Modeling and Synthesis of a Single-Precision Floating-point IEEE 754 Multiplier

Fabio Chiarani - VR445566

***Abstract*—The following document shows how two parallel and equal IEEE 754 single precision floating-point multipliers have been realized through the modelling and synthesis by two HDL languages (SystemVerilog and VHDL). The goal is to use one toplevel that controls both multipliers, thus managing a handshake and serializing the output when is done. The synthesis produced, will be compared with a synthesis of a C floating point using the High Level Synthesis.**

I. Introduction

The project want to model a toplevel that controls two multipliers that perform multiplication according to the IEEE754 standard. The toplevel must be synthetized for an FPGA with only 125 ports (I/O) available. It is therefore necessary that the toplevel manages and serializes the inputs and outputs.

The toplevel is tested through a testbench, which reads inputs from a file. After that will be synthetized and its maximum clock speed will be evaluated (with other timings and area reports).

The toplevel, written in VHDL and Verilog, it has also written in SystemC, to view how it is possible to describe and model a program in different levels of abstraction. Then, the timings report obtained will be compared with the C algorithm description of the IEEE 754 multiplication timing report retrived from the High Level Synthesis.

The Verilog toplevel is synthetized with 444 LUT, 395 FF, 1 BUFG, a minimum clock of $9ns$ and a power stimation of $0.112W$ with 26.4 celsius degree for the junction temperature.

The 'Background' section gives some information about HDL languages and the IEEE 754 standard. Inside 'Implementation' is described how the project is made: from the creation of the EFSM to the High Level Synthesis of the C algorithm.

Section 'Architecture of VHD/Verilog RTL' show the hardware architecture, interface and EFSM, while the section 'Architecture of SystemC RTL' show the architecture and the project structure for the SystemC modelling.

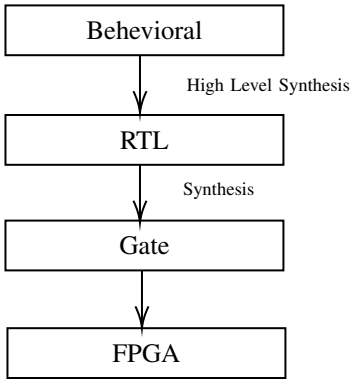
After that, the 'RTL Simulation' section report the simulation phase of the Verilog and SystemC multipliers. Then the synthesis process are described inside the 'Synthesis' section, followed by the 'Implementation' section that provides the timing and area reports. Finally, the data obtained by the C HLS is reported on the 'HLS' section.

II. Background

HDL is a specialized language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits. It enables a formal description of an electronic circuit that allows for the automated analysis and simulation of an electronic circuit. It also allows for the synthesis of a HDL description into a netlist, which can then be placed and routed to produce the set of masks used to create an integrated circuit. [1]

The two IEEE standard in common use are Verilog and VHDL. This two languages are common on RLT describing level. The High Level Synthesis (HLS) [2] is another choice that can be adopted to execute the automatic translation and synthesis of an algorithmic description written in C, C++ or SystemC.

In hardware design there are different levels of abstraction, on which different languages can operate, each with its own advantages and disadvantages. Generically, is possible to describe the system on RLT level with VHDL or Verilog, then making the synthesis to have the gate level. Another way is to make the HLS from a behavioral description by a SystemC, C or C++ description. A possible flow is shown below:



Through tools such as Vivado [3], is possible to set some constraints that must be respected (e.g. clock requency) by the design. In this way the tool shows if the design respects these constraints, proceeding with the implementation, or it is necessary to refine the system. After the synthesis and implementation is possible to view the timing reports, latency and area of the generated model.

In IEEE 754-2008 the 32-bit base 2 format is officially referred to as binary32. It was called single in IEEE 754-1985. The IEEE 754 standard specifies a binary32 as having 1-bit sign, 8-bits for exponent and 24-bits for significand precision. The IEEE 754 provides a A full description of the IEEE754 standard can be viewed on Wikipedia [4]. The IEEE 754 arithmetic (used in this project) can be viewed on Wikipedia [5].

III. Implementation

After analyzing the required specifications, it was decided to design the system through an EFSM. This type of development has been useful in understanding well the system flow and the calculations that must be performed. In a second phase of review, having an EFSM, makes possible to do some improvements e.g. minimizing the number of states and/or transactions.

After the EFSM definition, the RTL level components were developed first: the VHDL multiplier was created, tested through a minimal testbench to verify its correctness and, after that, the specular multiplier was written in Verilog.

The toplevel and main testbench of the project is written in Verilog. It has been chosen to create a testbench that sends the same data to both modules to verify its correctness at the same time. The data is loaded from a file.

Then I moved to the creation of the RTL part (composed by the same components and EFSM) in SystemC. The SystemC RTL project is however written with a synthesizable grammar. The testbench, having the same EFSM as the project written in VHDL, and properly tested, was not written by inputting values from files, but only 15-20 cases were tested to verify its correctness.

The last part of the project, the HLS, was performed from an algorithmic description of the floating point multiplication in C++. The project architecture can be seen divided into two parts: the testbench (*tb*) and the toplevel (*tl*). The same architecture is used within the RTL description in Verilog/VHDL and SystemC. The architecture will be described according to a bottom-up approach, from the internal IEEE754 component to the toplevel:

IV. Architecture of VHD/Verilog RTL

This section shows the architecture and the hardware description.

A. IEEE754 Multiplier

The multiplier component perform the floating point single-precision multiplication according to the IEE754 standard. The assumption is that the imputs are already normalized values, otherwise the multipler returns all res signal with all zeros.

