**RMC75E Clock Information**

*Project: RMC75E FPGA TEST BENCH*

*Author: Satchel Hamilton*

*Company: Delta Motion*

*Date: 7/13/2023*

*Last updated: July 13, 2023*

Contents

[Notes 1](#_Toc138319027)

[Signals 10](#_Toc138319028)

------------------------------------------------------------------------------------------------------------------------------------------

## Notes

Problem Statement: The following signals remain undefined after running the clock control test bench in ModelSim for 250 microseconds:

H1\_CLK -- 60 MHz MDT clock

H1\_CLK90 -- 60 MHz clock with 90-degree lag

SysClk --30 MHz system clock

These signals are instantiated in the clock\_gen and Clock\_Gen\_Clock\_Gen\_0\_FCCC modules, under the aliases GL0, GL1, and GL2, corresponding to the aforementioned signals in the given order.

------------------------------------------------------------------------------------------------------------------------------------------

* I have determined that the clock gen issue is ultimately not stemming from any problem with the modules or the unit tests, but rather with ModelSimPro.
* Despite having installed and mapped what I believe is the correct smartfusion2 library into MS, MS still does not seem to recognize the packages it contains.
* Even after getting ModelSim to recognize the packages correctly, the compiler now complains that the smartfusion2 library is not compatible with the compiler version. Go figure.
* This also seems to be true of the PolarFire library, which is interesting since that library comes pre-installed.

**Updated Solution: The universe must have a sense of twisted humor. After banging my head against the wall for days trying to get the smartfusion2 library to play nice with ModelSim, and then finally giving up and writing a whole new clock module yesterday, I finally figured out how to correctly map the smartfusion2 library so that the real clock modules will now work just fine. Counterintuitively, it had nothing to do with the library directories in the modelsim directory, rather I had to map the smartfusion2 instance in my MS project to a different subdirectory that was sitting in Libero. I guess the main takeaway is that I got the issue resolved (twice). Case closed.**

**Clock Module Notes:**

1. **Ensure that the smartfusion2 library is correctly mapped to the following path in Libero SoC via the vmap command, NOT ModelSimPro:**

**vmap smartfusion2 C:\Microchip\Libero\_SoC\_v2023.1\Designer\lib\modelsimpro\precompiled\vlog\smartfusion**

1. **If the compiler throws an error about a mismatch in the minimum time resolution units, simulate with the following command: vsim -t 1fs tb\_clockcontrolvsim -t 1fs tb\_clockcontrol**

------------------------------------------------------------------------------------------------------------------------------------------Outdated Solution: After exploring a number of different options including using a different simulator and trying a number of different methods to get the compiler to play nice with the needed library, I settled on simply writing a drop-in module that mimics the behavior of the two modules which suffered from the ModelSim Smartfusion2 compatibility issues. While it can’t account for device specific complexities such as clock skew and metastability, it should be able to emulate the behavior of the modules it replaces well enough to continue testing and debugging in ModelSim. It appears to propagate the clock signals correctly to the other modules during the sim, although I am still testing it to ensure it behaves correctly.   
------------------------------------------------------------------------------------------------------------------------------------------

The CLK\_GEN\_NEW.vhd module generates different clock signals based on the input and internal signals.

Let's break down how each part works:

The clock generator receives three inputs: CLK1\_PAD, PLL\_ARST\_N, and PLL\_POWERDOWN\_N. CLK1\_PAD is the primary input clock signal, PLL\_ARST\_N is the system reset signal from the power monitor, and PLL\_POWERDOWN\_N is the PLL reset signal generated by a command from the CPU.

It uses these inputs to produce four outputs: GL0, GL1, GL2, and LOCK. GL0 is a 60 MHz system clock, GL1 is a 60 MHz system clock with a 90-degree phase lag, GL2 is a 30 MHz system clock, and LOCK is the PLL lock signal.

