THE MYTHS OF CODE PORTABILITY

Chad Epifanio (Xilinx, San Jose, CA;
chad.epifanio@xilinx.com)
Manuel Uhm (Xilinx, San Jose, CA;
manuel.uhm@xilinx.com)

ABSTRACT

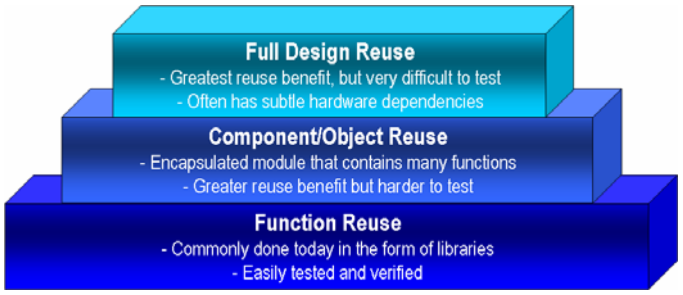
Code portability, or, more generally, code reuse, is a long-standing technique to reduce system development costs. It forms a key tenet of JTRS and other transformational defense programs. However, a number of ancillary assumptions must be met in order to reap significant cost reduction. Improper reuse may actually have the undesired effect of increasing development costs. In this paper we explore the myths and realities of code portability. We discuss the assumptions that must be true in order for reuse to succeed, and contrast that with the realities that we observe in the field. Many problems derive from the fact that the developer and reuser may reside in different, often competitive organizations. While the burdens of designing for reuse are borne by the developer, the benefits accrue to the reuser. The focus will be on FPGA code, as both the difficulty and the need for portability are arguably greater than for GPP/DSP code, though the general concepts presented are applicable to all.

1. INTRODUCTION

Code portability is a key tenant of defense transformational defense programs such as JTRS. The goal is to reduce development costs, reduce risk, and accelerate time-to-market. However, experience has shown that achieving the projected benefits is not easy, especially when dealing with FPGAs. In fact, improper code reuse can actually increase development costs. In this paper we describe some of the pitfalls in the chase for code portability.

To frame further discussion, we must first define what is meant by "portable code". In its most general sense, portable code is code that can be reused in another project. This in and of itself is not quite sufficient, as there are multiple levels of code reuse. At a basic level there is function reuse. This is fairly easy to achieve, as functions are small, coherent units that are easily tested and verified. In the context of FPGAs, we consider low level IP cores, such as FFTs or FIRs, to be equivalent to a software function in terms of reuse. Component or object reuse provides greater benefit than function reuse, in that they are aggregations of functions and perform more complex tasks. They are also more difficult to test and verify under all expected operating conditions. Porting problems can be localized fairly accurately. FPGA IP such as Forward Error Correction cores like Turbo coders or LDPC can be classified in this category. At the highest level, full design reuse provides the greatest benefit, but is also the most difficult to attain. One reason is that it is difficult to test and verify the design over all anticipated operating conditions and deployment platforms. Porting problems can be very difficult to diagnose and repair, as they may be the result of subtle interactions between components that did not appear in the original host platform. An example of a full design could be a complete application, such as a waveform. The range of code reuse is illustrated in Figure 1.

Figure 1: The range of code reuse


Full Design Reuse
- Greatest reuse benefit, but very difficult to test
- Often has subtle hardware dependencies
Component/Object Reuse
- Encapsulated module that contains many functions
- Greater reuse benefit but harder to test
Function Reuse
- Commonly done today in the form of libraries
- Easily tested and verified

Some defense programs attempt to go even further by trying to leverage full design reuse across separate companies. This is a very difficult task, for, as will be shown later, reuse even within a company is often not successful. To be clear, this paper is not an attempt to dismiss reuse as ineffective. Instead, we attempt to show that focusing solely on code reuse will not achieve the reductions in cost, risk, and schedule that are desired.

2. REALITIES OF PORTABILITY

The goal of any reuse effort should be cost reduction, where for simplicity we assume that other factors such as risk and schedule can be reflected as a type of cost. Therefore, if whatever we are doing does not result in cost reduction, then it does not help achieve this objective. One might question how reuse could not result in cost reduction, but consider this: "reusing code" is not the same as "reusable code". Reuse is not free. It takes extra effort up front to properly design code that can be reused by a different developer, perhaps even in another company. The code must be generalized and thoroughly tested over all anticipated operating conditions. It must be fully documented, and eventually supported and maintained.

Code reuse within a company makes sense, since the company will benefit in the future from extra effort spent today on developing reusable code. What is not clear is how well reuse will work across companies, especially companies who may be in open competition. The problem is that the burden is distributed unequally - the extra cost is borne by the developer, while the benefits accrue to the reuser. Where is the incentive for the developer? Developer altruism cannot be the basis of successful reuse. There must be some carrot, in the form of a profit-driven business model, or some stick, in the form of enforceable portability standards and requirements.