1) Component Interface: The Figure 1 shows the module interfaces. It have 5 inputs and 2 outputs:

- $[op1]$ is the first number of the multiplication.
- $[op2]$ is the second number of the multiplication.
- $[inrdy]$ tell to the module when the input are ready to compute the multiplication.
- $[reset]$ reset the module.
- $[clk]$ clock signal.
- $[res]$ result of multiplication.
- $[resrdy]$ tells when the result (res) is ready.

Inside on the VHDL module, on the architecture, there are 9 signals to handle data during the multiplication. Instead, on the Verilog module there are 9 registers. The data types are STD_LOGIC, choosed for the possibility to make a better testbench simulation with the type propreties.

Both VHDL and Verilog module are modeled with 2 processes: the fsm and the datapath. The fsm manages the current and the next state; the datapath calculates. The datapath is driven by the clock and the fsm by the state signal. The EFSM of the multiplier is show below.

2) Component EFSM: The multiplier EFSM showed in Figure 2 describes the multiplication algorithm. Is composed of 21 states, handles infinite, zero and NaN cases and the rounding algorithm according to the IEEE754 standard.

Short description of the sates:

- $[s_0]$ Initial and reset state.
- $[s_1]$ The machine stays here until input data are ready.
- $[s_2]$ Dispatch the input type.
- $[Inf]$ Handle infinity input value.
- $[Zero]$ Handle zero input value.
- $[NaN]$ Handle NaN input value.
- $[err]$ Return all zeros as output for show the invalid input error.
- $[s_6]$ Dispatch from normalized and denormalized input (if denormalized move to err).
- $[s_10]$ Make the multiplication.
- $[s_11]$ Normalize the value.
- $[s_12]$ Check for overflow or underflow.
- $[s_13]$ Handle overflow case.
- $[s_14]$ Handle underflow and correct case.
- $[norm]$ Reached when got a normalized input.
- $[denorm]$ Reached when got a denormalized input.
- $[s_15]$ Save on result the new exponent.
- $[s_16]$ Dispatch if need a shift.
- $[s_17]$ Save on result the mantissa.
- $[s_18]$ Shift operation.
- $[s_19]$ Save on result the exponent.
- $[out]$ Output the res value and return to the inital state.

B. TopLevel

The toplevel, written in Verilog, has the task of instantiating a VHDL and Verilog multiplier, for performing the multiplication.

1) Component Interface: Since the FPGA has 125 ports, and the inputs are composed by 32 bit, it was not possible for the top level to accept all the 4 inputs at the same time. The toplevel interface schema is shown on Figure 3, and the architectural choices were made as follows:

- The toplevel accepts in input two values at a time. Inputs can be sent to all the two internal machine (Verlog and VHDL) equally using the $inrdy$ bit with value 11. Otherwise, the inputs can be sent first to the Verilog machine and then to the VHDL machine with the $inrdy$ bit corresponding to 01 or 10.
  By doing this, is possible to redirect inputs to the machines in a different way, decreasing the number of ports used on the FPGA.
- Having two multipliers, therefore two results, it was chosen to use only one output result (res) and to use

a bit ($resrdy$) indicating if the result belongs to the first or second machine.
- The inputs are directly connected to the internal multipliers, controlled by the $inrdy$
- The outputs are controlled by the EFSM.

The total port used from the toplevel are 102 (85%).

2) Component EFSM: The toplevel EFSM showed in Figure 4 describes how it manages and serializes the outputs. It Is composed of 6 states. Short description of the sates:

- [s_0] Initial and reset state.
- [s_1] Dispatch the ready status.
- [s_2] Serialize on output the VHDL result.
- [s_3] Serialize on output the Verilog result.
- [s_4] Both results are ready, so store VHLD value and output the Verilog result.
- [s_4] Output the Verilog result and terminate. Returning to s_0 to accept a new value.

C. Testbench

The testbench is written in Verilog, because of its resemblance to the grammar of C and less verbose than VHDL. The choice was to give input to both multipliers, through toplevel, the same operands ($op1$ and $op2$, with the $inrdy$ value setted to 11). In this way the multiplier are tested that both gave the same result.

Nothing prohibits in the future to modify the testbench in order to input different values using $inrdy$ bit as 01 or 10.

The FSMD schema is showed in Figure 5.

V. Architecture of SystemC RTL

This section show the architecture of the SystemC RTL description.

A. SystemC RTL

The same RTL description and architecture adopted in the VHDL/Verilog showed in Figure 4, Figure 2, Figure 1 and Figure 3 was used for the SystemC RTL modelling.

The data types are sc_logic and sc_lv to remain consistent with the description made in VHDL/Verilog. The file are in *.cpp* format with the *.hh* header. The style used is to use the SC_CTOR on the header file and implement the function on the *.cpp* file. All the code (except the testbench) is synthetizzable.

The multiplier ($multiplier754\_fsmd\_sc$) is builded with two SC_METHOD: the "datapath" and the "fsm" (with same EFSM of VHDL/Verilog). The testbench (tb) is builded with two SC_THREAD: "clk_gen" used for generating the clock, and the "run" thats send the data to the toplevel (tl). The SystemC folder structure is showed below.

```
root (SystemC)
├─ bin
├─ build
├─ src
│  └─ .cpp files
├─ include
│  └─ .hh files
├─ bin
├─ obj
└─ Makefile
```

VI. RTL Simulation

A. SystemC RTL Simulation

Before running the simulation you need the $SYSTEMC$ path variable that points to the SystemC library (change it on the $Makefile$) This project use the SystemC `SystemC 2.3.2-Accellera` version. To run, on the root directory, run `make` and then execute it with `./bin/multiplier754\_rtl.x`.

