New Simics cpuid Definition and Implementation Notes

Revision 6

Grigory Rechistov

Contents

[Introduction 1](#_Toc383076407)

[Requirements 1](#_Toc383076408)

[Feedback after the first discussion 2](#_Toc383076409)

[Solution 2](#_Toc383076410)

[Specification Language 2](#_Toc383076411)

[Converter to C 3](#_Toc383076412)

[Example of Input Specification and Generated Code 4](#_Toc383076413)

[Simulation Algorithm Details 5](#_Toc383076414)

[Use Case Examples 5](#_Toc383076415)

[Adjusting cpuid Contents for a cpu Model 5](#_Toc383076416)

[Adding a New Feature that is Exposed Through a cpuid Flag 6](#_Toc383076417)

[Adding a New cpu Model 6](#_Toc383076418)

[Override a cpuid Field from Script 7](#_Toc383076419)

[Override a Complete cpuid Leaf/Subleaf 7](#_Toc383076420)

[Print cpuid Table 7](#_Toc383076421)

[Expected Source Code adjustments 7](#_Toc383076422)

[ebnf Specification of the Model List File Format 8](#_Toc383076423)

[Notes 8](#_Toc383076424)

# Introduction

These notes describe a new approach to the cpuid simulation in Simics ia‑32 cpu models.

# Requirements

The solution must fulfill several requirements.

1. Being able to encode all cpuid leafs/subleaves.
2. Support dynamic nature of certain output bits of cpuid result.
3. Allow users to override parts of cpuid results during simulation without a need of recompiling.
   1. “Bulk” writing/overriding of the whole out register
4. Be backwards compatible with existing C-code cpuid tables to allow to extended period of transition between schemes.

## Feedback after the first discussion

1. The most important goal is to reduce number of places that affect cpuid as possible.
2. cpuid has to keep ability to override whole leaves/subleaves/registers, not just fields
3. Customers would want to be presented with cli commands like get\_cpuid(core, leaf/subleaf) and set\_cpuid (core, leaf, subleaf, value)
4. It would be nice to have these types of functionality for print\_cpuid cli command:
   1. Highlight values that were overridden by customer (to ease support from iss side)
   2. Add possibility to have verbose output of “cpuid field-value” pairs, not just out regs
5. processor\_t.config.has\_\* have to be automatically populated with starting values by the translator script
6. cpuid specification should be similar to x86-model-list used for msrs, possibly combined into one file.

# Solution

1. Define a specification language for cpuid definition with *cpuid* field as a basic building block
2. Process the input cpuid specification with (Python) script that converts it to C used in Simics models and test generation.
3. Provide cpuid\_fields\_overrides list attribute that contains items similar to the cpuid field spec and takes precedence over compiled in values. Keep existing cpuid\_list override to have a complex and complete override method for those who need it.

Optionally:

1. Use the cpuid specification to generate unit tests by translating it with (Python) script to Python testing code.

## Specification Language

A cpuid definition consists of one or more *cpuid* fields. Each field has a symbolic name and defines a range of bits of output register 32-bit value. The field value can be either constant or calculated by a C function.

The complete cpuid definition may be stored in one or several input specification files, with its common and rarely changed part separated in common file.

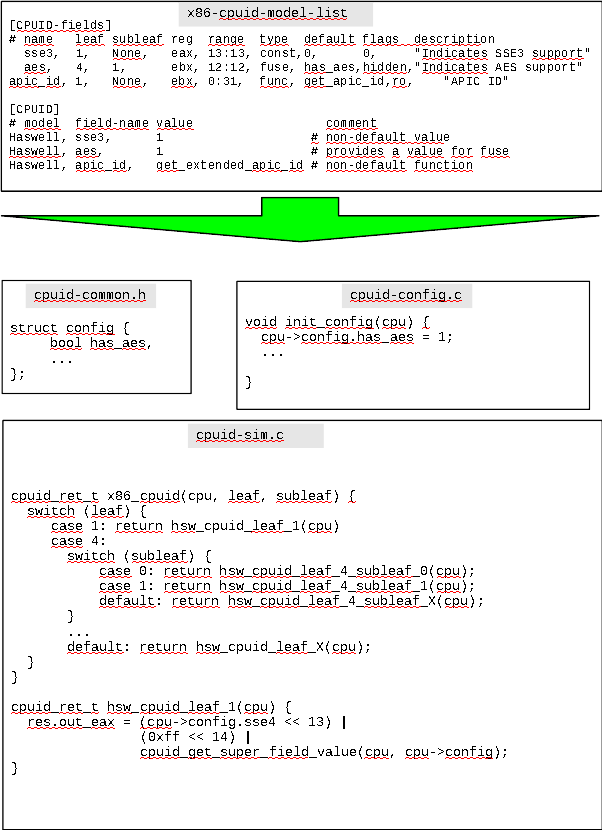
At simulation time, the whole list is looked for relevant fields and the output values are constructed from all fields that match.

