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, as a pre-requisite to completely understand this document, the reader should know Lisp in one of its many dialects. For soundness, be aware that CGEN is written in Scheme in the dialect implemented by Guile 1.8.

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.

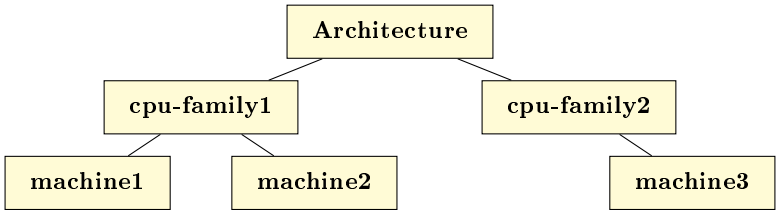

Figure 1: A graphical layout of top level RTL elements. The architecture is one of 'sparc', 'm32r', etc. Within the 'sparc' architecture, cpu-family might be 'sparc32', 'sparc64', etc. Within the 'sparc32' CPU family, the machine might be 'sparc-v8', 'sparclite', etc.

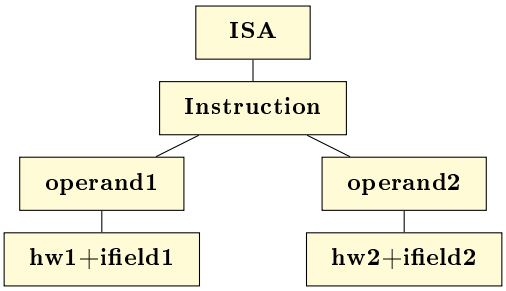

Figure 2: Instructions form their own hierarchy as each instruction may be supported by more than one machine

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.

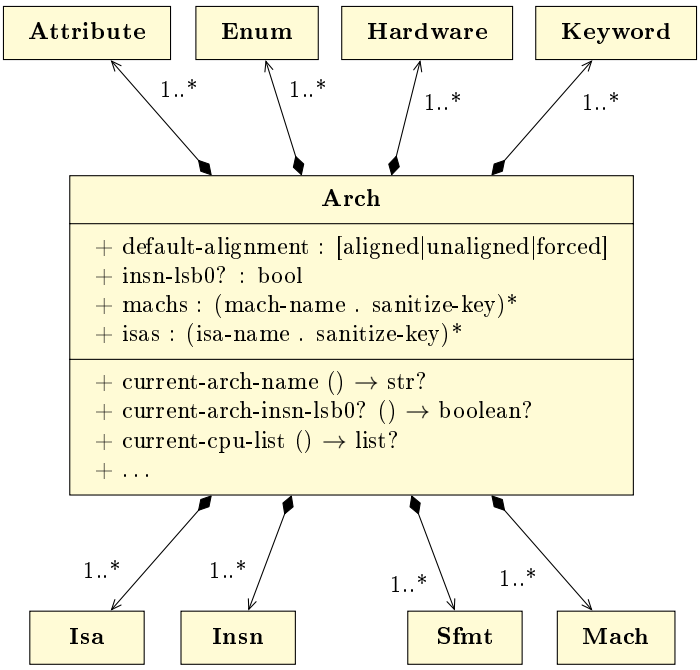

Figure 3: Class diagram of <arch> CGEN class

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. Please refer to the online manual for more information[9].

[9] https://sourceware.org/cgen/docs/cgen_3.html#SEC161

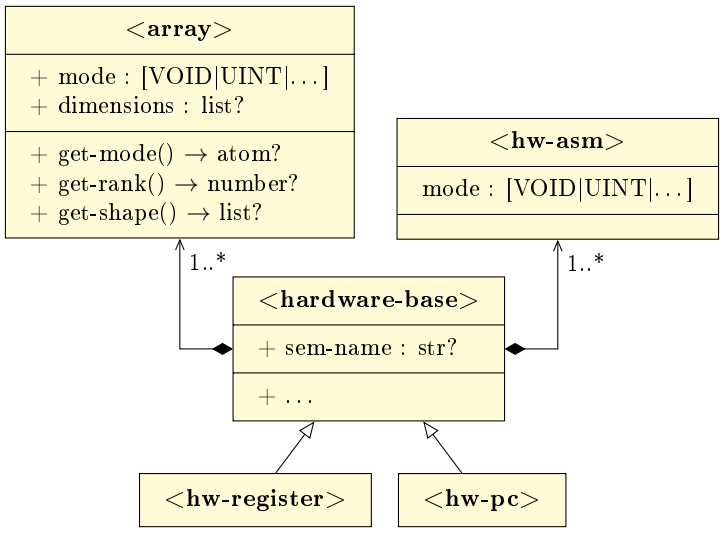

Figure 4: Class diagram of hardware.scm CGEN classes

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.

