TLM description and integration on a Virtual Platform of a Single-Precision Floating-point IEEE 754 Multiplier

Fabio Chiarani - VR445566

***Abstract*—The following document illustrates how an IEEE 754 single precision floating-point was simulated through different levels of abstraction with SystemC TLM: it was then simulated through the RTL, AT, LT and UT levels. The simulation consists of random multiplication tests repeated from 10^3 to 10^9 times, reporting and commenting on the data obtained. In a second phase the multiplier defined in Verilog and VHDL was connected to an existing virtual platform (COM6502-Splatters) verifying its functionality through the Vivado simulation.**

I. Introduction

The following document illustrates how an IEEE 754 single precision floating-point was simulated through different levels of abstraction with SystemC TLM 2.0: it was then simulated through the different coding styles such as AT, LT and UT levels comparing it to the RTL (SystemC RTL description) simulation. The simulation consists of multiplication tests repeated from 10^3 to 10^9 times, reporting and commenting data obtained. We want to see how the difference in abstraction improves and/or worsens the simulation time. In a second phase the multiplier defined in Verilog and VHDL on the previus report, was connected to an existing virtual platform (COM6502-Splatters) verifying its functionality through the Vivado simulation.

The 'Background' section gives some information about SystemC TLM and the Virtual Platfom pro and cons.

The Section 'TLM Implementation' shows how the TLM descriptions is created for RTL, AT, LT and UT levels, reporting the data obtained from the simulations.

The Section 'Virtual Platform Integration' shows how the HDL multiplier is connected to the COM6502-Splat platform, reporting the data obtained from the platform simulation.

II. Background

Platform-based design is the creation of (stable) microprocessor-based architectures, which can be quickly extended and modified for a set of applications, and given to consumers for rapid development. TLM is a golden model which describes the design with the transactions (*a transaction is the transfer of data from one module to another, represented by a generic payload, through primitive functions*) annotation and makes possible the verification and simulation on a layer above the RTL. The TLM description is used for the device performance exploration, and running the software on a virtual platform (VP) of the hardware platform, it optimizes the software verification and makes a fast simulation available before the RTL implementation.

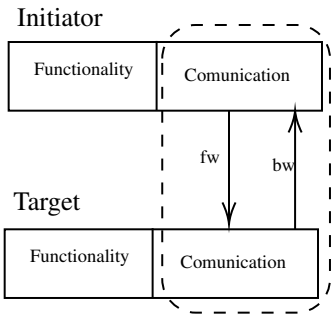
SystemC-TLM [1] is a SystemC version that supports the TLM description. The use cases, which lead to choosing a TLM solution are the development of a faster software, an architectural performance analysis and hardware verification. SystemC-TLM provides different code-styles (which are standardized on SystemC-TLM 2.0):

- LT (Loosely-timed): that has the details of how long it takes to boot up an OS and run a multicore system. It has a limited number of context-switch to 2: start and end of a transaction.
- AT (Approximately-timed): it is suitable for performance analysis and architectural exploration. The processes follow step by step with the simulation time and have 4 events: start and end of the request and, start and end of the response.

Another timing annotation is the UT (untimed) where time is not important. SystemC-TLM provides different interfaces based on the choice of the style of code adopted for allows the transaction between the initiator and a target:

- Blocking transport: used inside LT description, which has the time annotation on the socket; it uses only the forward path.
- Non-blocking transport: used inside AT description, which has the time and the transaction-phase on the socket; it can be used both on forward and backward path. The non-blocking transport is used when you want to specify more in details the interactions between the target and the initiatior.

COM6502-Splatters is a virtual platform composed by: CPU MOS 6502 (1975) with 16 bit addressing (16KB ROM, 16KB RAM) and 8 bit data width; Memory with ROM in one single bank, RAM splitted in 8 different blocks to enable multi read/write operations and a Clock divider that manage MMIO operation between Peripherals, Memory and manage multiple write on same cell between CPU and MMIO Interface; and a BUS ARM APB (Advanced Peripheral Bus) that supports up to 8 peripherals; an IO Module used to request or send data out from the platform. The use of platforms and/or virtual platforms allows to have a more general environment as well as only the hardware component. Another feature, thanks to the SystemC TLM 2.0 standard, let to reuse and/or use components already created to the platform reducing the time to market.



III. TLM Implementation

After analyzing the required specifications, it has been chosen to implement first the TLM description of the IEEE754-multiplier with SystemC, and later, connect the multiplier to the virtual platform and simulate it. On the TLM description, the `iostruct` used for the payloas as comunication between the initiator and the target for the TLM simulations (UT, LT, AT4) which is located inside the `define_*.h` is defined as follow:

```
struct iostruct
{
        sc_int<32> datain_op1;
        sc_int<32> datain_op2;
        sc_int<32> result;
};
```

Where `datain_op1` and `datain_op2` are the two input for the multiplier module. For handle the `uint` value and convert it to the float bit$-to-$bit, inside the same file was declared the `ieee754_single_precision` type as follow:

```
typedef union
{
        unsigned int uint;
        float floating_point;
} ieee754_single_precision;
```

Having obtained, on the previous report, wihch described the HDL implementation of the IEEE754 multiplier, through the synthesis and the timing constraints (such as clock constraint), it was found that the latency of the multiplier it is approximately of `9ns`. With that information, when the LT or AT4 style is implemented in TLM, in the target, the `timing_annotation` field is setted of how long is assumed that the target takes to run the multiplication, and in this case, having the `9ms` from the RTL level, this value is setted as follow:

```
timing_annotation += sc_time(9, SC_NS);
```

The next subsection shows the main concept for each timing annotation style and then report the data and a plot of the various simulations: the section'TLM: UT' shows the untimed implementation, 'TLM: LT' shows the loosely time implementation, 'TLM: AT4' shows the AT4 implementation, 'TLM: RTL' shows the RTL implementations, and later the section 'TLM: Report' wrap up the data and makes some comparison between the data obtained.

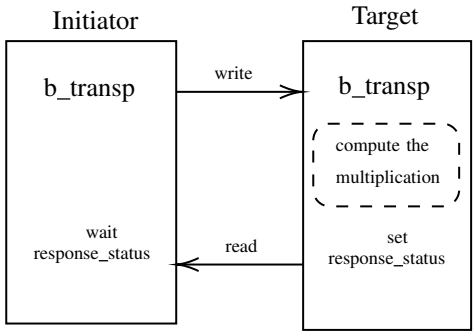


Figure 1. Schematic of the comunication at UT timing description.

Iterations	Real	User	System
10^3	0,003s	0,000s	0,003s
10^4	0,021s	0,000s	0,013s
10^5	0,206s	0,000s	0,146s
10^6	2,203s	0,016s	1,436s
10^7	24,591s	0,176s	16,320s
10^8	4m 8,303s	2,317s	2m 34,181s
10^9	44m 30,274s	20,135s	27m 22,367s

Table I
UT simulation time results.

Note that all the simulation are runned on a 8GB RAM and 16vCore (3.9GHz) virtual machine, this means that the data is scaled up compared to a common PC with lower performance.

