An Efficient Method to Verify Dynamic IP Clock Frequency on SOC Level by Using Auto-generated Active Clock Monitor

Jaecheon Kim[1], Gyuhong Lee[2], Yeonho Jeong[3], Hyunsun Ahn[4], Daewoo Kim[5] Seonil Brian Choi[6]

Samsung Electronics Co., Ltd., Seoul, Korea (Republic of)
([1]jcheon05.kim; [2]ggamid.lee; [3]yeonho.jeong; [4]hyunsun.ahn; [5]dw0912.kim; [6]seonilb.choi@samsung.com)

***Abstract*-This paper presents an efficient method for verifying the IP clock frequency on the Dynamic Frequency Scaling (DFS) level using debug-specialized clock monitor that we call active clock monitor (ACM). The ACM provide log based analysis for clock debugging to avoid dumping waveform which is one of the factors that increase verification turn-around-time (TAT). The ACM is based on the interface and automatically bound through the YAML format database which is including the clock tree information. This paper shows how to make ACM and how it applies to verification flow to reduce verification TAT. As a result, with ACM, simulation time is reduced by 66% and debugging point is reduced up to 75%.**

I. INTRODUCTION

As the number of IPs in SOC design is increasing to meet market demands on various user applications, the clock tree is getting more complex to provide the frequency required by each IP. In addition, Dynamic Frequency Scaling (DFS) which is one of the low power schemes requires each IP's clock frequency change. This leads to the challenge of how quickly and efficiently to verify numerous dynamic IP clock frequency to check whether the clock is supplied as intended.

In general, the monitor is used for design verification as well as clock specific verification to check function [1] [2]. However, the monitor specialized for debugging is required as well since debugging time is one of the critical factors in verification turn-around-time (TAT). In general, the waveforms used for debugging take a long time to be dumped and cause increased verification TAT. In case of clock debugging, even after getting the waveform for clock debugging, it can be taken a lot of time due to the complicated clock tree. These lead to consider two things as follows: one is to use log debugging rather than waveform debugging to reduced debugging time. The other is to provide meaningful debug information, rather than just raw data so that the user can debug the complicated clock tree effectively.

In this paper, an efficient method is proposed for verifying the clock frequency on the DFS level using debug-specialized active clock monitor (ACM). The proposed ACM provides the log information such as changed IP clock frequency and the root cause of clock change to enable log analysis for clock debugging, and internal checkers to easily check the clock in the verification scenarios. And it is automatically generated by referring to the design information of the SOC clock management unit (CMU) described in YAML format. This makes it possible to provide ACM functions without any extra effort.

II. BACKGROUND

The general debug method for clock debugging is required to dump the waveform and trace the CMU component to find the root cause. However, it takes a lot of time in debugging because the clock tree has become deeper and more complex as SOC design size increases. It is required to trace many CMU components, and if necessary, check the clock diagram document for debugging.

Fig. 1 simply shows the trace of CMU component values with waveforms for clock debugging. Generally, we start debugging from where the issue occurs. However, in the worst case, it may be faster to start tracing from the clock source, but the root cause is found before we realize it. This makes debugging time increasingly inefficient because the number of CMU components is increasing to meet the target clock frequencies required for the diverse IPs as mentioned earlier. If this debugging method is used to verify dynamic IP clock frequency according to the DFS level, verification TAT will take a lot of time because the number of checker will increase exponentially

depending on the number of IPs and DFS levels. This means that even if only a few errors occur, there are a lot of absolute errors to debug.

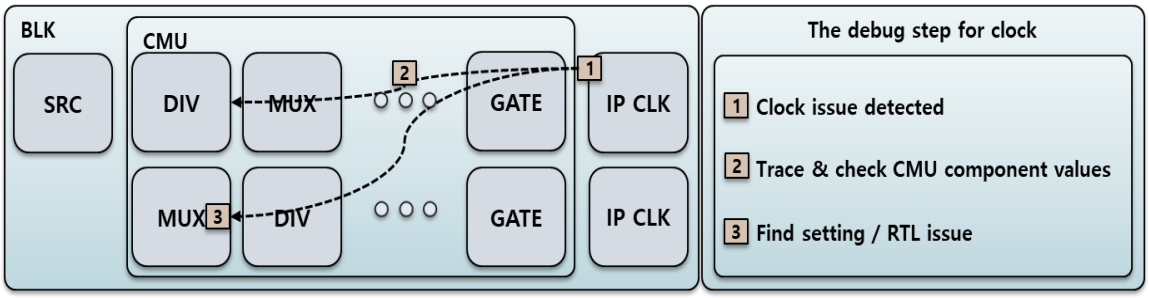

Figure 1. The debug step for clock debugging with waveform

III. PROPOSED APPROACH

This section describes the main features of the ACM and how to verify dynamic IP clock verification using ACM with some figure and example codes.

