2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Making Code Generation Favourable

Tero Isännäinen, Siruco, Espoo, Finland (tero.isannainen@siruco.com)

***Abstract*—Code generation is a method for increasing designer productivity. The generated code can be highly optimal regarding area, power, performance, and correctness. Code generation is comparable with having an enormous collection of IP blocks available. The prerequisite for code generation is repeating patterns in the designs. Code generation becomes favourable when opportunities are identified and the code generator tool design is well supported. This paper identifies opportunities for code generation and outlines various methods for making code generator tool development practical.**

***Keywords*—code generation; automation; compiler; register transfer level; domain specific language**

I. Introduction

Engineers work hard in order to avoid hard work. Code generation falls into this category. While code generation can be understood broadly and applied to generate any type of code, this paper concentrates in synthesizable RTL code generation.

Code generation means, in the context of RTL design, that RTL is produced automatically using a computer program based on a description. While the concepts and techniques presented in this paper address primarily RTL design, they are applicable to other kinds of code generation targets. Code generation for verification purposes is one example.

Large Language Model (LLM) based code generation is not considered. The current results are modest. After a significant effort performed on LLM based generator improvements, 64.4% of the generated RTL code compiles [1]. The designs are small single feature blocks, where as this paper aims for complete, production ready RTL blocks.

II. Theory

On high level, code generation is the process of turning the target description into the target, i.e., a transformation from *source* (input) to *target* (output).

The *target* (output) is, by definition, a program. Programs are written in programming languages. The *source* (input) is also a program. However, the source language is not a regular language, but a Domain-Specific Language (DSL) [2]. Code generators use, in general, their own highly specialized DSLs.

Language processing is divided into several phases (See: Figure 1).

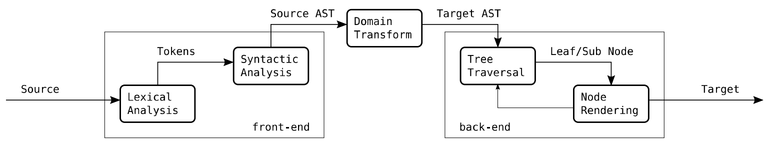

Figure 1. Language transformation.

The first phase is *lexical analysis*. Lexical analysis means that the program text characters are grouped into tokens [3]. Tokens are classified into different types: punctuation, keywords, operators, identifiers, etc.

Tokens are organized into a legal structure, defined by the language grammar, in the *syntactic analysis* phase. The resulting structure is called an Abstract Syntax Tree (AST) [3]. We'll use the term *parsing* to cover both *lexical analysis* and *syntactic analysis* phases.

For example, an arithmetic expression *(a + 3) * b*, would be converted to a sub-tree (See: Figure 2).

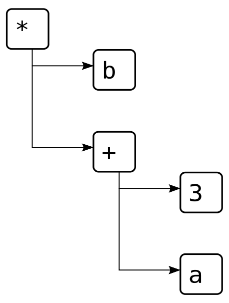

Figure 2. Arithmetic expression.

AST is traversed typically in depth-first ordering [3]. When the leaf level is reached, evaluation and other processing can be performed. Processing continues upwards. In Figure 2, the processing would start from node “a” and continue towards nodes “3” and “+”. Tree structure has the property, that any precedence and operator association rules a language might define, are removed and replaced by the explicit semantics of the tree structure.

Syntactical analysis is followed by the Domain Transformation. This is the main part of code generation. All essential information about the target is generated. Domain Transformation starts with semantic analysis of the source AST. Semantic checks are DSL specific. A common semantic check might consist of checking that two different parameters do not share the same name. Assuming the source AST is valid, the target AST can be generated. Typically the source description is compact and the implementation needs to elaborate the input data. The elaborated data is organized to a data structure that supports the generation of the target AST.

When the target AST is completed, the final target can be created. The target AST is traversed from top to bottom. The leaf level elements are rendered to the format of the target language. Leaf level elements are numbers and variable references, for example. The next higher level nodes are, for example, arithmetic expressions, assignments and IF statements. In order to get human readable output, the target renderer keeps track of the line indentation and folds statements on different lines, in order to produce human readable output. The phases in Figure 1 are fairly generic and therefore widely applicable for all kinds of code generators.

III. Opportunities for code generation

In general, we want a proper Return On Investment (ROI) when deploying code generation. Code generator is an investment and there should be a clear path for gaining benefits. Code generator benefits are mostly improved productivity in implementation and verification. However, in some complex cases the generated target could be better (in performance, power, area) than the manually created, since the manual effort could be limited by schedules.

