Vivante GCNanoUltraV Series

Vector Graphics IP

Hardware Implementation Manual

Revision 0.82

04 February 2022

This document is compatible with Vivante

GCNanoUltraV Series hardware release versions 2.0.x

Legal Notices

Preface

This document is the primary implementation reference manual for Vivante GCNanoUltraV Series vector graphics processing IP.

Audience

This document was prepared for hardware implementation engineers who are familiar with backend design flow methodology. Those who would benefit from this technical manual are:

• Engineers and managers who are implementing this IP for use in a system.

Conventions Used in This Manual

AHB – Advance High Performance Bus
APB – Advanced Peripheral Bus
ATPG – Automatic Test Pattern Generation
AXI – Advanced eXtensible Interface
BIST – Built In Self Test
DFT – Design for Test
DMA – Dynamic Memory Access
DRC – Design Rule Coverage
DV – Design Verification
ECO – Engineering Change Order
FE – Graphics Pipeline Front End/Fetch Engine
FPGA – Field Programmable Gate Array
GPU – Graphics Processing Unit
GUI – Graphical User Interface
HI – Host Interface
ICG – Integrated Clock Gating
IM – Imaging Engine
IR-drop – voltage drop
LEC – Logic Equivalence Checker
LVS – Layout Versus Schematic
MC – Memory Controller
P&R – Place and Route
PD– Physical Design
PE – Pixel Engine
PM – Power Management
RC – Resistance and Capacitance
RDL – Redistribution Layer
RTL – Resistor Transistor Logic
SDF – Synchronous Data Flow
SI – Signal Integrity
SoC – System on Chip
SPEF – Standard Parasitic Exchange Format
STA – Static Timing Analysis
TS – Tessellation Engine
VG – Vector Graphics

The word *assert* means to drive a signal true or active. Signals that are active LOW end in an "n."

Hexadecimal numbers are indicated by the prefix "0x" —for example, 0x32CF.

Binary numbers are indicated by the prefix "0b" —for example, 0b0011.0010.1100.1111

Code snippets are given in `Consolas` typeset.

Table of Contents

List of Figures

List of Tables

1 Introduction

This is the primary implementation reference for Vivante GCNanoUltraV Series Vector Graphics Processing Unit IP, with detailed descriptions of the key considerations for engineers who are implementing a version of this Vivante Vector Graphics core. Here we assume that you have some familiarity with an ASIC implementation flow.

The GCNanoUltraV Series Vector Graphics IP has been designed for easy SoC integration, providing high performance, high quality graphics, low power consumption, and a small silicon footprint for its class. The core is delivered as synthesizable RTL. It is technology independent and can be synthesized using a variety of libraries. Dynamic power consumption is minimized by the extensive use of multi-level hierarchical clock gating.

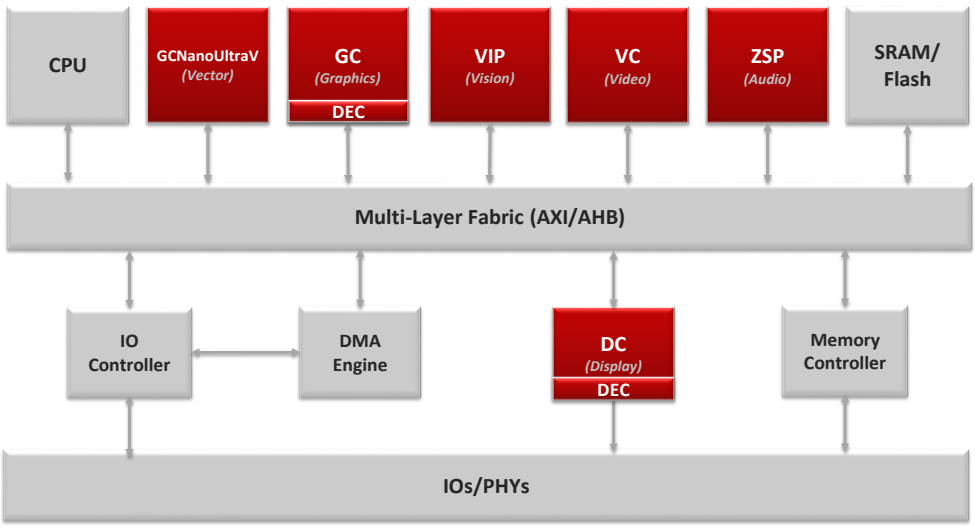

Figure 1. Typical SoC with Vivante GCNanoUltraV and other Vivante IP

1.1 GCNanoUltraV Variants

The GCNanoUltraV Series Vector GPU IP includes a 32-bit AHB interface for register access. For external memory access, variants include either 1x64-bit AXI or two 32-bit AHB. APB is an alternate to AHB.

For available variants for this custom design, refer to the Hardware Features document for this IP.

1.2 GPU Core Design Hierarchy

This GPU core has a simple design hierarchy. The top level contains two modules: a Power Management module and a Vector Graphics module.

