RTLog Framework: Yet another open HDL and compiler, this time for Relative-Timing design

Roberto Simone, Pablo D'Angelo, Ian Sztenberg, Francisco Badenas,
Francisco Dominguez, Agustin Ortiz, Guillermo Makar and Roberto Suaya
Departamento de Ingeniería Electrónica
Universidad Tecnológica Nacional, Facultad Regional Buenos Aires
Email: rsimone@frba.utn.edu.ar, fbadenas@est.frba.utn.edu.ar

***Abstract*—The development of a synthesis flow suitable for Relative-Timing asynchronous digital circuits design is presented. The compiler employs a four-phase handshake protocol to develop feed-forward micro pipelines. An associated HDL language, specifically devoted to describe such circuits, and a timing constraint generator are included in the present framework.**

***Index Terms*—HDL, Asynchronous, Compiler, OpenSource**

I. Introduction

Relative Timing [1] is a mature technique for applications demanding high performance silicon with low power budget. Its design flow, on the other hand, could benefit from additional work in the automation of repetitive tasks. This work presents a user friendly framework suitable for both Relative Timing as well as synchronous designs which is compatible with industry standard Electronic Design Automation (EDA) tools.

The starting point is the analysis of a simple pipeline stage [2] with launching and capturing registers as shown in Fig. 1.

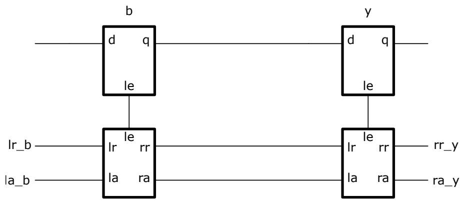


Fig. 1: Asynchronous Pipeline

In most asynchronous circuits with handshake the latching command is given by a linear controller that manages the data transfer between both latches. The circuit path where data is generated and consumed by the latches is referred to as datapath and provides all the necessary information about the circuit structure. The location of the controllers and the handshake signals can be inferred from that structure.

The datapath is completely defined by the data transfer operations between its registers. We introduce a language to describe such operations, which we name RTLog. The language can also be used to describe synchronous circuits.

RTLog skips a detailed description between the handshake connections, which is tedious and error prone. We develop a compiler that takes this asynchronous circuit specification in RTLog, further described in Section II, and transforms this specification into a Verilog source code that can be processed by EDA tools. A diagram of the overall flow is shown in Fig. 2.

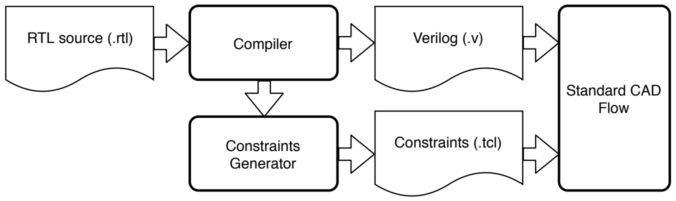


Fig. 2: RTLog Flow

II. The RTLog Language

RTLog language describes asynchronous circuits that employs a four-phase handshake protocol to handle data validity and consumption. The main purpose of the language is to describe data transfers between registers. Behavioral type constructs are not part of the repertory of the language. The language is not of the strongly typed variety. The circuit description starts with the declaration of Input and Output ports (I/O ports) as shown in Listing 1.

```
block foo
begin
    ports
    begin
        input  logic a, b
        output logic c[8]
    end
```

Listing 1: Port declaration

Given the focus on transfer operations, the main group of sentences are the ones related to assignments, as shown in Listing 2. Assignments work at bit or bus level, provided that there is no mismatch in length. Table I presents the operations allowed by the language and Table II shows the reserved words.

```
c = a and b or (not b)
e = (not s and a) or (s and b)
s = a xor b xor ci
```

Listing 2: Assign sentences

TABLE I: Operations supported by RTLog

not	Logic NOT	*	Multiplication
or	Logic OR	*x	Signed Multiplication
and	Logic AND	==	Compare equal
xor	Logic XOR	!=	Compare different
+	Addition	<	Compare smaller
+x	Addition with carry	<=	Compare smaller or equal
-	Subtraction	>	Compare greater
-x	Subtraction with borrow	>=	Compare greater or equal
>>	Right Shift	cat	Concatenation
>>s	Right Shift signed	rep	Replication
<<	Left Shift		

TABLE II: Keywords of RTLog

and	cat	if	natural	parameters	select
begin	constant	in	not	port	when
block	end	for	or	reg	xor
case	else	logic	others	rep	xnor

RTLog incorporates the usual arithmetic operations. Addition, subtraction and multiplication operators are present. There is an extra operator that allows to directly recover carry from additions. This feature, not available in Verilog, simplifies the coding process. A similar operator is available for subtractions.

