An Innovative Methodology for RTL and Verification IP Sharing Between Two Projects

Albert Xu, Intel, albert.xu@intel.com
Joonyoung Kim, Intel, joonyoung.kim@intel.com

ABSTRACT

This paper will describe a methodology for sharing RTL (**R**egister **T**ransfer **L**evel) databases and verification components between different product designs.

Today's silicon product designs are complex systems composed of multiple components (IP). Due to market demands for shorter product design cycles as well as market-segment specific design requirements, it is not practical for every product development team to independently design and validate all of the IP required for a product design. Therefore sharing RTL and verification IP is a necessity. In many cases, the IP blocks themselves are being developed in parallel with the product designs and sharing these dynamically changing IP blocks between multiple concurrent product design efforts is a challenge. This paper describes an innovative methodology to meet this challenge. This methodology consists of two different working models applied to two phases respectively during the design production. The first working model applied during the early phase design work, due to heavy changes including architecture and feature changes. The second model applied in the later phase of design work when the changes were more localized within IP and the requirement of turnaround time gained heavy weight.

INTRODUCTION

The Front End (FE) system data are normally collected and stored in a **R**egister **T**ransfer **L**evel (RTL) database. Each team maintains its own database with a handful of IPs shared or co-developed between them.

As demonstrated in Figure 1, databases DBX and DBY belong to two different design teams. Only IP blocks blk2/blkC and blk4/blkD are shared between the two projects. We chose a *commercially available* distributed revision control product (SCCS[1]) which has the capability to split the entire database into components such that each component is a standalone database itself. It can share (i.e. merge and keep the history) the data at any component level across the products. The "blk" in Figure 1 is equivalent to the component in SCCS system.

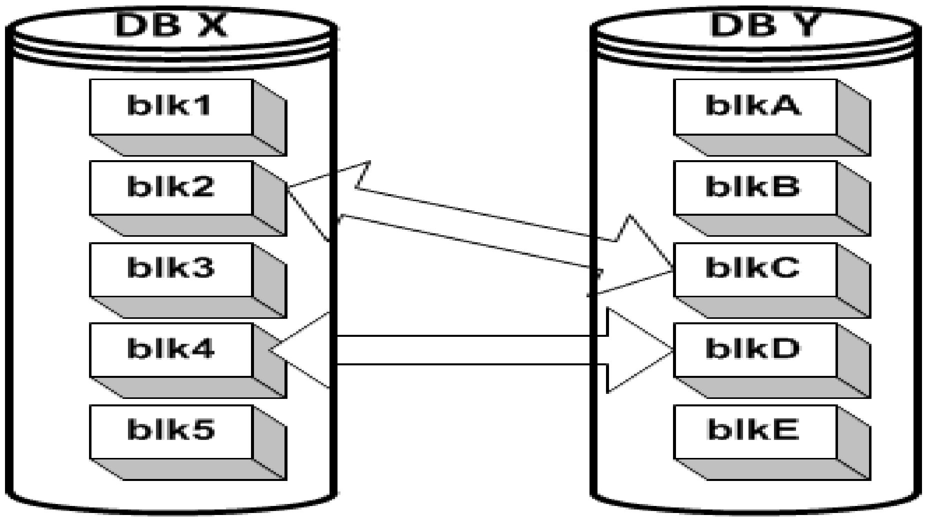

Fig.1 Partial data sharing between two products (DBs)

[1] SCCS stands for Source Code Control System. "SCCS" is used as an "encoded" tool name throughout this paper.

In Figure 2, Product X and Y are managed by separated teams and environments. E.g., product X is in "blue" environment and Y in "green" environment. The changed component still works in "blue" environment but may not work in "green" environment after sync. It is true in vice versa. Furthermore, the product lines X and Y are dynamically changing constantly.

