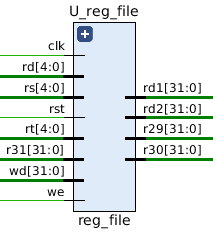
2.2.x. Reg\_File

Contributor: Fengyang Jiang (fj483)

2.2.x.1. Implementation

Block Diagram:



There are mainly these ports in Reg\_File:

In Ports:

* clk (clock signal) and rst (reset signal)
* rs, rt, rd: 5-bit inputs, which represent the address in Register File
* wd: 32-bit input, which contain the data that should be written into reg(rd)
* we: Write Enable signal
* r31: 32-bit input, for BTN & SW read-only

Out Ports:

* r29: 32-bit output, for LED Display
* r30: 32-bit output, for 7-Seg Display
* rd1, rd2: 32-bit outputs, which contain the data read from reg(rs) and reg(rt)

This component (Register File) is used to stage data between memory and the functional units in CPU.

The main functions of Reg\_File include:

* Asynchronous read: rd1 is always equal to the 32-bit data in the Register File, whose address is given by rs; and rd2 is always equal to the 32-bit data in RF with addres rt.
* Synchronous write: When clock rising edge, if the write enable signal “we” is high, the 32-bit data in wd will be written to RF with the address rd.
* The 32-bit output r29 is always equal to the value in reg(29), which will be used for LED Display. If we want to display something by LED, we should copy that value to reg(29).
* The 32-bit output r30 is always equal to the value in reg(30), used for 7-Seg Display. Similar to r29.
* The 32-bit input r31 contains the value obtained from SW and BTN buttons. reg(31) is always equal to r31, and cannot be modified by other operations (read-only).

2.2.x.2. Test Bench

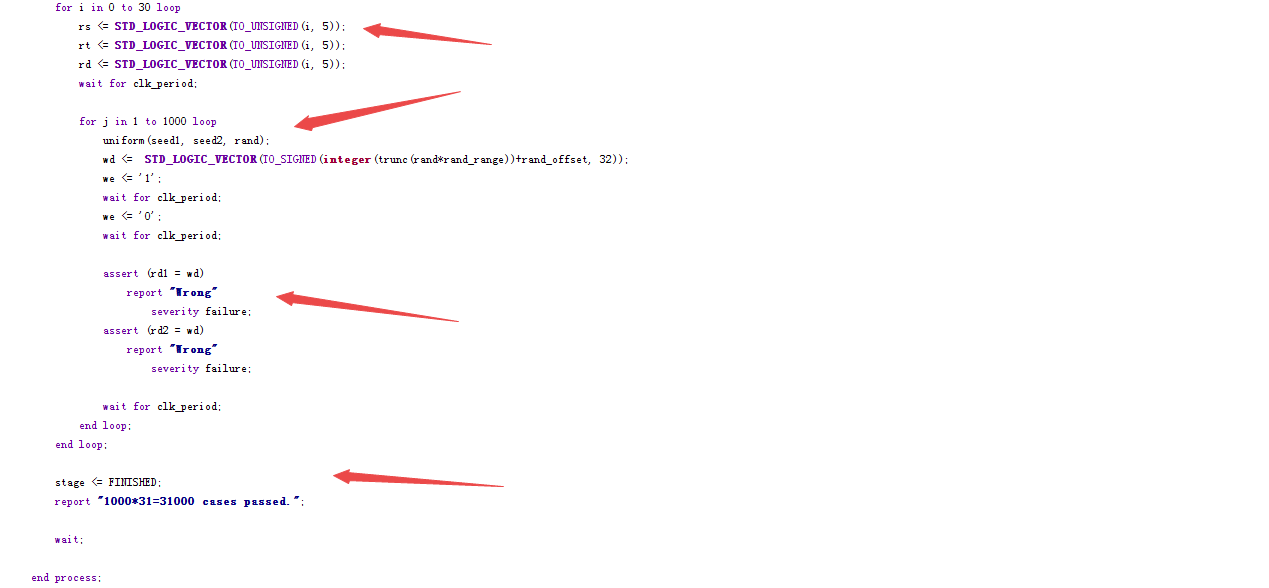
The checking operations in my test bench are described below:

* Let rs, rt, rd <= “00000” -- point to reg(0)
* Generate a random 32-bit number and let wd <= this number -- a random value to be written
* Set we <= ‘1’ to perform write function -- write the random value into reg(rd)
* Check if rd1 and rd2 are equal to wd -- check the values read from rs and rt

Then repeat 2-4 to generate 1000 random numbers in total to check if the read values are equal to the written values in all the 1000 cases.