Assuming that code generator pays itself back when target is generated at least 10 times, there are a smaller number of opportunities. But, if we take the number down to 3, for example, it is clear that the number of code generation opportunities is significantly more than the 10-to-3 ratio. Proper code generator tool development infrastructure will ensure substantially more code generation opportunities, because it lowers the payback threshold.

Code generators produce high quality and bug free RTL, after the generator itself is properly verified. Part of the verification for the code generator comes for free, since certain classes of programming errors are directly captured by just compiling the generated code. Any bug fix to a code generator will be visible in all related future outputs. The verification side benefits might be very significant for generated code. Therefore, design implementation productivity improvements should not be the only consideration when evaluating the feasibility of code generators. Less RTL bugs means less debugging and design iterations.

The code generation opportunities arise when there is repetition of some pattern. Repetition might occur over multiple projects, over multiple design units or within a design unit. For example, a well defined memory map description can be used for RTL code generation and various other targets. This expands to multiple projects, since most designs have a memory map. Control and Status Register (CSR) modules can be created and they appear in multiple different design units. Within the CSR modules, the same register types are repeated (read-only, read-write, etc.).

While many code generators would use their own DSL and the DSL would be created explicitly, there are also cases where code generator source can be taken (almost) directly from the design documentation. If a design portion is documented using lists, tables or tree-like structures, the format is easy to parse and code generation may become feasible. When a design is described as mentioned, there is likely a substantial amount of internal repetition. Additionally, it is typical that the specification will change multiple times over the course of the project and each RTL update will benefit from code generation.

IV. Case analysis

This chapter includes examples of three independent code generators. They have different characteristics in how often they can be applied and how difficult they are to create. Also, there are differences in their internal structure.

The syntax used in the examples is from the Scheme programming language [4]. Scheme belongs to the Lisp family of languages. In Scheme, data and code have the same syntax (homoiconicity). Scheme is well suited for language transformations. However, other dynamic languages (Python, Ruby, etc.) could also be used. Appendix A contains a condensed introduction to Scheme.

A. Parametrized counter

This case is a reduced example. The emphasis is on the ability to show more details of the internal phases than to present a realistic example.

For the target description, we are bypassing parsing and represent the counter directly as “Source AST”. The technique is called Embedded DSL (EDSL) [2]. Using EDSL simplifies the implementation of the first working version significantly. EDSL specializes the host language to an DSL, and parsing occurs directly and automatically within the host language environment.

Assuming that “Tree Traversal” and “Node Rendering” (see: Figure 1) exist already as libraries, the only missing component is the code generator specific Domain Transformation. While EDSL allows for semantic checking of the input and some level of error reporting, the error reporting is more accurate if full parsing is performed for the input. This can be added (i.e., by implementing proper parsing) to the “final” version of the code generator, if necessary. The addition is unobtrusive and would not affect the other phases.

We use only one parameter for the counter: count *range*.

```
(counter basic
         (range 18))
```

The description above indicates that the design is a *counter*. The name of the module is *basic* and the counter counts *18* steps. Obviously, the counter generator could support any number of parameters: count direction selection, restart control, early ready-indication, count value output, etc. These would be just added to the list of parameters.

In order to create the Target AST, the Domain Transformation component needs to include the base AST template for a counter module. Additionally, it needs to calculate the bit width of the counter variable (5 bits in this case). Counter variable *width* and *limit* are populated to the template.

Appendix B contains listings for the counter design. The Target AST format is language independent and it could be rendered to Verilog, SystemVerilog or VHDL. Verilog is chosen here.

For language independence, the AST abstraction level should be somewhat higher than the direct parsing result (thinking backwards from Target Verilog to Target AST). Here different types of ports are separated, especially clock and reset. Variables are collected to a list for attribute lookup. Sequential variables contain their reset values and all variables contain their bit widths. For example in:

```
("count" sync (0 unsigned 5 10))
```

The variable is called *count*. It is a sequential variable (*sync*) and it has a reset value of *0*. In detail, the specification defines that the value is *0* (in decimal), type is *unsigned*, bit width is *5*, and the base is *10*.

In summary, the Target AST abstracts module level information and the repeated process sub-structures (e.g., the reset if-branch). The behavioral statements have one-to-one correspondence to the direct parsing results.

B. *Hierarchy module generator*

RTL modules can be classified as Functional Modules and Hierarchy Modules. Functional Modules contain behavioral processes, which define the logic functions. Hierarchy Modules are defined here as pure. Hierarchy Modules only instantiate other modules and connect signals between the sub-modules.

When Functional Specifications are created for the design, it is fair to assume that most of the signal names match by default. In particular, it can be assumed that output port names match the names of the corresponding input port names. Here, the repeated pattern is not a specific design part, but the rule for making connections.

