Hardware/Software Co-Design via Specification Refinement

By

Wesley G. Peck

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Committee members

Dr. Perry Alexander, Chairperson

Dr. Andy Gill

Dr. Prasad Kulkarni

Dr. Xin Fu

Dr. Caroline Bennett

Date defended:

The Dissertation Committee for Wesley G. Peck certifies
that this is the approved version of the following dissertation :

Hardware/Software Co-Design via Specification Refinement

____

Dr. Perry Alexander, Chairperson

Date approved: ________________

Abstract

System-level design is an engineering discipline focused on producing methods, technologies, and tools that enable the specification, design, and implementation of complex, multi-discipline, and multi-domain systems. System-level specifications are as abstract as possible, defining required system behaviors while eliding implementation details. These implementation details must be added during the implementation process and the high effort associated with this locks system engineers onto the chosen implementation architecture.

This work provides two contributions that ease the implementation process. The Rosetta synthesis capability generates hardware/software co-designed implementations from specifications that contain low level implementation details. The Rosetta refinement capability extends this by allowing a system's functional behavior and its implementation details to be described separately. The Rosetta Refinement Tool combines the functional behavior and the implementation details to form a system specification that can be synthesized using the Rosetta synthesis capability. The Rosetta refinement capability is exposed using existing Rosetta language constructs that have, previous to this work, never been exploited.

Together these two capabilities allow the refinement of high level, architecture independent specifications into low level, architecture specific hardware/software co-designed implementations. The result is an effective platform for rapid prototyping of hardware/software co-designs and provides system engineers with the novel ability to explore different system architectures with low effort.

Page left intentionally blank.

Contents

Abstract . . . . **iii**

Contents . . . . **v**

List of Figures . . . . **vii**

List of Tables . . . . **viii**

List of Code Examples . . . . **ix**

1 Introduction **1**

1.1 Motivation 4

1.2 Research Contribution 4

2 Related Works **5**

2.1 Ad-Hoc Program Transformation 5

2.2 Program Transformation Tools 6

2.3 Program Synthesis 7

2.4 Approaches to System-Level Design 14

3 The Rosetta Specification Language **17**

3.1 Domains 17

3.2 Facets 19

3.3 Requirements Modeling and Specification 20

4 Synthesis of Rosetta Specifications **23**

4.1 Synthesizable Hardware Components 24

4.1.1 RTL Representation 29

4.1.2 Graph Representation 39

4.1.3 VHDL Production 48

4.2 Synthesizable Software Components 50

4.2.1 Facet State Representation 51

4.2.2 Behavioral Terms as Software 51

4.2.3 FSL I/O Support 54

4.2.4 Conversion into an MicroBlaze Instruction Stream 56

4.2.5 Deficiencies in Software Synthesis 56

4.3 Synthesis Example 56

5 HW/SW Co-Design of Top-Level Designs **65**

5.1 Processing Refinements 68

5.2 Implementation Resources 71

5.2.1 Target Physical Resources 73

5.2.2 Bridge Physical Resources 74
5.2.3 Memory Physical Resources 76
5.3 System Partitioning 77
5.3.1 Target Attributes 77
5.3.2 I/O Parameters 78
5.3.3 I/O Partitioning for Software 79
5.3.4 Default Partitionings 81
5.4 Component Implementation 81
5.5 Project Generation 82
5.6 Top-Level Design Example 84
6 Rapid Prototyping Refinements 89
6.1 Refinement Discovery and Application 90
6.2 Behavior Refinement 92
6.2.1 Hardware Clocks via Refinement 92
6.2.2 Hardware Resets via Refinement 95
6.2.3 Hardware Enables via Refinement 96
6.3 Data Refinement 98
6.3.1 Refinement of Integer Types 100
6.3.2 Refinement of Algebraic Data Types 101
6.3.3 Refinement of Declarations 107
6.3.4 Data Type Refinement Limitations 108
6.4 Communication Refinement 109
6.4.1 Communication Refinement in the Top-Level Design 110
6.4.2 Fast Simplex Link Protocol 111
6.4.3 Go/Done Protocol 116
7 Evaluation 119
7.1 RS-232 Loop-Back Example 121
7.2 State Machine Example 125
7.3 Trusted Platform Module Design 131
7.4 Evaluation Summary 137
8 Future Work 139
9 Conclusion 141
References 143
A Synthesis Constraints 146
B MicroBlaze LLVM Backend 148
C Hardware Example VHDL Output 149
D Software Example MicroBlaze Output 150
E Top-Level Design Resource Definitions 152
F Xilinx Platform Studio Project Files 154
G TPM Specification 162

List of Figures

2.1 Formal Specification in Program Synthesis Literature 7
2.2 Example Square Root Formal Specification 8
2.3 Square Root Proof as Program Example 8
2.4 Example Less-All Formal Specification 9
2.5 Example Less-All Program using Transformational Synthesis 10

4.1 Example Synthesized Hardware Circuit 27
4.2 Graph Comparator Canonicalization 43
4.3 Graph Multiplexer Canonicalization 44
4.4 Graph Register Canonicalization 46
4.5 Graph RAM Canonicalization 47
4.6 Rosetta If-Expression LLVM-IR 53
4.7 Rosetta Linear Feedback Shift Register Graph Representation 60
4.8 Rosetta Pseudo-Random Number Generator Graph Representation 61

5.1 Top-Level Design Physical Description 86
5.2 Top-Level Design Logical Description 86
5.3 Resulting Top-Level Design Implementation 87

List of Tables

4.1 Types Supported by Rosetta HW Synthesis 25
4.2 Parameters Supported by Rosetta HW Synthesis 25
4.3 Expressions Supported by Rosetta HW Synthesis 26
4.4 Representation of RTL Nodes 30
4.5 RTL Intermediate Form Type Representation 31
4.6 RTL Intermediate Form Parameter Representation 32
4.7 Conversion of Rosetta Expressions into RTL Nodes 36
4.8 Representation of Graph Nodes 40
4.12 Conversion of Explicit Graph into VHDL 49
4.13 Software Representation of Facet State 51
4.14 Conversion of Rosetta into LLVM-IR 52
4.15 Rosetta Linear Feedback Shift Register RTL Intermediate Form 58
4.16 Rosetta Pseudo-Random Number Generator RTL Intermediate Form 59

6.1 Guided Data Type Refinement Representations 104
6.2 Definitional Data Type Refinement Representation 107
6.3 Refinement of Abstract Communication into Fast-Simplex Link Signaling . . . 114
6.4 Refinement of Abstract Communication into Go/Done Signaling 118

7.1 RS-232 Rapid Prototyping Changes 122
7.2 RS-232 Example Implementation Efficiency 123
7.3 State Machine Rapid Prototyping Changes 128
7.4 State Machine Example Implementation Efficiency 129
7.5 Trusted Platform Module Rapid Prototyping Changes 134
7.6 Trusted Platform Module Example Implementation Efficiency 135

List of Code Examples

3.1 Subtle Difference Between Implementation and Specification 20
3.2 Obvious Difference Between Implementation and Specification 21
3.3 Underspecified Rosetta Function 21

4.1 Example of a Synthesizable Rosetta Facet 28
4.2 Example Synthesis Attributes 29
4.3 Importing Libraries for Hardware Synthesis 34
4.4 Example of Synthesizable Behavioral Terms 34
4.5 Rosetta Hardware Synthesis Example 57
4.6 Rosetta Software Synthesis Example 62
4.7 Rosetta Multiply-Accumulate LLVM Intermediate Representation 63

5.1 Shared Facet Definitions 69
5.2 Specialized Facet Definitions 69
5.3 Top-Level Design Resource Instantiation 72
5.4 Rosetta Implementation Target 73
5.5 Rosetta Communication Bridge 75
5.6 Rosetta Memory Resource 76
5.7 System Partitioning in a Top-Level Design 78
5.8 Full Top-Level Design Example 85

6.1 Refinement Discovery Example 91
6.2 Hardware Clock Refinement 92
6.3 Refined Hardware Clock 93
6.4 Hardware Reset Refinement 95
6.5 Refined Hardware Reset 96
6.6 Hardware Enable Refinement 97
6.7 Refined Hardware Enable 98
6.8 Data Representation in a Counter 99
6.9 Data Refinement of Integer 100
6.10 Refined Integer Data Type 101
6.11 Algebraic Data Type in Rosetta 102
6.12 Guided Algebraic Data Type Refinement 103
6.13 Definitional Algebraic Data Type Refinement 106
6.14 Data Type Refinement of Declarations 108
6.15 Incorrect Application of Data Type Refinement 108
6.16 Abstract Communication in Rosetta 109

6.17 Fast-Simplex Link Communication Refinement 111
6.18 Refined Fast-Simplex Link Communication Link 112
6.19 Go/Done Communication Refinement 116
6.20 Refined Go/Done Communication Link 117

7.1 RS-232 Loop-Back Specification 120
7.2 State Machine Specification 126

1 Introduction

System-level design is an engineering discipline focused on producing methods, technologies, and tools that enable the specification, design, and implementation of complex, multi-discipline, and multi-domain systems. Modern computer systems are composed of thousands to millions of heterogeneous components so technologies and tools for system-level design are aimed at helping engineers deal with the complexity of requirements, design, implementation, and integration when creating systems of this scale.

Traditionally, complexity is reduced by introducing abstractions. These abstractions, if chosen properly, prune unimportant details from the engineer's view and focus them on important details. For instance, when designing a large hardware system by composing together multiple hardware components, all of the individual components in the system are abstracted into input/output interfaces by throwing away the internal implementation details. Because the implementation details are no longer present, the engineer composing the system must reason about system behavior by examining the interaction of components via their input/output protocols. The abstraction is appropriate in this instance because it is focusing the engineer on the task at hand, component composition. Focusing on how every component in the system computes its outputs from its inputs would be needlessly complicated. However, abstractions must eliminated if systems are to be realized as concrete products. There are two ways that this can be done.

First, a compiler or synthesizer can be used to remove abstractions by filling in the details needed for a concrete implementation. Expanding on the example above, a synthesizer will fill in an instantiated component's interface with the internal implementation details producing a complete hardware circuit for the design. This approach is highly desirable because it reduces engineering effort, reduces time-to-market, and controls design and implementation costs. However, completely automated concretization via compilation and/or synthesis can produce inefficiencies. For many products, especially software products, these inefficiencies are tolerable and can be worked around by increasing the performance of the underlying

hardware. At the system-level, however, these inefficiencies add up quickly due to the scale of the problems. The resulting implementations are either undesirable or incorrect due to non-functional requirements not expressed in the design and not taken into account during the compilation process.

Second, an engineer can manually eliminate abstractions by replacing them with low-level equivalents. This method is common in system-level design due to inefficiencies introduced by automated synthesis. Highly abstract system models are developed using a system-level language so system properties can be accurately understood. It is then converted into an implementation by developing an equivalent low-level model containing only implementation-level abstractions. Any unsupported abstractions in the abstract model must be replaced with their low-level equivalents. Abstractions are never completely removed by a system engineer, they are simply iteratively refined until the engineer is content with the results produced via automated synthesis. The problem with first approach is not that tools are incapable of producing efficient implementations. Indeed, compilers for many languages produce more efficient code than most engineers are capable of and they do so without introducing errors. The problem is the size of the semantic gap between abstract specifications and concrete implementations.

The size of this semantic gap is dependent on the language being used and is caused by the classic trade-off in language design: expressiveness versus interpretability. Common languages, such as C/C++ and Java, sacrifice expressiveness so that they can easily be interpreted by a computer. The semantic gap for languages that lean towards interpretability is small because the abstraction levels are low. The semantic gap between a C/C++ design and the interpretation of that design at the microprocessor level is small. A compiler can easily navigate this gap on behalf of the engineer because there are few intelligent choices that need to be made. A good compiler could easily test all of the available choices and pick the best choice based on some optimization criteria set by the engineer. Most compilers for these languages already do this to some extent.

Languages that lean towards expressiveness sacrifice interpretability to gain high levels of abstraction. As the abstraction level rises, however, the semantic gap between design and implementation becomes larger and the number of intelligent decisions that a compiler must make grows. Eventually it is either impossible or undesirable for a compiler to make these decisions. For instance, expressive specification languages allow system engineers to purposefully underspecified systems by omitting the definition of a function. Obviously there is no way for a compiler to "fill in implementation details" for a function that is never specified.

Even when a compiler is able to provide the required details, their use may still be undesirable. To be desirable, the compiler must make implementation decisions on behalf of an engineer that the engineer considers both correct and intelligent. Compilers are typically very good about making correct decisions and, as such, they rarely introduce errors into an implementation. However, in highly abstract designs, compilers are not always capable of making intelligent decisions because *they lack the information needed to make intelligent implementation decisions* because this information is often elided in specifications.

The system design as described by a specification is ambiguous with respect to desired implementations because specifications are *suppose* to be ambiguous in this respect. Modern compilers are poorly suited to these highly abstract specification languages because the intentional ambiguity leads to poor implementation decisions. To achieve synthesizability, the most often used system-level design methodologies are built upon low-level abstractions with system engineers often hand-coding software components in C/C++ and hardware components in VHDL or Verilog. Even the next generation of system-level design tools only use moderate levels of abstraction. For instance, SpecC, SystemC, and SPARK are all next generation system-level synthesis tools that take C/C++ designs in various forms and synthesize systems from them [8, 10, 19, 20]. While C/C++ provides much more abstraction for hardware design than the traditional VHDL or Verilog, even these abstraction levels are far below those provided by specification languages such as Rosetta.

1.1 Motivation

The motivation for this work comes from personal experience in system-level design and implementation. System engineers want to raise abstraction levels because it eases the development effort by reducing the overwhelming amount of information in a system design into approachable chunks. Raising abstraction levels helps produce correct designs quicker and cheaper. In contrast, system engineers also desire control over every aspect of a product's final implementation so that non-functional requirements can be met. These non-functional requirements vary depending on the engineering domain, but include things such as minimizing computational latency, maximizing computational throughput, minimizing transistor counts, minimizing memory requirements, minimizing jitter, maximizing battery life, reducing cost, etc.

The great conundrum in system-level design, especially when multiple engineering domains are involved, is how to produce methods, technologies, and tools that promote rapid creation of functionally correct specifications and designs without compromising control over non-functional requirements.

1.2 Research Contribution

This work provides two major contributions, one to the engineering community and another to the research community. First, a Rosetta synthesis capability is provided that allows a subset of the Rosetta specification language to be automatically and efficiently implemented either as hardware circuits via VHDL or software programs via LLVM. Second, a system-level refinement capability is provided that supports rapid transformation of abstract, architecture independent specifications into low-level, architecture specific implementations. Together these two contributions provide an effective rapid prototyping platform for hardware/software co-design using the Rosetta specification language and give system engineers the novel ability to rapidly explore different implementation architectures without sacrificing control over the design of those architectures.

2 Related Works

The refinement capability developed in this work is implemented as a program transformation system that refines system-level specifications into HW/SW co-designed implementations. The transformation system identifies declarative refinements present in a system specification and then mechanically applies those refinements, creating a new system specification in the process.

Program transformation has a long and varied history in computing. Modern programming language compilers are built as complex program transformation systems that iteratively transform the high-level language syntax into low-level machine implementations. Many formal languages, specify evaluation, type-checking, and proof systems as term-rewriting systems and can be considered program transformation systems. Additionally, some modern programming languages include program generation facilities implemented as program transformation systems.

2.1 Ad-Hoc Program Transformation

Ad-hoc program transformation techniques facilitate user directed transformation using tool specific techniques often expressed outside a language's normal syntax. These techniques typically focus on providing program optimization transformations or on providing program generation capabilities aim at eliminating boilerplate code.

The Glasgow Haskell Compiler (GHC) supports program transformation by applying user defined rewrite rules on any code that matches a rewrite condition [21]. The rules themselves are specified and applied using GHC specific extensions to the Haskell programming language. The authors make it clear that this rule-based program transformation system should only be used to facilitate program optimizations beyond the capabilities of the compiler. For instance, short-cut deforestation is a common optimization applied using GHC's rule based program refinement [12].

As another example, the Groovy [6] programming language makes use of declarative annotations to support transformation of the compiler's internal AST representation. The program refinements are created by implementing custom Groovy compiler plugins. These refinements are most often applied by matching annotations that are specified declaratively within the Groovy program. The Groovy compiler uses these program transformations mainly to facilitate program generation targeted at eliminating boilerplate code. For instance, lazy field instantiation can be used for a Groovy object simply by annotating the object. The compiler will then transform the implementation of the object to include the necessary code to facilitate lazy instantiation.

These ad-hoc program transformations systems focus on performing transformation that preserve the functional behavior of a program but alter the performance or memory requirements of the program. In contrast, this work's transformation system focuses on the process of refining specifications into implementations via the addition of implementation details used to iteratively refine high-level specifications into low-level specifications. These additional implementation details are concerned chiefly with the process of mapping behaviors at the specification level onto resources at the implementation-level. Additionally, the process of mapping functional behavior onto implementation-level resources allows specification of specific implementation architectures, something not supported in existing program transformation tools.

2.2 Program Transformation Tools

Program transformation tools use program transformation techniques to aid in the development of a program. These tools range from standalone tools that transform specific languages, such as KBEmacs[36] and Inject/J[11], to transformation languages such as Stratego[35, 5] that help designers create custom transformations for any language, to runtime transformation engines that enable the construction of custom program transformation tools such as the ubiquitous byte-code engineering libraries for Java or the generic programming libraries Scrap Your Boilerplate[24] and Strafunski[23] for Haskell.

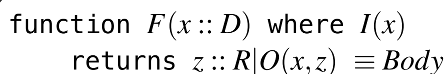
There are two classes of tools presented here. First, KBEmacs and Inject/J provide program transformation capabilities that aid in the development of a program. When using these tools a programmer will typically intermingle manual coding with automated program transformation to create the desired final program implementation. The second class of tools aid in the creation of custom program transformation tools. In the case of Stratego, a domain specific language is exposed that aids in the development of abstract syntax trees, rewrite rules, and traversals. This domain specific language can be compiled into a C program that implements the transformation system. Strafunski and Scrap Your Boilerplate are similar in that they aid in the development of custom transformation systems but are provided as Haskell libraries instead of as languages.

In comparison to the first class of program transformation tools, this work is chiefly concerned with providing rapid prototyping and hardware/software co-design. Instead of intermingling manual coding and automated program transformation, the approach is to expose transformations as special language constructs that are intermingled within the specification. This approach provides better support for rapid prototyping because refinements are kept separate allowing them to be easily modified to achieve different results.

In comparison to the second class of program transformation tools, this work is not concerned with aiding in the development of transformation systems. Indeed, this work's transformation system is built with the aid of Uniplate[29], a Haskell transformation library in the same spirit as Strafunski and Scrap Your Boilerplate.

2.3 Program Synthesis

Figure 2.1: Formal Specification[32, 22]



```
function F(x :: D) where I(x)
    returns z :: R|O(x,z)  ≡ Body
```

Program synthesis refers to a class of research that deals with automating software development with the goal of mechanically deriving correct and efficient programs from formal specifications. Research in this area can be broken down into three major categories: proofs as programs, transformational synthesis, and knowledge based program synthesis[22].

All of the program synthesis research operates over formal specifications that can be represented as a quadruple $F = < D, R, I, O >$ where D is the input type constrained by the input condition $I : D \mapsto boolean$ and R is the output type constrained by the output condition $O : D \times R \mapsto boolean$[32, 31, 22]. Syntactically, these specification can written as shown in figure 2.1. For a specification to be consistent, the expression *Body* and the output definition O must produce the same solutions for any input deemed valid by I.

Proofs as Programs

Figure 2.2: Square Root Specification

```
function sqrt(x :: ℕ) where x ≥ 0
    returns z :: ℕ|(z² ≤ x ∧ x < (z+1)²)
```

In the proofs as programs approach to program synthesis, specifications are hypotheses of the form $\forall(x :: D) . \exists(z :: R) . (I(x) \implies O(x,z))$. In words, hypotheses of this form state that for all valid inputs there exists a valid output. Program synthesis then proceeds by performing a constructive proof of the conclusion $O(x,z)$ using the premise $I(x)$. Because the proof is constructive, a program can be automatically extracted from the proof[22]. Proofs as programs techniques and tools include: QA3[18], Coq[25, 27], NuPrl[7], and Isabelle/HOL[4].

As an example of the proofs as programs approach, Kreitz[22] presents the formal specification of an integral square root algorithm as $\forall(x :: \mathbb{N}) . \exists(z :: \mathbb{N}) . (x \geq 0 \implies z^2 \leq x \wedge x < (z+1)^2)$. The syntactic representation of this formal specification is given in figure 2.2.

Figure 2.3: Square Root Proof[22]

```
function sqrt(x :: ℕ) where x ≥ 0
    returns z :: ℕ|(z² ≤ x ∧ x < (z+1)²) ≡
    if x = 0 then 0 else
        let z = sqrt(x − 1) in
        if z² = x − 1 then z + 1 else z
```

The constructive proof of this formal specification can be done inductively. The base case of the inductive proof, $x = 0$, is proved by supplying the witness 0 for z_0. The inductive case uses the inductive hypothesis "assume $\exists(z_{i-1} :: \mathbb{N}) . z_{i-1} = sqrt(x-1)$". To prove the inductive case a case analysis is done on z_{i-1}. In the first case $z_{i-1}^2 = x - 1$ and the witness $z_{i-1} + 1$ can be supplied for z_i to discharge the proof. In the second case $z_{i-1}^2 \neq x - 1$ and the witness z_{i-1} can be supplied for z_i to discharge the proof.

The complete specification with accompanying proof is shown syntactically in figure 2.3. It should be clear that a program could be extracted from the proof of the formal specification of *sqrt*. Research in the proofs as program domain focuses on the development of strong, constructive theorem proving languages and the extraction of efficient programs from these constructive proofs[22].

Transformational Synthesis

Transformational synthesis techniques typically view formal specifications as executable programs that are simply inefficient. Starting with these easily written but inefficient formal specifications, an efficient executable program is derived through the application of correctness preserving term rewriting rules. Efficiency is achieve by applying simplification and abstract data type removal, often at the expense of complicating the formal specification[26]. DEDALUS[30] is an example of a program synthesis tool in this area.

As an example, Manna and Waldinger[26] introduce a formal specification for the *lessall* predicate and give the derivation of an efficient program. Figure 2.4 shows the same specification using the notion of a specification as a quadruple $< D, R, I, O >$. This specification states that *lessall* is true only when the natural valued input x is less than all of the natural values in the input list l.

Figure 2.4: Less-All Specification

function $lessall(x :: \mathbb{N}, l :: List(\mathbb{N}))$ where $true$
    returns $z :: Bool | z = all(\lambda i \mapsto x < i, l)$

The DEDALUS[30] transformation system contains many generic rewrite rules allowing for things such as arithmetic simplification and case analysis. In addition to the generic rules, domain specific rewrite rules can be introduced. These rewrite rules could be considered axiomatic function definitions rather than operational function definitions. For instance, the following domain specific rewrite rules are used to transform this specification:

Vacuous All Rule: $all(P, l) \mid (l = nil) = true$
All Decomposition Rule: $all(P, l) \mid (l \neq nil) = P(head(l)) \wedge all(P, tail(l))$

The first transformation done to the *lessall* specification is to apply a case analysis rule over the input list l with the cases being $l = nil$ and $l \neq nil$. This rule introduces a conditional into the specification transforming the body of the specification into $if\ l = nil\ then\ all(P, nil)\ else\ all(P, l)$ where $P = \lambda i \mapsto x < i$.

At this point the vacuous rule can be applied to the then branch of the conditional and the decomposition rule can be applied to the else branch of the conditional. Application of these two rules produces $if\ l = nil\ then\ true\ else\ P(head(l)) \wedge all(P, tail(l))$. If the operations $<$ and *head* are considered primitive then the expression $P(head(l))$ cannot be simplified any further. Thus, further transformation concentrates on the expression $all(P, tail(l))$

Figure 2.5: Less-All Program[26]

```
function lessall(x :: ℕ, l :: List(ℕ)) where true
    returns z :: Bool | z = if l = nil then true
            else (x < head(l)) ∧ lessall(x, tail(l))
```

Noticing that $all(P, tail(l))$ is an instance of the original expression $all(P, l)$ with a smaller list, a recursion introducing rule can be applied to transform the expression $all(P, tail(l))$ into simplified expression $lessall(x, tail(l))$. The final program representation for the *lessall* specification is shown in figure 2.5. This program representation implements the original specification of *lessall* correctly and efficiently.

Knowledge Based Program Synthesis

Knowledge based program synthesis focuses on program synthesis as a development process requiring intimate knowledge of both software engineering and of the specification's engineering domain. Typically, knowledge based program synthesis tools provide formally defined, generic synthesis strategies that automatically construct algorithms via analysis of the specification. These algorithms can then be made more efficient though the application of optimizing transformations. The Kestrel Interactive Development System (KIDS) is representative of this category of program synthesis tools[32]. A modern, commercially supported descendent of KIDS is available in the form of SpecWare[28].

Consider the abstracted specification given in figure 2.1. In the KIDS nomenclature, this specification defines a problem F with a domain of D and a range of R. The expression $Body$ of the problem F may be omitted if the problem specification is abstract. If the expression is present, however, it defines how solutions to the problem F can be computed from the inputs. The input condition I and the output condition O can be considered constraints over valid inputs and constraints over valid solutions for a given input. In effect, I and O allow tighter input and output constraints for a problem specification than the implicit input and output constraints due to the problem's domain and range types.

From this definition, KIDS provides interactive support for automatic derivation of correct and efficient programs. To accomplish this a KIDS users typically goes through six steps, though the steps are not prescribed by the KIDS tool[31]. First, the user will develop a domain theory that describes the relevant concepts, operations, relationships, and properties of the problem that is being solved. Domain theories are constructed in KIDS by importing existing domain theories and then adding additional type definitions, function specifications, laws, and rules of inference.

Second, a problem specification will be created that uses the definitions provided by the domain to describe solutions to a specific problem. The problem specification is provided in the form shown in figure 2.1. The initial problem specification omits the expression $Body$ with valid solutions specified solely via the output constraints O. Then a design tactic us applied to the problem specification to create an initial operational definition (i.e. the design tactic provides an initial definition of $Body$). KIDS' design tactics are generalized problem solving algorithms such as divide-and-conquer, local search, or global search. These generalized algorithms are then specialized, using the KIDS tools, to solve a particular problem.

For example, a KIDS user may decide that global search is a good algorithm for solving their problem. To provide the initial operation definition for their problem F the user would:

1. Select a global search theory G_b from a library of existing general global search theories such that G_b enumerates all possible values for the range of F. Because G_b enumerates

all values in the range of F we can be sure that solutions to F as defined by O are in the set of values enumerated by G_b.

2. Produce a specialized global search theory G_f from the global search theory G_b with respect to the problem specification F by finding a mapping θ that maps inputs x of problem F into inputs $\theta(x)$ of G_b such that $\forall(z :: R_f).O_f(x,z) => O_b(\theta(x),z)$. This mapping can be determined automatically by the KIDS tools.
3. Derive a $necessary\ filter$ to prune the search space of invalid partial solutions and use it to create a global search program $Body$. This provides the initial operational definition for the problem specification F.

The initial operational definition formed during these first three steps is a correct behavioral description of the specification, however, the definition is typically far from efficient. To complete the process of generating a correct and efficient executable from the specification, the user would:

4. Apply optimizations to the initial operational definition of the problem F to create a more efficient operational definition. During this step the KIDS user manually identifies expressions within the operational definition of F and then applies optimizations to those expressions. The optimizations are performance automatically by KIDS and include operations such as simplification, partial evaluation / specialization, finite differencing, and case analysis[31].
5. Apply data type refinement to the operational definition to eliminate abstract data types. The abstract data types are replaced with specialized representations that efficiently represent the data used by the operational definition. It is important to note that the elimination of abstract data types respects the correctness properties of that data type. For example, when replacing a set with a list representation the transformation must ensure that an element does not appear in the list twice.
6. Compile the operational definition of the problem specification into an executable that will compute solutions to the problem when given inputs.

After completing these six steps, the KIDS users has automatically derived a correct and efficient executable representation of their specification.

Comparison to Proposed Work

Program synthesis tools, are related to this work in the sense they all desire to produce efficient program representations from high-level, difficult to synthesize specifications. Each of the categories of program synthesis also have some notion of how engineers interact with the program synthesis tool to achieve the desired results. In the proofs as programs category, engineers interact with the program synthesis tool via the creation of a proof where the choice of a particular proof construct will influence the eventual outcome. Tools in the transformational synthesis category vary on the level of engineer interaction with some tools being highly automated and other tools being mainly engineer driven[9]. Knowledge based transformation systems try to strike a balance by giving engineers control over the application and composition of transformations that are then mechanically applied.

Another relation between all program synthesis tools is that program transformations are describe either formally or semi-formally so that the result of applying the transformation is well-known to the engineer. The transformations supplied by this work are not formally defined but the intention is that the result of applying the transformations is well-known. Formalizing the transformations is left as future work.

The main difference between this work and existing program synthesis tools is the scope of the transformations that are provided. Existing program synthesis tools aim to produced a single, efficient, software implementation from abstract specifications. This work is concerned primarily with providing a rapid-prototyping capability that is capable of iteratively refining abstract specifications into one of many low-level, hardware/software co-designed implementations.

2.4 Approaches to System-Level Design

There are many existing tools and techniques for performing system-level design. Generally, these tools and techniques fall into one of two categories. First, some tools ease system-level design by generating hardware components from high-level programming languages. Tools such as Catapult C[15], Handle-C[16], and SPARK[20] generate circuits from algorithmic descriptions written in languages like C, C++, or SystemC[19]. BlueSpec[3] and Kansas Lava[13] take the abstraction levels used to implement hardware circuits even higher by leveraging functional programming techniques to describe circuit elements.

As an example, Kansas Lava is capable of producing hardware circuits from descriptions written in a domain specific language that is hosted in Haskell. During this process the system engineer describes functional behavior and communication using highly abstract constructs, manually refines behavior using worker/wrapper[14] transformations, and then produces a VHDL implementation using the Kansas Lava tools. The end result is a hardware system that implements the described functional behavior.

Compared to these tools, this work supports HW/SW co-designed implementations and rapid prototyping. Support for HW/SW co-design means that this work is capable taking a high level system specification and generating a system implementation composed of interacting hardware and software components. The tools described previously only support generation of hardware systems. To support rapid prototyping, a tool must allow rapid generation of many system implementations from a single system description. This work achieves this by separating high level functional behavior from low level implementation details. When using the tools described previously, functional behavior and implementation details are intermingled, making it difficult to produce different implementations from a single description.

The second category of tools generate HW/SW co-designed system implementations from a single source language. SpecC[10] and Impulse C[33] support HW/SW co-design of systems by synthesizing hardware circuits and software instruction streams from ANSI-C like

languages while Kiwi[17] is capable of generating HW/SW system by analyzing .NET C# programs.

As an example, Kiwi is capable of producing HW/SW system implementations from descriptions written in C# and compiled to the .NET virtual machine. During this process, a C# program is written to describe a system's functional behavior and communication and then compiled to the .NET virtual machine. Partitioning of this C# program's instruction stream is then done, with some components partitioned onto Xilinx FPGAs for implementation and other components left on the .NET virtual machine. Integration of the hardware and software components into a complete system implementation is done by connecting the components using some form of physical communications link, Ethernet switches being one example.

The main difference between this work and the second class of tools is that this work provides better support for rapid prototyping and design space exploration by separating high level functional behavior and low level implementation details. In this work, a system's implementation is derived from its behavioral specification by adding implementation details in the form of easily changed, declarative, system-level refinements. Different implementations can be generated rapidly by changing the refinements used to derive the implementation. The second class of tools conflate the behavioral specification of a system and the implementation of that system. Because of this, changes in a system's implementation must be achieved by changing the system's described behavior, making it difficult to rapidly generate different implementations.

Page left intentionally blank.

3 The Rosetta Specification Language

Modern system design is increasingly complex and makes use of heterogeneous models of computation so that individual components in a system can be designed and implemented in the most appropriate domain Additionally, modeling an understanding system-level properties, such as security or power consumption, is increasingly important. The Rosetta[1, 2] system-level design language is designed to meet the needs of modern system engineers. It places special emphasis on support for multi-domain modeling to support heterogeneous models of computation and allows the precise, formal expression of requirements in ways that are natural to the domain. Additionally, system-level properties can be expressed succinctly using behavior composition that is a core feature of the language.

3.1 Domains

The specification, design, and implementation of modern systems requires expertise in a wide variety of engineering and scientific domains. Consider for example, a modern mobile phone. Even the most basic modern mobile phone requires many components:

1. ADC and DAC - Conversion from and to analog representation.
2. DSP and CPU - Computation including processing of audio and video data.
3. RF, Bluetooth, and WiFi - Wireless communication using analog signals.
4. LCD - Video display and user interface.
5. Battery - Power for all of the components in the mobile device.

Expertise in many domains is needed to support all of these components. Expertise in analog signals, analog circuitry, digital signals, and digital circuitry is needed for the requirements modeling, specification, design, and implementation of most of the components. Expertise in electrical power sub-systems is required if the mobile device is to meet its power requirements. Expertise in networking is needed so that the device can communicate using its wireless communications components. Expertise in audio and video processing is required to support voice communications and audio/visual media. Expertise in wireless communications is needed to support the analog transmission and reception of signals. Additionally, expertise in computer security is needed to ensure a reasonable standard of privacy on behalf of the user. Even this

long list is not exhaustive because the amount of expertise needed is vast due to the number of system domains covered by the components in the device.

Modern systems necessitate support for the multi-domain requirements modeling, system specification, design, and implementation. At the design and implementation level this has traditionally been done by providing separate tools that target one specific domain. Integration of the separately designed and implemented components is normally done by hand in a separate, ad-hoc integration phase. This works well enough for many products, as evidenced by the number of complex yet working systems on the market, but recomposition of components during the integration phase can have a significant time and cost impact.

Using separate tools for requirements modeling and system specification is difficult to the point of being infeasible because of the complex interactions among different components in the system. For example, when modeling power requirements in a system the power usage of one component will affect the power usage and/or requirements of another component. Modeling the requirements using separate tools and integrating them by hand would quickly become too burdensome.