A. TLM: UT implementation

The UT (*Untimed annotation*) specify that we don't have the notion of time, so the type of the interface used is blocking. The `multiplier_UT_testbench` module, which is the initiator, communicates with the `b_transport` primitive with the `multiplier_UT` module, which is the target. On the target, when the `b_transport` primitive is triggered in `write` mode, it will compute the functionality, instead, when the primitive is in `read` mode, it sends the computation result back to the initiator. The Figure 1 shows an abstract comunication between the two modules. The initiatior (which is the *testbench*) call the target different couples of time: from 10^3 to 10^9.

Each simulation, runned with the linux `time` command, produces the values showed in Table I. A plot of the values is showed on the Figure 2.

B. TLM: LT implementation

The LT (*loosely timed*) coding style, use blocking interface, and two synchronization points (which is invocation and return) and the temporal decoupling. This style allows a faster simulations than the RTL. The `multiplier_LT_testbench` module, which is the initiator, use the `b_transport` primitive as same as the UT implementation, but in this implementation we have the notion of $time$ for the synchronization: when the `b_transport` primitive of the target (`multiplier_LT`) is triggered, it

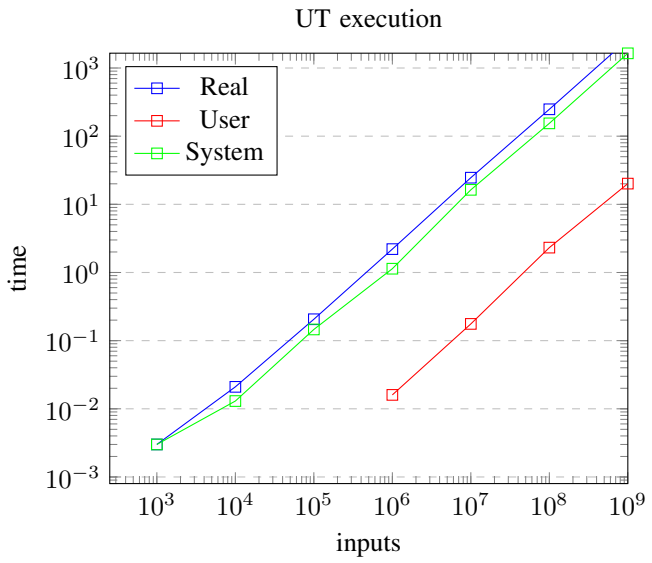


Figure 2. Plot of the Table I expressed in logaritmic scale.

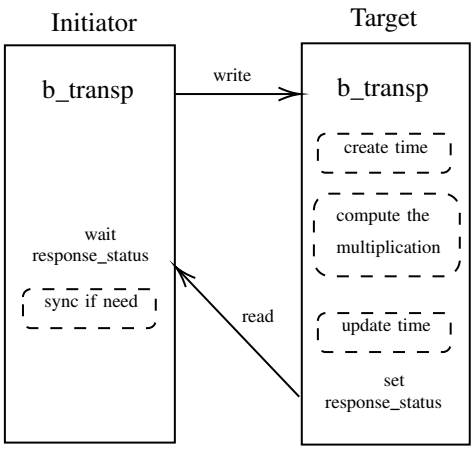


Figure 3. Schematic of the comunication at LT timing description.

create a `SC_ZERO_TIME` variable before the computation process, and on the *read* phase, it pass back this variable to the initiator with the value of `9ms` as expected timing for the computation. At this moment, the initiator that waits for the target response, syncs the comunication if it is required, and resumes back to an another executions. The Figure 3 shows an abstract comunication between the two modules. The initiatior (which is the *testbench*) calls the target different couples of time: from 10^3 to 10^9. Each simulation, produces the values showed in Table II. A plot of the values is showed in the Figure 6.

C. TLM: AT4 implementation

The AT (*approximately timed*) coding style, uses the non blocking interface provided from SystemC. The AT4 is a 4-phase handshaking protocol between the initiator and the target. This means that it will support a more detailed sequences of interactions between the two modules and makes the simulation similar of what happend on the RTL level. The non

Iterations	Real	User	System
10^3	0,004s	0,000s	0,005s
10^4	0,028s	0,000s	0,023s
10^5	0,266s	0,000s	0,206s
10^6	2,658s	0,012s	1,888s
10^7	27,591s	0,325s	18,040s
10^8	4m 43,521s	3,510s	3m 2,549s
10^9	47m 33,280s	33,321s	29m 39,531s

Table II
LT SIMULATION TIME RESULTS.

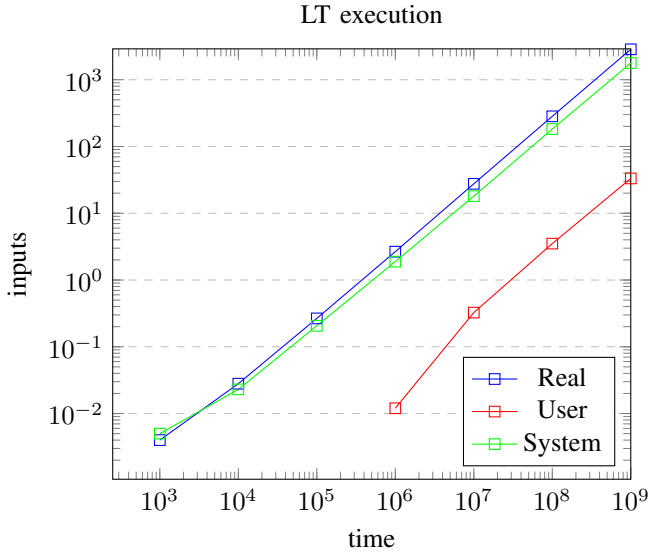


Figure 4. Graph rappresentationf ot the Table II

blocking transport interface, means that the module after the call of the transport primitive, instead of blocking it, will continue. The `multiplier_LT_testbench` module, which is the initiator, use the `nb_transport_fw` primitive to call the target, and then call the `wait(event)` primitive to wait the target that send back the data on the `nb_transport_bw` function when is ready. On the other side, the target, when the `nb_transport_fw` is triggered it calls the `ioprocess()`, unlocked by the `notify(t_event)` primitive after the computation process. This create a callback to the initiator.

The Figure 5 shows an abstract comunication between the two modules. The initiatior (which is the *testbench*) call the target different couples of time: from 10^3 to 10^9.

Each simulation produces the values showed in Table III. A plot of the values is showed in the Figure 6.

Iterations	Real	User	System
10^3	0,004s	0,000s	0,004s
10^4	0,029s	0,000s	0,029s
10^5	0,323s	0,000s	0,287s
10^6	3,456s	0,052s	2,768s
10^7	35,571s	0,558s	27,637s
10^8	5m 54,022s	7,184s	4m 21,377s
10^9	61m 0,820s	1m 20,609s	44m 25,017s

Table III
AT4 SIMULATION TIME RESULTS.

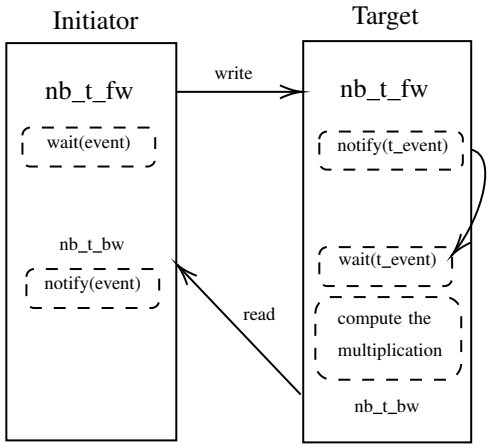


Figure 5. Schematic of the comunication at AT4 timing description.