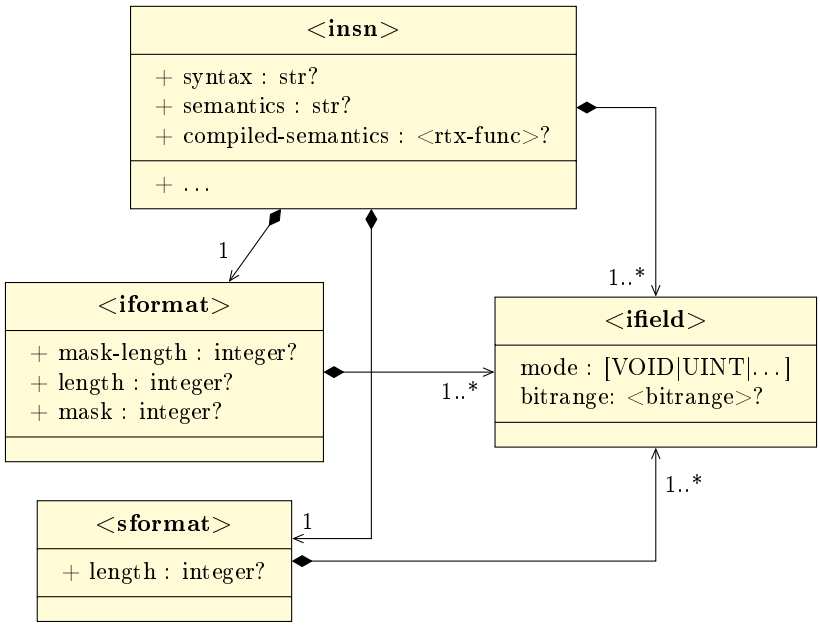

Figure 5: Class diagram of insn.scm and iformat.scm CGEN classes

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[10]

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

2.3 Code Analysis

In this section I want to give an analysis on GNU CGEN execution flow from the program launch to the entry of the various components. This should help the programmer to know what source files to edit if they want extended CGEN functionalities and be able to call them from command line.

In section 2.3.1 I am discussing about the command line interface the user exploits to run CGEN applications (e.g. simulators, disassembler, toolchain) and how to add new applications to it.

In section 2.3.2 I am providing some insights on how CGEN generates C code from an RTL expression. Even though CGEN LLVM-IR is unrelated to this component, the program logic to emit C++ or LLVM-IR code is just the same, so we better learn from the wise GNU CGEN author and contemplate.

Note for the reader: for the rest of this section I am writing about GNU CGEN project only, not CGEN LLVM-IR. This allows me to refer to files and procedure whose names are stable and reliable. Same concepts described hereinafter apply to CGEN LLVM-IR too. Still I am discussing about my project execution flow in section 3.

2.3.1 Entry Point

Let us begin with an example on how CGEN is run from command line

```
1 guile -l guile.scm cgen-sim.scm # lots of parameters
```

With parameter `-l` we are kindly asking `Guile` to load the following source files: `guile.scm` and `cgen-sim.scm`. What is happening? We should forget `guile.scm` since it "only" sets some debugging variables in Guile for CGEN and shift our focus on `cgen-sim.scm`.

