructions but with new 256-bit equivalents. The encoding of an instruction prefix (of course), VEX, uses either two or three bytes prior to the operation code. This prefix provides a compressed representation of the REX prefix, as well as various other operation prefixes, and expands the addressing mode, register number, and operand size and width.

Newer CPUs may also include Fused Multiply Add: "FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions. FMA extensions also provide 60 128-bit floating-point instructions to process 128-bit vector and scalar data. The arithmetic operations cover fused multiply-add, fused multiply-subtract, signed-reversed multiply on fused multiply-add and multiply-subtract." [5] (pp 5.3-5.35).

Table 1. x86-64 Mnemonics by name

Group	Mnemonics
General Purpose 32, 14, 6, 4, 9, 25, 31, 18[1], 8, 2, 11, 5, 10, 5, 2, 15	197
X87 FPU 17, 26, 14, 8, 7, 20	92
X87 FPU and SIMD State Management	2
MMX 2, 9, 17, 6, 4, 8, 1	47
SSE 8, 18, 4, 4, 3, 6, 2, 12, 5	62
SSE2 6, 14, 4, 4, 3, 13, 3, 14, 8	69
SSE3 1, 1, 2, 4, 3, 2	13
SSSE3 12, 6, 2, 2, 2, 6, 2	32
SSE4.1 2, 2, 1, 6, 8, 4, 7, 12, 1, 1, 1, 1, 1	47
SSE4.2	7[2]
AESNI and PCLMULQDQ	7
16-bit Floating Point Conversion	2
Transactional Synchronization	6
SHA Extensions	7
Advanced Vector Extensions 512 (AVX 512) 64, 18, 17, 3, 13, 6, 8	129
System	46[3]
64-bit Mode Instructions	10
Virtual Machine Support	13
Safer Mode Extensions (SMX)	8
Memory Protection Extensions	8
Security Guard Extensions	18

For AVX2, most of the SSE/SSE2/SSE3/SSSE3/SSE4 instructions are supported for 256-bit operands in addition to the 128-bit operands. This is handled by the VEX prefix encoding. An additional 29 "New Primitive AVX2 Instructions" are available as well [6].

The EVEX prefix is a four-byte instruction prefix which always starts with 0x62. The second byte (byte one) has some bit settings in common with REX. The two remaining bytes specify a source operand, the vector length (e.g. 256-bits), operand size prefixes which replace the usual 0x66 prefix, additional bits to expand the register number to 32, and other settings [7].

Based on the breakout in Table 1, culled from the intel instruction set reference, we add up 822 instruction mnemonics. But Heule et. al. states that the current x86-64 design "contains 981 unique mnemonics and a total of 3,684 instruction variants" [2]. However they do not specify which features are included in their count.

2.4 General Instruction Layout

There are numerous references, both online and from intel and AMD which describe the x86-32 and x86-64 instruction formats, and there is no need to duplicate all of this information here. However, an overview of one x86 feature is worth exploring before we discuss our approaches. This feature is the instruction prefix, which includes "legacy" prefixes – a holdover from previous instruction sets – and current prefixes which include REX, VEX, and others needed for advanced features.

2.4.1 Legacy Prefixes

The instruction decoding as outlined in Figure 1 (at end of document) indicates the presence of so-called legacy prefixes. These are holdovers from the early x86 days as described previously and include four types:

Group 1: LOCK (0xF0) was used for atomic memory accesses, REPNE (0xF2) and REPE (0xF3) are prefixes used to repeat string operations. Under certain CPUs, 0xF2 is used instead for the BOUND prefix. In theory, the repeat prefixes are only allowed for MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS.

Group 2: Segment overrides (0x2E, 0x36, 0x3E, 0x26, 0x64, 0x65) and branch hints (0x2E, 0x3E). These prefixes are ignored for x86-64 but permissible for backwards compatibility.

Group 3: Operand size override (0x66). This allows x86-64 to use a different data type than is normally accessed by the operation. The intel documentation also includes this ominous warning: "Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a three-byte sequence of primary opcode bytes may use 66H as a mandatory prefix to express distinct functionality".

Group 4: Address size override (0x67). Similarly, this indicates that the address size is other than the default for this instruction.