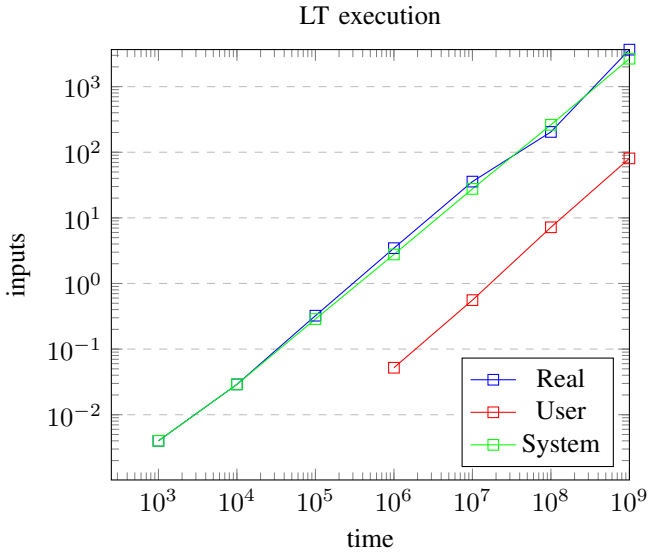


Figure 6. Plot of the Table II expressed in logaritmic scale.

D. TLM: RTL implementation

The SystemC-RTL implementation of the IEE754 multiplier, described in details on the previous report, it is tested with the same simulation of UT, LT and AT4, producing the values reported on the Table IV.

E. TLM: report

From the previous sections, is possible to view how the simulation time change from the UT to the RTL. The Figure

Iterations	Real	User	System
10^3	0,045s	0,005s	0,041s
10^4	0,430s	0,080s	0,350
10^5	4,147s	0,256s	3,890s
10^6	39,069s	3,732s	35,337s
10^7	6m 33,900s	33,844s	5m 59,745s
10^8	65m 52,598s	5m 22,510s	60m 30,056s
10^9	716m 38,452s	45m 42,970	670m 54,149s

Table IV
RTL SIMULATION TIME RESULTS.

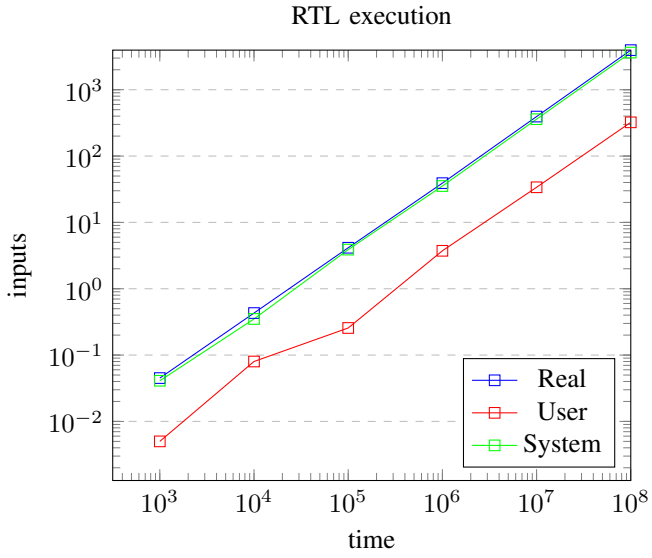


Figure 7. Plot of the Table IV expressed in logaritmic scale.

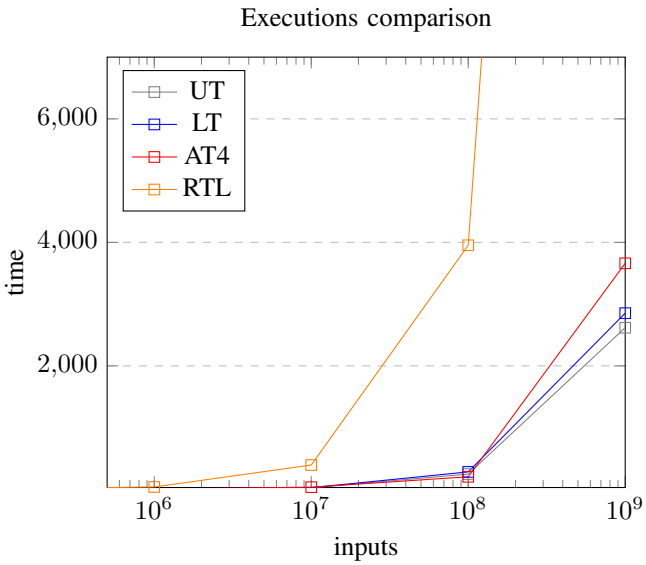


Figure 8. Plot of the different 'Real' time annotation with all TLM code styles.

8 displays the differences between the UT, LT, AT4 and RTL. The AT4 description, having a better description of the specification, and a detailed handshake protocol for the comunication between the target and initiator, it wil decrease the simulation speed compared with UT. But the greater detachment is shown with the simulation of 10^9 cycles by RTL. The system simulation time changed from 44m for UT up to 716m (ca.12h) for RTL. This means that in TLM, is optimal for simulation with and abstract description respect the RTL level, and with that information, we have to take care of the relationship between the module refining and the simulation speed when we are working on the hardware design.

The Figures [13, 14, 15, 16] show the CPU and memory usage during each simulation.

IV. Virtual Platform Integration

The virtual platform used in this project, showed in Figure 10, is composed by a CPU MOS 6502 (1976), with 16bit addressing and 8bit data width, a Memory block composed by ROM, RAM and clock divided (for manage the MIMO operations and memory), a APB ARM BUS, which supports 8 peripherals, and an I/O Moduled which is used for I/O from/to the platform.

A. VP: Integration

This VP allows to load a cross-compiled software and run a software with the I/O integration. Is choosed not to connect the IEEE754 multiplicator top level created on the previous report, but to connect directly the two VHDL and Verilog IEEE754 multiplicator as peripheral on the platform.

This choice was determined by the fact that it was wanted to test the VHDL and Verilog multipliers independently as if they were two slaves connected to the AMBA BUS, firstly checking that the VP is working with more peripheral connected, and then use on the other side of the AMBA BUS, the cross-compiled software with new routines and main function that give to the multipliers the same or different inputs, using the I/O module to verify that the result obtained from them is correct or not.

During the integretion one must take care to the clock of the platform: the multipliers are working with a 32bit data, but the CPU and Memory communicate with a 8bit. This means that the clock speed of the CPU is 4 times faster than the clock of the multiplier.