The clock\_process is the main process of the clock generator. It runs whenever there is a rising edge on the CLK1\_PAD signal. It first checks the state of PLL\_ARST\_N and PLL\_POWERDOWN\_N. If either of them is '1', it sets the internal\_clock to '0'. If they are both '0', it toggles the state of the internal\_clock. Thus, the internal\_clock frequency will be half of the CLK1\_PAD if no reset is active and will stop (remain at '0') when either of the resets are active.

There are three clock generation processes: clk\_60MHz\_gen, clk\_60MHz\_90deg\_gen, and clk\_30MHz\_gen. clk\_60MHz\_gen generates the 60 MHz clock by toggling the clk\_60MHz signal with each rising edge of the internal\_clock.

clk\_60MHz\_90deg\_gen generates a 60 MHz clock with a 90-degree phase lag with respect to clk\_60MHz. It uses an XOR gate to create this phase-shifted clock. An XOR gate outputs true only when the number of true inputs is odd. Therefore, when the internal\_clock and clk\_60MHz are different (which happens during their transition edges), clk\_60MHz\_90deg will be '1'. When they are the same, clk\_60MHz\_90deg will be '0'. This effectively shifts the clock by 90 degrees.

clk\_30MHz\_gen generates the 30 MHz clock by toggling clk\_30MHz with each rising edge of clk\_60MHz, thus producing a clock signal of half the frequency of the 60 MHz clock.

The generated clocks are then assigned to the corresponding outputs: GL0 for clk\_60MHz, GL1 for clk\_60MHz\_90deg, and GL2 for clk\_30MHz.

The LOCK signal indicates whether the clock generator is in a reset or power down state. If neither PLL\_ARST\_N nor PLL\_POWERDOWN\_N is asserted ('1'), the LOCK signal will be '1' indicating the PLL is locked and functioning correctly. If either is asserted, LOCK will be '0', indicating the PLL is either in reset or powerdown mode.

This design is an example of a simple clock generator using a PLL. In actual hardware, the timing would be defined by the characteristics of the PLL, but this model provides an idea of how different clocks can be generated and manipulated from a primary clock source.

Question: Does the clock generator work in an infinite or finite loop?

Answer: In hardware description languages like VHDL, the concept of an "infinite" loop is somewhat different from what it might mean in a high-level language like Python or Java. A better way to think of this is in terms of event-driven computation.

In the VHDL code for the new clock generator module, the process statements can be thought of as running concurrently and "forever" from a simulation perspective, but they are only active and do work when certain events occur. In this sense, it is an infinite loop. These events are defined by the sensitivity list of each process.

For example, consider the "clock\_process" process:

clock\_process : process(CLK1\_PAD)

begin

if rising\_edge(CLK1\_PAD) then

if PLL\_ARST\_N = '1' then

internal\_clock <= '0';

elseif PLL\_POWERDOWN\_N = '1' then

internal\_clock <= '0';

else

internal\_clock <= not internal\_clock;

end if;

end if;

end process clock\_process;

This process has a sensitivity list of (CLK1\_PAD), meaning it is only activated when there's an event (change in value) on CLK1\_PAD signal. Inside the process, it checks for the rising edge of CLK1\_PAD signal, then it proceeds to do some work.

Similarly, the other processes like "clk\_60MHz\_gen", "clk\_60MHz\_90deg\_gen", and "clk\_30MHz\_gen" are activated whenever there's a rising edge on their respective signals in the sensitivity list.

Now, the key point here is that VHDL is a hardware description language, not a sequential programming language. A VHDL process models hardware behavior, not a sequence of instructions. What may look like an "infinite loop" in a high-level language is really describing the continuous operation of a piece of digital hardware, which operates as long as it has power and a clock signal.

In an actual FPGA or ASIC, these processes translate to actual digital logic circuits, and these circuits respond to every clock edge for as long as the device is powered and the clock is running. The "looping" is just the inherent nature of digital synchronous circuits, which react to every clock cycle. Thus, while you might describe it as "infinite," it's really just continuously running for as long as the hardware conditions (power and clock) allow it.