[1] The string instruction mnemonics listed include "REP", "REPE/REPZ", and "REPNE/REPNZ" which are not instruction mnemonics but operation code prefixes.

[2] The documentation refers to "seven new instructions" but lists five in the description.

[3] There again is some overlap here and even the intel documentation is sometimes not correct. For example, "MOV" is listed as a system instruction twice, since one might be moving data into a control register or into a debug register. "LOCK" is listed as an instruction but is not an instruction but an instruction prefix.

2.4.2 Current Prefixes

When the CPU is executing in 64-bit mode, the size overrides 0x66 and 0x67 behave in a similar manner, selecting something other than the default operand or address size. And again, repeated size prefixes do nothing other than waste space. For example, repeating override 0x66 many times just serves to use additional bytes in the instruction.

The more important prefix for x86-64 is REX – the register extension byte. This byte has values from 0x40 through 0x4F and the corresponding instructions in 32-bit mode are no longer available; these were the "INC" and "DEC" instructions, and different instructions with the same effect are available in 64-bit mode. The REX prefix adds an additional bit to select registers, so that 16 are available instead of eight, it contains a flag to alter the operand length which works in conjunction with the size overrides, and it allows for additional addressing modes to be selected.

Additional prefixes for VEX and XOP instructions are also possible but were not considered in our study. Their use generally makes instructions longer than what we wish to consider.

2.4.3 The Impact

In our case both the legacy prefixes and the current prefixes have a direct impact on our quest to list all instructions.

First, since the legacy prefixes are ignored in 64-bit mode, one can simply add them on the front of the instruction and it makes no difference[4]. Also, for an instruction using more than one legacy prefix, the order is unimportant, but an interesting side effect is the expected behavior when more than one legacy prefix of the same group is used. The documentation from intel does not prohibit this, it simply says: "For each instruction, it is only useful to include up to one prefix code from each of the four groups" [3] (pp 2.1). "Only useful" is not the same as "it will crash". Different CPU models operating in 32-bit mode handle this differently, with some honoring the first prefix and ignoring subsequent prefixes, and others honoring only the last prefix encountered [7]. In 64-bit mode, since they are ignored, the order is immaterial.

For our research question this causes an issue, since an instruction requiring, say, five bytes actually has a 5-byte version, several 6-byte versions that contain one byte of legacy prefix, a 7-byte version with two bytes from the possibilities and in either order, and so on. Since multiple prefixes are ignored, we can fill the instruction to the 15-byte limit. This (non)restriction on prefixes allows a programmer to create any longer length just by adding superfluous prefixes in 64-bit mode. While detrimental for us, since for steganography we prefer short instructions, this can actually be useful in aligning code on boundaries, as for example executing one instruction that does nothing is faster than executing many instructions that do nothing [8].

In terms of the REX prefix, there are at least two issues. First, "the use of multiple REX prefixes is undefined, although processors seem to use only the last REX prefix" [7]. If one knows the expected result on this particular CPU model, leading with (or trailing with) the appropriate REX instruction is possible. Or to simplify things, a programmer can make the first of many REX bytes and the last of many REX bytes both have the correct value for the instruction. Again, creating longer instructions that contain this prefix is simple. Secondly, another quirk worth mentioning is that this prefix is required to be immediately preceding the opcode. According to intel, "Other placements are ignored" [3] (pp 2.8). Inserting a REX prefix before a legacy prefix wastes a byte and has no impact.

[4] One legacy prefix, "LOCK" can only be combined with certain instructions. See [6].

3. ENUMERATING WITH EXHAUSTIVE SEARCH

To create our data files, we utilized a C language library called Udis86 [9]. This library contains functionality to decode instructions and present a string representation of the original mnemonic. In the process it invokes a callback function supplied by the user to present the next byte in the decoding. We modified this library in the following ways:

- Internally to the instruction decoding we differentiated between bytes used as a part of the operation code versus bytes used as operands. Table 2 below will describe this in greater detail.
- The number of allowed prefix bytes can be limited programmatically when the library is initialized. This can be used to limit the number of legacy prefix bytes, for example. The library was modified so that if the number is exceeded for a certain type of prefix the instruction is considered invalid.
- We added a string representation of the incoming bytes which is constructed as they are used by the decoder, and made this string available for our exhaustive search method outlined below.