After these 1000 case, let rs, rt and rd point to reg(1) (“00001”) and do another 1000 random cases, then reg(2), reg(3), …… until reg(30). (reg(31) is not used for write operation)

So there are totally 31\*1000 = 31000 cases checked in this test bench. If anything goes wrong during the simulation, it will stop and report “Wrong”. If all cases pass, it will show that “1000\*31 cases passed”.



* Arrow 1: Outer loop, to check every register file from reg(0) to reg(30)
* Arrow 2: Inner loop, 1000 cases for each register. In each case, write a random number into it.
* Arrow 3: Check the data read from the register.
* Arrow 4: The report message after all cases pass.

2.2.x.3. Functional Simulation

Based on the test bench described above, we can run functional simulation now.

The sequence of the signals is:

rd1, rd2, clk, rst, we, rs, rt, rd, wd, state, clock\_period

(1)

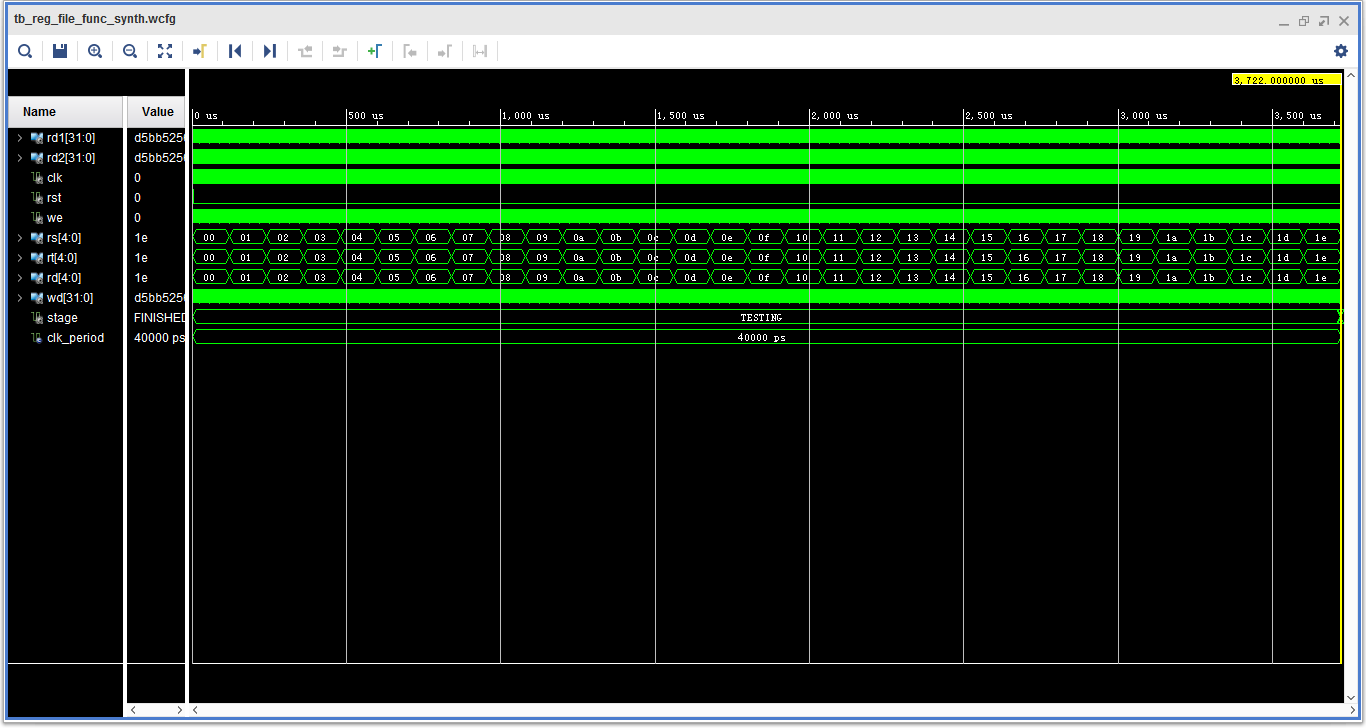


Figure 1. The whole simulation.