An example of the simulation is:

```
$ make
$ ./bin/multiplier754_rtl.x
$ Starting tb::clk_gen()
$ Starting tb::run()
$ Out[01]: 01000000010000000000...000
$ Out[10]: 01000000010000000000...000
$ Ending tb::run().
```

Where in this example $op1$ and $op2$ are both setted as 01000000000000000000000000000000 (value of 2) on the $tb.cpp$, expecting the results of 01000000100000000000000000000000 (hex $0x40800000$).

B. VHDL/Verilog RTL Simulation

The simulation was done through Vivado [3]. The testbench (written in Verilog) load the input data from a file ($SimulatonSources/sim\_1/Text/input.txt$). The binary row is the input of all 4 (op1/op2 VHD and op1/op2 Verilog) multipliers ports. The hex row is the expected result.

Running the simulation with the input file containing:

```
01000000000000000000000000000000
40800000
```

On the TLC Console (in Vivado) we can see the output of the simulation:

```
expected 3f800000
vhdl multiplier result: 3f800000
verilog multiplier result: 3f800000
```

Resource	Estimation	Available	Utilization
LUT	224	53200	0.42%
FF	174	106400	0.16%
DSP	2	220	0.91%
BUFG	1	32	3.13%

Table I
VERILOG MULTIPLIER SYNTHESIS

Resource	Estimation	Available	Utilization
LUT	149	53200	0.28%
FF	152	106400	0.14%
DSP	2	220	0.91%
BUFG	1	32	3.13%

Table II
VHDL MULTIPLIER SYNTHESIS

Resource	Estimation	Available	Utilization
LUT	444	53200	0.82%
FF	395	106400	0.37%
DSP	4	220	1.82%
IO	102	125	81.60%
BUFG	1	32	3.13%

Table III
TOPLEVEL SYNTHESIS (POST SYNTHESIS). ON THE POST-IMPLEMENTATION THE LUT CHANGED TO 437.

Timing	WNS	TNS	WHS	THS	Tot. Endpoints
Setup	0.585ns	0.000	-	-	818
Hold	-	-	0.066ns	0.000ns	818

Table IV
SYSTEMC MULTIPLIER SYNTHESIS TIMING

The input is a binary value on the IEEE754 format of 2.0 and the results are 4.0 coded in hex. It is possibile to view the signals of the simulation on the Figure 6. The Figure 7 and Figure 8 show the simulation of 10 different input values.

From the Figure 7 it is possible to view that from the toplevel the outputs ready every $30ns$ using $15clock$ (counting the clocks used for reset the machine). The output on the res port, controlled by the $resrdy$ is shown for one clock.

From the Figure 9 it is possible to view that near the marker at $9ns$, the $next\_state$ and $state$ signals shows the EFSM working, arriving at $23ns$ with the $next_state$ with value $22ns$ (end state). This means that the real multiplication requires $13ns$, with an estimation of $7cloks$.

From the Figure 10 it is possible to view the signals with an input of $infinity$.

The simulation is done before the synthesis, after the synthesis and after the implementation.

VII. SYNTHESIS

The synthesis and implementation process it is done for the PYLNQ with code `xc7z020clg400-1`.

A. Verilog/VHDL Multipliers Synthesis

The synthesis of the $mul\_ieee754\_vhdl$ module produce the values on the Table II. The synthesis of the $mul\_ieee754\_verilog$ module produces the values on the Table I. As we can see, the verilog module has an increment of LUT and FF, perhaps due to the operations performed in the datapath that may not be 100% optimized for synthesis [6].

The Table III shows the synthesis of the toplevel with an increment of all values. From the schematic it is possible to retrive that the toplevel (Verilog) has 174 netlist and 356 nets. The VHDL module has 737 nets vs 770 ot the Verilog, and 437 leaf cells (VHDL) vs 468 (Verilog).

VIII. IMPLEMENTATION

After the synthesis, through Vivado, the clock constraint has been added before running the implementation. On the initial phase, the clock is setted to $20ns$. After running the implementation several times decreasing the clock constraint, the lowest number that can work on is $9ns$.

After the implementation is possible to view and analyze the $Timing$ and $Power$ reports:

1) Timing: The Table IV shows the Setup and Hold time. From this values, the generated bitstream is useful since the timing values are all positive and TNS and THS are 0.

2) Power: From the power report it is possible to view that the implementation estimation $0.112W$ as *Total Power On Chip* with $26.4C$ as *junction temperature*. The power utilization is 14% dynamic with $0.018W$, and 86% static with $0.105W$.

IX. HLS

A. SystemC HLS

Changing the clock type to sc_in_clk the SystemC multiplier is synthetized getting more LUT than the VHDL or Verilog multiplier, showed in Table V.

B. C HLS

The HLS is done through an algorithm description of the multiplication in C:

Module	DSP48E	FF	LUT
datapath	2	0	342
fsm	0	0	473
Total	0	2	815

Table V
SYSTEMC MULTIPLIER SYNTHESIS

Clock	Target	Estimated	Uncertainty
ap_clk	10	5.127	1.25

Table VI
SYSTEMC MULTIPLIER SYNTHESIS TIMING

Name	BRAM_18K	DSP48E	FF	LUT
Istance	-	3	128	138
Multiplexer	-	-	-	21
Register	-	-	3	-
Total	0	3	131	159
Available	270	240	82000	41000
Utilization (%)	0	1	∼0	∼0

Table VII
C ALGORITHM HLS TIMING

Clock	Target	Estimated	Uncertainty
ap_clk	10	8.286	1.25

Table VIII
C ALGORITHM HLS CLOCK

```
void mul(
        float op1,
        float op2,
        float * res)
        {
    *res = op1 * op2;
}
```

The synthesis produces an intresting report showed in Table VII. The FF and LUT Istance usage are the half of the toplevel synthetized. The 3 FF registers are 3-bit and the LUT are 4-bit.

