Functional Verification from Chaos to Order: Using Continuous Integration for Hardware Functional Verification

Abdelouhab Ayari, Product Engineer, Siemens EDA, Munich, Germany
(*abdelouahab.ayari@siemens.com*)

Kirolos Mikhael, Product Engineer, Siemens EDA, Cairo, Egypt
(*kirolos.magdy@siemens.com*)

***Abstract*- In recent years, continuous integration and development have become crucial in organizing software development cycles. As a result, it has also become a way to streamline hardware flow, especially with the increased complexity of chips and SOCs. Integrating various teams that may be working on different systems, sub-systems, and IPs can be a challenge and may cause integration chaos. CI/CD plays an important role in organizing the hardware flow and making integration seamless, easy, and trackable using the ability of running multiple jobs in parallel it is capable of accelerating the workflow. This paper discusses the ability of hardware CI flow to accelerate the functional verification flow and increase productivity and quality.**

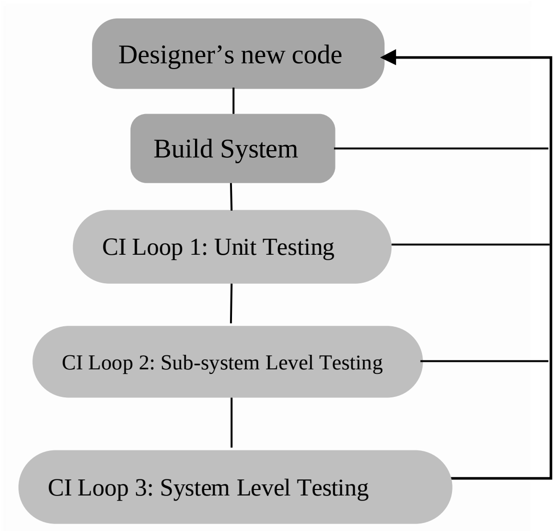
I. Introduction

As system complexity grows and tool performance hits the performance wall, the focus shifts towards enhancing productivity using existing tools. This paper will demonstrate how to leverage the CI flow in hardware to boost productivity, making the process more adaptable and smoother.
Wilson Research [1] reports that 24% of time is spent on creating tests and running simulations, and 41% on debugging. Using continuous integration (CI) in functional verification ensures automation and tight integration between various functional verification tools, resulting in reduced time and effort. This paper will cover the process of building a strong integration and its positive effects on time and effort savings.

II. Continuous Integration Framework

This chapter will cover the continuous integration framework which is designed to test code in an automated manner. With modern SoCs comprising multiple subsystems, the CI framework is implemented at both the subsystem and chip levels. This allows the code to be integrated into larger sub-systems and systems. The CI framework involves multiple loops, as depicted in Figure 1, where each loop represents a bigger subsystem [2]. The code passes through these loops until it becomes stable enough to be integrated into the main streamline of the chip/SoC.

Unit testing is a software method used in the validation of the smallest parts of the software [3]. Unit testing offers various benefits, such as detecting bugs at an early stage, improving code quality, aiding in refactoring, accelerating development, and enhancing collaboration


Designer's new code
Build System
CI Loop 1: Unit Testing
CI Loop 2: Sub-system Level Testing
CI Loop 3: System Level Testing
Figure 1: Continuous Integration Framework Loop

Next, after the unit testing process comes sub-system level testing, which involves integrating your units into a larger sub-system. To illustrate, if we consider a processor design, we can divide the system into multiple units such as the Address Unit (AU), Execution Unit (EU), Bus Unit (BU), and Instruction Unit (IU). Suppose the designer is working on a decoder in the IU; we can perform unit testing on the decoder and then integrate it into the IU for sub-system testing. After that, we can integrate this with all the other units and perform system-level testing. If the processor is part of another system, we can extend the testing loop even further. By adopting this approach, we can automate testing more efficiently and obtain results faster and earlier in the design stage, thus saving ourselves from integration difficulties or what the designers call "Integration Hell".

Using the provided framework flow, we can delineate the algorithmic perspective as depicted in "Algorithm 1". This will assist in crafting the code necessary to steer our CI system, ensuring robust integration and a holistic testing strategy given that every CI loop is based on our CI system infrastructure which will be discussed in the following section.

ALGORITHM 1: CONTINUOUS INTEGRATION FOR DESIGNER'S CODE

```
    Input: DesignerCode (The new code provided by the designer)
    Output: Status of the CI system
1   function BUILD(DesignerCode)
2       | return DesignerCode is valid ? Compile and Test () : Error
3   end function

4   function UNIT_TEST(DesignerCode)
5       | for each unit in it the DesignerCode: if not test(unit) then return Failure else return Success
6   end function

7   function SUBSYSTEM_TEST(DesignerCode)
8       | for each submodule in System that contains the DesignerCode: if not test(sub) then return Failure else
        | Success.
9   end function

10  function SYSTEM_TEST(DesignerCode)
11      | For the DesignerCode integrated in the full system; if not test(system) then return Failure else Success
12  end function

13  if BUILD(DesignerCode) is Error: Output "Build failed"; Stop
14  if UNIT_TEST(DesignerCode) is Failure: Output "Unit test failed"; Stop
15  if SUBSYSTEM_TEST(DesignerCode in SubSystem) is Failure: Output "Sub-system test failed"; Stop
16  if SYSTEM_TEST(DesignerCode in FullSystem) is Failure: Output "System test failed"; Stop
17  Output "All CI steps passed!"
```

III. CI SYSTEM INFRASTRUCTURE

To carry out the tests outlined in the preceding chapter, we require a robust infrastructure for our CI system that oversees the testing process and ensures comprehensive feedback and transparency. As depicted in Figure 2, we present our suggested infrastructure for the CI setup. This system encompasses several functional units, including the repository project data, the CI tool, the run process, a database for collating data, and a method for data visualization.