If the simulation is running, then yes, it appears to be an infinite loop because it will respond to every clock event until the simulation stops. However, in actual hardware, it will run until power is removed from the circuit or the clock signal is stopped.

**NOTE: Everything after this point is notes I took while trying to solve the smartfusion2 compatibility issue, and exists purely for reference.**  
------------------------------------------------------------------------------------------------------------------------------------------  
Our test bench needs to drive the 'DLL\_RST' signal properly in order to get the 'SysClk' output. If 'DLL\_RST' signal is not driven, it can cause 'SysClk' to remain undefined or inactive.

The 'SysClk' signal is internally controlled by the 'Clock\_Gen' component in your design, but there are also control signals which can effectively disable it. One of these signals is 'DLL\_RST' which is synchronized to 'SysClk' internally.

Additionally, the 'RESET' signal should also be controlled properly during the simulation. It is possible that the system is kept in a constant reset state causing the 'SysClk' to remain undefined.

------------------------------------------------------------------------------------------------------------------------------------------

The problem seems to be related to the initialization and configuration of our clock generator (CCC component in your VHDL) and the signals it generates.

Firstly, ensure that our inputs are correctly driven in our test bench. For example, the primary clock **H1\_PRIMARY** needs to be a correctly initialized 60 MHz clock, which is in our testbench as we have specified the clock period. Other necessary signals such as the **RESET** and **DLL\_RST** signals are also properly initialized and toggled during simulation. The power-down signal **PLL\_POWERDOWN\_N** is kept high ('1') throughout the simulation, which is correct as we don't want to power down the PLL.

Secondly, in the **Clock\_Gen\_Clock\_Gen\_0\_FCCC** module, ensure that the PLL and clock generation component (CCC) is correctly initialized and configured. This involves the correct generic map parameters (**INIT** and **VCOFREQUENCY**) and the correct mappings for all the input and output ports. Incorrect configuration can result in the PLL not being able to lock onto the input clock signal or generating incorrect output clocks.

The CCC component, which seems to be from Microchip's library, is given a **VCOFREQUENCY** of **960.000** and an **INIT** parameter with a specific value. This **INIT** value configures the PLL's dividers and multipliers to achieve the desired output clocks. If the **INIT** value is incorrect, the output clocks will also be incorrect. Make sure the **INIT** value corresponds to the desired output clock frequencies (60 MHz for **H1\_CLK** and **H1\_CLK90**, and 30 MHz for **SysClk**).

Lastly, it could be possible that the issue lies in how these clocks are used in the subsequent logic in the **ClockControl** component.

After ensuring the correctness of these components and signals, run the testbench again. In ModelSim, we can also inspect the waveforms of all signals in the design to see where the problem might be occurring.

------------------------------------------------------------------------------------------------------------------------------------------

It looks like the signals are well defined in module (ClockControl). The **Clock\_Gen** module is correctly instantiated, and the output signals **GL0**, **GL1**, and **GL2** are correctly mapped to **H1\_CLK**, **H1\_CLK90**, and **SysClk** respectively.

From the VHDL code snippet, we don't see anything that would cause the H1\_CLK, H1\_CLK90, and SysClk signals to remain undefined. This suggests that the issue could be coming from elsewhere, possibly the Clock\_Gen module or the way the test bench is driving the input signals. Here are a few possibilities to check:

1. Check the test bench's setup: Make sure that the test bench properly stimulates **ClockControl** inputs. **H1\_PRIMARY** is the main clock input to the **Clock\_Gen** module, so make sure this is correctly initialized and driven in the test bench. Similarly, check **RESET** and **DLL\_RST** signals. If these are not properly handled, this could cause the output signals to be undefined.
2. Check the Clock\_Gen module: We don't see the internal implementation of the **Clock\_Gen** module here, but this is the module that generates the H1\_CLK, H1\_CLK90, and SysClk signals. Check to ensure that there are no issues in this module that might cause these signals to be undefined.
3. Review the synthesis and simulation logs: These logs may provide insights into whether there are issues during the synthesis or simulation processes that could result in undefined signals.
4. Check the logic and timing constraints: In some cases, the issue could be due to timing issues or logic that is not correctly initialized or handled.