In this figure, we can just see the change of the addresses. In each address, there’s 1000 cases performed.

(2)

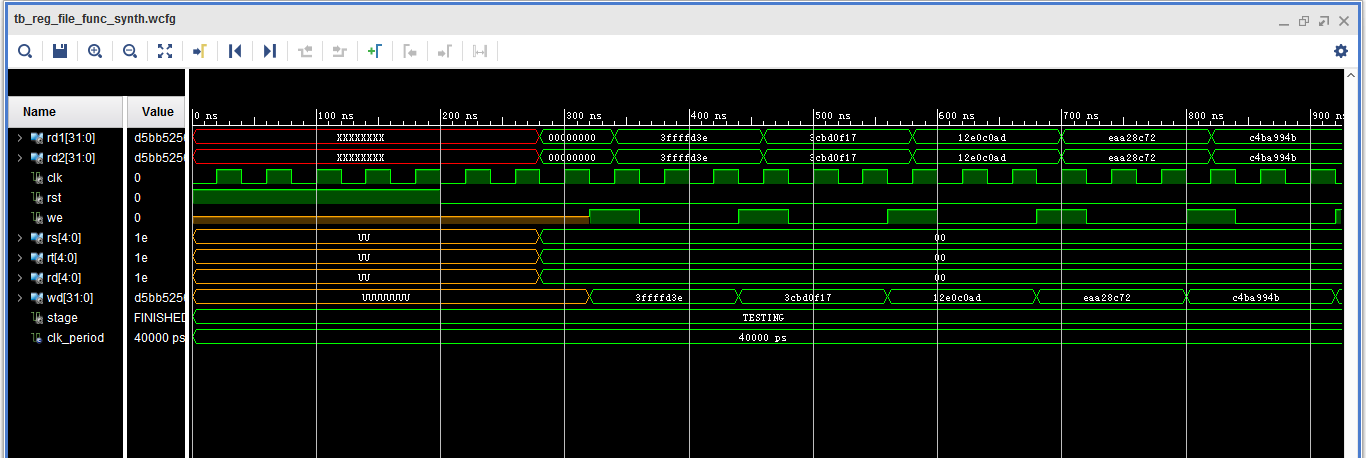


Figure 2. The first few cases

In each case, we give wd a random value, and set we signal to high. Then at the clock rising edge, rd1 and rd2 get the value which is equal to wd. (Notice that rs, rt and rd have the same value, so they point to the same address in RF)

(3)

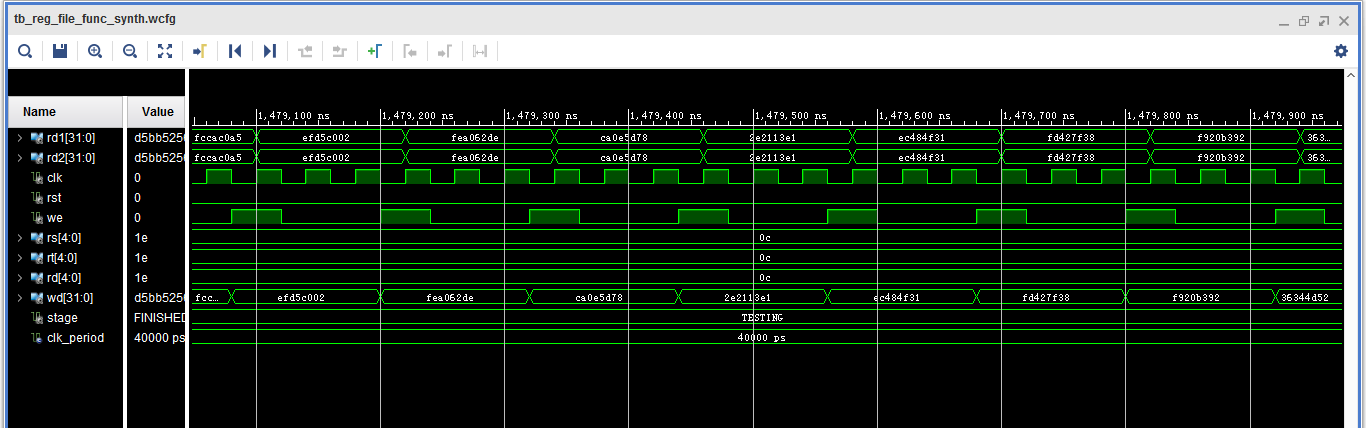


Figure 3. A few cases during the simulation

(4)

