Near Complete Formal Semantics of X86-64

Abstract

ToDo

1. Introduction

2. Preliminaries

3. X86-64 Instruction Semantics in K

3.1. Modeling Instruction Semantics

In this work we supported formal semantics of the input/output behavior of 2929 out of 3868 x86-64 Haswell ISA instruction variants. Figure 1 shows the classification of the instructions not supported using dotted ovals. The ones that are not supported can be categorized to System, Legacy mode, MMX, X87 and Cryptography instructions.

In order to get semantics of individual instructions, we build on top of project `Strata` [4] which automatically synthesized formal semantics of the input/output behavior for 1796 Haswell ISA X86-64 instructions. The key to their results is stratified synthesis, where they use a set of instructions whose semantics are known to synthesize the semantics of additional instructions whose semantics are unknown. Using a combination of stochastic search + pruning using testing (we refer as `initial search`) and subsequent refining of the search results using equivalence checking (`referred as secondary searches`), they first came up with the semantics of 692 register and $\sim$ 120 immediate instructions. The rest $\sim$ 984 are the immediate and memory variants obtained by generalization of 692 register instructions.

Strata Vs Stoke

Following are some of the immediate challenges that we needed to address.

1. **CH.1: Supporting *un-stratified* Instructions** The paper [4] mentions that adding some primitive instructions (like saturated add) as the base instruction might help stratified more instructions. We would like to explore similar directions. Moreover, it would interesting if we can leverage the manually written instruction semantics from project `Stoke`.
2. **CH.2: Getting Generic Formula for Immediates** The $\sim$ 120 immediate instructions, mentioned above, do not have a corresponding register-only instruction to generalized from. Therefore `Strata` tries to learn a separate formula for every possible value of the 8 bit immediate operand. We intend to have a more intuitive generic semantics (that works for all values of the immediate operand) for those instructions.
3. **CH.3: Modeling *undef* flags** There are instructions which conditionally sets some cpu flags to *undef*. For example, the shift left instruction `salq %cl, %rbx` sets flag *%of* to *undef* state if the count mask > 1. Also there are instructions like `blsr %eax, %ebx` which un-conditionally puts flags like *%pf* & *%af* into *undef* state.
   `Strata` while doing the `initial search` does not test the flags which *may* (for conditional *undef*s) or *must* (for un-conditional *undef*s) be taking undefined values. We intend to model the semantics of these flags with the same correctness guarantee as the other registers which do not result in *undef* and hence modeled by `Strata`.
4. **CH.4: Modeling *%af* flag** `Strata` chose not to model the *%af* flag as this is not commonly used. Supporting this flag fall within the scope of our work.
5. **CH.5: Generalization to Immediate and Memory** How reliable is the generalization of register instructions to memory or immediate variants? `Strata` states that the claim for the generalization is based on random testing.

Following is a key observation concerning stratification which help us handle the most of the above mentioned challenges.

Observation In order to get the semantics of a target instruction `I`, `Strata` uses `Stoke` along with a set `TS` of 6580 test cases to synthesize an instruction sequence which agrees with `I` on `TS` (which means the output behavior of the instruction sequence matches with that on real hardware for input `TS`). After having that `initial search`, they keep on searching additional sequences, called `secondary searches`, each agreeing with `I` on `TS`, in a hope of getting one which would prove non-equivalent to existing ones and thereby gaining more confidence on the search and probably an augmented test-suite (as `TS` might get augmented with a counter example from equivalence checker in the event of non-equivalence).

One unfavorable possibility for `Strata` is when all subsequent secondary search results proves equivalent to the one obtained from initial search and hence there are no conflicts among searches, in which case it means that secondary searches fail to add any "confidence" to the initial search result and end up giving the same correctness guarantee as provided by the initial search result. Even though in such unfavorable case, the secondary searches might have provided "better" choices to pick the final formula from. A better choice of formula do not contain uninterpreted functions or non-linear arithmetics and are simple.

In the paper[4], it is mentioned that there are only 50 cases, where they found a (valid) counterexample. That means, there are $762 = (692 + 120 - 50)$ instructions, whose the initial

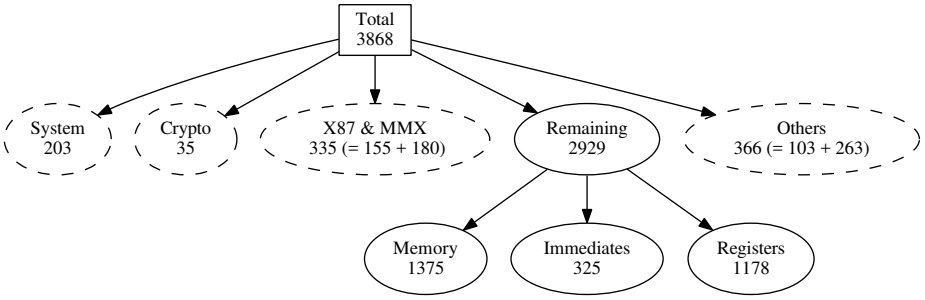

Figure 1: Instruction classification.
The solid ovals are the ones modeled by this work.

stoke search using augmented test-suite, containing 6630 ($= 6580 + 50$) tests, is sufficient enough to provide a semantics with the same correctness guarantee which `Strata` provides. In other words, in most of the cases, the correctness guarantee of secondary searches is same as that of the initial stoke search using the augmented test-suite (henceforth referred as `ATS`) which `Strata` ends up with.

