Evaluation of the HDLRuby Hardware Description Language by implementing an 8-bit RISC Processor

Lovic Gauthier^{a,*}, Yohei Ishikawa^a

^aInformation System Course, National Institute of Technology, Ariake College
Higashi-Hagio 150, Omuta, Fukuoka 836-8585, JAPAN

*Corresponding Author: lovic@ariake-nct.ac.jp

Abstract

HDLRuby is a new hardware description language (HDL) based on Ruby created for improving the productivity of HW designers. This paper presents a study of the implementation from scratch with HDLRuby of an 8-bit RISC processor called MEI8. This implementation required only little effort and its code is more than twice shorter than the equivalent VHDL code. The resulting processor was mapped onto a Virtex7 FPGA where it ran at 100MHz and was estimated to run at 28MHz when implemented as a 0.5 μ m IC.

Keywords: Electronic, Hardware Description Language, Ruby Language, FPGA, IC.

1. Introduction

Register-transfer level (RTL) is the long de-facto model used for describing HW. Yet, the design productivity of RTL stagnating, huge efforts have been spent to create new methods for synthesizing HW from more and more abstract representations. Yet, these methods are still limited, and in the end, designers still heavily rely on RTL synthesis. Thence, the time-to-market requirements pushed to adopt processor-centric devices even though their energy and power efficiency is much poorer than pure HW^(1,2).

When observing the academic and industrial works for improving HW design, it can be noticed that the main goal has been to use more and more SW-like models for describing HW. This culminates with High Level Synthesis (HLS)⁽³⁾ that tries to synthesize HW directly from SW code. The idea of getting closer to SW is indeed attractive, but it proved to be difficult to synthesize efficient HW using SW-oriented models of computation⁽⁴⁾.

Conversely, even though the design productivity of SW increased a lot, its model of computation remained mostly unchanged, based on the sequential execution of imperative instructions. Therefore, it can be advocated that the choice of model of computation may not be relevant for improving the productivity. In this context, we proposed HDLRuby^(5,6), an RTL-based HDL built upon the Ruby programming language⁽⁷⁾ for including paradigms independent of the model of computation that are efficient in SW design. Namely, we focused on the followings: object-orientated programming, genericity, metaprogramming and reflection. It must be noticed that the proposed approach is orthogonal with the traditional ways of abstracting HW, and that HDLRuby can be extended to support HLS-like algorithms.

The goal of this paper is to evaluate the benefits of using HDLRuby for designing from scratch a full-fledged circuit. For that purpose, the paper presents the implementation from scratch of a full processor with HDLRuby, compares its complexity with the corresponding VHDL implementation, evaluates the effort that has been required for obtaining the final result, checks its validity on an FPGA and an IC targets and estimates the final design productivity in gates. The processor is called MEI8 and is an 8-bit RISC Harvard processor including 8 general purpose registers, 37 different instructions and 2 external interrupt ports. In the present implementation, the instruction memory is an on-chip 256-bytes ROM. More details about this processor, and the corresponding source code are available online⁽⁸⁾.

The rest of the paper is organized as follows: section 2. presents some related works, section 3. presents the HDLRuby language, and section 4. details the study and its result and gives a discussion about its significance. Finally, section 5. concludes the paper.

2. Related Works

A main improvement in HW design has been the adoption about 25 years ago of the RTL model of computation⁽⁹⁾. While successful, RTL design is still much more time consuming than SW design. For this reason, tremendous efforts have been spent for improving further the design productivity of HW. Among these efforts we can cite: the early works on behavioral synthesis⁽¹⁰⁾, trying to synthesize HW from clock-free sequential code, component-based design^(11,12) generating wrappers for low-level HW/SW components allowing easy composition of complex systems, the introduction of SW-based HDL like SystemC⁽¹³⁾ or SpecC⁽¹⁴⁾, and the recent efforts for generating HW directly from SW code, like HLS approaches⁽³⁾ or HW synthesis from Matlab models⁽¹⁵⁾. Most of these approaches have in common that they try to synthesize HW using a model of computation closer to SW than to HW, which proved to be difficult in practice⁽⁴⁾. By contrast, with HDLRuby we do not try to change the model of computation, but instead we focus on improving the quality of the RTL code.