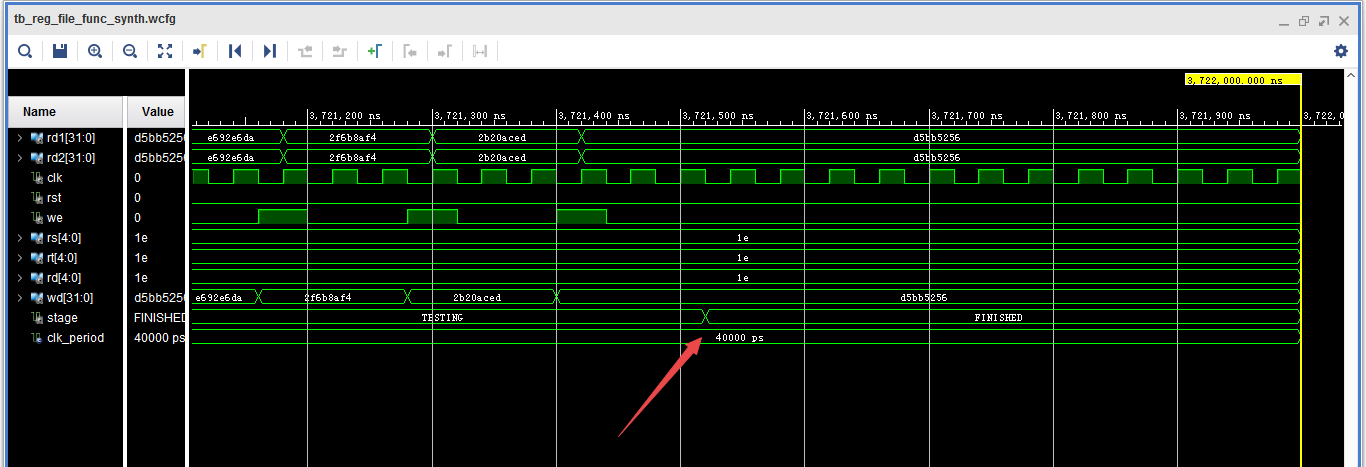


Figure 4. The last few cases.

Notice that after all the cases finished, the state will change from TESTING to FINISHED.

(5)

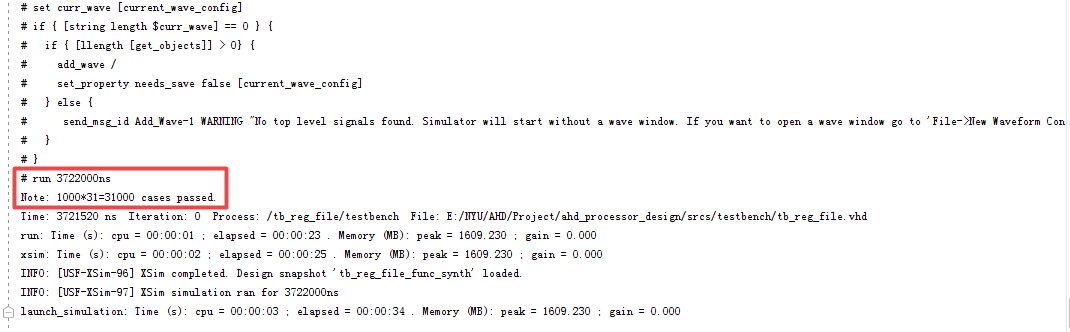


Figure 5. The report message after passing all the test cases.

At this time, if we check the report, we’ll find a “1000\*31=31000 cases passed” message. That means all our test cases succeed.

2.2.x.4. Timing Simulation

This time we will perform timing simulation on the same test bench.