Once the library is modified to track opcode bytes versus operand bytes, it is possible to construct instruction strings which indicate which is byte is which function. We use the following characters to indicate operands:

Table 2. Byte Indicators for Operands

I	Position of a 1-byte immediate value
J	Position of a 2-byte immediate value
K	4-byte immediate value
L	8-byte immediate value
W	Position of a 1-byte relative offset
X	Position of a 2-byte relative offset
Y	4-byte relative offset
Z	8-byte relative offset

For example, if our supplied callback function provides, a byte at a time, 0x00, 0x5C, 0x28, 0x00 the result is:

```
005C28WW        add [rax+rbp+0xWW], bl
```

where the "WW" indicates a one-byte relative offset in the instruction. The modifications we have made to the decode library provide us a means of knowing which part of the instruction (and original hex) are operands. Similarly:

```
664105JJJJ      add ax, 0xJJJJ
```

includes a size override (0x66 shifts to a 16-bit addition), and the instruction has a 16-bit constant built into the operation. Third,

```
4067C2JJJJ      a32 ret 0xJJJJ
```

is an example of a technically incorrect decoding; the 0x40 REX prefix is before the 0x67 address size override, and the latter serves no purpose since there is no address in the instruction. One last example is:

```
0F0F4424WW0C     pi2fw mm0, [rsp+0xWW]
```

This is a 3Dnow instruction and we include this so that the reader will notice one fact: generally the immediate operands or addresses are at the end of the instruction, but when the 3Dnow operations were added to the mix by AMD they opted to include a byte of operation code after the operand.

Our approach, since we are concerned with shorter instructions, is to perform a limited exhaustive search based on the number of bytes desired. Consider the case where we wish to create a list of all single bye instructions. Ask to decode the next instruction and when our callback is invoked, provide the decode library with 0x00. If it invokes our callback function again the implication is that 0x00 is not a valid one-byte instruction but might be a valid prefix for something longer. But, as we are concerned only with one-byte instructions, we inform the decoder that we have encountered the end of the file, reset the library, and start the process again with 0x01. When we get to 0x50 the decoder will actually yield an instruction (`push rax`), the first of the one-byte instructions, and this is added to the file.

In this way generating all one-byte instructions is simple and we can expand the method to do all instructions that are two bytes and so on. However we need to be somewhat intelligent in order to cut down on the workload. We say our approach is "limited exhaustive search" due to instructions such as one mentioned above:

```
0F0F4424WW0C     pi2fw mm0, [rsp+0xWW]
```

The next instructions to test after this are:

```
0F0F4424000D     pi2fd mm0, [rsp+0xWW]
0F0F4424000E     (not valid)
…
0F0F442400FF     (not valid)
0F0F4425 0000    (not valid)
```

In other words, we need to intelligently skip over that portion of the byte stream that contains any of the immediate constants or addresses which are irrelevant. In the example we are skipping ahead 256 attempts, but clearly this is increasingly important as the constants get larger. For example:

```
05KKKKKKKK       add eax, 0xKKKKKKKK
```

will send us down a blind alley of 2^{32} useless attempts, all of which decode to the same thing.

The search method uses an intelligent addition which looks at the previous string representation that was attempted, skips over bytes that are part of the constants outlined in Table 2, and increments "in the right place" for the next attempt. It also properly carries into the next (previous) byte.

Finally, note that in the process of producing all instructions which are – for example – five bytes long, we will encounter all of the four-byte instructions along the way, but discard them as too short. A command line switch can be used to include all shorter byte sequences at the same time. We have opted to keep the files separate since for longer instructions the results are large.

4. BUT IS IT CORRECT?

Different tools decode instructions differently, which does not help our efforts. The online disassemblers in particular are each slightly different. We tested several byte strings against different tools to provide a flavor if the difficulty of "getting a straight answer". Specifically we used the Online Disassembler [10], Shell-Storm [11], Udis86 library [9] in its original form (without our modifications), IdaPro [12], and the Linux tool "objdump" with the intel syntax. The results are in Table 4 (at the end of the document).

Several quirks are apparent.