In order to effectively reuse code, there must be a foundation of confidence. Blind reuse with no visibility into the development process is rarely successful. The reuser must have confidence that the code not only behaves as advertised, but will behave the same on the reuser's chosen platform. The way to build this confidence is to provide the reuser with all the artifacts that went into the development of the code. These include high and low level specifications, trade studies, models, fixed-point simulations, and test vectors. The reuser must be able to verify that the requirements and use cases are the same and identify areas where they are not. The fixed-point models are needed because there are no standard word sizes in FPGAs like there are in DSP/GPP. 18-bit multipliers are the most common, but 9-bit and 25-bit are also available. Unlike DSPs, where the computational engines more or less look the same due to convergent evolution, FPGA multipliers are fairly different across devices and vendors. The test vectors must completely cover the operating space. All systems work great when there is no noise, multipath, or interference. The real test of a radio is behavior at the extreme.

Good documentation is vital to successful reuse. The level of granularity is important; each coherent functional unit in the design should have documentation and test vectors. When trying to port large designs between disparate platforms, the reuser is almost assured to encounter anomalous behavior. Without functional unit descriptions and test vectors, the reuser has no choice but to start a tedious, and expensive, reverse engineering effort. The reuser must first identify what the code currently does, then try to infer what it was intended to do. From there, the reuser must forward engineer a solution. This is where blind reuse without good documentation can actually end up being more expensive than designing it from scratch from base principles.

We want to stress at this point that the final deployed executable code cannot be used as a means of specifying system behavior. By executable code, we mean code that is intended to be deployed in the final system. The executable code is an instance of a waveform, not the definition of it, as it always has some amount of platform-specific detail. For example, it may have a software-hardware (as in FPGA) partitioning that may not be able to be supported by the reuser's platform. The ultimate definition of the waveform must be the top level specifications. Unfortunately, many military waveforms have specifications that are woefully inadequate such that two independent waveform developers cannot build two independent systems and have any hope of them actually intercommunicating. This may be due in part to the acquisition process, where development of a waveform is typically awarded to a single company, sometimes on the basis of a science project that has not been project managed to support code reuse.
Commercial telecommunications waveforms, on the other hand, are typically far more and better specified since there are multiple companies involved in the process, each of which need to be able to build equipment that can interoperate. This allows anybody to develop compliant radios and compete on the open market.

Portable code could be seen as a possible mitigating factor for the general poor quality of some waveform specifications. However, is it possible to create truly portable FPGA code? There have been some successes in the past, but it really depends on the system requirements. Device size, implementation cost, and power consumption requirements very often force the developer to use platform-specific optimizations. This is especially common in small form factor and handset radios that rely on batteries. Because of the rapid evolution of FPGA technology, and the divergence of platform-specific hardware features, development tools currently are not able to abstract the hardware as efficiently as modern C++ compilers. Optimizations have to be instantiated at a low level, akin to inline assembly programming in the software world. In these cases, the executable code is a poor vehicle for reuse.

Let us take a step back and discuss what it is we are trying to do. The key goal is to reduce development costs. By focusing on portability and reuse, we are implicitly stating that the act of coding is the most expensive part of a product development program. In our experience, this is not the case. If the specification is clear, the architecture sound, the fixed-point analysis complete, the act of coding is relatively straightforward. One must not confuse "programming" with "coding": the former is the structured logical design process that specifies the behavior of the system, while "coding" is the act of casting it into the language of choice. Most problems occur when the two are intermingled, resulting in persistent, expensive rework.

The subtle point about trying to achieve reuse across company boundaries is that reuse of knowledge, not the raw code, is often the most important factor in success. It is far more important to clearly convey the nature of the design than it is to provide the implementation of the design itself. Certainly providing the implementation along with the design helps to clarify areas of potential confusion. But we believe it would be far more effective if a reference implementation was provided instead of the final deployable executable code. A reference implementation is designed for clarity and conveyance of information, not for real-time implementation in the final platform. This is common in the commercial telecommunications sector, where a fixed-point implementation is provided along with the specification. There are many ways to put the reference design together - fixed-point C++, MATLAB m-code, Simulink, UML, etc. We have no strong opinions on the matter, other than some platform-independent reference model must be provided by the waveform developer.