The project repository is overseen by the version control system and the CI system provides essential metrics and statuses to facilitate commit gating. For instance, when employing mechanisms like the 'pre-build branch merging' git plugin[5], you'll need to specify the criteria for a successful merge. The CI system can supply this information through communication APIs, bridging the CI tool and the run process. This run process encompasses procedures for static, formal, and simulation verification, delivering results in the form of a pass/fail metric.

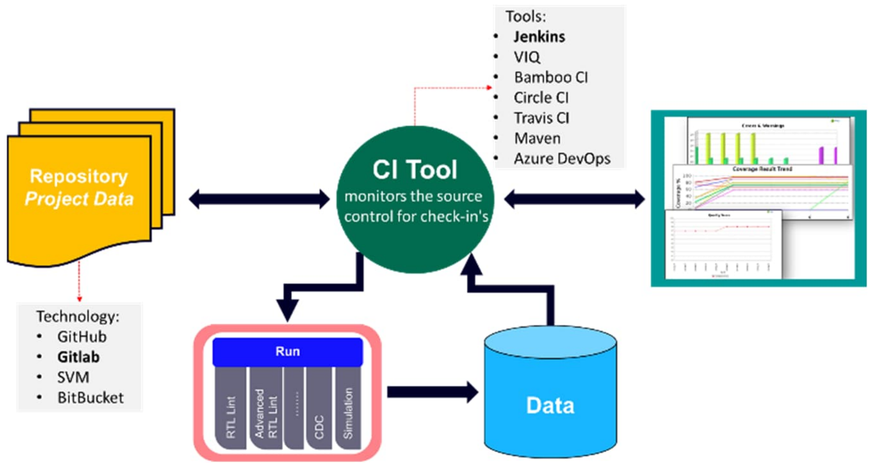


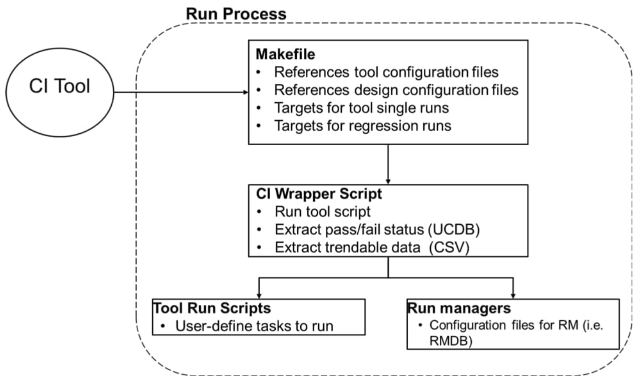
Figure 2: Continuous Integration System

In the run process stage, this primarily involves executing make targets and required scripts for simulation, static, and formal verification tools. During this stage, the CI system employs the provided APIs from these tools to gather all necessary data. By utilizing the run process with the tools and associated APIs, the CI system can generate data and metrics in multiple and varied formats (CSV, UCDB, XML, etc.). This data requires a communication layer between the static, formal, and simulation tools and the database portion of the CI system. This communication layer is achieved by a script that seamlessly runs the tools' APIs and creates the necessary data for debugging and visualization in the formats mentioned earlier and the data can be archived either by complicated databases or with the native database techniques from the CI tools.

Diving deeper into the CI infrastructure, we can explore the communication pathways and understand how data is sent and produced among various components and the CI tool. Initially, let's discuss the connection between the Source Code Management (SCM) and the CI tool. This connection largely relies on the SCM in use, and currently, multiple methods exist to establish this link, mainly through the APIs offered by the SCM. For instance, when "git" is utilized as an SCM system, it becomes straightforward to implement webhooks that detect code modifications and prompt the CI tool to initiate the associated tests.

Another vital connection to understand is the one between the CI tool and visualization. Given that the execution phase might yield varied outputs from processes like functional verification via static, formal, and simulation instruments, there are numerous ways to manage this link. The CI execution phase can produce data in several formats, such as CSV, UCDB, XML, and JSON, utilizing the APIs of the chosen functional verification tools. These diverse formats allow for visualization through various techniques. While some plugins within the CI tool can aid in visualization, there are also specialized tools tailored for visualizing these specific formats. In subsequent sections, we will delve deeper into the implementation details.

The connection between the CI tool and the run process, encompassing the runs for static, formal, and simulation, is shown in Figure 3. This communication originates from the CI tool, activating the makefile. This makefile refers to the tool configuration, offering multiple targets for tool execution, and can also specify targets for regression runs. Our communication approach between the CI tool and the run process hinges on triggering these "make" targets. This method ensures that the infrastructure remains scalable and deployment-friendly.


Run Process
CI Tool
Makefile
• References tool configuration files
• References design configuration files
• Targets for tool single runs
• Targets for regression runs
CI Wrapper Script
• Run tool script
• Extract pass/fail status (UCDB)
• Extract trendable data (CSV)
Tool Run Scripts
• User-define tasks to run
Run managers
• Configuration files for RM (i.e. RMDB)
Figure 3: Communication between the CI system and Run process

Furthermore, we've incorporated a scripting layer within the make targets. This layer invokes the runs, employing the tool APIs to generate varied data and conduct analyses, ultimately producing result metrics in a pass/fail format. Importantly, this scripting layer is versatile; it can manage configurations based on user-specified TCL scripts or even those associated with tool execution through regression managers for enhanced control over regression runs.