Let’s assume that the names match for 90% of the connections. The rest 10% includes: IP block ports, legacy blocks, multiple instantiations, generated code, and various other cases.

For new design units, it is possible that a significant amount of ports (close to 100%) have matching names. If all names match, the only missing information is the module instantiations.

The algorithm for signal connections and sub-module instantiations:

- Hierarchies are generated bottom up.
- The currently generated level instantiates named sub-modules.
- Each output port (also pin) of each sub-module is matched with one or more input ports of other sub-modules, and appear as wires.
- Unconnected sub-module input and output ports become respective ports of the generated module.

Figure 3 shows a simple hierarchy module example. *mod_a* and *mod_b* are the sub-modules and the generated module is *top_a_b*. Code generator parses Verilog (or VHDL) of *mod_a* and *mod_b*. Note that only port and parameter information requires parsing. Code samples are listed in Appendix C.

This hierarchy could be described as:

```
(hier top_a_b mod_a mod_b)
```

hier is a keyword and the name of the generated Hierarchy Module is *top_a_b*. The instantiated modules are *mod_a* and *mod_b*, and the instance names are the same as the module names (as a default).

Let’s change the setup and define that *mod_b* input port *a_to_b* is called *b_i1*. Also, we want a specific instance name for *mod_b*. With these changes, the hierarchy could be described as:

```
(hier top_a_b
      mod_a
      ((mod_b i_mod_b)
       (b_i1 a_to_b)))
```

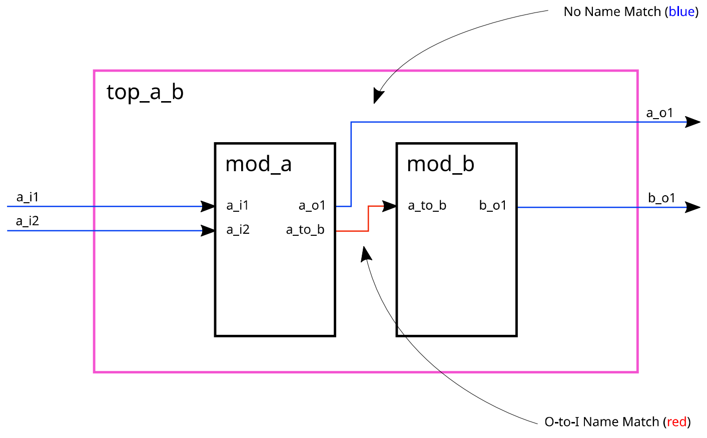

Figure 3. Simple hierarchy.

Now *mod_b* is instantiated as *i_mod_b*. Input port *b_i1* is mapped as *a_to_b* for the higher level and output port *a_to_b* of *mod_a* can be matched to it by name.

Overall, this type of hierarchy generator tool should support flexible port name mapping (glob/rexexp pattern based), output port branching (routing signals in current level as wires and also upwards as ports), and parameter management. Typically module parameters would be set in the hierarchy definitions, resolved, and numeric port widths would be used in the higher level modules. The tool should also perform checking on port widths and bit ordering, and it would be an error to connect signals with the same name, but different bit configuration.

It is estimated that hierarchy modules contain around 20-40% of all the design code lines. For example in [5], there are 28% of the lines (effective lines) in hierarchy modules. Note that not all hierarchy modules are pure in this reference.

The content of hierarchy modules change significantly, when new modules are added and existing modules are updated. Hierarchy Modules require numerous updates during the course of the development. In general, code generator execution speed is not relevant. The priority is mostly in optimizing the code generator creation and maintenance effort. However, since every RTL design requires hierarchical modules and they change frequently, these are valid reasons for optimizing the tool performance.

For example, when creating a very large Hierarchy Module, the tool performance can become an issue (Appendix C). A benchmark run for a large design indicates significant differences in performance. A test hierarchy including two sub-modules was investigated. Both modules had 10000 input ports and 10000 output ports. All outputs of module A were name matched to the inputs of module B. Therefore, module A inputs were used as inputs for the generated module, and module B outputs where used as outputs.

When the hierarchy module is generated with a dynamic language program (Scheme in the benchmark), the code generation runs in ~35 s. When the same hierarchy module is generated with a well designed C program, the code generation runs in ~20 ms. Hence, the C implementation is over 1500 times faster. Both versions consume approximately half of the runtime parsing the two Verilog modules, and the rest of the time is used for matching port names and writing out the result.