Handling CH.1 For an unsupported instruction `I`, we either model its semantics manually or borrowed it from project `Stoke`. Once we have this candidate, we test it against hardware using `ATS`. Once the test passes we claim (from the above observation) the semantics to have the same correctness guarantee which `Strata` provides for most of its cases. This helped us finding instruction semantics bugs in Intel Manual [1] and `Stoke` [3].

We understand that this is not as efficient as `Stoke`, which is fully automatic in getting these formulas, and we do not intend to make any contribution towards efficient generation of instruction semantics. The purpose of above mentioned effort is to deliver in cases where `Stoke` cannot without loosing much on the correctness guarantee.

Moreover, writing the semantics manually might alleviate the need of secondary search as a means to provide "better" formula as we can control the complexity and choice of operations to include in the formula. Also carefully written manual formula tend to need less number of conflicting searches than the onces generated by random search engines like Stoke.

We also tried the following other options, which we do not pursue further:

- **Augmenting the Base Set:** Coming up with a suitable set of base instructions, which help synthesizing the semantics most of the user level instruction, could be framed as an optimization problem, which we do not explore in this work. Why?
- **Reducing `Stoke` Search Space:** This option is based on the observation that `initial search` for some instructions (like `vfmaddsub132pd %ymm1 %ymm2 %ymm3`) times-out because of the huge search space to be explored by `Stoke`. We tried to limit the search space using manually learned heuristics. An example of one such heuristic is *If we know the semantics of an instruction with ymm operands and the target instruction, which we want to learn, is a variant of that instruction and uses xmm operand, then the search pool should contain some specific instructions*. This particular heuristic work well for few instructions. In the general case, getting the search pool for every target instruction, need an approximate insight about the semantics of the target instructions itself. Even though such information is available in manuals but we find it difficult to extract it in a way to create the search pool, which is the main reason we drop this venture.

Handling CH.2 The instructions in this category either have a separate formula for all or some of 256 possible values. We refer each of the separate formulas for instruction `I` as a concrete formula F_c^I for a particular constant value `c` of immediate operand. In either case, we get a generic formula, G^I either by writing it manually or borrowing it from `Stoke` project.

In the case where we have a separate F_c^I $\forall c \in \{0...255\}$, we do a Z3 equivalence check as follows: $\forall c \in \{0..255\} : F_c^I \equiv_{Z3} G[c]^I$, where $G[c]^I$ is obtained by replacing the symbolic inputs of G^I with constant value `c`. A successful equivalence check suggest G^I to be a generic formula with the same correctness guarantee that `Strata` has for any of the individual concrete formulas. For the case where we have a separate formula for a subset of constant values, we do the same equivalence check as before for that subset. The constants for which we do not have a separate formula we test $G[c]^I$ using `TS`, the final test-suite of `Strata`.

Handling CH.3 There are 474 ($= 141(\text{Reg}) + 109(\text{Imm}) + 224(\text{Mem})$) instructions that results in conditional (or *may*) *undef* (162 ($= 40 + 46 + 76$)) or un-conditional (or *must*) *undef* (312 ($= 101 + 63 + 148$)) in one or more cpu flag. The semantics of most of the cpu flags (which *may* or *must* take *undef* values) are already modeled in `Stoke`. We needed to model the semantics of flag registers for 40 instructions involving `shifts, rotates` [2].

For *may undef* cases, we tested against hardware, using `TS`, for the scenarios when the condition for undefinedness is not triggered. For the remaining cases, (1) *may undef*s where the condition for triggering *undef* is true and (2) *must undef*, we make sure that `K` execution halts when the undefinedness

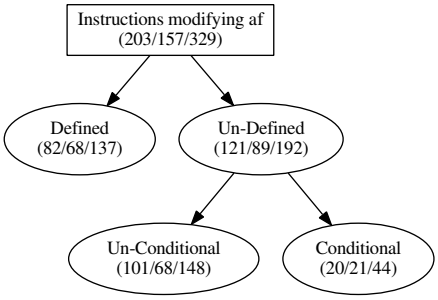

Figure 2: Instructions affecting *%af* flag. The numbers represent count of (Register/Immediate/Memory) Instructions.

condition is triggered. This help is find bugs in the `Stoke` implementation of 8 instructions [2] (Note that these 8 instructions are not stratified and hence we borrowed it from `Stoke`).