Rosetta proposes a solution to the problem by directly exposing multi-domain support in a single language. This has not been done traditionally because each domain uses different vocabulary for expressing requirements and constraints, a different model of computation, and different abstractions. For instance, requirements and constraints in the power domain would be expressed using terms such as *volt*, *current*, *ampere*, and *watt*, whereas terms such as *cycles*, *register*, *mux*, and *bit* would be used in the digital circuits domain.

Rosetta provides support for multiple engineering domains by allowing vocabularies, models of computation, and abstractions to be defined inside of the Rosetta language in the form of a Rosetta domain. Domains are hierarchical in natural in that one domain can inherit the vocabulary, model of computation, and abstractions provided by another domain and extend or constrain them as appropriate. System components are constructed using the vocabularies, models of computation, and abstractions provided by the component's domains. Additionally,

Rosetta allows engineers to express how different domains interact with one another so that system-level properties can be understood across domain boundaries.

3.2 Facets

Rosetta facets build upon Rosetta domains to describe one model in a system of models. The facet is expressed using the vocabulary, model of computation, and abstractions provided by the domain. For instance, a behavioral model of a JK flip-flop could be constructed in the *state based* Rosetta domain by expressing the next state, Q', of the JK flip-flop in terms of its current input bits J and K and its current state, Q:

$$Q' = (J \text{ and } (\text{not } Q)) \text{ or } (K \text{ and } Q)$$

In this definition the model of computation, the notion of *current state*, *next state*, *bit*, and the behavior of the *and*, *or*, and *not* operations are defined by the domain *state based*. We could just as easily provide a power model of the JK flip-flop:

$$P' = P + (\text{if } (J \text{ or } K) \text{ then } \mathit{switch} \text{ else } \mathit{leak} \text{ endif})$$

Likewise, switch and leak might be definitions that come from a power domain modeled in Rosetta. Their definitions could come from either a simple, generic model of power or from a more complex model of power that takes into account some fabric implementation technology, for instance 45*nm* CMOS. The simple, generic model would be faster to construct and could be used during the initial requirements and specification phases to get coarse grained estimates on power usage in the system while the more complex 45 *nm* model could be used once a more complete model has been produced to provide tighter bounds on the power consumption estimate.

Each different facet expresses some model of the component using the vocabulary, model of computation, and abstractions that are most convenient when developing that model. A final model of the component can then be created by composing the individual facets together to provide a new model using the compositional features of the Rosetta language.

Facet definitions permit both requirements modeling and specification of functional and non-functional properties. In addition, facets requiring pre-conditions and implications can be expressed using an extension of facets known as components.

3.3 Requirements Modeling and Specification

Inside Rosetta facets, requirements modeling and specification development is done using a rich expression language. This expression language is similar to most programming languages in that it supports primitive operations and values with which additional constructs can be derived using the available methods of composition. Unlike most programming languages, Rosetta is declarative. Models in declarative languages are defined by equations, in contrast to imperative programming languages where models are constructed by algorithms. Because of this, Rosetta, like most requirements modeling and specification languages, deviates from programming languages in two important aspects. First, expressions in the Rosetta language are used to represent properties of the system. This is in contrast to traditional programming language where expressions are used to represent computations. The distinction here can be subtle, as in listing 3.1, or obvious, as in listing 3.2.

Listing 3.1: Subtle Difference

```
1 (1) if x == 1
2         then y' = 1
3         else y' = 0
4     end if
5
6 (2) if x == 1
7         then y = 1
8         else y = 0
9     end if
```

In listing 3.1 there are two expressions with the first representing a property constraint in a system and the second representing a computation in a system. The first expression is stating that if x is equal to one in the current state then y is equal to one in the next state otherwise y is equal to zero in the next state. The second expression is stating that the storage location named y is updated to the value one if the storage location named x contains the value one otherwise y is updated to the value zero. The difference is subtle because the computation that updates y is effectively forming a new state in a way that is entirely consistent with the system properties stated in the first expression.

Listing 3.2 shows a more obvious distinction between expressing properties and expressing computation. The first expression in this example states that if *command* is not equal to $NOTHING$ then the next state of the system is one of two different states, either $READING$ or $WRITING$. The second equation is instead expressing a computation that computes a single next state by combining the data representation of $READING$ with the data representation of $WRITING$ using the *or* operator. More than likely the computation represents an invalid next state assignment whereas the expressed property is completely valid.

The second major deviation is a by-product of the first: expressions in Rosetta may be underspecified. Underspecification, in the context of requirements modeling and system specification, means that some details are omitted from the specification because they are deemed unimportant by the engineer. An engineer may even completely underspecify something to express the notion that they are only interested in the existence of something and are uninterested in any of its properties. In most cases underspecification is used so that system properties can be expressed axiomatically rather than definitionally.

Listing 3.2: Obvious Difference

```
1  (1) if command != NOTHING
2          then state' = READING or
3               state' = WRITING
4          else state' = IDLE
5      end if
6
7  (2) if command != NOTHING
8          then state = READING or
9                       WRITING
10         else state = IDLE
11     end if
```

Consider, for example, the specification given in listing 3.3. This specification defines the function *sin* but does not give a definition of how to compute the output of *sin* from its input *value*. Instead the specification simply constrains the range of the function *sin* to be between -1.0 and 1.0 using a declarative axiom.

Listing 3.3: Underspecification

```
1 sin( value :: real ) :: real;
2 forall( x :: integer |
3            sin(x) >= -1.0 and
4            sin(x) <= 1.0 );
```

Underspecification can be seen as a double edged sword. It is useful because complex specification logic can be reserved for the parts of system model that are unknown and must be explored. The parts of the model that are well known and have established behaviors can be

underspecified and their behaviors can be encoded axiomatically. However, if important parts of a model are left underspecified, then model exploration can be unintentionally constrained and important system properties may be missed or misunderstood. This is because engineers rarely understand complex systems in enough detail a-priori to known which parts of the system are well known, especially because interactions among different models are complex and not well defined.

Currently, engineers deal with this a-priori knowledge problem using manual specification refinement. A highly underspecified system is initially created and used to gain more insight into the problem. This additional knowledge is used to refine the original specification by removing some of the underspecification to gain more insight in to the problem. This process is repeated until the engineer finds that the model describes the system at the correct level of detail.

4 Synthesis of Rosetta Specifications

The system-level design tools developed in this work proceed through several steps during the refinement of high-level Rosetta specifications into low-level hardware/software co-design implementations. First, complex data-types, abstract communication, and functional behaviors are refined into lower level representations. Second, a specification's top-level design is partitioned and mapped to resources at the implementation level. Finally, each partition in the design is synthesized to either hardware circuits via VHDL or software programs via LLVM.

The goal of the low-level synthesis support is to be as close as possible to the eventual target implementation (either VHDL or LLVM). There are two major reasons for having this goal. First, having the directly synthesizable subset close to the implementation target means that an engineer can accurately predict the performance and area consumption of the low-level specification. This is important for a system-level design tool as the turn around time for pushing an implementation into a form that can be loaded into a physical system is much longer than for simple software programs. As such, system engineers want to know what the results will be like before this process begins. Second, being close to the implementation target eases the implementation of the synthesis engine, helping ensure that the implementation of the synthesis engine is free of bugs.

There are several implications of the close proximity between the synthesizable subset and the synthesis target. First, there are relatively few types and expressions in a specification language like Rosetta that are directly synthesizable to low-level hardware or software implementations. The purpose of the first two refinement steps is to ease this restriction, allowing a much richer set of Rosetta constructs to be synthesized.

Second, the subset of Rosetta that is synthesizable to hardware via VHDL is different than the subset of Rosetta that is synthesizable to software via LLVM. One reason for this is that we want to support all of the low-level constructs in the synthesis target in some form at the Rosetta specification level. This ensures that system engineers are not artificially constrained at the specification level and can achieve the desired implementation even if it means that

they must manually refine the specification. It is important to allow manual refinement as it is infeasible for the refinement system to support all possible refinements that the system engineer may want to perform.

Another reason that the synthesizable subset differs between hardware and software targets is that we do not want to force abstractions on the system engineer or impose a model-of-computation before the design process begins. If the synthesizable subset for both hardware and software targets were the same then there would necessarily be abstractions or models-of-computation in one or both targets that would not be present in the underlying implementation fabric. This is because the underlying model-of-computation for hardware and software components are vastly different. Software models-of-computation are inherently sequential because microprocessors force sequential execution of instructions while hardware models-of-computation are inherently parallel because current flows through all elements of the circuit at the same time. Forcing hardware to be sequential or software to be parallel would require the introduction of abstractions beyond the control of the engineer and beyond the ability of synthesis tools to eliminate efficiently.

Furthermore, allowing the synthesizable subsets to differ is precisely what allows this work's synthesis tool to be used efficiently and effectively for hardware/software co-design. Hardware/software co-design is an activity in optimizing system efficiency by leveraging the unique capabilities of software resources and hardware resources. In effect, by partitioning a system intelligently, we can gain all of the advantages of hardware over software or software over hardware while mitigating or eliminating the disadvantages. By doing so we gain system efficiency that would not be possible using exclusively software or exclusively hardware solutions.

4.1 Synthesizable Hardware Components

The Rosetta Synthesis Tool (RST) synthesizes Rosetta facets into hardware circuits via VHDL in four phases. First, the synthesizable Rosetta specification is lowered into an intermediate representation in register-transfer level form. Second, the intermediate form is converted into an explicit graph representation. Third, the graph representation of the circuit is optimized

Table 4.1: Supported Types

Rosetta Type	VHDL Type
bit	std_logic
word(W)	std_logic_vector(W-1 downto 0)
integer	std_logic or std_logic_vector
array(W,D)	array(D-1 downto 0) of std_logic_vector(W-1 downto 0)

and inferencing is done to discover high-level circuit constructs. Fourth, the graph representation is transformed into VHDL code.

The first step in this process, producing an RTL intermediate form, requires that the input specification already be in hardware synthesizable form. To be in this form, the synthesizable specification must only use the types given in table 4.1. This table lists the Rosetta types that are supported by the hardware synthesis process and gives the VHDL types used for their implementation.

The lowering of Rosetta types into VHDL types is extremely simple with bit and word(W) representing bits and bit-vectors at the circuit level. The array(W,D) Rosetta type is lowered into the VHDL array type. These three type transformations, though simple, are all that is needed to support all implementation level constructs for hardware circuits: wires, registers, logic gates, RAMs, and ROMs. The hardware synthesis transformation supports one additional type, integer. This Rosetta type is used to support the notion of an elastic width bit-vector. The exact width of these bit-vectors is determined automatically during the synthesis process, as discussed in section 4.1.2. The notion of elastic sizes could be extended to other types as well, if additional flexibility is required in the synthesis process.

Table 4.2: Supported Modes

Rosetta Mode	VHDL Mode
design	generic
input	in
output	out
	inout

All Rosetta facets in the synthesizable specification must meet certain requirements. First, the domain must be one of static, state_based, resource, memory, target, bridge, or system. The domains resource, memory, target, bridge, and system are used to describe the top-level system design and the resources instantiated within. They are discussed further in section 5.

The domains `static` and `state_based` are used to represent functional behavior in Rosetta specifications. These two domains are the only supported domains because they neatly capture the two different types of circuits at the implementation level. The `static` domain is used to describe stateless behavior. Stateless behaviors compute their output solely based on their current input, capturing the notion of combinational circuits. Comparatively, the `state_based` domain allows the description of stateful behavior and captures the notion of sequential circuits. RST ensures that any specification being synthesized meets the requirements on the domain but it performs no correctness checks. In particular, it assumes that the Rosetta type checking capability assures `static` and `state_based` facets meet their definitions according to the Rosetta language definition.

In addition to the domain requirements, all synthesizable Rosetta facets must only make use of the parameter modes given in figure 4.2. Like the transformation of Rosetta types into VHDL types, the transformation of Rosetta parameter modes into VHDL parameter modes is straight-forward. Rosetta `design` parameters become VHDL generics, Rosetta `input` parameters become VHDL `in` parameters, Rosetta `output` parameters become VHDL `out` parameters, and Rosetta parameters with no explicit mode become VHDL `inout` parameters. The types of these parameters must be one of the types given earlier with one exception, design parameters are allowed to be of type `boolean`, `string`, `real`, or `natural` in addition to those described earlier. The reason for this exception is that the expanded type support is necessary for the

Table 4.3: Supported Expressions

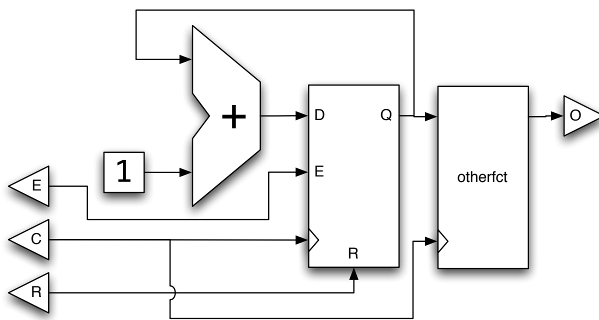
```
var  :  Rosetta variable
lit  :  [0..9]* | bottom
seq  :  b" (0|1)* "
     |  o" [0..7]* "
     |  x" [A..F,0..9]* "
pre  :  % | not
inf  :  + | - | * | / | & | =
     |  == | /= | > | < | >=
     |  <= | and | or | xor
     |  nand | nor | xnor
idx  :  <lit> , <lit>+
     |  <lit>,..<lit>
grd  :  <exp> | <var>'event
     |  <var>'event and <var>=1
     |  <var>'event and <var>=0
exp  :  <lit> | <seq>
     |  <pre> <exp>
     |  <exp> <inf> <exp>
     |  <var> ( <exp> )
     |  <exp> sub [ <idx> ]
     |  if <grd> then <exp>
        else <exp> end if
```

correct modeling of black-box Xilinx primitives. These primitives make use of the expanded types to defined generic configuration parameters.

The final two requirements are that all declarations in the facet must be of a supported type and all terms in the facet must be a facet instantiation, facet interface instantiation, library definition, library use, or equality between a “simple” variable and a synthesizable expression. Support for facet and facet interface instantiations provides the structural design capability that hardware engineers expect with facet interface instantiation providing “black box” component support. This allows use of vendor IP cores and hardware cores constructed in other languages.

Library definitions and library uses are special constructs used to allow the notion of libraries and library uses to be imported into a facet’s implementation. These are usually used to support “black box” entities instantiated within a Rosetta design. These entities are typically defined in an IP library that must be imported and used during implementation.

Figure 4.1: Example HW Circuit



Support for equalities provides the behavioral synthesis capability that has become common in the hardware design community. Synthesizable behavioral terms in the synthesizable Rosetta subset must have a left hand side (LHS) that is “simple”, meaning the LHS is a declaration, output port, or is the next state operator applied to either a declaration or output port.

The right hand side (RHS) of a behavioral term must be one of the synthesizable expressions given by the grammar in table 4.3. The expressions defined by this grammar support all of the implementation-level logic gates required to construct efficient hardware circuits and includes constant values, boolean logic gates, arithmetic gates, comparator gates, bit slicing, concatenation, multiplexers, and registers.

Listing 4.1: Synthesizable Facet

```
facet exfct(ITV :: design word(8); clk :: input bit; rst :: input bit;
            ena :: input bit;      ovl :: output word(8)) :: state_based is
    acc :: word(8); res :: word(16);
begin
    ovl  = res sub [7,..0];
    acc' = if clk'event and clk == 1 then
             if rst == 1 then ITV
             elseif ena == 1 then acc + 1
             else acc end if
         else acc end if;

    inst: otherfct(clk, acc, res);
end facet exfct;
```

As an example, a synthesizable Rosetta facet is given in listing 4.1. In this example, an accumulator, `acc`, is constructed using the clock signal `clk`, the reset signal `rst`, and the enable signal `ena`. The clock signal and the value of this accumulator are supplied as inputs to facet instantiation `inst` and the value `res` is driven by the output of `inst`. The output value `ovl` of the facet is then driven by the lower eight bits of the value `res`.

The hardware circuit that is produced by RST is shown in figure 4.1. Comparing the example facet's code to the resulting hardware circuit, we see that every element in the code is reflected in a simple, understandable way in the circuit. The input and output parameters of the facet become input and outputs of the circuit, the accumulator described by `v` becomes a circuit level accumulator using an adder and register and the facet instantiation becomes an entity in the circuit.

In addition to the synthesizable functional behaviors described above, RST supports a rich set of synthesis attributes that can be used to provide additional information to the Xilinx synthesis tools. Listing 4.2 shows an example of how synthesis attributes can be used to direct the Xilinx synthesis tools. In this case, a DCM is instantiation to provide a stable clock, `clk`, from a reference clock, `refclk`. The three attributes tell the Xilinx synthesis tool that `clk` is a clock signal, that `refclk` is the parent clock signal from which `clk` is generated, and that `clk`

Listing 4.2: Synthesis Attributes

```
1 facet genclk2x(refclk :: input bit; clk :: output bit) :: state_based is
2 begin
3     idcm: DCM(...,refclk,...,clk,...);
4     clk'sigis = "CLK"; clk'clk_inport = refclk; clk'period = 10*ns;
5 end facet exfct;
```

has a 10 nanosecond period. Synthesis attributes are derived directly from those supported by the Xilinx synthesis tools and are listed in appendix A.

RTL Representation

The first phase in the process of transforming synthesizable Rosetta code into VHDL is the conversion of Rosetta code into an RTL intermediate form. The RTL intermediate form is a list of RTL nodes as given by table 4.4. In this representation, connections between nodes are implicit with the inputs to each node stored as references.

The conversion process is performed individually on each facet in five basic steps: conversion of parameters, conversion of declarations, conversion of behavioral terms, conversion of structural terms, and conversion of synthesis attributes. Because each facet is processed separately from the rest of the system, the result of synthesizing an entire system is a list of RTL intermediate forms, one for each facet.

Representation of Types

During the first two steps, conversion of parameters and declarations into RTL intermediate form, Rosetta types must be converted into the appropriate RTL intermediate form. In the first and most common case, Rosetta types are converted into a simple integer bit width that is stored inside an RTL node and used to determine the number of output bits produced by that node. Table 4.5 shows the supported Rosetta types for this case and their corresponding bit widths.

One special case is the Rosetta type `array(D,word(n))`. This type can only be used as part of a Rosetta variable declaration and is used to model synthesizable RAMs in the Rosetta language.

Table 4.4: RTL Nodes

		RTL Circuit Nodes			
`RTLE`	-	Logic gate	`RTLI`	-	Input port
`RTLO`	-	Output port	`RTLIO`	-	Input/output port
`RTLG`	-	Generic value	`RTLN`	-	Named value
`RTLD`	-	Entity output port	`RTLL`	-	Library definition
`RTLU`	-	Library use	`RTLB`	-	RAM storage
`RTLBR`	-	RAM read	`RTLBW`	-	RAM write
		RTLE Logic Gate Types			
`Cons`	-	Constant value	`Unary`	-	Unary gate
`Binary`	-	Binary gate	`Select`	-	Mux Gate
`Split`	-	Slicing gate	`Entity`	-	Entity instantiation
		Const Value Types			
`Signed`	-	Signed value	`Unsigned`	-	Unsigned value
`Integer`	-	Integer value	`Natural`	-	Natural value
`String`	-	String value	`Boolean`	-	Boolean value
`Real`	-	Real value	`Derived`	-	Derived value
		Unary Gate Types			
`Not`	-	Not			
		Binary Gate Types			
`Add`	-	Addition	`Sub`	-	Subtraction
`Mul`	-	Multiplication	`Div`	-	Division
`Mod`	-	Modulo	`Rem`	-	Remainder
`Concat`	-	Concatenation	`And`	-	Bitwise And
`Or`	-	Bitwise Or	`Xor`	-	Bitwise Xor
`Nand`	-	Bitwise Nand	`Nor`	-	Bitwise Nor
`Xnor`	-	Bitwise Xnor			
		Select Test Types			
`Eq`	-	Equality	`Ne`	-	Inequality
`Gt`	-	Greater than	`Lt`	-	Less than
`Ge`	-	Greater than or equal	`Le`	-	Less than or equal
`Rs`	-	Rising edge	`Fl`	-	Falling edge
`Dl`	-	DDR edge	`Id`	-	Identity

Requiring the type to be used as part of a variable declaration ensures that RAMs are not used for facet I/O parameters, an unsynthesizable construct. The storage requirements for this type would be $D \times N$ bits but the RTL representation is only N. This is because types at the Rosetta level are represented as output bit widths in the RTL intermediate form. In this case, the extra depth parameter D is modeled separately by the RTLB node and combined with the output width to determine the total storage requirements for the RAM.

Table 4.5: RTL Types

Type	Width
integer	0
bit	1
word(N)	N
array(N,bit)	N
array(D,word(N))	N

The second type representation in the RTL intermediate form only applies to facet parameters using the design mode. In this case an expanded set of types are supported so that synthesizable Rosetta designs can more easily integrate with other synthesis tools. In addition to the bit-vector like types already supported, design parameters can have one of the extra types string, natural, boolean, or real. Each of these extra types is modeled naturally as their type indicates.

The use of design parameters inside of behavioral and structural terms in the synthesizable subset of the Rosetta language is restricted to the bit-vector like types given in table 4.5. This limits the use of the extra types allowed for design parameters to simply passing arguments to other design parameters during structural instantiation of a facet. Effectively, the extra types are useful only when instantiating a facet interface that is a hardware component implemented outside of the Rosetta language.

For example, A Xilinx Digital Clock Multiplier (DCM) can be modeled in the synthesizable subset of Rosetta as a facet interface that takes several design parameters that make use of the extra type support. An instantiation of this facet interface is supported in the synthesizable subset of Rosetta and will eventually lead to an implementation that contains the DCM with its corresponding design parameters.

Representation of Parameters

Lowering facet parameters into RTL intermediate form is accomplished by producing an RTL node that corresponds to the parameter's mode. As given in table 4.2, a parameter's mode must either be missing, `input`, `output`, or `design` for a facet to be synthesizable. The mapping of synthesizable parameter modes into RTL nodes representations is given by table 4.6

When lowering a parameter with mode `input`, RST creates an `RTLI` node with *size* defined by the parameter's type and *name* defined as the parameters name. The lowering of parameters with no mode or mode `output` is similar with `RTLIO` and `RTLO` nodes being produced respectively. The main difference is that `RTLO` and `RTLIO` nodes store an additional *ref* that is a reference to the node whose value is used for the output.

Table 4.6: RTL Parameters

Mode	RTL Node
`input`	`RTLI` *size name*
`output`	`RTLO` *size name ref*
	`RTLIO` *size name ref*
`design`	`RTLG` *size type*

Parameters using the `design` mode are considered design time constants and are lowered into `RTLG` with *size* and *type* derived from the parameter's type. The size of the design parameter represents the number of bits required to represent the parameter at the implementation level and is undefined for design parameters that use one of the extra types not supported by behavioral terms. The *type* representation for the node is a simple type encoding supporting all of the extra types that can be used for design parameters.

As an example, the facet shown in listing 4.1 has five facet parameters. These parameters would be converted into the following five RTL intermediate form nodes:

1. `ITV` :: `design word(8)` → `RTLG 8 (GenericBitVector 8)`
2. `clk` :: `input bit` → `RTLI 1 "clk"`
3. `rst` :: `input bit` → `RTLI 1 "rst"`
4. `ena` :: `input bit` → `RTLI 1 "ena"`
5. `ovl` :: `output bit` → `RTLO 1 "ovl" REF`

For the output node `ovl`, the value `REF` in the produced `RTLO` would be a reference to the node that was the result of synthesizing the Rosetta expression "`res sub [7,..0]`".

Representation of Declarations

Lowering facet declarations into RTL intermediate form is accomplished by producing either a named RTL node (`RTLN`) or an RTL RAM storage node (`RTLB`[1]). `RTLN` nodes are created for any declaration that is not of type `array(D,word(N))` while `RTLB` nodes are created for that type.

When an `RTLN` node is produced for a declaration the size and name of the declaration are stored inside of the node much like RTL input nodes (`RTLI`). `RTLB` nodes are very similar in that they store the size and name of the declaration as well. The main difference is that `RTLB` nodes store an additional value `D` that is the depth of the RAM that the node represents. It is combined with the size to determine the storage requirements for the `RTLB` node.

Declarations in the Rosetta language can also be given a constant value. In this case the two declaration nodes, `RTLN` and `RTLB`, store an extra value. `RTLN` nodes store an extra bit-vector like value that is the constant value of the declaration in bit-vector form while `RTLB` nodes store an array of bit-vector like values of size `D`.

Representation of Behavioral Terms

The lowering of a facet's behavioral terms into RTL intermediate form is the largest task faced during the Rosetta to RTL conversion process. During this stage of the conversion process, RST visits all of a facet's behavioral terms one at a time and attempts to convert them into an RTL representation.

As described in section 4.1, for a facet to be synthesizable the behavioral terms in that facet must be one of three special forms:

1. `add_library("library name");`
2. `add_use("library use");`
3. `lhs = rhs;`

[1]The acronym RTLB stands for RTL Block RAM and was chosen when RST only supported Block-RAMs. The current implementation supports Distributed-RAMs and Block-RAMs so the acronym is now a bit of a misnomer.

The first two forms support the notion of Vendor IP libraries and uses in the VHDL implementation of the facet. For example, in figure 4.3 the facet `libex` has one library definition term and one library use term. Together these two terms allow access to the Xilinx UNISIM IP library that contains the `DCM` entity. The process of lowering these first two special forms is simple. For every `add_library` term found a corresponding `RTLL` node is created containing the name of the library to define. Likewise, for every `add_use` term found a corresponding `RTLU` node is created containing the required use clause.

Listing 4.3: Library Terms

```
1  facet interface DCM(
2        ...
3      ) :: static
4  end facet interface;
5
6  facet libex :: static is
7  begin
8    add_library("unisim");
9    add_use("unisim.all");
10   idcm: DCM(...);
11 end facet;
```

The third form supported by RST is the main mechanism by which functional behavioral is defined. In this form two Rosetta expressions are defined to be equivalent. The left hand side (LHS) of this equality must be an output parameter, a declaration, or an index into a RAM declaration. Listing 4.4 gives an example of each of the allowed LHS expressions with line 10 showing a declaration, line 12 showing a RAM index, and line 22 showing an output parameter. The declaration and RAM index LHS forms can also have an optional tick postfix operator applied. In the Rosetta `state_based` domain, the postfix tick operator is used to mean "next state". When used in conjunction with one of the clock generating expressions described below, this allows the description of stateful registers.

Listing 4.4: Behavioral Terms

```
1  facet behex(c::input bit;
2              e::input bit;
3              a::input word(8);
4              b::output word(32))
5               ::state_based is
6      i :: word(9);
7      r :: array(512, word(32));
8      v :: word(32);
9  begin
10     i = e & a;
11
12     r(i)' = if c'event and c==1
13             then x"10010004"
14             else r(i)
15             end if;
16
17     v = if c'event and c==1
18         then r(i);
19         else v
20         end if;
21
22     b = 3 * v + 1;
23 end facet;
```

The right hand side (RHS) of the behavioral term must be a synthesizable expression as given in table 4.3. When synthesizing the behavioral term, RST first converts the RHS expression into an RTL intermediate form representation. The result of this process is an RTL node that represents the value of the expression. RST will then perform one of three actions depending on the LHS expression.

If the LHS expression is an output parameter then the RTL node corresponding to that parameter is found. The found node is then updated so that *ref* is a reference to the RTL node produced by the RHS. The effect is that the output parameter is connected to the value produced by the expression that it is equivalent to.

If the LHS expression is a declaration then the RTL node corresponding to that declaration is found. This node is then updated by replacing the contents of the found node with the contents of the node produced by the RHS expression. All references to the found node remain valid so that any connections between the found node an other nodes remain valid.

If the LHS expression is a RAM index then a `RTLBW` node is created. This node stores a reference to the RAM being indexed into, a reference to the index being used, and a reference to the RTL node produced by the RHS. As an example, the RAM index on line 12 of listing 4.4 would produce a RTL node `RTLBW (REF “r”) (REF “i”) (REF rhs)` where `rhs` is a reference to the RHS expression on line 12.

The conversion of the RHS expression into RTL intermediate form is a straight forward conversion of the synthesizable expressions ,as defined in table 4.3, into one of the RTL intermediate form nodes given in table 4.4. The conversion is shown in table 4.7. This table shows the eight major node types produced by RST when synthesizing the RHS of an expression. Except for RAM nodes, each node created from a RHS expression is a `RTLE` node that stores an operation and a set of operand references. The major nodes types are constructed as follows:

1. All constant value constructors in the Rosetta language are all reduced to a `Const` node.
2. Indexing into a Rosetta `sequence` or selecting a subset of bits using the `sub` operator are both converted into a bit slicing node.

Table 4.7: Rosetta Expressions as RTL Nodes

Rosetta		RTL	Rosetta		RTL
Constant Nodes					
$[0..9]*$	$\rightarrow$	RTLE Const v	b" ... "	$\rightarrow$	RTLE Const v
x" ... "	$\rightarrow$	RTLE Const v	o" ... "	$\rightarrow$	RTLE Const v
Bit Slicing Nodes					
$x(i)$	$\rightarrow$	RTLE Split x i i	x sub $[h,..l]$	$\rightarrow$	RTLE Split x h l
Mux Nodes					
if g1 then x elseif g2 then y else z end if	$\rightarrow$	RTLE Select [g1,g2] [x,y] z			
RAM Nodes					
$ram(i)$	$\rightarrow$	RTLBR ram i			
Unary Nodes					
not x	$\rightarrow$	RTLE Not x			
Binary Nodes					
$x+y$	$\rightarrow$	RTLE Add x y	$x-y$	$\rightarrow$	RTLE Sub x y
$x*y$	$\rightarrow$	RTLE Mul x y	x/y	$\rightarrow$	RTLE Div x y
$x\&y$	$\rightarrow$	RTLE Concat x y	x and y	$\rightarrow$	RTLE And x y
x or y	$\rightarrow$	RTLE Or x y	x xor y	$\rightarrow$	RTLE Xor x y
x nand y	$\rightarrow$	RTLE Nand x y	x nor y	$\rightarrow$	RTLE Nor x y
x xnor y	$\rightarrow$	RTLE Xnor x y			
Comparison Nodes					
$x==y$	$\rightarrow$	RTLE Eq x y	$x/=y$	$\rightarrow$	RTLE Ne x y
$x>y$	$\rightarrow$	RTLE Gt x y	$x<y$	$\rightarrow$	RTLE Lt x y
$x>=y$	$\rightarrow$	RTLE Ge x y	$x<=y$	$\rightarrow$	RTLE Le x y
Clock Nodes					
$x'event$ and $x==1$	$\rightarrow$	RTLE Rs x	$x'event$	$\rightarrow$	RTLE Dl x
$x'event$ and $x==0$	$\rightarrow$	RTLE Fl x			

3. All Rosetta if statements are converted into mux operations that have one operand reference for each guard, one operand reference for each clause, and a single operand reference for the default value from the `else` part of the if statement.
4. Indexing into a RAM is converted into a `RTLBR` node that contains a reference to the RAM being read and to an operand that is the index being read.
5. The Rosetta unary operation `not` is turned into a unary `Not` operation in the RTL intermediate form.
6. All Rosetta binary operations are converted into a corresponding binary operation in the RTL intermediate form.
7. All Rosetta comparison operations are converted into their corresponding comparison operation in the RTL intermediate form.
8. The Rosetta `'event` attribute is turned into a clock node in the RTL intermediate form. If the attribute is paired with an equality to `1` then a rising clock is generated and if it is paired with an equality to `0` then a falling clock is generated. If the attribute is not paired with any equality then a DDR clock is generated.

Representation of Structural Terms

The lowering of structural instantiations into RTL intermediate form is accomplished by generating two types of RTL nodes for every instantiation. First, an `Entity` gate is constructed inside of a `RTLE` node to represent the instantiation as a “black box” entity. To form this gate correctly, RST must be able to determine the parameters required by the entity via a facet or a facet interface declaration. These parameters are stored in the `Entity` so that the name, mode, and type of each parameters in the entity can be determined during the rest of the synthesis process.

Once the parameters for the instantiation are found, they are hooked up to their corresponding instantiation arguments. Design arguments and input arguments are stored in the `Entity` gate just like all other input connections are stored in the RTL intermediate form, as a list of references to the node whose output is used as the input to the entity.

Outputs from the instantiation produce the second type of RTL node. For each output node in the instantiated entity as single `RTLD` node is produced to represent the output. Each output is required to have its own node in the RTL intermediate form because all connections from the output of one node to the input of another node are stored as simple references to the node producing the output. Thus, each output in the entity instantiation needs to have its own node in the graph so that it can be referenced unambiguously by other nodes.

Each `RTLD` node stores a reference to the `RTLE` node created for the `Entity` gate, a size in bits, and the name of the output parameter that the `RTLD` node corresponds to in the entity instantiation. This information is used by RST in later stages to connect the `RTLD` node back to the correct output port on the entity instantiation that generated it.

As an example, the facet shown in listing 4.1 has a single facet instantiation, `inst`, with two input parameters and one output parameter. This term would be converted into RTL intermediate form as follows:

```
inst: otherfct(clk, acc, res);
→
RTLE (Entity "otherfct" [REF "clk", REF "acc"])
RTLD 16 (REF "otherfct") "otherfct_res"
```

The `RTLD` node that is created during the processing of the instantiation would be the node referenced whenever the variable `res` is used in other parts of the facets.

Representation of Synthesis Attributes

During both behavioral synthesis and structural synthesis, RTL intermediate form nodes are created as low-level representations of the specification being synthesized. The final step performed by RST is the extraction and processing of low-level synthesis attributes. This produces a table of attributes that are used to produce synthesis constraints for the Xilinx synthesis tool.

The semantics of each synthesis attribute is enforced outside of RST. As such, synthesis attributes are represented in the Rosetta language using Rosetta attribute constructs connected to either declarations, instantiations, or parameters. When RST processes the synthesis at-

tributes an appropriate Xilinx synthesis constraint is generated containing a reference to the correct declaration, instantiation, or parameter node. A list of supported synthesis constraints can be found in appendix A.