This kind of implementation described above can be confusing for a moment, so let's explain step by step the integration flow:

B. VP: Cross-compiled software

The project structure is shown on the follow directory tree:

```
root (splatters)
├─ application
├─ platform
└─ cc65
```

The `application` folder contains the sources of the cross-compiled software. `platform` contains the Vivado project (with the .v and .vhdl sources), used for simulation. `cc65` contains the compiler.
Because is chosen to link two multipliers on the VP, two routines was added on the cross-compiled software in `routines.c`: one for the verilog multiplier, and one for the VHDL multiplier. The Verilog is connected to the peripheral 3, and the VHDL at the 4. The two routines, accept as parameter the two operand having `uint32_t` as type. The VHDL routine is showed below:

```
reset_flags();
set_pwdata(op1);
set_psel(PSEL4);
set_penable(1);

set_penable(0);
set_pwdata(op2);
set_psel(PSEL4);
set_penable(1);

while (get_pready() == 0)
   __asm__("nop");

result = get_prdata();
set_penable(0);
set_psel(NO_PSEL);
```

The `main.c` file is changed to work as testbench, using the I/O module to read and write the data:

```
[...]
test_mul_verilog(op1, op2, &res1);
test_mul_vhdl(op1, op2, &res2);
if (res1 == res2) {
  io_write(1);
} else {
  io_write(0);
}
equals = io_read();
[...]
```

The main function calls the two multiplier routines, passing the same data as input. When the multipliers finish the computations, if the data obtained (`res1` and `res2`) are equals, it means that the two multipliers work and the software write with the I/O module the value of `1`. The routines previously created, anticipate the way of how our multipliers communicate with the VP showed on the next section.

C. VP: multiplier wrapping

After the creation of the routines, the multipliers are wrapped to the VP (now working inside the `platform` folder). Is not possible to connect directly the two multipliers to the AMBA BUS, because the multipliers and the AMBA BUS have different signals and protocols of comunication. So it was created an APB Wrapper for each multiplier: one in Verilog, and one in VHDL.

The wrapper has two functions: the first one is to communicate with the AMBA BUS, and the second one is to run as toplevel for the multiplier IEEE754 module. The Figure 9 shows a schematic of our peripheral connection.

The APB wrapper component (which is the toplevel for the multiplier module) works as an EFSM, showed in Figure 11 and it has 5 states. The protocol works like that:

- The `presetn` works as reset value.
- When the `penable` signal recieved from APB, has the value of `1`, it tells to the toplevel when the first operand is ready, and it is saved on a `tmp` signal at the state `ST_1`.

- When the `penable` now switches to `0` moves the FSM to the state `ST_2`. When it returns back to `1` it means that the second operand is ready and moves the FSM to the state `ST_3`. At this time the toplevel sets the `in_rdy` with value of `1` to make the nested multiplier module start to compute the multiplication.
- Now it waits until the `res_rdy` signal of multiplier is setted as `1`, to move on the FSM at the state `ST_4` and set the `prdata` signal with the result (and also `pready` for tell the data is ready).

So, the APB Wrapper comunication protocol uses the `penable` signal for handle the first and the second operand for the multiplier module. A sequence diagram of the protocol, with the comunication between master and slave is showed in Figure 12.

Some modifications to the toplevel and testbench of the platofm are required to integrate the multiplier and the APB wrapper.

On the `top_level.v` file were added the wires to handle the data binding for the two new peripheral deivces (previus Verilog and VHDL multipliers wrappers), where $'X'$ stands for the peripheral number:

```
wire   apb_X_pclk;
wire   apb_X_presetn;
wire   apb_X_psel;
wire   apb_X_penable;
wire   apb_X_pwrite;
wire   apb_X_pready;
wire   [31:0] apb_X_pwdata;
wire   [31:0] apb_X_paddr;
wire   [31:0] apb_X_prdata;
```

Now this signals were mapped to the `amba_apb_bus` as follow:

```
amba_apb_bus bus(
        .clk(clk_div4),
        .apb_master_presetn(presetn),
        [...]
        .apb_X_pclk(apb_X_pclk),
        .apb_X_presetn(apb_X_presetn),
        .apb_X_paddr(apb_X_paddr),
        .apb_X_psel(apb_X_psel),
        .apb_X_penable(apb_X_penable),
        .apb_X_pwrite(apb_X_pwrite),
        .apb_X_pwdata(apb_X_pwdata),
        .apb_X_pready(apb_X_pready),
        .apb_X_prdata(apb_X_prdata),
        [...]
    );
```

Then the signals are mapped to the APB wrapper. The Verilog APB wrapper is reported below:

APB BUS	Top Level	Wrapper	Multiplier
apb_X_pclk	apb_X_pclk	pclk	used (mapped to clk)
apb_X_presetn	apb_X_presetn	presetn	used (mapped to rst)
apb_X_paddr	apb_X_paddr	paddr	not used
apb_X_psel	apb_X_psel	psel	not used
apb_X_penable	apb_X_penable	penable	used
apb_X_pwrite	apb_X_pwrite	pwrite	not used
apb_X_pwdata	apb_X_pwdata	pwdata	used (mapped to op1, op2)
apb_X_pready	apb_X_pready	pready	used
apb_X_prdata	apb_X_prdata	prdata	used (mapped to res)

Table V
VP SIGNALS MAPPING BETWEEN MODULES.

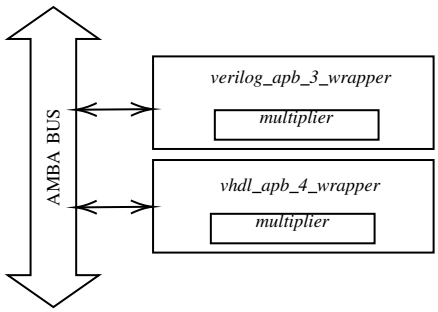


Figure 9. APB wrapper schema

```
mul_verilog_apb_wrapper mul_v(
        .pclk(apb_3_pclk),
        .presetn(apb_3_presetn),
        .paddr(apb_3_paddr),
        .psel(apb_3_psel),
        .penable(apb_3_penable),
        .pwrite(apb_3_pwrite),
        .pwdata(apb_3_pwdata),
        .pready(apb_3_pready),
        .prdata(apb_3_prdata));
```

That's all. Now the multiplier wrapper is connected to the AMBA BUS, and is able to receive the signals and command previously created on the cross-compiled software (inside the *routines.c*). Even if in the cross-compiled software we make the check for the multiplier correctness that write the value of `1` if the results are the same, on the `tb.v` file we check that the I/O module is writing on it. For that, this row were added on the `tb.v` that prints the value writed on I/O module from the software, displayed on the Vivado tlc console:

```
while(~dout_rdy)begin #20; end
$display("Out: %d ", dout);
```

The Table V shows how the signals are mapped togheder in the platform. The next section will report the simulation of the VP.

D. VP: simulation

For simulating the platform, firstly the software is cross-compiled with the cc65, that produces a `rom.mem` file which is imported in Vivado as a simulation source. Then the simulation was run. The value of op1 is 3 (*hex: 0x40400000*) and the value of op2 is 2 (*hex: 0x40000000*), expecting 6

(*hex: 0x40c00000*) as result.

The Figure 17 displays the waveforms of the simulation for the peripherical number 3, which is the verilog APB wrapper module. As we can see, on row 18 there is the `psel` signal which indicate that the peripheral is selected, then at row 19 there is the `penable` signal, and its change from value of 0 to 1 determines the state changes of the FSM of the wrapper that can be viewed at row 25. The vertical marker (*yellow*) shows the time where the peripherical start moving on the FSM excuting its process. The Figure 18 report a *zoom-in* when the APB wrapper is ready to return back the result, that can be viewed at row 29. Even if at the row 29 is difficult to see the result obtained from the multiplier, at the row 23 the `prdata` assume the value of *0x40c00000* which is our correct result!