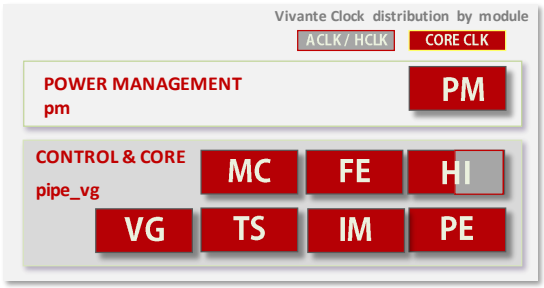

Figure 2. GCNanoUltraV Design Hierarchy

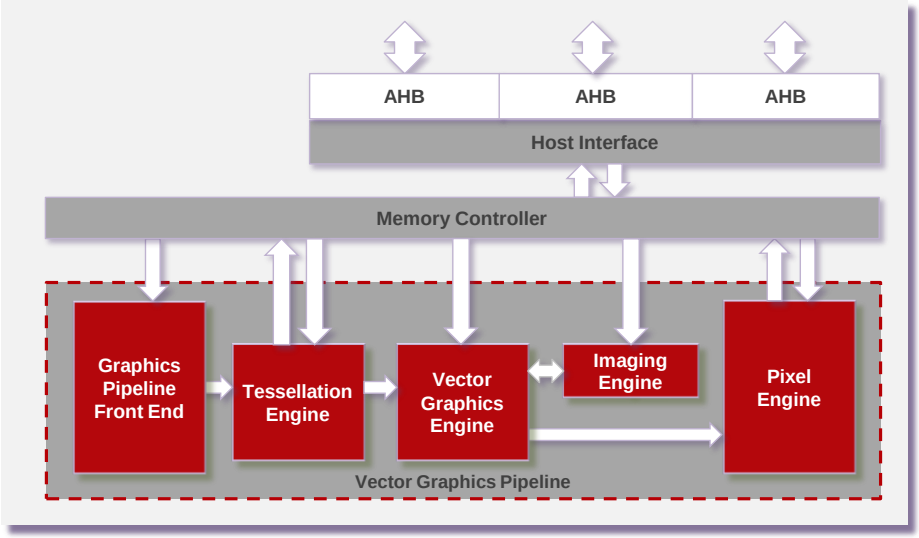

Figure 3. GCNanoUltraV Core Block Diagram with 3xAHB
(variants include 1xAXI=1xAHB)

1.3 GCCORE Design Description

The main functional blocks of the GCCORE are described here, and a block diagram is shown above.

Host Interface	Allows the GCCORE to communicate with external memory and the CPU through the AHB bus. In this block data crosses clock domain boundaries.
Memory Controller	Internal memory management unit that controls the block-to-host memory request interface.
Graphics Pipeline Front End	Inserts high level primitives and commands into the graphics pipeline.
Tessellation Engine	Transforms vertices and control points. Tessellates lines, quadratic and cubic Bezier curves.
Vector Graphics Engine	Rasterizer that converts primitives to pixels.
Imaging Engine	Paint and image generator that colors each pixel.
Pixel Engine	Renderer that combines different sources into the final pixel value.

1.4 Clock Domains and Structure

GCNanoLiteV employs a simple clocking scheme. There are three independent clock domains in the GPU IP:

- Core clock domain, which is derived from the clk1x pin,
- AHB interface clock domain for register access, which is derived from the HCLK pin, and
- AHB interface clock domain for memory access, which is derived from the ACLK pin.

ACLK is the AHB interface clock domain for memory access and *HCLK* is the AHB interface clock for register access. clk1x is the functional clock for the core logic.

Communication between the different domains in the GPU occurs mostly by way of asynchronous FIFOs. All the clocks are asynchronous to each other. There is no communication between HCLK and ACLK clock domains.

The core clock in the GPU core can be scaled down dynamically without reprogramming its source. This scaling is controlled through the use of an internal register.

Table 1. Clock Definitions

GPU Clock Pins	Derived Internal Clocks	Clock Disable Signal	Description
HCLK	-	-	AHB interface clock for register access; provided external to the IP.
ACLK	-	-	AHB interface clock for memory access; provided external to the IP.
clk1x	clk_2d	clk_2d_dis	clk1x is the main core clock; provided external to the IP.

GPU_CORE Clock information:

clk1x the main core clock; provided external to the IP
clk_2d is generated from clk1x

clk2d_dis Disables clk_2d

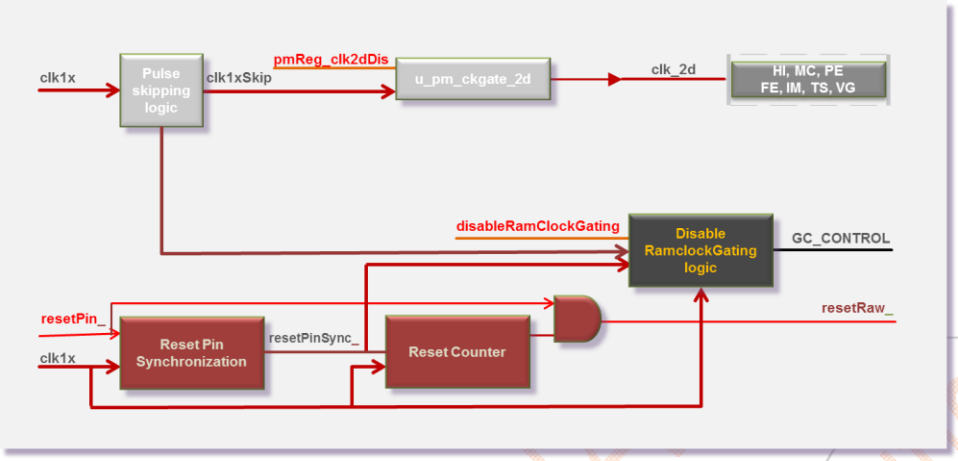

Figure 4. GCNanoUltraV Clock Logic Block Diagram for clk1x

1.5 Clock Gating Cells

There are some hard-coded ICG (Integrated Clock Gating) cells in the RTL code which are technology specific. The customer must edit this file to make sure the correct ICG cells for their target library are used. Generally, the SoC integrators or the front-end engineers should map these ICG cells into the specific target libraries. For more information on the implementation detail for Clock Gating and Clock Tree structure, see the Physical Implementation section on Clock Tree Synthesis. For more information on integrating the Clock Gating Components, refer to the Vivante Hardware Integration Guide.