The Table VIII shows how the clock estimated increment vs the Table VI that shows the HLS of a SystemC multiplier. The latency of C HLS is 2.

X. CONCLUSION

The project shows how starting from different levels of abstraction, such as RTL or behavioral, you can synthetize the program with the pros and cons of different languages. I found myself more enthusiastic about using an HDL description at RTL Verilog and VHDL level because it gives you more control over what you can achieve during the synthesis.

Although I did not start with the idea of making the “fastest“ or “smallest“ multiplier, I am satisfied with the WHS, THS, WNS and TNS values obtained. There are several parts I would have liked to go into, but the idea of having seen how to get on an FPGA from a description that can be behavioral or defined by an EFSM through a synthesis of a few lines of code like C, or a VHDL modeling is satisfactory.

REFERENCES

[1] W. HDL. Hdl. [Online]. Available: https://en.wikipedia.org/wiki/Hardware_description_language

[2] W. HLS. Hls. [Online]. Available: https://en.wikipedia.org/wiki/High-level_synthesis

[3] Xilinx. Vivado design suite. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[4] Wikipedia. Ieee 754 floating point single precision. [Online]. Available: https://en.wikipedia.org/wiki/Single-precision_floating-point_format

[5] ——. Arithmetic ieee 754. [Online]. Available: https://en.wikipedia.org/wiki/Floating-point_arithmetic

[6] Z. CPU. Minimizing lut. [Online]. Available: https://zipcpu.com/blog/2017/06/12/minimizing-luts.html

State	Operations

S0	m <= 0;
	m1 <= 0;
	m2 <= 0;
	tmpm <= 0;
	tmpexp <= 0;
	exp <= 0;
	s <= 0;
	res <= 0;
	resrdy <= 0;

S1	-

S2	m1(22:0) <= op1(22:0);
	m2(22:0) <= op2(22:0);
	s <= op1(31) XOR op2(31);

S6	m1(23) <= 1; m2(23) <= 1;
S10	tmpexp <= (0 & op1(30:23)) + (0 & op2(30:23)) - 127; tmpm <= m1 * m2;
S11	tmpm <= ’0’ & tmpm(((24) * 2) - 1 DOWNTO 1); tmpm >>1; tmpexp <= tmpexp + 1;
S12	-
S13	tmpm <= ’0’ & tmpm(((24) * 2) - 2 DOWNTO 0); tmpm <<1; tmpexp <= tmpexp - 1;
S14	tmpm <= ’0’ & tmpm(((24) * 2) - 2 DOWNTO 0); tmpm <<1; tmpexp <= tmpexp - 1;
NORM	-
DENORM	exp <= 0;
S15	exp <= tmpexp(7:0);
S16	-
S17	m <= tmpm(45:23);
S18	tmpm(47:22) <= tmpm(47:22) + 1;
S19	tmpexp <= tmpexp + 1;

NAN	exp <= 1;
	m <= (22 =>’1’, 21 DOWNTO 0 =>0);
	s <= 0;

INF	exp <= 1; m <= 0;
ZERO	exp <= 0; m <= 0;

OUT	resrdy <= 1;
	res(31) <= s;
	res(30:23) <= exp;
	res(22:0) <= m;

ERR	resrdy <= 1; res <= 0;