Graph Representation

The second phase performed when lowering synthesizable Rosetta specifications into hardware circuits is to convert the RTL intermediate form into graph based representation that has explicit connections between nodes. The process of lowering Rosetta specifications into graph based form is done in two steps so that each of the steps is more easily accomplished. The RTL intermediate form produced in the first step represents the hardware circuit as a set of RTL intermediate form nodes with connections between nodes represented as references stored in the consuming node and referencing the producing node. This intermediate form is easy to produce from the Rosetta language because the language represents connections between nodes as references to intermediary variables. However, examining, canonicalizing, and optimizing the RTL intermediate form directly is more difficult than necessary. Because of this, the RTL intermediate form is converted into a explicit graph representation before processing is done.

The process of converting the RTL intermediate form into the explicit graph based representation is relatively straightforward. The graph representation is stored as a traditional multi-digraph, $G = (V, E)$ with loops allowed in the graph. The vertices V in the graph are labelled with one of the operations in table 4.8 and determine the operation that the node applies to its incoming edges.

The edges E in the graph are labelled with a three tuple (F, T, S) with F and T representing the port on the source node and the port on the target node. S represents the bit-width of the edge. Each edge is labelled with source and target ports to allow the ordering of incoming and outgoing edges so that non-commutative nodes such as `GSub` and `GEnt` can correctly order their operands. Additionally, the bit-width of the edge is maintained to aid in graph processing and VHDL conversion.

Table 4.8: Graph Nodes

RTL Circuit Nodes					
GIn	-	Input port	GOut	-	Output port
GInOut	-	Input/output port	GGen	-	Generic value
GNam	-	Named value	GLib	-	Library definition
GUse	-	Library use	GVal	-	Constant value
GRam	-	RAM storage	GBram	-	Block-RAM storage
GReg	-	Register	GMux	-	Mux Gate
GIdx	-	Slicing gate	GEnt	-	Entity instantiation
GAdd	-	Addition	GSub	-	Subtraction
GMul	-	Multiplication	GDiv	-	Division
GMod	-	Modulo	GRem	-	Remainder
GCon	-	Concatenation	GNot	-	Not
GAnd	-	Bitwise And	GOr	-	Bitwise Or
GXor	-	Bitwise Xor	GNand	-	Bitwise Nand
GNor	-	Bitwise Nor	GXnor	-	Bitwise Xnor
GEq	-	Equality	GNe	-	Inequality
GGt	-	Greater than	GLt	-	Less than
GGe	-	Greater than or equal	GLe	-	Less than or equal
GRs	-	Rising edge	GFl	-	Falling edge
GDl	-	DDR edge			
Const Value Types					
Signed	-	Signed value	Unsigned	-	Unsigned value
Integer	-	Integer value	Natural	-	Natural value
String	-	String value	Boolean	-	Boolean value
Real	-	Real value	Derived	-	Derived value

Most of the nodes in the RTL intermediate form have vertices in the explicit graph representation with identical form except that references are not stored in the nodes but are instead converted into graph edges from the referenced node to the referencing node. The conversion of the RTL intermediate form to graph representation starts by selecting an unprocessed node from the RTL intermediate form, in no particular order. A corresponding vertex will then be created and added to the graph. For example, if a `RTLE Add x y` node is selected then a `GAdd` vertex will be created and added to the graph.

All of the RTL intermediate form node references in the selected node will then be converted into graph edges. There are two cases that need to be handled. First, if the reference is to a RTL intermediate form node that has already been processed then the vertex corresponding to

that node is selected as the source vertex SV and the node currently being processed is selected as the target vertex TV. Second, if the reference is to an unprocessed RTL intermediate form node then that node is converted into its corresponding graph representation and the resulting vertex is used as the source vertex SV with the target vertex TV being the vertex from the node being processed. Note that the process will correctly deal with loops in the resulting graph because a vertex for an RTL node is created before any references are resolved.

After the source vertex SV and target vertex TV have been determine, the label for the new edge must be determined by figuring out the source port F and target port T. The target port is determined by examining the order of the references in the RTL intermediate form node. For instance, in a binary node the first reference would have a target port of `0` and the second reference would have a target port of `1`. Determining the source port F involves examining the referenced node. If the referenced RTL intermediate form node is a `RTLD`, `RTLBW`, or `RTLBR` node then the source port is determined either by finding the correct output port from the `RTLD` node's associated entity using the port name stored in the `RTLD` node or by creating a new port on the `GRam` vertex associated with the RAM being read or written. If the RTL intermediate form node is not one of these then the source port is `0` because all other nodes have only a single output. After the source and target vertices have been determined and the source and target ports have been determined, a new edges (SV, TV) is created an added to the graph with the label $(F, T, 0)$. With the bit-width `0` indicates that the width of the edge is unknown.

`RTLD`, `RTLBR`, and `RTLBW` nodes are treated differently than other nodes when converting the RTL intermediate form representation into the explicit graph representation because the purpose of these nodes is to represent the input or output ports of another node. As such, these nodes do not create new vertices in the graph but are instead used to determine the correct port to use when creating edges, as described above.

Determining the Bit-Width of Graph Edges

The initial width given to each edge in the graph is `0` to indicate that the width is unknown. After all of the nodes in the RTL intermediate form have been processed, the explicit graph

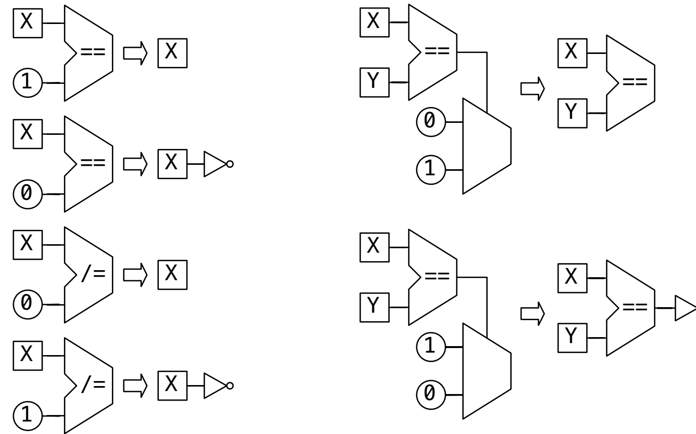
based representation is complete with all vertices and edges in the graph resolved correctly. At this point, the bit width of each edge is determined by visiting each vertex in the graph in topological order. When visiting each node, a simple algorithm validates the bit-width of all incoming edges and then determines the output bit-width by combining the information about the incoming widths with the node's label. Because vertices in the graph are visited in topological order, the algorithm to determine the width of each edge propagates information from input parameters, declarations, and constants to output parameters and other declarations. `GIn`, `GOut`, `GInOut`, `GNam`, and `GVal` vertices are decorated with their declared width by examining the RTL node from which the vertex was created. This provides the initial bit-width information that is propagated though the graph.

As the algorithm visits each vertex in the graph if first ensures that the incoming edges have appropriate widths. For instance, when visiting `GOut` or `GNam` vertices, the algorithm ensures that the incoming edges have the same width as the vertex's decorated width, ensuring that the equality in the original specification was an equality between bit-vectors of equivalent size. Once the width of each incoming edge has been validated, the width of each outgoing edge is assigned as appropriate. For instance, the width of the outgoing edge of a `GAdd` vertex is the same as all incoming edges while the width of a `GCon` vertex's output edge is the sum of all incoming edges. Upon successful completion, the complete graph based representation has been formed with vertices in the graph corresponding directly to the parameters, declarations, instantiations, and behavioral expressions in the original specification and edges in the graph representing either structural composition or equalities from behavioral terms.

Graph Canonicalization

The third phase performed in the process of synthesizing Rosetta specifications to hardware circuits is the canonicalization of the explicit graph representation. After the complete graph representation has been formed, a series of canonicalization and resource discovery transformations are performed on the graph to prepare the graph for conversion into a hardware circuit. The purpose of these transformations is not necessarily to optimize the hardware circuit.

Figure 4.2: Comparator Canonicalization



The main purpose is to discover resources described in the Rosetta specification that are not discovered by the lower-level hardware synthesis tool (in this case the Xilinx synthesis tool) when producing the naïve description of the circuit directly from the untransformed graph.

Canonicalization of Comparators

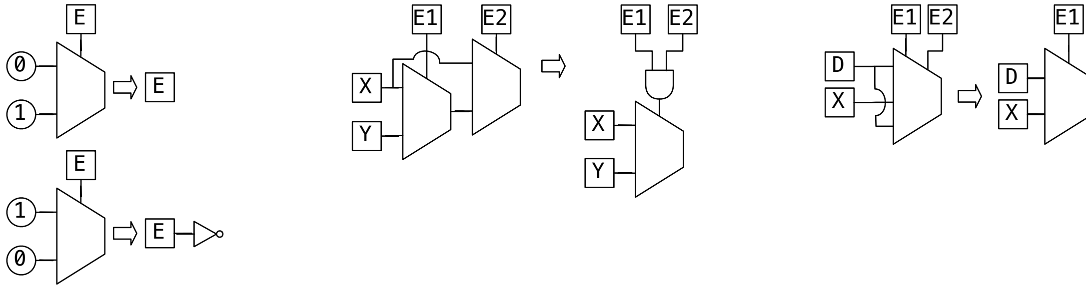
The first transformation performed on the graph representation is the canonicalization of comparators. Figure 4.2 shows the four transformations that are applied to the graph. Each of these four transformations only applies to vertices whose incoming edges have a bit-width of `1`, i.e. these transformation are only applied to bits not bit-vectors.

Table 4.9

X	1	X == 1
0	1	0
1	1	1

The first transformation eliminates comparators or replaces them with simpler `GNot` gates. If the output of a graph vertex `X` is compared for equality against the constant value `1` then we get the truth table shown in figure 4.9. As shown in the table, the output value from the equality comparison is the same as the output value from the vertex `X`. Thus, the equality comparator can be removed from the graph and any outgoing edges can be replaced by outgoing edges from the vertex `X`. There are three similar forms, as shown in figure 4.2, with two of the forms transforming the comparator into a `GNot` gate instead of eliminating it.

Figure 4.3: Multiplexer Canonicalization



The second transformation eliminates multiplexers or replaces them with simpler GNot gates. Like the first transformation, these two transformations apply only to bits, not bit-vectors. This transformation eliminates multiplexers whole selector input is a comparator and whose input values are the constants 0 and 1. The truth table for this circuit is shown in table 4.10. Like the first transformation, the truth table shows that the output of the circuit is the same as the output of the comparator so the multiplexer can be removed. In the second case the output of the circuit is the complement of the output of the comparator and so the multiplexer is transformed into a GNot gate.

Table 4.10

X == Y	0	1	MUX
0	0	1	0
1	0	1	1

The comparator canonicalization transformation is the only graph transformation applied multiple times by RST. This transformation is applied as the first transformation and is also applied after every other canonicalization and resource discovery transformation because these other transformations tend to produce the degenerate comparators eliminated by these transformation.

Canonicalization of Multiplexers

The canonicalization of multiplexers in the graph eliminates or simplifies multiplexers. Figure 4.3 shows the four basic multiplexer transformation that are applied. The first two transformations are a generalization of some comparator transformations described earlier. In these cases the output of the multiplexer is the sames as either the selector input or the complement of the selector so the multiplexer is eliminated or replaced with a GNot gate.

The second transformation replaces two multiplexers that are chained together with a single multiplexer that has an expanded selector input. The truth table for this circuit can be seen in table 4.11. This table shows that the output of the given circuit is only the output from the gate `Y` when both selector inputs are `1`. Thus, the circuit is replaced by a simpler circuit containing only a single multiplexer with the selector input to the new multiplexer a `GAnd` gate of the two original selector inputs.

The last transformation applied during multiplexer canonicalization multiplexers with inputs identical to the default multiplexer value. Multiplexers in the graph representation are slightly different than hardware multiplexer circuits. Multiplexers in the graph representation have multiple inputs with each input having its own selector. If the selector for an input is `1` then that value is output as the output of the multiplexer. When multiple selector inputs are `1` then the output of the multiplexer is resolved by assigning a priority to the selector inputs, in the figures shown the priority is represented by having inputs with higher priority be above inputs with lower priority. If the multiplexer has no selector inputs with a value of `1` then the default input is used as the output.

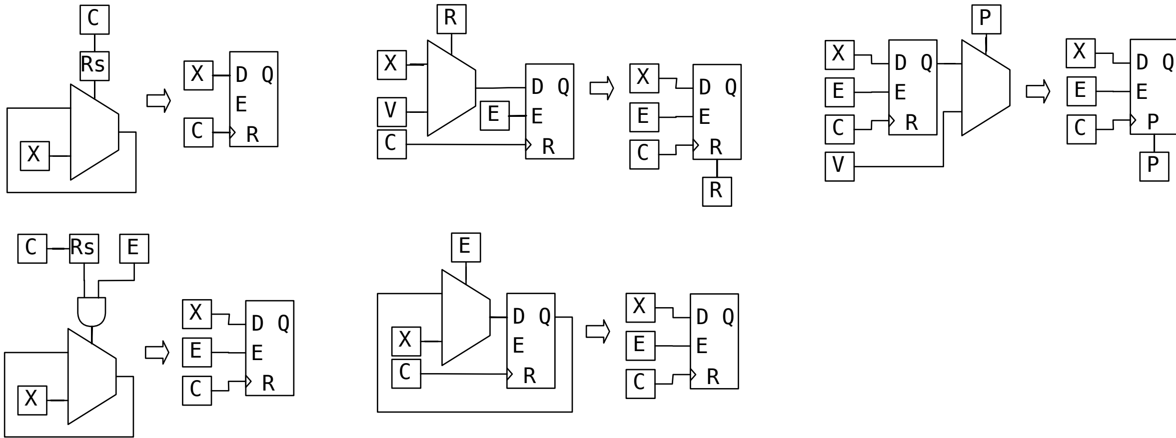
Table 4.11

E1	E2	RES
0	0	X
0	1	X
1	0	X
1	1	Y

The semantics of the multiplexer in the graph representation were selected because they match the semantics of the Rosetta language constructs that create multiplexers. If expressions in the Rosetta language can have multiple guard expressions each with an associated value when that guard is true. The guard statements are prioritized by the order in which they appear in the if expression and a default value is given by the else part of the expression to handle the case where no guard expressions are true.

The last transformation applied during multiplexer canonicalization eliminates inputs, and their associated selector, when that input is the same as the default input and no other selectors have lower priority. This maintains the priority encoded by if expressions at the Rosetta level but simplifies multiplexers by reducing the number of inputs they have.

Figure 4.4: Register Canonicalization



Register Discovery

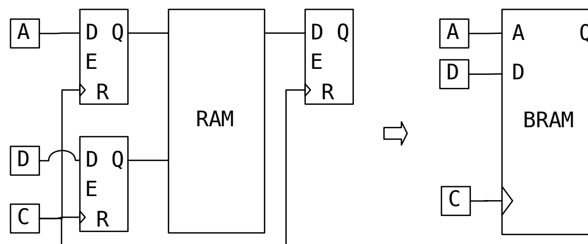
After the two canonicalization transformations have been performed, the graph representation has been sanitized so that comparators and multiplexers are in well-known form. At this point, a register discovery transformation is performed on the graph.

All of the vertices in the graph, before transformations have been performed, directly correspond to a syntactic construct in the original Rosetta specification. For instance, any `GAdd` node could be traced back to the exact syntactic + operator that it corresponds to. Thus, the process of converting Rosetta specifications into the graph representation has been syntax directed up to this point.

At this point RST attempts to determine if the semantics of some parts of the original Rosetta specification can be modeled in hardware using registers. This is done by looking for subgraphs that contain `GRs`, `GFl`, or `GDl` vertices that are connected as selectors to multiplexers. The three different vertex labels are those produced by the `'event` attributes used to as clocking expressions in Rosetta specifications.

The most basic register discovery transformation looks for clocking vertices that are used as selectors to a multiplexer. If this multiplexer's default input is connected to its own input, i.e. the vertex forms a self loop, then the semantics of the hardware circuit is that the output value

Figure 4.5: RAM Canonicalization



only changes on a clock edge. These semantics directly match those of a hardware register and so the multiplexer is replaced with a register.

The remaining four transformations performed during register discovery attempt to find enable signals and reset signals applied to registers. If these signals can be found in the graph then they are eliminated and hooked up directly to the register to ensure the register description given to the lower-level hardware synthesis tool produces the correct register resource. If any `GRs`, `GFl`, or `GDl` vertices remain in the graph after this transformation has completed then the specification cannot be synthesized because it contains a clocking expression that cannot be realized in hardware.

Block-RAM Discovery

After register discovery has completed, RST performs a Block-RAM (BRAM) discovery transformation on the graph. During the initial process of converting the RTL intermediate form representation into the explicit graph representation, all `RTLBR`, `RTLBW`, and `RTLB` nodes are combined to form a single `GRam` vertex with the correct number of read and write ports.

The BRAM discovery transformation attempts to determine if a `GRam` vertex in the graph can be changed into a `GBRam` vertex. The semantics of a `GRam` vertex are that the value of each read port is the value in the RAM at the associated read index and that any value appearing on any write port is used to update the RAM contents within a shorter period of time than the fastest clock in the system (i.e. RAM reads and writes appear instantaneous to synchronous elements).

`GBram` vertices are used to model the semantics of Xilinx BRAMs. BRAMs require that all read indexes, all write ports, and all read ports are registered. Because all `GRam` reads and writes look instantaneous to synchronous elements in the graph representation, this corresponds to `GRam` vertices where all incoming edges are from registers and all outgoing edges go to registers. As shown in figure 4.5, the BRAM discovery transformations looks for `GRam` vertices whose incoming and outgoing edges match these constraints. Any `GRam` vertex that matches the constraints is transformed into a `GBram` vertex and the registers associated with the `GRam` vertex are then removed.

VHDL Production

The fourth and final phase in the process of converting synthesizable Rosetta specifications into hardware circuits is the production of VHDL from the explicit graph representation. To produce the VHDL representation, the explicit graph based representation is traversed in topological order and each vertex in the graph is used to build up a data structure that stores the information needed to construct the final VHDL document. Table 4.12 shows the five different kinds of information stored in the data structure and gives examples of the kind of information stored.

After the explicit graph representation has been converted into a VHDL data structure representation the VHDL output is created by assembling the separate pieces in the data structure into complete VHDL document. First, the library and use statements are used to generate a VHDL header for the document and then the information about generic and I/O parameters are used to construct a VHDL entity declaration. Afterwards, the signal and RAM declarations are used to create a new VHDL architecture with the terms and instantiations being used to fill out the body of the architecture.

The VHDL output process also supports emitting VHDL attributes to support the synthesis constraints listed in section A. The VHDL representation of these constraints follows the Xilinx recommendations and are processed correctly when using the Xilinx synthesis tools to convert the VHDL representation into a hardware circuit for use on Xilinx FPGAs.

Table 4.12: Conversion of Explicit Graph into VHDL

Graph		VHDL	Graph		VHDL
Generics and I/O					
`GIn n`	$\rightarrow$	`n : in std_logic`	`GOut n`	$\rightarrow$	`n : out std_logic`
`GInOut n`	$\rightarrow$	`n : inout std_logic`	`GGen n`	$\rightarrow$	`n : natural`
Libraries and Uses					
`GLib l`	$\rightarrow$	`library l`	`GUse u`	$\rightarrow$	`use u`
Signals and RAMs					
`GRam n`	$\rightarrow$	`type r is array(...) of std_logic_vector(...)` `signal n : r`			
`GBram n`	$\rightarrow$	`type b is array(...) of std_logic_vector(...)` `signal n : b`			
`GNam n`	$\rightarrow$	`signal n : std_logic`			
`GNam n v`	$\rightarrow$	`constant n : std_logic_vector(...) := v`			
Terms					
`GAdd`	$\rightarrow$	$o \Leftarrow x+y$	`GSub`	$\rightarrow$	$o \Leftarrow x-y$
`GMul`	$\rightarrow$	$o \Leftarrow x*y$	`GDiv`	$\rightarrow$	$o \Leftarrow x/y$
`GMod`	$\rightarrow$	$o \Leftarrow x\%y$	`GRem`	$\rightarrow$	$o \Leftarrow x \texttt{ rem } y$
`GCon`	$\rightarrow$	$o \Leftarrow x\&y$	`GNot`	$\rightarrow$	$o \Leftarrow \texttt{ not } x$
`GAnd`	$\rightarrow$	$o \Leftarrow x \texttt{ and } y$	`GOr`	$\rightarrow$	$o \Leftarrow x \texttt{ or } y$
`GXor`	$\rightarrow$	$o \Leftarrow x \texttt{ xor } y$	`GNand`	$\rightarrow$	$o \Leftarrow x \texttt{ nand } y$
`GNor`	$\rightarrow$	$o \Leftarrow x \texttt{ nor } y$	`GXnor`	$\rightarrow$	$o \Leftarrow x \texttt{ xnor } y$
`GEq`	$\rightarrow$	$o \Leftarrow x = y$	`GNe`	$\rightarrow$	$o \Leftarrow x/=y$
`GGt`	$\rightarrow$	$o \Leftarrow x > y$	`GLt`	$\rightarrow$	$o \Leftarrow x < y$
`GGe`	$\rightarrow$	$o \Leftarrow x >= y$	`GLe`	$\rightarrow$	$o \Leftarrow x <= y$
`GIdx i i`	$\rightarrow$	$o \Leftarrow x(i)$	`GIdx h l`	$\rightarrow$	$o \Leftarrow x(h \texttt{ downto } i)$
`GMux`	$\rightarrow$	$o \Leftarrow x \texttt{ when } s \texttt{ else } y$			
`GReg`	$\rightarrow$	$o \Leftarrow v \texttt{ when rising_edge}(c) \texttt{ else } o$			
`GReg`	$\rightarrow$	$ev \Leftarrow v \texttt{ when } ena \texttt{ else } o$ $o \Leftarrow ev \texttt{ when rising_edge}(c) \texttt{ else } o$			
`GReg`	$\rightarrow$	$rv \Leftarrow IV \texttt{ when } rst \texttt{ else } v$ $o \Leftarrow rv \texttt{ when rising_edge}(c) \texttt{ else } o$			
Instantiations					
`GEnt i e`	$\rightarrow$	`i : entity e generic map (...) port map (...)`			

4.2 Synthesizable Software Components

In addition to hardware circuits, RST is also capable of producing software programs for MicroBlaze processors via the Low-Level Virtual Machine Intermediate Representation (LLVM-IR). Like the creation of hardware circuits, the creation of software programs is accomplished in several steps. First, an LLVM-IR data structure is created to represent the state of any facet being lowering into software. Second, an LLVM-IR function is created for each facet that implements the behavior described by the facet's behavioral specification. Finally, a top-level `main` function is created to serve as the software programs main entry point.

As with hardware, facets that are to be synthesized into software programs must meet a set of constraints. First, any declarations inside of the facet must be of type `word(N)`, `float(32)`, or `float(64)`. These Rosetta types represent the integer values, single precision floating point values, and double precision floating point values supported by the microprocessor. Additionally, behavioral terms in the facet must match a set of supported terms, as described below. The supported behavioral terms are designed to closely match the semantics of the microprocessor instructions so that the resulting implementation is as efficient as possible. Finally, the facet must have no parameters and no structurally instantiated facets. The requirement that software synthesizable facets have no parameters is due to the fact that software programs do not directly support I/O. Instead, software performs I/O using microprocessor instructions. Support for the notion of instructions that perform I/O is provided in the form of the uninterpreted Rosetta functions `mmioread`, `mmiowrite`, `fslread`, and `fslwrite`. These functions are described in more detail below.

The lack of support for structurally instantiated facets when synthesizing software programs is due to the fact that none of the examples that were constructed to evaluate the Rosetta synthesis and transformation tools required structural instantiation. It is believed that software support could be added for structural instantiation with a moderate amount of effort.

Facet State Representation

The first step in the synthesis of a software programs from Rosetta specifications is to form an LLVM-IR data structure to represent the state needed by any facet that will be part of the software program. This is accomplished by examining all of the declarations in each facet. Table 4.13 shows the supported Rosetta declarations types along with the LLVM-IR representation of those types.

The first five entries in this table list the types supported by RST that are directly supported by the MicroBlaze microprocessor. These types represent 8-bit, 16-bit, and 32-bit integer as well as 32-bit and 64-bit floating point values. Since these types are supported directly by the microprocessor being targeted, the use of these types will result in efficient implementation at the instruction level.

Table 4.13: Facet State

Rosetta Type	LLVM-IR Type
`word(8)`	`i8`
`word(16)`	`i16`
`word(32)`	`i32`
`float(32)`	`float`
`float(64)`	`double`
`word(N)`	`iN`

The final supported type represents arbitrary width integers. This type is converted to an LLVM-IR type of the same bit-width, which is directly supported by LLVM-IR but not directly supported by the MicroBlaze microprocessor. When compiling LLVM-IR into MicroBlaze assembly, LLVM will convert the type into an appropriate instruction level representation. Currently, this only works reliably in the LLVM MicroBlaze backend for bit-widths up to 64-bits. More information on the LLVM MicroBlaze backend is given in section B.

Behavioral Terms as Software

After the LLVM-IR state representation has been formed for a facet, a LLVM-IR function is created that implements the behavior of the facet. Because RST relies on the LLVM-IR to represent software semantics, the process of lowering Rosetta behavioral specifications into software is much simpler than the process required to create hardware circuits.

Table 4.14: LLVM-IR Representation

Constant Values

Expression		LLVM-IR	Expression		LLVM-IR
12	→	i32 12	b"001"	→	i3 1
x"FF"	→	i8 255	o"70"	→	i32 56
12.0	→	float 12.0			

Integer Unary Expressions

Expression		LLVM-IR
not x	→	%o = xor i32 %x, 0xFFFFFFFF

Integer Binary Expressions

Expression		LLVM-IR	Expression		LLVM-IR
$x+y$	→	%o=add i32 %x, %y	$x-y$	→	%o=sub i32 %x, %y
$x*y$	→	%o=mul i32 %x, %y	x/y	→	%o=udiv i32 %x, %y
x and y	→	%o=and i32 %x, %y	x or y	→	%o=or i32 %x, %y
x xor y	→	%o=xor i32 %x, %y	x nand y	→	%o=nand i32 %x, %y
x nor y	→	%o=nor i32 %x, %y	x xnor y	→	%o=xnor i32 %x, %y
$x\&y$	→	%xs=shl i64 %x, 32	x sub [...]	→	%m=and i32 ...
		%o=or i64 %xs, %y			%o=shr i32 ...

Integer Comparison

Expression		LLVM-IR	Expression		LLVM-IR
$x==y$	→	%o=icmp eq i32 %x, %y	$x/=y$	→	%o=icmp ne i32 %x, %y
$x<y$	→	%o=icmp ugt i32 %x, %y	$x<y$	→	%o=icmp ult i32 %x, %y
$x>=y$	→	%o=icmp uge i32 %x, %y	$x<=y$	→	%o=icmp ule i32 %x, %y

Floating-Point Binary Expressions

Expression		LLVM-IR	Expression		LLVM-IR
$x+y$	→	%o=fadd float %x, %y	$x-y$	→	%o=fsub float %x, %y
$x*y$	→	%o=fmul float %x, %y	x/y	→	%o=fdiv float %x, %y

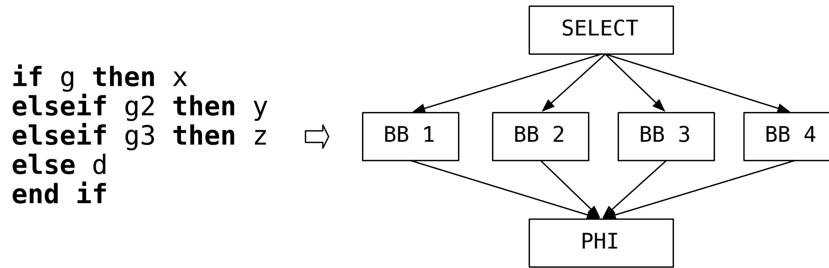
Floating-Point Comparison

Expression		LLVM-IR	Expression		LLVM-IR
$x==y$	→	%o=fcmp ueq float %x, %y	$x/=y$	→	%o=fcmp une float %x, %y
$x<y$	→	%o=fcmp ugt float %x, %y	$x<y$	→	%o=fcmp ult float %x, %y
$x>=y$	→	%o=fcmp uge float %x, %y	$x<=y$	→	%o=fcmp ule float %x, %y

I/O Access

Expression		LLVM-IR
mmioread(x)	→	%o=load i32 %x, %y
fslread(x)	→	%o=llvm.mblaze.fsl.nget(%x)

Figure 4.6: If-Expression LLVM-IR



Much like the case for hardware synthesis, behavioral terms that are to be synthesized to software programs must be I/O terms or terms of the form "`lhs = rhs;`". I/O terms are either "`mmiowrite(addr, val)`" or "`fslwrite(port, val)`". For the `mmiowrite` case, a memory write is generated to the address `addr` with the value `val` being written into this memory address. In the `fslwrite` case, an FSL `put` instruction is generated that puts the value `val` into the FIFO transmission queue of FSL port `port`. For other synthesizable terms, `lhs` must be a declaration and `rhs` must be an expression that can be synthesized into software. Table 4.14 lists the expressions that are supported by software synthesis along with the LLVM-IR representations of those expressions.

RST first creates an LLVM-IR implementation of the `rhs` expression. This produces an LLVM-IR value that is the result of the `rhs` expressions. It then creates and LLVM-IR `store` instruction to store the value into the facet's state. Access to the facet's state is done by having the state as a parameter to the facet's implementing function.

RST produces a facet's implementation in a top-down manner, meaning that the order of terms in a facet's body are preserved in the facet's LLVM-IR implementation. This breaks from the strict semantics of the Rosetta language but provides an efficient mechanism for describing the sequentiality provided by microprocessors. It is believed that the semantic difference between the software implementation and the Rosetta specification can be eliminated by specifying software facets in a special Rosetta domain that has the same semantics as the software implementation.

Not shown in table 4.14 but supported by RST is support for Rosetta if-expressions. Synthesizable if-expressions must be composed only of guards and values that are also synthesizable. If this is the case then RST will generate an LLVM-IR diamond control flow pattern as shown in figure 4.6. In the figure the four basic blocks represent the instruction sequences needed to implement the value expressions `x`, `y`, `z`, and `d`. The guard expressions `g1`, `g2`, and `g3` have their instructions placed in the `SELECT` block. This block ends with an LLVM-IR `select` instruction that branches to the correct basic block based on the guard conditions. At the end of each basic block an LLVM-IR `br` instruction is used to unconditionally branch to the `PHI` block. This block uses an LLVM-IR `phi` instruction to select the correct final result.

Software synthesis support for the Rosetta `sub` operation allows selection of sub-sequences from Rosetta bit-vectors. This operation is lowered into a sequence of LLVM-IR `and` and `shr` instructions that select the correct bits from the integer value that represents the Rosetta bit-vector.

Comparison of terms supported by hardware synthesis, shown in table 4.7, against the terms supported by software synthesis shows that software synthesis supports additional constructs in the form of floating point support, though, the software synthesis support lacks clock construct support and RAM support offered by hardware synthesis. Clock support is omitted purposefully because software programs have no need for clocks. The lack of RAM support is because software RAMs were not needed in the course of evaluating this work. Adding support for RAMs during software synthesis would not be difficult.

FSL I/O Support

RST provides support for Fast Simplex Link (FSL) communication to facilitate communication between MicroBlaze microprocessors and hardware circuits. This support is provided in the form of three uninterpreted Rosetta functions: `fslread`, `fslwrite`, and `fslcanread`. When used in a Rosetta facet that is synthesized to software, these three functions become MicroBlaze FSL `nget`, `put`, and `tnget` instructions.

MicroBlaze microprocessors support up to 16 bi-directional FSL communication channels that are implemented using 32 uni-directional FSLs. Each link can have an optional FIFO buffer with a depth of up to 8192 elements and each element can be up to 32-bits in width. Each FSL implements a point-to-point communication primitive and provides high-speed, low-latency, arbitration free data transmission.

FSL components can be used in a Rosetta specification to provide communication between software programs running on different MicroBlaze microprocessors, to provide communication between a MicroBlaze microprocessor and a hardware circuit, or to buffer communication between hardware circuits.[2] RST exposes FSLs as the only HW/SW co-design capable communication mechanism, although it is believed that other communication mechanisms could be easily added.

The three exposed FSL functions provide a mechanism to allow software to read data from an FSL port, to write data into an FSL port, or to query if data is available on an FSL port. The semantics of these operation match the semantics of the non-blocking MicroBlaze FSL read instructions and the blocking MicroBlaze FSL write instruction. The `fslread` function returns valid results only if `fslcanread` results in a true value. The FSL `fslwrite` function waits until there is space in the FSL port before writing data into the port.

Note that the three FSL functions are side-effecting functions but the Rosetta language does not provide good support for defining sequential computation. As described earlier, RST works around this problem using a top-down sequencing semantics outside of the Rosetta language.

Support for synchronizing when writing to an FSL port has not been implemented because it was not necessary for any of the example designs constructed to evaluate this work. Support for this functionality, in the form of the `fslcanwrite` function, would not be difficult to implement.

[2] This is typically done when the hardware circuits involved use different clock domains.

Conversion into an MicroBlaze Instruction Stream

Once the complete LLVM-IR representation has been formed for all facet's being implemented on the MicroBlaze microprocessor, RST invokes the LLVM toolset to convert the LLVM-IR representation into an instruction stream for the MicroBlaze processor. The result is a binary software program that can be loaded into memory and executed by a MicroBlaze processor.

Deficiencies in Software Synthesis

Currently software synthesis does not support implementation of recursive functions, implementation of recursive data types, or any form of dynamic resource allocation. These deficiencies limit RST's ability to produce large software programs because these constructs are highly important in the construction of such programs. Support for these constructs was not provided because the rapid prototyping and hardware/software co-design aspects of this work can be evaluated without them. Adding support for these features is possible and would make the tool much more useful but would require much more engineering effort.

4.3 Synthesis Example

Listings 4.5 and 4.6 are examples of Rosetta specifications that are synthesizable to hardware and software respectively. Listing 4.5 gives an example of a pseudo-random number generator (PRNG) implemented using three linear-feedback shift registers (LFSRs). In this specification, the PRNG facet is controlled via a clock signal, a reset signal, and an enable signal. Each clock cycle, when enable is high and reset is low, the PRNG facet produces a new random value as its output.

The specification contains both behavioral and structural design elements along with a total of 12 state elements, three for PRNG and three for each LFSR. The specification of the PRNG facet uses structural instantiation to provide three LFSR components. The output of the first LFSR is examined and used to select between the outputs of the other two LFSRs.