A. Active Clock Monitor

As descried earlier in introduction, the clock monitor can be used in general for clock verification for purposes of checker or debugging. The ACM is more specialized in debugging among the purposes of the monitor. It checks the operating status of the IP clock and displays the frequency changes in the log along with the change factor such as divide ratio and MUX selection. To support these features, ACM keeps a watch on the control and status information of all CMU components such as MUX, DIV and GATE between clock source and destination which can affect each IP clock. We call it as an active clock monitor because it monitors the clock frequency considering the relationship between CMU components rather than simply monitoring the clock.

Fig. 2 simply shows the behavior of ACM. The divide ratio, MUX selection and frequency values written in each box indicate the current status of each module. The log shows how the ACM works when the value changes according to step 1, 2 and 3 starting from initial state. It also provides RTL hierarchical path so that the user can know which block the CMU component is in and what module name it is. If only the IP clock has changed but any CMU components have not changed, it shows the frequency has changed just by clock source. Therefore, the frequency change of the clock source should also be monitored in consideration of this case.

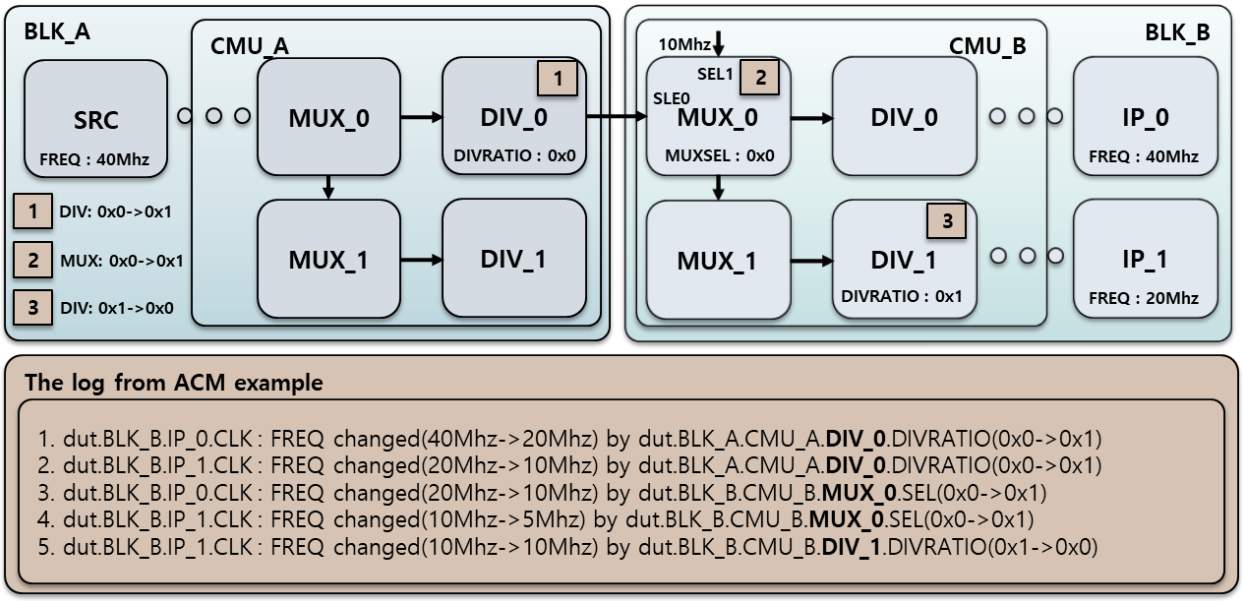

Figure 2. Example of the ACM log

The ACM always captures the clock edge of the every clock cycle in order to print the frequency value whenever clock frequency is changed. Therefore, two considerations is required to measure the exact IP clock frequency. One is to consider clock gating period by GATE component which is generally used to reduce power consumption. This requires the ACM to monitor all GATE enable signal related each IP clock. The other is to consider the period while MUX selection or divide ratio is changing because the clock can be gating while the value is changed. This means that ACM has to monitor the signals that can indicate the current status all MUX/DIV/GATE components between clock source and each IP clock. To satisfy these conditions, ACM has to be bound as shown in Fig. 3. One ACM exists for one IP clock and monitors the control value and status of all CMU components that can affect that clock.