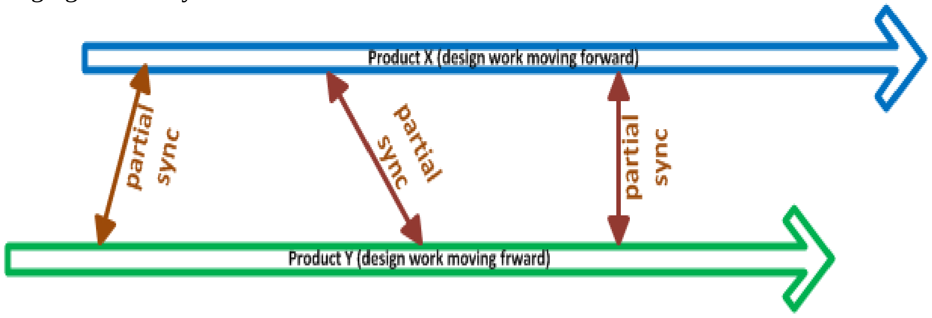
Product X (design work moving forward)
partial sync
partial sync
partial sync
Product Y (design work moving frward)

Fig.2 Partial IP sharing between two products (DBs)

In order to guarantee the data sharing does not interrupt either side of the product development process, a CI[2] tool developed at Intel was adopted to carry out the tasks. It is a software configuration management tool which was developed as part of the RTL system for the CPU microprocessor design work flow. It basically consists of a group of CI pipelines which

- Provide a multi-developer environment of rapid commits (new or changed codes) to the "golden" master repository (Product X or Y in the above example);
- Allow frequently integration going through a serious builds and regresses such that no errors can arise without developers noticing them and correcting them immediately. Therefore, the code changes from developers and migrated into the master repositories are guaranteed in a good quality;

Along with the distributed revision control system SCCS and continuous integration CI tools, we also adopted other Intel-developed tools serving the purpose of performance measuring (PM[3]) and job scheduler management (NM[4]).

PM is a performance unified measurement application. It is a system for measuring RTL (Register Transfer Level) environment performance in a unified manner among different projects. It provides a standard tool kit for tracking, analyzing, debugging and profiling performance issues for RTL model simulations/emulations, compilations, and turnins; NM is a distributed computing/batch-processing platform. It clusters thousands of compute servers and workstations and associates them with queues of jobs, priority schemes and policies. NM tool increases throughput of large computing-intensive jobs and increases utilization of computing resources. It is a "smarter NM" which manages and runs the flows.

METHODOLOGY DEFINED IN IP SHARING

As mentioned in abstract, two working models were developed for different design phases respectively. They are described in the following as methods I and II.

Master Repositories configuration (I)

Each design team carefully defines and partitions their respective master repository into components. Any shared (or common) component across the products must have the same directory structures so that SCCS and CI tools can do the port ("port" merges the contents from one component to the other while sustains the complete history) as shown in Figure 3.

[2] CI (Continuous Integration) is an "encoded" tool name in this paper.
[3] PM (Performance Measurement) is a performance measurement application developed in Intel.
[4] NM is an application to manage the large number of jobs in the batch mode.

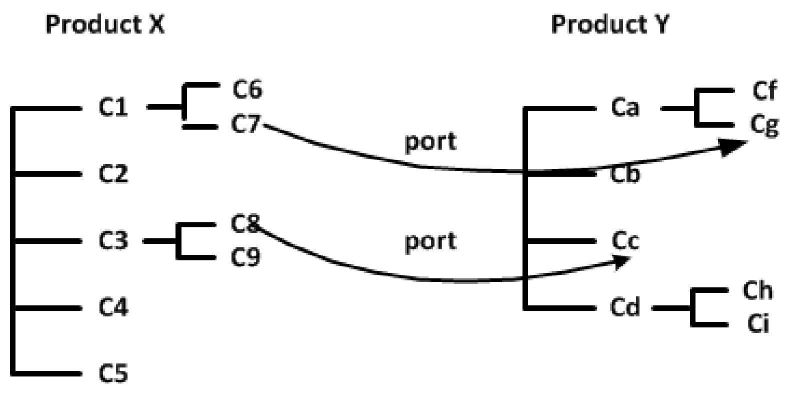

Fig.3 Simple port of component between the repositories

The above example shows Product X is the source side and Product Y the receiver side. All of the C's are components. The port operation is based on components. It seems quite simple at the first glance. However, as mentioned earlier, Product X and Y do not have exact the same environment (mainly tools, collaterals, build flow and test bench). The changes ported from Product X may not work in Product Y environment. It is not allowed to have any chance to "nuke" Product Y master repository because hundreds design engineers are working under it at the same time. Especially, in the early phase of design work, the changes in both products are very heavily involving architecture and feature changes. In other words, the big changes from one side broke the other side design work most of the time. To resolve the problem, a third master repository Product Z was introduced (Figure 4). The new repository becomes a "buffer zone" for "integration" purpose.

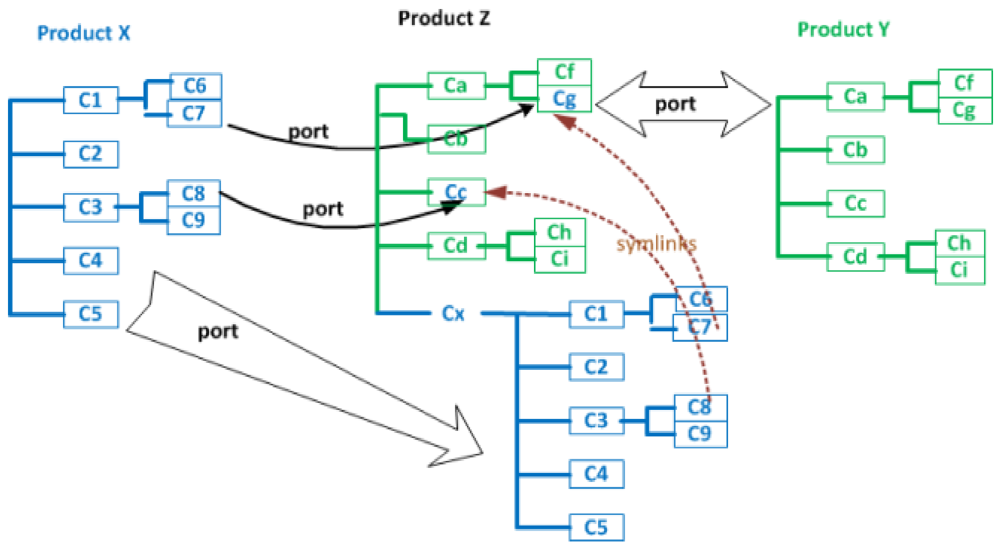

Fig.4 "Integration" repository is added in the middle of two tape-out repositories

In Product Z, trees from Product X and Y are built together. The entire tree of Product Y is ported to the upper tree of Product Z and Product X, except common components C7 and C8 in this example, is ported as Cx in Product Z. The common components are ported to the upper part of tree. The components of C7 and C8 of Cx in Product Z are the symbolic links pointing to the 'Product Y' portion of the tree. This is to make sure that all the components in the upper part tree are real entities so that Product Y doesn't get empty symlinks through "port" operation. In this configuration, the integration team builds and regresses against the 'Product X' and 'Product Y' portions separately in their own environment respectively. If an error occurs the team feeds the fix back to Product X followed by a new round of port operations. Meanwhile, the 'Product X' portion of Product Z is constantly synced with Product Y which is continually moving forward.

Master Repositories configuration (II)

The phase one configuration worked very well in the situation that both products were making big and rapid changes concurrently. The negative side is that the turnaround time is longer. The changes from one product went through two port operations and then merged into the other one.

When the development work gradually stabilized and the changes in the source side were mostly localized, in order to speed up the turnaround cycle, the “integration” repository (the middle “man”) was eliminated. The repositories configuration is similar as shown in Fig.3. This flow has only one stage port operation instead of two in the earlier phase therefore the turnaround is much shorter.

Integration flow

How do we ensure design quality of the “golden” master repository when the changes ported to the destination site? This is accomplished by “continuous integration” application “CI” developed by Phil Marden[5] at Intel. CI is a pipeline (queue) that hosts a serious of turnins. Each turnin is a complete repository containing sets of changes from individual design engineers. When the turnin is submitted to the CI, it will go through two phases: filter and integrate stages. In the filter stage, user’s repository will be merged with the master repository followed by standard builds/regresses. It continues to the second integrate stage if the first stage passes. In the second stage, user’s repository will be merged with the master repository plus all the turnins ahead in the pipeline. It then goes through builds/regresses again. If all pass, the user’s changes will be written back to the master repository. Either stage failed would result “rejected”.

With the help from CI, the ported changes are guaranteed to be healthy and the maser repository is kept as “golden” all the time.

AUTOMATION IMPLEMENTED IN IP SHARING

Automating “port” through cron

The port operation is automated by scripts using cron so that the frequency of operation can be controlled with full flexibility. Helped by other Intel internal tools like CI, NM, PM and etc., we achieved multiple goals:

- Robust and efficient sharing of IP blocks
- Ability to apply bug fixes in either product
- Assurance that the ported data is healthy
- Concurrent port process (seamless) along with the other users’ turnins
- Simple debugging and tracking of errors

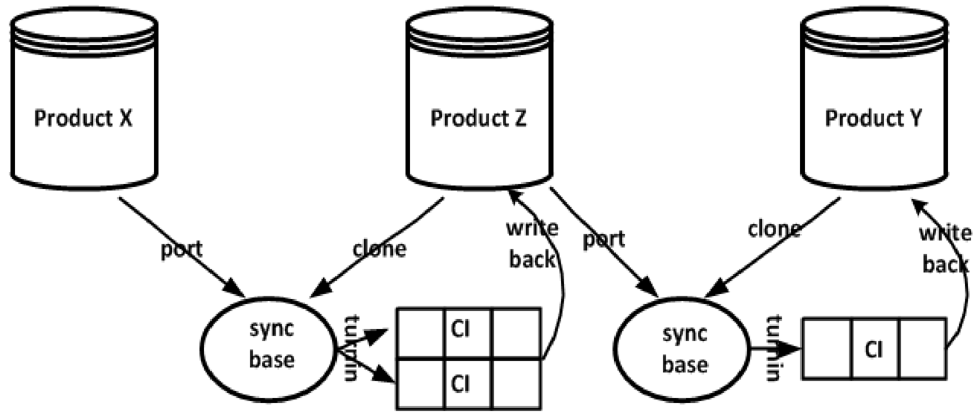

Fig.5 Automation of data port between the products

[5] Phil Marden is the application developer of “CI” in Intel.

Figure 5 demonstrated the flow in phase one working model. The port flow starts from Product X to Z. It clones the Product Z tree to the "sync base" and ports the changes in from Product X. Product Z has two sub-trees (refer to Fig.4), one from Product X and the other Product Y. From Product X, the common components are "conceptually" ported to both top and bottom sub-trees of Product Z. The rest components of Product X are ported to the bottom sub-tree of Product Z. Normally the port operation is automatically done by the scripts unless the merge conflict occurs. Once the port is completed, the scripts make the turnins to the CI pipelines which "forks" two independent turnins in parallel; one builds/regresses the top sub-tree in Product Y environment and the other the bottom sub-tree in Product X environment. The turnin will be rejected if either turnin has error occurred in build/regress; otherwise the turnin is written back to the master repository of Product Z. The rejected turnin will be debugged and fixed in Product X by design engineers, and the next round of port starts.

After the first stage port completed, the confidence of porting the upper part tree of Product Z to Product Y is very high. The scripts follow the similar process: clone the tree from Product Y, port from the upper tree of Product Z and turn in to a single CI pipeline (just like the other users) in Product Y environment. The turnin is written back to Product Y once it passes the pipeline.

You may ask why Product Z is required in this flow. Can we replace Product Y by Product Z at all? Keep in mind that Product Y is the real "tape out" database (DB) of the project. Product Z is just a "buffer zone" for integrating the common components from Product X to Product Y. There are two sub-trees (projects) in Product Z, it is hard to detect any "cross references" between the two sub-trees. We must ensure that Product Y's tree is not contaminated by collateral from Product X. Effectively we are guaranteeing isolation of the two trees, especially for the tape out product Y. Therefore, keeping Product Y with a single tape out tree is necessary.

As the design work in both products entered the later phase, the changes were more localized and the turnin successful rate got higher. In order to speed up the turnaround time, the second configuration and flow was introduced. The middle "man" Product Z was eliminated. The port automation flow is shown in Figure 6.

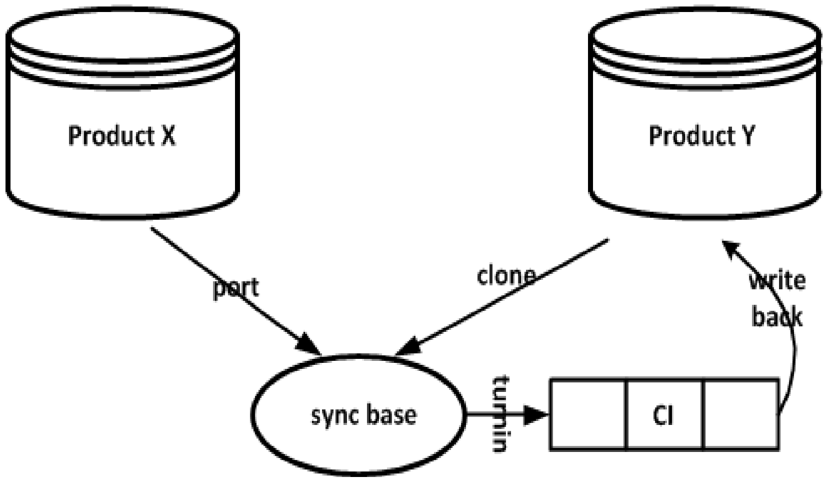

Fig.6 Automation of data port between the products in second phase

The second phase flow is very similar as the one in the first phase. The port flow starts from Product X to Product Y directly. It clones the Product Y tree to the "sync base" and ports the changes from Product X. The "sync base" has only one single tree, i.e. Product Y. If there is no merge conflict then the turnin goes to the CI pipeline in Product Y environment for build/regress. The turnin will write back to the master repository of Product Y when the process passed; otherwise, the rejected turnin will be debugged and fixed in Product X followed by another round auto-sync. This flow is simpler and has shorter turnaround time. At the same time, the design quality is very tightly controlled.

Using the same methodology and flow, the design teams can easily set two-way direction port operations to meet the IP co-development requirement.

Using tool "NM" to manage the scheduled jobs

In the automated flow, the job scheduler tool "NM" has contributed huge benefit to the port process and user turnins. It not only manages the jobs in the distributed computing system (utilizing the batch resources) and monitors all levels of execution, but also optimized the job dependencies and therefore maximally reduces the job latency. Through the GUI, users can keep track of their turnin progress, brows the log files for debugging the errors, and even collaborate with other users by looking at their turnin progress. When the first error occurs in the pipeline run, user can terminate his/her own turnin in the pipeline immediately and start the debugging. The user can also reference the same job in other users' batch jobs for debug purpose. The tool improves design engineers' productivity and reduces the load on computing batch resource.

Using performance tool "PM" to monitor and improve the efficiency

With the help from tool "PM", it is easier to identify the occurrence of performance degradation and thereby proactively eliminate the wasted jobs. The ultimate goal is to shorten the job turnaround time and improve the productivity.

During the port process, the tool monitors memory, CPU, leaking assertions and simulation stages. It tracks and analyzes RTL and validation performance. It compares the data between design projects and therefore promotes convergence across projects with enabling benchmarking and sharing by using common data models and environment tools. Data from PM dashboard provides the powerful information to the design/validation teams to improve the regress productivity and efficiency.

Values added by tools "NM" and "PM"

Both tools made a significant contribution. They 1) manage, track and analyze the batch jobs and benchmarks 2) help debug the errors and 3) shorten the job latency to improve throughputs.

RESULTS

The model we developed and built provides an efficient and robust flow to manage IP sharing or co-development between the products (or design teams) with different environment and design floor plan. We have numerous design engineers working on IP co-development. This working model saves significant effort in design and integration and contributes to overall design quality. More importantly, the sound and robust process ensure the quality of the data shared and the progress of entire design team. There is zero chance that the bad data will be introduced into the design databases (master repositories). Therefore less chance to have the production line broken. This is the big improvement of the productivity and guarantee of design quality.

SUMMARY

This methodology provided a convenient vehicle to share the FE system data (RTL codes) between the different products with different environments, especially the shared data is "scattered" within a huge tree. Generally, in IP world every single IP is an independent tree or repository. The receiver side just copies the IP contents and insert into the product and use it. However, in our scenario, the project is big and complicated therefore the repository tree is "huge". Due to the complexity and design flow reasons, each project has its own repository (or DB) independently. Our IP sharing is actually "IP co-development" in a more precise way, because we need merge capability and the codes change history on top of the contents. Moreover, our IP sharing is quite often two-way sharing, instead of normal one-way sharing. This is our definition of "IP" sharing which is also the motivation of this paper. The sharing flow described in this paper provides the convenient and robust way for frequent data sharing;

The second big challenge in our methodology is how to continuously integrate the shared IP into the current projects while the project itself is still under development and dynamically moving forward. When the shared IP comes in, it goes through the continuous integration tool ("piped" with other users) by running the standard builds/regresses. Therefore the incoming shared IP is guaranteed to be healthy. It will not corrupt the project master repository and interrupt the team ongoing design work. It also ensured the quality of microprocessor design.

On top of the above methodologies and flows, the "job scheduler" manager NM and performance monitor PM also made their contribution to the entire flow. The "job scheduler" manager tool largely helped to improve the throughput of the port flow. It also helped the users to manage, debug, and reference the jobs from port or turnins in the pipelines. The performance monitor helped the flow from the other direction. From the data it collected from monitoring memory, CPU, leaking assertions, simulation stages and turnaround time, it compares all the bench data across the projects. The analyses data helps us to pin point any performance degradation or process bloating which may be caused by some testers or new tool releases.

In conclusion, the methodology designed and implemented in this paper is robust and efficient. It helps the IP sharing (or co-development) and database integrate while ensures the data integrity of the receiving DB. It greatly improves the design productivity.

ACKNOWLEDGMENTS

We like to thank CI tool developer Philip Marden and the work groups for developing "job scheduler" flow manager, performance monitor tool and job scheduler. They provided excellent tools and flows to make this methodology successful. We also like to thank the members of Scott Bakers, Sam Huffman, Lynda Momberg and Saeid Monsef in VTM team (**V**alidation **T**ool and **M**odel build). Their contribution to the methodology defining and automation implementing is pivotal to this paper.

REFERENCES

[1] JL Gray & Gordon McGregor, "A 30 minute Project makeover Using Continuous Integration", DVCon 2012.
[2] Martin Fowler, "Continuous Integration." [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html.