- Different tools interpret superfluous size overrides differently. For example Udis86 will add "a32" or "o16" to prefixes that are not actually used, whereas Shell-Storm does not care, and IdaPro assumes it is an extra byte unrelated to the instruction.
- But on the other hand Udis86 will quietly and correctly allow but ignore a REX prefix that is not immediately in front of the opcode, but without adding anything to the string that is generated.
- Udis86 does not seem to always properly synchronize after what it considers an invalid opcode. The example in Table 4 (at end) starting with 0x2E, 0x3E shows this. The Online Disassembler has the correct interpretation.
- Objdump includes the mysterious nonexistent register RIZ [8].

So after all of this, what do we have? We created several files in the process, one for each instruction length from one through six. In the case of five- and six-byte instructions the files are split (e.g. a file of the five-byte instructions starting with 00-7F, …) so that they are not too large. Each contains what the Udis86 library considers as a valid decoded x86 instruction for the corresponding bytes, in the format above. For example:

```
6641014CA0WW add [r8+0xWW], cx
```

where the "WW" indicates the one-byte relative offset included in the operation code as described in Table 2. At the same time, this instruction is also included:

```
4166014CA0WW add [rax+0xWW], cx
```

This is one of the cases described previously, where the REX prefix is "too soon". However, the intel documentation indicates that this is a valid instruction but with the prefix ignored. Testing on an intel Core i5 verifies that this is correct, and as a result we have left these instructions in the list as well.

And our final results? Using the Udis86 library, Table 3 shows the numbers for each instruction length.

Table 3. Instruction Counts for Opcode Lengths

Bytes	Instruction Count
1	64
2	7,535
3	243,697
4	4,213,695
5	67,964,490
6	923,392,709

These files are available for review should the reader wish to contact us. In terms of potential future work, we are considering, our limited exhaustive search with Udis86 seems to work, but the occasional duplications are worrisome. On the one hand, an extraneous prefix is, according to the documentation, ignored. But in terms of creating all instructions, should we consider them valid or invalid? Also, a different approach might be to use the newer XED project [13] which claims to provide "a more detailed internal format describing all resources read and written", which may help to illuminate some of the dark corners of x86-64.

5. REFERENCES

[1] W. Mahoney, J. Franco, G. Hoff and J. T. McDonald, "Leave it to Weaver," in *SSPREW*, San Juan, Puerto Rico, 2018.

[2] S. Heule, E. Schkufza, R. Sharma and A. Aiken, "Stratified Synthesis: Automatically Learning the x86-64 Instruction Set," in *Programming Language Design and Implementation (PLDI)*, Santa Barbara, 2016.

[3] intel, Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z, intel, 2016.

[4] intel, "intel 8080 Microcomputer Systems User's Manual," September 1975. [Online]. Available: http://www.nj7p.info/Manuals/PDFs/Intel/9800153B.pdf. [Accessed 12 December 2018].

[5] intel, Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, intel, 2016.

[6] P. Gepner, "Using AVX2 Instruction Set to Increase Performance of High Performance Computing Code," *Computing and Informatics,* vol. 36, pp. 1001-1018, 2017.

[7] OSDev.org, "X86-64 Instruction Encoding," [Online]. Available: https://wiki.osdev.org/X86-64_Instruction_Encoding. [Accessed 13 December 2918].

[8] Answiz, "What is register %eiz?," [Online]. Available: https://www.answiz.com/questions/18663/what-is-register-eiz. [Accessed 13 December 2018].

[9] Udis86, [Online]. Available: http://udis86.sourceforge.net. [Accessed 1 December 2018].

[10] "Online Disassembler," [Online]. Available: https://onlinedisassembler.com/static/home/index.html. [Accessed 12 December 2018].

[11] "Online Assembler and Disassembler," [Online]. Available: http://shell-storm.org/online/Online-Assembler-and-Disassembler/. [Accessed 12 December 2018].

[12] "IDA: About," [Online]. Available: https://www.hex-rays.com/products/ida/. [Accessed 12 December 2018].

[13] M. Charney, "X86 Encoder Decoder," [Online]. Available: https://intelxed.github.io/ref-manual/index.html. [Accessed 3 January 2019].