Given that the VHDL code snippet seems to be correct, I would advise focusing on these areas to identify the source of the undefined signals.

------------------------------------------------------------------------------------------------------------------------------------------

The Clock\_Gen.vhd and Clock\_Gen\_Clock\_Gen\_0\_FCCC.vhd files are vital for understanding the clock generator functionality in the RMC75E modular motion controller. Let's analyze these modules and check if there are potential issues that could cause H1\_CLK, H1\_CLK90, and SysClk signals to remain undefined.

1. **Clock\_Gen.vhd**: This is the top-level module for clock generation, and it instantiates the Clock\_Gen\_Clock\_Gen\_0\_FCCC component. In this module, there doesn't appear to be any logic that directly manipulates the GL0 (H1\_CLK), GL1 (H1\_CLK90), and GL2 (SysClk) signals, as they're simply passed through the instantiated component. This suggests the problem might reside within the Clock\_Gen\_Clock\_Gen\_0\_FCCC component.
2. **Clock\_Gen\_Clock\_Gen\_0\_FCCC.vhd**: This module seems to be the heart of the clock generation functionality. It uses a component called CCC (Clock Conditioning Circuit, probably), which is likely responsible for generating the necessary clock signals. It appears that the INIT generic parameter of this CCC component is set with a specific value, and the VCOFREQUENCY generic parameter is set to 960.000. This configuration, if incorrect, could be causing the undefined clock signals.

Here are some steps to take to troubleshoot:

1. **Check the CCC Component**: You should verify the INIT configuration of the CCC component is correct. This configuration directly affects the frequency of the output clocks. Verify whether "00" & x"000007DC0000045174001F18C230AC270539DC40404040800101" is the correct setting for the desired output frequencies (60 MHz for H1\_CLK and H1\_CLK90, and 30 MHz for SysClk). Refer to the documentation of the CCC component for the correct settings.
2. **Inspect the Clock Signals**: Ensure the clock signals are well-behaved throughout the simulation. There are some instances in VHDL when the clock signals may not show up as expected if they are not properly handled. This is usually due to the fact that VHDL works on a scheduled event model. Ensure that the CLK1\_PAD, which is the primary input clock, is correctly handled in the VHDL code.
3. **Verify the Simulation Environment**: The simulation environment, including the VHDL simulator and any software driving the simulation, should be verified. Sometimes, the simulation environment might not fully support certain VHDL constructs or might have bugs that can lead to incorrect simulation results.
4. **Confirm Correct Entity and Architecture are Used**: In VHDL, a single entity can have multiple architectures. Ensure that the correct architecture is being used in the simulation.
5. **Check Power-Down and Reset Signals**: Ensure that the PLL\_POWERDOWN\_N and PLL\_ARST\_N signals are correctly handled throughout the simulation. If these signals are not correctly handled, the PLL in the CCC component might go into a power-down or reset state, which could result in undefined output clock signals.
6. **Inspect the Synthesis and Simulation Logs**: These logs can provide important clues about what might be going wrong. Check for any warnings or errors that might indicate a problem with the simulation.

------------------------------------------------------------------------------------------------------------------------------------------

With regards to the 90-degree phase shift, this functionality would typically be implemented within the "ClockControl" component. The testbench itself does not create or handle the 90-degree phase shift. It merely creates inputs to the "ClockControl" component and checks the outputs. The **H1\_CLK90** signal is expected to provide the clock signal with a 90-degree phase shift, and this functionality would be defined within the “Clock\_gen” module.