A few approaches are closer to ours. Some works tried to introduce object-orientation to HW design. For instance, SystemVerilog⁽¹⁶⁾ and SystemC⁽¹³⁾ include classes and high-level control constructs but limited to data types or non-synthesizable code. SystemC is also remarkable from being implemented on top of the C++ SW programming language. More recently, P. Tomson⁽¹⁷⁾ presented the draft of a HDL based on the Ruby language. However, this latter language did not evolve past the proof of concept.

There is little work about evaluating the productivity of HDL. The usual approach is to provide metrics like the number of gates produced per designer a day^(1,2). Such metrics are however product-dependent and give reliable results only when applied on a wide range of designs. For estimating a single design as it is the case of this paper, more relevant metrics can be found in SW design, e.g., source lines of code (SLOC), cyclomatic complexity⁽¹⁸⁾, or code churn⁽¹⁹⁾. Section 4.1 gives more details about the metrics considered for this paper.

3. Presentation of the HDLRuby Language

3.1 The Core of HDLRuby

HDLRuby⁽⁵⁾ is a HW description language based on the Ruby⁽⁷⁾ programming language. A preliminary version of

this language has also been presented in 2018⁽⁶⁾. The goal is to increase the productivity of the HW designers by adapting to HW successful SW paradigms not or only partially used in existing HDL like VHDL or Verilog HDL, while ensuring that the language remains synthesizable register transfer level (RTL). The main paradigms imported to HDLRuby are the followings:

- **object-oriented programming:** the elements of a description are considered as objects, i.e., collections of attributes and methods (algorithms), that interacts through messages.
- **generic programming:** elements of a description can be parameterized and reused in different contexts.
- **reflection:** the elements of a description can examine and modify their own structure and behavior.
- **metaprogramming:** code can be treated as data and be generated during execution, in our case, portions of code can be used as parameters for other elements of the description.

For that purpose, HDLRuby has been designed as a two-level language: a high-level generative language, used by the designer, whose execution produces a low-level set of data structures representing RTL constructs. The high-level generative layer is implemented on top of the Ruby programming language. In term of syntax, HDLRuby includes all the syntactic construct of the Ruby language, with additional ones for handling HW-oriented descriptions. These new constructs include for instance HW-specific literals (e.g., the "Z" state), and new constructs for describing signals, processes, instances or modules.

Synthesizability of the language is ensured by keeping an RTL model of computation, while the productivity-oriented features act only at the generative level, similarly to the implementation of object-orientation and genericity in the C++ language. In details, the SW paradigms were adapted to HW as follows:

- **object-oriented programming and reflection:** all the elements of an HW description are Ruby objects, and therefore include standard as well as reflection-oriented methods. However, these methods are programs that generate RTL code and not programs to be executed by the final circuit.
- **generic programming and metaprogramming:** they are inherited by construction from the underlining RTL code generation engine implemented in Ruby.

<pre> 1 system :sreg do typ 2 typ = typ.is_a?(Type) ? 3 typ : [*typ].to_type 4 5 input :clk, :rst 6 typ.base.input :d 7 typ.base.output :q 8 9 typ.inner :buf 10 11 par (clk.posedge) do 12 hif(rst) { buf <= 0 } 13 helse do 14 buf[0] <= d 15 buf[-1..1] <= 16 buf[-2..0] 17 q <= buf[-1] 18 end 19 end 20 end </pre>	<pre> # Instances # 8-bit vector sreg(8).(:sr8) # 8-Char buffer sreg(char[16]). (:sch8) # Float sreg(float).(:sf) </pre>
---	--

Fig. 1. An example of HDLRuby code declaring a shift register of generic type and some corresponding instances.