Table IX
OPERATION OF THE EFSM MULTIPLIER

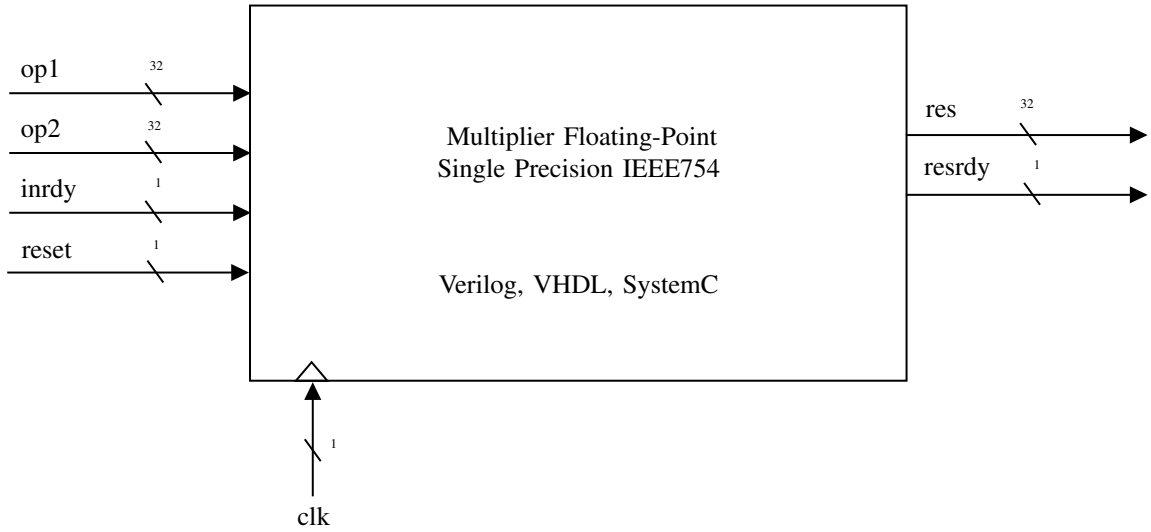


Figure 1. Multiplier interface schema

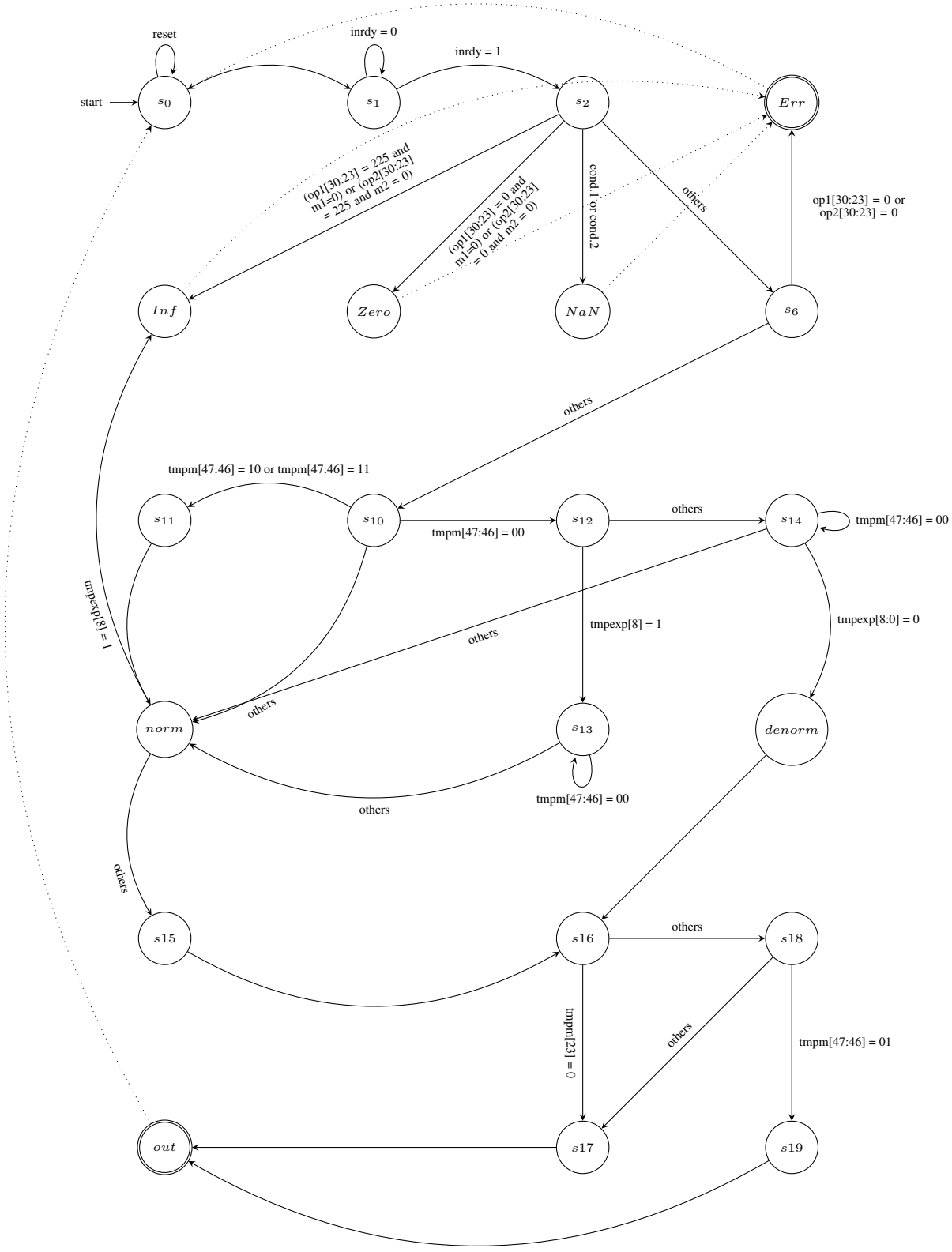


Figure 2. EFSM multiplier. The state operation are described on Table IX

