Tessent™ MemoryBIST and JTAG Tutorial

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

RESTRICTED RIGHTS LEGEND 03/97

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Contractor/manufacturer is:
Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/
Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

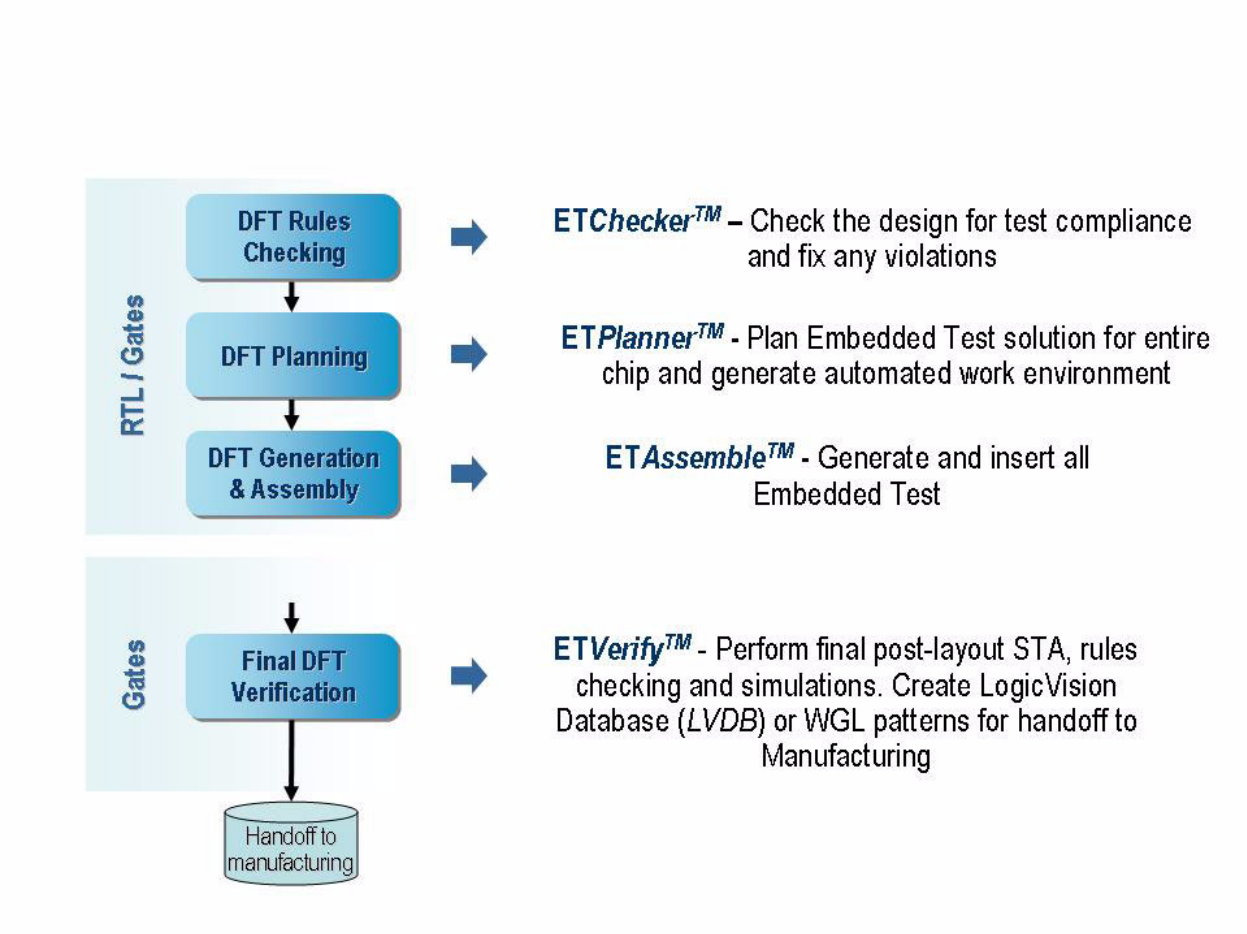
Table of Contents

Tessent™ MemoryBIST and JTAG Tutorial

Introduction

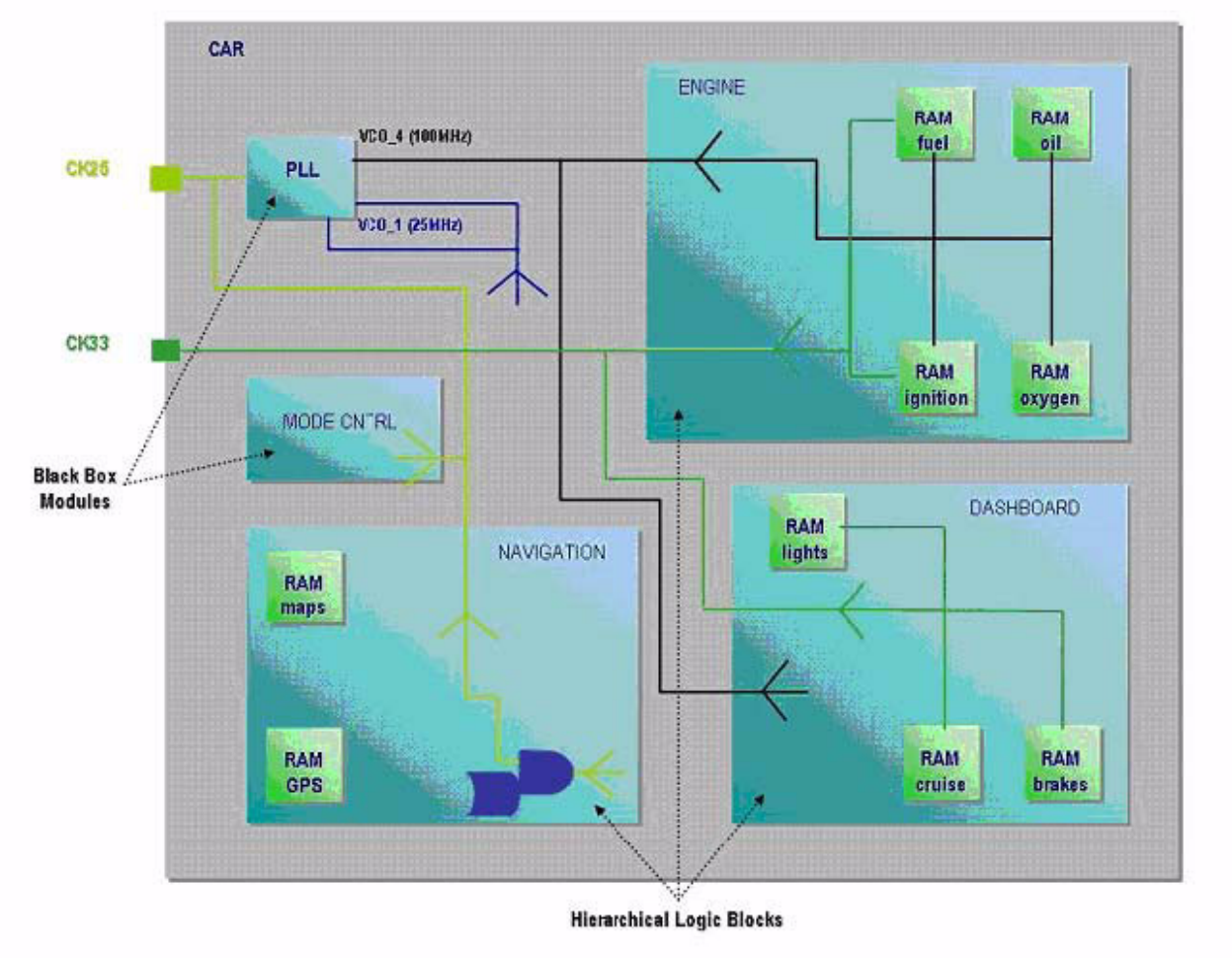
This tutorial is a quick introduction to the Tessent suite of tools dealing specifically with the memory BIST and JTAG portions of the flow. The simplified flow for Tessent MemoryBIST follows the check, plan and execute stages, with execute consisting of assemble and verify. In this tutorial you step through each of the separate stages and become familiar with the design flow.

The following figure highlights the separate stages involved and the appropriate MemoryBIST tool used.



The Tutorial Design

This tutorial uses the CAR design as a basis for the discussion. The diagram below shows the structure of the CAR design and the clocking scheme used.



The starting floor plan of the example design contains the following modules:

- **car**—a top-level module containing:
  - The inputs/outputs/bidirectional pads
  - Sub-blocks, called engine, dashboard, and navigation.
  - Top level logic
  - A PLL
  - The design has logical hierarchy as shown above, but is a physically flat design. For this tutorial you will be inserting the MemoryBIST and JTAG at just the top level.
- **engine**—a sub-block operating at 100 MHz including the following memories:
  - **oil**—a 128x8 single port SRAM
  - **oxygen**—a 64x16 single port SRAM

  - **fuel** and **ignition**—two 64x8 dual port SRAMs. On **fuel** the write port is clocked by the 100 MHz clock and the read port is clocked by the 33 MHz clock. On **ignition** the clocks are reversed.

- **dashboard**—a sub-block operating at 33 MHz including the following memories:
  - **lights**—a 8Kx32 single port SRAM
  - **brakes—**a 64x8 single port SRAM
  - **cruise—**a 64x8 single port SRAM
- **navigation**—a sub-block operating at 25 MHz including the following memories:
  - **sample_ram1**—a 64x8 single port SRAM
  - **sample_ram2—**a 64x8 single port SRAM
- **clocks**:
  - The clocks in this design will illustrate the most common clocking schemes and how to tackle them for BIST. When considering your design, please use these schemes as a starting point to guide you to the recommended tactic for BIST clock scenarios. Here is a description of each one.