For multiplications, the realization of a multiplier circuit depends on whether the operands are signed or unsigned. RTLog defines an operand for signed and another operand for unsigned multiplications, instead of introducing a data type for the operands. With this approach, operands remains as pure binaries words.

Operands to perform concatenation, replication, rotation and shifting are also present.

To instantiate a register, the reg modifier must be used as part of the signal declaration. For non-registered signals, the $logic$ modifier is used instead. The register description process in RTLog is simpler than its counterpart in Verilog. The usual description of a Flip-Flop D with asynchronous reset in Verilog involves several lines of code. The RTLog modifier reg embraces all this concepts in one word. An example of this is shown in Listing 3.

```
block foo
begin
    ports
    begin
        input  logic a, b
        output logic c, e
        output reg d
    end
reg f

c = a and b
d = c or b
f = c
e = f
end
```

Listing 3: Register declaration and use of RTLog

The $logic$ modifier is also used when declaring non-registered output ports.

The constructions if and $select\ case$ allow for the implementation of multiplexers with and without priority respectively.

RTLog allows $for - generate$ like constructs to describe iterative cells. Iteration over hard-coded indexes or over a range of values given by a relational expression can be performed using the $when$ clause. RTLog has special clauses for parametric design and for the declaration of constants.

III. The Compiler

The program flow can be conceptually divided in three sections. The first one consists of a parser which takes the RTLog source code and transforms it into a directed graph. This is the typical process of a recursive descendent parser [3]. After this step, a synchronous version of the circuit can be easily produced by adding a global clock. The second section adds the necessary controllers to the original directed graph, to obtain an equivalent asynchronous version (an augmented directed graph). The third section generates the Verilog source code from the augmented directed graph. The output can be taken as an input to a number of VLSI synthesizers. This flow has been tested using Synopsys Design Compiler [4].

The algorithm of the second section consists on:

A. Addition of controllers

To add the controllers to the original directed graph, the asynchronous version is generated following the next steps:

1) Replace the Flip-Flop nodes with Latch nodes.
2) Identify the Register-to-Register paths.
3) Insert the linear controller nodes for every Latch.

The first step is satisfied by browsing the cells in the graph and replacing the attributes of those identified as flip-flops with latch attributes. The second step consists of a graph traversal [5]. To avoid further decomposition into several unrelated register-to-register (R2R) paths, registers should be considered as a complete binary word, as seen in Fig. 3.

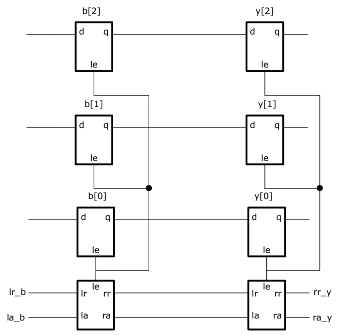
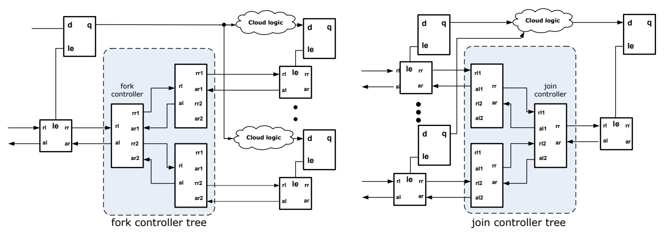


Fig. 3: Asynchronous pipeline for a bus

B. Fork and join controller trees

Asynchronous circuits typically require the insertion of $fork$ and $join$ controllers, in addition to the linear ones. The $fork$ controllers are necessary when a launching register drives multiple R2R paths to guarantee that every register captures the data. Fig. 4a shows this scenario.