1.6 Area and Power

Small Silicon Area

- Please refer to Product Brief for synthesis and P&R area numbers.
- "Push button" reference flow for Cadence and Synopsys
- Test insertion ready

Low Power Design

- Please refer to Product Brief for power numbers
- Dynamic power management extends battery life
- Software controllable power states
- Automatic clock gating of flip flops and RAMs

2 Physical Design Package

The physical design (PD) package consists of a README file that describes the PD directory structure, a documentation directory and process specific directories applicable to this core.Preliminary packages may not contain all directories.

2.1 Documentation Directory

The physical design documentation files in the **doc** directory have self-descriptive names:

Table 2.PD doc Directory Contents

doc Filename	Description
accurate_memory_instance_list	Behavioral memory specification

2.2 sec14lpp Directory

This directory contains constraint files and subdirectories for Cadence and Synopsys design flow in SEC 14nm LPP process technology. This directory will be referenced as the example in this document.

Table 3. PD sec14lpp Directory Contents

sec14lpp Filenames	Description
DEFINES	Customer-specific defines when reading RTL constraints
constraints subdirectory files	
README	Readme file for constraints. Refer to this file for descriptions if additional constraint files are included in your package.
VIVANTE_GPU.syn.sdc	Timing constraint file for synthesis
scripts subdirectories	Each includes scripts and a README file
dc	Directory of scripts for Synopsys synthesis
icc	Directory of scripts for Synopsys place-and-route
lec	Directory of scripts for Cadence formal verification
rc	Directory of scripts for Cadence synthesis

3 RAM Macros and Timing

3.1 Memory Macros

1-port and 2-port register files are used throughout the IP as temporary storage, FIFOs and caches. Some of the memory blocks require byte write capability. The IP is developed using behavioral memory models. Depending on the available memory configurations, synthesizable memory blocks are added by specific ifdef Verilog macros. Detailed memory information can be found in the "***…/doc/accurate_memory_instance_list***" file of the Physical Design Package. This file represents the design-intended memories used in the RTL, which is a technology independent memory list. Generally, the SoC integrators or the front-end engineers should map the memories into the specific target libraries on a given technology.

In Vivante designs two port memory typically provides separate READ and WRITE ports. The following generic naming conventions are frequently used for the memory pins:

Table 4. Naming Conventions for 1- and 2-port Memory Pins

*** Pins common for both 1 and 2 port memories:**	
D (data-in for write)	
Q (data-out for read)	
WEB (write enable)	
BWEB (write mask enable, low active; applicable if memory is mask enabled)	
*** 1-port (RF1P*) memory has these additional pins:**	*** 2-port (RF2P*) memory has these additional pins:**
A (address for either read or write)	AA (address for write) AB (address for read)
CEB (chip enable, low active)	REB (read enable, low active)
CLK (clock for read or write)	CLKR (clock for read) CLKW (clock for write)
WEB (write enable when low, else read enable)	WEB (write enable, low active)

3.2 RAM Timing Requirements

The timing diagram below illustrates the RAM timing requirements for inputs and outputs and also illustrates the RAM output characteristic.

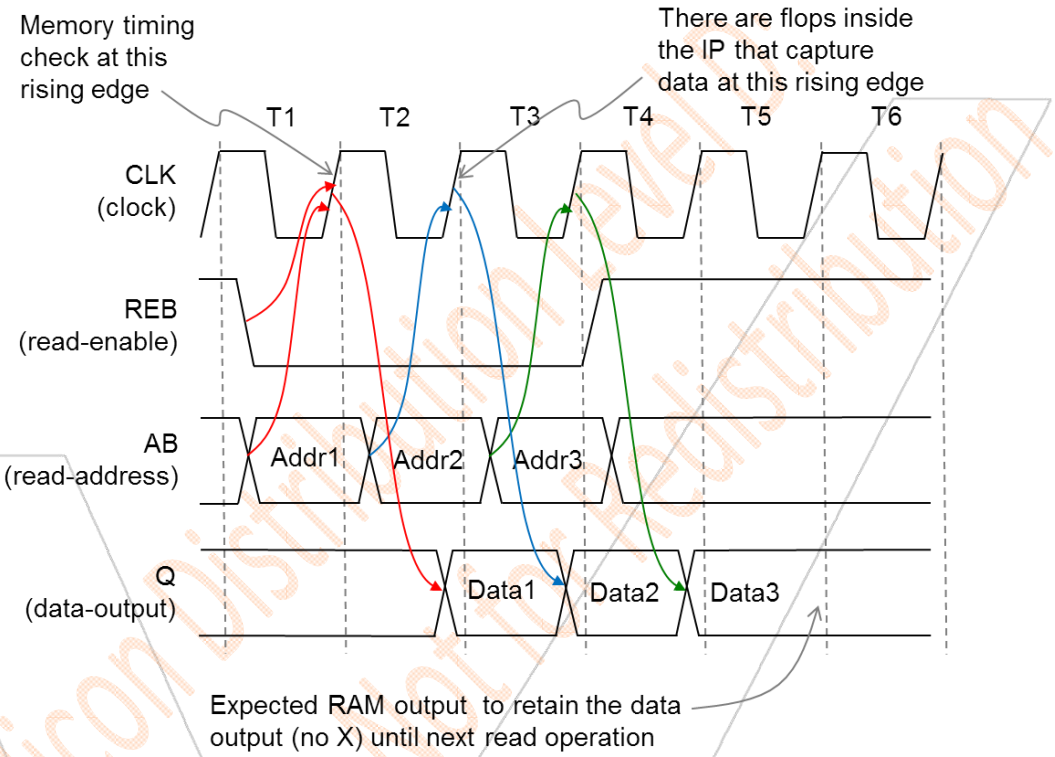

Figure 5. RAM Timing Requirements

4 Synthesis

This section describes the synthesis methodology used for the GPU IP as well as the synthesis scripts provided with the Physical Design Package. These scripts are for using with the Synopsys Design Compiler or with the Cadence RTL Compiler. The synthesis methodology presented here has been developed to help the integrator achieve both high operating frequency and a small silicon area. After timing is met, the synthesis focus is on low power optimization. The methodology outlined in this section has been proven to produce the best synthesis result across a number of process technologies and the standard cell libraries.