Since the previous publication⁽⁶⁾, HDLRuby has been improved and the syntax slightly changed. The left part of Fig. 1. gives an up-to-date example of a HDLRuby description for a shift register whose structure is defined from a generic argument that can be a bit width, a range or an explicit data type. In the figure, the first line declares the HW module named `sreg` with `typ` as generic parameter. The second and third lines analyze this parameter and convert it to a vector type in case it is not already a type. This statement makes usage of the reflection-oriented Ruby method `is_a?` that checks the class of an object. Lines 5-7 declare the input and output signals of the register, namely, `clk` for the clock, `rst` for the reset, `d` for the data input and `q` for the data output. The type of `d` and `q` is the type of one element of the register and is obtained by `typ.base`. If the data type does not support sub elements, a compile error will be raised. Line 9 declares the storage of the register named `buf` with `typ` as data type. It can be seen from these initial declarations that HDLRuby is object-oriented and reflection-centric: the data type of the elements of a vector type is obtained directly from it through the `base` method, and the declaration of a signal is done similarity through the respective `input`, `output` and `inner` methods of the relevant type element. For the case of `clk` and `rst` the type is implicitly set to `bit` (i.e., single bit). Lines 11-18 describe the process handling the update of the register. Line 11 indicates that the process's statements are non-blocking (`par`) and activated on the rising edge of `clk` (`clk.posedge`). The next line checks if there is a reset (`hif` and `helse` are keywords describing HW 'if' and 'else'). In case of reset, `buf` is set to 0, otherwise, each of its elements is linked in chain using the range objects of Ruby: `[-1..1]` and `[-2..0]` represent

respectively the range from the second (1) to the last element (-1) and the range from the first (0) to the second (-2) last element of `buf`.

The equivalent VHDL or Verilog HDL would look similar, however a generic type argument would not be supported – the only possible genericity would be the width of the register. To see the difference, the right part of Fig. 1 gives a few examples of instantiation of this register. The first instance is an 8-bit shift register, the second is a shifting buffer of 16 characters, and the last is a shift register containing a floating-point value. Those three instances would require each a different HW description with traditional HDL, i.e., about three times more code.

3.2 HW Design Patterns

Several kinds of circuits can be described as sets of finite state machines (FSM), decoders, and arithmetic and logic units (ALU). Components like FSM or decoders may look quite generic but are in practice target-specific and difficult to include in an HDL without losing generality. As a matter of fact, explicit constructs for such components are not present in the standard HDL like VHDL or Verilog HDL. The approach for HDLRuby, is to keep a very general language core, and to provide libraries of template components that can be parameterized and grafted into general RTL descriptions. Such libraries are possible thanks to the metaprogramming capability of the language⁽⁶⁾. We present here the FSM and the decoder templates that have been used in the description of the MEI8 processor.

a. The FSM Template

This template allows to describe synchronous, asynchronous, mixed and single or double-edge FSM by simply specifying the states, the corresponding actions and a few optional configuration parameters. Fig. 2. gives an example of a globally asynchronous FSM with only a few states that are synchronous. This figure is a simplified

```

1 fsm(clk.posedge,rst,:async) do
2   default { init <= 0; calc <= 0 }
3   # Hard reset.
4   reset(:sync) { pc <= 0; ir <= 0 }
5   # Soft reset state.
6   state(:re) { init <= 1 }
7   sync(:re) { pc <= 0; ir <= 0 }
8   # Fetch (synchronous)
9   sync(:fe) { ir <= prog.instr; pc <= pc+1 }
10  # Execute (asynchronous)
11  state(:ex) { calc <= 1
12    # If branch go to br
13    goto(branch,:br,:fe)
14  }
15  # Branch
16  state(:br) { goto(:fe) }
17  sync(:br) { hif(nbr) { pc <= npc-1 } }
18 end