The LFSRs themselves, are specified using a behavioral model. In this design, the shift register `sr` is updated every clock cycle that enable is high and reset is low to shift the current value

Listing 4.5: Rosetta Hardware Synthesis Example

```
1  facet prng(clk,rst,ena::input bit;o::output word(8))::state_based is
2      l0, l1, l2 :: word(9);
3  begin
4      o' = if l0(0) then (l1 sub [7,..0]) else (l2 sub [7,..0]) end if;
5      lfsr0: lfsr(o"265", clk, rst, ena, l0);
6      lfsr1: lfsr(o"654", clk, rst, ena and l0(0), l1);
7      lfsr2: lfsr(o"271", clk, rst, ena and (not l0(0)), l2);
8  end facet;
9
10 facet lfsr(IV::design word(9); clk,rst,ena::input bit;
11             o::output word(9) ) :: state_based is
12     sr::word(9); sh::word(8); fb::bit;
13 begin
14     o = sr; sh = sr sub [7,..0]; fb = (sr(4) xnor sr(0));
15     sr' = if clk'event and clk=1 then
16             if rst=1 then IV
17             elseif ena=1 then sh & fb
18             else sr end if
19           else sr end if;
20 end facet;
```

left by one-bit with the least significant bit being replaced by a linear function of the other bits in the register.

First Phase of Hardware Synthesis

The results of the first phase of hardware synthesis are shown in tables 4.15 and 4.16. Table 4.15 shows the RTL intermediate form that is produced from the specification of the LFSR facet. Comparing the RTL intermediate form to the original specification starting on line 10 of listing 4.5, we see that the design parameter becomes a generic node and that all of the input and output parameters become input nodes and output nodes respectively.

The three terms that appear on line 14 of the specification result in three multiplexer nodes that have no selectors present. RST produces these nodes as a way to alias another node. These nodes will be eliminated when the explicit graph representation is formed. The RHS

Table 4.15: **LFSR RTL Intermediate Form**

ID		RTL Node	ID		RTL Node
0	→	RTLG BitVector "IV"	1	→	RTLI "clk"
2	→	RTLI "rst"	3	→	RTLI "ena"
4	→	RTLO "o" 5	5	→	RTLE (Select [] 15)
6	→	RTLE (Select [] 8)	7	→	RTLE (Select [] 11)
8	→	RTLE (Split 7 0 5)	9	→	RTLE (Split 4 4 5)
10	→	RTLE (Split 0 0 5)	11	→	RTLE (Binary Xnor 9 10)
12	→	RTLE (Const 1)	13	→	RTLE (Const 1)
14	→	RTLE (Binary Concat [6,7])	15	→	RTLE (Select [Rs 1] [16] 5)
16	→	RTLE (Select [Eq 2 12,Eq 3 13] [0,14] 5)			

expressions in these three terms become the value used by output node 4, the bit-vector split operation of node 8, and the bit-vector split nodes 9 and 10 along with the xnor node 11.

The last term, starting on line 15, results in a couple of constant value nodes along with two multiplexer nodes and a concatenation node. The first multiplexer is used to implement the outer if expressions and produces a rising edge clock selector. The second multiplexer is used to implement the inner if expression and contains one equality selector for the reset case and another equality selector for the enable case.

Table 4.16 shows the RTL intermediate form that is produced from the specification of the PRNG facet. The parameters and behavioral terms of the PRNG facet are implemented very similarly to the LFSR facet. In this case, the representation of the structural instantiations is of more interest.

The first structural instantiation, `lfsr0`, is implemented by creating the entity node 16, and the entity output node 17. The entity node contains the names of all design parameters, input parameters, and output parameters that are needed by the instantiated facet. Additionally, for generic parameters and input parameters, the entity contains a list of node references that provide the values for those ports. In this case, node 11 provides the design parameter and nodes 0, 1, and 2 provide the input values. The name of the entity's only output is contained in the entity declaration but its value is represented by node 17. This nodes can be used to reference the correct output port of the instantiated entity, as seen by node 17's use in node 4.

Table 4.16: PRNG RTL Intermediate Form

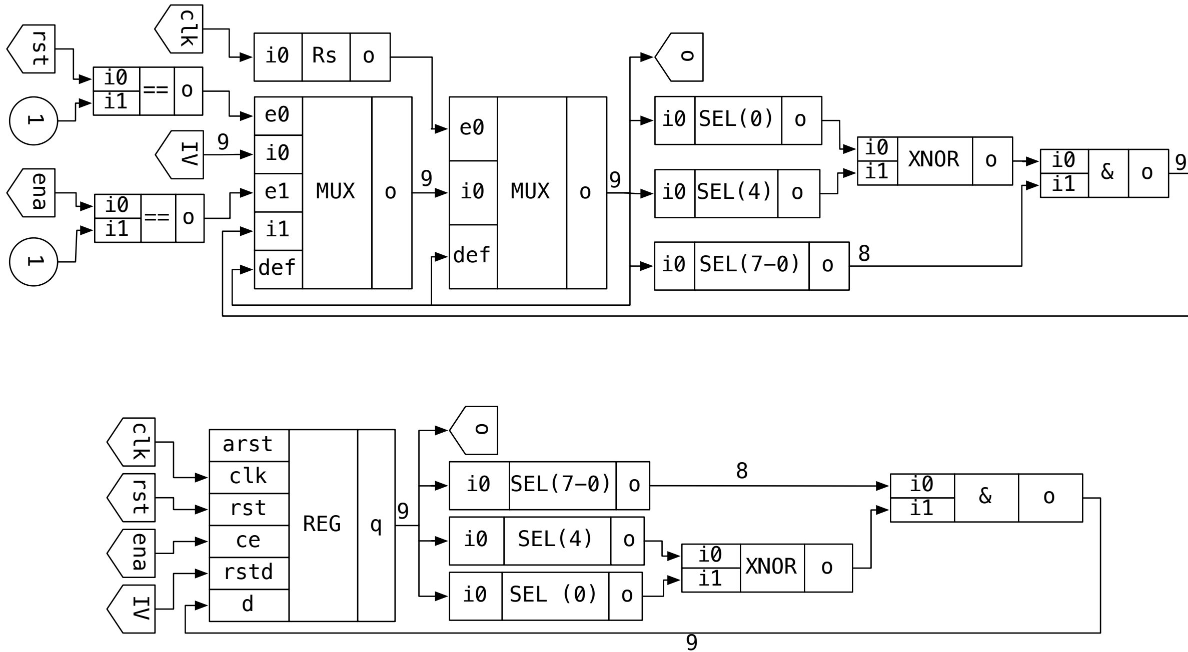
ID		RTL Node	ID		RTL Node
0	→	RTLI ”clk”	1	→	RTLI ”rst”
2	→	RTLI ”ena”	3	→	RTLO ”o” 3
4	→	RTLE (Select [] 17)	5	→	RTLE (Select [] 26)
6	→	RTLE (Select [] 36)	7	→	RTLE (Split 0 0 4)
8	→	RTLE (Split 7 0 5)	9	→	RTLE (Split 7 0 6)
10	→	RTLE (Select [TId 7 8] 9)	11	→	RTLE (Const 181)
12	→	RTLE (Const 0)	13	→	RTLE (Const 0)
14	→	RTLE (Const 0)	15	→	RTLE (Const 0)
17	→	RTLD ”o” 16	18	→	RTLE (Const 428)
19	→	RTLE (Const 0)	20	→	RTLE (Const 0)
21	→	RTLE (Const 0)	22	→	RTLE (Split 0 0 4)
23	→	RTLE (Binary And [2,22])	24	→	RTLE (Const 0)
26	→	RTLD ”o” 25	27	→	RTLE (Const 185)
28	→	RTLE (Const 0)	29	→	RTLE (Const 0)
30	→	RTLE (Const 0)	31	→	RTLE (Split 0 0 4)
32	→	RTLE (Unary Not 31)	33	→	RTLE (Binary And [2,32])
34	→	RTLE (Const 0)	36	→	RTLD ”o” 35
16	→	RTLE (Entity ”lfsr” [”IV”] [11] [”clk”,”rst”,”ena”] [0,1,2] [”o”])			
25	→	RTLE (Entity ”lfsr” [”IV”] [18] [”clk”,”rst”,”ena”] [0,1,23] [”o”])			
35	→	RTLE (Entity ”lfsr” [”IV”] [27] [”clk”,”rst”,”ena”] [0,1,33] [”o”])			

The other two structural instantiations have implementations that are very similar to the first node. The results are two additional entity nodes, nodes 25 and 35, along with two entity output nodes, nodes 26 and 36. The remaining nodes in the RTL intermediate form output are used to form the implementations of the PRNG facet's outputs and to implement the arguments used by the LFSR instantiations.

Second and Third Phase of Hardware Synthesis

The results of the second and thirds phases of hardware synthesis are shown in figures 4.7 and 4.8. Figure 4.7 shows the results of the second phase of hardware synthesis for the LFSR facet on the top of the figure and the third phase on the bottom of the figure. In the top figure we can see that the explicit graph representation has mostly the same content as the RTL intermediate form from which it is generated. All of the parameters show up in the explicit graph representation, as well as the two multiplexers, equality selectors, rising edge clock selector, bit-vector slices, xnor, and concatenation.

Figure 4.7: LFSR Graph Representation

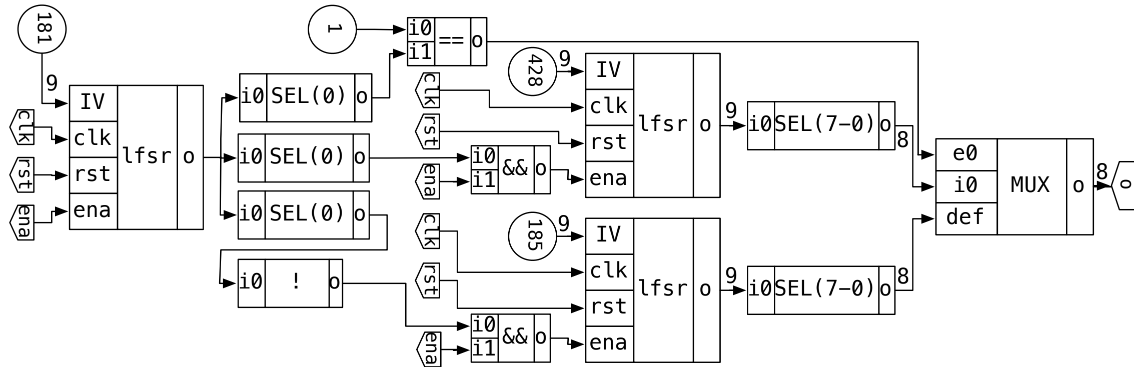


Obviously, the most major difference between the RTL intermediate form representation and the explicit graph representation is that node references in the RTL intermediate form have been resolved into edges with known width. Additionally, node aliases, nodes 5, 6, and 7, in the RTL intermediate form have been eliminated and selectors have been split out into their own vertices in the graph instead of being contained inside of the multiplexer node.

The lower diagram in figure 4.7 shows the results of performing canonicalization and resource discovery. In this case, both multiplexers and all of the selectors have been eliminated from the graph and a register with a clock signal, reset signal, and enable signal has been created in their place. The rest of the graph remains the same.

Figure 4.8 shows the second and third phases of hardware synthesis for the PRNG facet. Like the explicit graph representation for the LFSR facet, the explicit graph representation of the PRNG facet contains mostly the same contents as its corresponding RTL intermediate form, with alias nodes and selector vertices being the main difference. In this graph however, the entity output nodes have also been eliminated during the conversion from RTL intermediate

Figure 4.8: PRNG Graph Representation



form to explicit graph representation. The entity nodes from the RTL intermediate form have become vertices in the graph with multiple output ports and the nodes from the RTL intermediate form that represented these output ports have been used to create the proper edges in the graph.

Unlike the LFSR facet, the explicit graph canonicalization and discovery phase does not change the graph representation of the facet. Thus, figure 4.8 represents the results of both the second phase and the third phase of hardware synthesis.

Fourth Phase of Hardware Synthesis

The last phase in hardware synthesis is the production of VHDL for the hardware circuits. For this example, two VHDL files are created, one for the LFSR hardware circuit and another for the PRNG hardware circuit. The content of these two VHDL files can be seen in appendix C. The LFSR VHDL file contains a simple VHDL entity declaration with one generic port, three input ports, and a single output port. The architecture definition for the LFSR VHDL entity contains the behavioral implementation of the facet. The PRNG VHDL file contains a VHDL entity declaration with three input parameters and a single output parameter. The architecture definition for the entity is composed mainly of the three LFSR instantiations.

Listing 4.6: Rosetta Software Synthesis Example

```
1  facet macc :: state_based is
2      xr,yr::bit; x,y,acc::word(32);
3  begin
4      xr' = fslcanread(0);
5      yr' = fslcanread(1);
6
7      x' = if (xr == 1) and (yr == 1) then fslread(0) else x end if;
8      y' = if (xr == 1) and (yr == 1) then fslread(1) else y end if;
9
10     acc' = if (xr == 1) and (yr == 1) then
11                if (x == x"00000000") and (y == x"00000000")
12                then x"00000000"
13                else acc + x*y
14                end if
15            else acc end if;
16
17     if (xr == 1) and (yr == 1) then
18         fslwrite(2, acc)
19     else true end if;
20 end facet;
```

Software Synthesis

Listing 4.6 gives the specification of a simple, software-based multiply-accumulate component. Since this components is designed for synthesis into a software program, the facet has no I/O parameters. Instead, all communication between the component and other components that may exist in a larger system is done using FSL communication primitives.

The specification of this facet relies on the informal top-down synthesis semantics of RST. Thus, the first two steps done by the software program are to determine if there is any data available on FSL ports `0` and `1` of the MicroBlaze processor. The program then updates the internal state variables `x` and `y` to read the new values from the FSL ports if they are available. If data is not available on the FSL ports then the state is not changed.

Likewise, the accumulator state is updated only if data is available on the FSL ports. If data is available then the accumulator is either reset to `0` or is updated by adding the multiplication

Listing 4.7: Rosetta MACC LLVM Intermediate Representation

```
1  define void @main() noreturn {
2  _L0:   br label %_L1.o
3  _L1.o: %macc_state.4.0.ph = phi i32 [ 0, %_L0 ], [ %11, %_L9.i ]
4         br label %_L1
5  _L1:   %0 = tail call i32 @llvm.mblaze.fsl.tnget(i32 0)
6         %1 = tail call i1 asm sideeffect "addc $0, r0, r0\0A", "=r"()
7         %2 = tail call i32 @llvm.mblaze.fsl.tnget(i32 1)
8         %3 = tail call i1 asm sideeffect "addc $0, r0, r0\0A", "=r"()
9         %4 = and i1 %1, %3
10        br i1 %4, label %_L1, label %_L4.i
11 _L4.i: %5 = tail call i32 @llvm.mblaze.fsl.nget(i32 0)
12        %6 = tail call i32 @llvm.mblaze.fsl.nget(i32 1)
13        %7 = or i32 %6, %5
14        %8 = icmp eq i32 %7, 0
15        %9 = mul i32 %6, %5
16        %10 = add i32 %9, %macc_state.4.0.ph
17        %11 = select i1 %8, i32 0, i32 %10
18        br label %_L9.i
19 _L9.i: tail call void @llvm.mblaze.fsl.put(i32 %11, i32 2)
20        %12 = tail call i1 asm sideeffect "addc $0, r0, r0\0A", "=r"()
21        br i1 %12, label %_L9.i, label %_L1.o
22 }
```

of x and y to the accumulated value. Finally, if data is available on the FSL ports then the new value of the accumulator is written into FSL port 2.

LLVM Intermediate Representation

The LLVM-IR that is created by RST is shown in listing 4.7. For simplicity, the LLVM-IR shown has been highly optimized by LLVM's optimization framework. The raw LLVM-IR produced by RST can be seen in appendix D. The state representation of the facet, that is clearer in the raw LLVM-IR, is two boolean values and three 32-bit integer values and correspond directly to the declarations on line 2 of the specification. In the LLVM-IR, the state for the acc declaration is maintained on line 3 of result. Lines 6, 8, 11, and 12 correspond to the other four declarations.

The two `fslcanread` functions are implemented as lines 5-6 and 7-8. The values resulting from these two instruction sequences are booleans that indicate whether there is data available on FSL ports `0` and `1`.

The conditional branch on line 10 is used to implement the guard statements that wrap all of the other terms in the `macc` facet's specification. In this case, if there is nothing available on the FSL ports then the branch goes back to label `_L1`, causing the program to continue to wait on data from the FSL ports. If data is available then the branch goes to label `_L4.i`.

Lines 11-12 in the LLVM-IR implement the `fslread` functions from the facet's specification and correspond to the expressions inside of the then clause on lines 7 and 8 of the specification. Lines 13-17 are the implementation of the accumulator from the then clause starting on line 11. Finally, lines 19-21 in the LLVM-IR provide the implementation of the `fslwrite` function. Once the FSL put instruction has finished successfully, the entire process happens again so that the next input is processed.

MicroBlaze Instruction Stream

After the LLVM-IR has been formed, a MicroBlaze instruction stream is created using the LLVM framework. The resulting MicroBlaze instruction stream can be seen in appendix D. The instruction stream follows the LLVM-IR closely with most operation having directly corresponding instructions on the MicroBlaze. The most major difference is that the 32-bit multiply on line 15 of the LLVM-IR has been turned into a function call in the instruction stream. This is because the multiply instruction on the MicroBlaze microprocessor is optional and the LLVM framework was told not to make use of the instruction. The resulting instruction stream has lower performance but the MicroBlaze takes up less area in this configuration.

5 HW/SW Co-Design of Top-Level Designs

The primary goal of the Rosetta Refinement Tool (RRT) is to provide a rapid prototyping environment for system-level, hardware/software co-design using the Rosetta language. As such, RRT is designed so that implementation decisions can be delayed as long as possible during the specification, design, and implementation process. When designing systems using RRT, the goal is to specify all facets in the most implementation agnostic manner while still maintaining an ability to refine that specification into an implementation. These facets can then be instantiated and refined using declarative refinements that are exposed in the Rosetta language.

Ideally, all refinements would be located in the top-level design, although this is not a requirement enforced by RRT. If this is the case then we have essentially built an implementation agnostic specification all the way up to the top-level. The design is then transformed, from the top-level down, into an implementation. If done correctly, the refinements applied at the top-level can be changed to generate different kinds of implementations. This provides the basis for rapid prototyping.

The final output from RRT is a hardware/software co-designed implementation. Currently, RRT supports refinement into Xilinx FPGA based hardware/software co-design implementations, but the refinement mechanisms used by RRT should extend easily to other platforms. RRT takes implementation agnostics specifications written in Rosetta and augments them with implementation details in the form of declarative refinements. The current set of refinements provide the implementation details required for hardware/software co-design implementation on FPGA platforms but other platforms could be supported by providing additional refinements to support any additional implementation details required by that platform.

To produce a system implementation, RRT requires that three pieces of information be described by a system engineer in the form of a top-level design. First, a logical system is created to describe the required behavior of the system. This description is created by structurally instantiating Rosetta facets inside of the top-level design and connecting these facets via their

I/O parameters. Behavioral terms are not allowed as part of the logical system description because it is difficult to describe the mapping from a behavioral term to a physical resource[1]. This constraint is not viewed as a limitation because the same effect can be achieve by placing the behavioral term into a Rosetta facet that can then be instantiated inside of the top-level design.

Second, a physical system is created to describe the physical layout of the implementation architecture. This description is created just like the description of the logical system, by structurally instantiating Rosetta facets and connecting them together. The only difference is that facets that make up the physical system are defined in one of the three domains RRT defines for describing physical resources. These physical resource domains are described in section 5.2.

Third, a system partitioning description is created by mapping the components and communications in the logical system to the components and communications in the physical system. This mapping must define a physical component for every logical component but does not necessarily need to define a logical component for every physical component.

The top-level design produced by the system engineer must meet several requirements. These requirements are enforced by RRT with an error being generated when the requirements are not met. First, the top-level design must not have any parameters. The implementations currently produced by RRT target Xilinx FPGAs. As such, the implementations produced are system-on-chip designs with the chip boundary containing the entire implementation. Off-chip communication in these systems is achieved by routing signals from input pins or to output pins. It was decided that an explicit representation of this in the Rosetta language would be better than the implicit representation using Rosetta parameters. Thus, off-chip communication is achieved by creating a declaration in the top-level design and then routing that declaration to an implementation-level I/O pin using one or more of the synthesis attributes described in appendix A.

[1]This is difficult because structural terms in Rosetta are given names but behavioral terms are not.

Second, the top-level design must be composed only of resource instantiations, component instantiations, declarations, declarative refinements, component mappings, and synthesis attributes.[2] This requirement ensures that the top-level design contains only the description of the logical system, the description of the physical system, and the description of the system partitioning. The component instantiations and some of the declarations are part of the logical system description while the resource instantiations, synthesis attributes, and other declarations are part of the physical system description. The component mappings describe the system partitioning. Finally, the declarative refinements are eliminated by RRT during the implementation of the system and are used to refine high-level components into synthesizable components. The declarative refinements provide the rapid-prototyping capability of RRT.

Third, every component in the top-level design must be synthesizable to its target physical resource, as described by the system partitioning. While refining the top-level design into its implementation, RRT processes each component in the logical system description using either RST's hardware generation capabilities or its software generation capabilities so each component must meet RST's synthesis requirements. This constraint is not enforced by RRT but an error will be generated either by RST or by the Xilinx synthesis tool during the synthesis process if this is not followed.

If the top-level design contains a valid logical system description, a valid physical system description, and a valid system partitioning, then RRT is capable of refining the Rosetta specification defined by the top-level design into a hardware/software co-designed implementation in the form of a Xilinx Platform Studio project. This Xilinx project can then be synthesized, using the Xilinx synthesis tools, into a bit-stream that can be loaded onto a Xilinx FPGA for execution.

The resulting system is capable of refining high-level Rosetta specifications all the way down to executable bitstreams for Xilinx FPGAs. This system supports rapid-prototyping in the form of easy to modify and mechanically performed declarative refinements and can produce hard-

[2]The synthesis attributes are used to provide additional details to the Xilinx tools that can be used to guide the final synthesis process. These attributes are described in appendix A.

ware/software co-designs in the form of system-on-chip implementations containing both VHDL circuits and MicroBlaze software programs.

The complete flow of RRT, after validating that the top-level design meets all of the requirements, is as follows:

1. Perform all refinements that exist in the design in the order that they appear in the design. The transformations in each facet are processed separately and the order that facets are processed is undefined.
2. Determine all of the physical resources that have been defined and classify them as targets, bridges, or memory resources.
3. Determine the system partitioning by examining the mapping of logical components to physical components and ensure that every logical component has a defined mapping to a physical resource.
4. For every physical target, implement the logical components that are partitioned onto that target using RST.
5. Generate a Xilinx project that includes all of the defined resources and all of the implemented components.

5.1 Processing Refinements

The first step in the process of refining a top-level design into an implementation is to discover and mechanically apply all of the refinements declared in the specification. At a basic level, RRT visits each of the facets in the specification one at a time in no defined order. For each facet, RRT locates the next declarative refinement and then performs the required transformations. These transformations may change one or more facets in the specification. This process repeats until there are no more refinements left in the specification. Full details on how refinements are processed are given in section 6. This section give mores explicit details about the mechanisms used to discover and perform refinements and gives detailed descriptions of all of the refinements supported by RRT.

Listing 5.1: Shared Facet Definitions

```
1  facet ex1(a :: input bit; b :: output bit) :: state_based is
2  begin
3      b' = a;
4  end facet;
5
6  facet tld :: system is
7      a1, b1, a2, b3 :: bit;
8  begin
9      i1: ex1(a1, b1); i2: ex2(a2, b2);
10 end facet;
```

Listing 5.2: Specialized Facet Definitions

```
1  facet ex1(a :: input bit; b :: output bit) :: state_based is
2  begin
3      b' = a;
4  end facet;
5
6  facet i1_ex1(a :: input bit; b :: output bit) :: state_based is
7  begin
8      b' = a;
9  end facet;
10
11 facet i2_ex1(a :: input bit; b :: output bit) :: state_based is
12 begin
13     b' = a;
14 end facet;
15
16 facet tld :: system is
17     a1, b1, a2, b3 :: bit;
18 begin
19     i1: i1_ex1(a1, b1); i2: i2_ex2(a2, b2);
20 end facet;
```

Before discovery and application of refinements begins, specialization is done to prepare the specification for refinement. During this processes every facet in the specification is visited one at a time and every structural instantiation in that facet is given a unique facet definition, generating a unique facet definition for every structural instantiation in the specification.

For example, listing 5.1 shows a system where two instantiations, `i1` and `i2`, share a single facet definition, `ex1`, for their behavioral description. The specialization process produces the

system shown in 5.2. The `ex1` facet has been duplicated twice, once for each instance, and the structural instances in `tld` have been updated to use the new facet definitions. Duplicating facet definitions like this for each structural instance allows the facet definition for each instance to be refined separately.

Specialization is completed by applying a partial evaluator to the specification as an initial simplification step. High-level specifications often define facets with many design and I/O parameters for flexibility and many times these design and I/O parameters are instantiated with constant values. After this step has been completed, every structural instantiation in the specification has been given a unique facet definition and this definition has been specialized to that instantiation's applied arguments.

The partial evaluator used by RRT is currently capable of:

1. Removing constant value arguments from an instantiation and replacing that argument in the facet definition with its constant value.
2. Replacing named constants in a facet definition with their defined values, including locals constants, package defined constants, and globally defined constants.
3. Replacing all function applications with the given definition of the function, ensuring that parameters are replaced by their arguments.[3]
4. Replacing case expressions with equivalent if expressions as case expressions are not synthesizable but if expressions can be synthesized.[4]
5. Simplifying expressions involving constant values, including all arithmetic expressions, all boolean expressions, if expressions, and concatenation.

After the specification has been specialized, all declarative refinements are mechanically applied to the specification as described in 6. After this process completes, RRT performs partial evaluation a second time to cleanup any constructs generated during the mechanical transformation of the specification. Additionally, a set of default data type refinements are applied to ensure that all data types are removed from the specification. The default refinements are

[3]Recursive functions are not processed by the partial evaluator to avoid infinite recursion.
[4]Support for pattern matching is limited to un-nested data constructors and variable bindings.

specified by the system engineer, using the `encoding` and `alignment` attributes, when defining a data type. These data type refinements are performed by the data type refinement logic described in section 6.3.

5.2 Implementation Resources

The second step performed by RRT is to identify all physical resources defined in the top-level design. In the top-level design, both the logical system being implemented and the physical system that is being used for the implementation are defined by structurally instantiating and connecting Rosetta facets defined outside of the top-level design. In effect, the second step is to determine what facet instantiations are part of the logical system and what facet instantiations are part of the physical system.

To simplify this step, RRT defines three Rosetta domains that must be used to describe facets that describe physical resources:

target: Physical resources in the `target` domain are used to define target fabrics that can be used to implement the behavior of the logical components that are partitioned onto them. The `target` domain is described fully in section 5.2.1.

bridge: Physical resources in the `bridge` domain are used to define communication pathways between two or more physical resources. The `bridge` domain is described fully in section 5.2.2.

memory: Physical resources in the `memory` domain are used to defined storage that can be connected to one or more physical resources. The `memory` domain is described fully in section 5.2.3.

Facets defined in one of these three domains can be structurally instantiated in the top-level design to define the physical resources available in that design. Listings 5.4, 5.5, and 5.6 give examples of the three kinds of physical resources. Listing 5.3 shows how these facets can be instantiated inside of a top-level design to define the physical resources available in the design. In this case, the top-level design is instantiating two target fabrics, `fpga` and `mbl1`, along

Listing 5.3: Top-Level Design Resource Instantiation

```
1  facet tld :: system is
2  begin
3      fpga: genesys(sys_clk,sys_rst,rx,tx);
4      mbl1: mblaze(clk0,rst);
5      mbl1_bram: bram_memory(8192);
6      mbl1_imem: lmb_v10(clk0,rst);
7      mbl1_dmem: lmb_v10(clk0,rst);
8      fsl0: fsl_v20(1,0,0,0,32,1,0,clk0,rst,_,clk0,fsl0_m_data,
9                    fsl0_m_ctl,fsl0_m_wr,fsl0_m_full,clk0,fsl0_s_data,
10                   fsl0_s_ctl,fsl0_s_rd,fsl0_s_exists,_,_,_);
11     mbl1'target = fpga;
12     ...
13 end facet;
```

with one memory, `mbl1_bram`, and three communication bridges, `mbl1_imem`, `mbl1_dmem`, and `fsl0`.

Physical resources defined in one of the three supported domains do not directly describe the logic required to create an implementation of that resource. This logic is contained within RRT and RST. Instead, physical resource facets simply represent the link between the Rosetta language and RRT. As such, behavioral terms and structural terms are not allowed in physical resource facets and so these facets tend to be implemented as facet interfaces. One exception is that physical resource facets are allowed to contain synthesis attributes as these are related to the physical system being described.

Because physical resources are a link between the Rosetta language and RRT, each resource is essentially a container that describes the information necessary for RRT to create an implementation of that facet. This information is encoded as declarations that RRT examines to determine the type of physical resource being described. As a consequence, the set of supported physical resources is limited to those physical resources understood by RRT, i.e. new physical resources are exposed by modifying RRT not by modifying the specification being implemented.

Listing 5.4: Rosetta Implementation Target

```
1 facet genesys( sys_clk :: input bit;
2                sys_rst :: input bit;
3                sys_rs232_rx :: input bit;
4                sys_rs232_tx :: output bit )
5                             :: target is
6     language :: string is ”vhdl”;
7     device   :: string is ”xc5vlx50tff1136-1”;
8 begin
9     sys_clk’io_pin = ”AG18”;
10    sys_clk’io_std = ”LVCMOS33”;
11    sys_clk’period = 10 ∗ ns;
12    sys_clk’sigis  = ”CLK”;
13    ...
14 end facet;
```

Target Physical Resources

Physical resources in the `target` domain are used to define target fabrics that can be used to implement the logical components instantiated in the top-level design. As such, target fabrics capture the notion of computation in the physical system described in the top-level design. Listing 5.4 shows, as an example, the target fabric `genesys` defined as a Rosetta facet in the `target` domain.

As shown, `target` physical resources can have parameters, declarations, and terms just like normal Rosetta facets. The parameters are combined with the synthesis attribute terms in the facet's body to determine the I/O pins associated with target fabric. For instance, the `genesys` facet that is shown describes a `sys_clk` parameter that is given an I/O pin, an I/O standard, a clock period, and a signal interface using several synthesis attributes. The top-level refinement uses this information to create a corresponding physical pin in the Xilinx project.

The declarations in the facet are examined to determine several pieces of information required to create implementations for logical components that are mapped onto the described target fabric. Every target fabric must contain a declaration named `language`. This declaration must be of type `string` and must have a constant value that is either "vhdl" or "mblaze". If the

language is “vhdl” then RRT will implement all logical components mapped to the target fabric using RST’s hardware circuit generation capabilities. Likewise, if the language is “mblaze” RRT will use RST’s software generation capabilities to create implementations. Currently, these are the only two supported output languages but, with additional support in RRT, other languages could be supported as well. If the `language` is defined as “vhdl” then the `target` fabric must also contain a declaration named `device` that is of type `string` with a constant value. This string is used by RRT to determine the Xilinx device that the target fabric describes. Currently, it is simply passed on to the Xilinx tools via the generated Xilinx project.

In addition to the required `language` and `device` declarations, an optional set of port declarations can be contained in the facet. Port declarations capture the notion of physical communication end-points. Connections between target fabrics and communication bridges are formed by connecting their declared ports together. The target facet shown in listing 5.4 does not contain any port declarations but an example can be seen in section 5.6.

Currently, RRT requires that there only be one “root” target fabric with roots defined as target fabrics that have not been included in any another target fabric. The root target fabric must be an FPGA device with a “vhdl” `language` declaration. Conversely, some target fabrics require implementation using another fabric. For instance, the MicroBlaze microprocessor fabric is a configurable soft-processor that must be implemented inside of an FPGA, meaning that the MicroBlaze target fabric must be included inside of an FPGA target fabric. The inclusion of one target fabric inside of another target fabric is described by the system engineer in the top-level design using a `target` attribute that is attached to the included target fabric’s instantiation. As an example, line 11 of listing 5.3, shows how the `mbl1` target fabric is included inside of the `fpga` target fabric.

Bridge Physical Resources

Physical resources in the `bridge` domain are used to to define communication links between two or more physical resources. As such, bridges capture the notion of communication in

Listing 5.5: Rosetta Communication Bridge

```
1 facet interface fsl_v20(
2     C_EXT_RESET_HIGH,C_ASYNC_CLKS,C_IMPL_STYLE,
3     C_USE_CONTROL, C_FSL_DWIDTH,C_FSL_DEPTH,
4     C_FSL_READ_CLOCK_PERIOD :: design integer;
5     FSL_Clk, SYS_Rst :: input bit;
6     FSL_Rst :: output bit; FSL_M_Clk :: input bit;
7     FSL_M_Data :: input word(C_FSL_DWIDTH);
8     FSL_M_Control, FSL_M_Write :: input bit;
9     FSL_M_Full :: output bit; FSL_S_Clk :: input bit;
10    FSL_S_Data :: output word(C_FSL_DWIDTH);
11    FSL_S_Control :: output bit; FSL_S_Read :: input bit;
12    FSL_S_Exists,FSL_Full,FSL_Has_Data,
13    FSL_Control_IRQ :: output bit
14    ) :: bridge is
15    implementation :: string is "fsl";
16    master         :: port("fsl", "master");
17    slave          :: port("fsl", "slave");
18 end facet interface;
```

the physical system described in the top-level design. Listing 5.5 shows, as an example, the communication bridge `fsl_v20` defined as a Rosetta facet in the `bridge` domain.

Like target fabrics, communication bridges are allowed to have parameters, declarations, and terms. In this case, the parameters represent the physical wires that are used to perform the physical communication. Similar to the `language` declaration in target fabrics, bridge fabrics must contain a `implementation` declaration that is of type `string` and that has a constant value of "plb", "lmb", "fsl", or "mdm". These values correspond to the Processor Local Bus (PLB) communication primitive, the Local Memory Bus (LMB) communication primitive, the Fast Simplex Link (FSL) communication primitive, and the MicroBlaze Debug Module (MDM) communication primitive. These communication primitives represent the communication protocols that can be implemented by RRT.

