IP Reuse: A Novel VHDL to Verilog Translation Flow

Alessandro Fasan
STMicroelectronics, New Ventures Group, S.I.C.L., San Jose, CA, USA.
alessandro.fasan@st.com

Andrea Fedeli
STMicroelectronics, Central R&D, D.A.I.S., Agrate, Italy.
andrea.fedeli@st.com

Abstract

IP Reuse and customization are emerging topics in nowadays electronic industry. This paper reports a detailed description of the process to translate a 280,000 gate equivalent VHDL source (more than 60,000 lines of code) into the corresponding Verilog without technology mapping. This translation process was performed in less than three man-months using a custom simulation environment with formal verification of design equivalence.

Keys: *Language Translation, Simulation Environment, Formal Verification.*

Object description

Our target design was an embeddable MPEG-2 decoder. We had to make it quickly, correctly, and using our custom simulation environment. What better than an IP[1] reuse program to meet this goal? We decided to collect different blocks from divisions within our Company (fig. 1). We took the *Decode Pipeline* from one group, the *Slice Parser* from another one and the *Motion Compensation* from a third.

The Decode pipeline, already used in real projects (i.e.: silicon proven), was assumed to be correct. For the other two components the debug activity had to be completed; we also had to write the glue needed to put together all the pieces. All block descriptions had to be synthesizable, including the new code that we added. As any testbench designer knows, creating a testbench requires much effort. We were fortunate enough to have an existing testbench which used real MPEG streams: the expected results were known.

But we had a problem: all the IP was written in VHDL. Some testbenches were written in Verilog, and our simulation environment was strongly Verilog-based. Furthermore, most of the designers in the group had experience in Verilog code development (writing, testing, debugging), and few of them had knowledge of typical VHDL idiosyncrasies. Last, but not least, our VHDL simulation environment was inferior in time and flexibility performances to the Verilog one and thus less suited for the debugging phase of the project.

[1] Actually we were building an IP using other IP's.

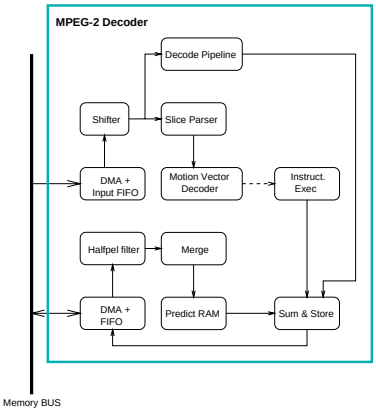


Figure 1: The MPEG-2 decoder block diagram.

Looking for a Good Translator

A possible solution could have been to adopt a co-simulation environment, but one of our project constraints was to maximize the knowledge reuse in each phase, simulation included. Thus, we had to adapt the designs to our simulation environment rather than the other way around.

Our aim was, therefore, to translate all the VHDL descriptions into corresponding Verilog sources. This could be performed in one of three ways:

1. Translate all the files by hand;
2. Pass through a synthesis step;
3. Translate the description using code translators;

Solution 1 was not viable, due to circuit dimension and source number (the whole project involved 124 files).

As far as a combination of correctness and translation times is concerned, solution 2 is the best; but one of our

aims was to obtain a Verilog description suitable for simulation on which we had to run a reasonable amount of MPEG streams. By using a gate level netlist in simulation we would have lost the time gained in the translation phase, not to mention that the debugging and the design modifications of a gate level netlist is prohibitive.

Therefore we decided to find a good translator that, by being able to convert VHDL into Verilog without any technology mapping, let us mostly reuse our wellknown Verilog simulation environment.

The Search

It took us a certain amount of time (see table 1) to find a good candidate. A translation study case, based on the Motion Compensation block, allowed us to identify the more accurate and better-supported tool. Three candidates were considered. The choice of the study case was determined by the existence of a Verilog testbench and the chance to be supported by one of the designers who worked on that block[2]. After a couple of weeks spent debugging the Verilog RTL version of the original VHDL design, 16 of the given 17 patterns were correctly passing. At that time we ended the evaluation of the translators; we chose the VHDL2Verilog from ASC[3][1],[3].

Go Marching in

Next step was to work on the Decode Pipeline; the original design, written in VHDL, contained DesignWare (DW) components. This implied huge interventions on the translated code; pragmas had to be added manually to the generated Verilog code to maintain DesignWare directives. To our knowledge no translator supports DW components yet. During this phase the *compare_design* command of Synopsys' Design Compiler was heavily used to verify the functional equivalence of original VHDL and derived Verilog.