The high level algorithm is the same for both implementation. For example, both versions use string hashing for fast name matching. The lower level implementation details are significantly different. Dynamic languages perform many operations during runtime that compiled languages perform at compile time [6]. Every access to a primitive data type in dynamic languages include execution of many CPU instructions, which is not needed for compiled C programs. The C version manages all memory resources carefully, while the Scheme version uses garbage collection. Garbage collection is generic and runs without guidance from the program. No claims are made about how fast the Scheme version could be. Both versions have been created using idiomatic practices of the respective languages.

C. Finite Impulse Response filter generator

Finite Impulse Response (FIR) filters include a repeated pattern in their main structure. One of the main parameters for FIR filters is the order (direct form FIR). The order is defined by the number of delay elements in the filter (see: Figure 4).

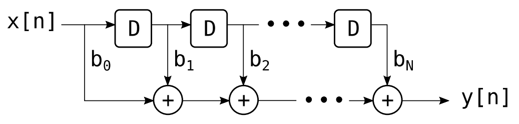

Figure 4. FIR filter in direct form.

When filter order is N, it has N+1 taps (coefficients). FIR performs a running, weighted average of the input samples. Input sample is scaled and summed with the scaled versions of the previous samples. Sample delays are represented with *D* and scaler coefficients are represented with the indexed variable *b*. The repeated pattern is the delay-coefficient-sum section.

It is fairly common that FIR filters are designed to be linear phase. A FIR filter is linear phase when coefficients are symmetrical to the center coefficient (odd count) or to the two center coefficients (even count).

Appendix D contains the FIR description and the resulting RTL code.

The FIR generator parameters:

- *iport*: name and width of the sample input port.
- *oport*: name and width of the sample output port.
- *sample-period*: period of input and output samples in clock cycles.
- *coeff-width*: bit width for coefficients.
- *coeff-float*: floating point or integer coefficients.
- *coeff-symmetry*: none, even, or odd count.
- *coeff-list*: list of coefficients.

Odd and even *coeff-symmetry* means that the total coefficient count is odd or even, respectively. With odd *coeff-symmetry*, the given coefficients are repeated excluding the last coefficient. With even *coeff-symmetry*, the given coefficients are repeated including the last coefficient.

This code generator uses full precision in the internal calculations and saturates the output at overflow. The code generator optimizes the number of multiplication and addition resources by taking advantage of the *sample-period* parameter. For example, in Appendix D the *sample-period* is 8. The RTL has 8 clock cycles for producing one output sample. The number of required multiplication and addition units become therefore the ratio of coefficients (47) to sample period (8). In this case, we have 6 (47 / 8 => 6) multiplication and addition units.

The filter description is a "high level" description. It can be used to parameterize a reference model, which is then used for verification of the generated RTL. The description contains 14 lines of text and the target RTL (full version) contains 399 lines. The description is easy to maintain. Experimentation of RTL port bit widths (and the related calculation precision), for example, involves little effort. The code generator eliminates many classes of potential errors related to manual RTL creation. For example, it is quite tedious work to update the different bit widths for variables and intermediate results. The Domain Transformation part of the FIR code generator is 174 lines of code.

The code generator includes the following automation features:

- Calculation of bit ranges for all variables.

- Duplication of coefficients (with odd or even symmetry).
- Quantization of floating point coefficients to selected precision.
- Indexing of delayed samples and coefficients.
- Optimization of arithmetic resources.
- State machine for calculation phase control and related calculations.

V. Code generator tool development

Code generators are developed in reverse, from target to source. The target should contain repetition that is not feasible to manage using other means such as IP blocks or parametrized RTL. The wider the target domain (variation in outputs), the more parameters are needed in the source DSL. The term *parameter* should be understood broadly in this context. The source DSL is descriptive in nature, and therefore the term *parameter* suites well for describing the control of the target output.

It is important to start small and get the first prototype code generator working early. The source DSL design is typically an iterative process, and it is not easy to find the correct set of parameters immediately. We are balancing between the effort in creating the code generator and how widely it is applicable. When starting small, less iterations are likely required or at least the overall amount of work is minimized.

Organizations benefit from preexisting libraries for front-end and back-end (See: Figure 1). Front-end libraries are needed, for example, for parsing documentation originated information sources. The selection of exact formats, depends on what format is desirable for the manual document updating (i.e., the master source).

Usually one back-end library is enough. If only one code generator implementation language is used and the target language remains the same, a single library is sufficient. For every code generator tool, the Domain Transformation component needs to produce target DSL data that is compatible with the back-end library.

The back-end libraries should support hooks (callbacks). While the most interesting design information goes through the code generation flow, there are sometimes a need for side streams. For example, we might want that a specific standard comment header exists in each RTL source code file. It does not make sense to capture this changing and somewhat arbitrary information in the source DSL. Instead, the registered hooks can insert this type of information to the output.