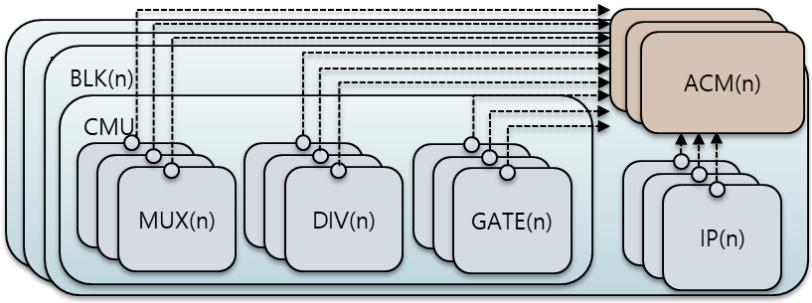

Figure 3. The overview of binding ACMs

The ACM interface instance has many RTL hierarchical paths of CMU components to get values and print RTL hierarchy as shown in Fig. 4. The parameters are used for printing the module path and indicating the number of the CMU components between clock source and IP clock. The basic operating principle of ACM is that when it is detected that the status of CMU component is changed, the log is printed if the value is different from the previous value saved and the current value. This binding is auto-generated by clock information database.

```
clkmon_intf #(.NAME(`DEFINE_PRINT(`cmu_blk_a_ip_b_clk_c)),  // IP clock hierarchical path
    .MUX0("CMU_A.MUX_A"),                                    // MUX hierarchical path
    .MUX1("CMU_A.MUX_B"),
    …
    .DIV0("CMU_A.DIV_A"),                                    // DIV hierarchical path
    .DIV1("CMU_A.DIV_B"),
    …
    .NMUX(2), .NDIV(2)                                       // The number of MUX and DIV
) cmu_blk_a_ip_b_clk_c (
        .CLK (IP_B.CLK),
        .EN (|(CMU_A.GATE_A.EN&CMU_A.GATE_B.EN)),           // All Gate enable
        …
        .MUX_STAT ({CMU_A.MUX_A.STAT,CMU_A.MUX_B.STAT}),
        .MUX_SEL0 (CMU_A.MUX_A.SEL),
        .MUX_SEL1 (CMU_A.MUX_B.SEL),
        …
        .DIV_STAT ({CMU_A.DIV_A.STAT,CMU_A.DIV_B.STAT}),
        .DIV_RATIO0 (CMU_A.DIV_A.STAT),
        .DIV_RATIO1 (CMU_A.DIV_B.STAT),
        …
);
```
Figure 4. Example code of the ACM interface

B. CMU structure based on YAML format

The most important point in ACM binding is to know the SOC design information such as the relationship between CMU components. This requires a database format that can represent the clock tree structure efficiently and simply. In our case, YAML format was used to include various design information with high readability. It contains all the structural contents of complex clock tree, and it makes it easy for end user to understand. And it allows the verification environment such as ACM binding to be automated.

Fig. 5 shows a simple example of the relationship between CMU components and IP clock. It describes the properties of CMU components in CMU, and the IP attribute which is connected to the output of the CMU components. It indicates that the CMU_A has a MUX, DIV and GATE named MUX_B, DIV_C and GATE_D respectively. We can know that the MUX_B and DIV_C are connected and then DIV_C and GATE_D are

connected through CLKIN/CLKOUT properties. In case of IP_E, the clock source comes from GATE_D output. This means that the GATE_D and the IP_E are connected. If CLKIN source cannot be found within one CMU, it means that the source comes from the CMU of other blocks.

This information allows you to know what components are between clock source and IP clock. If there is a MUX in the middle, the total number of cases where the IP clock can be controlled will increase according to the input count of the MUX. Even in this case, the ACM should be known about all CMU components related IP clock.

```
################################################
CMU:
################################################
   - NAME: CMU_A
################################################
     COMPONENT:
        - NAME: MUX_B
          CATEGORY: MULTIPLEXER
          REGISTERS:
          ...
          CLKIN:
             - CLKIN_CLK_MUX_B_0
             - CLKIN_CLK_MUX_B_1
          CLKOUT: [MUX_B_CLKOUT]
          ...
        - NAME: DIV_C
          CATEGORY: DIVIDER
          REGISTERS:
          ...
          CLKIN: [MUX_B_CLKOUT]
          CLKOUT: [DIV_C_CLKOUT]
          ...
         - NAME: GATE_D
          CATEGORY: GATING
          REGISTERS:
          ...
          CLKIN: [DIV_C_CLKOUT]
          CLKOUT: [GATE_D_CLKOUT]
          ...
################################################
     IP:
        - NAME: IP_E
          CLOCK:
             - NAME: ACLK
               SOURCE: GATE_D_CLKOUT
               ...
          ...
################################################
   - NAME: CMU_B
   ...
   - NAME: CMU_C
   ...
```