(1)

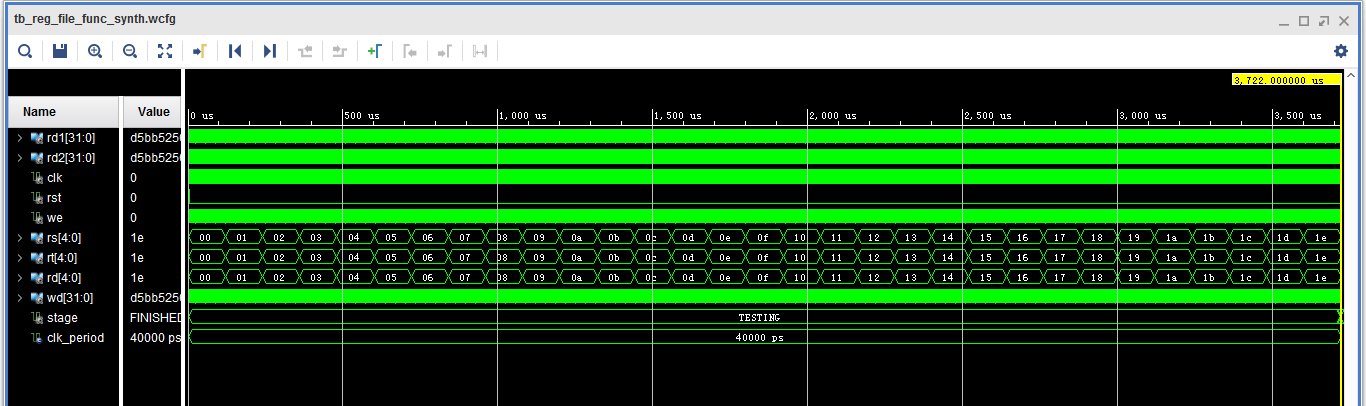


Figure 1. The whole simulation

This operations are the same as functional simulation.

(2)

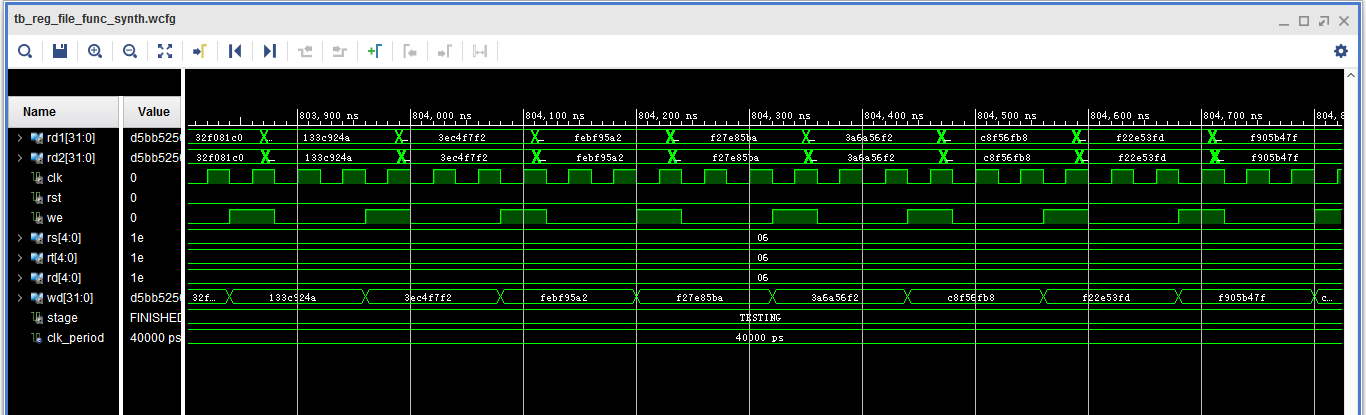


Figure 2. A few cases during the simulation

We can see that when we write different values to the register, it can be read properly.