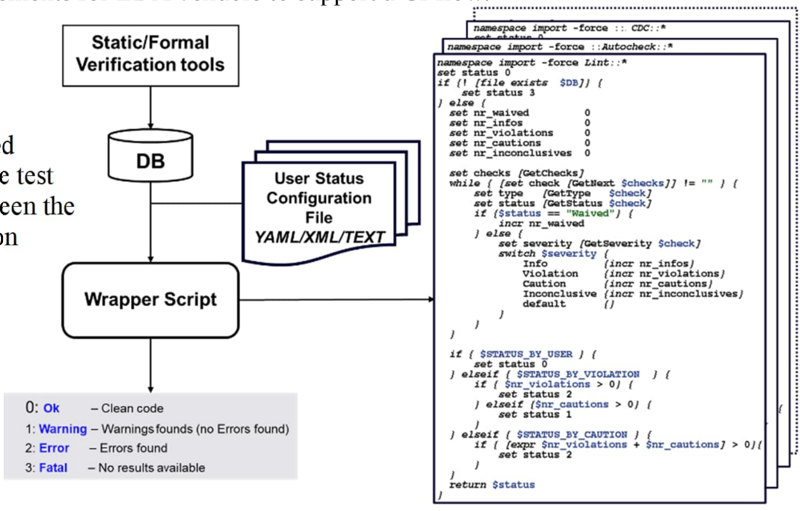
In summary, the infrastructure proposed provides a flow that can be easily and seamlessly integrated with the existing configuration and flows in hardware design. Most of the tool users are using the Makefile

IV. Requirements for EDA Vendor Tools to Support CI

We will discuss the most important requirements for EDA vendors to support a CI flow.

A. *Pass/Fail Status API mechanism*

In this work, we introduced a CI wrapper that utilizes the functional verification tools to generate data and track the status and this is called "wrapper script". In the CI integration system, the test status is crucial for exchanging information between the CI modules. The wrapper facilitates the generation of the test status, which can take on four states: OK, Warning, Error, and Fatal. These states are configurable. The user can easily configure the status generation using the wrapper script and implement a checker list for verification. The checker list will depend on which CI loop the verification runs at and the level of verification.


Static/Formal Verification tools
DB
User Status Configuration File YAML/XML/TEXT
Wrapper Script
0: Ok – Clean code
1: Warning – Warnings founds (no Errors found)
2: Error – Errors found
3: Fatal – No results available

```
namespace import -force :: CDC::*
namespace import -force ::Autocheck::*
namespace import -force Lint::*
set status 0
if {! [file exists  $DB]} {
    set status 3
} else {
  set nr_waived          0
  set nr_infos           0
  set nr_violations      0
  set nr_cautions        0
  set nr_inconclusives   0

  set checks [GetChecks]
  while { [set check [GetNext $checks]] != "" } {
      set type   [GetType   $check]
      set status [GetStatus $check]
      if {$status == "Waived"} {
          incr nr_waived
      } else {
          set severity [GetSeverity $check]
          switch $severity {
              Info         {incr nr_infos}
              Violation    {incr nr_violations}
              Caution      {incr nr_cautions}
              Inconclusive {incr nr_inconclusives}
              default      {}
          }
      }
  }

  if { $STATUS_BY_USER } {
      set status 0
  } elseif { $STATUS_BY_VIOLATION  } {
      if { $nr_violations > 0} {
          set status 2
      } elseif {$nr_cautions > 0} {
          set status 1
      }
  } elseif { $STATUS_BY_CAUTION } {
      if { [expr $nr_violations + $nr_cautions] > 0}{
          set status 2
      }
  }
  return $status
}
```

Figure 4: API Mechanism

The CI tool will be in charge of starting the run/execution process for the verification. To make this step easy to implement and handle, it is highly recommended to have a common way/script to run all the verification tools. Moreover, the pass-fail status of the verification tool run should be provided simply and easily. The status is used by the pipeline to decide on the next actions. Tools have also to provide a simple interface to configure the pass-fail mechanism, this will add more flexibility to the CI flow.

As shown in Figure 4, the API mechanism takes the databases (DB) from the functional verification runs, and under the hood, it can run the APIs on it to extract the needed metrics/data and it also can produce these metrics in different formats that are suitable to different usages (i.e., visualization, commit gating, track the quality of the code, etc.). Hence, tools used in CI should be capable of having a unified way to configure the analysis and generate data.

B. Common Output Data Format to be Processed by CI Tools

It is essential to have a standardized way of outputting data in a CI system because it integrates multiple tools, each producing different results. Collecting these results in one form makes analysis and decision-making more straightforward and identifiable. The CSV format is a suitable way to exchange data, as it is simple and can be integrated with various tools. On the other hand, UCDB is more integrated with simulation results, as it is coverage-based and uses the same syntax to generate results, making it easy to integrate with different tools. Therefore, tools used in the CI system should be capable of generating both UCDB and CSV formats for the results.

C. Fine Granularity of Tool Configurations

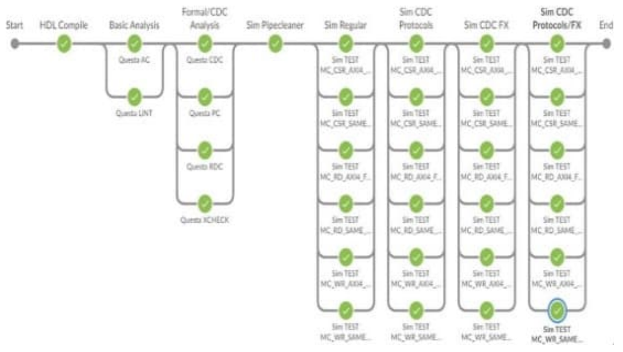
- To have a configurable and scalable CI system, tools should support different modes:
- Light mode: Tools should be able to run in a mode that can execute a specific set of checks which should be faster. This supports the CI unit testing loop.
- Full mode: To run the tools in the full functional mode and this could be used in the regression analysis.
- User mode: Tools should have the configurability to have different run options in different situations so it's flexible to be configured using the CI system.