Figure 5. Example of CMU information YAML format

C. Log Based Clock Debugging

As described earlier, The ACM provides debug information on which component and which value change the IP clock through the log. Figure 6 shows the result of simply sorting the IP clock to debug to the module name using the text editor tools. The any script is not needed to analyze the log since the log can be sorted simply through the editor tools. It can shows easily which step the clock frequency is wrong and the ultimate target frequency for IP can be created. Even if the frequency is changed by the CMU of another block, the RTL hierarchical path of CMU component is displayed as well, so it can be easily checked.

In our case, fortunately all frequency mismatches were SW setting issue not RTL bugs. However, RTL bug can be indirectly detected through the case that the target frequency cannot be created with any divide ratio or MUX selection combination, or even if a combination is created, it may affect other IP clocks.

```
UVM_INFO (CLKMON) dut.BLK_A.IP_A.CLK : FREQ changed(50Mhz->100Mhz) by
dut.BLK_A.CMU_A.MUX_A.SEL(0x00000000->0x00000001)

UVM_INFO (CLKMON) dut.BLK_A.IP_A.CLK : FREQ changed(200Mhz->300Mhz) by
dut.BLK_A.CMU_A.MUX_A.SEL(0x00000001->0x00000003)

UVM_INFO (CLKMON) dut.BLK_A.IP_A.CLK : FREQ changed(300Mhz->400Mhz) by
dut.BLK_A.CMU_A.MUX_A.SEL(0x00000003->0x00000005)

UVM_INFO (CLKMON) dut.BLK_A.IP_A.CLK : FREQ changed(400Mhz->100Mhz) by
dut.BLK_A.CMU_A.DIV_A.DIVRATIO(0x00000000->0x00000003)
```

Figure 2. Example of the log analysis for clock debugging

D. ACM Internal checker

Each unit ACM has an internal checker so that easily checks the clock for various uses as shown Fig. 7. The OneFreqCheck() task checks if the clock matches the target frequency when it called once. It is used to check if the desired value comes out after setting the clock control logic. Meanwhile, LongFreqcheck() task checks whether the clock is maintained at the target frequency from the moment of call. It is useful when verifying the clock whose value should not change. Finally, ArrFreqCheck() task can be used to check if a series of target frequency is satisfied once. For example, If TargF[] argument has three target frequencies, it checks if the clock is generated once for all three frequencies. Therefore, unlike the other two checker, when the scenario is over, it is necessary to check through GetResult() task if all target frequency are touched.

These checkers allow the ACM to be used in various SOC verification as well as dynamic IP clock verification. The users do not need to make clock checkers or interfaces for clock verification in their blocks. In case of verifying IP clock verification, only OneFreqCheck() task is used because it is a scenarios to set control values in order and check if the clock frequency comes out as desired.

```
interface clkmon_intf(
  …
  task OneFreqCheck(input int TargF, input int Margin = 2);
    …
  Endtask
  task LongFreqCheck(input int TargF, input int Margin = 2);
    …
  endtask
  task ArrFreqCheck(input int TargF[], input int Margin = 2);
    …
  endtask
  task GetResult(); // Only for ArrFreqCheck task
     …
  endtask
  …
endinterface: clkmon_intf
-----------------------------------------------------------------
class cmu_ip_dvfs_checker_vseq_c extends base_vseq_c;
  …
  virtual task body();
    …
    case (blk)

       BLK_A : begin
          if(level == 0) begin
             fork
                …
                `INTF.cmu_blk_a_ip_b_clk_c.OneFreqCheck(100)
                `INTF.cmu_blk_a_ip_b_clk_c.OneFreqCheck(200)
                `INTF.cmu_blk_a_ip_b_clk_c.OneFreqCheck(300)
                …
             join_none
          end
          else if …
          …
       end
       …
    endcase
    …
  endtask: body
```

Figure 3. Example code of ACM internal checker

E. Verification Flow with Active Clock Monitor

The IP clock frequency depending on DFS level verification flow is improved using ACM, as shown in Fig. 8. This verification requires a lot of iteration because it is inevitable to check numerous dynamic IP clock frequencies according to the DFS level and the number of IPs in SOC. The ACM is used to skip dumping the waveform for clock debugging to solve this issue.