4.1 Synthesis Methodology

The GPU is synthesized using a well-qualified flow which has been proven to yield the best results in terms of speed and area. Modern synthesis tools in general can process designs of this size and larger very efficiently.

The most critical paths are in the data path units, which exist in many sub-blocks. The synthesis tool should be focused on producing the best logic structure that meets the timing requirements. The following guidelines can help in achieving optimal results:

- **Use Zero Wire-Load** for synthesis.This has been proven by Synopsys Design Compiler and Cadence RTL Compiler to be the most effective way to reach the optimal timing results while keeping the area as small as possible.
- **Use multiple VT cells** in order to achieve the target speed in a given process. Using the high VT and standard VT cells should be sufficient to achieve the target speed.
- **Datapath Optimization** is critical to achieving the target speed in synthesis. DC Ultra is preferred when using the Synopsys Design Compiler, as it will automatically enable datapath optimization.
- **Clock Gating cells**. There are some hard-coded clock gating cells in the RTL. For DFT, please make sure the tool is configured properly in order to connect to those gating modules correctly.

4.2 Synthesis Preparation

In order to successfully run synthesis, you may need to consider the following key points.

4.2.1 Clock Gating Cell Setup

There are some hard-coded ICG (Integrated Clock Gating) cells in the RTL code which are technology specific. SoC integrators or front-end engineers must choose an integrated clock gating cell to match their target library.If this step is not done at the RTL simulation stage, then the PD engineers will need to do the clock gating cell setup to ensure a successful synthesis run.

Edit the file "***rtl/gcnanoultrav/v2d_GC_CG_MOD.v***" to choose an integrated clock gating cell from the target library for synthesis. An example of the clock gating cell from the SEC14LPP standard cell library is provided for your reference. Shown below is the code section from "***v2d_GC_CG_MOD.v***" where editing needs to be done.

```
`ifdef AQ_SEC14LPP_A9TR_C14_LIB
  PREICG_X4N_A9PP84TR_C14 GC_CG_INST(
.E (enable),
.CP (ck_in),
.TE (test),
.Q (ck_out));
`else
```

The correct clock gating cell from the target library should be instantiated to ensure a successful synthesis run. The synthesis tool should not synthesize this logic from the ***`else*** section which contains the default.

The correct clock gating cell for the module can be found by looking for the Synopsys dotlib of the target library for an integrated clock gating cell with one of the following attributes:

```
latch_posedge_postcontrol_obs
latch_posedge_precontrol
latch_posedge_postcontrol
```

An example from the SEC 14nm A9TR library is shown:

```
cell(PREICG_X1N_A9PP84TR_C14) {
area : 0.72576 ;
cell_footprint : PREICG_X1N ;
clock_gating_integrated_cell : "latch_posedge_precontrol" ;
…
```

4.2.2 Memory Macro Setup

There are technology-specific memory macros in the RTL code which are used throughout the IP as temporary storage, FIFOs and caches. The customer must review and possibly modify these macros to make sure the correct memories match those specified in the target library. Generally, the SoC integrators or the front-end engineers should map the memories into the specific target libraries. For PD engineers, please make sure to use the correct define to call the desired memory macros.

In the "***v2d_RAMxxx.v***" files of the "***rtl/rams***" directory, there are accurate memory models. You can generate the corresponding memory cells based on those memory specifications. An example of memory macros from the SEC 14nm library is provided solely for your reference, as it might not be accurate for your design. Shown below is the code section from "***rtl/gcnanoultrav/v2d_gc_*_wrapper.v***"where the editing needs to be done.

```
    `ifdef AQ_SEC14LPP_RAM_MODEL
    wire enableReadClock  = ~(rEn0_) | GC_CONTROL[4];
    `ifdef CG_IN_RAM
    wire rclk0En = rclk0;
    `else
    wire rclk0En;
    gc8k_GC_CG_MOD GC_CG_READ(.enable(enableReadClock), .ck_in(rclk0),
       .ck_out(rclk0En), .test(GC_CONTROL[0]));
    `endif
    wire enableWriteClock0 = ~(wEn0_) | GC_CONTROL[4];
    `ifdef CG_IN_RAM
    wire wclk0En0 = wclk0;
    `else
    wire wclk0En0;
    gc8k_GC_CG_MOD GC_CG_WRITE0(.enable(enableWriteClock0),
       .ck_in(wclk0), .ck_out(wclk0En0), .test(GC_CONTROL[0]));
    `endif
    wire [16-1:0] ramRData0;
reg  [16-1:0] ramWData0;
    always @(wData0[14:0]) begin
        ramWData0 = 16'd0;
        ramWData0[15-1:0] = wData0[14:0];
    end
    assign rData0[14:0] = ramRData0[15-1:0];
    RF2P256W16B2M2BK0S0HD0WM ram0 (
       .QA({ramRData0}),
.CLKA(rclk0En),
.CENA(rEn0_),
.AA(rAddr0[8-1:0]),
.CLKB(wclk0En0),
.CENB(wEn0_),
.AB(wAddr0[8-1:0]),
       .DB(ramWData0),
.STOV(1'b0),
.EMAA(3'b010),
.EMASA(1'b0),
.EMAB(3'b010),
.RET1N(1'b1));
    `else
```