Like target fabrics, communication bridges can also optionally declare a set of ports to describe the communication end-points. These ports are connected to other ports in the top-level design to form communication links in the physical system. Listing 5.5 shows the definition of

Listing 5.6: Rosetta Memory Resource

```
1 facet interface bram_memory(MEMSIZE :: design integer) :: memory is
2     implementation :: string is "bram";
3     plba :: port("plb", "slave");
4     plbb :: port("plb", "slave");
5     lmba :: port("lmb", "slave");
6     lmbb :: port("lmb", "slave");
7 end facet interface;
```

two ports, `master` and `slave`. These two ports model the physical FSL primitive which implements a unidirectional, point-to-point communication topology with one master and one slave.

RRT allows ports to be declared as single end-points as shown, as a finite set of end-points using a declaration of type `array`, or as an infinite set of end-points using a declaration of type `sequence`. Ports are defined using two parameters that describe the end-point type, "fsl" in the example shown, and the end-point direction, "master" or "slave" in the example shown.

Currently, all bridges that can be implemented by RRT require an implementation inside of an FPGA because all of the supported bridges correspond to Xilinx IP cores. As such, bridges must be included inside of an FPGA target fabric using a `target` attribute attached to the bridge instantiation inside of the top-level design, much like the MicroBlaze microprocessor.

Memory Physical Resources

Physical resources in the `memory` domain are used to defined storage that can be connected to one or more physical resources. Memories capture the notion of data storage in the physical system described in the top-level design. Listing 5.6 shows, as an example, the memory `bram_memory` defined as a Rosetta facet in the `memory` domain.

Like target fabrics and communication bridges, parameters, declarations, and terms are supported for physical memories. Parameters are examine by RRT and are used to determine how to implement the physical memory. Memories must have a declaration named `imple-`

`mentation`. In the case of memories, the value of this declaration must be either “mpmc” or “bram”. These correspond to the Multiport Memory Controller (MPMC) primitive and Block RAM (BRAM) primitive respectively and represent the two types of physical memory that RRT can create.

Additionally, an optional set of ports can be declared for the memory just like for target fabrics and communication bridges. These ports are used to connect the memory to other physical resources. In the example shown, there are four ports allowing the memory to be connected either to a PLB communication bridge or to a LMB communication bridge. Like bridges, all of the memories that can be implemented by RRT correspond to Xilinx IP cores and must be included inside of an FPGA target fabric using a `target` attribute in the top-level design.

5.3 System Partitioning

Once RRT has identified all physical resources, the system partition mapping logical components to physical resources is identified by examining the `target` attributes and I/O connections present in the top-level design. As part of this, RRT validates the system partitioning by ensuring each logical component has an associated physical resource.

Target Attributes

In a top-level design, `target` attributes are used to specify the system partitioning of facet instantiations onto physical resources. As stated, `target` attributes applied to physical resources are part of the physical system description, as such, they are not considered part of the system partitioning. To be identified as a system partitioning, the `target` attribute must be applied to a logical component and the specified target must be a physical resource. Listing 5.7 shows an example top-level design with target attributes on lines 7, 16, 19, and 22.

The target attribute on line 7 is not a system partitioning[5], but is part of the physical resource description. This leaves lines 16, 19, and 22 as the three system partitioning attributes in the top-level design . These attributes map the three logical components in the system, `lcomp0`,

[5]This is determined by examining the domain of the target facet.

Listing 5.7: System Partitioning in a Top-Level Design

```
1  facet tld :: system is
2      clk, rx, tx, wr1, wr2, rd1 :: bit;
3      idat, odat, dat1, dat2, dat3 :: word(8);
4      ...
5  begin
6      fpga: genesys(...); mbl1: mblaze(...);
7      mbl1'target = fpga;
8
9      fsl0: fsl_v20(...,wr1,dat1,...); fsl1: fsl_v20(...,wr2,dat2,...);
10     fsl2: fsl_v20(...,rd1,dat3,...);
11
12     fsl0.slave = mbl1.sfsl0; fsl1.slave = mbl1.sfsl1;
13     fsl2.master = mbl1.mfsl0;
14
15     lcomp0: uart(clk,rx,tx,idat,odat);
16     lcomp0'target = fpga;
17
18     lcomp1: comm(clk,idat,odat,wr1,dat1,wr2,dat2,rd1,dat3);
19     lcomp1'target = fpga;
20
21     lcomp2: macc();
22     lcomp2'target  = mbl1;
23
24     ...
25 end facet;
```

`lcomp1`, and `lcomp2`, to the physical resources that implement them. In this case, a hardware/-software partitioning is performed that maps the first two components to hardware circuits implemented inside of `fpga` and the third component to a software program running on `mbl1`.

I/O Parameters

The second method of partitioning logical components onto physical resources is to connect physical I/O parameters and logical I/O parameters together using a declaration in the top-level design using normal Rosetta structural design techniques. This method is used to map logical components in the top-level design to communication bridges for implementation.

As an example, listing 5.7 shows several declarations on lines 2 and 3. These declarations are used on lines 9, 10, 15, and 18 in the top-level design to connect several instantiated compo-

nents together using the normal Rosetta structural design mechanisms. Just like the `target` attribute, not all of the declarations are used to perform system partitioning. In this case, `idat` and `odat` represent a communication requirement in the logical system being implemented, not a system partitioning. The declarations `wr1`, `wr2`, `rd1`, `dat1`, `dat2`, and `dat3` represent system partitionings.[6]

RRT differentiates between declarations used to perform system partitioning and other declarations by examining the domains of the facets using the declaration as an argument. If one domain is a physical resource and the other domain is not a physical resource then the declaration is being used to partition a logical communication requirement onto a communication bridge. Otherwise, the declaration is being used either to describe a communications requirement in the logical system or to describe a physical communications link in the physical resource description. Currently, RRT requires all system partitionings involve only a single physical resource and a single logical component. Declarations not being used to describe the system partitioning can have more than two facets involved, as normal.

I/O Partitioning for Software

The method of describing the system partitioning for communication described above seems to prohibit logical component partitioned onto software target fabrics from communicating with the rest of the logical system because logical components targeted as software fabrics cannot have I/O parameters as per RST's requirements. The partitioning of logical communication onto physical communication bridges was designed this way on purpose.

For components implemented as hardware circuits, I/O parameters model communication requirements because this matches closely with the abstractions present in hardware. Any implementation of these components must provide a physical implementation that meet the requirements. In the case described above an engineer is explicitly stating that the logical communication requirement can be met in implementation by respecting the system partitioning.

[6]Other declarations would probably be partitionings as well but this is not shown in the example.

Communication requirements are not modeled in the same way for components implemented as software because the communication abstractions present in processors is different. Communication abstractions for software are in the form of microprocessor instructions that cause physical communication when executed.

Communication requirements for software bound components are described using the software synthesizable I/O instructions described in section 4.2. These instructions need not be partitioned onto a physical resource in the top-level design because the logical component's implementation, in the form of a MicroBlaze instruction stream, is executing on a MicroBlaze and thus the I/O instructions are executed on that MicroBlaze The partitioning of a logical component onto a MicroBlaze implies that all of the communication requirements in that logical component can be satisfied in the physical implementation using the processor's I/O instructions.

For correct implementation of the logical system, however, the physical I/O on the MicroBlaze must be connected correctly in the physical system description. A correct example is shown in listing 5.7. As described early, this system is partitioned into a hardware/software co-design with the first two logical components implemented as hardware circuits and the third component implemented as a software program. Hardware/software communication is achieved by partitioning the communication requirements of the first two logical components onto the three FSL communication bridges. As described, the communication requirements of the third logical component is satisfied by the MicroBlaze that the component has been partitioned onto. Hardware/software communication is accomplished in the implementation by connecting the MicroBlaze to the FSLs, on lines 12 and 13, in the physical system description.

These mechanisms for system partitioning of communication requirements onto physical resources make it difficult to understand the communication in the logical system because not all communication is explicit in the top-level design. They were chosen because they map onto the abstractions provided by hardware and software implementation fabrics. The communi-

cation refinements presented in section 6 ease this difficulty by mechanically deriving these system partitionings.

Default Partitionings

Examining listing 5.7 shows that not all of the communication requirements have been mapped to physical resources for implementation. RRT does not require all communication in the logical system to be partitioned onto a physical resource for implementation because this would be needlessly complicated. Large systems will have hundreds or thousands of communication links and mapping each one of these onto a physical resource would be extremely tedious.

If a communication requirement in the logical system is not explicitly partitioned onto a physical communication bridge then RRT relies on RST to provide a reasonable physical implementation. For software programs this has already been described, communication in software bound logical components must be represented using software synthesizable I/O instructions and are implemented as such. For hardware components, unpartitioned communication is transformed into wires. This is accomplished simply by refining any unpartitioned I/O parameters into VHDL signals. When the Xilinx synthesis tools are used to process this VHDL they will implement these signals as wires on the FPGA.

5.4 Component Implementation

After RRT has identified and validated the system partitioning the process of creating implementations for each logical component in the top-level design begins. This process is relatively simple as RST creates the actual implementations. RRT simply needs to use RST on the correct logical components.

RRT accomplishes this by visiting each target fabric in the top-level design one at a time. For each target fabric, RRT consults the system partitioning identified in the previous step to determine what logical components to implement on the target fabric. If the `language` declaration in the target fabric is “vhdl” then RRT will use RST to generate hardware circuits for

each logical component with the result being a single VHDL implementation for each logical component.

If the `language` declaration is "mblaze" then RST is used to generate LLVM for all of the logical components to form a single software program that will execute on the processor. Currently, RST does not take care to prevent the execution of one logical component from blocking another logical component. For example, communication instruction on the MicroBlaze can block if there is no data available. Care must be taken by the system engineer to prevent starvation when targeting multiple logical components at a single MicroBlaze.

5.5 Project Generation

The final step in the process of refining a top-level design into a hardware/software co-designed implementation is the generation of a Xilinx Platform Studio (XPS) project. This project includes all of the physical resources defined in the top-level design with each resource implementing the appropriate logical components as per the top-level design's system partitioning.

Most of the files generated during this process are simple templates that contain the same information for all projects generated by RST. Examples of these template output files are contained in appendix F. As an example, the Makefile generated for every project is identical while the XMP file generated for each project differs only in the Xilinx device number and number of MicroBlaze programs.

The exceptions are the user constraints file (UCF), the microprocessor hardware specification (MHS), and the IP core generated to implement the logical components targeted at the root target fabric in the top-level design. The constraints file is generated by gathering all of the synthesis attributes from the physical system description and then combining those with constraints generated by RRT during the processing of each physical resource[7]. These final set of constraints are then converted into the appropriate format for the Xilinx synthesis tools. An example constraints file is shown in appendix F.

[7]For example, RRT will create constraints to ensure the correct operation of DDR memories.

The MHS file describes all of the physical resources defined in the top-level design in the format required by the Xilinx synthesis tool. Every component declared in the MHS file, with one exception, corresponds directly to one of the target fabrics, communication bridges, or storage memories defined in the top-level design. RRT generates the definitions contained in the MHS file by examining each physical resource in the top-level design one at a time. RRT processes each resource with logic that is specific to that resource, this is why each resource has a `language` or `implementation` declaration. This logic gathers the information necessary to implement the physical resource using the Xilinx synthesis tools and forms a definition for the MHS file as appropriate. An example MHS file is shown in appendix F.

The one exception to the direct correspondence between physical resources in the top-level design and definitions in the MHS file is the IP core generated by RRT as to implement the hardware circuits targeted on the root target fabric. During the process of implementing all of the logical components in the top-level design all of the components targeted at the root target fabric were implemented as VHDL. These VHDL files represent the bulk of the IP core. In addition to the VHDL implementations, a Peripheral Analyze Order (PAO) file is created, a Microprocessor Peripheral Definition is created, and a top-level IP core VHDL file is generated. The PAO file and the MPD file are required by the Xilinx tool so that the IP core can be included in the generated MHS file for inclusion in the final implementation. The PAO file simply lists all of the VHDL and Verilog files that form the definition of the IP core and RRT simply copies that information into the file. The MPD file give several pieces of information about the IP core, most importantly the top-level I/O ports exposed by the core. RRT generates this file based on the top-level VHDL file that it creates. An example of both of these files can be found in appendix F.

The top-level VHDL file is formed by examining all declarations that represent communication between a logical component in the root target fabric and logical component in a different target fabric. These declarations become the I/O interface of the top-level VHDL file. Each logical component targeted at the root target fabric is then instantiated in the top-level VHDL

file and connected, as appropriate, to the other logical components in the root target fabric. The top-level VHDL file now contains implementations of all of the logical components targeted at the root target fabric, implementations of all of the communications in the root target fabric, and implementations of all of the communications leaving or entering the root target fabric from another target fabric.

The processing performed by RRT is complete once all of the Xilinx project files have been generated. At this point the user can process the project into an actual implementation using the Xilinx synthesis tool via the generated Makefile. The Xilinx synthesis tool may fail to implement the design even if RRT does not issue an error because the RRT does not validate the synthesis constraints and only partially validates the physical system description. If the Xilinx synthesis tool successfully implements the design then the it can be loaded onto an appropriate FPGA device for execution.

5.6 Top-Level Design Example

As an example of a complete top-level design, listing 5.8 shows a simple hardware/software co-designed system specified in Rosetta using the synthesizable subset of the language. This system implements a multiply-accumulator that receives two values in over a RS-232 UART, multiplies those two values, and then adds the results to an accumulator. The Xilinx project resulting from refining this top-level design into an implementation is shown in appendix F.

As described earlier, the top-level design must contain a description of the physical resources available for the implementation, a description of the logical system behavior, and a description of the system partitioning. Figure 5.1 shows the physical resource description given in the `tld` top-level design. In this design there are two target fabrics, five communication bridges, and one storage memory. The `fpga` instantiation defines the root FPGA target fabric for the design with `mbl1` being a MicroBlaze target fabric included within that FPGA. These two physical resources provide the hardware computational resources and software computational resources that will be used to implement the logical design.

Listing 5.8: Full Top-Level Design Example

```
1  facet tld :: system is
2      sys_clk,sys_rst,rx,tx,clk0,rst,ren,ten,rrdy,trdy, f0m_ct,f0m_fl,
3      f0m_wr,f0s_ct,f0s_ex,f0s_rd,f1m_ct,f1m_fl,f1m_wr, f1s_ct,f1s_ex,
4      f1s_rd,f2m_ct,f2m_fl,f2m_wr,f2s_ct,f2s_ex,f2s_rd::bit;
5      rdat,tdat::word(8);
6      f0m_dt,f0s_dt,f1m_dt,f1s_dt,f2m_dt,f2s_dt::word(32);
7  begin
8      fpga: genesys(sys_clk,sys_rst,rx,tx);
9
10     mbl1: mblaze(clk0,rst); mbl1'target=fpga;
11     mbl1b: bram_memory(8192); mbl1b'target=fpga;
12     mbl1i: lmb_v10(clk0,rst); mbl1i'target=fpga;
13     mbl1d: lmb_v10(clk0,rst); mbl1d'target=fpga;
14
15     mbl1i.master = [mbl1.ilmb]; mbl1d.master = [mbl1.dlmb];
16     mbl1i.slave = [mmport(mbl1b.lmba,x"00000000",x"00007FFF")];
17     mbl1d.slave = [mmport(mbl1b.lmbb,x"00000000",x"00007FFF")];
18
19     f0: fsl_v20(1,0,0,0,32,1,0,clk0,rst,_,clk0,f0m_dt,f0m_ct,f0m_wr,
20                 f0m_fl,clk0,f0s_dt,f0s_ct,f0s_rd,f0s_ex,_,_,_);
21     f1: fsl_v20(1,0,0,0,32,1,0,clk0,rst,_,clk0,f1m_dt,f1m_ct,f1m_wr,
22                 f1m_fl,clk0,f1s_dt, f1s_ct,f1s_rd,f1s_ex,_,_,_);
23     f2: fsl_v20(1,0,0,0,32,1,0,clk0,rst,_,clk0,f2m_dt,f2m_ct,f2m_wr,
24                 f2m_fl,clk0,f2s_dt,f2s_ct,f2s_rd,f2s_ex,_,_,_);
25     f0'target=fpga; f1'target=fpga; f2'target=fpga;
26     f0.slave=mbl1.sfsl0; f1.slave=mbl1.sfsl1; f2.master=mbl1.mfsl0;
27
28     clks: clockgen(3,2,2.0,sys_clk,sys_rst,clk0,_,_,_,_,_,_,_,_,rst);
29     clks'target = fpga;
30
31     clk0'sigis       = "CLK";
32     clk0'clk_inport  = sys_clk;
33     rst'sigis        = "RST";
34
35     iuart: uart(clk0,rst,ten,trdy,tdat,tx,ren,rrdy,rdat,rx);
36     iuart'target = fpga;
37
38     hlpr: helper(clk0,rst,ten,tdat,rrdy,rdat,f0m_dt,f0m_ct,f0m_wr,
39                  f0m_fl,f1m_dt,f1m_ct,f1m_wr,f1m_fl,f2s_dt,f2s_ct,
40                  f2s_rd,f2s_ex);
41     hlpr'target = fpga;
42
43     mcc: macc(); mcc'target = mbl1;
44 end facet;
```

Communication is provided in the design using `f0`, `f1`, and `f2`. Each of these is an FSL with one end-point connected to the MicroBlaze processor and the other end-point left available for connecting to a hardware circuit. Two additional communications bridges, `mbl1i` and `mbl1d` are also physical resources in the design, they connect the MicroBlaze to its storage memory `mbl1b` forming the instruction and data memory communication channels. Off-chip physical I/O is present in the design in the form of `sys_clk`, `sys_rst`, `rx`, and `tx`, though this is not shown in listing 5.8. The `genesys` target resource description, as given in appendix E, identifies these as physical resources using several synthesis attributes.

Figure 5.1: Physical Description

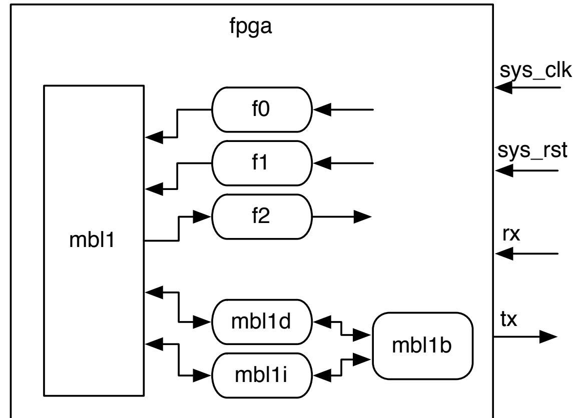


Figure 5.2 shows the logical description of the system. In this example, the logical system contains four components. Three of the components, `clks`, `iuart`, and `hlpr` are connected together directly using declarations within `tld`. The fourth component, `mcc`, is not connected directly to any other component. The important details in this figure are unconnected I/O parameters. These represent places in the logical design where there is an unmet communication requirement. This requirement must be satisfied using one of the physical communication resources. The three unconnected I/O parameters on `mcc` are not described in the top-level design but are software I/O instructions contained in the definition of the `macc` facet.

Figure 5.2: Logical Description

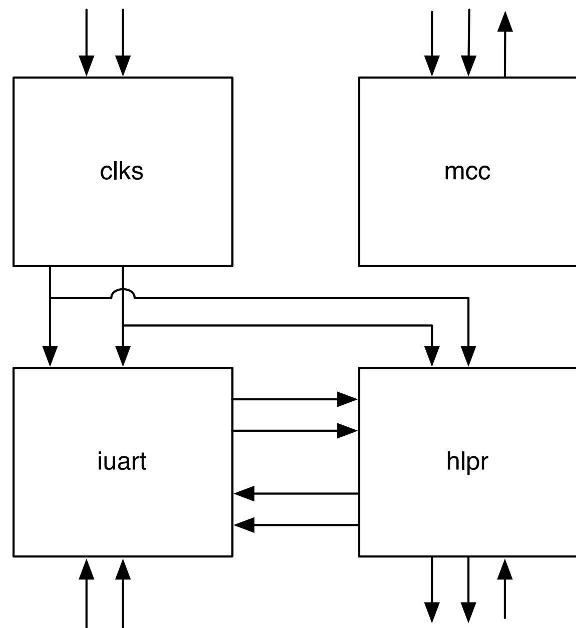
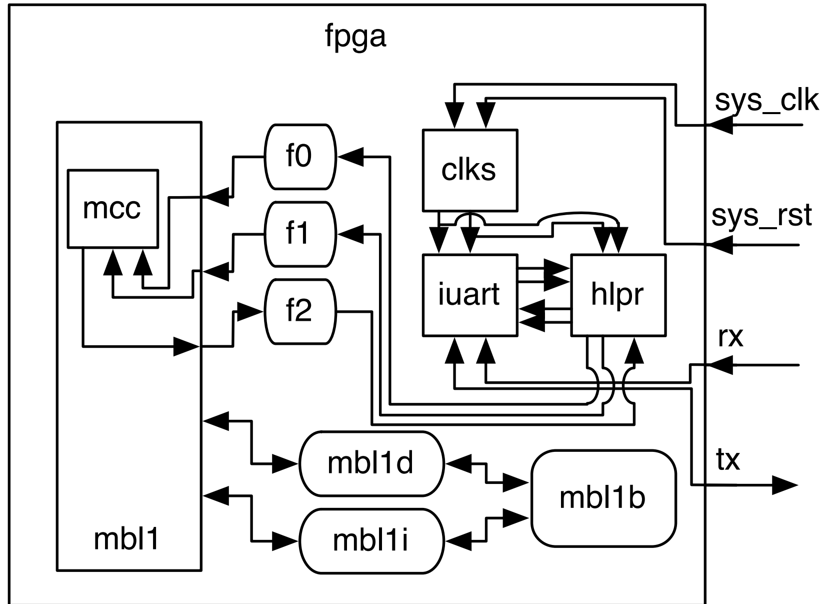


Figure 5.3: Resulting Implementation



The system partitioning is defined by several `target` attributes along with the declarations that interconnect physical resources and logical components. In this case, the top-level design describes a partitioning where three logical facets, `clks`, `iuart`, and `hlpr`, are partitioned onto `fpga` to be implemented as hardware circuits. The fourth logical component, `mcc`, is partitioned onto `mbl1` to be implemented as a software program.

As described earlier, the three unconnected communication ports on `mcc` are implemented as MicroBlaze instructions, thus, they are partitioned onto the MicroBlaze. In this case the MicroBlaze has been connected to three FSLs so the these three ports are partitioned onto the FSL communication bridges. The unconnected sides of the three FSL communication bridges are connected to the `hlpr` logical component to complete the communications links. Viewed as a whole, we have effectively formed a communications link that can cross the hardware/software boundary. The remaining four unconnected communication ports in the logical design are connected to the four I/O pins and provide off-chip communication using `fpga`.

Figure 5.3 shows the implementation that is produced for this top-level design. This figure shows that the resulting implementation is an amalgamation of the physical system and the

logical system. All of the logical component in the top-level design have been implemented on their target fabrics according to the system partitioning. Additionally, all communication in the system has been correctly wired together, with no unconnected ports appearing in the implementation. Of particular interest are the FSL communication links. These links provide hardware/software communication support in the system and are responsible for bridging the gap between the hardware domain and the software domain in this hardware/software co-designed implementation. We see in this figure that the FSL instructions used on the MicroBlaze processor to perform I/O correspond to the FSL links that are connected to the hardware circuitry running in the FPGA.

This example shows the low-level hardware/software co-design support provided by RRT. A synthesizable specification was created containing four logical components. The computation and communication requirements present in this logical design were partitioned onto two fabrics and three communication links for implementation. In the example, each of the four logical components is specified using synthesizable Rosetta constructs specific to each logical component's target fabric. For instance, the `mcc` logical component makes use of `fslread` and `fslwrite` instructions to specify communication requirements. Specifying components using target fabric specific constructs conflates the specification of functional behavior with the implementation details needed to implement that functional behavior. This makes it difficult to modify the implementation of a logical component and prohibits rapid prototyping. To solve this problem, rapid prototyping refinements can be used to refine implementation agnostic specifications into implementation specific specifications.

6 Rapid Prototyping Refinements

As stated earlier, the chief goal of RRT is to provide a rapid prototyping environment for system-level, hardware/software co-design using the Rosetta language. The top-level design refinement just described supports the notion of system-level, hardware/software co-design, leaving a need for an accompanying rapid prototyping capability. The refinements described in this section are all designed to support rapid prototyping by allowing high-level, implementation agnostic, unsynthesizable specifications in the Rosetta language to be refinements into low-level, implementation specific, synthesizable specifications by inserting simple, declarative refinements into the specification. These refinements are then used to mechanically transform the Rosetta specification as appropriate. As described in section 5, the process of discovering and mechanically applying rapid prototyping refinements is the first step performed by the top-level design refinement when a top-level design is being refined into an implementation.

RRT supports mechanically applying several different declarative refinements , each of these refinements is described in detail below. Each refinement can be classified into one of the following categories based on the Rosetta language elements that are being processed:

Behavior Refinements

The rapid prototyping capability supported by RRT is designed to allow high-level Rosetta specifications to be refinement into efficient, specialized representations for the target fabrics that they are partitioned onto for implementation. Behavioral refinements can be used by a system engineer to refine a specification into a more efficient representation for implementation by intentionally changing the behavior of the specification in suitable ways. These refinements are described in detail in section 6.2.

Data Refinements

Data refinements lower unsynthesizable data representations in a Rosetta specification into data representations that are synthesizable. Using these refinements allows a system engineer to develop a specification using high-level Rosetta data types and helps prevent implementation details from seeping into the functional behavior of the system being

specified. The data refinements supported by the RRT can then be used to mechanically derive an efficient, specialized representation of that data type for implementation on a specific target fabric.

Data refinements are applied to data types and refine any declaration or parameter of that data type inside a specific structural instantiation. Data type refinements do not materially change the functional behavior of a specification but immaterial differences may be caused due to different data access patterns. All of the data refinements supported by RRT are described in detail in section 6.3.

Communication Refinements

Communication refinements lower unsynthesizable communication constructs from the Rosetta language into representations that can be synthesized. Like data refinement, communication refinement allows specifications to be developed using easier to work with, high-level constructs and helps prevent implementation details from seeping into the behavioral requirements being described in the specification. Efficient and specialized communication constructs can then be mechanically derived for a specific target fabric using the declarative communication refinements.

Communication refinements are applied to declarations that link two difference structural instantiations via their parameters. High-level Rosetta communication expressions involving those parameters are refined into a lower-level communication protocols. Like data refinements, communication refinements only produce immaterial behavioral changes in the specification because difference protocols are accessed differently. All of the communication refinements supported by RRT are described in detail in section 6.4.

6.1 Refinement Discovery and Application

RRT discovers and applies refinements in a specification one facet at a time with all refinements applied to the specification before processing the next facet. The order that facets are visited during this stage is undefined. Care is taken by RRT to ensure that this does not lead to non-deterministic results. For instance, in listing 6.1 facets `ex2` and `tld` both have refine-

Listing 6.1: Refinement Discovery Example

```
1 facet ex1(a :: input bit; b :: input bit) :: state_based is
2 begin
3     b' = a;
4 end facet;
5
6 facet ex2(a, b :: input bit; c :: output bit) :: state_based is
7     rst :: bit;
8 begin
9     i: ex1(a or b, c);
10    active_high_reset(i, rst);
11 end facet;
12
13 facet tld :: system is
14     a, b, c, clk :: bit;
15 begin
16     i: ex2(a, b, c);
17     rising_edge_clock(i, clk);
18 end facet;
```

ments that must be eliminated, but the order that they are eliminated is undefined. Additionally, both refinements affect facet ex1 so RRT must ensure both refinements are applied in a deterministic order to that facet. This is done by ensuring that any refinements applied to an instance of a facet are applied after the refinements declared in that facet's definition.

In the example, if ex2 is refined first then reset logic will be added to facet ex1 first and clocking logic will be added after, when tld is processed. If tld is refined first then clocking logic is added to ex2 first and, as part of this, a new clock refinement is added to that facet. This new refinement is added as the last term in the facet, ensuring that the refinement will happen last. Thus, when ex2 is processed, the reset logic is added first and the clocking logic is added second and the results produced are identical.

As alluded to, processing of refinements within a single facet happens top-down. This gives control over the application of refinements to the system engineer and allows different results to be obtained by varying the order of application. For instance, synchronous reset versus asynchronous reset can be achieved by varying the order of applying reset logic and clocking logic.

Listing 6.2: Hardware Clock Refinement

```
facet ex(a :: input bit; b :: output bit) :: state_based is
    x, y :: bit;
begin
    x' = a;
    y  = a;
    z: other(x xor y, b);
end facet;

facet tld :: system is
    clk, a, b :: bit;
begin
    i: ex(a, b);
    rising_edge_clock(i, clk);
end facet;
```

6.2 Behavior Refinement

Each of the behavioral refinements supported by RRT are described below. Currently, all of these refinements are designed to help with the process of refining the behavior of Rosetta specifications so that they can be efficiently implemented as hardware circuits. There are no behavioral refinements for software bound component specifications.

Hardware Clocks via Refinement

Efficient hardware circuit implementations rely on hardware clocks to synchronize the operation of the circuit. Large pieces of combinatorial circuitry are inefficient because the propagation delay through that circuit becomes prohibitively large, especially in FPGAs because they are designed to support synchronous logic efficiently. In hardware circuits, large propagation delays result in slow clock speeds and low performance.

When developing a system specification a system engineer does not want to get bogged down with the tedious process of specifying the behavior of synchronous elements in the design, especially since all synchronous elements follow a basic pattern. The hardware clock refinements supported by RRT captures this basic pattern and allow system engineers to apply this pattern to a specification after the specification has been completed.

Listing 6.3: Refined Hardware Clock

```
facet ex(clk, a :: input bit; b :: output bit) :: state_based is
    x, y :: bit;
begin
    x' = if clk'event and clk=1 then a else x end if;
    y  = a;
    z: other(x xor y, b);
    rising_edge_clock(z, clk);
end facet;

facet tld :: system is
    clk, a, b :: bit;
begin
    i: ex(clk, a, b);
end facet;
```

The benefit over hand coding is the ease in describing synchronous behavior on many elements at once using a single declarative refinement. This supports the rapid prototyping capabilities of the refinement tool because the amount of work required to change the synchronous behavior is substantially less than if it had been hand coded.

As an example of the transformation performed by a hardware clock refinement, consider the specification given in listing 6.2. In this specification, the behavior of the facet `ex` has been fully specified without introducing any notion of clocks. The facet does, however, describe a stateful nature by making use of the high-level Rosetta support for state transitions in the form of the next state operation (the tick operator). The specification states that in the next state `x` is equivalent to `a` and that `y` is equivalent to `a` in the current state. Additionally, part of the facet's behavior is specified via structural instantiation.

The rising edge clock refinement in facet `tld` is applied to an instance of the `ex` facet and is used by the system engineer to declare that stateful transitions in `ex` should only happen on when the supplied clock is a rising edge. The effect, is that the behavior described in the original specification is changed, slightly, so that the facet can be implemented more efficiently.

The result of applying this transformation to the specification shown in listing 6.2 is shown in listing 6.3. The refinement has been eliminated from the specification and in its place three

changes have been mechanically applied. First, a clock parameter `clk` has been added to the target facet. This clock parameter is connected to the `clk` declaration in the `tld` facet via the corresponding argument in the instantiation. The name of the clock to use for the refinement is given as part of the declarative refinement term. The desired behavioral change is implemented with an if expression guarding the state transition. Instead of changing during every state transition, as was the original behavior, the refined facet now specifies that changes only happen during rising edges of `clk`. Finally, the clock refinement is propagated to all instantiations inside of the facet being transformed by adding a new clock refinement term to the end of the facet being transformed.

One important thing to notice is that the behavioral specification of the `x` declaration is updated by the refinement but the behavioral specification of `y` remains unchanged. All of the behavioral refinements performed by RRT, including the clock refinement, are concerned with behavioral refinement of stateful elements in a specification. RRT considers `x` to be a stateful element because of the presence of the next state operator.

Allowing a system engineer to distinguish stateful and non-stateful parts of a specification is important for efficient implementation of hardware circuits. Stateful behaviors correspond directly to implementation using sequential circuitry while non-stateful behaviors correspond directly to implementation using combinational circuitry. The distinction between stateful and non-stateful behaviors can be exploited by a system engineer for efficient implementation of specifications as hardware circuits.

RRT supports three different clock refinements. The rising edge clock refinement shown is one of them with the other two supported refinements being falling edge clock and double data rate clock variations. The implementation of these two refinements is nearly identical to the rising edge clock refinement. The falling edge clock implementation differs by using `clk=0` as part of the if guard and the double data rate clock implementation omits the extra guard condition altogether.

Listing 6.4: Hardware Reset Refinement

```
1 facet ex(a :: input bit; b :: output bit) :: state_based is
2     x :: bit initial is 1;  y :: bit;
3 begin
4     x' = a;
5     y  = a;
6     z: other(x xor y, b);
7 end facet;
8
9 facet tld :: system is
10    rst, a, b :: bit;
11 begin
12    i: ex(a, b);
13    active_high_reset(i, rst);
14 end facet;
```

Hardware Resets via Refinement

Reset circuitry is required in the implementation of a hardware circuit to ensure that the circuit enters into a known state before nominal behavior begins. However, specification of reset behavior is tedious and, like clocking circuitry, follows a basic pattern. Ideally, the initial state could be specified declaratively with reset behavior derived. Listing 6.4 gives an example of a specification showing how RRT allows the initial state to be declaratively stated. RRT can mechanically transform these declarative statements into the functional behavior specified in listing 6.5.

In the original specification, the initial state of the `ex` facet is specified declaratively using the `initial` attribute. In this case, the system engineer has specified an initial value of `1`. The initial value of `y` has not been specified, because the system engineer knows that this declaration is not stateful as there are no state transitions defined for the declaration. In the top-level design, the system engineer declares that the behavior of the '`i`' instantiation should be changed so that the initial state is enforced by implementing active high reset functional behavior with `rst` representing the reset signal.

