CGEN LLVM-IR
Design Document

Leonardo Arcari
Politecnico di Milano

February 2018

Contents

1 Introduction

1.1 Scope

This document is meant to provide a resource to those who are going to work with GNU CGEN and my extension to it: CGEN LLVM-IR. The purpose of this paper is to introduce the reader first to GNU CGEN from a code perspective, as GNU CGEN already provides a user guide. The reader will find in this document a code analysis, with a, possibly more clear, description of the main classes in Scheme source code in order to use them effectively.

In second place, I will provide a similar description of the code that I wrote in order to extend GNU CGEN to allow the generation of C++ programs capable of translating binary programs into a semantically equivalent representation in LLVM-IR language.

1.2 Out of scope

In this paper I am not going to describe several topics related to GNU CGEN

- How to run GNU CGEN. There is a manual online for it.[1]
- What is the plethora of features of GNU CGEN. There is a manual online for it.[2]
- What is CGEN RTL and what each language feature does. There is a manual online for it.[3]
- How to write a CGEN application to define your CPU architecture in RTL. Guess what? There's a manual online for it.[4]

Also, a pre-requisite to understand completely this document, the reader should know Lisp in one of its dialects. For soundness, be aware that CGEN is written in Scheme in the dialect implemented by Guile 1.8.0.

1.3 Project History

CGEN LLVM-IR generator is part of the project I was assigned to while taking the *Code Transformation and Optimization* course held by Professor G. Agosta[5] in the A.Y. 2017/2018. The idea of extending GNU CGEN, in order to generate C++ translators capable of producing a semantically-equivalent representation in LLVM-IR of a binary for a given architecture, is from Alessandro Di Federico, PhD[6].

[1] https://sourceware.org/cgen/docs/cgen_2.html
[2] https://sourceware.org/cgen/docs/cgen_1.html
[3] https://sourceware.org/cgen/docs/cgen_3.html
[4] https://sourceware.org/cgen/docs/cgen_8.html
[5] https://home.deib.polimi.it/agosta
[6] https://clearmind.me/

2 GNU CGEN

2.1 Introduction to CGEN

In this section I would like to give a high-level presentation of GNU CGEN, what it is useful for and why we think that provides enough value for the purposes of our project.

Goal "The goal of CGEN (pronounced seejen, and short for "Cpu tools GENerator") is to provide a uniform framework and toolkit for writing programs like assemblers, disassemblers, and simulators without explicitly closing any doors on future things one might wish to do. In the end, its scope is the things the software developer cares about when writing software for the cpu (compilation, assembly, linking, simulation, profiling, debugging, ???)"[7].

They way CGEN plans to achieve this goal is centered around having a CPU description language, called *RTL*, totally agnostic about the final goal. In RTL the programmer can describe:

CPU architectures General purpose registers, status registers

ISA Instructions, operands, instruction formats, instruction fields

Semantics What is the output, what registers change and how when instruction A is executed?

And a lot more[8].

Project idea The idea behind our project, CGEN LLVM-IR, is the following. CGEN is already able to generate GDB simulators for any architecture given its description in RTL language. Simulators, very simplistically, accept a binary program as input, emulate the hardware architecture in memory by means of variables to represent registers and emulate the execution of the input program line of code by line of code. This looks a lot like our objective.

If we were required to outline the execution of our project, in fact, that would by sketched by the following steps:

- Allocate a set of LLVM-IR global variables to mock general purpose registers, program counter and CPU status registers.
- Disassemble the binary input to reconstruct the assembly instructions and their operands.
- Through LLVM framework, emit LLVM-IR code that mimics the semantic of each istruction and sets our *mock registers* correctly.

[7] https://sourceware.org/cgen/docs/cgen_1.html#SEC3
[8] https://sourceware.org/cgen/docs/cgen_3.html

With this workflow in mind, our approach was as much as conservative as we could. We wanted to reuse CGEN code as much as possible, so we analyzed CGEN source code deeply. We started by looking at the those components that were responsible of generating the GDB simulator.

We discovered that the frontend part of CGEN could be easily reused. Frontend components tackles the problem of parsing RTL language to build an internal representation of language constructs and access their data efficiently.