  - **PLL—**a 25MHz clock coming in (via **CK25** pin) and generates both 25MHz (via PLL's **VCO_1** pin) and 100MHz (via PLL's **VCO_4** pin) internal clocks. The 25MHz PLL clock is feeding some top level logic. The 100MHz PLL clock is feeding both the **dashboard** and the **engine** blocks. From a BIST point of view, these two 100MHz clocks are synchronous to each other.
  - **33MHz—**a 33MHz clock coming in (via **CK33** pin) and feeds both the **dashboard** and the **engine** blocks. Between these two blocks, the 33MHz clock is assumed to be neither insertion-delay nor skew balanced; however, it is assumed that within each of the two blocks, the 33MHz clock is skew balanced. From a BIST point of view, these two 33MHz clocks are asynchronous to each other.
  - **25MHz—**the 25MHz clock feeding the PLL also feeds the **navigation** block.
  - **navigation block clock division—**the 25MHz inside the navigation block is divided down by a factor of 2 inside the **host_interface** block and used there. The divide-logic is the **clock_divider** block inside the host_interface.
- **other considerations**:
  - **car**—there is a custom boundary scan cell, **in_buf_bscan**, on the pin **EN0**, which has to be taken into consideration for JTAG boundary scan insertion.
  - **car**—there is a not-intended-to-be-tested module, mode_control.
  - **car**—the **PLL** is also not-intended-to-be-tested.
  - **car**—these pins are for JTAG purposes: **TRST**, **TCK**, **TMS**, **TDI**, and **TDO**.

Exercise 1: Tutorial Part I

Initial Setup

The user is required to get the tar file and execute the following commands to extract the tutorial files required for this tutorial:

1. The user should use the following tutorial file:

   tessent_MBIST_Day1.gtz

2. Uncompress and untar the file by executing the following command:

   gtar –zxpf tessent_MBIST_Day1.gtz

3. You should now see the following tutorial directory available:

   tessent_MBIST_Day1

4. Change into the tutorial directory to begin the tutorial:

   cd tessent_MBIST_Day1

   You should see a number of files and directories

   Dolphin/ — Contains the libraries needed.

   DESIGN/RTL — Contains the RTL and the GATE level files respectively.
   DESIGN/GATE

   DESIGN/MEM/ — Contains the Tessent memory and simulation models.

   DESIGN/PLL/ — Contains PLL and RTL files respectively.