V. Configurability, Adaptability, and Tightness

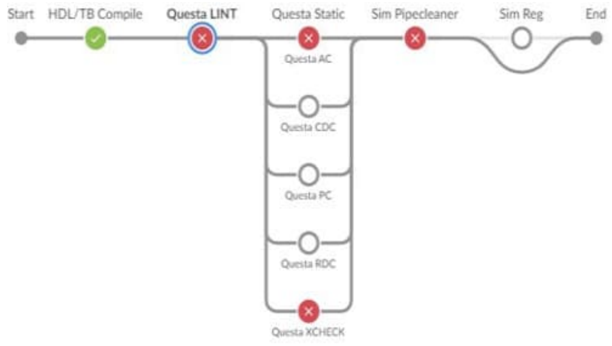
In the previous chapter we went through the framework steps, and this was going through starting from the building loop with unit testing, to the system level with bigger loops. So, this means that the CI system configuration must contain the configurability features.

Figure 5 illustrates an example of various configurations for the same CI pipeline. In Figure 4 (a), the full pipeline for a complete system regression, including functional verification and simulation, is displayed. However, as there are different levels of CI operation depending on the purpose of running the CI pipeline and the user, we require more configurability levels, such as a light mode that runs for a shorter time to provide instant feedback. This increased degree of freedom enables us to adapt the analysis according to the purpose. For instance, if a verification engineer adds a test case and wishes to examine its impact only, they can run the pipeline in light mode, as demonstrated in Figure 5 (c), and receive the results promptly, without having to go through the entire regression mode. This not only saves engineers' time but also conserves resources.

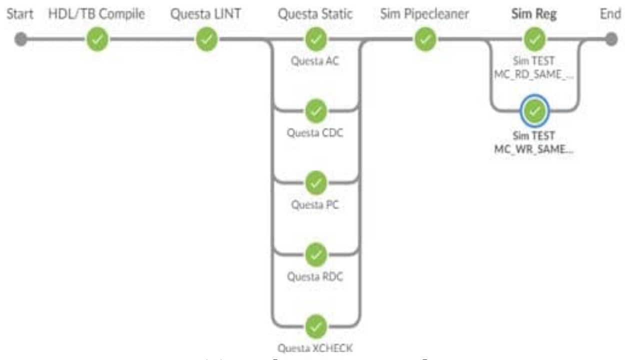
Apart from configurability, our CI system also boasts adaptability, which can be achieved in various ways. For instance, the pipeline can skip certain stages based on the results of other stages, as depicted in Figure 5 (b). This means that in case of a failure, the pipeline can avoid unnecessary steps, saving time and effort in debugging. Furthermore, adaptability can be implemented by passing artifacts through different runs, allowing the CI system to conduct a comparative analysis of the results. Based on this analysis, some stages can be skipped while others can be run specifically to obtain more information about the differences.



(a) Full system mode



(b) Skip based on results



(c) Light system mode

Figure 5: CI Configuration modes

To fully support configurability and adaptability the run process (Functional verification and simulation tools) should support the light modes and be able to adapt to different configurations based on the state.

VI. Experiment

In this section, we will utilize the CI flow on AXI4-Lite from a family of ARM® AMBA® AXI control interfaces to APB4 [4] bridge verification and simulation. We can utilize the discussed framework approach for CI integration for a way of communication system as shown in Figure 3. The experiment was done on the design to follow the given approach of using CI in the hardware design, so it was done on different levels from block level to system level. The AXI4-Lite to APB4 bridge has multiple blocks that can be considered as a CI loop (Unit-testing/Sub-system Level) and for each loop seamlessly we used the CI system and collated all these results in UCDB files and then plotted and visualized using any CI platforms (i.e., Questa VIQ [6], Jenkins, etc.).

As depicted in Figure 6, the graphic illustrates the primary module of the block diagram, enabling us to segment the CI analysis based on these individual blocks. Before delving into the specifics of defining the principal CI loop, it's essential to understand the tools used in our approach.

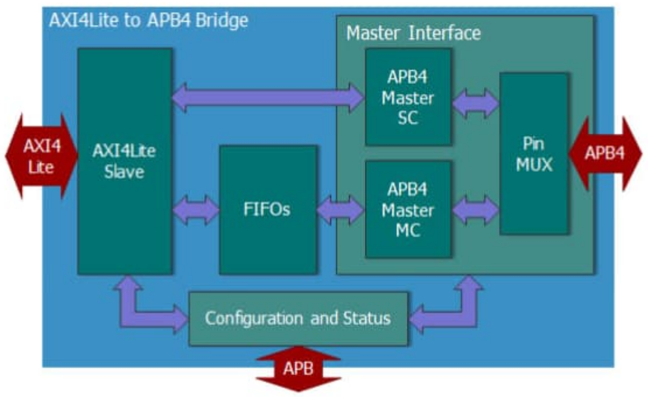
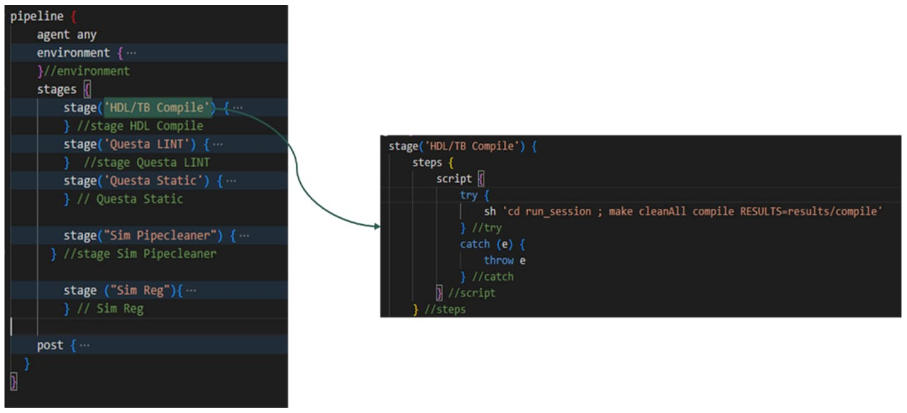