(a) Circuit with fork controllers (b) Circuit with join controllers

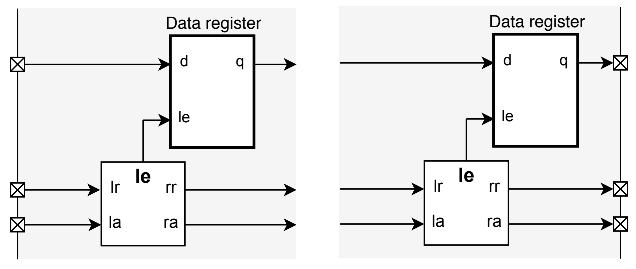
Fig. 4: Tree configurations

The algorithm counts the number of paths from a single launching register to all capturing registers from the set of R2R paths. With this information all the necessary fork controllers are inserted and the handshake signals are wired. Using 2-way fork controllers, a fork controller tree structure can be obtained to drive more than two R2R paths per launching register.

An analogous situation occurs when multiple paths converge to the same capturing register. Here the device that grants a correct exchange between registers is the join controller. As in the previous case, this implementation requires a tree of 2-way join controllers, as shown in Fig. 4b.

C. Handshake ports

Next, the insertion of top-level handshake ports is performed. The registers connected to the top-level input ports should handle the incoming handshake signals and the registers connected to the top-level outputs ports should handle the outgoing handshake signals. These scenarios are shown in Fig. 5a and Fig. 5b.



(a) Input Handshake (b) Output Handshake

Fig. 5: Handshakes for IO ports

At this point, the asynchronous 4-phase handshake protocol circuit is completed and the Verilog source code is generated for its ulterior synthesis. The output code includes both structural and concurrent Verilog sentences. The code includes a separate library of controllers and latches, described in Verilog, which are Process Design Kit (PDK) dependent. The designer can update the PDKs with ease.

IV. Code Examples

As a first example of the advantages of RTLog, the source code for a 2-stage First-In First-Out (FIFO) Shift Register circuit with an inverted output is shown in Listing 4. Line 12 of this example can be described in Verilog as shown in Listing 5. RTLog shows a significant reduction in total lines of code.

```
 1 block fifo_async
 2 begin
 3         ports
 4         begin
 5                 input logic a
 6                 output logic b
 7         end
 8 
 9         reg q_1 , q_2
10 
11         q_1 = a
12         q_2 = not q_1
13         b = q_2
14 end
```

Listing 4: FIFO en RTLog

```
 ...
 always @ (posedge rst) begin
         if (rst) begin
                 q_2 <= 1'b0;
         end else if (l_en_2) begin
                 q_2 <= q_2_next;
         end
 end

 assign q_2_next = not q_1;
 assign b = q_2;

 C_element lc_2 (
      .le(l_en_2),
      .lr(rr_int), .la(ra_int),
      .rr(rr_2),   .ra(ra_2)
      );
```

Listing 5: Register Transfer and Latch Controller

This benefit is further improved for more complex designs. Number of lines of code demanding manual generation to Verilog designer which are automatically generated using RTLog are shown for a FIR filter in Table III.

TABLE III: Verilog lines automatically generated for a FIR filter

Element	Number	Lines per element	Total lines
Linear controllers	5	7	35
Join controllers	3	9	27
Fork controllers	3	9	27
Wires	26	1	26
Total number of Verilog lines needed			115

Table IV displays the equivalent savings for each of the 64 stages of a SHA-256 [6] hasher circuit when described in RTLog.

TABLE IV: Verilog lines automatically generated for a SHA-256 Hasher

Element	Quantity	Lines per element	Total lines
Linear controllers	24	7	168
Join controllers	15	9	135
Fork controllers	16	9	144
Wires	138	1	138
Total Verilog lines needed			585

V. Constraints generation

There are a significant number of approaches to asynchronous circuits design. This work is developed under the Relative Timing (RT) methodology.

The RT methodology constrains every R2R path of the circuit to avoid timing violations and guarantee data validity. A second set of constraints is needed to ensure that the synthesis tool (oriented to synchronous circuits) preserves the combinational loops of the controllers.