```

Fig. 2. An example of FSM description in HDLRuby.

```

1 type STATE_T is (RE,FE,EX,BE);
2 signal state, next_state : STATE_T;
3 process(clk,rst)
4 begin
5   if (rising_edge(clk)) then
6     if (rst = '1') then
7       ir <= 0; pc <= 0;
8     else
9       case state is
10        when RE => ir <= 0; pc <= 0;
11        when FE => ir <= instr; pc <= pc + 1;
12        when BR => if (nbr = '1') then
13          pc <= npc - 1;
14        end if;
15        when others =>
16          end case;
17      end if;
18    end if;
19  end process;
20  process(state, branch)
21  begin
22    case(state)
23    when RE => next_state <= FE; init <= 1;
24    when FE => next_state <= EX;
25    when EX => if (branch = '1') then
26      next_state <= BR; calc <= 1;
27    else
28      next_state <= FE;
29    end if;
30    when BR => next_state <= FE;
31    when others =>
32      next_state <= FE; init <= 0; calc <= 0;
33    end case;
34  end process;

```

Fig. 3. The equivalent FSM in VHDL.

version of the MEI8 main FSM where interrupts, IO bus accesses and specific instructions handling have been removed. In the figure, the first line is the header of the FSM and indicates that by default its output signals are generated asynchronously (`:async`), that the state transitions are performed on the rising edge of signal `clk` and that the reset is done on signal `rst`. Line 2 gives the default actions and line 4 gives the action to perform in case of reset (here, setting the program counter and the instruction register to 0). The first actual state is described from line 6 and is named `:re`. Here, `state` is used for defining a state. This state, asynchronous by default, has also a synchronous part added line 7 through `sync`. For this state, no transition is specified and therefore the FSM will go by default to the next declared state, i.e., `:fe`. State `:fe` is only synchronous and is therefore added through `sync`. This state does not have any transition specified either and therefore goes to the following `:ex` state. An explicit transition can be specified using `goto`, like in line 16 where the next state is set to `:fe`. It is also possible to set multiple alternative next states depending on a condition as it is done line 13 where depending of whether signal `branch` is 1 or 0, the next state will be `:br` or `:fe` (`goto` is implemented like a multiplexer so that the number of possible target states is not limited).

For comparison, Fig. 3. gives a the equivalent VHDL code. The code is significantly longer and complex (e.g., it includes two processes, two case statements).

```

1 dst <= 0
2 decoder(ir) do
3   # Format 0
4   entry("00000000") { wr <= 0 }      # nop
5   entry("00xxxxyy") {
6     hif (x == y) { alu.(15,0,0) }    # mov 0,y
7     helse { alu.(7,regs[x]) } # mov x,y
8     dst <= y }
9   # Format 1: standard binary ALU computation
10  entry("01oooyyy") {
11    wf <= 1
12    # Destination is also y in case of inc/dec
13    hif (ir[6..4] == _101) { dst <= y }
14    alu.(o,a,regs[y])} # binary alu
15 end

```

Fig. 4. An example of a decoder description in HDLRuby.

The Decoder Template

This template allows to describe a decoding circuit by providing a list of decoding formats and the corresponding actions. Fig. 4. gives an example of a decoder with three different formats. This figure is also an example taken for the description of the MEI8 processor, more precisely it is a part of its instruction decoder. In the figure, before the decoder is described, line 1 sets accumulator `a` (index 0 in the register file) to be the default destination register of the ALU by assigning its index (0) to signal `dst`. Line 2 is the header of the decoder and indicates that signal `ir` (instruction register) is to be decoded. The remaining lines describe the behavior of the decoder as a list of entries, the first one having the highest priority. The circuit of this example being an instruction decoder, the action of each entry is mainly to set up the links between the arithmetic and logic unit (`alu`) and the registers. The first entry of the decoder describes the case where all the bits of `ir` are 0. It corresponds to the `nop` instruction (no operation) and sets signal `wr` to 0, indicating that the destination register should not be written to. The next entry describes the other cases where the two upper bits of `ir` are equal to 00. It corresponds to the register moves (copy between general purpose registers), or to the assignment of 0 to the destination register. For this entry, `ir` is decomposed into two fields, one three-bit field `x` and one three-bit field `y` – notice: a field name is always one character long, its occurrences in the entry indicating the bits used for the field. If both `x` and `y` are equal, the ALU of the processor is set to produce a 0, otherwise, the ALU is set to transfer the value of register number `x`. Finally, destination register index `dst` is set to `y`. The last entry of the example describes the case where the first two upper bits of `ir` are equal to 01. It corresponds to the standard arithmetic and logic operations. For this entry, `ir` is decomposed into field `o` (3-bit) that indicates the operation code, and field `y` (3-bit) that gives the number of the second source register. Line 14 sets up

the links to the ALU circuit with respectively the operation (field `o`) the first source register (accumulator `a`) and the second source register (whose index is obtained from field `y`).