Figure 6: Block diagram for AXI4Lite to APB bridge

A. *Source Control*

We utilized Git as our source code manager, deploying it through an in-house GitLab setup. Our choice was influenced by Git's widespread popularity and its plethora of plugins, which facilitated the implementation of webhooks and gated commits techniques in our experiment. For our CI platform, we selected Jenkins, hosted on an in-house server to enhance security and maintain control. Jenkins' extensive plugin support ensured smooth integrations throughout our process. In the grander scheme, the CI tool acts as the central brain, regulating executions and communications, making its stability paramount. Choosing an apt CI tool hinges on several factors including scalability, user-friendliness, customization capabilities, feedback efficiency, orchestration, and cost considerations. Assessing our options, Jenkins emerged as the ideal choice for our needs. Consequently, our primary objective was to develop a pipeline project to generate diverse pipeline versions, catering to various CI loops as determined by the user.

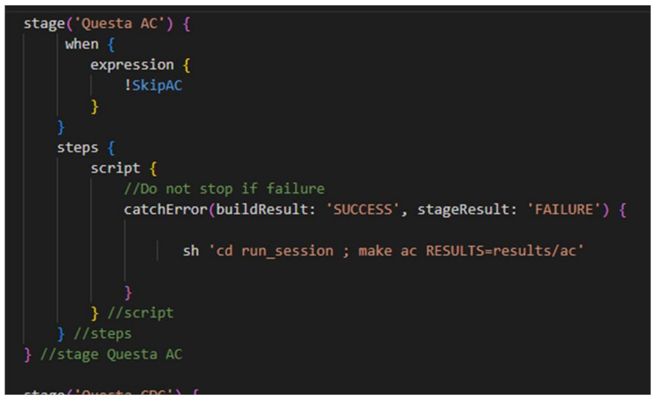


```
pipeline {
    agent any
    environment {…
    }//environment
    stages {
        stage('HDL/TB Compile') {…
        } //stage HDL Compile
        stage('Questa LINT') {…
        }  //stage Questa LINT
        stage('Questa Static') {…
        } // Questa Static

        stage("Sim Pipecleaner") {…
      } //stage Sim Pipecleaner

        stage ("Sim Reg"){…
        } // Sim Reg

    post {…
  }
}
```

```
stage('HDL/TB Compile') {
    steps {
        script {
            try {
                sh 'cd run_session ; make cleanAll compile RESULTS=results/compile'
            } //try
            catch (e) {
                throw e
            } //catch
        } //script
    } //steps
```

Figure 7: Jenkins Pipeline file and make targets

Looking at Jenkins, we chose to implement our flow using Jenkins Pipeline project, so we used Jenkins Pipeline with Groovy syntax. This pipeline file delineates our procedural directives and orchestrates interactions between various components in the CI system. Recognizing that verification and design engineers often manage their scripts and make targets, our goal was to minimize reliance on the Jenkins pipeline file. As such, our approach, depicted in Figure 3 and actualized in Figure 7, was to primarily invoke the make targets via the pipeline file, awaiting feedback to ascertain a pass or fail status. The pipeline file also played a vital role in behavior modulation. As previously highlighted, we have two operational modes: "Light Mode" and "Full Mode". The decision of which mode to initiate is based on specific conditions. Within our pipeline file, we adopted a mechanism utilizing certain variables. Depending on analysis outcomes, these variables are adjusted, determining whether tools operate in "Full Mode" or "Light Mode".



```
stage('Questa AC') {
    when {
        expression {
            !SkipAC
        }
    }
    steps {
        script {
            //Do not stop if failure
            catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {

                sh 'cd run_session ; make ac RESULTS=results/ac'

            }
        } //script
    } //steps
} //stage Questa AC
```

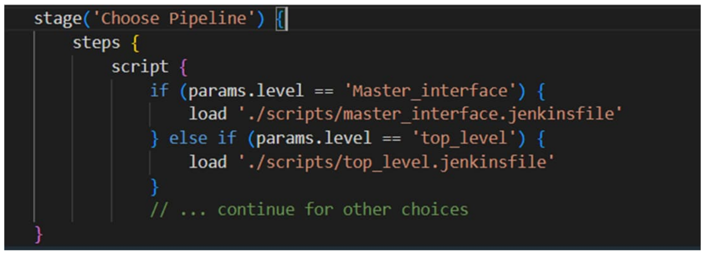
Figure 8: Conditional runs in Pipeline-file

This flexibility in tool operation, influenced by variable conditions, can be visualized as conditional execution in the pipeline, further illustrated in Figure 8.