Table 4. Comparing Different Disassembly Tools

Hex Instruction	Tool	Result
66 67 05 11 11	Online Disassembler	a32 add ax, 0x1111
	Shell-Storm	add ax, 0x1111
	udis86	a32 add ax, 0x1111
	IdaPro	db 67h add ax, 1111h
	objdump -M intel	addr32 add ax, 0x1111
40 66 05 11 11	Online Disassembler	rex add ax, 0x1111
	Shell-Storm	add ax, 0x1111
	udis86	add ax, 0x1111
	IdaPro	add ax, 1111h
	objdump -M intel	rex add ax, 0x1111
2E 3E 26 64 65 36 F0 F3 66 05 11 11	Online Disassembler	cs ds es fs gs ss lock repz add ax, 0x1111
	Shell-Storm	add ax, 0x1111
	udis86	Invalid adc [rcx], edx
	IdaPro	db 2Eh, 3Eh, 26h, 64h, 65h, 36h lock rep add ax, 1111h
	objdump -M intel	cs ds es fs gs ss lock repz add ax, 0x1111
66 89 C0 / 66 8B C0	Online Disassembler	mov ax, ax / mov ax, ax
	Shell-Storm	Invalid opcode(s) / Invalid opcode(s)
	udis86	mov ax, ax / mov ax, ax
	IdaPro	mov ax, ax / mov ax, ax
	objdump -M intel	mov ax, ax / mov ax, ax
66 0f 19 34 60	Online Disassembler	nop WORD PTR [rax+riz*2]
	Shell-Storm	nop word ptr [rax]
	udis86	o16 nop [rax]
	IdaPro	nop word ptr [rax]
	objdump -M intel	nop WORD PTR [rax+riz*2]

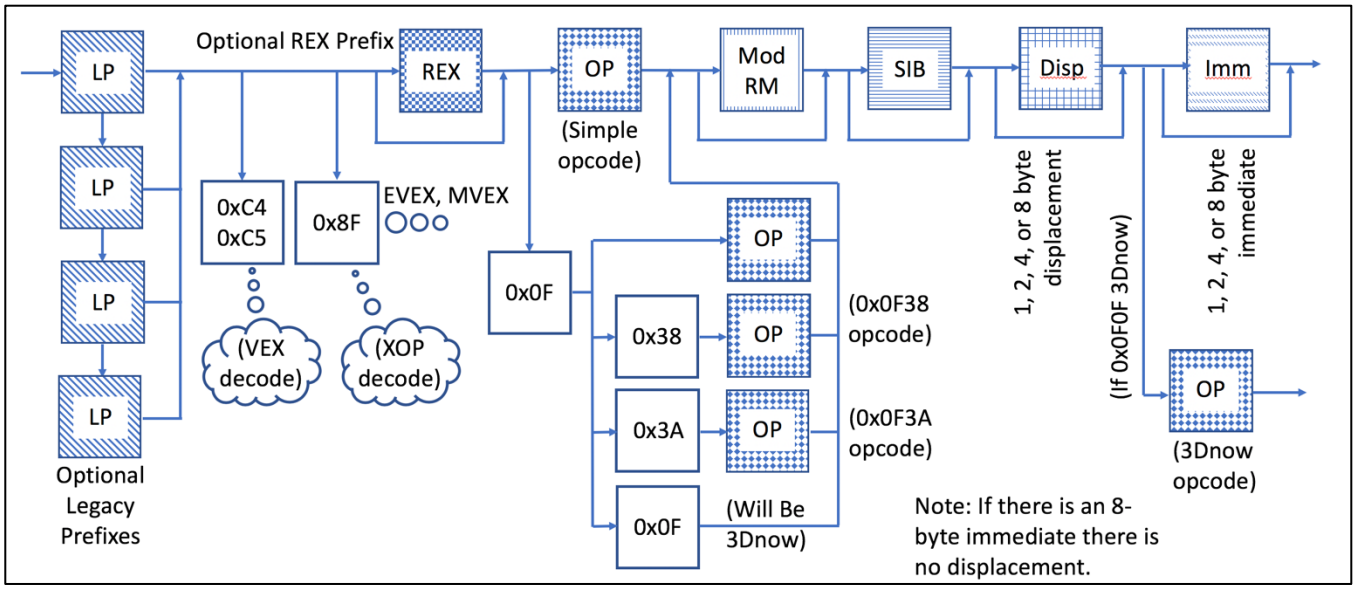

Figure 1: Partial Overview of x86-64 Decoding