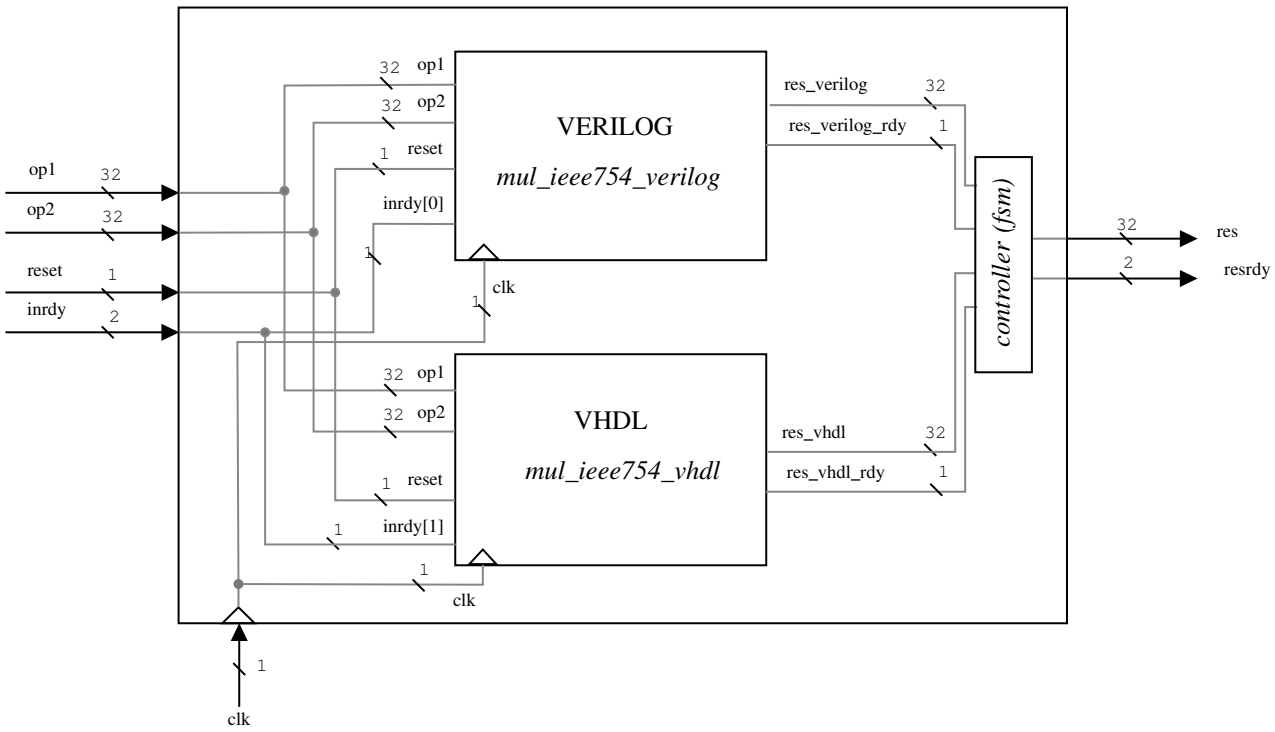


Figure 3. Toplevel interface schema

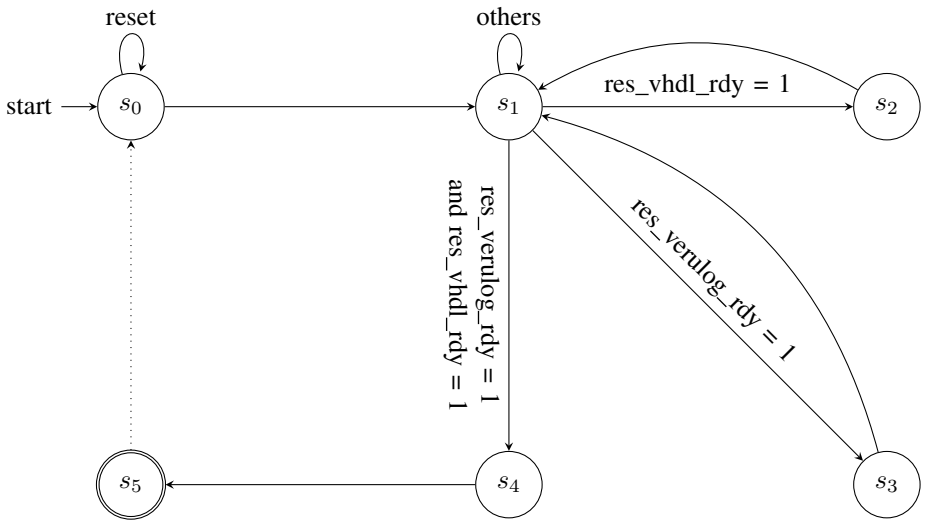


Figure 4. EFSM toplevel

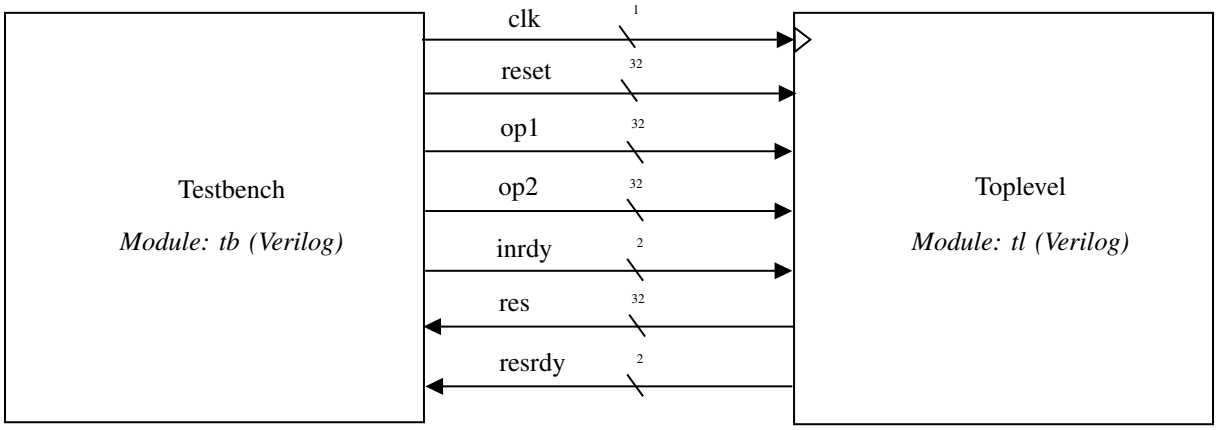


Figure 5. FSMD testbench

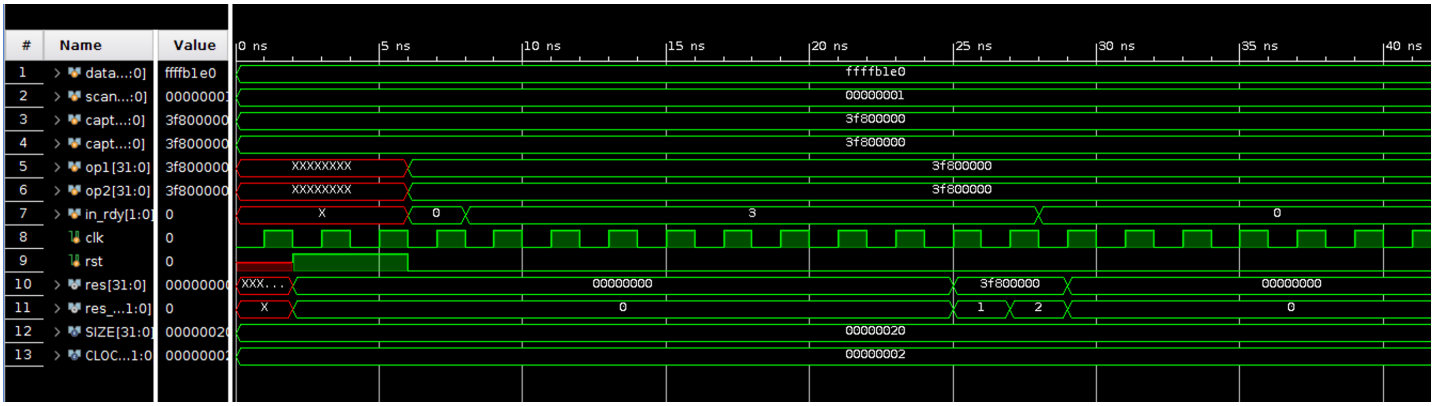