`cgen-sim.scm` is one of the many CGEN applications that are part of the GNU project. According to its name, it is responsible to generate `GDB` simulators for a given architecture. And here we find our first takehome lesson: do you want to extend CGEN for another purpose? Provide an application source file. In fact if we take look inside `cgen-sim.scm` we read code for:

- Setting up **init, clean-up and analyze** procedures
- Loading **additional source** files
- Handling command line **parameters**

read.scm One little thing I should mention is the importance of this line of code:

```
1 (load (string-append srcdir "/read.scm"))
```

Without going in too much details, `read.scm` loads all the Scheme source files to internally represent RTL language constructs (some of them were mentioned in section 2.2) and runs their `init procedures`. The ordering in which those procedure calls are done is meaningful and things blow up in no time. So I strongly advice the reader to simply add the above line of code in their CGEN application sources and forget about it.

argv The last thing I am mentioning is how command line parameters are defined and handled. Within `read.scm` a procedure named `cgen` is defined. Its purpose is to gather all those launch parameters I was referring to a few lines above. A sample of its invokation is provided:

```
(cgen
  #:argv argv
  #:app-name "sim"
  #:arg-spec sim-arguments
  ; Init procedure
  #:init sim-init!
  ; Clean-up procedure
  #:finish sim-finish!
  ; General purpose .cpu file
  ; analysis routine, not used
  ; in general
  #:analyze sim-analyze!))
```

A careful eye must have spot the atom `sim-arguments`. That is our procedure name setting up the CGEN application command line parameters handlers. The programmer can provide their own implementation. A sample if provided hereinafter.

```
(define sim-arguments
  (list
    (list "-A" "file" "generate arch.h in <file>"
      #f
      (lambda (arg) (file-write arg cgen-arch.h)))
    ; other params...
   )
```

A bit of documentation:

"-A" Command line argument.

"file" The formal parameter name following argument `-A`.

"generate..." Argument description.

#f You can place a string comment instead.

arg The actual value for formal parameter `file`. Its value is the one passed from the command line.

Just because it took me a while to figure it how, note that `cgen-arch.h` is a procedure and it returns the C code to write on file `arg`.

2.3.2 RTL-C Generator

In this section I want to write about the architecture of `rtl-c.scm`, the component to generate C code given an RTL expression as input. Even if possibly unnecessary, after all RTL is very popular nowadays, I recommend a refresh on the language features for a better understanding[11].

Let us start from a user perspective. I have an RTL expression

```
(add SI (const SI 1) (const SI 2))
```

and I want to compile it into C code. I can simply run

```
(rtl-c
  ; Expression mode
  DFLT
  ; RTL expression
  '(add SI (const SI 1) (const SI 2))
  nil)
```

and I get a C expression (you can tell we are true because it is a macro)

```
ADDSI (1, 2)
```

So far so good. Now we want to dig a little bit deeper and understand what is going on. First of all, the output we got is part of the `<c-expr>` class

[11] https://sourceware.org/cgen/docs/cgen_3.html#SEC47

Listing 6: Part of <c-expr> class declaration

```
(define <c-expr>
  (class-make '<c-expr> nil
    '(
      ; The mode of C-CODE.
      mode
      ; The translated C code.
      c-code
      ; The source expression, for debugging.
      expr
      ; other things follow...
    )
    nil))
```

`<c-expr>` class is ubiquitous in `rtl-c.scm` component because it ties together RTL and C code. We now ask ourselves how its fields are filled in.

RTL→C generators First of all, when an RTL expression is passed to `rtl-c` it gets compiled by `rtl.scm` component which I am not discussing in this document. Afterward, RTL expression is translated into C by means of a **translation table** that maps each RTL operation to a Scheme procedure to handle it. Within `rtl-c.scm` this table is named `-rtl-c-gen-table` and I mention it because it is **the key** to implement the translation from RTL to another language. The programmer must provide a map similar to this and most of the code in `rtl-c.scm` can be left unchanged[12]. This is a huge gain in terms of code reusage and CGEN author must be praised for this.

Get and set hardware In RTL the programmer can assign values to hardware elements and read from them. This is a key feature to allow the description of instruction semantics or complex disassembling procedures. So how do we deal with it in emitting C code? Hardware get/set is handled by means of class methods. In fact each hardware class[13] must answer to the following messages

get-mode Return mode of the hardware element.

cxmake-get Return a `<c-expr>` to get its value.

gen-set-quiet Return string of C code to set operand's value (no tracing).

gen-set-trace Return string of C code to set operand's value.