Here comes the Formal Verification

There was still a pattern in the translated block in which the simulation was failing. Instead of debugging it in the old fashioned way, we decided to use something innovative to what we were doing: that is why we decided to try Synopsys' *Formality*, an Equivalence Checking[4] tool. We applied Formality on the Motion Compensation block: in just a week we found the reason why the 17^{th} pattern used in the regression was not passing correctly; in fact a bug in the translation process was detected. ASC acknowledged its translator bug, fixed it and provided us with the new translator release in a very short time. In this first phase we learned that Formality could help us write a synthesizable (Verilog in this case) RTL code that was better than the original (VHDL in this case): in fact we could hand modify the code where we had warnings from Design Compiler and use Formality to prove the equivalence. As explained further in the text, *Formality* uncovered other issues with the translation.

To co-simulate or not to co-simulate? That is (not) the question:

Just to get the idea about the effort needed to co-simulate, we tried to co-simulate the Decode Pipeline using an environment based on *Leapfrog*. All the tests passed without problems, but the perfomance was unacceptable for our needs. We budgeted only one week for this activity, having already understood that Formality was going to be the perfect companion to successfully complete this challenging job.

Equivalence Checking, what else?

As mentioned above, all the blocks we had to check were synthesizable. This was essential for the use of an Equivalence Checking tool as all the current[5] state-of-the-art tools in this field are able to compare only synthesizable descriptions. After the Decode Pipeline was verified, the verifications of other blocks with *Formality* highlighted at least two different classes of problems that would have been difficult to detect via simple simulation:

- Signal name swaps / semantically correct typos.
- Arithmetic function translation.

Furthermore, it is important to perform the verification at different levels in the hierarchy because in one case the designs matched at one level but encountered problems at a higher level. The equivalence checking phase was preceded by a simulation of the Verilog translated description that hung. In this case we had a simulated object, quite

[2] In fact, the Verilog testbench for a VHDL design was due to the needs of the first customer for that IP. The team that provided us the Motion Compensation block synthesized the VHDL design, and created a Verilog gate level netlist; this was used by the first customer to perform a Verilog simulation.

[3] ASC is Alternative System Concepts, Inc.

[4] That is a —relatively new— verification technique, different from usual simulation, belonging to the field of Formal Methods; in a few words, Equivalence Checking allows to perform exhaustive Verification without pattern definition and simulation.

[5] September 1998.

large and with heavy arithmetic content, that was a forest of hiding places for bugs. We decided to check it bottom up, taking advantage that Equivalence Checkers work **without** testbenches. Thus *all* blocks, even those that do not have their own testbench already set up, can be verified with a very low preparation effort. This generally speeds up verification especially when, as it was (by construction) in our case, designs have the same hierarchy.

Logic Cones and Subtle Typos

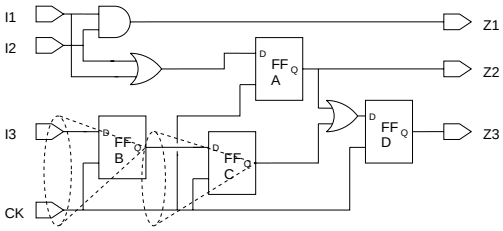
A check with the Equivalence Checker revealed immediately a difference in the input set of two *logic cones*. To make sure our reader feels comfortable with what we are talking about, let us give a very brief description of the logic cone concept. To be able to compare two designs, an Equivalence Checking tool has to establish what has to be compared. Normally Equivalence Checking tools take the primary outputs and the state (or output value) of circuit memory element (i.e.: flip-flops and latches) as comparison *relevant* points (fig. 2). Relevant points are then connected to (i.e.: their value depend on) other relevant points or to primary inputs value by globs of combinational logic (degenerated into a wire, in the simplest case). Each relevant point, as termination of its glob of combinationals, can be seen as the vertex of a sort-of cone (of logic) whose base collects the set of relevant points that with their state/value determines the value of the vertex.

When two relevant points that should match show differences between their logic cones input sets, a problem is perhaps underlying[6].

Now, in our case, some logic cones were reported to have different sets of input; the matter was due to a small set of typos (a manual intervention, here, was the origin of the problem) that acted like a swap between signals when passing from VHDL to Verilog description. The highlighted differences allowed us to focus our attention on a restricted section of code where the difference happened. Actually, that mapping had to be mediated by our circuit knowledge, as no code back-reference[7] was available. It has to be noted then, that a more subtle set of typos, not translating itself into a difference between logic cone input set could have hidden the swaps. The Equivalence Checking tool would have revealed a difference in functional dependency of those two edges, and a little deeper investigation would have also determined the problem origin in this case.