RTLog uses feed-forward pipelines (input to output, without feedback) to build the datapaths. Therefore the constraint generation can be automatized by browsing the list of R2R, I2R and R2O paths provided by the compiler.

This constraints can be grouped in three types, as seen in Fig. 6:

- Capture path
- Data path
- Internals to the linear controller

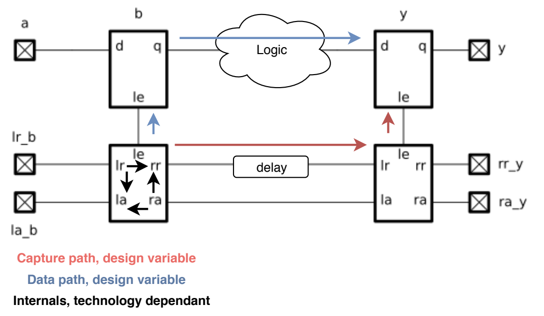

a
b
d
q
le
Logic
y
d
q
le
y
lr_b
le
lr
rr
la
ra
delay
le
lr
rr
la
ra
rr_y
ra_y
la_b
Capture path, design variable
Data path, design variable
Internals, technology dependant

Fig. 6: Basic constraints of RT methodology

The internal constraints prevent the synthesis tool from modifying the internal circuit of the asynchronous controllers using $set\_disable\_timing$ and set_size_only commands. The control and data paths are constrained to ensure that data is available at the input of every latch before the enable signal arrives and avoid timing violations. The data path employs a set_max_delay command and the capture path employs a set_min_delay command. The $delay$ element is inserted between the controllers as re-sizable buffer so the synthesizer can meet constraint values. The compiler in its current state supports the automatic generation of constraints with precalculated values related to the specific fabrication technology. The RT model is significantly more comprehensive than what is being shown here [1].

VI. Future work

Both RTLog and the code compiler tool are in an advance stage of development. There is of course room for refinement and improvements. Two dimensional arrays and module instantiation are being incorporated. Comparisons with HDL languages, like FIRRTL [7], and compilers are pending. Our tests were performed with commercial tools from Synopsys, a temporary approach to support the initial development of the framework. A migration to OpenSource EDA tools, like Yosys [8] and OpenTimer [9], is part of our main objective.

With regard to the process of automatic constraint generation, we are already working on techniques to reduce the number of manual checks required. Iterations with PrimeTime [10] tool to address performance and power optimization, as well as area, will be included.

VII. Conclusions

This work presents a novel approach to HDL coding, RTLog, for describing asynchronous circuits that follow a four-phase handshake protocol with promising advantages over Verilog. A compiler tool to output a Verilog source code suitable for synthesis has been developed. Supporting elements were created to approximate a complete framework. The complete work is coded in C++. The current state of the project is alpha and can be found on GitHub [11].

Acknowledgment

We would like to thank the Universidad Tecnológica Nacional - Facultad Regional Buenos Aires for supporting this work with five undergraduate scholarships under the project EIUTNBA0004541.

References

[1] K. S. Stevens, R. Ginosar, and S. Rotem, “Relative timing [asynchronous design],” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 1, pp. 129–140, Feb 2003.
[2] I. E. Sutherland, “Micropipelines,” *Commun. ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989. [Online]. Available: http://doi.acm.org/10.1145/63526.63532
[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.
[4] Synopsys, *Design Compiler User Guide*.
[5] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, 1st ed. Norwell, MA, USA: Kluwer Academic Publishers, 2000.
[6] U. D. of Commerce, N. I. of Standards, and Technology, *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*. USA: CreateSpace Independent Publishing Platform, 2012.
[7] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, “Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations,” in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 209–216.
[8] C. Wolf, “Yosys open synthesis suite,” http://www.clifford.at/yosys/.
[9] T. Huang and M. D. F. Wong, “Opentimer: A high-performance timing analysis tool,” in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2015, pp. 895–902.
[10] Synopsys, *Primetime User Guide*.
[11] RTLog Repository. [Online]. Available: https://github.com/VLSI-UTN-FRBA/RTLog