4.2.3 Design Defines

The correct "Design Defines"must be set to read the RTL Verilog files when running synthesis if your design has defines to control the RTL. For example, in our internal PD qualification we will use the SEC14LPP target libraries. Use the ***AQ_SEC14LPP_A9TR_C14_LIB*** define in ***rtl/gcnanoultrav/v2d_GC_CG_MOD.v*** to set the clock gating cells. Use the ***AQ_SEC14LPP_RAM_MODEL*** define in "***rtl/gcnanoultrav/v2d_gc_*_wrapper.v"*** to set the memory macros. Thus, the instantiated ICG cells and the memories are both mapped into the SEC 14nm LPP process.

If you are using a different process technology, then you must edit the files to specify the clock gating cells and memory macros that match your design. For details, please refer to the "Clock Gating Cell Setup" and the "Memory Macro Setup" sections, above. See the table below for a listing of key defines and their description.

Table 5. Synthesis Design Defines

Define (Verilog `define)	Description
AQ_SEC14LPP_A9TR_C14_LIB	Use SEC 14nm technology specific ICG cells. Reference files: Synopsys dotlib (Refer to the "Clock Gating Setup" section.) Modify in file: ***rtl/gcnanoultrav/v2d_GC_CG_MOD.v***
AQ_SEC14LPP_RAM_MODEL	Use the SEC 14nm technology-specific memory library. Reference files: ***rtl/rams/v2d_RAMxxx.v*** (Refer to the "Memory Macro Setup" section.) Modify in file: ***rtl/gcnanoultrav/v2d_gc_*_ram_wrapper.v***
Define Cancellation (Verilog `undef)	
`undef <DEFINE> as specified in **rtl/undef/<prefix>_vsi_undef.v**	Cancels previously defined text macro definitions. NOTE: Please compile ***rtl/undef/<prefix>_vsi_undef.v*** as the last Verilog file for this RTL.

4.2.4 Don't-Use Cells

Not all standard cells are suitable for synthesis. The Synopsys attribute "*dont_use*" may be set on some certain low-drive cells, clock buffers, clock inverters, delay cells, etc. to prevent the tool from using them during synthesis. This attribute is removed in the Place-and-Route step. A designer performing synthesis is responsible for creating this cell list based on recommendations from the ASIC standard cell vendor. The *dont_use* cell list should be added to the library setup file when it is first created. Refer to the ***sec14lpp/scripts/dc/setup/sec14lpp.dont_use.tcl*** file in the Physical Design Package for an example of cell types that have been added to the "*dont_use*" list.

4.2.5 Max Transition Parameter

For most technology libraries, the use of the command "*set_max_transition*" is not recommended during the synthesis stage. Since the synthesis tool does not have the cell placement information, the interconnect delays are only estimated. Hence, the transition time is not accurate. It is best to let this command default to the maximum signal transition value that is set in the standard cell library. The "*set_max_fanout*" command is a better choice for controlling the fanout and signal transition during the synthesis stage.

4.2.6 Synthesis Constraints

Synthesis constraints define the design goals that guide the synthesis process. They consist of the input and output delay, load, input clock characteristics, and timing exceptions. Refer to the section on Static Timing for a detailed description.

4.3 Running Synthesis

Fully functional scripts to run synthesis for the Vivante GCCORE are provided in the Physical Design Package at:

…/sec14lpp/scripts/dc
…/sec14lpp/scripts/rc

Readme files and comments within the scripts themselves provide additional information. Please modify the file "**main.tcl**" as needed for your project.

Table 6. Synthesis Scripts Description

Directory and Script Name	Description
dc/main.tcl	Recommended script for synthesis with Synopsys Design Compile (automatically enables datapath optimization)
dc/setup/	Directory containing the project-related setup file
dc/setup/common.tcl	Design independent script for synthesis with Synopsys Design Compiler
dc/setup/dc.tcl	Recommended setup script for synthesis with Synopsys Design Compiler
dc/setup/default.tcl	Sets the significant digits default for reports for synthesis with Synopsys Design Compiler
dc/setup/flow.setup	Example setup flow
dc/setup/lib_rvt_c14_ssa_sigcmax_max0p72v_m40c.tcl	Recommended script for library setup
dc/setup/sec14lpp.dont_use.tcl	Recommended dont_use cells
rc/rc.tcl	Recommended script for synthesis with the Cadence RTL Compiler

5 Scan Insertion

This section describes some scan insertion related items to consider. Vivante GCCOREs have a relatively simple scan interface. You can implement your own DFT design based on your comprehensive test strategy and methodology.

Since different customers select different DFT methodologies when integrating their Vivante GCCORE into their SoC (for example, number of scan chains, compression or no compression, etc.), only scan flop swapping is performed in the reference flow, while the actual scan chain is not constructed. (Vivante does perform a full scan on all cores as a routine part of Vivante's internal physical design qualification flow.)

BIST insertion is not included in the reference flow because the BIST methodology used by each customer can be very different. This needs to be considered in the context of the entire SoC DFT/BIST methodology. Also, ATPG is not included in the reference flow.

5.1 Scan Mode

We have an input port at the top level, called "**scanMode**." This signal enables the hard-coded ICG cells. These cells are technology-specific and should be manually inserted.