As a minimum for the first prototype, we need the source DSL with some parameters and the Domain Transformation. The source DSL can be implemented as an EDSL and therefore no new parsers are needed. If back-end libraries are available, there is no work involved for the back-end phase.

The first prototype is used in the project and feedback is collected. Based on the project feedback, DSL parameters can be updated. At this point it is feasible to evaluate the compromise between the target domain features and the tool complexity. When all code generator features have been matured, the front-end can be updated to include proper parsing of the source, which improves the tool usability.

In summary, the phases for creating a code generator tool:

1. Identify candidate target for code generation. Typically this relates to a repetitive patterns: internal, external, or both.
2. Define the number of parameters and parameter value ranges based on the amount of variation in the output targets. Often the current situation contain excessive and unnecessary variation, which should be removed. The reduction in variation improves the value proposition of the code generation tool.
3. Identify all targets than can be generated from the same set of input information and identify target specific information sources. Domain Transformation might require multiple inputs.

4. Design the code generator architecture around the information sources.
5. Implement and document the “minimum viable product” with future extensions taken into consideration. Start with a simple implementation.
6. Deploy the code generator to project use and improve based on the feedback.

VI. Conclusions

Code generation is the best solution for certain type of repeating design features. Some code generators are applied across different projects and some should be developed during a live project. For live project development, it is important that the first results can be delivered quickly. This enables wider application of the code generators.

The path to first results can be substantially shortened through the use of EDSL techniques and readily available back-end libraries. A small scale code generator is possible to develop in a few hours.

The initial generator can be improved, when the required feature set has stabilized. New DSL parameters can be added and existing updated. The usability of the code generator is improved by performing proper parsing of the source DSL. The tool is able to pin point input errors to the exact character (line and column) and the error message is clear about the violation, in user level terms.

Design organizations should be code generator aware. Designers should actively look for opportunities for code generation. Repetition might be implicit and therefore not immediately visible. The repetition could be in the applied rules and not directly in the design parts themselves.

Version control systems and tool flows should support the existence of generated source code files. In practice this means that some kind of meta data is maintained about the design files. The flow utilizes this data and performs the design file transformations automatically.

References

[1] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, Siddharth Garg, “VeriGen: A Large Language Model for Verilog Code Generation”, ACM Transactions on Design Automation of Electronic Systems, Jan 2024, https://doi.org/10.1145/3643681

[2] Domain-specific language. In Wikipedia. Retrieved June 5, 2024, from https://en.wikipedia.org/wiki/Domain-specific_language

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, “Compilers: Principles, Techniques, and Tools (second edition)”, Pearson Education, Inc, 2006

[4] Alex Shinn, et al. “Revised^7 Report of the Algorithmic Language Scheme”, 2021 https://standards.scheme.org/

[5] https://github.com/corundum/corundum/tree/master/fpga/lib/eth/rtl. Retreived June 5, 2024

[6] Dynamic programming language. In Wikipedia. Retrieved June 5, 2024, from https://en.wikipedia.org/wiki/Dynamic_programming_language

VII. Appendix A

Scheme is a language from the Lisp family. The language syntax is very simple. Language specification is only 80 pages [4]. The syntax consists of atoms and lists. Atoms are primitive data types like: numbers, strings, symbols. Lists contains zero or more elements. A list element is either an atom or another list (sub-list).

Lists are evaluated, by default, as expressions. The first element should be a procedure and the rest of the list elements are arguments for the procedure. Typically the procedure is referenced through a variable with a procedure value. However, Scheme has also lambda expressions, which are literal definitions for procedures (functions).

In addition to the default expression evaluation, Scheme has a few special forms. Special forms mean than the list expressions are not fully evaluated, as above. The *define* form is used to create a variable binding: "(define a 10)". *if* form evaluates either the TRUE or the FALSE sub-expression depending on the value of the condition. *lambda* treats the list elements as code and does not, for example, result to resolving identifier tokens to variable values. *lambda* form evaluation is compile time activity and *lambda* value (the procedure) application is runtime activity. *quote* prevents evaluation and passes it's arguments as literal values.

Almost all expressions result to a usable value in Scheme. In practice this means, that for example *if* expressions may appear almost anywhere in the code, also in the argument position of a procedure call.

Scheme is a *functional language* by nature, but variable value mutation is allowed and can be used when beneficial. Most Scheme code is based on composing a tree of procedure applications (i.e., function calls) and each application returns a fresh data value for the higher levels in the tree.