Equivalent code in Verilog HDL or VHDL will require several if and case statements, additional signals declaration for assigning the fields of the `ir` register, and the connections to the ALU circuit would also require extra signals and statements since in such HDL, function call-like connection to instances is not supported. The sample code is omitted for the sake of conciseness but a VHDL version can be found at the MEI8's code repository⁽⁸⁾.

4. Evaluation of the Code

4.1 Methodology

In its current state, the HDLRuby toolchain can automatically compile and convert an HDLRuby description into Verilog HDL⁽²⁰⁾ or VHDL⁽²¹⁾. For our experiments the toolchain has been used to produce VHDL code compatible with both FPGA and IC RTL synthesis. The resulting MEI8 cores have been tested for executing a program including all the instructions of the processor and routines for handling interrupts¹ and system calls.

In order to estimate the potential of HDLRuby for improving the design productivity, we compared the code describing the MEI8 processor with the corresponding VHDL code using several code metrics. However, while the HDLRuby code has been written from scratch, the VHDL code is based on the code generated by the HDLRuby design tool. Please refer to section 4.3 for more details about this choice. Several metrics exist for estimating the quality of software code⁽²²⁾. While there is a lack of metrics for estimating the quality of HDL descriptions, the similarity in structure between HW and SW descriptions makes it possible to use the existing SW metrics for comparing HDLRuby with VHDL. In this paper we considered the following metrics:

- **Lines of Code:** the number of lines of code (LOC, also called SLOC, for Source Lines of Code).
- **Variables:** the number of variables and signals.
- **Assignments:** the number of signal assignments and connections.
- **Operations:** the number of operations (arithmetic and logic operations, bit selection, moves, and casts).

- **Controls:** number of control statements.
- **Cyclomatic complexity:** the number of independent decisions in the code, i.e., the number of independent alternatives in 'if' and 'case' statements.

In addition to these SW-oriented metrics we added the following HW-oriented metrics:

- **Processes:** the number of explicit processes.
- **Bit literals:** the number of bit vector literals.
- **Bits in bit literals:** the total number of bits in bit vector literals.

There is a lack of research about metrics for estimating the quality of HW code. However, the number of processes can be a source of errors since they make it more difficult to track the state of a signal. The number of literals is usually not considered as an issue when estimating the quality of SW. Yet, in HW descriptions there are often a lot of bit vector literals that are much more error-prone than the literals used in SW. Since, the probability of an error increases with the size of the literals, the total number of bits in bit literals is also used as complexity estimator.

In addition to the code complexity, we estimated the effort required for the implementation from scratch of the HDLRuby code. This estimation has been done using code churn-based metrics. More precisely, we counted the lines of code added and deleted for each commit to the repository and extract from these data the following metrics⁽¹⁹⁾:

- **Number of commits:** the total number of commits to the code repository.
- **Churned LOC:** the total number of lines of code added to the baseline HDLRuby description of MEI8.
- **Deleted LOC:** the total number of lines of code deleted from the HDLRuby description of MEI8.
- **Churned rate:** the rate between the churned LOC and the final LOC.
- **Deleted rate:** the rate between the deleted LOC and the final LOC.

In order to summarize the total effort required for designing the processor with HDLRuby, the following metric have also been added:

- **Written LOC:** the total number of LOC written when designing the HDLRuby description of MEI8.
- **Written rate:** the rate between the written LOC and the final LOC.

These code churn-based metrics have not been used for the VHDL code because it has been written based on the already designed HDLRuby code.

¹ Both interrupts have been raised during the test.