We aim to enhance our current system to make it dependent on specific CI loops or, more aptly, to be dependent on different design levels. This is achieved by merging the "Multi Branch Pipeline Project" methodology with parameterized runs in the Jenkins Pipeline file. Put simply, if a digital designer is working on the RTL for the "Master Interface", they would initiate a branch, proceed with their development, and upon completion, would want to review the results.

B. Configurable runs

Given this scenario, there are multiple versions of the primary pipeline file, each tailored for distinct design levels. When committing changes, git hooks can prompt the designer about the specific design block they've been working on. Based on their response, the appropriate Jenkins pipeline file, tailored to that particular design level, will be sourced. This efficient setup ensures that if a user's work is confined to a branch and hasn't yet been merged to the main branch, analysis can be performed at the subsystem level. This means there's no need for time-consuming formal analysis on the top module, which might not yield the results we're looking for. Instead, we can channel our efforts into a focused formal analysis of just that block, resulting in a faster process and more accurate outcomes. This efficiency isn't just limited to formal analyses but extends to static verification and simulation as well. Figure 9 illustrates this concept, depicting conditional execution based on the specific design block or, equivalently, the particular design level being worked on.



```
stage('Choose Pipeline') {
    steps {
        script {
            if (params.level == 'Master_interface') {
                load './scripts/master_interface.jenkinsfile'
            } else if (params.level == 'top_level') {
                load './scripts/top_level.jenkinsfile'
            }
            // ... continue for other choices
    }
```

Figure 9: Conditional execution of different pipelinefiles

C. Smart Analysis

In this design, we conducted experiments to explore advanced CI functionalities. By intelligently identifying which module in the design was modified and cross-referencing it with our list of design files, we could ascertain that, for certain modules, it's possible to bypass the CDC analysis if the changes are exclusive to that module. This technique considerably reduces the time required and accelerates the CI analysis, especially for preliminary runs. Algorithm 2 shows a pseudocode illustrating the implementation of such a feature.

ALGORITHM 2: SMART ANALYSIS USING DESIGN AWARENESS

```
   Input: DesignerCode (The new code provided by the designer)
   Output: Which pipeline stages will be skipped or run
1  Begin
2     IF changed(src) is in lint_skip_defined(sources) THEN
3         Set the variable with Lint Skip
4     Else IF changed(src) is in CDC_skip_defined(sources) THEN
5          Set the variable with CDC Skip …
5     End IF
6  End
```

Also, in the smart analysis features we can define some policies to define the analysis based on which design level we are on as discussed before. For this design, we could add some policies to precisely verify our design modules. For FIFO RTL, if you are on the module design level we utilized our QFL[1] with Questa PropCheck to use the verification assertions IP to have an exit criterion for this module design level. This policy can be applied using the naming convention of the submitted RTL module (branch) if it contains any info about a FIFO implementation. Also, if we are on some defined sub-module level our exit criteria can be extended to add CDC analysis to verify crossing in this submodule. Also, if we are working on the top module then we have a strict set of analyses as this can present the work of integration so at this level, we utilize most of our static, formal, and simulation verification tools.

D. Archive the results

After exploring the CI tool and demonstrating how to utilize the system for intricate analysis steps while connecting the architecture, our focus now shifts to elucidating the concept of preserving our work. Various methods exist for archiving data, and in this section, we will elucidate how to select the most suitable approach for this task. Several factors come into play when determining the technique for archiving data. Beginning with simplicity, using Groovy scripts within the Jenkins framework is a straightforward method for data preservation. Alternatively, more sophisticated techniques like SQL databases and other forms of databases can be employed, but they inherently introduce a level of complexity. This brings us to the second criterion—considering the nature of the data. Jenkins and simpler techniques are apt for archiving items such as logs, test reports, and binaries, while more intricate methods

[1] Questa Formal Library is a set of comprehensive protocol assertions that allow Questa Formal users to exhaustively prove design correctness

are better suited for data with structured, complex relationships. Additional criteria for consideration include ease of integration with the CI platform, performance, and scalability. In our experiment, data archived from static and formal tools primarily took the form of binary databases (DBs), logs, UCDB, and CSV. For our experiment, we opted to utilize the straightforward Jenkins technique for archiving our data. The size of these databases was relatively small, as we will illustrate in the results, mitigating potential memory issues. Furthermore, utilizing these databases enabled us to efficiently retrieve logs, reports, and files representing sophisticated metrics (i.e., UCDBs and CSV), optimizing storage usage. This approach simplifies the archival process, requiring only the preservation of the databases for static and formal tool analyses, with the ability to regenerate and reproduce the necessary data.

The archives prove invaluable in performing diverse and intricate tasks. In our experiment, we utilized the archives to distinguish between two builds – the one that failed and the last one that succeeded. This technique enables us to generate reports illustrating these distinctions. Such an approach significantly expedites the debugging process and enhances overall productivity.

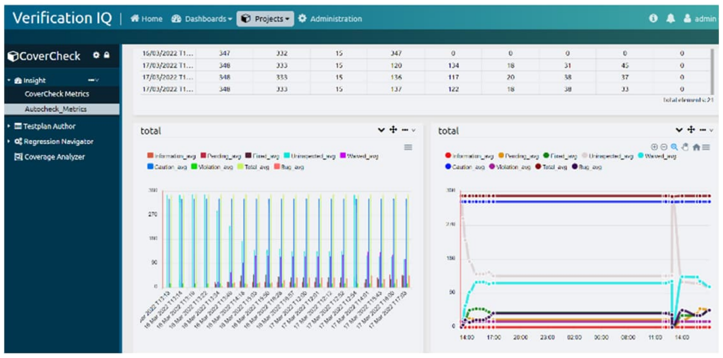
E. Visualization and Notification

Leveraging Jenkins as our platform, we successfully integrated diverse methods for visualizing data and metrics. Utilizing Jenkins plot functionality, we trended essential metrics from various static, formal, and simulation tools, as depicted in Figure 10. The proposed architecture allowed us to generate metrics in different formats, enabling the creation of a prototype for interactive visualization through Questa Verification IQ plugins. interactive visualization through Questa Verification IQ plugins. Figure 10 shows multiple and different ways to show the metrics using the CI system proposed in this paper.