   DESIGN/ETChecker/ — The working directory for ETChecker only.

   DFT/ — The working directory for everything else.

   TutorialSource/ — The golden reference copy of the tutorial.

   This tutorial uses a gate-level flow. As such, all modules in the design have been synthesized already and are located in the *DESIGN/GATE* directories. ETChecker will use these files to analyze the design and ultimately forward the information obtained to ETPlanner and ETAssemble. The RTL flow is similar to the GATE flow with the exception that ETChecker analyzes the RTL modules instead. However, the RTL flow requires synthesizing the design modules after the ETAssemble step. Therefore, it is simpler to run the GATE flow with pre-synthesized modules.

   The tutorial is run in the *DESIGN/ETChecker* and *DFT* directories. This directory structure illustrates that the DESIGN and DFT portions can be separated clearly from a functional point of view. The design rule check part of this tutorial is run in *DESIGN/ETChecker/*. The DFT part of this tutorial is run in *DFT*. Start with the DESIGN part and change into the tutorial directory to begin.

```
shell> cd DESIGN/ETChecker
```

DFT Rule Checking (Step 1)

This stage is split into two parts: **clock extraction** which must be done first, followed by **rule checking**. The aim of this stage is to verify the compatibility of the RTL/Gate level chip description with the embedded test insertion flow. All the pertinent DFT information is extracted from the architecture and used to help drive the embedded test insertion flow.

Clock Extraction

1. The first task is to generate a template for the design. For this tutorial you use a flat, top-down methodology so that a template is generated for the top block of the design.

   ```
   shell> etchecker car -gentemplate ON
   ```

   Two files are created: *car.etchecker* and *car.etchecker.README*. The *.README* file describes all the steps needed to modify *car.etchecker* for your specific needs.

2. Edit the template file that was created (*car.etchecker*), to include all the design level information. This file is the key to the design, and specifies all the DFT information.

   a. Specify the target type as Top (since you are checking rules on a complete design — the top-level CAR.)

   ```
   lv.Target -type Top
   ```

   If the root module contains the entire circuit, then Target is specified with **`-type Top`**. If the root module being checked is a core module, then it will have its own logictest controller and use the ELT flow. Therefore, use the option **`-type ELTCore`**. If the root module being checked is any other module, then specify **`-type Block`**. This property is mandatory and must be specified before running in ClockInfo mode.

   b. Embedded test features that you need to turn on are boundary scan (**`bscan`**) and memory BIST (**`memory`**). Since this tutorial only concentrates on boundary scan and memory BIST, logic BIST is being turned off. All other options have no effect, as they only are concerned with logic BIST, and this option must specifically be turned off since the default is **`On`**.

   ```
   lv.EmbeddedTest -bscan On -memory On -logic Off
   ```

   c. Hook up the JTAG pins

   ```
   lv.JTAGOption -pin TDI -option TDI
   lv.JTAGOption -pin TDO -option TDO
   lv.JTAGOption -pin TRST -option TRST
   lv.JTAGOption -pin TMS -option TMS
   lv.JTAGOption -pin TCK -option TCK
   ```

   These pins should have already been created (with their pads connected) in the top-level even though they are not connected anywhere.

d. Add in any black boxes for the design and describe how they are to be handled.

```
lv.BlackBoxModule -name PLL
lv.BlackBoxModule -name mode_control -isolation VerifyExisting
```

A blackbox is any block in your design that cannot be understood and thus not tested by BIST. Having a blackbox in your design will cause you to lose some test coverage (in LogicBIST) in the following way: any path to the inputs of a blackbox, either from a pin or from other flip-flops (FFs), cannot be observed or tested. Any path from the outputs of a blackbox, either to a pin or to other FFs, cannot be controlled or tested. This constraint should be used with caution such that loss of coverage is fully understood and may be compensated via other means.

e. Alternatively, you can type the following command:

```
shell> cp ../../TutorialSource/DESIGN/ETChecker/ \
car.etchecker.pre_etchecker_clock ./car.etchecker
```

This is the modified *car.etchecker* file that you can copy over to save some typing.

3. Once the *.etchecker* configuration file has been edited/copied, it is time to run ETChecker in **`clockInfo`** mode. ETChecker needs the <*designName*>*.etchecker* file along with all design files, libraries, and search directories. Use the run script to help. View the file called *run_etchecker_clocks* that includes the information below. The command **`cat run_etchecker_clocks`** should produce the following:

```
etchecker car \
../GATE/CAR/car.v \.
./GATE/DASHBOARD/dashboard.v \.
./GATE/ENGINE/engine.v \.
./GATE/NAVIGATION/navigation.v \
-y ../PLL \
-y ../MEM \
-y ../../Dolphin/tsmc13/lvision \
-y ../../Dolphin/tsmc13/verilog \
-v ../../Dolphin/tsmc13/verilog/pads.v \
+libext+.v \
-memLib ../MEM/*.lvlib \
-mode clockInfo \
-padLib ../../Dolphin/tsmc13/lvision/pad.library \
-batch off
```

The **`-memLib`** option identifies all the memories in the design that you want to test via memory BIST by pointing to the Tessent memory library files. The **`-padLib`** option identifies all the I/O pad types for JTAG and boundary scan. This option points to the file that defines the pad cells for boundary scan insertion and stitching.

If you have the **`-batch`** option set to off (the default), the GUI for ETChecker will be displayed, and you can run the analysis from there. Run the analysis by selecting the menu option **Run > Analysis**, or click the **Run** button. The GUI also allows design browsing with a schematic viewer by simply double-clicking the module you wish to view and selecting **Tools > Modular Schematic**, or **Tools > Incremental Schematic**

You also can click the **Incremental Schematic** or **Modular Schematic** button). The Incremental Schematic is useful for troubleshooting errors. Choose **File > Exit** to exit.

You always can run `etchecker -help` for more information on usage.

```
shell> run_etchecker_clocks
```

4. Click the **RUN** button

   You now need to review any errors, warnings and info messages. This can be done within the GUI or by checking the logfiles that were generated. Within the GUI there is a Tab (in the lower portion of the screen) that will have all the errors listed in a drill down fashion.