Handling CH.4 Figure 2 represents the distribution of instructions affecting the *%af* flag in a defined or un-defined way (which could be conditional or un-conditional). We tested all the instructions for the defined cases using `ATS`. For conditionally undefined cases, we tested for the scenarios when the condition for undefinedness is not triggered. For all remaining cases, we make sure that the `K` execution halts when the undefinedness condition is triggered.

Handling CH.5 While testing we found instructions like `movss xmm m64, movsd xmm 64` where the generalization from the corresponding register variant is not faithful. Followings are the semantics of `movsd xmm1, xmm2` and its memory variant `movsd xmm1, m64`. Clearly, the memory variant cannot to obtained using generalization of the corresponding register instruction.

```
// movsd xmm1, xmm2
S1.  DEST[63:0] ← SRC[63:0]
S2.  DEST[MAXVL-1:64] (Unmodified)

// movsd xmm1, m64
S1.  DEST[63:0] ← SRC[63:0]
S2.  DEST[MAXVL-1:64] ← 0
```

4. Evaluation

5. Application

6. Related Work

A. An Example of `Strata` Formula

Following is the `Strata` formula for an instruction `vpxor %ymm3, %ymm2, %ymm1`,

```

(let ((a!1 (bvxor ((_ extract 255 192) ymm3)
                  ((_ extract 255 192) ymm2)
                  (bvor ((_ extract 255 192) ymm3)
                        (bvxor ((_ extract 255
    192) ymm3)
                               ((_ extract 255
    192) ymm2)))))
      (a!3 (bvxor ((_ extract 191 128) ymm3)
                  ((_ extract 191 128) ymm2)
                  (bvor ((_ extract 191 128) ymm3)
                        (bvxor ((_ extract 191
    128) ymm3)
                               ((_ extract 191
    128) ymm2)))))
      (a!5 (bvxor ((_ extract 127 64) ymm3)
                  ((_ extract 127 64) ymm2)
                  (bvor ((_ extract 127 64) ymm3)
                        (bvxor ((_ extract 127 64)
     ymm3)
                               ((_ extract 127 64)
     ymm2)))))
      (a!7 (bvxor ((_ extract 63 0) ymm3)
                  ((_ extract 63 0) ymm2)
                  (bvor ((_ extract 63 0) ymm3)
                        (bvxor ((_ extract 63 0)
    ymm3) ((_ extract 63 0) ymm2))))))
(let ((a!2 (bvxor ((_ extract 255 192) ymm3)
                  ((_ extract 255 192) ymm2)
                  (bvor ((_ extract 255 192) ymm3)
                        (bvxor ((_ extract 255
    192) ymm3)
                               ((_ extract 255
    192) ymm2)))
                  (bvor a!1
                        ((_ extract 255 192) ymm2)
                        ((_ extract 255 192) ymm3)
    )))
      (a!4 (bvxor ((_ extract 191 128) ymm3)
                  ((_ extract 191 128) ymm2)
                  (bvor ((_ extract 191 128) ymm3)
                        (bvxor ((_ extract 191
    128) ymm3)
                               ((_ extract 191
    128) ymm2)))
                  (bvor a!3
                        ((_ extract 191 128) ymm2)
                        ((_ extract 191 128) ymm3)
    )))
      (a!6 (bvxor ((_ extract 127 64) ymm3)
                  ((_ extract 127 64) ymm2)
                  (bvor ((_ extract 127 64) ymm3)
                        (bvxor ((_ extract 127 64)
     ymm3)
                               ((_ extract 127 64)
     ymm2)))
                  (bvor ((_ extract 127 64) ymm2)
    ((_ extract 127 64) ymm3) a!5)))
      (a!8 (bvxor ((_ extract 63 0) ymm3)
                  ((_ extract 63 0) ymm2)
                  (bvor ((_ extract 63 0) ymm3)
                        (bvxor ((_ extract 63 0)
    ymm3) ((_ extract 63 0) ymm2)))
                  (bvor ((_ extract 63 0) ymm2) ((
    _ extract 63 0) ymm3) a!7))))
  (concat a!2 a!4 a!6 a!8)))
```

where as following is the formula obtained from `Stoke` (hand-written) and Z3 took 88.70 secs to prove that they are equivalent.

```
%ymm1   : (bvxor %ymm2 %ymm3)
```

References

[1] Bug reported in Intel Developer Zone: Possible errors in instruction semantics. https://software.intel.com/en-us/forums/intel-isa-extensions/topic/773342, April 2018. Last accessed.

[2] Bug reported in Stoke: Modelling the behavior of flags which may or must take undef values. https://github.com/StanfordPL/stoke/issues/986, May 2018. Last accessed.

[3] Bug reported in Stoke: Semantic bugs. https://github.com/StanfordPL/stoke/issues/983, April 2018. Last accessed.

[4] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 237–250, New York, NY, USA, 2016. ACM.