This way each "complex" operand emits the C code for reading and writing its value(s). *Delegation* for the win.

[12] Well, actually you have to write a copy of it, changing most of the procedure names because there is no overloading in Scheme. Still, though tedious, it is better than writing it from scratch

[13] Section 2.2.3

2.4 Conclusions

In this section we went through many aspects of the design of GNU CGEN and what conventions its author adopted. I mentioned only those aspects that I found relevant in extending CGEN for our purposes, many other parts of CGEN are still unknown to me. Still, being gone through all of this provides us all the knowledge we need to understand CGEN LLVM-IR, what extensions I made and how I designed the translators CGEN LLVM-IR can generate.

Figure 6: "Over every mountain there is a path, although it may not be seen from the valley." *(Theodore Roethke)*

3 CGEN LLVM-IR

3.1 Introduction to CGEN LLVM-IR

In this section I am presenting CGEN LLVM-IR, its goal and a high level overview of the components required to achieve such goal.

Goal The goal of CGEN LLVM-IR is to provide a software to generate binary translators. A binary translator is a program that takes a binary executable as input and outputs a semantically-equivalent translation of that program in another language. In our scenario, the output language is LLVM-IR[14]. The choice is driven by the already available set of LLVM compiler backends to target a wide set of architectures, but it could also be decompiled into higher level languages to restore the source code of that program or even a "porting" to another language. A lot of things are possible once we have a working LLVM-IR translation, so let us move toward it.

Overview In order to achieve such ambitious goal we can sketch an outline of the required components and their logical functionalities for the **binary translators** to generate:

Architecture emulation Programs are compiled to run on an architecture that in general provides a set of memory locations (registers). Because we want to translate it into another language we need to emulate the presence of such registers. So the idea was to allocate a set of LLVM-IR global variables to mock general purpose registers, program counter and CPU status registers. This way any instruction working on memory elements would interact with logical variables instead.

Disassembler We are targeting binary programs as input. Therefore we need to disassemble the binary input stream to reconstruct in memory what are the assembly instructions and their operands.

Translation Through LLVM framework, emit LLVM-IR code that mimics the semantic of each istruction and reads/sets our *mock registers* correctly.

So far so good. This was done many times in the past and decompilers such that already exist. There is something to mention, though: most of the decompilers available today are coded fixing the input architecture and the output language. For every architecture we want to support, we must build from scratch a new decompiler. Wouldn't it be great to have a way to simply *describe the architecture in a problem-agnostic way* and have a software to generate such decompiler for us? Now that sounds interesting. Our idea was then to pickup GNU CGEN and its architecture agnosticity and extend it to generate decompilers for a given input *cpu description*.