   Once you have run the analysis in the GUI, you will see some INFO, ERROR and/or WARNING severity messages. Most likely, for this run you only see INFO messages. However, if there are WARNING messages, for this tutorial, they safely can be ignored. These are mostly information messages on where to find analysis results. To review these information messages, look at *etCheckInfo/etchecker.log_clockInfo* and *etCheckInfo/etchecker.rpt_clockInfo* for more detailed information. ETChecker uses the Atrenta Spy Glass tool to do the analysis of the design (part of this is a quick internal synthesis of the design). The rule order for ETChecker is

Table -1.

Severity Message	Action
Fatal	Must fix all errors
Error	Fix most errors
Warnings	Review carefully
Info	Background information

5. Sometimes you may want to view an error visually, or you may just want to look at the schematics of a design. Should you want to view either the Modular Schematic or Incremental Schematic of an element in the design, first double-click the item with the LMB. This activates the **Modular Schematic** and **Incremental Schematic** buttons in the toolbar.

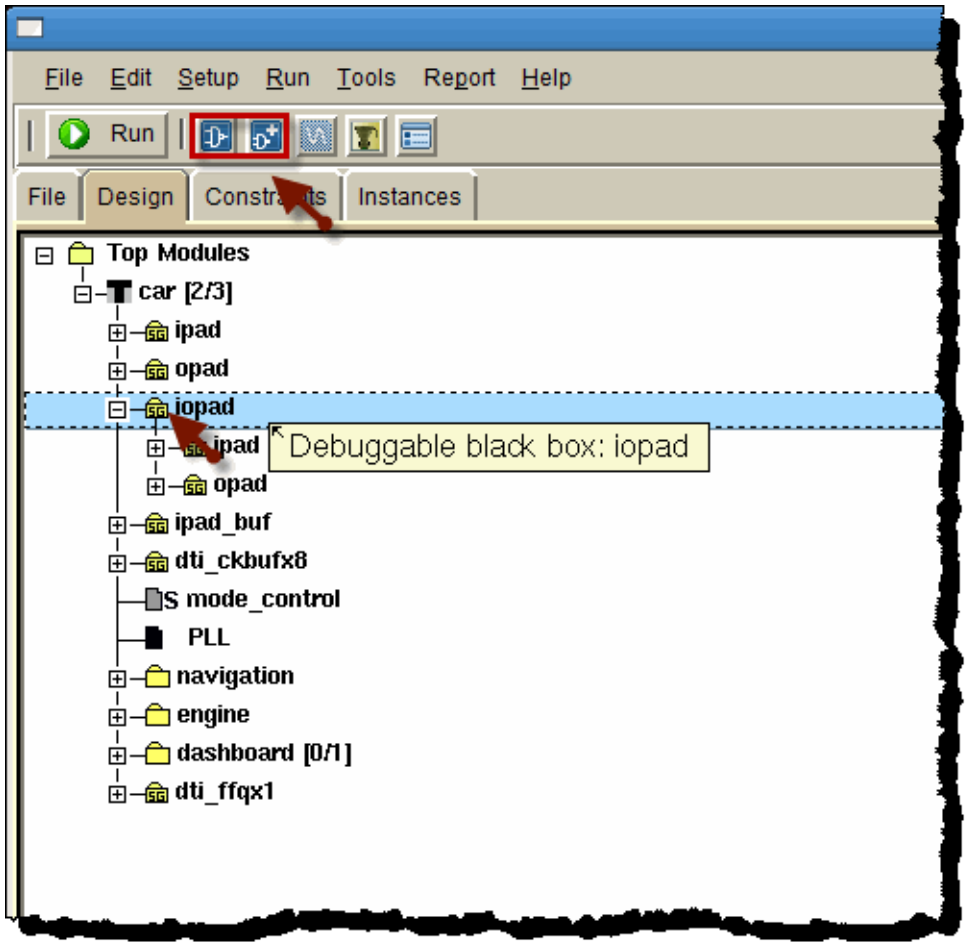
   For example, in the upper left hand pane of the GUI, there are four tabs: **File, Design**, **Constraints**, and **Instances**.

   a. Click the **Design** tab.

   b. Expand the dirctories **Top Modules/car [2/3]**.

   c. Double-click the *iopad* folder.

   d. Click the **Modular Schematic** button. The schematic of design elements loads into the Modular Viewer window.



Experiment with the tabs and folders to see what elements can be viewed in either one or both schematic viewers.

6. When you have finished exploring the schematic and other items while you are in the SpyGlass GUI for *clockInfo* mode, close the window by clicking the 'X' in the upper right hand corner of the window. If you are prompted 'Are you sure you want to exit SpyBlass?', click **Yes**.

7. Two files will have been created in the directory *etCheckInfo/etp/*. These files are either incorporated into the *car.etchecker* and are subject to your review:

- *car.etpClockTree*

  This file contains a clock graph of your circuit. Examine this graph to confirm the clock domains are correctly understood by etchecker. If the graph is not correct, you may be missing or have incorrectly specified constraints in your *.etchecker* configuration file.

- *car.etpClockDomainInfo*

This file contains a list of clock domain bases and clock sources found within your circuit. Copy the extracted **lv.clockDomainBase** and **lv.InternalClockSource** constraints from this file to your *car.etchecker* configuration file and adjust the **-frequency** and **-label** options, or copy the *car.etchecker.pre_etchecker_rules* file from the *TutorialSourc*e directory.

Type the following command to include the clock information in your *car.etchecker* file:

```
shell> cat etCheckInfo/etp/car.etpClockDomainInfo >> car.etchecker
```

You need to open the *car.etchecker* file and go to the bottom of the file. Once there, update the clock domain base and internal clock source information to match the following:

```
lv.ClockDomainBase -pin "car.PLL.VCO_4" -frequency 100
-label CLK100MHz -polarity 1
lv.ClockDomainBase -pin "car.CK33" -frequency 33 -label CLK33MHz
-polarity 1
lv.ClockDomainBase -pin "car.CK25" -frequency 25
-label CLK25MHz -polarity 1 -injectPin car.CK25MHz_CLK.A
```

These constraints are used to define the base of clock domains. The module name must always be the root module.

```
lv.internalClocksource -pin "car.PLL.VCO_4" -referencePin CK25
-freqRatioRelToPin 4
```

This constraint is used to define pins of a module as internal clock sources. Any clock driving memories in your design should have a corresponding constraint here, unless your memory is driven by a Clock Domain Base (CDB) described above.