(3)

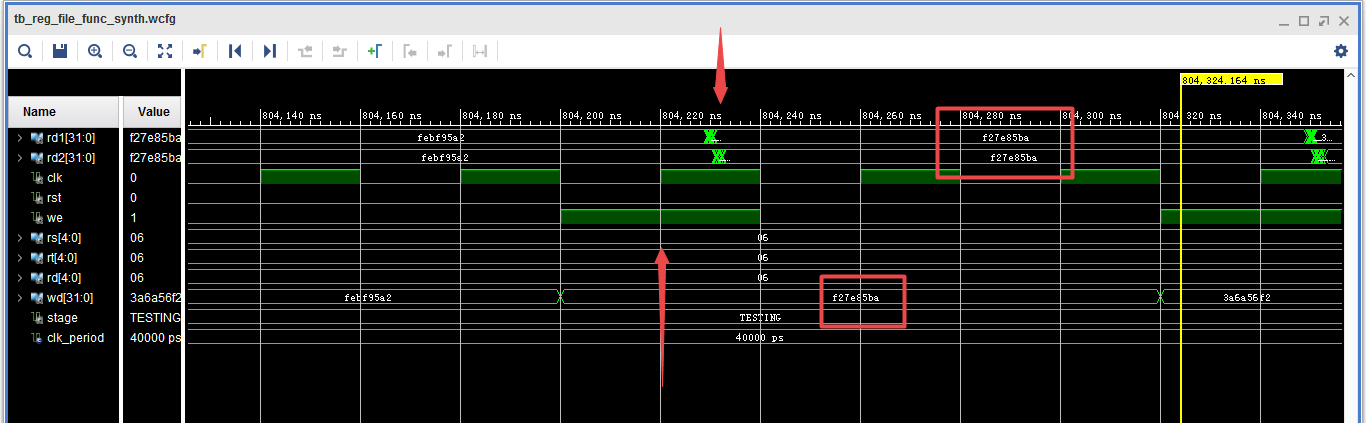


Figure 3. One case in the simulation

We can see there’s a delay (about 10ns) between the clock rising edge and the data being read, because it will take some time to write a data into RF and then read it.

(4)

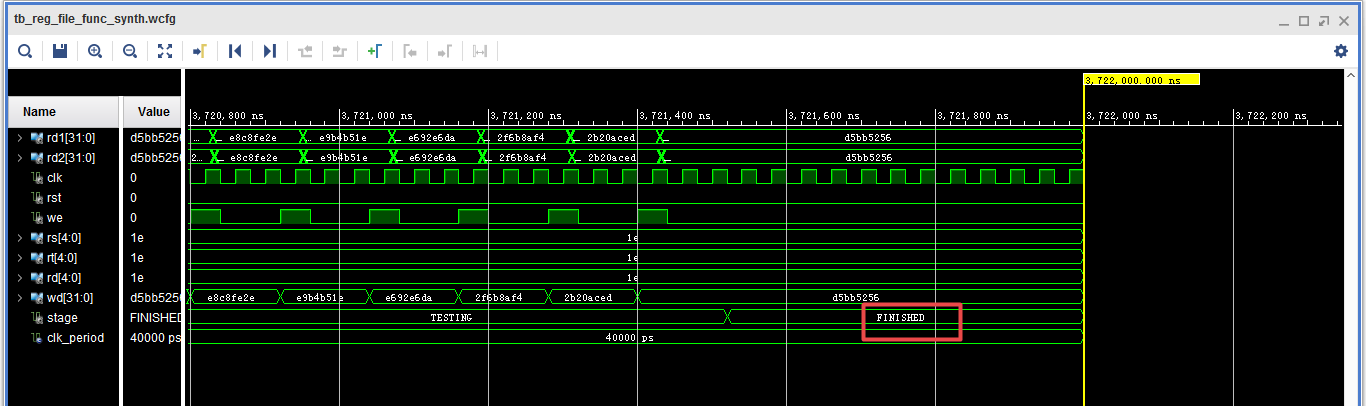


Figure 4. Last few cases

After all the cases finish, the state becomes “FINISHED”.

(5)