In the new specification the initial state declarations are still present but new reset logic has been added. In the original specification, the value of `x` in the next state was always equivalent

Listing 6.5: Refined Hardware Reset

```
1 facet ex(rst, a :: input bit; b :: output bit) :: state_based is
2     x :: bit initial is 1; y :: bit;
3 begin
4     x' = if rst=1 then 1 else a end if;
5     y  = a;
6     z: other(x xor y, b);
7     active_high_reset(z, rst);
8 end facet;
9
10 facet tld :: system is
11     rst, a, b :: bit;
12 begin
13     i: ex(rst, a, b);
14 end facet;
```

to the parameter a. In the refined specification, the behavior is changed so that the value of x in the next state is the declared initial value if the reset signal is active and a otherwise. Just like the clock refinement, this behavioral change is implemented by guarding the transition with an additional if expression, but in this case the if expression is choosing between two values rather than limiting when state changes are allowed. Note that, like the clock refinement, y is not updated because it is not a stateful declaration. Additionally, the active high reset logic is propagated to all instantiations inside of the refined facet ex by appending a new reset refinement declaration to the end of the facet's specification.

RRT supports refining initial state declarations into both active high reset logic and active low reset logic. The only difference is the implementation of the guard expression used in the additional if expression.

Hardware Enables via Refinement

Like clocking and reset signals, enable signals are a common design element in hardware circuits. Unlike clocks and resets, enable signals do not always have a common pattern used to specify when functionality is enabled and when it is not enabled. Additionally, often times different pieces of logic in a specification will be enabled or disabled at different times, mak-

Listing 6.6: Hardware Enable Refinement

```
facet ex(a :: input bit; b :: output bit) :: state_based is
    x, y :: bit;
begin
    x' = a;
    y  = a;
    z: other(x xor y, b);
end facet;

facet tld :: system is
    ena, a, b :: bit;
begin
    i: ex(a, b);
    active_high_enable(i, ena);
end facet;
```

ing it impossible to mechanically add enable behavior to that specification. Because of these complexities, enable logic is often manually specified by a system engineer.

RRT is capable of adding enable logic to a specification in the case where all specified state transition should be inhibited by a single enable signal. If there are no manually specified enable signals in the specification then this represents typical enable behavior. If manual enable signals already exist in the specification then the additional logic acts as a "global" enable ensuring no transitions when not enabled but allowing normal transition logic otherwise.

Listing 6.6 gives an example specification showing a facet `ex` that does not restrict when the value of `x` is allow to transition. In the top-level design, the system engineer uses the active high enable refinement to declare that the behavior of the instantiation '`i`' should be modified so that no state transitions occur unless the `ena` parameter is high.

Listing 6.7 shows how RRT processes the declarative enable refinement into new functional behavior. In the refined specification, the `ex` facet now contains an additional if expression. Similar to the clock and reset refinements, this if expression modifies the behavior of the facet by specifying that state transitions are only allowed when `ena` is high. In addition to the if expression, the active high enable logic is propagated to all instantiations inside of the refined

Listing 6.7: Refined Hardware Enable

```
1 facet ex(ena, a :: input bit; b :: output bit) :: state_based is
2     x, y :: bit;
3 begin
4     x' = if ena=1 then a else x end if;
5     y  = a;
6     z: other(x xor y, b);
7     active_high_enable(z, ena);
8 end facet;
9
10 facet tld :: system is
11     ena, a, b :: bit;
12 begin
13     i: ex(ena, a, b);
14 end facet;
```

facet ex by appending a new enable refinement declaration to the end of the facet's specification.

RRT supports refining specifications to insert both active high enable logic and active low enable logic. The only difference is the implementation of the guard expression used for the added if expression.

6.3 Data Refinement

Data representation at the implementation level is an important concern with respect to efficiency in both the hardware domain and the software domain. Importantly, efficient data representation in hardware is often orthogonal to efficient data representation in software. Consider, for example, the specification of a counter as shown in listing 6.8. This specification is high-level in the sense that the data types used in the specification are abstract, mathematical types. The `integer` type used in this specification does not mean "a value of some bit-width" as in languages like C or Java but instead has the mathematical meaning "an element of the infinite set $\mathbb{Z}$".

A truly infinite set of values cannot be represented in either hardware or software implementations. To create an implementation, the system engineer must refine the data type. If the

Listing 6.8: Data Representation in a Counter

```
1 facet cntr(tick :: input bool; o :: output integer) :: state_based is
2     value :: integer initial is 0;
3 begin
4     value' = if tick then value + 1 else value end if;
5     o = value;
6 end facet;
```

specification is going to be implemented using a 32-bit processor then it would be reasonable to implement this counter using a 32-bit two's-complement representation to match the processor's direct support for this representation. If implemented as a hardware circuit, a 3-bit two's-complement representation might be better, for example, if the counter only needs to count up to eight.

To create the implementation the system engineer would need to manually modify the specification to use the newly chosen data representation. There are several problems with this approach. First, manually modifying the specification could be extremely difficult. In the example given the change from integer to a 32-bit value would be quick, but in a larger specification it would take much longer. Some synthesis tools side step this problem by allowing unsynthesizable data types to be synthesized with the understanding that the resulting implementation is not faithful to the data type contained in the specification. For example, a synthesis tool might allow the integer data type above to be synthesized by stipulating that it is always implemented using a 32-bit two's-complement representation.

The second problem is that structural design techniques allow specifications to be built up into larger specifications by including the smaller specification via instantiation and the system engineer may want different instantiations to use different data representations. There are two traditional solutions to this problem. First, the specification can be duplicated for each instantiation with so that each instantiation can make use of the desired data representation. This approach is tedious and error prone. Second, the data representation can be parameterized by a design time parameter so that the data type can be refined at the instantiation site. Effectively, a set of data types is described by the specification and one element of this set can

Listing 6.9: Data Refinement of Integer

```
1 facet cntr(tick :: input bool; o :: output integer) :: state_based is
2     value :: integer initial is 0;
3 begin
4     value' = if tick then value + 1 else value end if;
5     o = value;
6 end facet;
7
8 facet tld :: system is
9 tick :: bit; o :: word(8);
10 begin
11     i: cntr(tick, asinteger(o));
12 end facet;
```

be chosen for use at the instantiation site. This approach is common in languages like VHDL and Verilog, allowing some flexibility. This approach cannot always provide the desired level of flexibility. For instance, this approach would not allow data representation to change from a two's-complement representation to a gray-code representation.[1]

RRT alleviates these two problems by providing data type refinement as a mechanised transformation. The refinement to be applied to the specification is described by the system engineer so that control over the final representation is maintained but the refinement is applied mechanically by RRT so that the tedious and error prone process of modifying the specification is eliminated.

Refinement of Integer Types

RRT supports the refinement of integer data types into two's-complement representation as a demonstration of data refinement of unsynthesizable primitive data types. Listing 6.9 is an example system specification that includes the refinement of an integer data type into an 8-bit, two's-complement representation. The refinement is declared by the system engineer by using `asiniteger` function on `o`, an 8-bit, two's complement value.

[1]Technically this would be possible, but only by duplicating all of the logic for the two representations and then choosing between them based on a design time parameter.

Listing 6.10: Refined Integer Data Type

```
facet cntr(tick :: input bool; o :: output word(8)) :: state_based is
    value :: integer initial is 0;
begin
    value' = if tick then value + 1 else value end if;
    o = value;
end facet;

facet tld :: system is
tick :: bit; o :: word(8);
begin
    i: cntr(tick, o);
end facet;
```

The resulting specification is shown in listing 6.10. RRT has transformed the original specification to remove the function application and has implemented the data type refinement, ensuring that the argument types in the application match the parameter types in the facet definition.

Nothing else has been changed by the transformation, including the `value` declaration in the `cntr` facet that remains of type `integer`. The integer data type refinement performed by RRT relies on the RST's elastic bit-width support to figure out that `value` can be implemented as an 8-bit value because only eight bits are ever used as part of the output value `o`.

Currently, the usefulness of the integer data type refinement is limited by the simplicity of the transformation that is performed. As described earlier, this transformation can be achieved at the language level simply using design parameters to parameterize a bit-vector length. It is believed that refinement of primitive types would be useful with better support from RRT. For instance, refining the integer counter in this example into a gray-code counter would be an interesting and useful refinement but these more interesting refinements are currently not supported.

Refinement of Algebraic Data Types

More useful data refinement support is supplied by RRT in the form of algebraic data type refinement. The Rosetta language provides support for algebraic data types in the form of

Listing 6.11: Algebraic Data Type in Rosetta

```
1 machine_instr :: type is data
2       SPush(push_value :: word(8)) :: isSPush | SAdd :: isSAdd   |
3       SSub :: isSSub                          | SHalt :: isSHalt
4 end data;
```

data declarations. For instance, listing 6.11 shows an algebraic data type that models instructions for a simple stack machine. Algebraic data types are not directly representable in either hardware or software implementations and so this data type would need to be refined into a synthesizable representation if a stack machine were to be implemented.

There are several parts to this algebraic data type that must be considered when performing data type refinement. First, data values are defined by applying the constructors to their required arguments. For instance, a push value can be defined with an expression like `SPush(x"FF")`. Second, the constructor used to defined a value can be observed by applying recognizers to a value. The expression `isSPush(val)`, for instance, could be used as a guard in an if expression to determine if a value was defined using the `SPush` constructor. Lastly, arguments supplied to constructors when defining a value can be retrieved using the data type's observers. This could be done, in this case, using an expression such as `push_value(val)`, the result of which would be an 8-bit value. The constructors, recognizers, and observers defined for an algebraic data type must all be refined along with the data type representation.

Guided Refinement of Algebraic Data Types

To refine this unsynthesizable algebraic data type into a synthesizable representation the system engineer must describe the new representation to RRT. Currently there are two ways to provide this description: guided and definitional. In the guided approach, RRT generates automatic representations that meet high-level goals that are described by the system engineer. Facet `tld` in listing 6.12 shows how the high-level goals are described to RRT. The `refine_type` declaration describes to RRT that a data type refinement should be performed and, in this case, the system engineer has specified that the refinement is to be applied to the `inst` instance and

Listing 6.12: Guided Algebraic Data Type Refinement

```
1    facet tld :: system is
2    begin
3        inst: example( ... );
4        refine_type(inst, machine_instr, ”binary”,   ”packed”);
5    end facet;
```

that the `machine_instr` algebraic data type is being refined. The final two parameters describe the high-level goals.

The second to last argument, “binary”, describes how the different constructors in the algebraic data type should be represented. In this case RRT is being told to generate a simple, incrementing binary value for each tag. In this example, a 2-bit tag would be generated for each constructor with `SPush` having tag `00` and `SHalt` having tag `11`. Valid values for this argument are “binary”, “graycode”, and “onehot”. Binary representations are simple and typically used when refining algebraic data types for software implementation. Gray-code and one-hot encodings are useful representations in hardware as these encodings can reduce signal glitches, reduce transition logic, and simplify decoding logic.

The last argument, “packed”, describes how the constructor tag and constructor arguments are aligned in the new data representation. In this case the tag and all arguments are bit-packed as closely as possible in the new representation. For instance, the value `SPush(x”FF”)` would be represented as `b”0011111111”` in the example given with the tag and argument packed together. Bit-packing is an efficient representation for hardware because no bits are wasted and unpacking can be performed with no efficiency loss. For software, however, additional instruction are required to unpack the encoding. The alignment can also be “aligned” in which case the tag and constructor arguments are aligned to byte boundaries for efficient unpacking in software implementations.

Table 6.1: Guided Data Type Refinement Representations

Binary/Packed

```
Data Type      →  machine_instr::type is word(10);
Constructors   →  SPush(a::word(8))::word(10) is b"00"& a;
                  SAdd()::word(10) is b"01"& x"00";
                  SSub()::word(10) is b"10"& x"00";
                  SHalt()::word(10) is b"11"& x"00";
Recognizers    →  isSPush(v::word(10))::boolean is (v sub [9,..8])==b"00";
                  isSAdd(v::word(10))::boolean is (v sub [9,..8])==b"01";
                  isSSub(v::word(10))::boolean is (v sub [9,..8])==b"10";
                  isSHalt(v::word(10))::boolean is (v sub [9,..8])==b"11";
Observers      →  push_value(v::word(10))::word(8) is v sub [7,..0];
```

Binary/Aligned

```
Data Type      →  machine_instr::type is word(16);
Constructors   →  SPush(a::word(8))::word(16) is b"00"& b"000000"& a;
                  SAdd()::word(16) is b"01"& b"000000"& x"00";
                  SSub()::word(16) is b"10"& b"000000"& x"00";
                  SHalt()::word(16) is b"11"& b"000000"& x"00";
Recognizers    →  isSPush(v::word(16))::boolean is (v sub [15,..14])==b"00";
                  isSAdd(v::word(16))::boolean is (v sub [15,..14])==b"01";
                  isSSub(v::word(16))::boolean is (v sub [15,..14])==b"10";
                  isSHalt(v::word(16))::boolean is (v sub [15,..14])==b"11";
Observers      →  push_value(v::word(16))::word(8) is v sub [7,..0];
```

Gray-Code/Packed

```
Data Type      →  machine_instr::type is word(10);
Constructors   →  SPush(a::word(8))::word(10) is b"00"& a;
                  SAdd()::word(10) is b"01"& x"00";
                  SSub()::word(10) is b"11"& x"00";
                  SHalt()::word(10) is b"10"& x"00";
Recognizers    →  isSPush(v::word(10))::boolean is (v sub [9,..8])==b"00";
                  isSAdd(v::word(10))::boolean is (v sub [9,..8])==b"01";
                  isSSub(v::word(10))::boolean is (v sub [9,..8])==b"11";
                  isSHalt(v::word(10))::boolean is (v sub [9,..8])==b"10";
Observers      →  push_value(v::word(10))::word(8) is v sub [7,..0];
```

Gray-Code/Aligned

```
Data Type      →  machine_instr::type is word(16);
Constructors   →  SPush(a::word(8))::word(16) is b"00"& b"000000"& a;
                  SAdd()::word(16) is b"01"& b"000000"& x"00";
                  SSub()::word(16) is b"11"& b"000000"& x"00";
                  SHalt()::word(16) is b"10"& b"000000"& x"00";
```

continued on next page

Table 6.1: Guided Data Type Refinement Representations (continued)

Recognizers	$\rightarrow$	isSPush(v::word(16))::boolean is (v sub [15,..14])==b”00”;
		isSAdd(v::word(16))::boolean is (v sub [15,..14])==b”01”;
		isSSub(v::word(16))::boolean is (v sub [15,..14])==b”11”;
		isSHalt(v::word(16))::boolean is (v sub [15,..14])==b”10”;
Observers	$\rightarrow$	push_value(v::word(16))::word(8) is v sub [7,..0];
		One-Hot/Packed
Data Type	$\rightarrow$	machine_instr::type is word(12);
Constructors	$\rightarrow$	SPush(a::word(8))::word(12) is b”0001”& a;
		SAdd()::word(12) is b”0010”& x”00”
		SSub()::word(12) is b”0100”& x”00”
		SHalt()::word(12) is b”1000”& x”00”
Recognizers	$\rightarrow$	isSPush(v::word(12))::boolean is (v sub [11,..8])==b”0001”;
		isSAdd(v::word(12))::boolean is (v sub [11,..8])==b”0010”;
		isSSub(v::word(12))::boolean is (v sub [11,..8])==b”0100”;
		isSHalt(v::word(12))::boolean is (v sub [11,..8])==b”1000”;
Observers	$\rightarrow$	push_value(v::word(10))::word(8) is v sub [7,..0];
		One-Hot/Aligned
Data Type	$\rightarrow$	machine_instr::type is word(16);
Constructors	$\rightarrow$	SPush(a::word(8))::word(16) is b”0001”& x”0”& a;
		SAdd()::word(16) is b”0010”& x”0”& x”00”;
		SSub()::word(16) is b”0100”& x”0”& x”00”;
		SHalt()::word(16) is b”1000”& x”0”& x”00”;
Recognizers	$\rightarrow$	isSPush(v::word(16))::boolean is (v sub [15,..12])==b”0001”;
		isSAdd(v::word(16))::boolean is (v sub [15,..12])==b”0010”;
		isSSub(v::word(16))::boolean is (v sub [15,..12])==b”0100”;
		isSHalt(v::word(16))::boolean is (v sub [15,..12])==b”1000”;
Observers	$\rightarrow$	push_value(v::word(16))::word(8) is v sub [7,..0];

The data type, constructor, recognizer, and observer definitions generated for guided data type refinements are shown in table 6.1. These definitions are placed into the declarations section of the Rosetta facet that is being refined and then the partial evaluator, described in section 5.1, is used to ensure that the data type, constructors, recognizers, and observers are eliminated from that facet and replaced with their new definitions.

Definitional Refinement of Algebraic Data Types

The definitional approach requires more work by the system engineer but allows greater flexibility in defining the refinement. Facet `tld` in listing 6.13 shows an example of this method. The body of this facet is very similar to the previous example, the only difference being that

Listing 6.13: Definitional Algebraic Data Type Refinement

```
1  facet tld :: state_based is
2      mi2bits(mi :: machine_instr) :: (word(8),word(16),word(8)) is
3      case mi is
4             SPush(v) -> (x"00", x"1000", v)
5           | SAdd     -> (x"01", x"0100", x"00")
6           | SSub     -> (x"02", x"0010", x"00")
7           | SHalt    -> (x"03", x"0001", x"00")
8      end case;
9  begin
10     inst: example( ... );
11     refine_type(inst, machine_instr, mi2bits);
12 end facet;
```

the description of the data type refinement is given by the function `mi2bits` rather than being high-level goals given by name.

When using this method to refine an algebraic data type to a synthesizable representation the system engineer creates a special function, in this case `mi2bits`, that maps constructed values from the algebraic data type into the synthesizable representation. This function must meet several requirements or RRT will fail to apply the refinement:

1. The function's domain must be the algebraic data type being refined.
2. The function's co-domain must be a tuple and each element must be synthesizable.
3. The function's body must be a case expression that destructs the algebraic data type.
4. The case expression must match all constructors of the algebraic data type.
5. Each tuple value must include only constants and constructor arguments.
6. Each tuple value must include exactly one instances of each constructor argument.
7. Each tuple value must include at least one constant value.

If these requirements are met then RRT will refine the given algebraic data type into a concatenation of the synthesizable tuple elements. For instance, the constructed value `SPush(x"FF")` would be refined in the example given into `x"001000FF"`.

Table 6.2 shows the representation generated from the `mi2bits` function in listing 6.13. The data type generated for this data type refinement is a bit-vector with a length equal to the summation of individual elements in `mi2bits`'s co-domain. The generated constructors sim-

Table 6.2: Definitional Data Type Refinement Representation

Data Type	→	machine_instr :: **type is word**(32);
Constructors	→	SPush(a :: **word**(8)) :: **word**(10) **is** x”00”& x”1000”& a;
		SAdd() :: **word**(32) **is** x”01”& x”0100”& x”00”;
		SSub() :: **word**(32) **is** x”02”& x”0010”& x”00”;
		SHalt() :: **word**(32) **is** x”03”& x”0001”& x”00”;
Recognizers	→	isSPush(v :: **word**(32)) :: **boolean is**
		((v **sub** [31,..24]) == x”00”) **and**
		((v **sub** [23,..8]) == x”1000”);
		isSAdd(v :: **word**(32)) :: **boolean is**
		((v **sub** [31,..24]) == x”01”) **and**
		((v **sub** [23,..8]) == x”0100”) **and**
		((v **sub** [7,..0]) == x”00”);
		isSSub(v :: **word**(32)) :: **boolean is**
		((v **sub** [31,..24]) == x”02”) **and**
		((v **sub** [23,..8]) == x”0010”) **and**
		((v **sub** [7,..0]) == x”00”);
		isSHalt(v :: **word**(32)) :: **boolean is**
		((v **sub** [31,..24]) == x”03”) **and**
		((v **sub** [23,..8]) == x”0001”) **and**
		((v **sub** [7,..0]) == x”00”);
Observers	→	push_value(v :: **word**(32)) :: **word**(8) **is** v **sub** [7,..0];

ply concatenate together all of the elements of the co-domain and the observers simply bit-slice the constructor arguments from the correct location in the new representation.

The recognizers are generated by matching against all of the constant values in the co-domain of the function. Recall that RRT requires at least one constant value to appear in the result. This assures that we have something to match in the recognizer definitions. Currently, RRT does not validate uniqueness of constructor representations in the refined representation. If the data type refinement function maps different constructors to the same representation then the recognizers generated for one constructor will match values defined by the other constructor. It is the responsibility of the system engineer applying the data type refinement to avoid this situation.

Refinement of Declarations

In addition to performing algebraic data type refinement to structural instantiations, RRT can also perform refinement on declarations within a facet. For example, listing 6.14 shows a top-level design that refines the declaration ex using a data type refinement. Refinement of dec-

Listing 6.14: Data Type Refinement of Declarations

```
1 facet tld :: system is
2     ex :: machine_instr;
3 begin
4     refine_type(ex, machine_instr, "binary", "packed");
5 end facet;
```

Listing 6.15: Incorrect Application of Data Type Refinement

```
1 facet tld :: system is
2     comm :: machine_instr;
3 begin
4     inst1: example1( comm, ... );
5     refine_type(inst1, machine_instr, "binary", "packed");
6
7     inst2: example2( comm, ... );
8     refine_type(inst1, machine_instr, "graycode", "packed");
9
10    refine_type(comm, machine_instr, "onehot", "packed");
11 end facet;
```

larations works the same as refinement of instantiations but modifies the facet specification containing the declarative refinement rather than an instantiation inside of that facet.

Data Type Refinement Limitations

Currently RRT does not validate or correct situations where a common data type is refined into incompatible representations in different parts of the specification. Listing 6.15 shows a specification where `machine_instr` is refined into three incompatible representations even though the data type is used to connect the different parts of the specification. In this specification `machine_instr` uses binary tag representation in `inst1`, gray-code tag representation in `inst2`, and one-hot tag representation in the top-level design. This incompatibility may or may not be detected during the synthesis process but will probably result in run-time errors. It is the responsibility of the system engineer to prevent incompatible data type refinement. Note that if a data type is used independently in different parts of the specification then it is perfectly correct, and often desirable, to refined that data type into different representations for implementation.

Listing 6.16: Abstract Communication in Rosetta

```
1 facet sndr(ena::input bit; val::output word(32)) :: state_based is
2 begin
3     if ena=1 then val'send(x"00000000") else true end if;
4 end facet;
5
6 facet rcvr(val::input word(32); isz::output bit) :: state_based is
7 begin
8     isz' = if val'receive then (val == x"00000000") else 0 end if;
9 end facet;
```

Additionally, when refining data types all of the constructor parameters must be synthesizable when the refinement is performed. This restriction does not prevent nesting algebraic data types but does require that nested types be refined to synthesizable representations before the containing algebraic data type can be refined. Currently, RRT does not support recursive or mutually recursive data types.

6.4 Communication Refinement

Rapid prototyping of hardware/software co-designs via refinement requires the ability to represent communication between components abstractly and the ability to refine the abstract communication into physical representations at the implementation level. Listing 6.16 shows how communication between two components can be represented abstractly in the Rosetta language.

In this example the Rosetta communication synchronization attributes `send` and `receive` are applied to the output and input parameters used to perform the abstract communication. The semantics of these operations can be understood by viewing `val` as an abstract communication channel. The semantics of `send` are that it results in the value `true` only when the given value is placed into the channel. The semantics of `receive` are that it results in the value `true` only when data has been retrieved from the channel.

Essentially, send allows data to be placed into the channel and the result indicates whether or not the data was accepted for transmission. Receive allows us to determine when new data

has arrived over the communication channel. In the receiving facet, the value of parameter representing the communication channel is defined as the last value received over the channel. In the example given, the system specified has a sender that places the value `x"00000000"` into the communication channel as quickly as possible whenever `ena` is high. The receiver output `isz` indicates whether the value just received was a `x"00000000"`.

Communication Refinement in the Top-Level Design

RRT allows abstract communication in Rosetta specifications to be refined to one of two physical implementations. Currently, these refinements must be placed in the top-level design because RRT generates physical bridge resources to implement the communication and physical resource instantiations are only allowed in the top-level design.

Listing 6.17 shows an example top-level design that contains instantiations of the `sndr` and `rcvr` facets shown previously along with an FPGA target and a Fast Simplex Link (FSL) communication refinement denoted by `communications_link`. The arguments to the communication refinement are used by the system engineer to describe the type of refinement to perform, and, thus, describe the type of communication to use in the physical implementation.

The first argument, in this case "fsl", describes the communications protocol to use for the physical implementation. Valid values for this argument are either "fsl" to select a Fast Simplex Link protocol or "go/done" to select a Go/Done signaling protocol. The second argument identifies the top-level declaration that represents the high-level communication channel to be refined, in this case `val`. The number and meaning of the remaining arguments depends on the selected refinement and are discussed below.

RRT ensures that the declaration selected for refinement is used as an output parameter from a single instantiation and as an input parameter to a single instantiation. This ensures that the high-level communication channel identified by the declaration is describing a point-to-point topology, the only currently supported communication topology. It is believed that support for other communication topologies could be provided with some additional work.

Listing 6.17: Fast-Simplex Link Communication Refinement

```
1  facet tld :: system is
2      isz :: bit; val :: word(32);
3      clk, rst, ena :: bit;
4  begin
5      fpga: genesys( sys_clk, sys_rst, rx, tx );
6
7      isnd: sndr(ena, val);
8      isnd'target = fpga;
9
10     ircv: rcvr(val, isz);
11     ircv'target = fpga;
12
13     communications_link("fsl", val, clk, rst);
14 end facet;
```

Fast Simplex Link Protocol

FSL communication refinements can be described using one of two forms:

1. `communications_link("fsl", comm, clk, rst, size)`
2. `communications_link("fsl", comm, mclk, sclk, rst, size)`

The first form describes a synchronous FSL communication bridge. The third argument, `clk`, is used as the clock signal for the new FSL bridge and `rst` is used as the reset signal. The final argument, `size`, is used to determine the FIFO depth of the FSL bridge. The width of the FSL communication links is determine by the bit-width of the declarations being refined, in this case the width would be `32` because `val` is a 32-bit wide bit-vector. The `rst` and `size` arguments are both optional with `size` defaulting to `1`.

The second form describes an asynchronous FSL communication bridge. In this case there are two clocks, the master clock `mclk` for the sender and the slave clock `sclk` for the receiver. The reset and size arguments are the same as for the first form. Asynchronous FSL communication bridges can be used to communicate between two hardware bound logical components that are placed in different clock domains in the final implementation.

Listing 6.18: Refined FSL Communication Link

```
1  facet tld :: system is
2      clk,ena,f0m_ct,f0m_fl,f0m_wr,f0s_ct,f0s_ex,f0s_rd,isz,rst::bit;
3      f0m_dt,f0s_dt,val::word(32);
4  begin
5      fpga: genesys(sys_clk, sys_rst, rx, tx);
6      f0: fsl_v20(1,0,0,0,32,1,0,clk,rst,_,clk,f0m_dt,f0m_ct,f0m_wr,
7                  f0m_fl,clk,f0s_dt,f0s_ct,f0s_rd,f0s_ex,_,_,_);
8      isnd: sndr(ena,f0m_dt,f0m_ct,f0m_wr,f0m_fl);
9      ircv: rcvr(f0s_dt,f0s_ct,f0s_rd,f0s_ex,isz);
10
11     f0'target = fpga;
12     isnd'target = fpga;
13     ircv'target = fpga;
14 end facet;
15
16 facet sndr(ena :: input bit; f0m_dt :: output word(32);
17            f0m_ct :: output bit; f0m_wr :: output bit;
18            f0m_fl :: input bit) :: state_based is
19     val :: word(32);
20 begin
21     f0m_ct = 0;
22     f0m_wr = %(ena == 1);
23     f0m_dt = x"00000000";
24 end facet;
25
26 facet rcvr(f0s_dt :: input word(32); f0s_ct :: input bit;
27            f0s_rd :: output bit; f0s_ex :: input bit;
28            isz :: output bit) :: state_based is
29     val :: word(32);
30 begin
31     isz' = if f0s_ex == 1 then %(val == x"00000000") else 0 end if;
32     val = f0s_dt;
33     f0s_rd = f0s_ex;
34 end facet;
```

Example FSL Refinement

Listing 6.18 shows the specification that results from the refinement of the specification shown in listing 6.17. In the top-level design a new FSL bridge, `f0`, was added using the clock and reset signals described in the refinement. Additionally, the `val` parameter in both `sndr` and `rcvr` has been replaced by the physical FSL signals.

In the `sndr` facet, the `val` parameter was replaced by parameters used to represent the FSL signals and the abstract `send` communication primitive was replaced by assignments to the FSL signals. In this case, the refinement determined that the communication was guarded by the `ena` signal and changed this into an appropriate implementation via the `f0m_wr` signal.

Likewise, in the `rcvr` facet, the `val` parameter was replaced by the appropriate FSL signals and the abstract `receive` communication primitive was replaced by the FSL's data exists signal `f0s_ex`. The `val` parameter shows up in the refined specification as a declaration so that the uses of `val` to read the current data value of the channel do not need to be changed. In the new specification, `val` is simply defined as equivalent to the FSL communication channel's current data, `f0s_dt`. Finally, the new specification ensures that new data arriving over the FSL link is made available by making the FSL read signal, `f0s_rd`, equivalent to the FSL exists signal, `f0s_ex`.

The communication refinement performed has mapped the unsynthesizable abstract communication in the original logical design into a synthesizable representation. It then instantiated a bridge resource to describe the required communication link in the physical design and described the system partitioning required to map the communication in the logic design to the communication in the physical design.

Refinement of Abstract Communication into FSL Hardware Signaling

The abstract `receive` communication primitive in the Rosetta language can easily be refined into FSL signaling constructs in a manner that preserves the semantics of the operation. Recall that the semantics of the receive operation are that the channel variable refers to the last value

Table 6.3: Refinement of Communication into FSL Link Signaling

Rosetta Term		FSL Signaling
`comm'send(val);`	→	`f0m_ct = 0;` `f0m_wr = 1;` `f0m_dt = val;`
`if grd then comm'send(val) else true end if;`	→	`f0m_ct = 0;` `f0m_wr = %(grd);` `f0m_dt = val;`
`if comm'send(val) then thn else els end if;`	→	`f0m_ct = 0;` `f0m_wr = not f0m_fl;` `f0m_dt = val;` `if (f0m_fl == 0) then thn else els end if;`
`if grd then if comm'send(val) then thn else els end if else true end if;`	→	`f0m_ct = 0;` `f0m_wr = grd and (f0m_fl == 0);` `f0m_dt = val;` `if grd and (f0m_fl == 0) then thn else els end if;`

received on the channel and the `receive` operation results in true when a new value has been received on the channel. These semantics are implemented during refinement by specifying that the channel variable and the FSL's data signal are equivalent, meaning that the channel variable is always equal to the current FSL data value. The `receive` operation is refined into the FSL's data exists signal, specifying when new data has been received over the channel. To finalize the FSL signaling, the FSL's data read signal is specified as equivalent to the FSL's exists signal, which means that the data is read out of the channel whenever available.

The semantics of the Rosetta `send` operation do not map as directly to the FSL's write signaling as the `receive` operation. Because of this, RRT currently only recognizes and refines `send` operations that occur in the term patterns shown in table 6.3. These patterns represent the abstract `send` protocols that RRT knows how to implement. Abstract `send` operations in terms not matching those given in the table are not refined and will cause an error during synthesis. The valid terms, however, represent a large collection of different communication protocols.

In the first refinement shown, the semantics of the abstract `send` term is that an attempt to put a new value into the channel is always made and that this attempt is specified as always successful[2]. This `send` operation is refined into the FSL signaling shown. The semantics are

[2] Rosetta specifications assert terms are valid, meaning all terms must have a `true` value. As such, for this specification to be valid `send` must result in `true` and we must refine the specification to ensure this.

preserved by always attempting to write an appropriate value into the FSL communication link.

The second refinement shown adds an if-expression around the abstract `send` operation. This if-expression specifies that the semantics described for the first refinement hold only when the guard expression has a `true` value. This `send` operation is refined similarly to the first refinement with the only different being that the FSL write signal is active only when the guard expression is `true`.

The third refinement shown specifies that an attempt to put a new value into the channel is always made, but unlike the first refinement, the operation is not specified as always successful. With this form the FSL's write signal is active only when the FSL is not full, meaning that we only write into the FSL if we know that this write will be successful. The original semantics are preserved because we write into the FSL only when the FSL communication channel is not full, ensuring that the write will be successful.

The fourth form is a simple combination of the second and third refinements. The semantics of the abstract `send` operation and refinements performed are a combination these two other forms. The refined specification states that the FSL is only written to when the guard is `true` and the FSL is not full. The generated if-expression is guarded similarly.

Refinement of Abstract Communication into FSL Software Instructions

The refinement of abstract `send` and `receive` communication into software instructions is simpler than the refinement for hardware because the refinements can be mapped directly onto supported FSL instructions on the MicroBlaze processor. The `receive` operation is refined into a combination of `fslcanread` and `fslread` and is synthesized into a non-blocking FSL read instruction. The semantics of the operation are preserved because the non-blocking FSL instruction will retrieve new data from the FSL if available and indicate whether new data was available, just like the semantics of the `receive` operation.

Listing 6.19: Go/Done Communication Refinement

```
1 facet tld :: system is
2     isz :: bit; val :: word(32);
3     clk, rst, ena :: bit;
4 begin
5     fpga: genesys( sys_clk, sys_rst, rx, tx );
6
7     isnd: sndr(ena, val);
8     isnd'target = fpga;
9
10    ircv: rcvr(val, isz);
11    ircv'target = fpga;
12
13    communications_link("go/done", val);
14 end facet;
```

The `send` operation is refined into `fslwrite` and is synthesized into a blocking FSL write instruction. This refinement matches the semantics of the `send` instruction because the blocking FSL write instruction ensures that the write can be performed successfully before proceeding.

Go/Done Protocol

Unlike the FSL communication refinement, the Go/Done refinement can only refine structural instantiations targeted at hardware. Go/Done communication refinements are described using the declarative refinement `communications_link("go/done", comm)` where `comm` is the declaration representing the communication channel to refine.