The analysis of the clock domains is done in such a way that clock domains are reported back to the user when one of the following occurs:

a. The source was traced back to a primary input

b. The source was traced back to an output of a black box

c. The source was traced back to an output of a register (a clock divider is a common example)

For this design, the 33MHz clock does not go through any clock manipulation before feeding into the dashboard and engine blocks. Thus, it only shows up as a **lv.ClockDomainBase** constraint but not as a **lv.internalClocksource** constraint.

The 25MHz clock does not go through any clock manipulation, similar to the 33MHz clock. Off this 25MHz clock, there is this internal divide-by-2 logic inside the navigation block.

The file *car.etpConstraints* in the *etCheckInfo/etp* directory is also a crucial file to review. This file lists the specified constraints that were ACCEPTED by the tool, and

then also lists the inferred constraints that the tool used. These need to be reviewed carefully to ensure that ETChecker successfully used all your specified constraints and that the inferred constraints are acceptable. If this is not the case, your *car.etchecker* file needs to be modified, as this is the control file of the tool.

Alternatively, you can enter the following command:

```
shell> cp ../../TutorialSource/DESIGN/ETChecker/ \
car.etchecker.pre_etchecker_rules ./car.etchecker
```

This is the modified car.etchecker file that you can copy over to save some typing. You will find all of the information that you just typed in this file.

Rule Checking

ETChecker is again used but this time in **`ruleCheck`** mode, rather than the **`clockInfo`** mode used previously. ETChecker uses the <*designName*>*.etchecker* file that was modified by the user and contains the correct clock info, along with all the design files, libraries and search directories. Use the run script to help. View the file called *run_etchecker_rules* that includes the following information.

1. Issue the command **`cat run_etchecker_rules`** to produce the following:

```
etchecker car \
../GATE/CAR/car.v \
../GATE/DASHBOARD/dashboard.v \
../GATE/ENGINE/engine.v \
../GATE/NAVIGATION/navigation.v \
-y ../PLL \
-y ../MEM \
-y ../../Dolphin/tsmc13/lvision \
-y ../../Dolphin/tsmc13/verilog \
-v ../../Dolphin/tsmc13/verilog/pads.v \
+libext+.v \
-memLib ../MEM/*.lvlib \
-padLib ../../Dolphin/tsmc13/lvision/pad.library \
-mode ruleCheck \
-batch off
```

Note the **`-mode`** switch is set to **`ruleCheck`**. Again SpyGlass is used for the analysis.

If you have the **`-batch`** option set to off (the default), the GUI for ETChecker will be displayed, and you can run the analysis from there. Run the analysis by selecting the menu option **Run > Analysis**, or click the **Run** button. The GUI also allows design browsing with a schematic viewer by simply double-clicking the module you wish to view and selecting **Tools > Modular Schematic**, or **Tools > Incremental Schematic** You also can click the **Incremental Schematic** or **Modular Schematic** button). The Incremental Schematic is useful for troubleshooting errors. Choose **File > Exit** to exit.

```
shell> run_etchecker_rules
```

2. Once inside of the GUI, click the **Run** button.

3. Review messages.

   You need to review any errors, warnings and info messages. Do this within the GUI or by checking the logfiles that were generated.

   a. GUI—Within the GUI there is a tab (in the lower portion of the window) that has all errors listed in a hierarchical fashion. Click down into some of the messages to review.

      You will see a few INFO severity messages, filtered messages and warnings. The INFO severity messages are mostly information on where to find analysis results, while the filtered messages are mostly analysis results for this particular design, such as the number of ports, instances, etc. Warnings are design-related recommendations. All of these messages should be reviewed and can be safely ignored afterward.

   b. Informational Messages—To review these messages, review *etCheckInfo/etchecker.log_ruleCheck* and *etCheckInfo/etchecker.rpt_ruleCheck* for more detailed information. The files *etCheckInfo/car.etpClockTree* and *etCheckInfo/car.etpClockDomainInfo* should also be reviewed again to make sure that these structures agree with your design intention.

   Upon successful completion of ETChecker in **`ruleCheck`** mode, a file, **`car.etCheckerInfo`**, is created in the *etcHandoff* directory. This file is created only when there is no error or fatal severity message reported.

4. When you have finished, exit SpyGlass.

DFT Planning (Step 2)

Now that design checking is complete, you can enter into the planning stage. This is the start of the DFT portion of the design flow. During this stage you make test trade-offs based on several criteria including: test quality, test time, power and area. These trade-offs are made based on all design constraints extracted from the previous stage using ETChecker. The prerequisite of this stage is the file <*module*>.*etCheckerInfo* located in the *etcHandoff* directory. This file feeds forward information gained from the previous step. This flow generates the test plan, specifies test requirements, validates that test requirements are met, and generates the environment.

1. Change into the *DFT* directory.

   ```
   shell> cd ../../DFT
   ```

   Notice that there is an existing directory called *CentralFiles* that contains the global CAD environment files. These will be discussed and used later on in the tutorial.

Plan Generation

Global definition files are useful to maintain consistency between design groups, and even within a design and multiple modules. When global definition files are used by everyone on the

same project, they create consistency such that maintenance and support by a single DFT entity is possible, and no designers are left behind for lack of environmental details.

For the purpose of this tutorial, and to help highlight their use, you use global definition files for the **CADEnvironment** and **ICTechnology** sections of the *car.etplan* file. These can be conveniently passed to ETPlanner by the **CADEnvFile** and **ICTechFile** options to ETPlanner.

For this stage, the ETPlanner tool is used. Remember that it requires the *.etCheckerInfo* file, along with libraries and search directories. Use the run script to help. Review the file called *run_etplan_gen* that includes the following information.