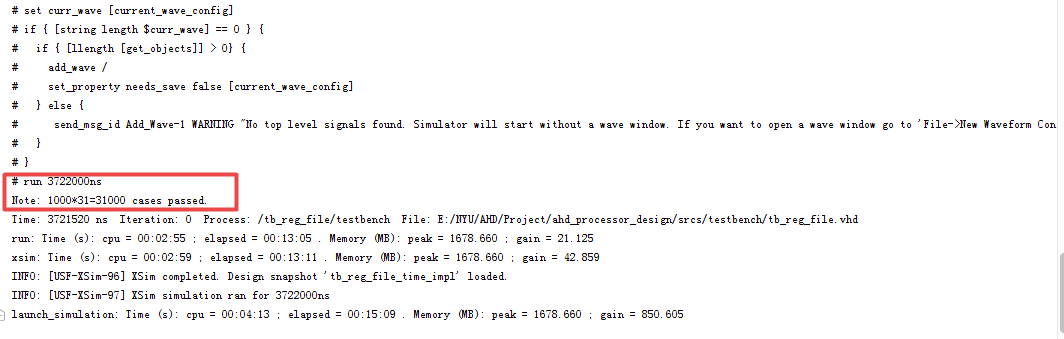
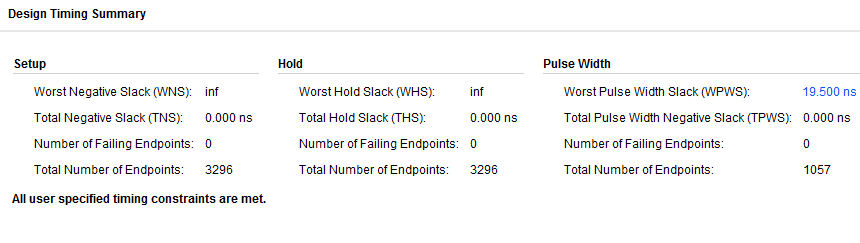


Figure 5. The report message

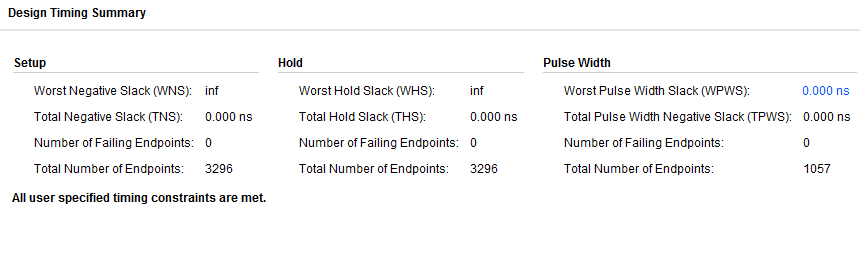
Still, when the simulation is done, a “1000\*31=31000 cases passed” message appears, which indicates the success of our test.

2.2.x.5. Delay

First we set clock period = 40ns and check the timing summary:



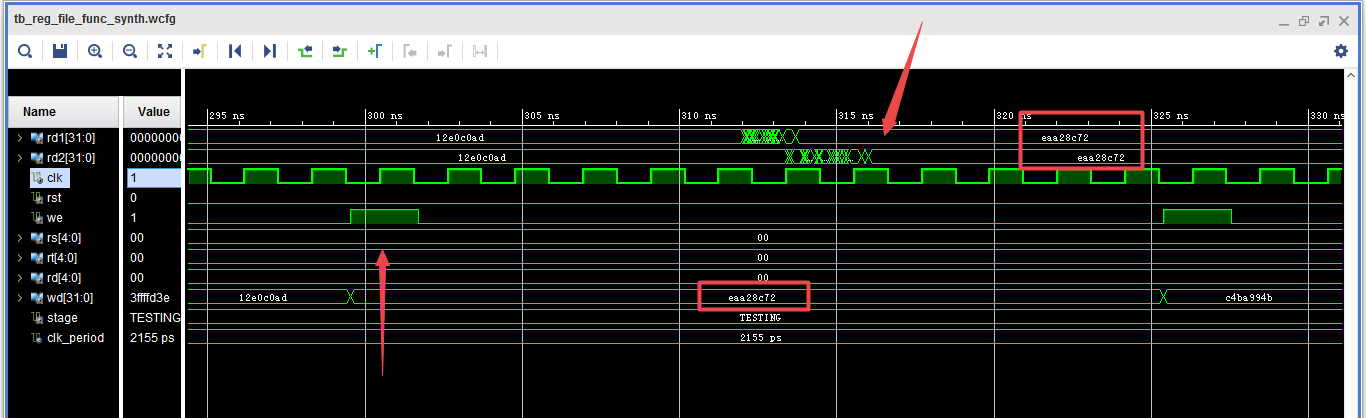
All timing constraints are met. The WPWS is positive, so we gradually decrease the clock period and check the timing summary, and finally when we set clock period = 2.155ns, we get this summary:



WPWS becomes 0ns. If we further decrease the clock period, it will fail.