## Translator to C

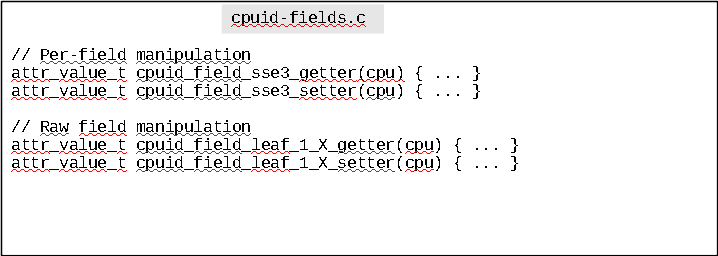
A script accepts input cpuid spec­­ifications, checks it to be valid, perform certain additional checks, if necessary (e.g., all bits are covered by bit fields, no two fields ranges intersect, no duplicates etc.) and converts it into C code to be incorporated into a cpu model build.

Input language notation may vary. In its simplest form, it consists of Python objects definitions that are sources by a Python script.

## Example of Input Specification and Generated Code



Getters and setters for attributes are generated too, as show on the figure below.



# Simulation Algorithm Details

The most important features of generated code are outlined below.

1. Getters and setters for attributes of individual cpuid fields are created and registered.
2. A linear search of leaf/subleaf is not used. Instead, up to two indirect jumps are used (i.e., switch-switch) to locate a subroutine that assigns values to out registers.
3. Bit values of None leaf are always used before values of a matching subleaf. This violates a rule that every out register bit has to be calculated only once, but allows to compensate for idiosyncrasies of leaf 11.
4. Generated body of each leaf/subleaf that combines inlined accesses to selected fields (to reduce algorithmic complexity of out register calculation)

# Use Case Examples

In this section, typical intended scenarios for performing basic operations by different groups of developers/customers are described. Implementation notes are provided for each case.

## Adjusting cpuid Contents for a cpu Model

If a (hypothetic) field supermath is to be changed for Haswell, x86-cpuid-model-list for that model has to be adjusted in the relevant [CPUID] section:

Haswell, supermath, 1 # Super Mathematics functions are enabled

This will be translated into cpuid-config.c with different assignment for config.has\_supermath field.

### Types of cpuid Fields

* const—equal for all CPUs. Examples: “GenuineIntel”.
* field—per-instance editable.
* fuse—per-class non-editable.
* func—per-instance function with getter (and setter?).

## Adding a New Feature that is Exposed through a cpuid Flag

Two new architectural features are to be supported in Troznarf cpu model: architecture extension Foo, and cache parameter Bar, 16-bit wide, is calculated with complex formula. Foo is a top secret field and cannot be exported through attributes.

Therefore, x86-cpuid-model-list needs two changes.

1. In [CPUID-fields] section—two descriptions of respective fields:

foo, 0xb, 1, edx, 23:23, fuse, has\_foo, hidden, "Foo support"

bar, 0x10, None, edx, 0:15, func, get\_foo, 0, "Bar cache size"

1. In [CPUID] section—values for new fields has to be set:

Troznarf, foo, 1

# No line for Bar - the default getter get\_foo() is used

The translator will add new field has\_foo to the config and will call get\_foo() inside x86\_cpuid() for leaf 0x10.

## Adding a New cpu Model

For a new cpu model, called Troznarf, a cpuid specification was provided in some form (document/code/rumor). We now need to encode it into the x86-cpuid-model-list. There are two formats that translator accepts.

1. Encode every cpuid field with a non-default value in the [CPUID] section:

Troznarf, sse41, 0

Troznarf, rdrand, 1

Troznarf, l2\_slices, get\_troznarf\_slices

< etc... >

1. Use compact format to specify leaf/subleaf:

Troznarf, 4:1, 0x2010043, 0x01, 0x20, 0xfff

The compact formats is leaf:subleaf. Full 32-bit values have to be specified. The translator splits them into fields and assigns config fields/fuses itself. It also issues warnings for inconsistencies:

1. a constant cpuid field was assigned to a different value;
2. a functional field has been overridden with a constant value.