Another thing that can be see from the Figure 18 its the difference from the `pclk` of the APB wrapper (row 15) and the clk of the CPU (row 1): the CPU clock is 4 times faster of the APB wrapper. This cause the delay between the first time the `penable` goes up to 1 and the second time for transfer the data.

The vhdl wrapper simulation is the same as the one already observed; The Figure 19 shows the waveforms and correctness of the binding signals between the AMBA BUS and the APB wrapper. Meanwhile the Figure 20 displays an entire simulation with the most important signals.

Finally, after the execution of the simulation, on the tlc console is possible to view that the cross-compiled software checks correctly the multiplier result and writes the check valus on the IO module readed on the testbench:

```
restart
run 2ms
Out: 1
```

Where 1 means that its multiplication results is correct.

V. Conclusion

The table on the right speaks for itself: SystemC simulation is optimal for time to market, co-design and model verification, improving with the virtual platforms the possibility to reuse IPs and/or integrate new ones already existing, but you can easily notice from the data obtained that a description at TLM or RTL level changes considerably the simulation time. The choice to describe the operation of the HW component through LT or AT transitions is a choice of the level of refinement you want to go on: in the case that you choose a minor detail and a simulation closer to the RTL, but you don't want to simulate at RTL level, or you don't have the RTL description, through SystemC-TLM and AT (AT4) style you can find a good compromise between development speed, simulation and verification compared to a module described at RTL level with an HDL language.

Values	Real				User				System			
	UT	LT	AT4	RTL	UT	LT	AT4	RTL	UT	LT	AT4	RTL
10^3	0,003s	0,004s	0,004s	0,045s	0,000s	0,000s	0,000s	0,005s	0,003s	0,005s	0,004s	0,041s
10^4	0,021s	0,028s	0,029s	0,430s	0,000s	0,000s	0,000s	0,080s	0,013s	0,023s	0,029s	0,350s
10^5	0,206s	0,266s	0,323s	4,147s	0,000s	0,000s	0,000s	0,256s	0,146s	0,206s	0,287s	3,890
10^6	2,203s	2,658s	3,456s	39,069s	0,016s	0,012s	0,052s	3,732s	1,436s	1,888s	2,768s	35,337s
10^7	24,591s	27,591s	35,571s	6m 33,9s	0,176s	0,325s	0,558s	33,844s	16,320s	18,040s	27,637s	5m 59,7s
10^8	4m 8,3s	4m 43,5s	5m 54,0s	65m 52,5s	2,317s	3,510s	7,184s	5m 22,5s	2m 34,1s	3m 2,5s	4m 21,3s	60m 30,0s
10^9	44m 30,2s	47m 33,2s	61m 0,8s	716m 38s	20,135s	33,321s	1m 20,6s	45m 42s	27m 22,3s	29m 39,5s	44m 25,0s	670m 54s

References

[1] Accellera. Systemc tlm. [Online]. Available: https://www.accellera.org/community/systemc/about-systemc-tlm

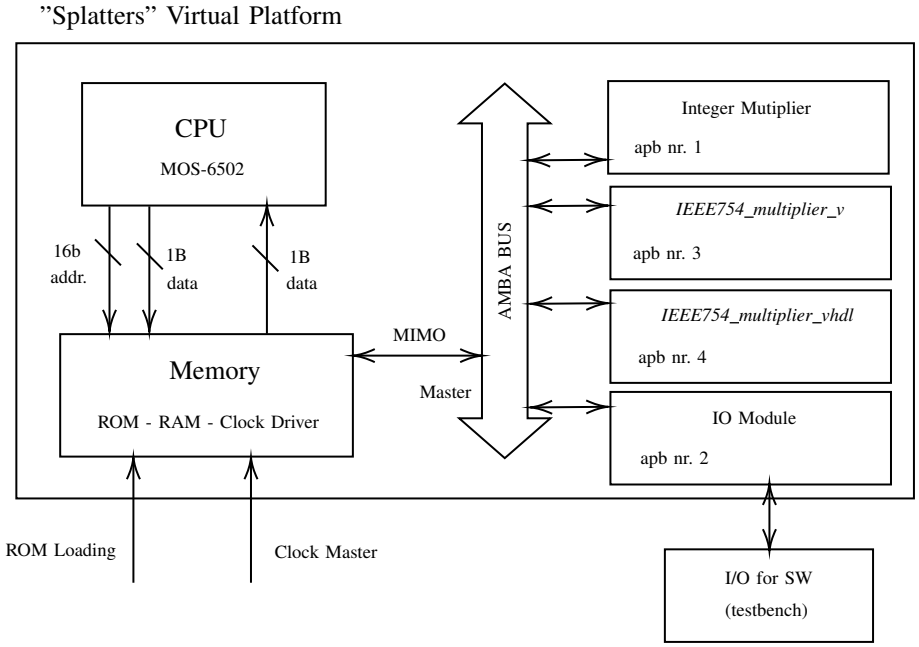


Figure 10. Schematic of the Virtual Platform ”splatters”.