1. Enter the following command at the shell prompt:

   ```
   shell> cat run_etplan_gen
   ```

   This produces the following information:

   ```
   etplanner car \
   -mode genPlan \
   -etCheckerInfoFile\
   ../DESIGN/ETChecker/etcHandoff/car.etCheckerInfo \
   -CADEnvFile ./CentralFiles/HWLib.CADEnv \
   -ICTechFile ./CentralFiles/Dolphin_tsmc13.LVICTech \
   -etDefFile ./CentralFiles/HWLib.ETDefaults \
   -etplanFile car.etplan \
   -physicalInfoFile car.physicalInfo \
   -memLib ../DESIGN/MEM/*.lvlib \
   -outDir outDir \ -log etplanner.log_genPlan
   ```

2. Enter the following command at the shell prompt:

   ```
   shell> run_etplan_gen
   ```

   The plan generation stage will create a plan file, *car.etplan*, that specifies all DFT requirements to be implemented, together with a *Makefile*. You will be editing the *.etplan* file next.

   You always can run **etplanner -help** for more information on usage.

   This step generates a *car.etplan.README* file for reference, providing more details on the *car.etplan* file.

Note

If you get an error message **"1 out of 3 hunks FAILED -- saving rejects to file outDir/car.etplan_beforePatching.rej, (etcetera)..."** delete the file *outDir/car.etplan_diffs* and rerun ETPlanner. Note that this action—whether you delete the file or not—will not impact the outcome of the flow.

3. Edit the car.etplan file.

Ensure that the **CADEnvironment** and **ICTechnology** sections point to the files in the *CentralFiles* directory.

4. Within the **DesignSpecification** section, edit the **ModulesGate** section to point to the appropriate locations for the GATE level files. If you were using an RTL flow, you also would have to edit the **ModuleRTL** section in order to point to the appropriate location for these files (refer to *car.etplan.README* for more information).

```
DesignSpecification {
   RTLExtension : vb;
   GateExtension : v;

   ModulesGate( car ) {
      SimModelDir (CAR_GATE) : ../DESIGN/GATE/CAR ;
      SimModelDir (ENGINE_GATE) : ../DESIGN/GATE/ENGINE ;
      SimModelDir (DASHBOARD_GATE) : ../DESIGN/GATE/DASHBOARD ;
      SimModelDir (NAVIGATION_GATE) : ../DESIGN/GATE/NAVIGATION ;
   }
} // DesignSpecification
```

   Each **SimModelDir** results in a soft link being created (with the name specified within parenthesis), linking it to the directory. A soft link approach is used, because if there are any changes in the design, and multiple designers are working on it, there is only one place to change the design and everyone receives the changes simultaneously. This saves time and effort in maintaining consistency across all designers. Again, your internal revision control methods have to be considered in conjunction with BIST set up.

5. The next section in the *car.etplan* file, **EmbeddedTest**, is the start of the planning for the design. Here you describe what you want to happen, and also confirm and update what DFT structures have already been found in the design. This section also highlights what items in *./CentralFiles/HWLib.ETDefaults* differs from the built-in default. If the **GlobalOptions** wrapper in your *.etplan* file does not contain the statement **EmbeddedTestMergeFlow : Gate ;** add this now.

   Within the **ModuleOptions** wrapper, you specify all of the manufacturer information for your device. This information is read out at the board level to ascertain which parts from various manufacturers, as well as each part's revision, have been placed on the board. This information is checked during manufacturing test. If these fields are specified as anything other than 0s (the default), they will be forwarded automatically to each *.etassemble* configuration file in each *ETAssemble* directory of each workspace. Note that these values can be specified using hex or binary. Specify this information by adding the following lines to your **ModuleOptions** wrapper:

```
ModuleOptions (.*){
   deviceIdCode : 16'h0101 ;
   RevisionCode : 4'hA ;
   ManufacturersIDCode : 11'h01A ;

}
```

6. The **MemBistStepOptions** section details how the memory BIST controller will be configured. The algorithm is selected, how to configure interfaces, what type of comparator is needed, and what to do with the Compstat signal.

```
MemBistStepOptions(.*:.*) {
   RAMAlgorithm : SMARCHCHKBCIL;
   LocalComparators : No;
}
```

7. The next *car.etplan* section, **Auto Generated Embedded Test Specifications** is an automatically generated section, and should be reviewed to ensure that all memories have been identified correctly, and was generated in compliance with all specifications. It is highly recommended that if you copy the file from *TutorialSource,* that you use the section that you copied. If you need to make corrections, it is best to go and adjust the **EmbeddedTestDefaults** specification in the *CentralFiles/HWLib.ETDefaults* file. Also, you can define different **clusterIDs** for you memories in the *.etchecker* file.

8. Once again, to save time you can copy the file from the *TutorialSource* directory:

```
shell> cp ../TutorialSource/DFT/car.etplan.pre_make_checkplan \
./car.etplan
```

   This is a modified *car.etplan* file that you can copy over. Even if you choose not to copy, you can always use this file as a reference—it contains information needed to complete the DFT planning step.

9. Once *car.etplan* has been edited and confirmed, you can run **make checkPlan**. This target uses the *car.etplan* file that you just modified,

   Run the make target:

```
shell> make checkPlan
```

   The result can be viewed in the *car.ETSummary* file under the *outDir* directory. This shows you what the embedded test features for each block will look like. If these do not meet your specifications, you either need to return back to the ETChecker section, or modify the plan to get the desired results.

```
Module CTLType   CTLName                 TestTime   Power     Warnings
car      MBIST car_CLK100MHz_MBIST1   120 us
                  Step: 0                          240 mW
         MBIST car_CLK33MHz_MBIST1     11 ms
                  Step: 0                          132 mW
         MBIST car_CLK25MHz_MBIST1    126 us
                  Step: 0                          35 mW
```

   Furthermore, also in the *outDir* directory, the *etplanner.log_checkPlan* file reveals the following warning:

   Warning [PPHW01-00371328] : You have run ETChecker with 'lv.EmbeddedTest -memory on -logic off', but you did not specify the ThirdPartyATPGTestModePin in your .etplan file. It is recommended that you specify this property to ensure the Tessent memory BIST logic will be placed in scan bypass mode when used with your 3rd party scan solution.

This warning is pretty self explanatory and safely can be ignored for this tutorial.