So we use 2.155ns as the clock period to run timing simulation, and we could still read the correct data which has been written into the EF.

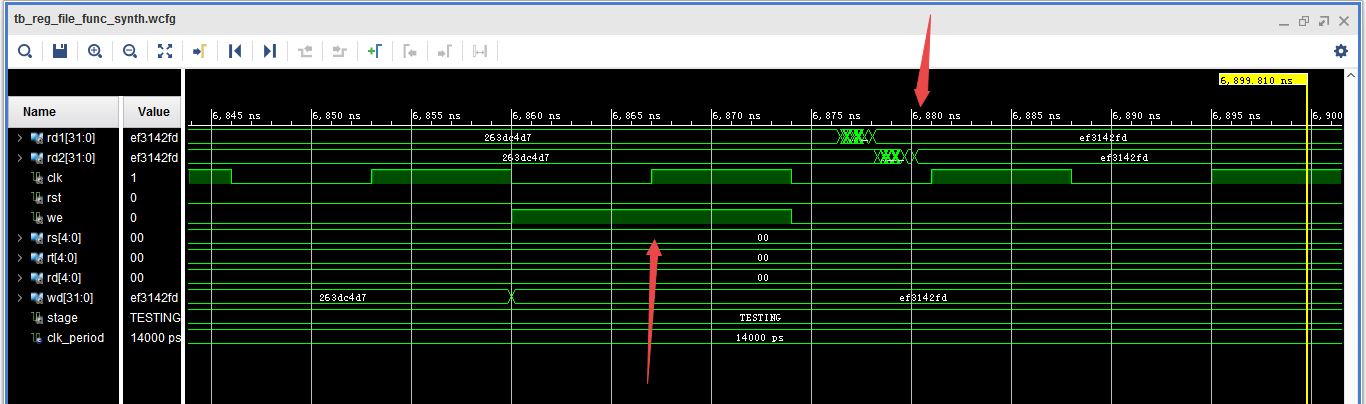
However, there’s a problem. Let us show the simulation waveform:



Although we could read the correct data, the delay between them is about 7.5 clock cycles (16ns). That means both write and read operation may take longer than one cycle to complete.

In our Top CPU, each instruction only takes one cycle to execute, thus if we write something into RF, it must be valid within one cycle, so that when the following instructions come, the RF could provide a correct value.

Since both write and read operations take time (due to the buffers), the write operation itself may take not as long as 16ns to complete. However, to ensure the function’s correctness, we choose to set clock period = write time + read time, which is around 16ns. Based on this value, we further tried several different clock period and run simulations. Finally, we choose clock period = 14ns. The result is showed below:

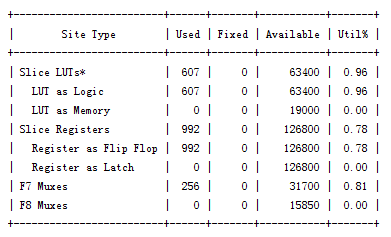


We could see this is just a safe value. So for the Reg\_File, we regard 14ns as the minimum clock period.

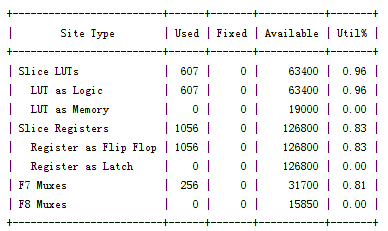
|  |  |
| --- | --- |
| Critical Path Delay | 14ns |
| Frequency | 71.43MHz |
| Latency | 1 cycle |
| Propagation Delay | 14ns |

This delay (14ns) is just for the single part Reg\_File. It indicates that the clock period in Top CPU can’t be shorter than 14ns. Actually, the clock period in top module usually takes a much longer value.

2.2.x.6. Resource Utilization



After Synthesis



After Post-Route Phase

|  |  |  |
| --- | --- | --- |
|  | Synthesis Stage | Place and Route Stage |
| Slice LUTs | 607/63400 (0.96%) | 607/63400 (0.96%) |
| FFs | 992/126800 (0.78%) | 1056/126800 (0.83%) |

Notice that we won’t report IOB usage for this single part, because all input and output signals in Reg\_File are actually internal signals in the Top CPU.