So language parsing and data structures were there to be used. We could not say the same for components dealing with simulators generation.

It should be noted that GDB simulators are C programs, so CGEN was coded to emit C lines of code. Those components would have been a great reference for the logic that drives disassembling and instruction simulation, but they were required to be completely rewritten to emit C++ code. Unfortunately the C code generation was so tightly coupled in them that we had to write a whole new set of components to address our needs. More details are provided in section 3.

2.2 CGEN RTL classes

In this part of the document I want to provide an insight of Scheme classes in CGEN that represent internally the language constructs of RTL and allow the programmer to access the data written in the CPU description file.

To better understand the relationship between classes, I first present an example of the structure of an RTL description.

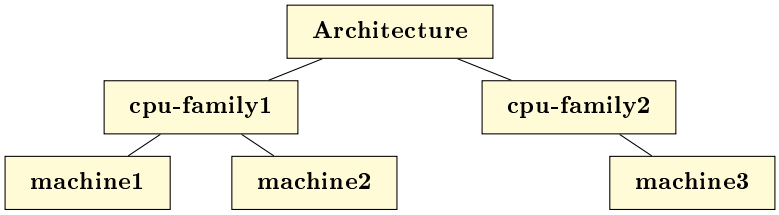

Figure 1: A graphical layout of top level RTL elements. The architecture is one of 'sparc', 'm32r', etc. Within the 'sparc' architecture, cpu-family might be 'sparc32', 'sparc64', etc. Within the 'sparc32' CPU family, the machine might be 'sparc-v8', 'sparclite', etc.

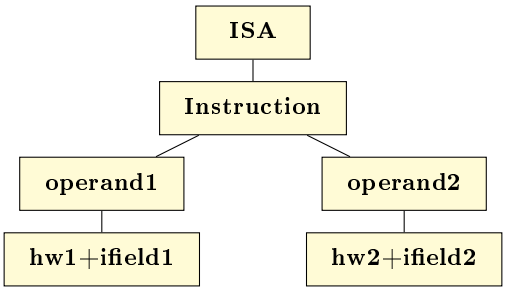

Figure 2: Instructions form their own hierarchy as each instruction may be supported by more than one machine

2.2.1 CGEN's object system - cos.scm

Although Guile, the Scheme implementation supported by CGEN, provided an official object system in the 1.8 release, the CGEN author thought that things might have changed and he wanted to be sure not to be required to change the entire CGEN code base in case that happened. Thus he decided to implement his own object system and we must deal with it. I'm going to give a presentation of those feature that you might come across while working on CGEN codebase and you might need to know.

Class Classes is CGEN are implemented (of course) as vectors of information defining your class, as you can see in listing 1

Listing 1: A class in CGEN looks like this

```
#(class-tag
  class-name
  parent-name-list
  elm-alist
  method-alist
  full-elm-initial-list
  class-descriptor)
```

The fields you should care about are the following:

class-name A name uniquely defining the class. E.g: `<arch>`

parent-name-list A list of the names of parent classes (the inheritance tree).

elm-alist A list of `(symbol private? vector-index . initial-value)` for this class only.

method-alist An alist of `(symbol . (virtual? . procedure))` for this class only.

To declare a new class: `(class-make name parents elements methods)` An example of class declaration is available at listing 2

Listing 2: An example of class declaration in CGEN

```
(define <mach>
  (class-make
    '<mach>
    '(<ident>)
    '(
      ; cpu family this mach is a member of
      cpu
      ; bfd name of mach
      bfd-name
      ; list of <isa> objects
      isas
    )
    nil)
)
```

The above example shows a common practice in CGEN. Methods are defined after class declaration with the help of some macros/procedures.