Table 1. The quality metrics for the VHDL and the HDLRuby code describing the MEI8 processor.

Metric	VHDL	HDLRuby	Ratio
Lines of code	754	256	2.95
Variables	80	60	1.33
Assignments	272	78	3.49
Operations	245	126	1.94
Controls	174	83	2.10
Cyclomatic complexity	163	76	2.14
Processes	11	5	2.20
Bit literals	131	25	5.24
Bits in literals	628	123	5.11

Fig. 5. The code churns in LOC for each commit.

4.2 Results

Table 1. compares the quality metrics for the HDLRuby and the VHDL code respectively, the lower part being dedicated to the HW-specific metrics. In the table, “Ratio” is the ratio between the VHDL metric and the corresponding HDLRuby one. On average on all the SW-oriented metrics, the VHDL code is 2.33 times more complex than the HDLRuby code, with a standard deviation of 0.77. For the HW metrics, the number of processes is consistent with the SW metrics whereas the bit literal metrics are much more in favor of the HDLRuby description. Globally, the complexity of the VHDL description is more than twice the one of HDLRuby.

Fig. 5. gives the number of lines added and removed for each commit to the code repository of the HDLRuby description of the processor. From these raw data, Table 2. gives the resulting churn metrics. In the table, the small number of commits, and the small rate between the total written LOC and the final LOC of 3:1 tend to show that the design effort was indeed small, even so relatively to the small LOC of the HDLRuby implementation. In total, the estimated design time of the processor is about 20 hours².

HDLRuby has been designed to be synthesizable RTL so that the increase in productivity should have little impact

Table 2. The used churn metrics.

Metric	Value	Ratio w.r.t LOC
Commits	21	n/a
Churned LOC	466	1.82
Deleted LOC	524	2.05
Written LOC	780	3.05

Table 3. The FPGA implementation report.

Type	Number	Utilization
LUT slices	418	0.14%
Register slices	117	0.02%
Multiplexers	10	0.01%
I/O pins	24	3.43%
Buffers	1	0.15%
Clock buffers	2	6.25%
Timing	Period	Frequency
FPGA clock	5.000ns	200MHz
MEI8 clock	10.000ns	100MHz
Required time	9.713ns	n/a
Arrival time	5.407ns	n/a
Slack	4.306ns	n/a

Table 4. The IC implementation report.

Type	Value
Nb. of cells	6475
Nb. of nets	35501
Nb. of gates	9199
Width	2.515mm
Height	2.510mm
Surface	6.313mm ²
Clock freq.	28MHz

on the performance of the result. For evaluating if this assumption holds, the processor has been mapped onto the Xilinx Virtex-7 FPGA VC707 Evaluation Kit board⁽²³⁾ (28nm technology) using the Vivado⁽²⁴⁾ tool chain and implemented down to DRC check using the Alliance⁽²⁵⁾ tool chain targeting a 0.5μm CMOS technology with average cell area of 1837μm² (the default for the tool chain). Table 3. gives the synthesis reports for the FPGA, and Table 4. gives the reports for the IC mapping. As it can be seen in the tables, the resulting processor is very small. This was expected, since it is an 8-bit RISC processor. But it is also fast enough to run at 100MHz for the FPGA implementation and 28MHz for the post-synthesis simulation for the IC implementation. Since the processor is able to execute one instruction per two cycle, with an extra cycle required for branches and other extra cycles required when accessing the external memory, its average performance is between 50 and 40MIPS (Million Instructions Per Second) for the FPGA implementation and between 12 and 14MIPS for the IC implementation.

At last, the productivity in term of gates per designer a day can be estimated using the number of gates of the IC of Table 4. and the design time in days (assuming 8-hours working days) as follows: $9199 \div (20 \div 8) \approx 3680$ gates, i.e., an order of magnitude of 3000 gates a day. While coarse, this value is significantly higher than the average one for RTL design, i.e., about 800 gates per designer a day⁽¹⁾.

² Only limited time a day could be assigned to this design.