If the "ClockControl" and “Clock\_gen” modules work as expected, they should be able to provide the **H1\_CLK90** output as a 90-degree phase-shifted version of the **H1\_CLK** signal based on the input signals (**H1\_PRIMARY** and **H1\_CLKWR**).

We can add another test case to verify this behavior by comparing **H1\_CLK** and **H1\_CLK90**, but since these signals are updated simultaneously, direct comparison within a clock cycle may not yield the expected results. A more meaningful test would be based off of the internal implementation of "ClockControl" / “Clock\_gen”, where the particular characteristics of the phase-shifted clock (such as delay from **H1\_CLK**) could be checked.

------------------------------------------------------------------------------------------------------------------------------------------

The component **Clock\_Gen** produces various clock signals. The SysClk signal is produced by **Clock\_Gen** entity and is affected in the clock control process.

In order to resolve the issue with SysClk being undefined, we might want to check the following:

1. **Instantiation and usage of Clock\_Gen:** Check whether the instantiation and connection of **Clock\_Gen** are correct. Also, we might need to check the internals of the **Clock\_Gen** module to see if the SysClk signal is correctly generated there.
2. **Value of RESET:** If the RESET signal is high, it will set SysClk to '0'. Check if RESET is asserted unintentionally.
3. **Simulator timing:** Make sure we are running the simulation for a long enough time period to allow the system clock to be generated.

------------------------------------------------------------------------------------------------------------------------------------------

In the clockcontrol.vhd module, we see a component, **Clock\_Gen**, which is generating three different clock signals:

1. A 60 MHz clock (**GL0** or **H1\_CLK**), which is a primary clock.
2. A 60 MHz clock with a 90-degree phase lag (**GL1** or **H1\_CLK90**).
3. 90-degree phase shift == shifted by a quarter cycle.
4. A 30 MHz system clock (**GL2** or **SysClk**).

So, to my question, yes, the **ClockControl** module generates a 90-degree phase shift output signal **H1\_CLK90**, as indicated in the port mapping of the **Clock\_Gen** component.

The **Clock\_Gen** component is not defined in this code but is presumably defined elsewhere and instantiated here. Based on the usage in this file, it takes an input clock at 60 MHz (**H1\_PRIMARY**) and outputs two 60 MHz clocks: one without a phase shift (**H1\_CLK**) and one with a 90-degree phase shift (**H1\_CLK90**).

The **ClockControl** module also provides various other clock signals (**SysClk**, **Enable**, **SlowEnable**) and synchronization mechanisms for reset signals and power control. It ensures that the system reset (**SysRESET**) is activated when either the external reset signal (**RESET**) is active or when the Phase Locked Loop (PLL) is not locked (**DLL\_LOCK\_Int**). The PLL locked state (**DLL\_LOCK**) is generated after a delay mechanism to ensure the stability of the PLL lock status.

The **Enable** and **SlowEnable** signals are derived from the **SysClk** signal and are high every fourth and eighth **SysClk** cycle, respectively. They seem to be useful for slower clock domains in the system operating at 15 MHz (**Enable**) and 7.5 MHz (**SlowEnable**).

Overall, the **ClockControl** module seems to handle multiple clock domains in the system and provide synchronization mechanisms for safe system operation.

------------------------------------------------------------------------------------------------------------------------------------------

The ClockControl VHDL code shows that the 90-degree phase shift is being generated inside a component called **Clock\_Gen**. The output **GL1** (mapped to **H1\_CLK90**) is supposed to be a 90-degree phase shifted version of the input **CLK1\_PAD** (mapped to **H1\_PRIMARY**).

In order to validate the phase shift, we can compare the rising edges of **H1\_CLK** (which is the non-phase-shifted clock) and **H1\_CLK90** (which should be 90-degrees phase shifted). This test will involve a delay element in the test bench code to align the rising edges of the two clock signals.