[14] https://llvm.org/docs/LangRef.html

3.2 CGEN LLVM-IR components

From a component level perspective this is how CGEN LLVM-IR was designed

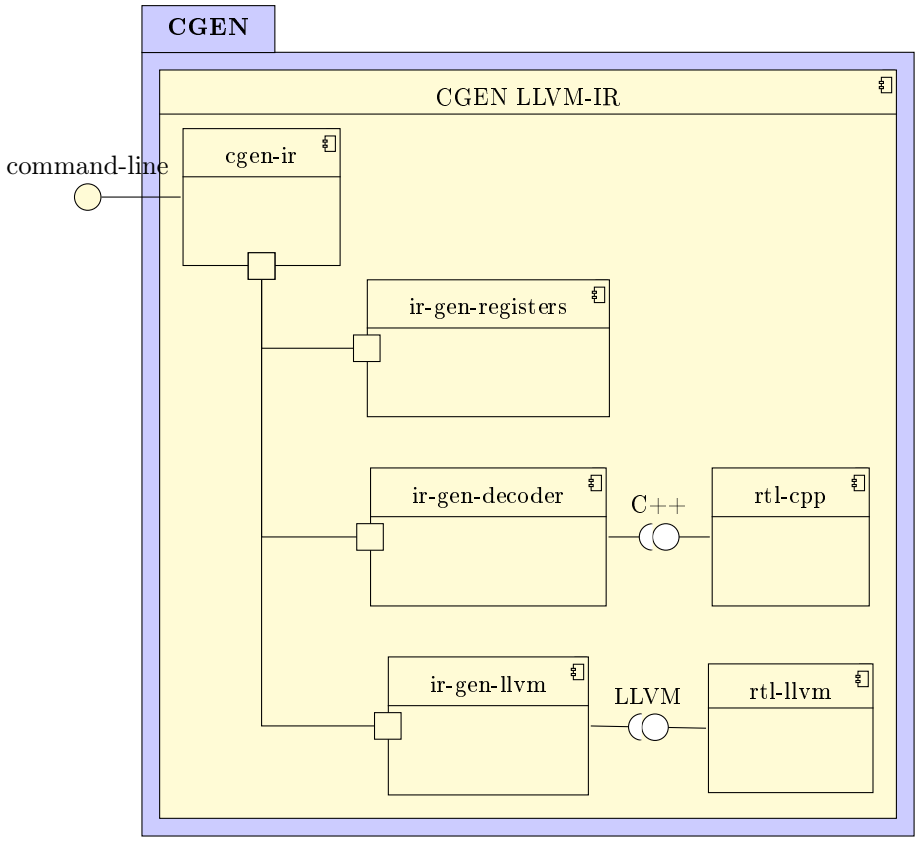

Figure 7: Component diagram of CGEN LLVM-IR

cgen-ir The application entry point. It accepts command-line parameters to drive the execution of the *ir-gen-** components.

ir-gen-registers Generates a C++ source file exploiting LLVM framework to allocate LLVM-IR global variables that emulate the input architecture registers.

ir-gen-decoder Generates C++ sources to define C++ classes that internally represent the assembly instructions for the input architecture and their operands. More details on the generated translator design in section 3.3.

rtl-cpp Translates RTL into C++ code. This is the lowest level of translation, where each RTL instruction is translated into C++ code that executes the RTL instruction semantics. E.g. `(add SI (const SI 1) (const SI`

`2))` is translated into a template function call `Add(1, 2)` which actually computes the *sum between 1 and 2*.

ir-gen-llvm Generates C++ sources to implement those functions that emit LLVM-IR code semantically equivalent to a given assembly instruction for the input architecture. **This component is not implemented yet**.

rtl-llvm Generates C++ code that emits LLVM-IR code that is a translation of input RTL. This is a higher level of translation, because a given RTL expression `e` has to be translated into code that generates code semantically equivalent to `e`. **This component is not implemented yet**.

3.3 Translators design

Another design factor of CGEN LLVM-IR is how the generated translators are structured. During the early stages of development I had to go through a huge amount of C code generated by GNU CGEN for simulators to learn and understand which path could be followed in my project. It was a nightmare for sure. C is very far from being self-expressive, even less if it is machine generated.

In my project I wanted generated C++ sources to be as human-readable as possible. C++ allows to structure your code in an object-oriented fashion, introducing abstractions to let the programmer stay sane. In addition I was not sure about the successful outcome of CGEN LLVM-IR and I forsaw that I would have had to edit the generated translators manually to test the many ways of doing things. I was, and still I am not, an expert in managing such low-level data so I needed a solution to abstract details away and encapsulate them in higher level components.

The class architecture I came up with is shown in figure 8.

Hereinafter I am providing a rationale that drove this design.

Instruction::make() A *factory method* to instantiate an implementation of Instruction. Let us set the following scenario: In disassembling a binary program all you have is a stream of bytes. Say you know the instruction length, `L`. You take the first `L` bytes of the stream that form a `word`. Now you check the first bytes of the `word` to understand what instruction that `word` is, say it is an `AddInstruction`. Ok, `AddInstruction` is dissassembled doing this and that. Now repeat. Next instruction is a `LoadInstruction`. Oh, I have to disassemble it in that other way. And so on and so forth.