Getters and Setters declaration To add getters and setters method to a class two convenient macros are provided:

```
define-getters (class class-prefix elm-names)
define-setters (class class-prefix elm-names)
```

Other methods declaration For all other kinds of methods two procedures are available:

```
(method-make!  class name lambda)
(method-make-virtual!  class name lambda)
```

Listing 3: Example of methods declaration for a class

```
; Define getters for class <mach> for members
; 'cpu', 'bfd-name' and 'isas' and name them
; 'mach-<member>' where <member> is
; [cpu|bfd-name|isas]
(define-getters <mach> mach (cpu bfd-name isas))

; Define setter for class <ifield> for member
; 'follows' and name it 'ifld-follows'
(define-setters <ifield> ifld (follows))

; Define a method for class <ifield> named
; 'get-field-value whose implementation is
; defined by the lambda
(method-make!
 <ifield> 'get-field-value
 (lambda (self)
   (elm-get self 'value))
)
```

Method invokation CGEN's object system follows the Smalltalk way of implementing object orientation, that is by means of *messages*. Thus we can invoke a method on an object with:

```
(send object method-name .  args)
```

Listing 4: Example of methods invokation

```
; We wrap a method invokation in a standard
; Scheme procedure for simplicity of usage
(define (ifld-set-value! self new-val)
  (send self 'set-field-value! new-val)
)
```

2.2.2 Arch - mach.scm

Arch is the top level class in CGEN that records everything about a CPU. After parsing a .cpu file the programmer can refer to a global variable named

`CURRENT-ARCH` to access an instance of Arch.

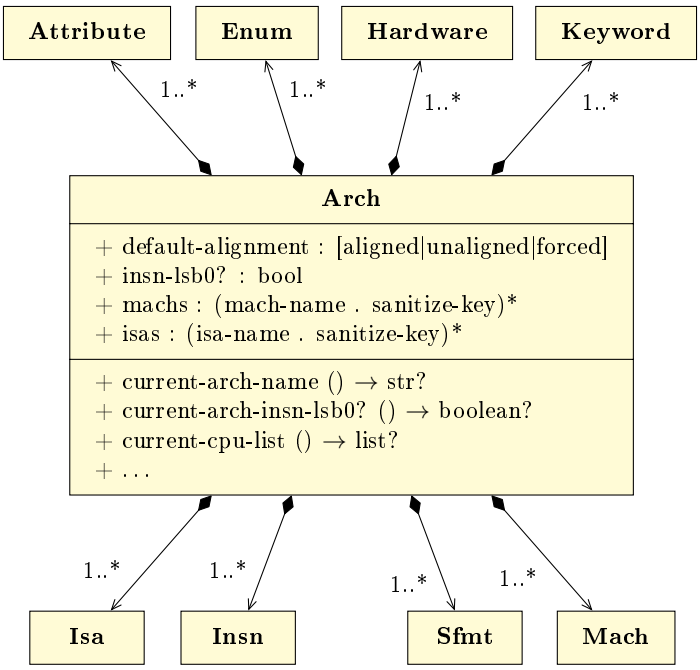

Figure 3: Class diagram of <arch> CGEN class

2.2.3 Hardware - hardware.scm

`<hardware-base>` is the base class for all hardware descriptions. The actual hardware objects inherit from this (e.g. register, immediate). This is used to describe registers, memory, and immediates.

`mode` in diagram 4 refers to one of the many data types you can specify in RTL. Look here for more information.

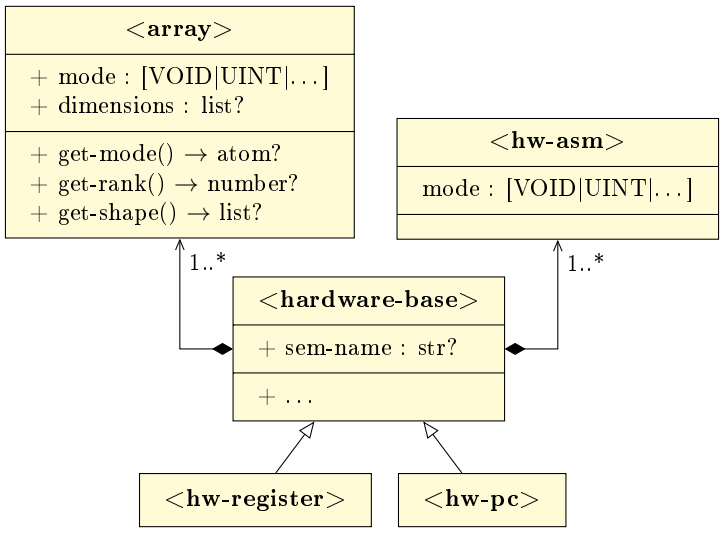

Figure 4: Class diagram of hardware.scm CGEN classes

2.2.4 Instruction - insn.scm

`<insn>` is the class to hold an instruction. This class is very important as it is an entry point to deal with instruction disassembling and translating into LLVM-IR.