When Scheme code is "read", it is just parsed and the result is program data (nested lists). Program data can be easily manipulated. When Scheme code is "loaded", it is first parsed and then evaluated as program code. Program data can be evaluated as program code at will. This allows, for example, that a part of the input code is manipulated before all resulting code is evaluated.

```
;; Comments start with ";" and continue to the end-of-line.
14                           ; Number 14.
"hello\n"                    ; String ending with newline.
#t                           ; Boolean true.
#f                           ; Boolean false.
'hello                       ; Symbol (identifier) "hello".
=> hello
(cons 1 2)                   ; Pair of values.
=> 1 . 2
'(1 2 foo)                   ; List (quoted).
=> (1 2 foo)
(list 1 2 'foo)              ; List using "list" procedure.
=> (1 2 foo)
'()                          ; Empty list.
=> ()
(cons 1                      ; List using nested pairs.
      (cons 2
            (cons 'foo
                  '())))
=> (1 2 foo)
(+ 1 2)                      ; Expressions are lists with a procedure as the first element.
=> 3
+                            ; "+" is a variable with a procedure value.
=> #<procedure +>
(quote +)                    ; Quoted variable.
=> +
'(+ 1 2)                     ; Quoted list.
=> (+ 1 2)
(define n 12)                ; Variable definition with value 12.
(lambda (a b) (+ a b))       ; Procedure value.
(define my-add               ; Variable definition with a procedure value.
  (lambda (a b) (+ a b)))
(define (my-add a b)         ; Same as above using compact syntax.
  (+ a b))
(my-add 1 (my-add 2 3))      ; Nested expressions: 1 + (2 + 3).
=> 6
`(1 ,(my-add 2 3))           ; Quasiquoted list with evaluated sub-expression.
=> (1 5)
(if #t "yes" "no")           ; (IF <cond-expr> <true-expr> <false-expr>).
=> "yes"
(if (not #t)                 ; If expression reduces to <false-expr> value.
    4
    (+ 1 2))
=> 3
```

```
(define (factorial n)        ; Define procedure "factorial".
  (define (sub n res)        ; Local procedure "sub" as tail-recursive.
    (if (= n 0)              ; Terminate recursion when "n" is zero.
        res                  ; "sub" returns value of "res".
        (sub (- n 1)         ; Recurse with tail-optimization, i.e., no stack pushed.
             (* n res))))    ;   New call replaces the old stack frame.
  (sub n 1))                 ; Result: apply "sub" to initial "n" and to initial result 1.
(factorial 5)                ; Apply "factorial" to "5".
=> 120
(let ((n 10))                ; Create local variable "n" with value 10.
  (if (= n 12)               ; This is different variable from the outer scope "n".
      "yes"
      "no"))                 ; Return "no", since inner scope "n" is 10.
=> "no"
(even? 3)                    ; Predicate procedures end with "?", by convention.
=> #f
(map even? '(1 2 3 4))       ; Map applies (calls) "even?" for each element.
=> (#f #t #f #t)             ; Return value is a new list with the same dimensions.
(let lp ((lst                ; named-let: implicit, named procedure "lp".
          '(1 2 3 4))        ; "lst" starts as the original list.
         (ret '()))          ; Return list starts as empty.
  (if (pair? lst)            ; "pair?" returns #t if list is not empty.
      (lp (cdr lst)          ; Recurse with list head removed, i.e., with tail/rest.
          (cons (even?       ; Add #t/#f to head of return list.
                 (car lst))  ; car=head, cdr=tail.
                ret))
      (reverse lst)))        ; Return list in correct order.
=> (#f #t #f #t)             ; Same result as from "map".
```

VIII. Appendix B

This appendix contains listings for the “parametrized counter” example.

Counter description:

```
(counter basic
         (range 18))
```

Target AST:

```
((name "basic")
 (clock "clk")
 (reset "rst")
 (iports "enable")
 (oports "ready")
 (vars ("clk" port (#f unsigned 1 2))
       ("rst" port (#f unsigned 1 2))
       ("enable" port (#f unsigned 1 2))
       ("ready" sync (0 unsigned 1 2))
       ("count" sync (0 unsigned 5 10)))
 (procs (sync ("ready" "count")
              (stmt-ass
                (varref "ready")
                (number (0 unsigned 1 2)))
              (stmt-if
                ((varref "enable")
                 (stmt-if
                   ((op-eq (varref "count")
                           (number (17 unsigned 5 10)))
                    (stmt-ass
                      (varref "count")
                      (number (0 unsigned #f 16)))
                    (stmt-ass
                      (varref "ready")
                      (number (1 unsigned 1 2))))
                   (#f
                    (stmt-ass
                      (varref "count")
                      (op-add (varref "count") (number (1 #f #f 10))))))))))))
```

Target Verilog:

```
module basic
  (
   clk,
   rst,
   enable,
   ready
   );
   input  clk;
   input  rst;
   input  enable;
   output ready;
   reg    ready;
   reg    [4:0] count;
   always @( posedge clk or negedge rst ) begin
```

```
      if ( !rst ) begin
         ready <= 1'b0;
         count <= 5'd0;
      end else begin
         ready <= 1'b0;
         if ( enable ) begin
            if ( count == 5'd17 ) begin
               count <= 'h0;
               ready <= 1'b1;
            end else begin
               count <= count + 1;
            end
         end
      end
   end
endmodule
```

IX. Appendix C

This appendix contains listings for the “simple design” and “very large” examples.