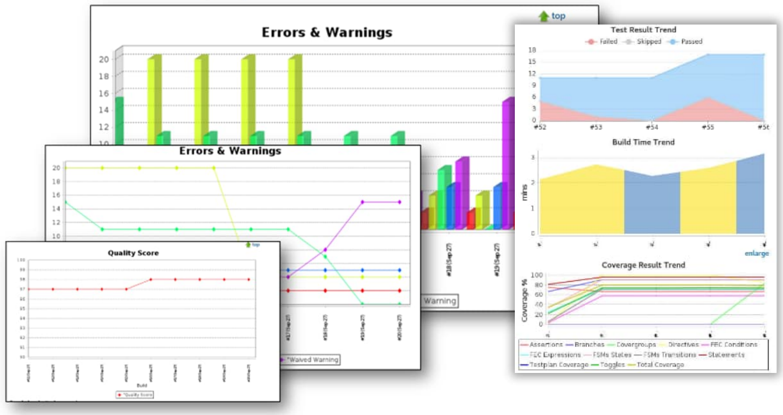
Having a good system with good feedback requires a good notification system. So, using Jenkins and Jenkins pipeline file we were able to generate an email notification system with the high-level details of the Jenkins job run. The email contains the main resolution of the build for each tool and each analysis step. The email can contain also the required logs if there is a failure that could help the Design and Verification engineer to accelerate their work and increase productivity.

F. Using Verification Run Manager

As discussed in the CI infrastructure, we were able to conduct the same experiment with the same configuration using Questa Verification Run Manager and we were able to use RMDB[2] to run our make targets and achieve the same infrastructure proposed before. The aim of using VRM was to orchestrate the parallel runs in our flow and using this layer we were able to achieve the same results but with better parallel orchestration.



a) Questa Verification IQ Dashboard



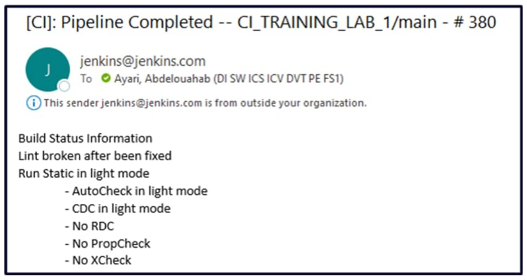
b) Jenkins Plots Dashboard

Figure 10: Visualization of the CI results

In summary, in this experiment using the different methods and techniques proposed we were to create a full CI system that can suit the design nature itself and create a re-usable and easy integration environment. In the following section, we will show how such an architecture was able to help Design and Verification engineers accelerate and increase productivity.

VII. RESULTS & CONCLUSION

Using the above CI flow, we were able to parallelize the process and reduce the analysis time with the same configuration to achieve a 2x reduction in total time. Working up from the block level towards the top



[CI]: Pipeline Completed -- CI_TRAINING_LAB_1/main - # 380

jenkins@jenkins.com
To Ayari, Abdelouahab (DI SW ICS ICV DVT PE FS1)
This sender jenkins@jenkins.com is from outside your organization.

Build Status Information
Lint broken after been fixed
Run Static in light mode
- AutoCheck in light mode
- CDC in light mode
- No RDC
- No PropCheck
- No XCheck

[2] RMDB is a run manager configuration file that is used by Questa VRM.

also saves the hardware verification engineers from “integration hell”. In the paper, we will share specific case studies from our end-users in the context of the AMBA DUT example.

Figure 11: Email Notification

The paper presents a CI architecture implementation aimed at enhancing hardware functional verification using static, formal, and simulation tools. The primary goal is to achieve close integration and supply essential metrics for verification tracking. This verification process can be applied at various points in the digital design flow, from the initial to the sign-off stages. An optimized functional verification flow allows for the early detection of bugs, reducing the need for later adjustments. With our proposed architecture, we successfully incorporate multiple tools into a single pipeline, including Lint, CDC, RDC, CDC Protocol Verification, CDC Simulate Effects[3], Sim CDC Protocols/FX, and Functional simulation. Achieving such an integrated process is complex, yet it allows for a seamless verification procedure and offers a visual representation of each step.

In the previous chapter, we introduced an experiment let’s look closer at the results as shown in Table 1

Design/Run Step	**Compile stage time (s)**	**Lint time with compile (s)**	**Lint time without recompiling(s)**	**Lint Stage (Light Mode) (s)**	**Formal/CDC Stage time (s)**	**Sim regular (s)**
AXI4Lite-to-APB bridge (Without VRM)	3	13	10	8	42	20
AXI4Lite-to-APB bridge (With VRM)	3	13	10	8	42	20

Table 1: Execution time example

looking at these results we can see that using the conditional runs as discussed in our example can save the designer and verification engineer a lot of analysis time. Let’s take into consideration for example a failure in the Lint stage in a particular part, in this case, we won’t need to go through all these analysis steps instead we can skip some of the formal verification which saves us in this experiment about 23s which is about 25% analysis time. This percentage can be huge in big designs that take longer analysis time so with small and smart tricks we can save a lot of analysis time and increase productivity. Also, we didn’t find any change in run-time between running using Questa VRM and without using it. This gives us the intuition of the integrability between Questa VRM and the Questa Static and formal solutions.