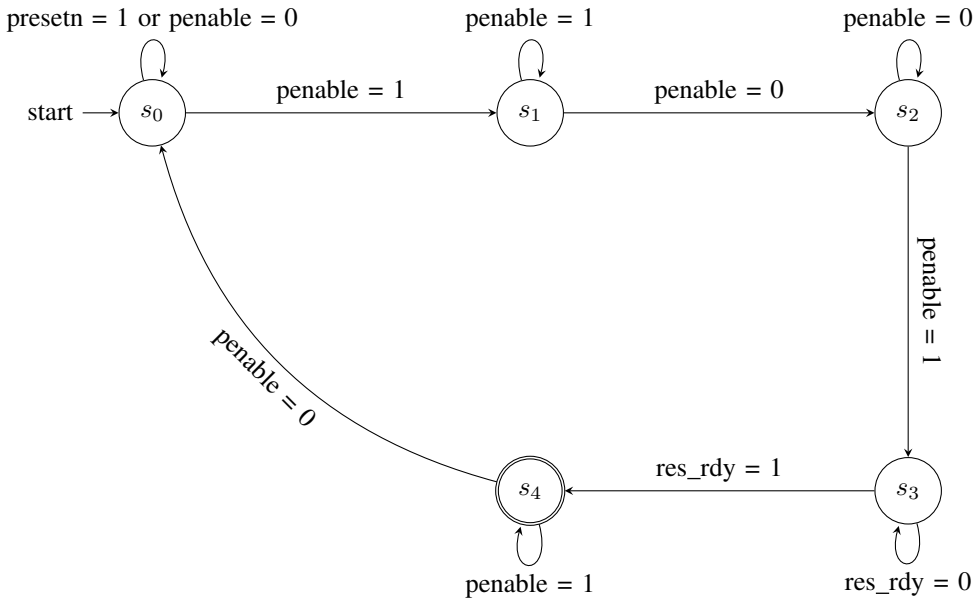


Figure 11. Wrapper Toplevel

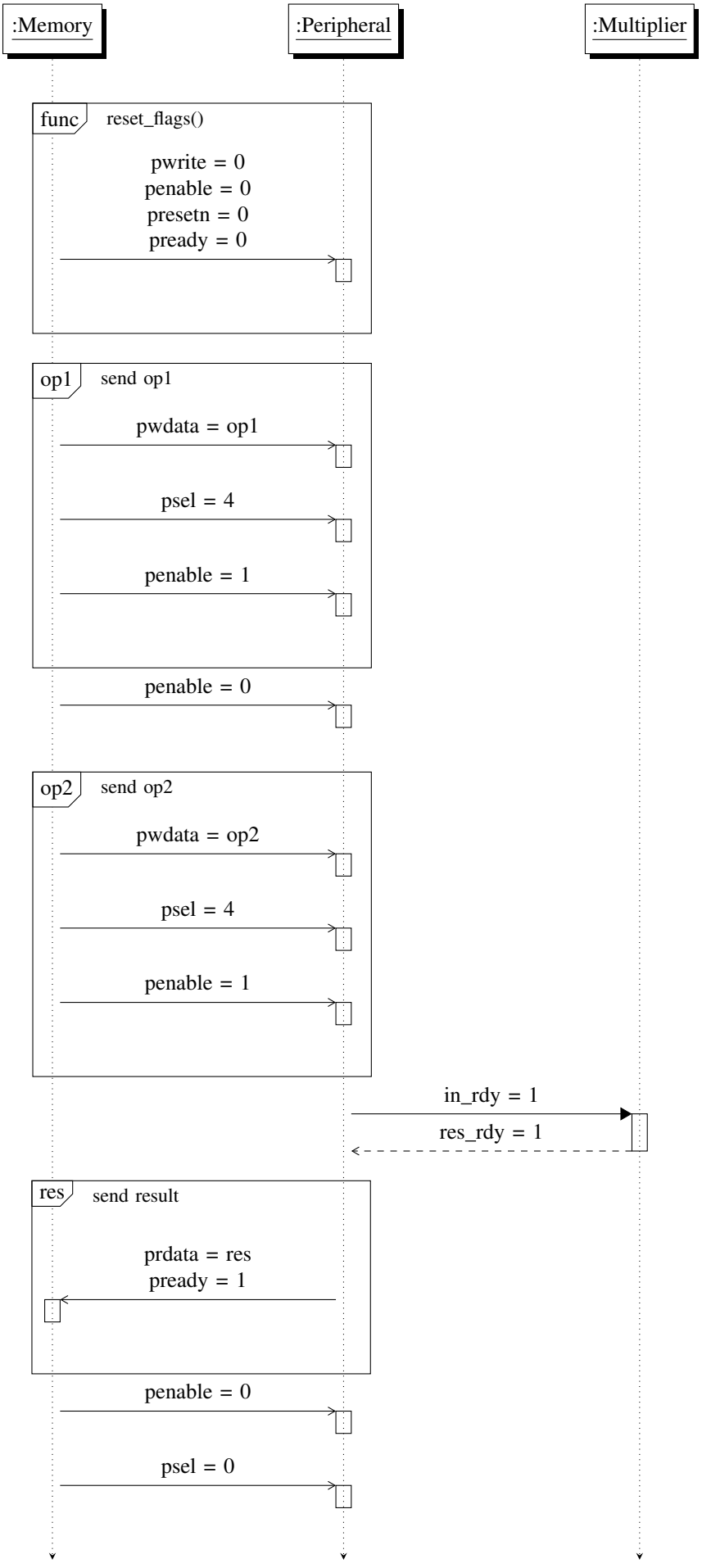


Figure 12. Protocol Sequence Diagram

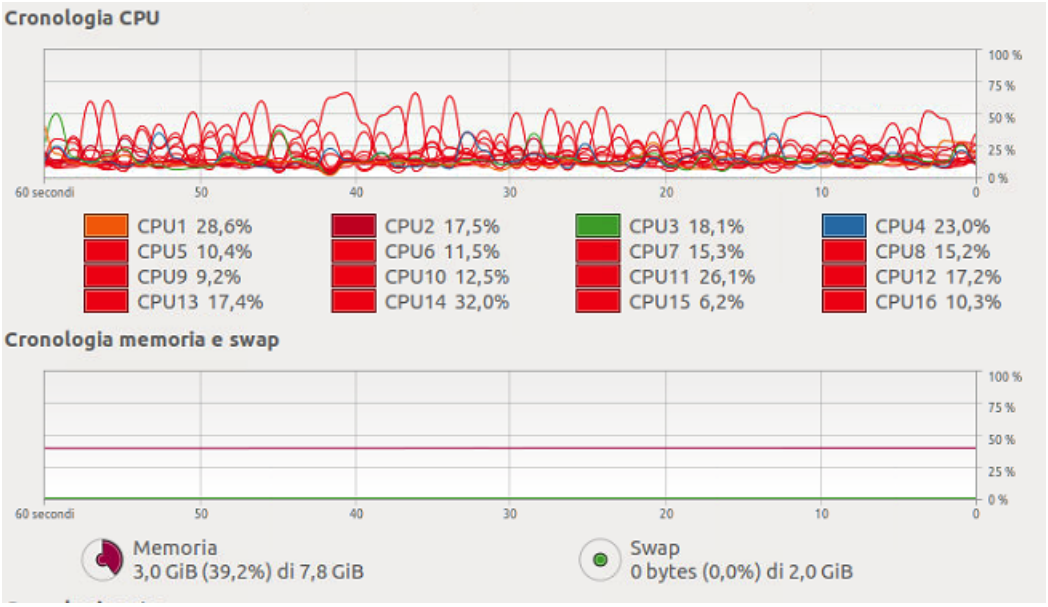


Figure 13. CPU screenshoot during 10^9 simulation with TLM-UT