However, directly testing the 90-degree phase shift between **H1\_CLK** and **H1\_CLK90** in a VHDL testbench is challenging, because we're dealing with idealized digital signals in VHDL simulation. In reality, a clock signal with a 90-degree phase shift would lead or lag another by one quarter of a clock period, but in VHDL simulation, clock signals are generally modeled as signals with sharp transitions with no time duration. This is a limitation of digital logic simulators which assume ideal clocking conditions.

One possible approach is to feed both **H1\_CLK** and **H1\_CLK90** to a Digital to Analog Converter (DAC) model and perform a check on the resulting analog waveforms. This method is computationally expensive and may be overkill for our needs.

Another method would be to perform a functional verification, where the logic using **H1\_CLK90** and **H1\_CLK** can be observed for correct operation, under the assumption that the **Clock\_Gen** component is producing a 90-degree phase shift.

The VHDL code provided describes the ClockControl module that is used to generate multiple clock signals from the input clock source. In the code, you can see that the Clock\_Gen module, a clock generation module is instantiated. This Clock\_Gen module appears to be the output of a Microchip's CCC (Clock Control Circuit) configurator, which is a tool provided by Microchip to generate the necessary hardware design for the generation of various clock signals.

From the dependency perspective, clockcontrol.vhd is dependent on clock\_gen.vhd, which itself is dependent on clock\_gen\_FCCC.vhd.

As far as propagation is concerned, it starts with clock\_gen\_FCCC.vhd, which provides the fundamental clock generation. The signal is then received by clock\_gen.vhd, which in turn manipulates these signals to create the specific frequencies and phases required by the system. Finally, clockcontrol.vhd is receiving these clock signals and controlling them based on the system's requirement.

The ClockControl module receives the primary clock H1\_PRIMARY and a 60MHz input clock H1\_CLKWR. From these inputs, it generates several output clock signals: H1\_CLK and H1\_CLK90 are 60MHz clocks with a 90-degree phase difference, SysClk is a 30MHz system clock, Enable is a 15MHz clock active every 4th 60MHz clock cycle, and SlowEnable is a 7.5MHz clock active every 8th 60MHz clock cycle. The Clock\_Gen module instantiated within ClockControl is responsible for generating these clock signals.

In terms of the commented-out code, it is quite common to find code that has been commented out in digital design. There could be several reasons for this. For example, the commented-out code may represent previous iterations or alternative strategies that were considered during development and were left in the code as a reference. There might also be sections of the code that are currently not in use but might be needed for future functionality or for debugging purposes. It's also possible that these sections were used for testing and were left in the code inadvertently.

The commented-out code is not needed for the operation of the current design as it stands because it is not active. However, whether it's deprecated or not depends on the development and maintenance process. It could be potentially re-activated in the future if needed. It's always a good practice to clean up the code and remove unnecessary commented out sections once they are truly deemed unnecessary, for clarity and maintainability.

------------------------------------------------------------------------------------------------------------------------------------------

## Signals

ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL0 16.667 59.999 3.321 WORST

ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL1 16.667 59.999 15.393 WORST

ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL2 33.334 29.999 10.417 WORST

WriteClk60MHz 16.667 59.999 8.770 WORST

MainClockIn60MHz 16.667 59.999 WORST

Clk\_30M 33.334 29.999 -0.898 WORST

------------------------------------------------------------------------------------------------------------------------------------------