I see a lot of room for *Strategy pattern* here. How about having a *factory method* that only checks the first bytes of a `word` to understand what instruction that `word` represents and *delegates* the decoding to the corresponding class? And here it comes `parseSfmt()`.

parseSfmt() A *strategy method*, `virtual` in the base class and implemented by each concrete `Instruction` derived class. Its purpose is straightforward: take the instruction `word`, disassemble it and fill the class members with

the decoded values. This is possible because when `parseSfmt()` is invoked we know what assembly instruction those bytes represent.

wordType trait The type of an instruction word (e.g. `uint32_t`).

chuckType trait The type of a word chuck (e.g. `uint16_t`). Some architectures require a word to be read from byte stream at chunks of smaller size.

endianness trait The endiannes of the architecture.

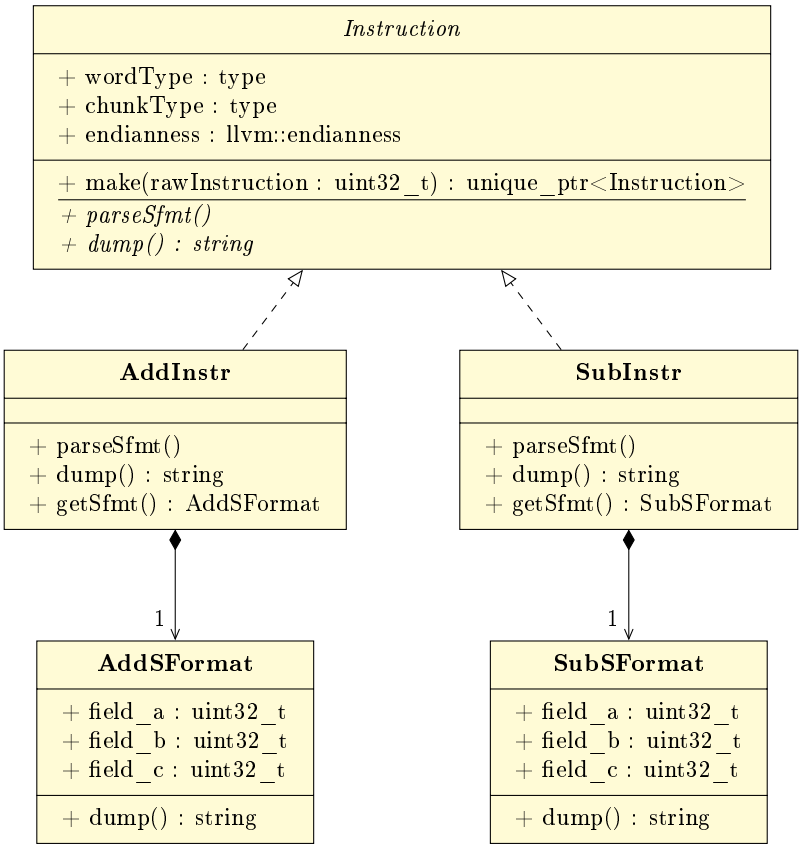

Figure 8: Translators design - Instructions and their formats

3.4 CGEN-IR commons

In order to support the translators execution a I wrote the `cgen-ir-common.h` source file to define those functions common to all generated translators. They support the input binary reading and the decompilation step by means of `template functions`.

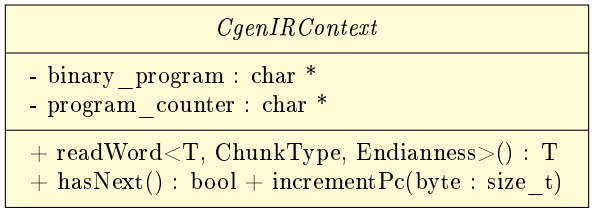

Figure 9: CGEN LLVM-IR translation context

3.5 Where to go from here?

At the moment of writing the development of CGEN LLVM-IR stopped at a working disassembling feature. Possible future steps could be implementing the `ir-gen-llvm` and `rtl-llvm` components that are currently missing. They should follow the same structure of `ir-gen-decoder` and `rtl-cpp`, adding a new virtual method to `Instruction` class and implemented by concrete derived classes, let's say *generateIR()*.

Please refer to the CGEN LLVM-IR Git repository[15] for complete source code, how to run it and a sample of a working generated translator for ARC700 architecture[16].

[15] https://polimicg.org/gitlab/leonardo.arcari/cgen-llvm-ir-generator

[16] https://www.synopsys.com/designware-ip/processor-solutions/arc-processors/arc-700-family.html