## Override a cpuid Field from Script

A customer wants to override a particular cpuid field by addressing it by name.

This functionality is implemented through a number of generated attributes. The translator generates cpuid-fields.c file with code for definitions and registration of attributes for overriding cpuid fields. For a field foo an attribute cpuid\_foo\_override is registered with type i|n. It defaults to None.

cpuid fields declared as read-only, constant or functional generate warnings if an attempted to modify them is taken; still, the new value is saved and later used. Const and hidden fields do not have associated override attributes.

For leaves not present in the specification an attribute cpuid\_extra\_leaves (type [[ii|niiii)]\*]) can be used to provide values.

## Override a Complete cpuid Leaf/Subleaf

A customer wants to modify certain results of cpuid instruction. Simics provides cli command <core>.set\_cpuid leaf subleaf register value. If functional or read-only fields are touched, warnings are issued.

For user’s convenience, a pseudo-attribute cpuid\_leaf\_override (type [i(i|n)(i|n)(i|n)(i|n)(i|n)], write-only) is generated for automatically splitting values into distinct fields:

core.cpuid\_leaf\_override = [4, None, 0x11, 0x22, 0x33, 0x44]

This results in assignments of individual cpuid\_xyz\_override attributes that constitute 32-bit out registers.

## Print cpuid Table

The cli command <core>.print\_cpuid is modified to mark (with star symbol) values that have at least one bit overridden through attributes.

The translator will generate a Python function that queries for cpuid, splits 32-bit out registers into fields and prints a list of non-hidden fields.

# Source Code Adjustments to Be Done

* x86ex—all code for cpuid is removed.
* x86—external interfaces for cpuid querying are left untouched. Functionality is moved to generated files. Makefile is adjusted to invoke the cpuid specification translator to code.
* Component (Python) code — should be rewritten to address specific cpuid fields. Generated files are included for providing new commands of reading/writing cpuid fields and registers.
* Makefile system — new generated sources are included to compiling and linking stages.
* x86ex\_cpu — a model-specific data should be set up in the cpu\_new() function that is called when a new object is created.

# ebnf Specification of the Model List File Format

TODO This spec may be incomplete

specification file = { section };

comment = “#” , { alphanum };

alphanum = letter | number ;

digit = “0” – “9” ;

number = decimal number | hexadecimal number;

decimal number = digit, {digit};

hexadecimal number = “0x”, digit, {digit};

string chunk = quote, [ascii], [ascii], [ascii], [ascii], quote;

quote = “"”;

letter = “a” – “z” | “A” – “Z” | “\_” ;

ident = letter , { alphanum };

section = general section | fields section | cpuid section;

general section = general header , {general line};

general header = “[GENERAL]”, newline ;

general line = targets line | target group line | comment;

targets line = “Targets:” , ident, {ident}, [comment], newline ;

target group line = “Target-group: ”, ident,“=”,ident,{ident},[comment], newline;

fields section = fields header, {fields line};

fields header = “[FIELDS]”, newline ;

fields line = name, “,”, leaf, “,”, subleaf, “,”, reg, “,”, range, “,”, type, “,”, default, “,”, flags, “,”, description, newline;

name = ident;

leaf = number;

subleaf = “None” | number;

reg = “eax” | “ebx” | “ecx” | “edx”;

range = number [“:” number];

type = “field” | “const” | “fuse” | “const”;

default = “None” | number | ident ;

flags = “0” | {flag} ;

flag = “legacy” | “hidden” | word;

description = [ascii] ;

cpuid section = cpuid header, {cpuid line};

cpuid header = “[CPUID]”, newline ;

cpuid line = cpu class, “,”, field definition | four register definition, [comment];

cpu class = ident;

field definition = ident, “,”, number | ident;

four register definition = leaf, “:”, subleaf, reg definition, “:”, reg definition, “:”, reg definition, “:” reg definition, newline;

reg definition = “None” | number | string chunk ;

# Notes

## Attributed Storage Format

TODO This is to be done during refactoring.

* <field>\_override—pseudo attribute to store cpuid fields/fuse/func . override. Type: i|n
* cpuid\_override\_raw\_fields—“raw” storage for overridden fields. Type: [ [i i|n i i i i]\* ], which is a list of (leaf, subleaf, register, bit start, bit end, value). The rationale behind storing raw fields instead of named ones is that it allows to change the fields layout without breaking checkpoint capabilities.