Integration and testing is often an unexpectedly large fraction of the development effort, and thus the development cost. Reuse might be seen as a way to reduce the testing burden, since the original design was (presumably) proven to work correctly. This works well in the GNU open-source community, for example. There are two important differences for FPGA designs. First, general-purpose processors (GPP) are fairly similar, relatively speaking, and the tools are more mature. As we have said previous, this is not the case for FPGAs. Second, and more importantly, application software depends on an underlying operating system that abstracts much of the details of the underlying hardware. It is because of this abstraction that one can write code that will run on, say, an Intel and a Freescale processor. At present, there is no equivalent to an operating system on an FPGA, and so a means of abstracting the hardware is primitive compared to GPPs. On the other hand, it is well known that there is often a very significant level of rework needed when porting an operating system itself between processors. Why should we expect that a similar level of rework would not occur when porting waveforms on FPGAs?

In order to reduce the costs of integration and testing, we are back to stressing the need for documentation and test vectors. The best way to reduce costs is to insure that the integration/test engineers know exactly what it is the system should do at every significant test point in the system

3. KEYS TO SUCCESS

Thus far we have presented common flaws in code portability. Since portability and reuse is not a new issue, one should be able to look at past projects and extract lessons learned. Unfortunately, there are not many cases where wide scale reuse occurs between companies. What we can do is look at recent studies on code reuse within a single company. This is a far easier problem than reuse between companies, and should be viewed as an overly optimistic look at the problem. Unfortunately, failure is more common than one would wish. Even in a large study that pre-filtered down to a small subset of companies most likely to have successful reuse, the failure rate was a depressing 33% [1]. Failure is loosely defined, but generally refers to software not being reused, and/or development costs not reduced.

One interesting fact that stands out is that there is no technological solution to the problem. Reuse success is not affected by choice of language, middleware, computer-assisted software engineering (CASE) tools, and so on [2]. In other words, there is no short-cut. Furthermore, while the creation of a repository was shown to be useful, in and of itself, it is not sufficient to insure success.

The overriding prerequisite for a successful reuse program is commitment from upper management. Without this, few if any company reuse programs have been successful. Extending this to an entire program implies that both the management of the waveform development and the program oversight management have to be committed to the process. This means that there must be some way to incentivize or reward development of reusable designs (note we purposefully say "design", not "code"), and/or punish development with poor reuse potential.

An additional factor in successful reuse programs is the reuse of high-level software artifacts [3]. This includes such things as high- and low-level designs, simulations, models, and test vectors. This is in line with the position we have laid out above. Another factor to success is the introduction of a common process that specifies how reuse activities are to be done. It is interesting that there is little commonality in the process itself - every company had their own methodology. The key factor appears to be that a common process is created and forced upon all users.

The last key factor in success is a rather nebulous "human factor" [4]. This refers to behavioral aspects of the people involved in the reuse program, the level of training, the amount of motivation, and so forth. Failing to account for the human factors often led to failure in the reuse program

4. MOVING FORWARD

In order to propagate code reuse, the necessary preconditions for success must be put in place. The above key success factors are certainly a means to this end. However, even these are insufficient in the absence of a business model that rewards and enforces code reuse. This means that in the context of an entire defense program, a business model must be put in place such that code reuse can be supported between companies. The commercial telecommunications industry has been able to put a model in place that involves relatively open specification and ratification whereby competing companies are incented to keep each other honest. Beyond that, an entire industry has been built upon cross-licensing of key and valuable IP that can enable advances in waveform development. This incentivizes companies to continue to invest in R&D not only to capture more value via competitive advantage, but also in order to be able to successfully engage in cross-licensing, since it is virtually impossible that a single company could single-handedly develop an all new method of communications. In order for an entire defense program to successfully be able to leverage code reuse, the acquisition strategy must put in place such a business model that supports code reuse.

3. CONCLUSION

In this paper we have made the case that achieving waveform code portability requires more than a focus on the code itself. In a real sense, knowledge must be transferred between the company that develops the waveform and the company that reuses the design. Final executable code is not a good vehicle for this knowledge reuse. It requires good, thorough documentation, trade studies, models, simulations, fixed-point analysis, and comprehensive test vectors. Equally important is the creation of a viable business model to encourage the development of reusable code and artifacts. There is no technological solution to the reuse problem. It requires a common process, strong oversight, and a means of verifying proper design practices.

[1] M. Morisio, M. Ezran, and C. Tully, "Success and Failure Factors in Software Reuse", *IEEE Trans on Software Eng*, Vol. 28, No. 4, pp 340-57. 2002.

[2] W. Frakes and C. Fox, "Quality Improvement Using a Software Reuse Failures Model", *IEEE Trans on Software Eng*, Vol. 23, No. 4, pp275-79. 1996.

[3] D. Rine and R. Sonneman, "Investments in Reusable Software: A Study of Software Reuse Investment Success Factors", *J. Systems and Software*, Vol 41, pp17-32. 1998.

[4] M. Griss, "Software Reuse: Objects and Frameworks are Not Enough", *Object Magazine*, pp77-87. 1995.