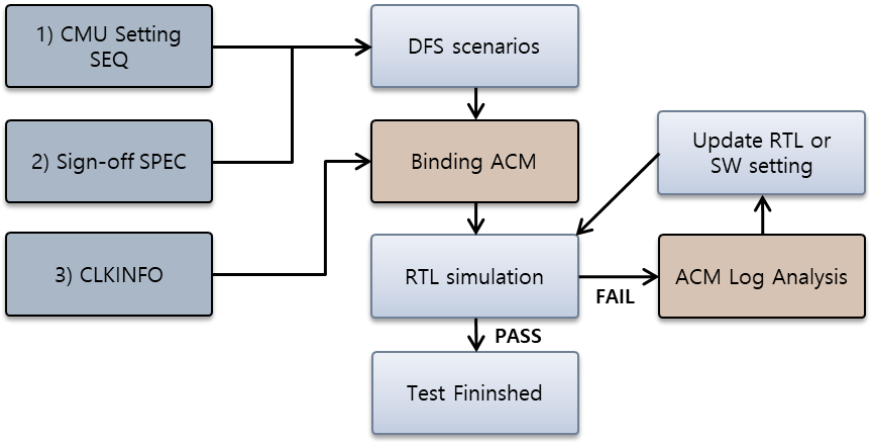

Figure 4. Improved verification flow

F. Automated Testbench Generation by Metadata Processing

Automated testbench generation is one of the efficient methods in SOC verification. The three input files are required to the automate verification environment generation for dynamic IP clocks verification. 1) Dynamic frequency sequence includes clock logic setting values and orders depending on DFS levels, 2) Clock Information (CLKINFO) includes the tree information and components (mux, divider, gating cell, etc.) of each block clock logics and 3) Sign-off SPEC includes target frequency values of each IP according to a dynamic frequency level. 1) and 2) are provided by RTL generator [3] and 3) is delivered from the design team. Using this information, the verification scenarios for dynamic IP clock and binding the ACM is auto-generated, as shown in Fig. 8.

IV. Experimental Result

The dynamic IP clock verification environment through ACM has been applied to the actual SOC design verification flow in the flagship mobile SOC project. Table 1 shows the comparison result of the simulation time and debugging points. In order to show the effect of the proposed ACM in this paper, a comparison with the old debugging method which is based on waveforms dumping is used. In comparing simulation time, the biggest difference between old debugging method and the proposed method is a waveform dumping time and it makes a big different results. The simulation time result is normalized based on the without ACM. With ACM shows a time reduction of 66% from 1 to 0.33 on full-chip simulation. As simulation repeats with frequent DUT and test bench updates, efficiency increases exponentially.

Meanwhile, the target debugging point shows the number of CMU component traces during clock debugging. Usually, 1~4 traces based on Fig. 2 is required to get the root cause in waveform debugging, and it spends a huge amount of time to debug many clock errors one by one. However, no waveform trace is needed in the proposed ACM based log debugging method because the root causes are logged whenever clock frequency is changed.

In our case, the number of target IP clock is 1748. This number will be increased according to the number of DFS levels. If it has 3 levels, the total number of clock corners will be 5244(1748 * 3). It means even if clock errors occur under 1%, 52 IP clocks should be analyzed. The log-based debugging for the clock is useful in this case.

Table 1. The effect of using clock monitor

	Simulation Time	Target Debugging Point	Debugging Time
Waveform debugging (w/o ACM)	1	1~4 (based on Fig. 2)	↑
Log debugging (w/ ACM)	0.33	1	↓ (No waveform trace)

V. CONCLUSION

In this paper, a novel debugging method which enables efficient the clock debugging for IP clocks according to DFS level through the ACM that is generated automatically using metadata. The proposed ACM is more specialized in design debugging among the purposes of the monitor. It checks the operating status of the IP clock and displays the frequency changes in the log along with the change factor such as divide ratio and MUX selection. The main goal of the proposed ACM is to prevent waveform dumping that takes a long time and to easily identify key information related to the clock to reduce debugging time. The experimental results show that debugging time can be shortened dramatically by using the ACM. If simulation should be repeated due to a difficult root-causing and complex causes, the time reduction effect will increase exponentially. In addition, this proposed approach can be used in overall SOC verification since the clock debugging is inquired in any verification.

REFERENCES

[1] Purvi D. Mulani, "SoC Level Verification using System Verilog" 2009 Second International Conference on Emerging Trends in Engineering & Technology. IEEE, 2010

[2] Siddharth Garg, et al., "Efficient methodology for verification of Dynamic Frequency Scaling of clocks in SoC", Retrieved from https://bit.ly/3jrQJ0Z

[3] Changgyu Bak, et al., "System IP Design Automation Software and its Maintenance Challenges." 2020 Design Automation Conference. DAC, 2020