The programmer can retrieve the parsed list of ISA instructions with the nullary procedure `current-insn-list`.

`semantics` member of `<insn>` contains the RTL source code explaining the instruction semantic. This gets compiled by CGEN and transformed into an `<rtx-func>` object representing the RTL expression in Scheme. The `<rtx-func>` object is stored in `compiled-semantics` member.

`bitrange` member of `<ifield>` contains the field's offset, start, length, word-length and orientation ($msb == 0$, $lsb == 0$). Although this seems promising data, it is *not trustworthy*. In fact, current stable release of CGEN (1.1 at the moment of writing) has issues in dealing with ISAs with variable length instructions, thus some values like `length` or `word-length` might be wrong. According to my research on this topic, only ISAs with instruction of fixed length (say 32bit) allow the programmer to exploit and trust values within `bitrange` member. For more complex architectures that value is misleading so it should be ignored. Some *.cpu* declaring weird istruction sets provided a custom way to fetch instructions from binary programs. This requires more investigation.

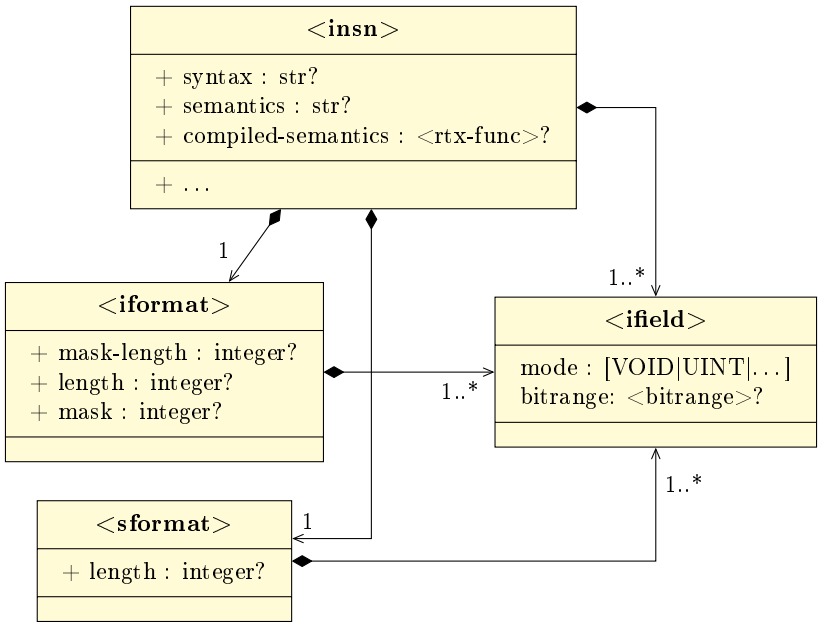

Figure 5: Class diagram of insn.scm and iformat.scm CGEN classes

2.2.5 Ident - a common base class

One thing I did not mention so far is that every class described in this section inherits from a general base class: `<ident>`.

Listing 5: <ident> class declaration

```
1 (define <ident>
2   (class-make '<ident> '()
3     '(name comment attrs)
4     '()))
5
6 ; getters and setters...
```

name Names must be valid Scheme symbols.

comment Comments may be any number of lines, though generally succinct comments are preferable.

attributes A list of attributes[9]

[9] https://sourceware.org/cgen/docs/cgen_3.html#SEC56

2.3 Code Analysis

2.3.1 Entry Point

2.3.2 RTL-C Generator

3 CGEN LLVM-IR

3.1 CGEN-IR common

3.2 IR-Gen registers

3.3 IR-Gen decoder

3.4 RTL-CPP Generator