Figure 6. Vivado Simulation Example nr.1 with one input

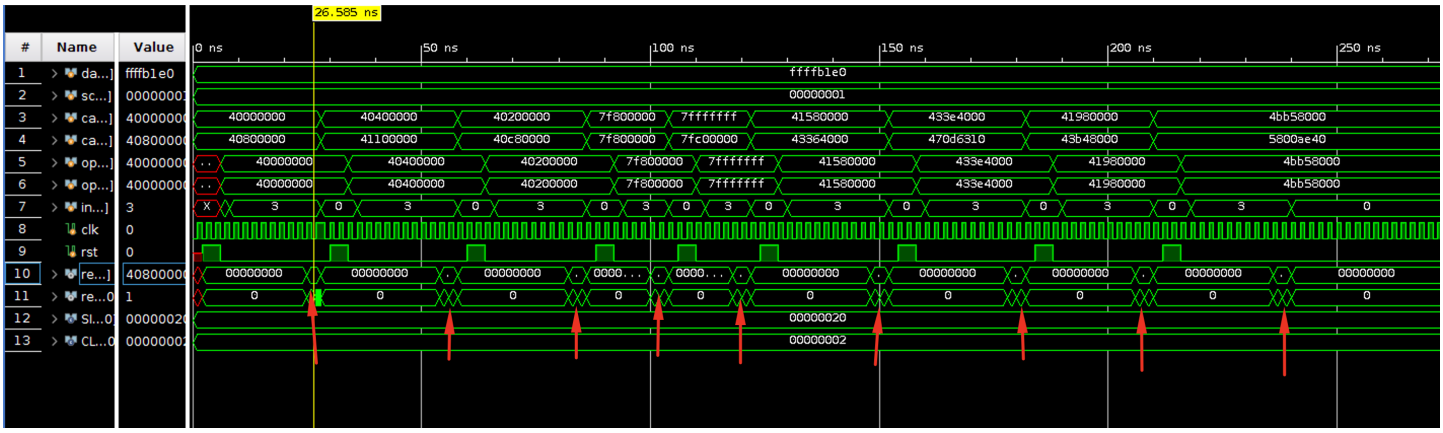


Figure 7. Vivado Simulation Example nr.2 with 10 inputs. The red arrows indicate when the res is outputting the value