[6]This is actually a white lie: different other aspects could be taken into account, for instance, topological similarities could be used to lessen the differences; we think it suffices, for our exposition, to give the basic idea here.

[7]That is the ability to show, on the source (VHDL, Verilog, etc.) code, the origin of a certain difference.



relevant point	input set
Z1	I1, I2
Z2	FFA(Q)
Z3	FFD(Q)
FFA	I1,I2,CK
FFB	I3,CK
FFC	FFB(Q),CK
FFD	FFA(Q),FFC(Q),CK

Figure 2: A sample circuit, with dotted logic cones for two relevant points (FFB, FFC), and its relevant points input set table.

Arithmetic Shifts and Sums of Signed quantities

Another cause of differences between descriptions is the way signed arithmetic operations have to be exploited in VHDL and in Verilog. Such a problem existed in the *LIQ*, a block inside the Slice Parser, engaged in micro instruction generation for Motion Compensation Block, (cfr. fig. 1). The translator could have done more than it did, but perhaps not much more. The translation of operators like + or >> had to be patched by hand especially with operations involving arrays of different length. VHDL shift right with signed quantities were translated into unsigned Verilog shifts, e.g.:

```
PMV1 <= conv_std_logic_vector(shr
          (signed(vect),''1''),12)
```

translated into

```
PMV1 <= (vect >> 1'b1);
```

Bottom-Up verification: Asynchronous FSM interaction and Non blocking assignments drawbacks

The last block we had to verify (the *Slice Parser* in fig. 1) was the toughest; it was quite big and with several hierarchy levels, and even if each sub-block correctly passed the Equivalence Checking phase, its Verilog simulation was stuck in an infinite loop. That loop was not present in the VHDL simulation.

The origin of the lock was a zero-delay loop due to the coupling of non blocking assignments in two finite state machines, placed in different blocks. The difference between simulation behaviors, together with the validation of functional correspondence of single blocks, pointed our

Phase	Time (man/week)
Evaluation	4
Co-simulation	1
Decode Pipeline (Compare_design)	3
Decode Pipeline (Formality)	1
LIQ	1
Slice Parser	1
Total	**11**

Table 1: Timetable for described operations.

attention to a difference between the event models assumed by simulation environments, and from that to the effective cause of the loop. Our (formal) verifications checked the correctness of those two blocks each by itself. We did not take into account the effects of the interaction. The solution to our problem was quite simple: we just had to substitute non blocking assignment with blocking ones (in *that* case the substitution was safely applicable), but our experience should raise a general warning: in all those cases where blocks with feedbacks are present, an upper level verification should be performed, to be sure that unexpected interactions do not take place. Therefore particular attention should be put on Asynchronous Finite State Machines interactions, as they can be, each one already by itself, a well known race-condition source.

Conclusions

We reached our target: the translation ended successfully, and the new design was even better than the original one, because during the translation we found bugs in the starting description. As summarized also in [2], this experience taught us more than one thing:

1. VHDL to Verilog translation, without gate-level intermediate mapping is feasible on quite big designs;

2. The market is NOT offering, today, a push-button solution for HDL code translation: indeed there are tools that help engineers accomplish this task, but such tools have to be used with a good comprehension of undergoing translation process issues and limits;

3. The kind of aid that an equivalence checker tool can give is **really** relevant, in time and depth of the saved effort.

With our experience we have proven that a VHDL to Verilog translation flow, in view of *real* IP Reuse, is an affordable task.

Acknowledgments

The authors wish to thank Umberto Rossi, for his hints on Equivalence Checking and for his precious suggestions and continuous, widespread support. We would also like to thank Frank Palazzolo and Karen Poelakker for their patience in correcting this paper and for the suggestions and advices they shared with us.

A. Fasan wishes to thank Inwhan Choi for his support during Motion Compensation Block debugging and Andy Betts for his testbench code for the Decode Pipeline used in the co-simulation environment and, most of all, for his fruitful efforts in IP reuse in our Company.

References

[1] Alessandro Fasan, “A VHDL to Verilog commercial translators survey”, STMicroelectronics internal documentation, San Jose, CA, USA, May 1998.

[2] Andrea Fedeli, “The ASC VHDL2Verilog Translator, Application notes”, STMicroelectronics internal documentation, Agrate, Italy, July 1998.

[3] Jake Karrfalt (Alternative System Concepts, Inc.), Thomas Oberthür (SICAN GmbH); “IP Design Flow Centers on Automatic HDL Translation”, *Integrated System Design*, April 1998.