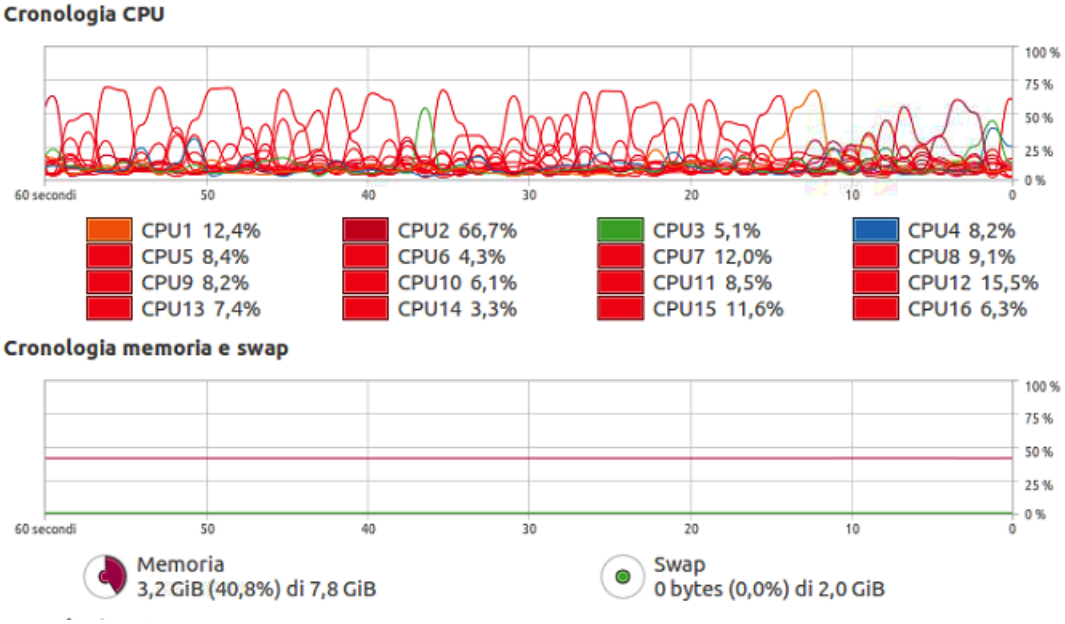


Figure 14. CPU screenshoot during 10^9 simulation with TLM-LT

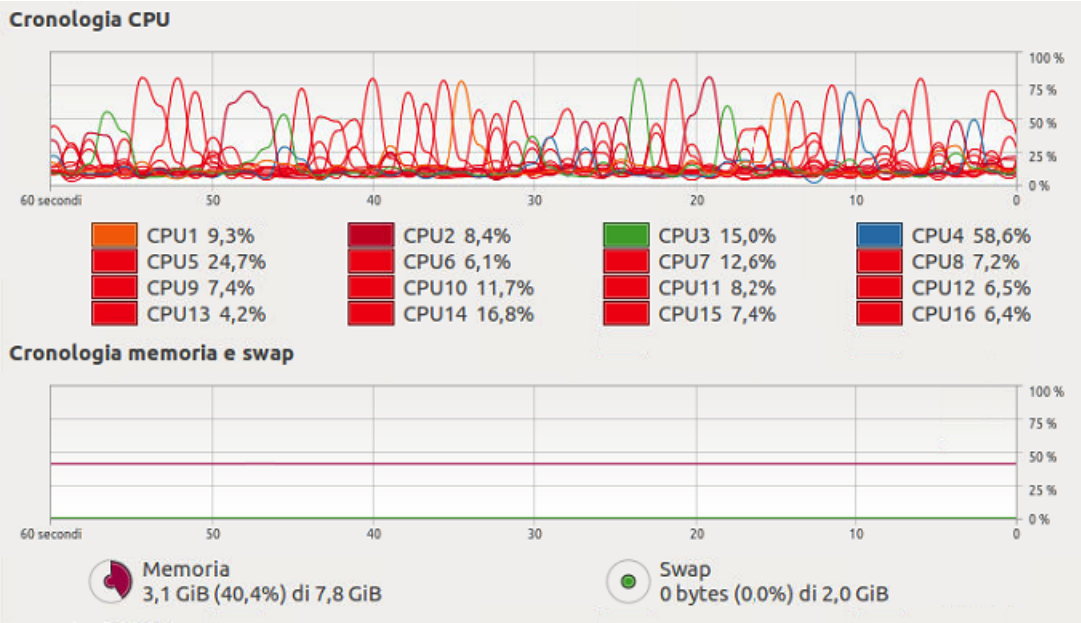


Figure 15. CPU screenshoot during 10^9 simulation with TLM-LT

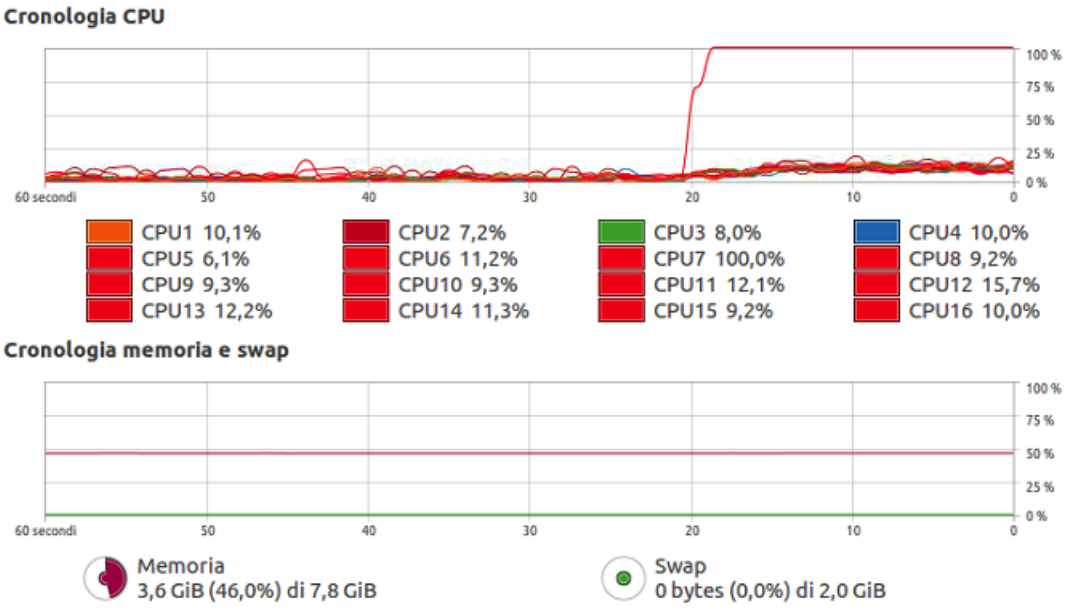


Figure 16. CPU screenshoot during 10^9 simulation with TLM-RTL