Despite the numerous stages in the pipeline and the intricate analysis, the architecture introduced simplifications that promoted ease of reuse and expedited the implementation phase. The suggested Jenkins pipeline file transforms the runs into make targets, executes the wrapper TCL scripts, and awaits the generation of the status. This level of abstraction simplifies deployment, as changes can be made to the target or run scripts without the need to overhaul the pipeline file. Such a framework has facilitated the architecture's expansion across various projects and streamlined deployment in our customers' environments. This resulted in the ease of choosing which stage to run and when as shown before so if we are skipping CDC for some design parts because of a clean Lint this will save a lot of time up to 50% of the analysis time.

The system was able to communicate with different version control systems seamlessly as we chose to do the deployment using Jenkins as a CI tool/platform, so we were able to integrate with Git easily. Given these numbers we can easily integrate the commit gating techniques as the feedback is smart and fast so with just a commit you can get your feedback analysis and the system can determine if this commit can pass or not.

Through the workflows and experiments conducted, the CI system demonstrated the capability of executing parallel runs utilizing various methods. One is employing the Questa Verification Run Manager flow, management becomes straightforward using the VRM, after we define the RMDB file for such parallel operations. Also, a Jenkins resource pool facilitates this parallelism. With the Jenkins pipeline file, resources can be easily allocated for specific stages, aiding in efficient parallel execution and time-saving. For instance, running all tools sequentially for the AXI4Lite to APB4 bridge example leads to 95s execution time. However, using parallel execution in the pipeline reduces this duration to 48s which means ~2x of reduction in the analysis time which can accelerate the analysis significantly. Such strategies not only save considerable analysis time but also ensure rapid feedback for digital design/verification engineers. This leads to enhanced productivity and early bug detection in the design phase.

From the previously discussed experiment, it's evident that Jenkins artifacts can be conveniently used to archive analysis databases. In our trial using Questa Static and Formal solutions, we successfully archived the tool-generated DBs. As illustrated in Table

Tool Used	**Questa Lint**	**Questa CDC**	**Questa Autocheck**
Database Size	729 KB	521 KB	560 KB

Table 2: DBs sizes

[3] Questa CDC Simulate Effects: This is for metastability injection.

2, these databases are merely several megabytes in size, yet they allow us to retrieve all the debugging data efficiently and easily. The relatively small size of the databases provides us with the capability to efficiently archive results across various runs and long time frames. This not only allows for archiving more databases through different runs but also enables the maintenance of a larger history of databases. This, in turn, provides designers and verification engineers with better insights into the performance of their designs. Additionally, the small-sized databases facilitate the seamless execution of tools' APIs to generate reports and retrieve debugging details. This eliminates the necessity of archiving these reports and debugging details, resulting in storage savings and efficient achievement of debugging goals on a larger scale. Moreover, this CI flow readily supports the integration of more sophisticated database techniques, including SQL, MySQL databases, and so on.

Utilizing the CI infrastructure and the APIs, we generated various types of trendable data. This facilitated the creation of multiple visualization and trending data solutions. One such solution was realized through Jenkins plugins, as demonstrated in the Jenkins Pipeline file. Furthermore, with Questa VRM, we effortlessly employed the plugin supported by Jenkins to produce trends using UCDB files. The versatility of the CI infrastructure didn't just allow integration via Jenkins plugins; its capability to generate metrics in diverse formats streamlined our integration process. This allowed us to prototype with Questa Verification IQ and craft an engaging visual representation of the required metrics for our static, formal, and simulation results, as shown in Figure 11 this shows an email with the abstract details of the Jenkins run that can easily help the user to understand the results of running the job and give him a starting point for debugging. Such trends and visualizations can help the verification/design engineers build an insight into the code quality.

The introduced CI architecture marks a significant leap in hardware functional verification. Through its efficient parallelization, seamless tool integration, and adept data archiving via Jenkins, we've realized considerable time savings and enhanced analysis precision. The system's adaptability, evident in its diverse metric generation and intuitive visualizations, streamlines processes for verification and design engineers. Coupled with real-time feedback mechanisms, this architecture ensures swift and informed responses to issues, underscoring its transformative potential in the realm of digital design verification.

This work also has been approved in a customer environment. Through this work, we have been working closely with the customer and thanks to their work they were able to help us to approve these results and ideas on a larger industrial scale. They were able to have the same architecture on their site which showed them a great value for accelerating the analysis time and increasing the productivity of the engineers. This also approved the scalability of this architecture as our experiment was on the medium-to-small design they were able to do the same on a larger scale with larger teams and a more complex environment.

Acknowledgment

References

[1] Wilson Research Group, 2020 Wilson Research Group functional verification study: IC/ASIC functional verification trend report, Mentor, A Siemens Business, 2020.

[2] Engblom, J. Continuous Integration for Embedded Systems using Simulation. Wind River, Kista, SwedenJ.

[3] Duvall, Paul M., Steve Matyas, and Andrew Glover. Continuous Integration: Improving Software Quality and Reducing Risk.

[4] ARM Ltd., "AMBA AXI and ACE Protocol Specification," ARM Limited, 2013.

[5] Dickol John. "Advanced Usage Models for Continuous Integration in Verification Environments." Samsung, DVCon US, 2015

[6] Questa Verification IQ, user manual