5.2 Scan Clock

This GPU employs a simple clocking scheme, with the following clocks that enter the core: ***ACLK***, ***HCLK,*** and ***clk1x.*** The clocks are all asynchronous to each other. ***ACLK***, is the AHB clock for memory access, ***HCLK*** is the AHB interface clock for register access, and ***clk1x*** is the functional clock for the core logic.

For the main functional logic, you can define ***clk1x*** to a scan clock. For the AHB interface to memory, you can define ***ACLK*** to a scan clock. For the AHB interface for register access, you can define ***HCLK*** to a scan clock. For the at-speed DFT test, you can define the DFT region based on your own requirements.

5.3 Clock Gating Cell Test Enable

The "***.test***" pin of the Clock Gating Cell is connected to the "***scanMode***" signal in our RTL code. You can edit this connection to match your own requirements. Here is an example of the Vivante defaults.

```
GC_CG_MOD GC_CG_CKGATE (
.enable(),
 .ck_in(),
 .ck_out(),
 .test(scanMode)
);
```

To achieve higher coverage for the clock gating cells, ***scanEnable*** may be a better control signal. Our ***.test*** signal connection is just for your reference. Please refer to your DFT documents to improve testability in clock gating.

6 Formal Verification

This section discusses the use of Cadence Encounter Conformal Equivalence Checker to complete the formal verification between the Vivante GCCORE RTL (golden model) and the synthesized netlist (revised model). Formal verification is an important step to ensure any resulting synthesized netlist is functionally equivalent to the RTL. We do not include the Synopsys Formality tool in the reference flow. Please contact Synopsys Technical Support if you need more information or help in running Formality.

To facilitate the ease of running formal verification the formal equivalence check is done in two steps.

Step 1 **RTL vs. hierarchical netlist**. During this step, the hierarchy of the modules should be preserved and the modules should be restricted from boundary optimization. The hierarchical netlist produced during this Step is used for formal verification only. Step 1 will help the formal tool complete faster.

Step 2 **Hierarchical netlist vs. final netlist**. Once steps 1 and 2 pass, the RTL is verified against the final net list.

6.1 Running the Cadence Encounter Conformal Equivalence Checker

Scripts for running a Cadence Encounter Conformal LEC (Logic Equivalence Checker) for the Vivante GCCORE are located in the following directory in the Physical Design Package.

…/sec14lpp/scripts/lec

Table 7.Conformal LEC Scripts Description

lec Filename	Description
dc.abort.list	List of module names which can cause LEC to abort. So that the LEC will be able to compare hierarchically, do not modify the boundary for this synthesis.
rtl_to_dct.do	LEC do-file to compare RTL to DC netlist.
rc.abort.list	List of module names which can cause LEC to abort. So that the LEC will be able to compare hierarchically, do not modify the boundary for this synthesis.
rtl_to_rc.do	LEC do-file to compare RTL to RC netlist.
modify_do.pl	Perl script to make the LEC compare the difficult modules listed in ***dc.abort.list/rc.abort.list*** in a more datapath-friendly fashion.

Note:You must set the correct locations for the do-files and the design in the Conformal LEC run script.

We suggest preserving boundaries when running synthesis, because it will be easier to do hierarchical comparison.

7 Static Timing

This section discusses static timing analysis. For help with the Synopsys PrimeTime tool, please contact Synopsys Technical Support. For help with the Cadence Encounter Timing System, please contact Cadence Technical Support.

Timing constraint details are provided in the Physical Design Package at **…/sec14lpp/constraints/.**

Filename	Timing Constraint File Description
README	Readme file for constraints. Refer to this file for descriptions if additional constraint files are included in your package.
VIVANTE_GPU.syn.sdc	Timing constraint file for synthesis

7.1 Clock Definitions

The timing constraint files contain definitions for clock characteristics such as periods, waveform, IO delays, etc. for each clock.

7.2 IO Timing Constraints

These constraints define the arrival and departure time for the IO signals. The majority of the IO signals belong to either the ***ACLK*** domain or the ***HCLK*** domain. In order to use the same timing constraints throughout the implementation flow, virtual clocks are defined for each clock that has IO signals associated with it, such as: ***ACLK***, ***HCLK***, and ***clk1x***. These virtual clocks are ***ACLK_io***, ***HCLK_io***, ***clk1x_io***. This removes any need to re-adjust the arrival or departure time in the timing constraint file pre- or post-clock tree insertion.

You may need to adjust the input and output delay values in the constraint files based on your own design system. And you may need to change the input drive and output load cells/values based on your own foundry libraries.

7.3 Timing Exceptions

Some sets of timing exceptions are defined in the constraint files provided in the Physical Design Package at **…/sec14lpp/constraints/**.

7.3.1 Disable the paths between the asynchronous clock groups

Each source clock and its derivative clock form a clock domain group. The clock domain groups are asynchronous to each other. For example:

```
set_clock_groups -asynchronous -name gpu_async_group \
                 -group { ACLK ACLK_io} \
                 -group { HCLK HCLK_io} \
                 -group { clk1x clk1x_io }
```

7.3.2 Choose the timing mode by setting the case analysis

For example:

```
set_case_analysis 0 [get_ports {scanMode}]
```

Note: One GPU version may have slightly different timing constraints from another. For timing details, please refer to the specific "***…/sec14lpp/constraints/*.sdc***" file provided in the Physical Design package.

8 Physical Implementation

The following is an overview of the recommended methodology for the physical implementation. Sample scripts for Synopsys ICC are available in the Physical Design Package at:

…/sec14lpp/scripts/icc

This section is not intended to replace the design documentation provided by the specific ASIC vendors, manufacturers, and library providers. Rather, it is intended to present the generic guidelines for the implementation of the GPU core.

8.1 Hierarchy and Floor Plan

8.1.1 GPU Core Hierarchy

The GPU hierarchical structure has the following components.