Example Go/Done Refinement

Listing 6.19 the same top-level design as the FSL communication refinement example except the FSL refinement has been changed into a Go/Done refinement. The results of applying this refinement are shown in listing 6.20. In the refined specification the two facet instantiations have an additional argument, `gd`, that represents the done signal from the sender and the go signal to the receiver. A system partitioning is not defined for this signal so `gd` will eventually synthesize into a wire connecting the two communication end-points.

Listing 6.20: Refined Go/Done Communication Link

```
1  facet tld :: system is
2      isz, clk, rst, ena, gd :: bit;
3      val :: word(32);
4  begin
5      fpga: genesys(sys_clk, sys_rst, rx, tx);
6      isnd: sndr(ena, gd, val);
7      ircv: rcvr(gd, val, iszero);
8
9      isnd'target = fpga;
10     ircv'target = fpga;
11 end facet;
12
13 facet sndr(ena :: input bit; done :: output bit;
14            val :: output word(32)) :: state_based is
15 begin
16     val = x"00000000";
17     done = %(ena == 1);
18 end facet;
19
20 facet rcvr( go :: input bit; val :: input word(32);
21            isz :: output bit) :: state_based is
22     val :: word(32);
23 begin
24     isz' = if go==1 then (val == x"00000000") else 0 end if;
25 end facet;
```

The `sndr` facet was refined to include the required additional parameter and the abstract `send` operation was refined into a go/done protocol specification. The guard expression controlling the `send` operation in the original specification is used to generate the done signal. The `rcvr` facet is refined similarly by transforming the `receive` operation into the `go` signal.

Refinement of Abstract Communication into Go/Done Hardware Signaling

When refining a communication channel into a Go/Done protocol the `receive` operation is refined into the expression `go == 1` and the communication channel continues to name the last data received on the channel. Table 6.4 shows how terms involving the `send` operation are refined. If the `send` operation is specified as “always successful” in the original specification then the `done` signal is always asserted and the communication channel is specified as equiv-

Table 6.4: Refinement of Abstract Communication into Go/Done Signaling

Rosetta Term		Go/Done Signaling
`comm'send(val);`	$\rightarrow$	`done = 1; comm = val;`
`if grd then comm'send(val) else true end if;`	$\rightarrow$	`done = %(grd); comm = val;`

alent to the value being transmitted. If the `send` operation is specified as successful whenever a predicate is true then that predicate is used to assert the `done` signal. The semantics of the original specification are preserved in either case.

Unlike the FSL communication refinement, the Go/Done refinement currently does not support examining the success of the `send` operation by using `send` as the guard in an if-expression. This restriction could be lifted by implementing a ready/wait signal in addition to the Go/Done signal.

The rapid prototyping capability of RRT, in the form of declarative refinements, can be combined with the hardware/software co-design capability to form a tool capable of rapid exploration of the hardware/software co-design space. Functional behavior in the specification can be described in an architecture independent manner and refinements can be used to provide the implementation details required to form an architecture specific implementation. Using this approach provides for effective rapid prototyping because the specification of functional behavior is kept orthogonal to the specification of implementation details.

7 Evaluation

Evaluation of RRT was done by constructing and synthesizing several example specifications. These specifications where then refined into implementations running on a Xilinx Virtex-5 FPGA.[1] Each implementation was then verified for correct operation by loading the implementation onto the FPGA and then evaluating the correctness using custom software running on a standard computer and interacting with the implementation via a RS-232 communications link.

Implementation metrics for each refined design were gathered by examining the synthesis diagnostic logs generated by the Xilinx synthesis tool when synthesizing the implementation into an FPGA bitstream. These diagnostic logs contain information about the FPGA resources used by the implementation and the maximum frequency at which the implementation can operate. These metrics are used to evaluate the effectiveness of RRT with respect to hardware/software co-design efficiency.

No run-time performance measurement was done because the RS-232 communication link used in the designs is the limiting factor with respect to performance. In circuit performance measurement was not done because it is costly in terms of implementation time, because additional circuitry inserted into the design to measure performance would skew the results, and because the Xilinx area and timing diagnostics produced by the Xilinx synthesis tools are highly accurate[38].

The effectiveness of RRT's rapid prototyping capability is evaluated by tracking the amount of time involved in refining specifications into different hardware/software co-designed implementations and by determining the number of changes required to produce the new implementation.

Listing 7.1: RS-232 Loop-Back Specification

```
1  facet echo(tdat::output word(8); rdat::input word(8)) :: state_based is
2  begin
3      if rdat'receive then tdat'send(rdat) else true end if;
4  end facet;
5
6  facet tld :: system is
7      sys_clk,sys_rst,clk,rst,tx,rx::bit;
8      tdat,rdat::word(8);
9  begin
10     fpga: genesys(sys_clk,sys_rst,rx,tx);
11
12     clks: clockgen(3,2,2.0,sys_clk,sys_rst,clk,_,_,_, _,_,_,_,_,rst);
13     clks'target = fpga;
14
15     iuart: uart(tdat, tx, rdat, rx);
16     active_high_reset(iuart, rst);
17     rising_edge_clock(iuart, clk);
18     iuart'target = fpga;
19
20     iecho: echo(tdat, rdat);
21     active_high_reset(iecho, rst);
22     rising_edge_clock(iecho, clk);
23     iecho'target = fpga;
24
25     communications_link("fsl", tdat, clk, rst);
26     communications_link("fsl", rdat, clk, rst);
27
28     clk'sigis       = "CLK";
29     clk'clk_inport  = sys_clk;
30     rst'sigis       = "RST";
31 end facet;
```

7.1 RS-232 Loop-Back Example

An RS-232 loop-back example is the first specification used to to evaluate the effectiveness of RRT and is shown in listing 7.1. The `echo` facet specifies that whatever is received over the `rdat` communication channel is simply echoed back out on the `tdat` communication channel. The top-level design combines this with a clock generator and RS-232 UART to form the logical system that will be implemented. The RS-232 UART and clock generator specifications can be found in appendix G.[2]

The physical system described by the top-level design consists of one target resource, `fpga`, and two communication bridges. No memory resources are used in the initial physical system. The initial system partitioning described in the design maps the echo facet, the clock generator. and the UART facet onto `fpga` for implementation. Additionally, the communication channels `tdat` and `rdat` are mapped onto the FSL bridges for implementation.

There are six rapid prototyping refinements applied in this design. The first four, lines 16, 17, 21 and 22, are behavioral refinements used to prepare the logic inside of the echo and UART facets for implementation as hardware circuits. In this case, a rising edge clock and an active high reset are inserted into the logic in the facets enabling them to be synthesized into hardware. The other two refinements, lines 25 and 26, are communication refinements that add the FSL bridges to the physical system, replace the abstract communication primitives in the echo and UART facets with synthesizable FSL logic, and map the FSL signals onto the physical FSL bridges as appropriate.

Effectiveness of Rapid Prototyping Tools

The initial specification in listing 7.1 represents the starting point where design space exploration begins. At this point the rapid prototyping features and hardware/software co-design support of RRT can be exploited to generate varied implementations. These implementations

[1] A Genesys development board, available through Digilent Inc., containing a `LX50T` Virtex-5 FPGA with a `-1` speed grade was used for implementation and run-time evaluation of all designs described in this section.

[2] The RS-232 UART specification from the TPM specification was changed, slightly, to use Rosetta's abstract communication channel primitives for the transmit and receive data.

Table 7.1: Rapid Prototyping Changes

FSL Hardware to Go/Done Hardware Conversion		
`communications_link("fsl",tdat,...)`	→	`communications_link("gd",tdat)`
`communications_link("fsl",rdat,...)`		`communications_link("gd",rdat)`
FSL Hardware to Software Conversion		
`tdat,rdat::word(8)`	→	`tdat,rdat::word(32)`
`iecho'target = fpga`	→	`iecho'target = mbl1`

can then be compared by synthesizing FPGA bitstreams for each implementation using the Xilinx synthesis tools.

Design space exploration starts with an examination of the initial specification to select places where the implementation can be meaningfully changed. In this example, the mapping of logical components onto physical resources and the selection of communication protocol are the two most interesting implementation details. Three total implementations were created from the initial specification:

1. The specification refined into two hardware circuits using FSL protocols.
2. The specification refined into two hardware circuits using go/done protocols.
3. The specification refined into hardware and software using FSL protocols

Table 7.1 shows the changes to the initial specification required to create the two alternative implementations. In the first case only two lines are changed. The declarative communication refinements are changed to indicate that communication channels should be refined into go/done protocols instead of FSL protocols. The refinement of the initial specification into this alternative implementation took, at most, a minute to complete.

In the second case, the most interesting change is the conversion of the communication channels into 32-bit channels instead of 8-bit channels[3] and the conversion of the echo facet's target mapping. The addition of the MicroBlaze target and the changes to the echo and UART facet specifications are not shown for brevity. In total 25 lines in the initial specification were

[3]Currently, RRT and RST are not capable of FSL communication links in any width other than 32-bits due to restrictions present in the Xilinx synthesis tools. Additional work could be done on RST and RRT so that this step would not be required.

Table 7.2: Implementation Efficiency

High Performance Implementation					
	# of REGs	*# of LUTs*	*# of Slices*	*# FF Pairs*	*Max Frequency* *
FSL	64 (1%)	49 (1%)	21 (1%)	72 (1%)	120.019 MHz
Go/Done	45 (1%)	44 (1%)	17 (1%)	53 (1%)	120.019 MHz
HW/SW	1327 (4%)	1533 (5%)	568 (7%)	1989 (7%)	7.426 MHz
Low Area Implementation					
	# of REGs	*# of LUTs*	*# of Slices*	*# FF Pairs*	*Max Frequency* *
FSL	64 (1%)	47 (1%)	21 (1%)	73 (1%)	120.019 MHz
Go/Done	45 (1%)	46 (1%)	16 (1%)	55 (1%)	120.019 MHz
HW/SW	822 (2%)	1236 (4%)	427 (5%)	1447 (5%)	7.287 MHz

* The MicroBlaze in the design synthesized to a maximum frequency of 111.383 MHz and 109.302 MHz respectively for the high-performance and low-area implementations.

changed to create the second alternative implementation, these changes were implemented in approximately five minutes.

For this simple example, RRT provides a highly effective rapid prototyping platform. Two alternative implementations where generated from the initial specification in minutes with less than 30 lines of code changes. The three resulting implementations cover a large gamut of the design space for this specification and provide a basis for empirical evaluation of several non-functional requirements.

Effectiveness of Hardware/Software Co-Design

The effectiveness of RRT in generating hardware/software co-designed systems was evaluated by synthesizing the resulting implementations using the Xilinx synthesis tool. The functional correctness of each implementation was verified by loading the resulting FPGA bitstream onto the Xilinx device and performing a simple validation test. Afterwards, empirical evidence as to the efficiency of each implementation was gathered by examining the diagnostic logs produced by the Xilinx tools during synthesis. These logs provide very accurate area and performance metrics.

Table 7.2 shows the area and performance metrics gathered for each design. There are two sets of metrics shown in this table. The first set is the area and performance metrics generated when the Xilinx synthesis tool is instructed to generate a high performance implementation while the second set is the area and performance when the Xilinx synthesis tool is instructed to generate a low area implementation.[4]

The area metrics gathered from the Xilinx synthesis tool show the number of registers used, the number of six-input lookup tables used, the number of slices used, and the number of flip-flop pairs used. These four metrics give a general idea of the amount of combinational, sequential, and macro resources used in the implementation. The performance metrics are presented in the form of maximum operating frequency. For hardware circuits this represents the maximum speed at which the circuit can operate in the device. For software components this number is generated by taking the maximum frequency of the MicroBlaze processor, as reported in the diagnostics log, and dividing that by the approximate number of cycles required by the software implementation. This value represents the maximum frequency at which the software implementation is capable of transforming inputs into outputs. The number of cycles required is estimated by counting the number of instructions generated for the software implementation on the longest execution path. This estimate assumes that instructions take a single cycle to execute on the MicroBlaze, which is the case for the majority of the instructions.

At this point, the metrics for the three implementations can be compared and the system engineer can decide on a final implementation path based on some non-function requirements not formally expressed in the design. For instance, if a high-speed, low-latency design was required the full hardware implementation using go/done signaling would be a good choice because the implementation has the highest maximum frequency and the lowest latency.[5] This design would also be appropriate for a low-area implementation as it also uses the least number of resources.

[4]These are the most commonly used implementation strategies supported by the Xilinx synthesis tool.
[5]FSL protocols have a minimum one-cycle latency.

If the system engineer desires implementation flexibility and the ability to perform field updates[6] then the hardware/software implementation might be chosen because it can still process several million characters per second albeit with a higher area and latency cost.

RRT is effective as a hardware/software co-design tool for this example. The resources used by the two hardware examples are consistent with hand-coded equivalents in terms of both area and performance. The hardware/software implementation suffers only from a slightly inefficient implementation of FSL I/O on the MicroBlaze processor. A hand-coded example could used just three instructions, `get`, `put`, and `br` while the generated version uses 15. In this case the inefficiency is due to the fact that software generated for the MicroBlaze always uses a non-blocking I/O implementation while in this case a blocking implementation would be more appropriate. The inefficiency is also exaggerated by the simplicity of the example because the resulting implementation is entirely I/O based. If computation were mixed in with the I/O then the inefficiency would not be as great.

7.2 State Machine Example

The second specification created to evaluated RRT is shown in listing 7.2. This specification contains a simple algebraic data type, `state`, that defines a set of states for a state machine. The state machine transition and output rules are specified in the `stm` facet. The state machine transition rules are designed to match either "variables" consisting of all lower-case, alphabetic characters, or "integers" consisting of a sequence of digits between zero and nine. The specification of the transition rules uses the space character to separate variables and integers from one another.[7] If a variable is matched then the output "v" is produced and if an integer is matched then an output "i" is produced. The output "e" is produced whenever an input is supplied that does not match either a variable or an integer.

---

[6] In an FPGA the hardware circuit implementation could be field upgraded as well, but if the final implementation were an ASIC then this would not be possible.

[7] RST currently does not implement characters or string data types so the characters are represented using their binary equivalents via the ASCII table. In this table `0x61` is '`a`', `0x7A` is '`z`', `0x30` is '`0`', `0x39` is '`9`', `0x20` is a space, `0x76` is '`v`', `0x65` is '`e`', and `0x69` is '`i`'.

Listing 7.2: State Machine Specification

```
1  state :: type is data IDLE|VARS|INTS|DONE(done_value::word(8)) end data;
2
3  facet stm(tdata :: output word(8); rdata :: input word(8)) :: state_based is
4      st :: state initial is IDLE;
5  begin
6    st’ = case st is
7              IDLE  -> if rdata’receive then
8                          if ((rdata >= x”61”) and (rdata <= x”7A”)) then VARS
9                          elseif ((rdata >= x”30”) and (rdata <= x”39”)) then INTS
10                         else DONE(x”65”) end if
11                       else IDLE end if
12           | VARS   -> if rdata’receive then
13                          if ((rdata >= x”61”) and (rdata <= x”7A”)) then VARS
14                          elseif (rdata == x”20”) then DONE(x”76”)
15                          else DONE(x”65”) end if
16                        else VARS end if
17           | INTS   -> if rdata’receive then
18                          if ((rdata >= x”30”) and (rdata <= x”39”)) then INTS
19                          elseif (rdata == x”20”) then DONE(x”69”)
20                          else DONE(x”65”) end if
21                        else INTS end if
22           | DONE(_) -> IDLE
23           | _ -> IDLE
24         end case;
25
26   if (isDONE(st)) then tdata’send(done_value(st)) else true end if;
27 end facet;
28
29 facet tld :: system is
30   sys_clk,sys_rst,clk,rst,tx,rx::bit; tdat,rdat::word(8);
31 begin
32   fpga: genesys(sys_clk,sys_rst,rx,tx);
33
34   communications_link(”fsl”, tdat, clk, rst);
35   communications_link(”fsl”, rdat, clk, rst);
36
37   clks: clockgen(3,2,2.0,sys_clk,sys_rst,clk,_,_,_, _,_,_,_,_,rst);
38   iuart: uart(tdat, tx, rdat, rx);
39   istm: stm(tdat, rdat);
40
41   active_high_reset(iuart, rst); rising_edge_clock(iuart, clk);
42   active_high_reset(istm, rst); rising_edge_clock(istm, clk);
43   refine_type(istm, state, ”binary”, ”aligned”);
44
45   clks’target = fpga; iuart’target = fpga; istm’target = fpga;
46 end facet;
```

Like the RS-232 loop-back example, the `stm` facet is combined in the top-level design with a UART and a clock generator to form the logical system to be implemented. The initial physical system consists of a single target, `fpga`, and two FSL communication bridges. All facets in the logical system are mapped to `fpga` for implementation and the two communication channels are mapped to the FSL bridges.

There are seven rapid prototyping refinements used in the top-level design. Like the RS-232 example, four of them are used to refine the UART and `stm` facets into hardware synthesizable versions and two of them are used to implement synthesizable communication in the system. The final refinement, line 43, is a data type refinement that is used to remove the unsynthesizable algebraic data type from the specification, replacing it in the implementation with a synthesizable representation. In this case, an automatic representation is chosen with constructors represented using a binary sequence and constructor arguments bit-packed in the representation.

Effectiveness of Rapid Prototyping Tools

Again, rapid prototyping starts with an examination of the specification to determine what parts of the design space should be explored. For this example, the design space is explored with respect to data representation and system partitioning. In total 11 alternative implementations were refined from the initial specification to form the 12 implementations used for comparison.

The first five alternative implementations leave the initial system partitioning intact but change the data representation for the `state` algebraic data type. Table 7.3 shows the changes required to the initial specification to form these five alternative implementations. In each of the first five cases only a single change is required to form the new implementation with each of the six hardware designs implementing the six automatic data representations supported by RRT.

Table 7.3: Rapid Prototyping Changes

Hardware Conversion 1		
refine_type(istm,state,"bin","pck")	→	refine_type(istm,state,"bin","alg")
Hardware Conversion 2		
refine_type(istm,state,"bin","pck")	→	refine_type(istm,state,"gry","pck")
Hardware Conversion 3		
refine_type(istm,state,"bin","pck")	→	refine_type(istm,state,"gry","alg")
Hardware Conversion 4		
refine_type(istm,state,"bin","pck")	→	refine_type(istm,state,"oht","pck")
Hardware Conversion 5		
refine_type(istm,state,"bin","pck")	→	refine_type(istm,state,"oht","alg")
Hardware to Software Conversion		
active_high_reset(istm,rst) rising_edge_clock(istm,clk) istm'**target** = fpga	→	istm'**target** = mbl1

The final set of changes shown in table 7.3 show the interesting changes needed to transform each of the six hardware implementations into hardware/software co-designed implementations. Not shown for brevity are the changes required to add the MicroBlaze to the design[8] and the changes required to change the 8-bit communication channel into a 32-bit communication channel. The result is six hardware/software co-design implementations using the same six automatic data representations as the hardware only implementations.

For this example RRT is effective at exploring the design space due to the automated data refinement support. In total 25 lines of changes where required, 15 to change the initial hardware only specification into a hardware/software co-design implementation and 10 additional lines of changes to explore the five alternative data representations. All 12 implementations were created in approximately half of an hour. Most of this time was ensuring that the hardware/-

[8]Adding a MicroBlaze to a top-level design is matter of copying an existing template and placing it into the design. There are approximately 15 lines of code that need to be added to complete this addition.

Table 7.4: Implementation Efficiency

HW Packed/Aligned - High Performance Implementation					
	# of REGs	# of LUTs	# of Slices	# FF Pairs	Max Frequency
Binary	7 (0%)	14 (0%)	6 (1%)	15 (1%)	705.219 MHz
Gray – Code	7 (0%)	16 (0%)	6 (1%)	16 (1%)	848.176 MHz
One – Hot	8 (0%)	22 (0%)	10 (1%)	23 (1%)	431.593 MHz
HW Packed/Aligned - Low Area Implementation					
	# of REGs	# of LUTs	# of Slices	# FF Pairs	Max Frequency
Binary	2 (0%)	14 (0%)	8 (1%)	15 (1%)	702.247 MHz
Gray – Code	2 (0%)	15 (1%)	8 (1%)	15 (1%)	475.285 MHz
One – Hot	4 (0%)	20 (0%)	8 (1%)	22 (1%)	429.553 MHz
SW Packed Implementation					
	# of REGs	# of LUTs	# of Slices	# FF Pairs	Max Frequency*
Binary	964 (3%)	1356 (4%)	406 (6%)	1616 (6%)	1.579 MHz
Gray – Code	964 (3%)	1356 (4%)	406 (6%)	1616 (6%)	1.579 MHz
One – Hot	964 (3%)	1356 (4%)	406 (6%)	1616 (6%)	1.596 MHz
SW Aligned Implementation					
	# of REGs	# of LUTs	# of Slices	# FF Pairs	Max Frequency*
Binary	964 (3%)	1356 (4%)	406 (6%)	1616 (6%)	1.487 MHz
Gray – Code	964 (3%)	1356 (4%)	406 (6%)	1616 (6%)	1.487 MHz
One – Hot	964 (3%)	1356 (4%)	406 (6%)	1616 (6%)	1.563 MHz

* The MicroBlaze in the design synthesized to a maximum frequency of `151.630 MHz`.

software representation operated correctly before creating the five other hardware/software implementations.

Effectiveness of Hardware/Software Co-Design

For this example, empirical evaluation of the resulting implementation concentrates on the implementation characteristics of the `stm` facet because this facet contains the majority of the implementation details. Like the previous example, results are shown for both high-performance oriented synthesis and low-area oriented synthesis. For hardware implementations, the bit-packed representation and the aligned representations synthesized to the same results, so the results have been grouped together. This was because the Xilinx synthesis tool determined

that the aligned representation had unused bits and then eliminated those bits from the design. In a more complex design,[9] determining the unused bits would be impossible and so bit-packed representations would use less resources than aligned representations.

The results in this table show several interesting alternatives that could be used for a final system implementation. First, the fastest implementation is embodied by the high-performance, hardware only implementation using the gray-code data representation for the state machine. This implementation can operate at almost 850 MHz and is approximately 20% faster than the second highest performing implementation. The system engineer could also chose an implementation that balances performance and area through the low-area implementation of the hardware only design using the binary data representation. This design is only 20% slower than the fastest design but uses much less area.[10] In this example, the one-hot data representation is not desirable. In other designs this data representation might be more useful.

The software implementations of the state machine represent the slowest and highest area solutions. Software is typically slower than corresponding hardware implementation so this is expected. The large area of the software implementation compared to the hardware implementation is because the MicroBlaze processor being instantiated is heavily underutilized because of the simplicity of the state machine implemented. Larger specifications, specifications with heavy memory access patterns, or specifications using unbounded or dynamic resources would all be a better fit for software implementation. Currently, RRT lacks support for these types of specifications because recursive functions, recursive data types, and dynamic resource allocation is not supported by RST. Adding support for these constructs would be possible with extra engineering effort and would provide better use cases for software components.

Even with these drawbacks, the software representation still represents the most flexible and easily upgradable implementation. The empirical evidence shows that the one-hot, bit-backed data representation is the most efficient representation for software. This result was surprising

[9]Typically, communicating data over an I/O channel would prevent bits from being optimized out.

[10]The number of slices used is greater than some other designs. This is because the Xilinx synthesis tool is not packing logic into the slices at tightly as it could because the design is much smaller than the size of the FPGA being used for implementation.

at first because aligned data can be accessed faster than packed data on a microprocessor. The reason that packed representation is faster is that both representations use a single 32-bit value to represent the state so only a single memory access is required to fetch the state value from memory. Furthermore, the aligned data representation requires larger immediate values to encode the bit-position of the data constructors in the representation. As such, more `imm` instructions are required to encode the values in the instruction stream.

For this design, RRT is effective in generating hardware/software co-design implementations. The hardware generated by RRT is consistent with the expected results, attaining very high performance and utilizing a very small amount of logic. The software implementations use between 95 and 105 instructions to implement the state machine. This was compared to a hand-coded C version of the program that required approximately 65 instructions. The main inefficiency in the generated software was due to constructor tags and the constructor data being packed into a single 32-bit value[11] while in the hand coded version these were implemented using two different 32-bit values. This allowed more efficient access to the separate pieces of data. It is believed that RRT could be updated to support alignment of data on word boundaries, instead of just byte boundaries, to support the more efficient data representation in this example.

7.3 Trusted Platform Module Design

The final example constructed to evaluate RRT is the specification of a small subset of the Trusted Platform Module[34], with several minor additions. This specification is the largest specification used for evaluation, using about 2200 lines of Rosetta[12] and containing 35 facet definitions, 7 packages, and 14 algebraic data types. The specification is paired with an additional 312 lines of Verilog code that help implement the SHA-1 calculation. In total, 26 components are instantiated from the top-level design to form the logical system. These components

[11] In the aligned version data is aligned to byte boundaries not word boundaries.
[12] Including comments and whitespace.

range in complexity from a simple multiply-accumulate calculation up to a SHA-1 calculation. Due to the size, the complete specification has been placed in appendix G.

The logical system described in the top-level design supports processing nine different commands, seven commands from the TPM specification and two additional commands used during initial correctness testing of the communications subsystem. The nine commands are:

PCR Read:
Given a PCR index, return the 160-bit SHA-1 hash contained in the PCR register file at the given index.

PCR Extend:
Given a PCR index and a 160-bit SHA-1 hash, extend the PCR register at the given index by concatenating the existing 160-bit hash with the given 160-bit hash, performing a SHA-1 calculation on the resulting 320-bit value, and then storing the final 160-bit hash back into the PCR register file at the given index.

SHA-1 Start:
Clear out the existing SHA-1 state and initialize the state for new data.

SHA-1 Continue:
Given a 512-bit block of data, extend the existing SHA-1 state with the new data block.

SHA-1 Finish:
Given a block of data 512-bits or less, extend the existing SHA-1 state with the new data block and then return the final SHA-1 calculation.

RNG READ:
Calculate the next random number using the random number generator and return the random value.

STIR RNG:
Given a block of data, use that block to add entropy to the random number generator by mixing the data into the current state.

MUL ACC:
Given two 32-bit values, multiply the two values and add the resulting number to the current accumulator state. Return the final accumulator value as the result.

FIND MAX:
Given a 32-bit value, update the maximum value state with the maximum of the new value and the current state. Return the final value as the result.

In addition to the logic required by the commands themselves, the TPM specification contains a RS-232 UART communications subsystem that receives and decodes commands and then encodes and transmits the response, a 16x160-bit register file used to store the PCR register digests, and a pseudo-random number generator designed using linear feedback shift registers. This logic is combined in the top-level design and used as the logical system to be

implemented. The initial physical system is composed of one FPGA target, one MicroBlaze target, one BRAM memory, and eight FSL communication bridges. The initial implementation partitions all of the logical components onto the FPGA for implementation except for the multiply-accumulate facet which is partitioned onto the MicroBlaze. The FSLs are used for both hardware-to-hardware and hardware-to-software communication.

In the initial top-level design there are 40 refinement declarations used to prepare the logical system for implementation. Eight of these are used to create the FSL bridges and refine the abstract communication logic in the design into FSL logic for implementation. Of the remaining refinements, 12 behavioral refinements are used to prepare parts of the design for implementation as hardware circuitry and 20 are data type refinements used to replace the algebraic data types in the design with their chosen representation in the implementation.

Effectiveness of Rapid Prototyping Tools

Like the previous two examples, evaluation of the rapid prototyping capabilities starts with an examination of the initial specification to determine where the design space should be explored. In this example, the design space was explored with respect to communication protocols and the partitioning of components onto physical resources.

Three alternative implementations were created from the initial specification to form four total implementations for comparison. The changes required to create these alternative implementations are shown in table 7.6. The first alternative implementation uses a single MicroBlaze to implement the random number generator component, the find maximum value component, and the multiply-accumulator component. To create this alternative four behavioral refinements where removed and two partitionings were modified, totaling six lines. The changes were completed in under one minute.

The second alternative implementation removes the MicroBlaze, implementing all logical components as hardware circuits on the FPGA. As shown in the table, the interesting changes are partitioning modifications, used to change the implementation target, and the modification of

Table 7.5: Rapid Prototyping Changes

Converting SW 1 Implementation into SW 3 Implementation		

`active_high_reset(rng, rst);`
`rising_edge_clock(rng, clk)`
`rng'`**`target`**` = fpga`
	→	`rng'`**`target`**` = mbl1`

`active_high_reset(fmax, rst)`
`rising_edge_clock(fmax, clk)`
`fmax'`**`target`**` = fpga`
	→	`fmax'`**`target`**` = mbl1`

Converting SW 3 Implementation into MB 3 Implementation		
`mcc'`**`target`**` = mbl1`	→	`mcc'`**`target`**` = mbl2`
`fmax'`**`target`**` = mbl1`	→	`fmax'`**`target`**` = mbl3`
Converting SW 1 Implementation into HW Implementation		

`communications_link("fsl",macc_a,...)`
`communications_link("fsl",macc_b,...)`
`communications_link("fsl",macc_acc,...)`
`communications_link("fsl",prng_en,...)`
`communications_link("fsl",prng_sv,...)`
`communications_link("fsl",prng_val,...)`
	→	`communications_link("gd",macc_a)`
		`communications_link("gd",macc_b)`
		`communications_link("gd",macc_acc)`
		`communications_link("gd",prng_en)`
		`communications_link("gd",prng_sv)`
		`communications_link("gd",prng_val)`

`mcc'`**`target`**` = mbl1`	→	`active_high_reset(mcc, rst)`
		`rising_edge_clock(mcc, clk)`
		`mcc'`**`target`**` = fpga`

the communication protocol to go/done. In addition, 15 lines were removed from the specification to remove the MicroBlaze target and BRAM memory[13] from the physical system. In all, 24 lines of code changed and only a couple of minutes were required to perform the changes.

The third alternative implementation adds two additional MicroBlaze targets to the physical system and partitions the multiply-accumulate component, the find maximum component, and the random number generator component onto the three MicroBlaze processors. This implementation is derived from the first alternative implementation instead of the initial implementation because the changes necessary for software implementation of the three logical components were already present. From there, the most interesting change was the partitioning modification to make use of the additional processors. In addition, 30 lines of code were

[13]The BRAM memory is used as the instruction and data memory for the MicroBlaze.

Table 7.6: Implementation Efficiency

High Performance Implementation					
	# of REGs	*# of LUTs*	*# of Slices*	*# FF Pairs*	*Max Frequency*
SW 1	5167 (17%)	4466 (15%)	1818 (25%)	6670 (23%)	100.321 MHz
HW	3988 (13%)	2948 (10%)	1218 (16%)	4752 (17%)	98.270 MHz
SW 3	5222 (18%)	4626 (16%)	1893 (26%)	6934 (24%)	100.301 MHz
MB 3	7727 (26%)	7616 (26%)	2987 (41%)	10823 (38%)	100.776 MHz
Low Area Implementation					
	# of REGs	*# of LUTs*	*# of Slices*	*# FF Pairs*	*Max Frequency*
SW 1	4563 (15%)	4158 (14%)	1641 (22%)	6075 (21%)	101.245 MHz
HW	3774 (13%)	3103 (10%)	1199 (16%)	4650 (16%)	97.876 MHz
SW 3	4575 (15%)	4260 (14%)	1730 (24%)	6323 (22%)	101.605 MHz
MB 3	5908 (20%)	6285 (21%)	2159 (29%)	7929 (28%)	100.725 MHz
Software Instruction Count					
	MUL ACC	*FIND MAX*	*GET RNG*	*STIR RNG*	
SW 1	29	-	-	-	
SW 3	100	113	139	94	
MB 3	29	47	119	74	

added to create the two additional MicroBlaze processor and two additional BRAM memories. These changes took several minutes to complete.

Even for this large example, RRT is effective for exploring the design space. Four implementations were generated from the specification in a very short amount of time and the implementations cover a good amount of the design space. For large designs, the time spent synthesizing implementations into bitstreams for Xilinx FPGAs far outweighs the time spent exploring the design space. For these examples, the Xilinx synthesis tool takes well over an hour to synthesis each implementation.

Effectiveness of Hardware/Software Co-Design

The empirical measurements gathered for the four implementations are shown in table 7.6. Unlike the previous two examples, the maximum frequency of the designs containing MicroBlaze software have not been normalized in the table to account for the number of instructions

required by the software implementation because the some of the processing is still done in hardware in these designs.

Like previous examples, there are several possibilities that could be used for a final system implementation. The highest performance design is not as clear cut as in previous examples. The full hardware implementation suffers from a slightly lower maximum operating frequency because the multiplier required by the multiply-accumulate command has a long delay path. Because of this, the highest performance design depends on the relative frequency of the commands being issued to the TPM. If multiply-accumulate is a very common operation then it would be best to use the full hardware implementation. If multiply-accumulate is not a very common operation then it would be best to use the one of the hardware/software co-design implementation as the hardware circuitry containing the more common commands would be faster due to the removal of the multiply from the critical path.

If the smallest area design were desired then the full hardware implementation is clearly the best. In all of the examples given so far the amount of logic moved into software is smaller than the amount of logic required to place a MicroBlaze into the design. As mentioned before, specification more suited to software implementation such as large designs or ones using dynamic resources would make the use of microprocessors more attractive. Additionally, some Xilinx FPGAs contain existing PowerPC 405 or PowerPC 440 processors that require no FPGA resources for their use. Using software on a platform with an existing, no-cost processor would also make software more attractive for low-area implementations.

If flexibility and upgradability are important concerns then one of the three implementations containing processors would be used. The design containing three processors represents most flexible solution because it has the most unused processing power, the implementation containing one processor and implementing one logical component represents a medium amount of flexibility, and the implementation with one processor implementing three logical components represents the least amount of flexibility. The best solution would depend on the exact requirements of the system engineer.

Due to the complexity of this example, evaluation of the effectiveness of RRT with respect to hardware/software co-design is difficult because equivalent hand-coded version do not exist and it would be infeasible to construct them for comparison purposes. However, the amount of resources consumed in these implementations and the operating frequency of the circuits seems reasonable considering the complexity of the design. The communication component in the design requires a minimum of 2048 flip-flops due to the 1024-bit shift registers used to buffer command being received and results being transmitted. Additionally, the SHA-1 component requires several 160-bit and 512-bit registers to buffer hashes and data blocks during the SHA-1 calculation. These requirement alone represent a substantial portion of the implementation area.