1. Clock Name: CCC\_INST/CLK1\_PAD
   * Frequency: 60 MHz
   * Period: 16.6667 ns
2. Clock Name: CCC\_INST/GL0
   * Source: PLL (0)
   * Frequency: 60 MHz
   * Period: 16.6667 ns
   * Source Frequency: 60 MHz
   * Actual Output Frequency: 60 MHz
3. Clock Name: CCC\_INST/GL1
   * Source: PLL (90)
   * Frequency: 60 MHz
   * Period: 16.6667 ns
   * Source Frequency: 60 MHz
   * Actual Output Frequency: 60 MHz
4. Clock Name: CCC\_INST/GL2
   * Source: PLL (0)
   * Frequency: 30 MHz
   * Period: 33.3333 ns
   * Source Frequency: 60 MHz
   * Actual Output Frequency: 30 MHz
5. Clock Name: H1\_CLK\_IN
   * Frequency: 60 MHz
   * Period: 16.6667 ns
6. Clock Name: ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL0
   * Frequency: 60 MHz
   * Period: 16.6667 ns
7. Clock Name: ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL1
   * Frequency: 60 MHz
   * Period: 16.6667 ns
8. Clock Name: ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL2
   * Frequency: 30 MHz
   * Period: 33.3333 ns
9. Clock Name: Clock\_Gen\_Clock\_Gen\_0\_FCCC|GL2\_net\_inferred\_clock
   * Frequency: 100 MHz
   * Period: 10.0 ns
10. Clock Name: Top|H1\_CLKWR

* Frequency: 100 MHz
* Period: 10.0 ns

1. Clock Name: Clock\_Gen\_Clock\_Gen\_0\_FCCC|GL0\_net\_inferred\_clock

* Frequency: 100 MHz
* Period: 10.0 ns

1. Clock Name: Clock\_Gen\_Clock\_Gen\_0\_FCCC|GL1\_net\_inferred\_clock

* Frequency: 100 MHz
* Period: 10.0 ns

1. Clock Name: H1\_CLKWR

* Frequency: 60 MHz
* Period: 16.6667 ns

1. Clock Name: H1\_CLK\_IN

* Frequency: 60 MHz
* Period: 16.6667 ns

1. Clock Name: H1\_CLK

* Frequency: 60 MHz
* Period: 16.6667 ns

1. Clock Name: SysClk

* Frequency: 30 MHz
* Period: 33.3333 ns

1. Clock Name: H1\_CLK90

* Frequency: 60 MHz
* Period: 16.6667 ns

------------------------------------------------------------------------------------------------------------------------------------------

Condensed:

1. CCC\_INST/CLK1\_PAD: Frequency 60 MHz (Period 16.6667 ns)
2. CCC\_INST/GL0: Frequency 60 MHz (Period 16.6667 ns)
3. CCC\_INST/GL1: Frequency 60 MHz (Period 16.6667 ns)
4. CCC\_INST/GL2: Frequency 30 MHz (Period 33.3333 ns)
5. H1\_CLK\_IN: Frequency 60 MHz (Period 16.6667 ns)
6. ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL0: Frequency 60 MHz (Period 16.6667 ns)
7. ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL1: Frequency 60 MHz (Period 16.6667 ns)
8. ClkCtrl\_1/clk\_1/Clock\_Gen\_0/GL2: Frequency 30 MHz (Period 33.3333 ns)
9. Clock\_Gen\_Clock\_Gen\_0\_FCCC|GL2\_net\_inferred\_clock: Frequency 100 MHz (Period 10.0 ns)
10. Top|H1\_CLKWR: Frequency 100 MHz (Period 10.0 ns)
11. Clock\_Gen\_Clock\_Gen\_0\_FCCC|GL0\_net\_inferred\_clock: Frequency 100 MHz (Period 10.0 ns)
12. Clock\_Gen\_Clock\_Gen\_0\_FCCC|GL1\_net\_inferred\_clock: Frequency 100 MHz (Period 10.0 ns)
13. H1\_CLKWR: Frequency 60 MHz (Period 16.6667 ns)
14. H1\_CLK\_IN: Frequency 60 MHz (Period 16.6667 ns)
15. H1\_CLK: Frequency 60 MHz (Period 16.6667 ns)
16. SysClk: Frequency 30 MHz (Period 33.3333 ns)
17. H1\_CLK90: Frequency 60 MHz (Period 16.6667 ns)