- VIVANTE_GPU
  - pm
  - pipe_vg
    - fe
    - hi
    - im
    - mc
    - pe
    - ts
    - vg

8.1.2 Floor Plan Diagram

Check with the Vivante Physical Implementation support team for additional advice and/or documentation for floorplan memory placement and optimization.

8.2 Power Mesh

The power grid scheme should be decided by the SoC vendor based on the requirements of the entire chip, considering bumps, RDL routing, power and IR requirements.

8.3 Cell Placement

In order to achieve optimum timing and routing, a concurrent placement/timing optimization tool, such as IC Compiler (ICC) from Synopsys, or Encounter-Digital Implementation System (EDI) from Cadence, should be used to generate the standard cell placement.

These tools require a netlist-bound floorplan as input. Timing constraints, such as clock definitions, I/O delays, loads, and input drivers will need to be specified as well. Scan re-ordering should also be performed at this step to minimize the wire length on the full scan network.

8.4 Clock Tree Synthesis

Our GPU core has a relatively simple clock scheme. Use of a high layer of metal for clock routing, especially those for the clock tree buffer nets, is recommended. Extra spacing for the clock routing is also preferred.

Both Synopsys ICC and Cadence Encounter can route the clock tree as part of the clock tree insertion process. This is the preferred method, as it usually can yield a clock network with the smallest clock skew.

Clock latency is very important to meet the timing for the enable signals. As long as the latency is close to the period of ***clk1x***, there should be no issues.

Localized clock gating is used throughout the design in order to minimize dynamic power consumption. Almost 100% of the registers are gated after synthesis.

We have multi-level clock gating ICG cells inserted in our RTL design. In high performance implementations, there may be violations from the clock gating enable flops to those module level clock gates. It is desirable to shorten these enable flops' clock latencies. The solution can be using the useful skew method or using the skip/exclude pin to control it during clock tree synthesis. You can also make a tree from the root of ***clk1xSkip***.

8.4.1 Clock Gating and Clock Tree Implementation

The key components of the Vivante Clock structure include clock gating elements which allow for independent control for the GPU level, and sub-modules.

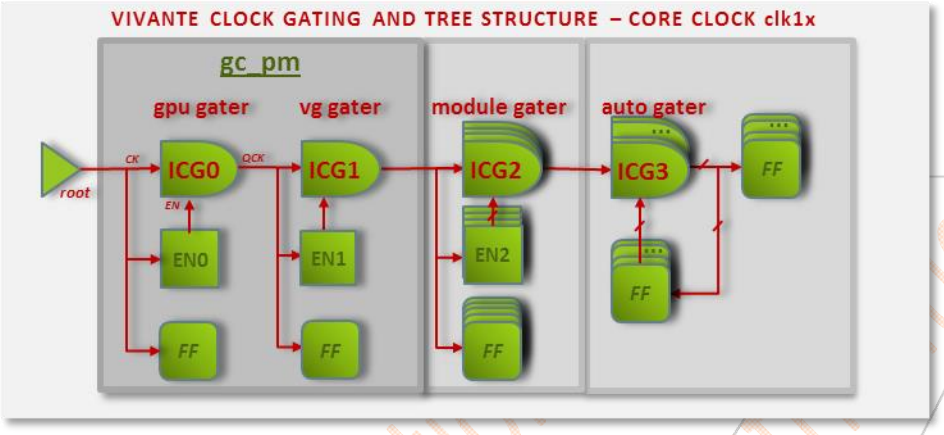

Figure 6. Clock Tree Structure for clk1x

8.4.2 RTL Inserted Clock Gating Cells

These are RTL Inserted Clock Gating cells which control the levels.

This ICG gates the whole GPU

```
pm/u_pm_coreclk/u_pm_pulseskip/GC_CG_PULSESKIP
```

These ICG gate the pipeline and sub-modules:

```
pm/u_pm_coreclk/u_pm_ckgate_2d/GC_CG_CKGATE
pm/u_pm_coreclk/u_pm_ckgate_2d2x/GC_CG_CKGATE
pipe_vg/im/clkGaterIM/GC_CG_MODULECLK
pipe_vg/pe/clkGaterPE/GC_CG_MODULECLK
pipe_vg/ts/clkGaterPE/GC_CG_MODULECLK
pipe_vg/vg/clkGaterVG/GC_CG_MODULECLK
```

8.5 Routing

The design can be routed following the clock tree and high-fanout-net synthesis. In order to reach timing closure, the timing and Signal Integrity (SI) driven routing should be enabled to minimize the design iteration.

8.5.1 Clock Nets

Generally, the clock nets should be routed first. Routing rules should be assigned to the clock nets, such that they are routed in double space. In order to reduce overall clock tree power consumption, avoid the use of double width wire at the leaf level. Clock insertion tools can route the clock network as part of the clock tree insertion process, which is the preferred method. If it is not done this way, you should use the net group routing capability in your layout tool to pre-route the clock nets.

8.5.2 Signal Nets

The remaining nets can be routed using the router functions of your layout tool. Some manual intervention may be needed if there are DRC violations that cannot be resolved by the tool's routing engine.

8.5.3 Antenna Fixes

Antenna DRC requirements will vary with each technology and vendor. Antenna parameters are available from the FAB, and sometimes they are available from the library vendors. Routing tools usually have the capability to repair the majority of antenna violations through metal-only adjustments for a given target metal-to-gate area ratio. However, this step may require diode cell insertion if the top-level design will use a higher metal layer than that of the GPU core. Some standard cell libraries may have antenna diodes built into the cells, so fixing the antenna during routing will not be necessary.