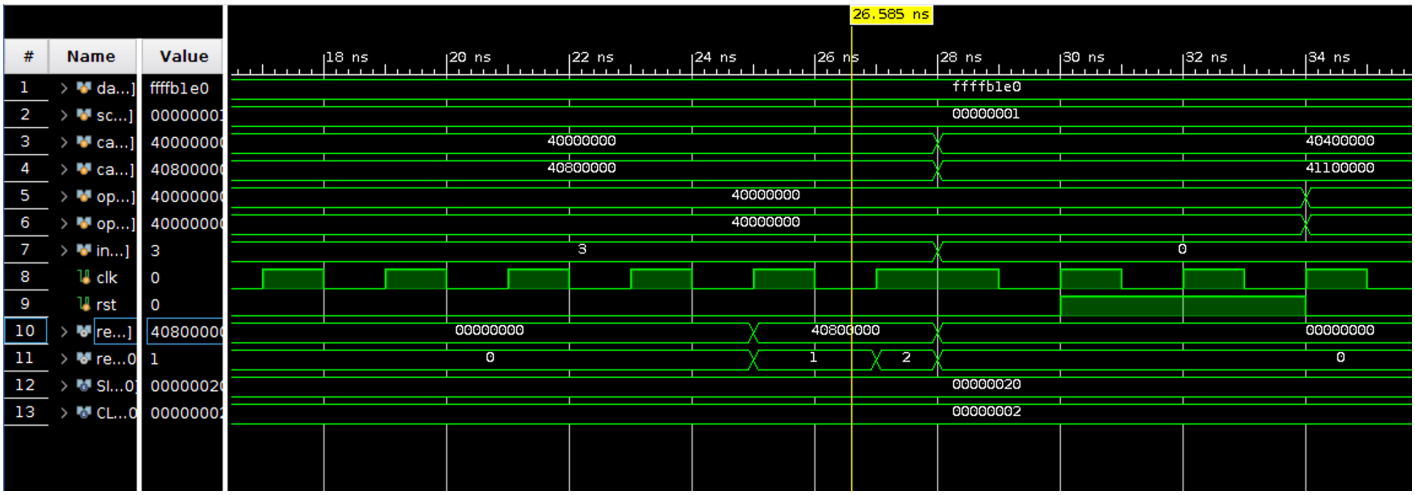


Figure 8. Vivado Simulation Example nr.3, Zoom at the $26,585ns$ of the simulation nr.2

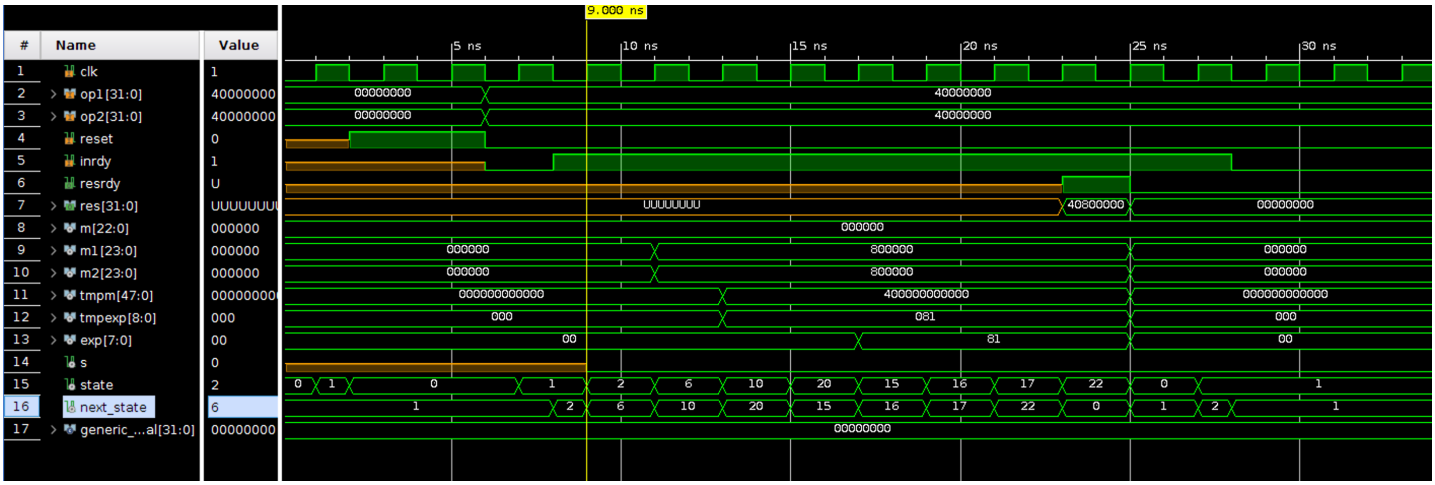


Figure 9. Vivado Simulation Example nr.4, Zoom inside the VHDL multiplier component

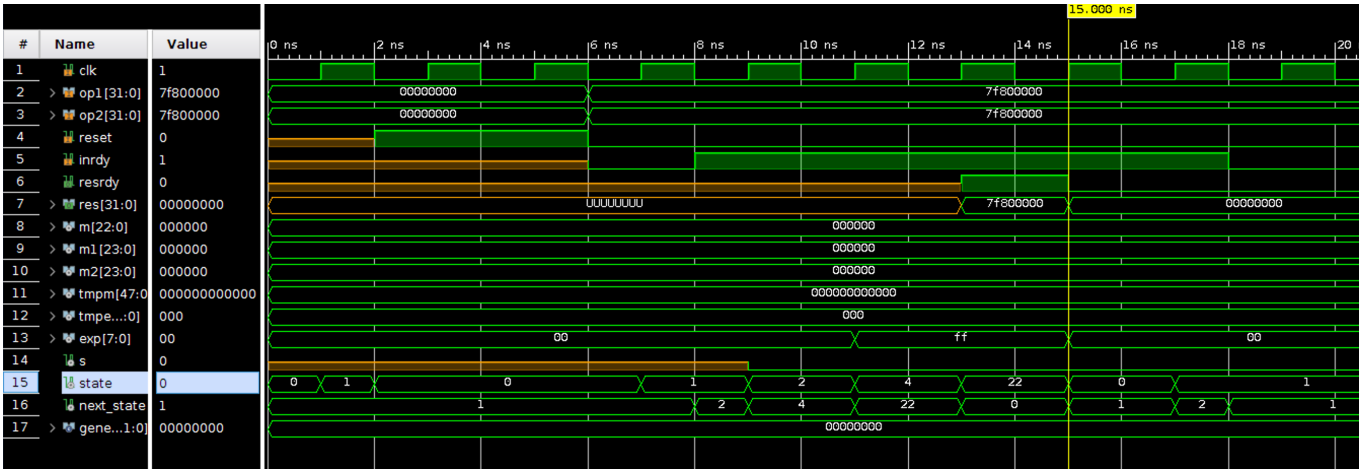


Figure 10. Vivado Simulation Example nr.5, Zoom inside the VHDL multiplier component