Simple Design hierarchy description:

```
(hier top_a_b
      mod_a
      mod_b)
```

Simple Design Verilog modules:

```
module mod_a
  (
   input  a_i1,
   input  a_i2,
   output a_o1,
   output a_to_b
   );
   assign a_o1 = !a_i1;
   assign a_to_b = a_i1 && a_i2;
endmodule

module mod_b
  (
   input  a_to_b,
   output b_o1
   );
   assign b_o1 = !a_to_b;
endmodule
```

Simple Design generated hierarchy module Verilog:

```
module top_a_b
  (
   a_i1,
   a_i2,
   a_o1,
   b_o1
   );
   input  a_i1;
   input  a_i2;
   output a_o1;
   output b_o1;
   wire   a_to_b;
   mod_a mod_a
     (
      .a_i1( a_i1 ),
      .a_i2( a_i2 ),
      .a_o1( a_o1 ),
      .a_to_b( a_to_b )
      );
   mod_b mod_b
     (
      .a_to_b( a_to_b ),
      .b_o1( b_o1 )
      );
endmodule
```

Very Large hierarchy description:

```
(hier top_a_b
      mod_a
      mod_b)
```

Very Large Verilog modules (reduced from 2*40011 lines):

```
module mod_a
 (
 clk,
 rst,
 input_port_0,
 input_port_1,
 ...
 input_port_9999,
 signal_port_0,
 signal_port_1,
 ...
 signal_port_9999
 );
 input  clk;
 input  rst;
 input  input_port_0;
 input  input_port_1;
 ...
 input  input_port_9999;

 output signal_port_0;
 output signal_port_1;
 ...
 output signal_port_9999;

endmodule

module mod_b
  (
   clk,
   rst,
   signal_port_0,
   signal_port_1,
   ...
   signal_port_9999,
   output_port_0,
   output_port_1,
   ...
   output_port_9999
   );

   input  clk;
   input  rst;
   input  signal_port_0;
   input  signal_port_1;
   ...
   input  signal_port_9999;

   output output_port_0;
   output output_port_1;
   ...
   output output_port_9999;

endmodule
```

Very Large generated hierarchy module Verilog (reduced from 90024 lines):

```
module top_a_b
  (
   clk,
   rst,
   input_port_0,
   input_port_1,
   ...
   input_port_9999,
   output_port_0,
   output_port_1,
   ...
   output_port_9999
   );

   input  clk;
   input  rst;
   input  input_port_0;
   input  input_port_1;
   ...
   input  input_port_9999;

   output output_port_0;
   output output_port_1;
   ...
   output output_port_9999;

   wire   signal_port_0;
   wire   signal_port_1;
   ...
   wire   signal_port_9999;

   mod_a mod_a
     (
      .clk( clk ),
      .rst( rst ),
      .input_port_0( input_port_0 ),
      .input_port_1( input_port_1 ),
      ...
      .input_port_9999( input_port_9999 ),
      .signal_port_0( signal_port_0 ),
      .signal_port_1( signal_port_1 ),
      ...
      .signal_port_9999( signal_port_9999 )
      );

   mod_b mod_b
     (
      .clk( clk ),
      .rst( rst ),
      .signal_port_0( signal_port_0 ),
      .signal_port_1( signal_port_1 ),
      ...
      .signal_port_9999( signal_port_9999 ),
      .output_port_0( output_port_0 ),
      .output_port_1( output_port_1 ),
      ...
      .output_port_9999( output_port_9999 )
      );

endmodule
```

X. Appendix D

This appendix contains partial results of the FIR code generator.