10. Once the plan has been confirmed, the final step is to generate the embedded test workspaces so that the plan can be finalized.

```
shell> make genLVWS
```

This make target generates all embedded test workspace directories required to implement the plan. For the top block, there is a corresponding *_LVWS* directory, *car_LVWS*. There also is the technology library directory, *Dolphin13* and the directory for the aggregation of the concatenated netlists, *concatenated_netlists*. You move to the *car_LVWS* directory next and follow the *.README* directions. The basic flow consists of using the created *Makefiles* to automate the entire test insertion and verification process. A *.README* file is always generated to accompany a *Makefile* in order to provide guidance. You start the flow of the next section in the *car_LVWS* sub-directories.

The *outDir/car.membist_checkPlan* file describes the memory BIST architecture that will be implemented.

The *car.ETSummary* file describes embedded test features that will be implemented.

This completes the ETPlanner stage, and you are now ready to move on to the ETAssemble stage.

Review of DFT Rule Checking and DFT Planning

In the previous steps, two stages of the Tessent memoryBIST design flow were shown:

- ETChecker was used to check the design for test compliance. It ran in two modes: clock checking mode followed by the rule checking mode. Successful completion of this section was demonstrated by the generation of *etcHandoff/car.etCheckerInfo*.
- ETPlanner was used to plan the embedded test solution for the entire chip and then generated the automated work environment. Successful completion of this section was demonstrated by the generation of the LV workspace directories and the *car.ETSummary* file, describing exactly what is required to complete the process.

Exercise 2: Tutorial Part II

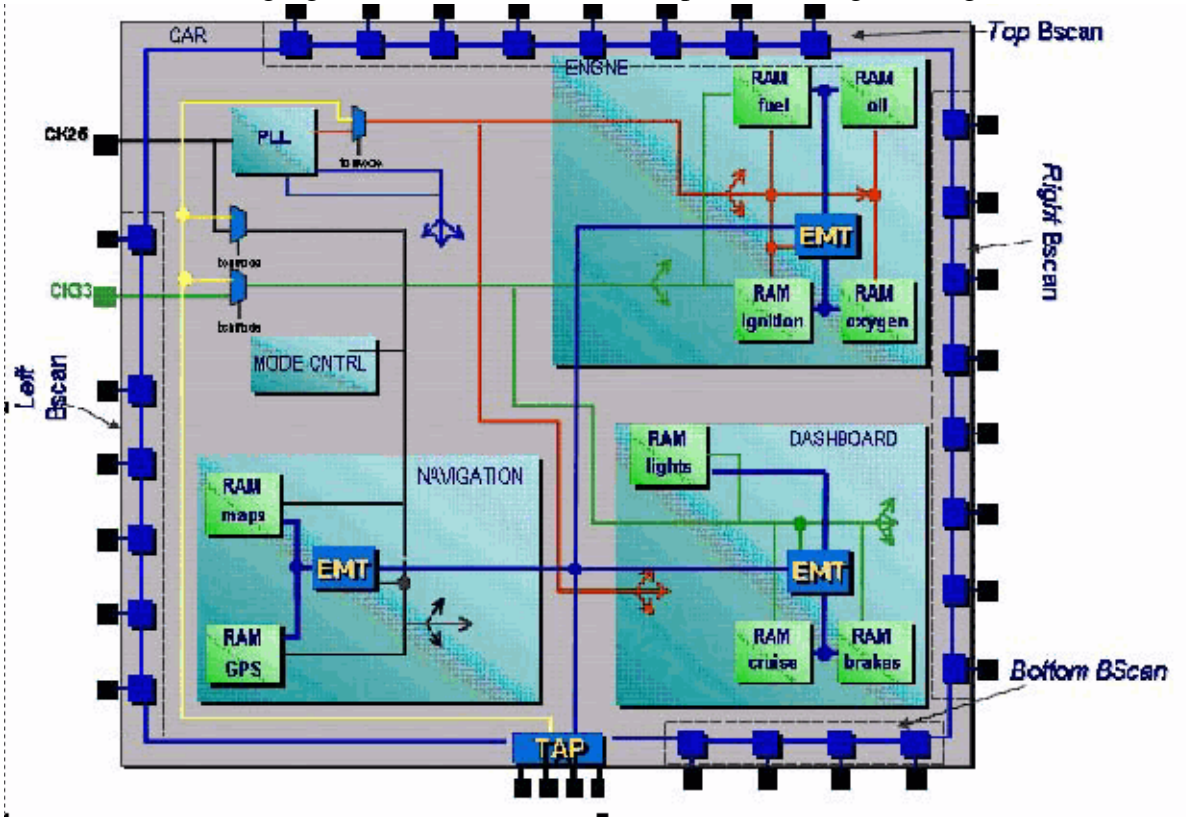
Previously, you checked the design and prepared the plan for the embedded test implementation. In this section you look at actually implementing the solution. In this stage you create the memory BIST controller, memory interfaces, and verification testbenches using Tessent's memory BIST generation and verification tool. The generated testbenches are then used to simulate the memory BIST controller and interfaces before they are assembled into the design. The simulation can be run on both the RTL and gate-level files of the BIST circuits.

Simulating the controller and interfaces at this point is always recommended, so that any problem caused by incorrect information in a setup file can be debugged easily and early in the process, when there is less logic to inspect.

ETPlanner created Makefiles along with *.README* files which are supplied for easy and efficient execution of these tasks.

There are 3 steps to complete this stage: ETAssemble, synthesis, and ETSignOff.

Refer to the following figure to review what is accomplished during this stage.



There are EMTs (Embedded Memory Test) blocks for the appropriate modules. At the top-level there is a TAP to control the EMTs. Boundary scan cells also are inserted and grouped into appropriate physical placement blocks.

DFT Generation and Assembly (Step 3)

1. Change to the *car_LVWS* directory.

   ```
   shell> cd car_LVWS
   ```

2. Inspect the *.README* file.

3. Move to the *ETAssemble* directory.

   ```
   shell> cd ETAssemble
   ```

4. View the *.README* file. This file indicates the steps that must be followed. You should use this the tutorial in conjunction with this *.README* file to successfully complete the ETAssemble step.