8.6 Parasitic Extraction

Once the routing is clean, the database parasitic extraction can be performed. Usually, a Standard Parasitic Exchange Format (SPEF) file is generated by an extraction program such as StarRC from Synopsys or QRC from Cadence. The extraction tool reads the layout database and calculates the RC parasitics.

Technology parasitic and layer-mapping input files may be delivered with the standard cell library, or these may be provided by the foundry that will manufacture the chip.

It is important to note that parasitic extraction will vary by context. This means that when the GPU core is placed in the top-level design, other nets may be routed near to or over it. These nets will alter the capacitance values inside the GPU core. Therefore, the final timing of the GPU core should be performed in context.

8.7 Timing Closure

8.7.1 Setup Time Violation

Once the Static Timing Analysis (STA)reports are generated, the worst-case setup reports must be analyzed to determine the maximum operating frequency. The frequency capability of a timing-critical path is determined by:

```
1000 / (Clock Period (ns) - Setup Slack (ns)) = Frequency (MHz)
```

If the operating frequency is lower than desired, a detailed analysis of the paths in question may yield a strategy for improving or eliminating the setup violations. This analysis must be performed through the implementation flow. Prime Time full_clock timing reports (report_timing -path full_clock) are useful for determining whether the negative setup slack is a result of clock skew problems.

8.7.2 Setup Time Improvement Method

Methods for achieving timing closure will vary greatly depending on the place-and-route tools used, the quality of placement, the desired frequency, and the severity of the timing violations.

If negative setup slack is a result of inter-clock skew, one of the clocks may need to have its insertion delay increased or decreased by the addition or removal of clock buffers or inverter cells.

If negative setup slack is the result of same-clock skew between different branches, clock buffers or inverters may be added at strategic points in the path to improve the balance. Sometimes the data or clock arrival time can be hastened by improved routing, greater routing width, or higher powered clock buffers. If a certain branch is causing multiple violations, placing clock buffers at higher levels in the tree can alleviate several problems at once.

If setup violations are severe compared to the placement timing reports and clock trees do not induce the severe violation, a postRoute optimization step using the postRoute optimization engine can be used to recover much of the timing slack.

8.7.3 Hold Time Violation

If the placement tool has taken the hold violations into account while doing the timing-driven placement, the hold violations should be minimal at the postRoute stage. If hold violations do exist, they are usually handled by adding some delays at the strategic points of the offending data pins. If there are a large number of hold violations, a script or tool should be used to add delays or swap flops for zero-hold versions at the required locations. Otherwise, the delay cells can be manually added via ECO, and should be placed such that their routing will not impact the timing of other nets.

8.8 Metal and Well Filling

Most process technologies place minimum and maximum utilization requirements on the poly and metal layers. Usually, the layout tool will provide a function to fill the empty metal routing tracks.

8.9 DRC and LVS Checking

A signoff-quality tool for Design Rule Checking (DRC) and Layout Versus Schematic (LVS), such as Mentor Calibre, should be used to verify that the GPU core implementation is error free. Design rule checks may vary greatly between technologies, foundries, and library vendors.

9 Low Power Considerations

To reduce the power in the GPU all memories and almost all flops are clock gated.

Module level clock gating is provided to reduce the power further. Whenever a module is inactive, a signal is generated to gate off the flops in that module. The module level clock gating signals are designed to have 1 cycle slack. They should be inserted close to the root of the clock without any timing violations.

Add minimum logic for BIST and Scan. The flops added for BIST and Scan should be clock gated as much as possible. For optimal power savings, there will be a technology-dependent balance between clock gating cell power and the minimum number of flops to be gated.

To reduce clock tree power consumption, the number of ICG cells auto-inserted by the synthesis tool should be well controlled. It is not necessary to set a strict maximum fanout number for the clock gating cells unless there are clock gating timing violations on certain paths.

To reduce the clock tree power consumption, the clock buffer number should be well controlled. It is not necessary to set strict maximum fanout/transition/capacitance or to use very limited clock inverter/buffer types. Preferably you should set the maximum transitions for the sinks and buffers in order to reduce buffer counts on the clock buffer trees except at the leaf level.

To reduce clock tree power consumption and capacitance, it is not necessary to use wire shielding. It is preferable to use minimum width wires at the leaf level and at the levels close to them. Double space is also preferred.

Running the power measurement tool after the clock tree is inserted is highly recommended. The tool can analyze the clock tree power and the power due to non-clock-gated flops. Any discrepancies found at this stage can be corrected before final tape out.

Check with your Vivante representative for additional information or documentation on low power considerations.

Document Revision History

This section describes differences between document revisions.

Note: This document is not necessarily updated for each patch or minor revision. The information in this document tends to be stable across a revision (nnn) series.

Doc Version	Doc Date	Compatible Hardware	Description
0.82	2022-02-04	GCNanoUltraV e.g., GCNanoUltraV 2000f GCChipDate 0x20210925 GCChipRev 0x00002000 gcregHIChipPatchRev 0x6 gcregHIProductId 0x0302655	Legal Notices, General: Update branding layout to include VeriSilicon. Miscellaneous format refinements.
0.81	2021-11-12	GCNanoUltraV e.g., GCNanoUltraV 2000f GCChipDate 0x20210925 GCChipRev 0x00002000 gcregHIChipPatchRev 0x6 gcregHIProductId 0x0302655	Various: remove F variant Section 4.2.3: add undef Miscellaneous refinements
0.80	2020-07-13	GCNanoUltraV, V2 pre-release e.g., GCNanoUltraV 1001w GCChipDate 0x20210629 GCChipRev 0x00001001 gcregHIChipPatchRev 0x17 gcregHIProductId 0x0302655	Initial Edition