4.3 Discussion

How to estimate the productivity of a language is a difficult research topic and as far as we know has never been addressed in the context of HW design – please note that we are not talking here about the performance of the synthesis tools. In this paper we used estimators that could be questioned. For instance, the LOC is often said to be a poor estimator of code quality⁽²⁶⁾. Hence, several other estimators have been used. It can also be objected that SW estimators are not relevant for HW design, and so we also used a few HW-oriented ones. Still, such estimators have not been studied so they are to be taken with caution. Another difficulty for estimating the productivity of HDLRuby, is that some of its features (e.g., reflection) are mostly overlooked by the code quality estimators.

The evaluation of the design effort has been made using the code churns. This is to our knowledge the first time that such metrics have been used in HW design, so that even though the 3:1 ratio of written over final LOC is a lower than average effort, it is hard to draw a definite conclusion from this result. By contrast, the evaluation of the designer productivity in number of gates a day is commonly used, and our result, indicating a significant increase (3680 against about 800 on average⁽¹⁾) is promising. However, this estimate depends on the target circuit. For a processor, the productivity can be low, since such a circuit lacks the regularity that would greatly benefit from generic programming. Moreover, the choice of making a design from scratch forbade the use of IP components while this is usual the case with recent designs.

Ideally, the HDLRuby and the VHDL code should have been written in parallel by experimented HW designers. Such a setup was difficult in practice for our research structure where only one person was available for the task. Then, a same person reimplementing an already existing circuit would have suffered from influences from the existing code and design choices. That is why a less time-consuming compromise has been selected: since the design in HDLRuby would anyway bias the other implementations, it has been decided to use the code generated by the HDLRuby tool chain as basis for writing the VHDL code. The core of the work was then to improve the compactness and the style of the VHDL code. Further, it has been decided to count as line of code the line of comments in the HDLRuby code, while the corresponding VHDL code have been left without any comment in order to avoid an artificial increase of the code size. The largest

drawback with this approach is that it was not possible to estimate the design time using VHDL for the processor. Another drawback is that the resulting VHDL code may be in fact shorter than VHDL code written from scratch due to a heavy usage of compact logic expressions generated by the HDLRuby engine that are usually avoided by the designers in favor of control flow-like constructs.

5. Conclusions

This paper presented the implementation from scratch of MEI8, an 8-bit RISC processor using the HDLRuby language. Then, it compared the resulting code with an equivalent VHDL implementation and gave an evaluation of the design effort required for making the HDLRuby implementation. Finally, it gave details about the resulting circuit in FPGA and IC versions and used these figures for obtaining an estimate of the average productivity in number of gates produced a day. The comparison showed that the HDLRuby code's length was less than a half of the VHDL's. Moreover, with HDLRuby, the required design effort proved to be low while the productivity was about four times higher than a standard RTL approach.

While this study shows some of the potentials of HDLRuby, the limitations of the used estimators invites to perform evaluations with several other kinds of circuits. Moreover, not all the features of HDLRuby have been evaluated in this paper, especially the extensive generic programming and reflection features of HDLRuby have not been fully addressed. Preliminary such evaluations have been made⁽²⁷⁾, but more throughout works are required.

Regarding HDLRuby, we plan to take advantage of the plasticity of the language for providing libraries for supporting IP, the dynamic partial reconfiguration capabilities of FPGA-based devices, the description of SW executed on processor cores, and the description of high-level communication protocols.

Acknowledgment

This research is founded by the Japanese Grants-in-Aid for Scientific Research (KAKENHI) No. 18K11284.