5. The first step is to prepare the boundary scan pin order file. If you skip this step and the pin order file does not exist, a default file, *car.pinorder* will be created automatically during the next step.

   ```
   make embedded_test cmdOptions="-genPinTemplate On"
   ```

   View the output *car.pinorder* file.

   Notice that columns PinType and Side are blank. You would edit this file per your package numbers, following the steps outlined in the *.pinorder* file.

   How you group boundary scan pins influences how boundary scan cells are grouped. This affects the back end of the process.

   For now, copy the *.pinorder* file from *TutorialSource*:

   ```
   shell> cp ../../../TutorialSource/DFT/    \
   car_LVWS/ETAssemble/car.pinorder.pre_embedded_test ./car.pinorder
   ```

   Detailed instructions are available at the top of this file to enable so you can customize it. The *.pinorder* file provides package pin data as well as the boundary scan chain order information. It also provides ETAssemble with special pin information such as:

   - Pins that are reserved for JTAG (TDI, TDO, TMS, TCK, TRST)
   - Pins that are power, ground, or are not found in the netlist, but exist on your package
   - Pins that are clock, set, or reset
   - Pins which cannot have a boundary scan cell on them
   - A definition of user-specified groups for the boundary scan cells

6. Generate and insert the TAP, boundary scan and memory BIST controllers into the root module—CAR. Edit the *car.etassemble* file if you need any customizations.

   ```
   shell> make embedded_test
   ```

The generated BIST output files are written into the local *outDir* directory. The BIST-inserted RTL files are written out to the same directory the tools read them in from. The modified RTL files are written out with a different extension (defaulted to *_et*) to distinguish them from your original files. **Any time a functional change to your RTL or gate level netlist is needed, your ORIGINAL RTL/gate files should be modified and the process re-run. The memory BIST-generated files should never be modified.** The generated *Makefiles* bring a level of automation and repeatability to the DFT insertion task.

It is always beneficial to check your log files in the *outDir* directory to make sure that there are no error messages. If there are, every error message should be fixed and every warning should either be fixed or completely understood before moving forward with the flow. The messages are usually self-explanatory and corrective actions can be deduced from them.

7. The second make target to run is **`designe`**. The *car.tcm* file is created by designExtract, which parses the design hierarchy and looks for all the BIST instantiations and connections. The output *car.tcm* is a prerequisite of the steps that follow this one.

   ```
   shell> make designe
   ```

   The output log file is in *outDir/designe.log_car*. The make target uses the file *car.designe* as input.

8. The next make target *config_etSignOff* generates the *car.etSignOff* testbench configuration file. This file contains a list of test steps that are required to perform early verification of all embedded test features in the car design. You typically do not need to edit this file.

   ```
   shell> make config_etSignOff
   ```

9. The make target **`synth`** runs the TCL script *outDir/car_synthesis.script_tcl* to synthesize all embedded test structures. Note that this synthesis step only synthesizes the Tessent- generated RTL. It is up to the user to synthesize design files (if the design is in RTL). For this tutorial, this has been done for you.

   ```
   shell> make synth
   ```

10. Generate the post-Tessent netlist containing all the synthesized modules and embedded test structures.

    ```
    shell> make concatenated_netlist
    ```

11. Generate a pre-layout circuit database. This step is required to create a vector-less signoff database, testbenches or WGL patterns for your design. The generated *lvdb* will be stored in the *../ETSignOff* directory.

    ```
    shell> make lvdb_preLayout
    ```

12. Generate the testbench based on the *car.signOff* configuration file and the data from the pre-layout *lvdb*.

```
shell> make testbench
```

13. Simulate the testbench created preciously. This will simulate all the test steps for all the controllers specified in the *car.etSignOff* configuration file with the pre-layout netlist.

```
shell> make sim
```

Final DFT Verification and Sign-Off (Step 4)

The final step to perform is the checking of the top level and generating a *lvdb* database. This is done only when the design is ready to be signed off, i.e., after the physical design is complete.

1. Move to the *ETSignOff* directory.

```
shell> cd ../ETSignOff
```

2. Inspect the *.README* file. This file indicates the steps that must be followed. You should use the tutorial in conjunction with the *.README* file to successfully complete ETSignOff.

3. Since this is the top level sign off of the design, the post-layout netlist will be used to convey all information needed to complete this step. Verify that the following file exists before going further:

   ../../concatenated_netlists/car.netlist_final

4. For simplicity, create a soft link pointing *car.netlist_final* to *car.netlist_prelayout*:

```
shell> cd ../../concatenated_netlists
shell> ln -s car.netlist_prelayout car.netlist_final
shell> cd ../car_LVWS/ETSignOff
```

5. Generate the *car.etManufacturing* pattern configuration file(s).

```
shell> make config_etManufacturing
```

   This file contains a list of all test steps to be run on a tester during IC manufacturing test. Do not modify the **`SimulationScript`** properties because the *Makefile* was created assuming the default values will be used.

6. Generate the Tessent circuit database. This step is required to create a vector-less signoff database. Testbench and/or WGL pattern generation is also dependent upon the existence of the database. The primary purpose of having a *lvdb* is to have a vector-less signoff database.

```
shell> make lvdb_final
```

   The generated lvdb will be stored in the directory *../../finalLVDB*.

7. Generate the testbench based on the *car.etSignOff* configuration file that was generated in the ETAssemble step, and the data from the *lvdb*.

```
shell> make testbench
```

8. Run the simulation script. This simulates all test steps of all controllers that are specified in the c*ar.etSignOff* configuration file using the concatenated netlist.

   ```
   shell> make car_sim
   ```

9. Generate the patterns for manufacturing. The manufacturing test is specified in the *car.etManufacturing* file.

   ```
   shell> make patterns
   ```

   This is the end of the tutorial. Hopefully, this tutorial has provided you with a quick introduction to the Tessent MemoryBIST and Tessent BoundaryScan suite of tools and flow. To reiterate, the flow for the design is: check, plan, and execute, with execute consisting of assemble and verify. You have stepped through each of the separate stages and have become familiar with the flow and used many of the features. Please feel free to provide feedback.

NOTES: