Refactoring to Prepare RTL for Reuse

Steve Haynal
Strategic CAD Labs, Intel Corporation
steve.haynal@intel.com

Abstract

Preparing a design for reuse, especially one not originally written with reuse in mind, often requires changing the RTL. Ideally, these changes should be made as quickly as possible and without introducing bugs. In this paper we introduce RTL refactoring as an efficient and safe mechanism for making such required RTL changes. Finally, we present a case study of applying several RTL refactors to Intel production-level SystemVerilog RTL so that it can be reused by another team.

1 Introduction

Legacy RTL, such as found in large established companies, poses a challenge for reuse. Often this legacy RTL is not written with reuse in mind and it must be rewritten, sometimes from scratch, to enable reuse. This rewriting is necessary to enable new tool flows, better modularity, new interface protocols, reusability coding guidelines, or wider applicability. Unfortunately, the costs of manually rewriting RTL are high in terms of human effort and potential for introducing bugs.

To help address this problem, we borrow an idea from software development called *refactoring*. Refactoring is defined as

> a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. [3, 4]

In his book on refactoring [3], Martin Fowler identifies "bad smells in code", or code that would benefit from refactoring, and then catalogs refactors to address these problems. Although originally a disciplined manual technique, tools exist to automate refactoring. The popular software integrated development environment (IDE) Eclipse [5] includes automated refactoring support for Java and other languages. For Python, there is Bicycle Repair Man [1] which integrates with emacs, vi and Eclipse. Examples of some typical automated refactors include rename, remove or add member methods and variables as well as move member methods or variables up or down in class hierarchy. For RTL, typical refactors might include rename signal (through all levels of hierarchy), restructure hierarchy or replace signal with inlined logic.

This paper focuses on refactors that transform RTL to meet more stringent coding guidelines [8], although we believe RTL refactoring has more general application than just this. In our specific case, code written for an ASIC flow needed to be reused in a more manually-driven backend CPU flow. Hence we created a library of refactors to automatically transform this RTL to adhere to the more stringent CPU coding guidelines and enable IP reuse. The examples from this library described in this paper are

- **UseNode**: Redefine all `wire`, `reg`, `logic` or `net` types as a user-defined node type
- **UseCasez**: Convert all case statements to `casez`
- **IsolateDeclarations**: Split a combined `wire` declaration and assignment into separate declaration and assignment
- **IsolateFFs**: Convert all `always` block with inferred flip-flops into purely combinational blocks, with the flip-flops as separate statements

Figure 1: SystemVerilog Refactoring Flow

Successful refactoring keeps the system fully working after each small refactoring. This reduces the chances that the system can get seriously broken during the restructuring. In fact, a unit test suite is considered essential to validate that each refactoring has not introduced bugs [3, 4]. When refactoring synthesizable RTL, there is the advantage that tests can be created on the fly for each refactor. We can synthesize the original code fragment as well as the corresponding code fragment after the refactor and use logic equivalence checking (LEC) [7] to verify that both indeed implement the same functionality. It is important to note that we need not run the entire design through LEC (which may be a costly proposition) but just the portions of the design altered by the refactor.

In the remainder of this paper we provide an overview of our refactoring implementation then describe a case study where we applied several refactors. Finally, we draw conclusions and outline next steps.

2 SystemVerilog Refactoring Tool

Fig. 1 shows our SystemVerilog refactoring flow. On the left is the original Verilog file. In step 1, center and top of the figure, this Verilog is analyzed and elaborated, refactors are constructed and applied, and finally a new refactored Verilog file is generated. For analysis, elaboration and machine comprehension of Verilog, we use Verific's Verilog front-end [10]. This is distributed as a source code package which allowed us to modify and extend the front-end to better support refactoring. In particular, we added support to link every SystemVerilog statement to its elaborated netlist. This one-to-one correspondence enables generation of localized LEC tests for each refactor as discussed later. In the figure, our extended version of Verific is referred to as *Verific++*.

We chose to first implement our refactors outside of a GUI as scripts called directly from a command prompt. This follows the style of Bicycle Repair Man which integrates nicely with emacs, vi and Eclipse. Future work may follow the footsteps of Bicycle Repair Man and integrate RTL refactoring with emacs, vi or Eclipse. To enable easy scripting, we created a Python interface to Verific++ using Swig. Most of our refactoring algorithms and code are in Python. Refactors are constructed either by directly modifying Verific's abstract syntax tree via Python and calling the native pretty printer to generate a new statement or for simple refactors, directly creating a new text string. Then we use Verific's text-based design modification infrastructure to make the final textual changes to the Verilog files.

Step 2 verifies that the refactored statements implement the same functionality as the original statements. We us an internal logic equivalence checking (LEC) tool [7] for this task. Rather than check the entire original file against the entire refactored file, we are able to generate a subproblem for each refactored statement. This simplifies the LEC problem and allows us to filter which refactors to apply at the finer-grained statement level. Based on the complexity of the refactor, we choose to apply LEC either at the statement level or at the file level.

Finally, in step 3 the same refactoring script is applied again to the original Verilog but this time input also includes a list of refactors which passed LEC. A new refactored file, with only verified refactors, is generated as final output.

3 SystemVerilog Refactoring Case Study

We worked with an Intel group that is writing soft IP for PCIe Gen3 as well as other internal interface standards. This IP first targets chipsets, which follow a fairly traditional ASIC flow. This IP must later be used in CPUs, which follow a more manually-driven flow. Because of this manually-driven flow, there are more constrained coding guidelines [8] for CPUs to provide the "hooks" in the RTL required to manually optimize the design. This soft IP team did not want to adhere to the stricter CPU coding guidelines in their day-to-day coding, so instead we developed a refactoring library to enable automatic compliance with important CPU coding guidelines at release time. As representative of the types of refactoring we are able to do, we describe four refactors from this library.

UseCasez: The stricter CPU coding guidelines require that all case statements be `casez`. Statements `case` and `casex` are prohibited. This is to provide consistant don't care comparison in all case statement branches. Although this is a simple change to make in the RTL, it is essential to check that the new `casez` does not change the 4-value semantics that the designer intended. Our refactoring script uses LEC for each refactored case statement to ensure this. If LEC fails, which is rare, then that case statement is not refactored and the designer is notified to manually intervene.

UseNode: To provide finer control over 2-value or 4-value simulation, the stricter coding guidelines require that all signals be declared as type `node` [9] instead of `wire`, `reg`, `logic` or `net`. Unfortunately, a simple search and replace may accidently change portions of names or comments where the words `wire`, `reg`, `logic` or `net` appear. Since we have access to analyzed and elaborated RTL via Verific, we are able to only refactor true declarations of `wire`, `reg`, `logic` or `net`. For this refactor, we do apply LEC to the entire file as these refactors are numerous (many individual LEC tests would be costly) and typically do not alter the elaborated logic. Our LEC tool is able to quickly identify circuit isomorphisms in this case and hence the verification is cheap.

In cases like these, where the simulation semantics of the design are changed, it is important to take extra care. First, the LEC tool must be able to verify 2-value and 4-value logic, including at module ports. Second, to ensure complete *simulation* equivalence, the LEC step must be run twice in both 2-value and 4-value mode.

IsolateDeclarations: For style consistency with SystemVerilog, the stricter CPU coding guidelines prohibit combined wire declarations and assignments as shown below.

```
wire a = b | c;
```

Such combined declarations and assignments must be separated into a declaration and assignment.

```
wire a;
assign a = b | c;
```

As with the **UseNode** refactor, we do not create individual LEC tests for each refactor but rather verify the entire refactored file against the original.

IsolateFFs: To provide finer control over mapping of flip-flops and latches, the CPU coding guidelines prohibit inferring state. Instead, all state must be explicitly created with a MACRO or simple `always_ff` block. Below is an example which infers a flip-flop for signal `arbst` with a complex case statement at the flip-flop input.

```
always@(posedge prim_clk or negedge prim_rst_b) begin
  if (!prim_rst_b) begin
      arbst            <= stECP;
  end
  else begin
    unique casez (arbst)
      stECP   : arbst <= src[`ERR] ? stCPE : src[`CMP] ? stEPC : stECP;
      stEPC   : arbst <= src[`ERR] ? stPCE : src[`PMS] ? stECP : stEPC;
      stCPE   : arbst <= src[`CMP] ? stPEC : src[`PMS] ? stCEP : stCPE;
      stCEP   : arbst <= src[`CMP] ? stEPC : src[`ERR] ? stCPE : stCEP;
      stPEC   : arbst <= src[`PMS] ? stECP : src[`ERR] ? stPCE : stPEC;
      stPCE   : arbst <= src[`PMS] ? stCEP : src[`CMP] ? stPEC : stPCE;
      default : arbst <= stECP;
    endcase
  end
end
```

After the **IsolateFFs** refactor, this code is transformed into the following.

```
logic [2:0] arbst_d ;
always_comb begin
  arbst_d = arbst ;
  unique casez (arbst)
  stECP :
    arbst_d = (src[`ERR] ? stCPE : (src[`CMP] ? stEPC : stECP)) ;
  stEPC :
    arbst_d = (src[`ERR] ? stPCE : (src[`PMS] ? stECP : stEPC)) ;
  stCPE :
    arbst_d = (src[`CMP] ? stPEC : (src[`PMS] ? stCEP : stCPE)) ;
  stCEP :
    arbst_d = (src[`CMP] ? stEPC : (src[`ERR] ? stCPE : stCEP)) ;
  stPEC :
    arbst_d = (src[`PMS] ? stECP : (src[`ERR] ? stPCE : stPEC)) ;
  stPCE :
    arbst_d = (src[`PMS] ? stCEP : (src[`CMP] ? stPEC : stPCE)) ;
  default :
    arbst_d = stECP ;
  endcase
end
always_ff @(posedge prim_clk or negedge prim_rst_b)
  if ((!prim_rst_b))
    arbst <=  stECP ;
  else
    arbst <=  arbst_d ;
```

There are several important points to note in this refactor. First, a new signal `arbst_d` must be declared. This is declared as the same type as `arbst` and is guaranteed to have no name clash. So that we can uniformly handle whatever combinational logic is present, we leverage the sequential nature of statements within an `always` block and assign the default value to `arbst_d` as the first step. There is no chance to infer a state element given this default assignment to `arbst_d`. Second, the combinational logic from the original statement is repeated but with blocking assigns to the new signal `arbst_d`. Since these assignments appear after the initial default assignment of `arbst_d`, they will make the final governing assignment if activated. Also notice that preprocessor references such as `` `CMP `` are preserved. Care is taken to match the original SystemVerilog whenever possible in sophisticated refactors such as this. Finally, we see that the flip-flop is now separated out as a stand-alone `always_ff` block with same reset condition and can be furthered refactored to use a MACRO instantiation of a flip-flop.

With **IsolateFFs** it is especially important to verify each refactor individually to reduce the complexity of the sequential LEC problem. (Our internal tool [7] is capable of both combinational and sequential LEC.) Using this approach, we have had no capacity issues and have applied **IsolateFFs** to `always` blocks with multiple inferred flip-flops and hundreds of bits of state. Again, we apply a very conservative approach to refactoring. If the verification fails, the refactor is not applied and the human is notified to manually address any problems. Given the relatively more intrusive changes made by **IsolateFFs**, we do encounter failures that must be addressed manually, although they are the exception rather than the rule.

4 Conclusions

In this paper, we show how refactoring can be used to transform SystemVerilog and enable soft IP reuse between different design teams with different coding guidelines and tool flows. The refactoring engine is built on an extended Verific-based front-end, Python and an internal LEC tool. LEC is used to verify correctness of each refactor so as not to introduce bugs. We applied a refactoring library to production SystemVerilog and demonstrated that it produces correct and human-readable transformed SystemVerilog.

Although the focus of this paper was on preparing RTL to meet coding guidelines for reuse by a specific team, we believe RTL refactoring has more general application. For instance, RTL refactoring can be used to abstract and understand a design [6], prepare a design for other purposes such as validation or elastization [2], optimize a design for specific tools such as synthesis or to simply improve the design of existing code [3].

Our future work in RTL refactoring will push in two directions. First, we can categorize "bad smells" in RTL in similar fashion to what is done for software [3] and create a rich library of refactors to address these. Furthermore, we believe it is an easier problem to create more sophisticated yet correct refactors for synthesizable RTL than general software since the semantic model for synthesizable RTL is well-defined and LEC tools can be used to automatically generate tests. Second, we can enhance usability by integrating RTL refactoring into an IDE such as Eclipse. Given such a user interface, a designer can easily apply hundreds of refactors to RTL to evolve it in any direction he or she wishes.

References

[1] "Bicycle Repair Man, a Refactoring Tool for Python", [Online] Available `http://bicyclerepair.sourceforge.net`, September 2008.

[2] J. Cortadella, M. Kishinevsky and B. Grundmann. "Synthesis of Synchronous Elastic Architectures", *Proc. Design Automation Conference 2006*, pp. 24-28, July 2006.

[3] M. Fowler. *Refactoring: improving the design of existing code*, Addison-Wesley, Boston, Massachusetts, 1999, ISBN 0-201-48567-2.

[4] M. Fowler. "What is Refactoring?", [Online] Available `http://www.refactoring.com`, September 2008.

[5] D. Gallardo. "Refactoring for everyone", [Online] Available `http://www.ibm.com/developerworks/library/os-ecref`, September 2008.

[6] S. Haynal, et. al. "A SystemVerilog Rewriting System for RTL Abstraction with Pentium Case Study", *Proc. MEMOCODE 2008*, pp. 79-88, June 2008.

[7] D. Kaiss, S. Goldenberg, Z. Haana and Z. Khasidashvili. "Seqver: A Sequential Equivalence Verifier for Hardware Designs", *IEEE International Conference on Computer Design*, pp. 267-273, 2007.

[8] M. Keating and P. Bricaud. *Reuse methodology manual for systems-on-a-chip designs*, Kluwer Academic, Boston, Massachusetts, 2002, ISBN 978-0-387-74098-0.

[9] M. Maidement. "A SystemVerilog User Experience", *DAC2003 Accellera SystemVerilog Workshop*, [Online] Available `http://www.systemverilog.org/pdf/1b_DesignUser.pdf`, slide 81, 2003.

[10] Verific Employees. *SystemVerilog Front-end*, [Online] Available `http://www.verific.com`, September 2008.

I've found that the group configuration page in the Studio is not RTL'ed at all. So I've fixed that. This PR is a cherry-pick this from the Edraak repo (https://github.com/Edraak/edx-platform/pull/199). Such refactoring can make the composition more effective with respect to GPU resource usage especially when combined with suitable scheduling. Here we propose a methodology where developers of highly tuned kernels can enable application designers to optimize performance of the composition. Kernel developers characterize the performance of a kernel through its â€œperformance signatureâ€.Â Often, the kernel components are generic in nature and have a potential of being reused in other applica-tion contexts. An application comprising of a set of kernel components may have data dependencies, requiring that they be run in sequence, or they may be run concurrently. If you deploy an API to an IBM API Connect for IBM Cloud Management server by using the developer toolkit command line, you can use the $ref field in your OpenAPI (Swagger 2.0) YAML and JSON API definition files to reference a fragment of OpenAPI (Swagger 2.0) code that is defined in a separate file. When IBM API Connect for IBM Cloud processes the source API definition file, the $ref field is replaced with the contents of the target file. Prepares a reusable cell for reuse by the table view's delegate.Â If a UITableViewCell object is reusableâ€”that is, it has a reuse identifierâ€”this method is invoked just before the object is returned from the UITableView method dequeueReusableCell(withIdentifier:). For performance reasons, you should only reset attributes of the cell that are not related to content, for example, alpha, editing, and selection state. The table view's delegate in tableView(_:cellForRowAt:) should always reset all content when reusing a cell. If the cell object does not have an associated reuse identifier, this method is not called. If you override this method, you must be s