References

- (1) Michael Meredith and Steve Svoboda : “The Next IC Design Methodology Transition Is Long Overdue”, 2012

- https://workspace.accellera.org/resources/articles/icdesi_gntrans
- (2) International Technology Roadmap for Semiconductors, 1999
 - (3) Philippe Coussy and Adam Moraweic : “High-Level Synthesis: from Algorithm to Digital Circuit”, Springer Science & Business Media, 2008.
 - (4) Grant Martin and Gary Smith : “High-Level Synthesis: Past, Present, and Future”, Design & Test of Computers, Vol. 26, No. 4, pp. 18-25, 2009
 - (5) Lovic Gauthier : “HDLRuby Source Code and Documentation”
<https://github.com/civol/HDLRuby>
 - (6) Lovic Gauthier and Yohei Ishikawa : “HDLRuby, a new High Productivity Hardware Description Language”, Proceedings of the 5th International Conference on Business and Industrial Research (ICBIR), pp. 215-220, 2018
 - (7) Yukihiro Matsumoto and Kiju Ishituka : “The Ruby Programming Language”, Addison Wesley, 2002
 - (8) Lovic Gauthier : “MEI8, a Simple 8-bit RISC Harvard Processor”
<https://github.com/civol/MEI8>
 - (9) Louis Hafer and Alice C. Parker : “A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 2, No. 1, pp. 4-18, 1983
 - (10) Pierre Paulin and John P. Knight : “Force-directed scheduling for the behavioral synthesis of ASICs”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8, No. 6, pp. 661-679, 1989
 - (11) Luca de Alfaro Thomas A. Henzinger : “Interface Theories for Component-Based Design”, Proceedings of the 2001 International Workshop on Embedded Software (EMSOFT), pp 148-165, 2001
 - (12) Wander O. Cesário, Amer Baghdadi, Lovic Gauthier, Damien Lyonnard, Gabriela Nicolescu, Yanick Paviot, Sungjoo Yoo, Ahmed Amine Jerraya, Mario Diaz-Nava : “Component-based design approach for multicore SoCs”, Proceedings of the 39th Design Automation Conference, pp. 789-794, 2002
 - (13) O.S. Initiative : “Functional Specification for SystemC 2.0”, 2001
<http://www.systemc.org>
 - (14) Daniel D. Gajski and Jianwen Zhu and Rainer Dömer and Andreas Gerstlauer and Shuqing Zhao : “SPECC: Specification Language and Methodology”, Springer Science & Business Media, 2012
 - (15) MathWorks ® : “Hardware Design with MATLAB and Simulink”
<https://www.mathworks.com/discovery/hardware-design.html>
 - (16) Stuart Sutherland, Simon Davidmann and Peter Flake : “SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling”, Springer Science & Business Media, 2006
 - (17) Phil Tomson : “RHDL – Ruby Hardware Description Language”
<https://github.com/philtomson/RHDL>
 - (18) Thomas James McCabe : “A Complexity Measure”, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320, 1976
 - (19) Nachiappan Nagappan and Thomas Ball : “Use of Relative Code Churn Measures to Predict System Defect Density”, Proceedings of the 27th International Conference on Software Engineering (ICSE), pp. 284-292, 2005
 - (20) IEEE SA: “1364 – IEEE Standard for Verilog Hardware Description Language”
 - (21) IEEE SA: “1076 – IEEE Approved Draft Standard for VHDL Language Reference Manual”
 - (22) Tibor Gyimothy and Tibor Bakota and Peter Kortvelyesi and Rudolf Ferenc and Peter Hegedus : “A Probabilistic Software Quality Model”, Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), pp. 243-252, 2011
 - (23) Xilinx : “VC707 evaluation board for the Virtex-7 FPGA user guide (v1.8)”
<https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html>
 - (24) Xilinx : “Vivado Design Suite Quick Reference Guide”
https://japan.xilinx.com/support/documentation/sw_manuals_j/ug975-vivado-quick-reference.pdf
 - (25) LIP6 : “Alliance – A free VLSI/CAD System”
<https://www-soc.lip6.fr/equipe-cian/logiciels/alliance/>
 - (26) Capers Jones and Associates LLC : “A Short History of Lines of Code (LOC) Metrics”
concepts.gilb.com/dl187
 - (27) Hiroya Kitahara and Sho Kogawa and Lovic Gauthier : “Evaluation of the Productivity of the New Hardware Description Language HDLRuby”, Proceedings of the 2nd Education and Research International Workshop of Electronic Devices, Circuits, Illuminations, and Systems, 2018