Filter description:

```
`(fir fir-tap-47
      (iport x 14)
      (oport y 14)
      (sample-period 8)
      (coeff-width 14)
      (coeff-float #t)
      (coeff-symmetry odd)
      (coeff-list -0.00100343   8.19935e-05   0.00138234   0.000969069
                  -0.00149488   -0.00287014   0.000214866  0.00497573
                  0.00359393    -0.00512623   -0.00958052  0.00048227
                  0.0147818     0.0104958     -0.0138518   -0.0259068
                  0.00076921    0.0392412     0.0292861    -0.0389579
                  -0.0819605    0.000952093   0.200343     0.373183
                  ))
```

Target Verilog (reduced from 399 lines):

```
module fir_tap_47
  (
   clk,
   rst,
   x,
   x_en,
   y,
   y_en
   );

   localparam b0 = 14'sh3ff8;
   localparam b1 = 14'sh1;
   ...
   localparam b23 = 14'shbf1;
   input  clk;
   input  rst;
   input  signed [13:0] x;
   input  x_en;


   output signed [13:0] y;
   output y_en;


   reg    signed [13:0] y;
   reg    y_en;
   reg    [7:0] x_en_d;
   reg    signed [32:0] sw;
   reg    signed [32:0] s;
   reg    use_alu;
   reg    signed [13:0] d0;
   reg    signed [13:0] d1;
   ...
   reg    signed [13:0] d46;
   reg    signed [13:0] mu0_i0;
   reg    signed [13:0] mu0_i1;
   reg    signed [13:0] mu1_i0;
   reg    signed [13:0] mu1_i1;
   ...
   reg    signed [13:0] mu5_i0;
   reg    signed [13:0] mu5_i1;
   reg    signed [26:0] mu0_o;
   reg    signed [26:0] mu1_o;
   ...
   reg    signed [26:0] mu5_o;


   always @( posedge clk or negedge rst ) begin
      if ( !rst ) begin
         y <= 14'sb0;
         y_en <= 1'b0;
         d0 <= 14'sb0;
         d1 <= 14'sb0;
         ...
         d46 <= 14'sb0;
         x_en_d <= 8'b0;
         s <= 33'sb0;
      end else begin
         y_en <= 1'b0;
         x_en_d <= { x_en_d[6:0], x_en };
         if ( x_en ) begin
            d0 <= x;
         end
         if ( x_en_d[7] ) begin
            if ( sw > 27'sh3ffffff ) begin
               y <= 14'sh1fff;
            end else if ( sw < 27'sh4000001 ) begin
               y <= 14'sh2001;
            end else begin
               y <= sw[26:13];
            end
            y_en <= 1'b1;
            s <= 33'sb0;
            d1 <= d0;
            d2 <= d1;
            ...
            d46 <= d45;
         end else begin
            s <= sw;
         end
      end
   end


   always @* begin
      mu0_o = 'sb0;
      mu1_o = 'sb0;
      ...
      mu5_o = 'sb0;
      sw = s;
      if ( use_alu ) begin
         mu0_o = mu0_i0 * mu0_i1;
         mu1_o = mu1_i0 * mu1_i1;
         ...
         mu5_o = mu5_i0 * mu5_i1;
         sw = s + mu0_o + mu1_o + mu2_o + mu3_o + mu4_o + mu5_o;
      end
   end


   always @* begin
      use_alu = 1'b0;
      mu0_i0 = 'sb0;
      mu0_i1 = 'sb0;
      mu1_i0 = 'sb0;
      mu1_i1 = 'sb0;
      ...
      mu5_i0 = 'sb0;
      mu5_i1 = 'sb0;
      case ( x_en_d )
        8'h1: begin
           use_alu = 1'b1;
           mu0_i0 = d0;
           mu0_i1 = b0;
           mu1_i0 = d8;
           mu1_i1 = b8;
           mu2_i0 = d16;
           mu2_i1 = b16;
           mu3_i0 = d24;
           mu3_i1 = b22;
           mu4_i0 = d32;
           mu4_i1 = b14;
           mu5_i0 = d40;
           mu5_i1 = b6;
        end
        8'h2: begin
           use_alu = 1'b1;
           mu0_i0 = d1;
           mu0_i1 = b1;
           mu1_i0 = d9;
           mu1_i1 = b9;
           mu2_i0 = d17;
           mu2_i1 = b17;
           mu3_i0 = d25;
           mu3_i1 = b21;
           mu4_i0 = d33;
           mu4_i1 = b13;
           mu5_i0 = d41;
           mu5_i1 = b5;
           8'h80: begin
              use_alu = 1'b1;
              mu0_i0 = d7;
              mu0_i1 = b7;
              mu1_i0 = d15;
              mu1_i1 = b15;
              mu2_i0 = d23;
              mu2_i1 = b23;
              mu3_i0 = d31;
              mu3_i1 = b15;
              mu4_i0 = d39;
              mu4_i1 = b7;
              mu5_i0 = 'sb0;
              mu5_i1 = 'sb0;
           end
           default: begin
              use_alu = 1'b0;
           end
         endcase
      end

   endmodule
```