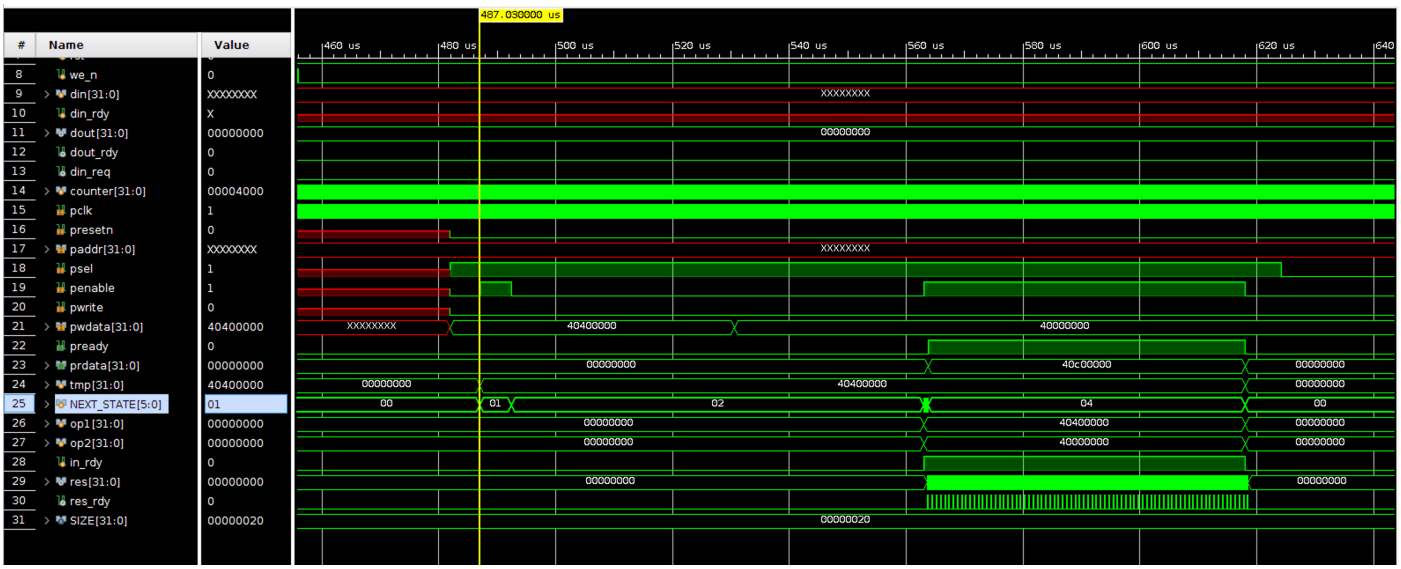


Figure 17. Waves of verilog

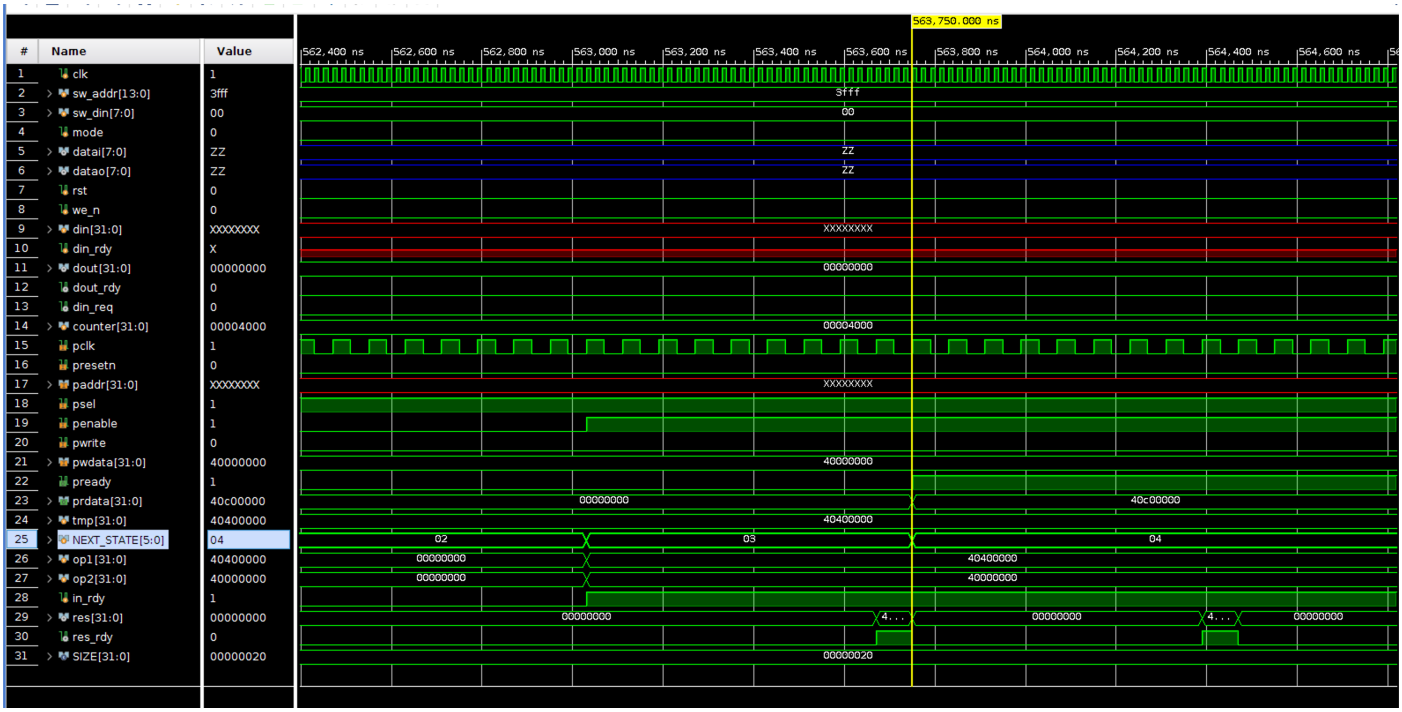


Figure 18. Waves of verilog zoom

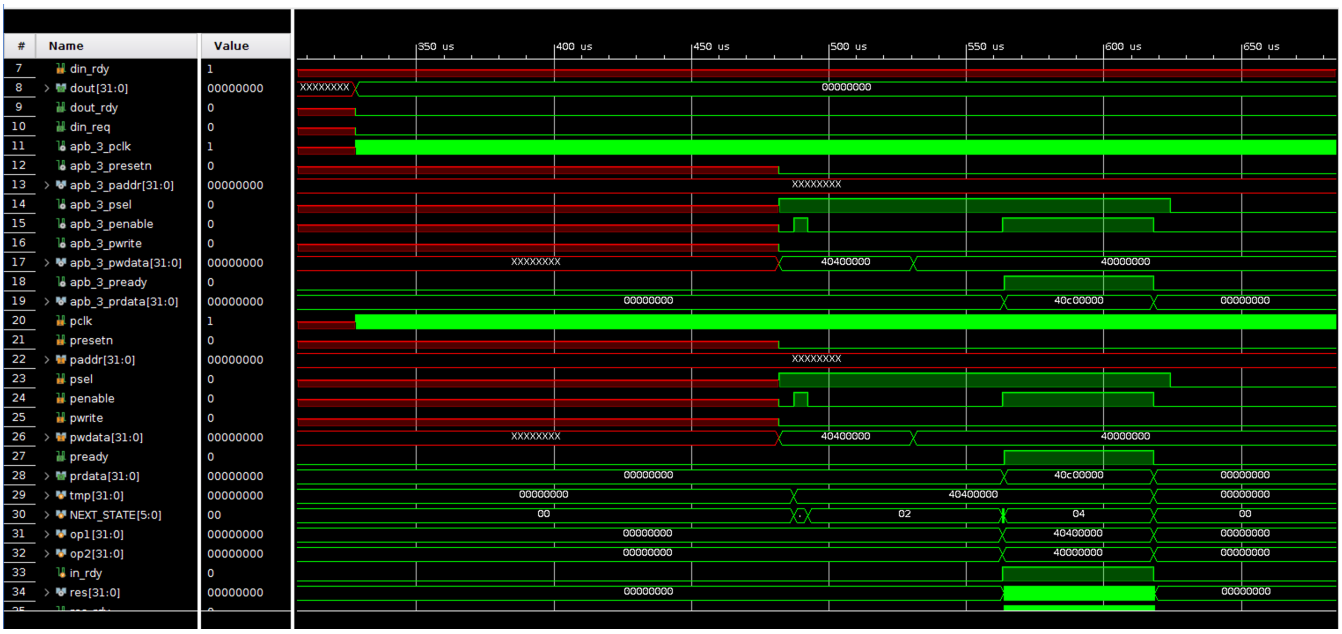


Figure 19. Waves of verilog zoom

Figure 20. Waves of verilog zoom