Results from implementation of logic using software also seem to be consistent with the efficiency that could be expected from a hand-coded implementation. Implementation of the multiply-accumulate and find-maximum commands using software required 29 and 47 instructions respectively when implemented on their own processor. A substantial portion of both of these is related to performing non-blocking I/O using FSLs. The random number generator is implemented using 134 instruction, with 119 in the code path for generating random numbers and 74 in the code path for stirring the random state. The overlapping instructions are mostly I/O related. This is reasonable considering the implementation of the random number generator as three linear feedback shift registers with four taps used for each shift register.

7.4 Evaluation Summary

Overall, RRT is effective as a rapid prototyping system and as a hardware/software co-design tool. The data, communication, and behavioral refinements exposed can be leveraged effectively allowing a wide gamut of a specification's design space to be explored and abstractions present in high-level specifications can be removed with little to no impact on the resulting implementation's efficiency.

Currently, the main weakness in RRT is lack of additional support for software-like constructs. Software synthesis supports only the constructs supported for hardware synthesis, limiting the effectiveness of microprocessors in hardware/software co-designs. Expanding software synthesis to allow recursive functions, recursive data-types, and dynamic resource allocation would go a long way towards eliminating this weakness and would make processors a much more attractive option in hardware/software co-design implementations.

Limited support for these constructs could also be implemented for hardware components in the form of a defunctionalization transformation. Support for this would extend flexibility to system engineers and would allow even higher-level Rosetta specifications to be synthesized into hardware/software co-designed implementations.

8 Future Work

This work builds on the work of many others and should provide a platform for others to build upon. Several extensions of this work, of interest to both the research community and the engineering community, are discussed in this section.

Additional Program Transformations

The refinements implemented by RRT were chosen as examples of the three different classes of refinements described. However, the usefulness of a refinement tool is directly correlated to the number of refinements that are supported by the tool. To this end, RRT could be made much more useful by the implementation of additional transformations or by making existing transformations more flexible.

Leveraging Rosetta's Reflective Language Features

Currently, the refinements supported by RRT are implemented via a Haskell program that modifies a specification's AST. The Rosetta language, however, allows a specification to describe modifications to its own AST in the form of Reflective language constructs. Implementing refinements using reflection inside of the Rosetta language would bring more formalism to the refinements and would allow system engineers to develop their own refinements.

Formalizing Refinements

Many of the refinements supported by RRT change the semantics of the target specification under the direction of the system engineer. Currently, the onus of correctness is left to the engineer. This burden could be lessened if refinements not only mechanically changed the semantics of the specification but also generated additional correctness assumptions and implications. These assumptions and implications could then be used to discharge correctness proofs using automated proof tools. This would be similar in spirit to the type-check conditions generated by the PVS proof tool.

Extended Software Synthesis

Currently, support for synthesizing software programs from Rosetta specifications has several limitations that could be eliminated to better support software implementations. Structural instantiations, recursive functions, and recursive algebraic data-types are all commonly used Rosetta constructs that are not currently supported by RST for software synthesis. Implementing these features would provide make the Rosetta language far more applicable to software implementation.

Marshalling of Data Representations

Currently, any data representation used for communication must be identical on both sides of a communications link, for obvious correctness reasons. This restriction could be lifted if RRT could automatically marshall data correctly between different representations on each side of a communications link. Support for this would be beneficial in designs where processing dominates communication so separate, specialized representations for software and hardware would provide performance benefits even in the face of data marshalling.

Expansion of Supported Targets

The current implementation of RRT makes the basic assumption that specifications are being implemented as system-on-chip designs for embedded devices. The physical targets, communication bridges, and memories are all tailored for this kind of implementation. I believe that the approach taken by this work could be expanded to support other kinds of implementation strategies as well. Typical high-performance clusters, multi-threaded systems, and cloud computing applications could all be described using the concept of mapping logical components onto physical computing resources, physical communication resources, and physical storage resources.

9 Conclusion

This work resulted in two major contributions. First, the Rosetta Synthesis Tool was constructed to support synthesizing low-level Rosetta specifications into either hardware circuits via VHDL or software programs via LLVM. The low-level Rosetta subset supported by this tool is predictable, ensuring system engineers can understand the results that will be produced, and efficient, ensuring that system engineers can construct hardware circuits and software programs using the Rosetta language that meet their requirements.

Second, the Rosetta Refinement Tool was constructed to provide a rapid prototyping environment for the Rosetta specification language. This tool helps system engineers push high-level, architecture independent, unsynthesizable Rosetta specifications into low-level, architecture specific, synthesizable implementations by allowing them to describe implementation details as declarative specification refinements. These refinements are mechanically applied to the specification by the Rosetta Refinement Tool. Rapid prototyping is achieved because the declarative refinements can be easily changed and re-ordered to produce different implementations with much less work than would be required to manually change the implementation.

The two contributions can be used together to perform rapid prototyping of hardware/software co-designed implementations. Evaluation of this capability was done by constructing, refining, and synthesizing several example specifications, including the specification of a Trusted Platform Module subset that can be refined into one of many different hardware/software co-designed implementations. This evaluation verified that the two contributions are effective at pushing high-level specifications into low-level implementations via specification refinement.

Page left intentionally blank.

References

[1] Perry Alexander. *System Level Design with Rosetta*. Systems on Silicon. Morgan Kaufmann Publishers, first edition, 2006.

[2] Perry Alexander. Rosetta: Standardization at the System Level. *IEEE Computer*, 42:108–110, January 2009.

[3] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification Invited Talk. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '03, pages 249–, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Stefan Berghofer. Program Extraction in simply-typed Higher Order Logic. In *Types for Proofs and Programs (TYPES 2002), LNCS 2646*, pages 21–38. Springer, 2002.

[5] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[6] codehaus.org. *Groovy User Guide*. `http://groovy.codehaus.org/User+Guide`, October 2010.

[7] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. Implementing Mathematics with The Nuprl Proof Development System, 1986.

[8] P. Coussy, D.D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *Design Test of Computers, IEEE*, 26(4):8 –17, july-aug. 2009.

[9] H. Dayani-fard, J.I. Glasgow, and D. A. Lamb. A Study of Semi-Automated Program Construction, 1998.

[10] D.D. Gajski, Jianwen Zhu, R. Dömer, A. Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology*. Springer, 2000.

[11] T. Genssler, V. Kuttruff, and F. Brosig. *Inject/J Software Transformation Language - Language Specification*, November 2007.

[12] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, New York, NY, USA, 1993. ACM.

[13] Andy Gill and Andrew Farmer. Deriving an Efficient FPGA Implementation of a Low Density Parity Check Forward Error Corrector. In *The 16th ACM SIGPLAN International Conference on Functional Programming*, 09 2011.

[14] Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[15] Mentor Graphics. Catapult C Synthesis, 12 2011.

[16] Mentor Graphics. Handel-C Language Reference Manual, 2011.

[17] David Greaves and Satnam Singh. Distributing C# methods and threads over Ethernet-connected FPGAs using Kiwi. In *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 1–9. IEEE, 07 2011.

[18] Cordell Green. Application of theorem proving to problem solving. In *Proceesings IJCAI-69*, pages 219–239, 1969.

[19] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[20] S. Gupta, R. Gupta, N.D. Dutt, and A. Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, 2004.

[21] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC, 2001.

[22] Christoph Kreitz. Program synthesis. In *Automated Deduction - A Basis for Applications*, pages 105–134. Kluwer, 1998.

[23] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[24] Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37. ACM Press, 2003.

[25] Pierre Letouzey. Extraction in Coq: An Overview. In *CiE '08: Proceedings of the 4th conference on Computability in Europe*, pages 359–369, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] Z. Manna and R. Waldinger. Synthesis: Dreams => Programs. *IEEE Trans. Softw. Eng.*, 5(4):294–328, 1979.

[27] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[28] James McDonald and John Anton. SPECWARE - Producing Software Correct by Construction. In *Kestrel Institute Technical Report KES.U.01.3*, March 2001.

[29] Neil Mitchell and Colin Runciman. Uniform Boilerplate and List Processing. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 49–60. ACM, September 2007.

[30] H. Partsch and R. Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):199–236, 1983.

[31] Douglas Smith. KIDS: A Semi-Automatic Program Development System. *IEEE Transactions on Software Engineering — Special Issue on Formal Methods*, 16(9), September 1990.

[32] Douglas Smith. *Automating Software Design*, chapter KIDS: A Knowledge-Based Software Development System Software Development System, pages 483–514. MIT Press, 1991.

[33] Impulse Accelerated Technologies. Combining Impulse C with uClinux for MicroBlaze-based FPGAs.

[34] Trusted Computing Group. *TPM Main Specification*, July 2007. Specification 1.2 Level 3 Revision 103.

[35] Eelco Visser. The Stratego Tutorial. *Institute of Information and Computing Sciences, Universiteit*, June 2000.

[36] R.C. Waters. The Programmer's Apprentice: A Session with KBEmacs. *Software Engineering, IEEE Transactions on*, SE-11(11):1296 – 1320, nov. 1985.

[37] Xilinx. Constraints Guide, January 2010.

[38] Xilinx. Command Line Tools User Guide, March 2011.

A Synthesis Constraints

Most of the synthesis constraints supported by the Rosetta Synthesis Tool correspond to the constraints supported by the Xilinx Synthesis Tool [37]. The supported Rosetta constraint attributes are:

add_library: Add a library definition to the generated VHDL output.
add_use: Add a library use to the generated VHDL output.
async_reg: Mark a signal as an asynchronous register.
bel: Lock a signal to a given BEL site in a slice.
blknm: Assign a block name to a signal.
bufg: Map a signal to a global net.
clock_dedicated_route: Follow clock placement rules for a signal.
collapse: Collapse a combinatorial signal into all of its fanouts.
cool_clk: Save power by combining clock division and dual edge circuitry on a signal.
config_mode: Select a configuration for a dual purpose pin.
data_gate: Save power by gating a data signal.
dci_cascade: Cascade two DCI banks together.
dci_value: Set the buffer behavioral models associated with an IOB.
disable: Disable path tracing controls on a signal.
drive: Set the drive amperage of an output signal.
drop_spec: Drop timing constraints for a signal from the timing analysis.
enable: Enable path tracing controls on a signal.
enable_suspend: Set the behavior of the SUSPEND power mode on a signal.
fast: Increase the speed of an IOB on a signal.
feedback: Define a signal as a DCM feedback delay.
file: Set a file name to use for a block box instantiation.
float: Allow a tri-state signal to float.
hblknm: Assign a hierarchical block name to a signal.
hlutnm: Assign a hierarchical LUT name to a signal.
huset: Assign a HUSET name to a signal.
ibuf_delay: Add additional static delay to a signal.
ifd_delay: Add additional static delay to a signal.
inreg: Enable fast input optimization on a signal.
iob: Map a signal to an IOB.
iob_delay: Specify an IOB input path delay.
io_std: Assign an I/O standard to a signal.
keep: Prevent a signal from being merged into other elements.
keeper: Retain the value of an output net that a signal is attached to.
keep_hierarchy: Maintain the structural hierarchy throughout synthesis.
io_pin: Assign an absolute placement for an element.
lock_pins: Prevent the swapping of LUT pins.
lutnm: Assign a LUT name to an element.
map: Enable pin swapping and other merging operations.
max_delay: Set the maximum allowable delay on a signal.
max_pt: Set the maximum number of product terms on an element.
max_skew: Set the maximum allowable skew on a signal.

no_delay: Eliminate the delay built into IOB flip-flops.
no_reduce: Prevent minimization of logic terms.
open_drain: Implement an output signal as an open drain.
opt_effort: Set the optimization effort used during synthesis.
optimize: Perform optimization on an element.
period: Define the clock period for a signal.
pulldown: Pull a tri-state signal to low when not driven.
pullup: Pull a tri-state signal to high when not driven.
pwr_mode: Set an element to lower power or high power mode.
reg: Set a register implementation for a signal.
save: Prevent the removal of an unused signal.
schmitt_trigger: Configure an input signal with a Schmitt Trigger.
sigis: Set the signal interface standard of a signal.
slew: Set the slew rate of an input signal.
slow: Use slew rate limit control on a signal.
suspend: Use the SUSPEND power reduction mode on a signal.
system_jitter: Set the system jitter of an element.
tig: Ignore timing on all paths that go through a signal.
uset: Group design elements with an attached RLOC.
use_rloc: Use the attach RLOC for an element.
use_lowskewlines: Use low skew routing resources for a signal.
wire_and: Implement an element using a wired AND function.
xblk_name: Assign a block name to an element.

B MicroBlaze LLVM Backend

MicroBlaze programs are produced by the Rosetta Synthesis Tool in the form of Low-Level Virtual Machine (LLVM) assembly that is then processed by the LLVM tools into MicroBlaze assembly. Support for the MicroBlaze architecture was added to the LLVM tool-suite as part of this work. These additions to the LLVM tool-suite were contributed to the LLVM project on February 24, 2010 and are available in LLVM version 2.7 and later.

Most of the work in compiling LLVM assembly into MicroBlaze assembly is performed in the architecture independent portion of the tool-suite. The LLVM assembly is parsed, heavily optimized, and canonicalized for the target architecture before it is compiled into architecture specific assembly. The LLVM optimization framework is largely responsible for the efficiency of the MicroBlaze software programs generated by the Rosetta Synthesis Tool.

MicroBlaze architecture support was added to LLVM by creating a new target architecture definition in the LLVM source code. Currently, this architecture definition supports v4.00 and higher of the MicroBlaze processor. All of the registers and machine instructions supported by the MicroBlaze are modeled by the architecture, including optional instructions supported by the soft-processor. During code generation, the use of optional instructions is controlled by command line arguments. In addition, the LLVM MicroBlaze architecture models both the 3-stage and the 5-stage pipeline model to enhance instruction scheduling.

The MicroBlaze architecture is currently considered experimental because extensive testing has not been completed to verify the correctness of the generated assembly. The assembly was generated correctly for all of the examples in this work.

C Hardware Example VHDL Output

PRNG Output

```
1 library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_UNSIGNED.ALL; use IEEE.NUMERIC_STD.ALL;
2
3 entity prng is
4 port(clk:in std_logic; rst:in std_logic; ena:in std_logic; o:out std_logic_vector(7 downto 0));
5 end entity prng;
6
7 architecture str of prng is
8     signal and23:std_logic; signal and33:std_logic; signal idx22:std_logic; signal idx31:std_logic;
9     signal idx7:std_logic; signal not32:std_logic; signal idx8:std_logic_vector(7 downto 0);
10    signal idx9:std_logic_vector(7 downto 0); signal lfsr16:std_logic_vector(8 downto 0);
11    signal lfsr25:std_logic_vector(8 downto 0); signal lfsr35:std_logic_vector(8 downto 0);
12    signal mux10:std_logic_vector(7 downto 0);
13 begin
14    idx22 <= lfsr16(0);
15    and23 <= '1' when (ena = '1' and idx22 = '1') else '0';
16    idx31 <= lfsr16(0);
17    not32 <= not idx31;
18    and33 <= '1' when (ena = '1' and not32 = '1') else '0';
19    idx7 <= lfsr16(0);
20    idx8 <= lfsr25(7 downto 0);
21    idx9 <= lfsr35(7 downto 0);
22    mux10 <= idx8 when (idx7 = '1') else idx9;
23    o <= mux10;
24
25    lfsr16_inst : entity work.lfsr
26    generic map(IV => o"265")
27    port map(clk=>clk, rst=>rst, ena=>ena, o=>lfsr16);
28
29    lfsr25_inst : entity work.lfsr
30    generic map (IV => o"654")
31    port map(clk=>clk, rst=>rst, ena=>and23, o=>lfsr25);
32
33    lfsr35_inst : entity work.lfsr
34    generic map (IV => o"271")
35    port map(clk=>clk, rst=>rst, ena=>and33, o=>lfsr35);
36 end architecture str;
```

LFSR Output

```
1 library IEEE; use IEEE.STD_LOGIC_1164.ALL;
2 use IEEE.STD_LOGIC_UNSIGNED.ALL; use IEEE.NUMERIC_STD.ALL;
3
4 entity lfsr is
5 generic(IV:std_logic_vector(8 downto 0) := "000000000");
6 port(clk:in std_logic; rst:in std_logic; ena:in std_logic; o:out std_logic_vector(8 downto 0));
7 end entity lfsr;
8
9 architecture str of lfsr is
10    signal con14:std_logic_vector(8 downto 0); signal fb:std_logic; signal idx10:std_logic;
11    signal idx9:std_logic; signal s16_0:std_logic_vector(8 downto 0);
12    signal s16_0_e:std_logic_vector(8 downto 0); signal s16_0_v:std_logic_vector(8 downto 0);
13    signal sh:std_logic_vector(7 downto 0); signal sr:std_logic_vector(8 downto 0);
14 begin
15    s16_0_v <= IV when (rst = '1') else s16_0_e;
16    s16_0_e <= con14 when (ena = '1') else s16_0;
17    s16_0 <= s16_0_v when (rising_edge(clk)) else s16_0;
18    sr <= s16_0;
19    idx9 <= sr(4);
20    idx10 <= sr(0);
21    fb <= '1' when (idx9 = '1' xnor idx10 = '1') else '0';
22    sh <= sr(7 downto 0);
23    con14 <= sh & fb;
24    o <= sr;
25 end architecture str;
```

D Software Example MicroBlaze Output

Optimized MicroBlaze Output

```
1         .text
2         .globl  main
3         .align  2
4         .type   main,@function
5         .ent    main
6  main:
7         .frame  r1,16,r15
8         .mask   0x708000
9         addik     r1, r1, -16
10        swi       r15, r1, 0
11        swi       r20, r1, 12
12        swi       r21, r1, 8
13        swi       r22, r1, 4
14        addk      r20, r0, r0
15 $BB0_1: tnget    r3, rfsl0
16        addc r3, r0, r0
17        addk      r4, r0, r0
18        andi      r5, r3, 1
19        tnget     r3, rfsl1
20        addc r3, r0, r0
21        cmp       r4, r4, r5
22        beqid     r4, ($BB0_3)
23        nop
24        andi      r3, r3, 1
25        bneid     r3, ($BB0_1)
26        nop
27 $BB0_3: nget     r6, rfsl0
28        nget      r5, rfsl1
29        addk      r21, r0, r0
30        brlid     r15, __mulsi3
31        or        r22, r5, r6
32        addik     r4, r0, 1
33        cmp       r5, r21, r22
34        beqid     r5, ($BB0_5)
35        nop
36        addik     r4, r0, 0
37 $BB0_5: bneid    r4, ($BB0_7)
38        nop
39        addk      r21, r3, r20
40 $BB0_7: put      r21, rfsl2
41        addc r3, r0, r0
42        andi      r3, r3, 1
43        bneid     r3, ($BB0_7)
44        nop
45        brid      ($BB0_1)
46        addk      r20, r21, r0
47        .end    main
48 $tmp0:  .size   main, ($tmp0)-main
```

Unoptimized LLVM-IR Representation

```
1  @macc_state = internal global {i1,i1,i32,i32,i32} zeroinitializer
2  define internal void @macc({i1,i1,i32,i32,i32}*) {
3  _L0:  %1 = call i32 @llvm.mblaze.fsl.tnget(i32 0)
4        %2 = call i1 asm sideeffect alignstack "addc $0, r0, r0\0A", "=r"()
5        %3 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 0
6        store i1 %2, i1* %3
7        %4 = call i32 @llvm.mblaze.fsl.tnget(i32 1)
8        %5 = call i1 asm sideeffect alignstack "addc $0, r0, r0\0A", "=r"()
9        %6 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 1
10       store i1 %5, i1* %6
11       %7 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 0
12       %8 = load i1* %7
13       %9 = icmp eq i1 %8, true
14       %10 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 1
```

```
15       %11 = load i1* %10
16       %12 = icmp eq i1 %11, true
17       %13 = and i1 %9, %12
18       br i1 %13, label %_L2, label %_L1
19 _L1:  %14 = call i32 @llvm.mblaze.fsl.nget(i32 0)
20       %15 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 2
21       store i32 %14, i32* %15
22       br label %_L2
23 _L2:  %16 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 0
24       %17 = load i1* %16
25       %18 = icmp eq i1 %17, true
26       %19 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 1
27       %20 = load i1* %19
28       %21 = icmp eq i1 %20, true
29       %22 = and i1 %18, %21
30       br i1 %22, label %_L4, label %_L3
31 _L3:  %23 = call i32 @llvm.mblaze.fsl.nget(i32 1)
32       %24 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 3
33       store i32 %23, i32* %24
34       br label %_L4
35 _L4:  %25 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 0
36       %26 = load i1* %25
37       %27 = icmp eq i1 %26, true
38       %28 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 1
39       %29 = load i1* %28
40       %30 = icmp eq i1 %29, true
41       %31 = and i1 %27, %30
42       br i1 %31, label %_L6, label %_L5
43 _L5:  %32 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 2
44       %33 = load i32* %32
45       %34 = icmp eq i32 %33, 0
46       %35 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 3
47       %36 = load i32* %35
48       %37 = icmp eq i32 %36, 0
49       %38 = and i1 %34, %37
50       %39 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 4
51       %40 = load i32* %39
52       %41 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 2
53       %42 = load i32* %41
54       %43 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 3
55       %44 = load i32* %43
56       %45 = mul i32 %42, %44
57       %46 = add i32 %40, %45
58       %47 = select i1 %38, i32 0, i32 %46
59       %48 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 4
60       store i32 %47, i32* %48
61       br label %_L6
62 _L6:  %49 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 0
63       %50 = load i1* %49
64       %51 = icmp eq i1 %50, true
65       %52 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 1
66       %53 = load i1* %52
67       %54 = icmp eq i1 %53, true
68       %55 = and i1 %51, %54
69       br i1 %55, label %_L7, label %_L8
70 _L7:  ret void
71 _L8:  %56 = getelementptr inbounds {i1,i1,i32,i32,i32}* %0, i32 0, i32 4
72       %57 = load i32* %56
73       br label %_L9
74 _L9:  call void @llvm.mblaze.fsl.put(i32 %57, i32 2)
75       %58 = call i1 asm sideeffect "addc $0, r0, r0\0A", "=r"()
76       br i1 %58, label %_L9, label %_L10
77 _L10: ret void }
78 define void @main() {
79 _L0: br label %_L1
80 _L1: call void @macc({ i1, i1, i32, i32, i32 }* @macc_state)
81      br label %_L1 }
```

E Top-Level Design Resource Definitions

Genesys - Xilinx FPGA Target

```
1 facet genesys(sys_clk,sys_rst,sys_rs232_rx1::input bit; sys_rs232_tx1::output bit) :: target is
2     language :: string is "vhdl";
3     device   :: string is "xc5vlx50tff1136-1";
4 begin
5     sys_clk'io_pin="AG18"; sys_clk'io_std="LVCMOS33"; sys_clk'period=10*ns; sys_clk'sigis="CLK";
6     sys_rst'io_pin="E7";sys_rst'io_std="LVCMOS25"; sys_rst'tig=true; sys_rst'sigis="RST";
7     sys_rs232_rx1'io_pin="AG15"; sys_rs232_rx1'io_std="LVCMOS33";
8     sys_rs232_tx1'io_pin="AF19"; sys_rs232_tx1'io_std="LVCMOS33";
9 end facet;
```

MBlaze - Xilinx MicroBlaze Target

```
1 facet interface mblaze( mb_clk :: input bit;
2                         mb_rst :: input bit)
3                                :: target is
4     language :: string is "mblaze";
5     mfsl0,mfsl1,mfsl2,mfsl3,mfsl4,mfsl5,mfsl6,mfsl7::port("fsl","master");
6     mfsl8,mfsl9,mfsl10,mfsl11,mfsl12,mfsl13,mfsl14,mfsl15::port("fsl","master");
7     sfsl0,sfsl1,sfsl2,sfsl3,sfsl4,sfsl5,sfsl6,sfsl7::port("fsl","slave");
8     sfsl8,sfsl9,sfsl10,sfsl11,sfsl12,sfsl13,sfsl14,sfsl15::port("fsl","slave");
9     dplb,iplb::port("plb","master");
10    dlmb,ilmb::port("lmb","master");
11    debug :: port("mdm","master");
12 end facet interface;
```

FSL_v20 - Fast Simplex Link Bridge

```
1 facet interface fsl_v20(
2     C_EXT_RESET_HIGH :: design integer; C_ASYNC_CLKS  :: design integer; C_IMPL_STYLE :: design integer;
3     C_USE_CONTROL    :: design integer; C_FSL_DWIDTH  :: design integer; C_FSL_DEPTH  :: design integer;
4     C_FSL_READ_CLOCK_PERIOD :: design integer;
5     FSL_Clk,SYS_Rst :: input bit; FSL_Rst :: output bit; FSL_M_Clk :: input bit;
6     FSL_M_Data      :: input word(C_FSL_DWIDTH); FSL_M_Control,FSL_M_Write   :: input bit;
7     FSL_M_Full      :: output bit; FSL_S_Clk :: input bit; FSL_S_Data :: output word(C_FSL_DWIDTH);
8     FSL_S_Control   :: output bit; FSL_S_Read :: input bit;
9     FSL_S_Exists,FSL_Full,FSL_Has_Data,FSL_Control_IRQ :: output bit) :: bridge is
10    implementation :: string is "fsl";
11    master :: port("fsl","master");
12    slave  :: port("fsl","slave");
13 end facet interface;
```

LMB_v10 - Local Memory Bus Bridge

```
1 facet interface lmb_v10(clk,rst::input bit) :: bridge is
2     implementation :: string is "lmb";
3     master :: sequence(port("lmb","master"));
4     slave  :: sequence(mmport("lmb","slave"));
5 end facet interface;
```

PLB_v46 - Processor Local Bus Bridge

```
1 facet interface plb_v46(clk,rst::input bit) :: bridge is
2     implementation :: string is "plb";
3     master :: sequence(port("plb","master"));
4     slave  :: sequence(mmport("plb","slave"));
5 end facet interface;
```

MDM - MicroBlaze Debug Module Bridge

```
1 facet interface mdm( rst :: input bit ) :: bridge is
2     implementation :: string is "mdm";
3     master :: array(8,port("mdm","master"));
4     splb   :: port("plb","slave");
5 end facet interface;
```

BRAM - Block RAM Memory

```
1 facet interface bram_memory(MEMSIZE :: design integer) :: memory is
2     implementation :: string is "bram";
3     plba,plbb::port("plb","slave");
4     lmba,lmbb::port("lmb","slave");
5 end facet interface;
```

DDR - MultiPort Memory Controller DDR Memory

```
1 facet interface ddr_memory(clk0,clkdv,clk90,clk2x,rst::input bit) :: memory is
2     implementation :: string is "mpmc";
3     splb0,splb1,splb2,splb3,splb4,splb5,splb6,splb7::port("plb","slave");
4 end facet interface;
```

F Xilinx Platform Studio Project Files

Example Project Makefile - “Makefile”

```
TARGETS=all
TARGETS+=netlist
TARGETS+=bits
TARGETS+=exporttosdk
TARGETS+=init_bram
TARGETS+=ace
TARGETS+=download
TARGETS+=sim
TARGETS+=simmodel
TARGETS+=netlistclean
TARGETS+=bitsclean
TARGETS+=hwclean
TARGETS+=simclean

$(TARGETS): system.make
		@$(MAKE) -f system.make $@

system.make: data/system.tcl
		@echo "Building system.make, please wait..."
		@xps -nw -scr data/system.tcl 2>&1 1>/dev/null

data/system.tcl: system.xmp
		@echo "xload xmp system.xmp" > data/system.tcl
		@echo "exit" >> data/system.tcl

clean:
		@$(MAKE) -f system.make clean
		$(RM) data/system.tcl

.PHONY: clean $(TARGETS)
```

Example Project UCF File - “data/system.ucf”

```
NET "sys_clk_pin" IOSTANDARD = "LVCMOS33";
NET "sys_clk_pin" LOC = "AG18";
NET "sys_clk_pin" PERIOD = 10000 ps;
NET "sys_rst_pin" IOSTANDARD = "LVCMOS25";
NET "sys_rst_pin" LOC = "E7";
NET "sys_rst_pin" PULLUP;
NET "sys_rst_pin" TIG;
NET "sys_rs232_rx1_pin" IOSTANDARD = "LVCMOS33";
NET "sys_rs232_rx1_pin" LOC = "AG15";
NET "sys_rs232_tx1_pin" IOSTANDARD = "LVCMOS33";
NET "sys_rs232_tx1_pin" LOC = "AF19";
```

XPS Impact File - “etc/download.cmd”

```
setMode -bscan
setCable -p auto
identify
assignfile -p 1 -file implementation/download.bit
program -p 1
quit
```

Example IP Core PAO - "pcores/system_tld_v1_00_a/data/system_tld_v2_1_0.pao"

```
lib system_tld_v1_00_a system_tld vhdl
lib system_tld_v1_00_a tld_clks vhdl
lib system_tld_v1_00_a uart vhdl
lib system_tld_v1_00_a uartrx vhdl
lib system_tld_v1_00_a uarttx vhdl
lib system_tld_v1_00_a helper vhdl
```

Example XPS Project - "system.xmp"

```
#Please do not modify this file by hand
XmpVersion: 13.1
VerMgmt: 13.1
IntStyle: default
ModuleSearchPath: pcores
MHS File: system.mhs
Architecture: virtex5
Device: xc5vlx50t
Package: ff1136
SpeedGrade: -1
UserCmd1:
UserCmd1Type: 0
UserCmd2:
UserCmd2Type: 0
GenSimTB: 0
SdkExportBmmBit: 1
SdkExportDir: SDK/SDK_Export
InsertNoPads: 0
WarnForEAArch: 1
HdlLang: VHDL
SimModel: BEHAVIORAL
UcfFile: data/system.ucf
EnableParTimingError: 0
ShowLicenseDialog: 1
Processor: mbl1_microblaze
ElfImp: mbl1/main.elf
ElfSim:
```

XPS Bitgen File - "etc/bitgen.ut"

```
-g TdoPin:PULLNONE
-g DriveDone:No
-g StartUpClk:JTAGCLK
-g DONE_cycle:4
-g GTS_cycle:5
-g TckPin:PULLUP
-g TdiPin:PULLUP
-g TmsPin:PULLUP
-g DonePipe:No
-g GWE_cycle:6
-g LCK_cycle:NoWait
-g Security:NONE
-g Persist:No
```

Example IP Core MPD - "pcores/system_tld_v1_00_a/data/system_tld_v2_1_0.mpd"

```
BEGIN system_tld
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = VHDL
OPTION DESC = Top-level System Design
OPTION LONG_DESC = Top-level System Design
OPTION RUN_NGCBUILD = FALSE
OPTION STYLE = HDL
BUS_INTERFACE BUS = MFSL_f0, BUS_STD = FSL, BUS_TYPE = MASTER
BUS_INTERFACE BUS = MFSL_f1, BUS_STD = FSL, BUS_TYPE = MASTER
BUS_INTERFACE BUS = SFSL_f2, BUS_STD = FSL, BUS_TYPE = SLAVE

PORT sys_clk = "", DIR = I, SIGIS = CLK, CLK_FREQ = 100000000
PORT sys_rst = "", DIR = I, SIGIS = RST
PORT sys_rs232_rx1 = "", DIR = I
PORT sys_rs232_tx1 = "", DIR = O
PORT mbl1i_clk = "", DIR = O, SIGIS = CLK, CLK_INPORT = sys_clk
PORT mbl1i_rst = "", DIR = O, SIGIS = RST
PORT mbl1d_clk = "", DIR = O, SIGIS = CLK, CLK_INPORT = sys_clk
PORT mbl1d_rst = "", DIR = O, SIGIS = RST
PORT f0_fsl_clk = "", DIR = O, SIGIS = CLK, CLK_INPORT = sys_clk
PORT f0_sys_rst = "", DIR = O, SIGIS = RST
PORT f0_fsl_rst = "", DIR = I
PORT f0_fsl_m_clk = FSL_M_CLK, DIR = O, BUS = MFSL_f0
PORT f0_fsl_m_data = FSL_M_DATA, DIR = O, BUS = MFSL_f0, VEC = [31:0]
PORT f0_fsl_m_control = FSL_M_CONTROL, DIR = O, BUS = MFSL_f0
PORT f0_fsl_m_write = FSL_M_WRITE, DIR = O, BUS = MFSL_f0
PORT f0_fsl_m_full = FSL_M_FULL, DIR = I, BUS = MFSL_f0
PORT f0_full = "", DIR = I
PORT f0_has_data = "", DIR = I
PORT f0_control_irq = "", DIR = I
PORT f1_fsl_clk = "", DIR = O, SIGIS = CLK, CLK_INPORT = sys_clk
PORT f1_sys_rst = "", DIR = O, SIGIS = RST
PORT f1_fsl_rst = "", DIR = I
PORT f1_fsl_m_clk = FSL_M_CLK, DIR = O, BUS = MFSL_f1
PORT f1_fsl_m_data = FSL_M_DATA, DIR = O, BUS = MFSL_f1, VEC = [31:0]
PORT f1_fsl_m_control = FSL_M_CONTROL, DIR = O, BUS = MFSL_f1
PORT f1_fsl_m_write = FSL_M_WRITE, DIR = O, BUS = MFSL_f1
PORT f1_fsl_m_full = FSL_M_FULL, DIR = I, BUS = MFSL_f1
PORT f1_full = "", DIR = I
PORT f1_has_data = "", DIR = I
PORT f1_control_irq = "", DIR = I
PORT f2_fsl_clk = "", DIR = O, SIGIS = CLK, CLK_INPORT = sys_clk
PORT f2_sys_rst = "", DIR = O, SIGIS = RST
PORT f2_fsl_rst = "", DIR = I
PORT f2_fsl_s_clk = FSL_S_CLK, DIR = O, BUS = SFSL_f2
PORT f2_fsl_s_data = FSL_S_DATA, DIR = I, BUS = SFSL_f2, VEC = [31:0]
PORT f2_fsl_s_control = FSL_S_CONTROL, DIR = I, BUS = SFSL_f2
PORT f2_fsl_s_read = FSL_S_READ, DIR = O, BUS = SFSL_f2
PORT f2_fsl_s_exists = FSL_S_EXISTS, DIR = I, BUS = SFSL_f2
PORT f2_full = "", DIR = I
PORT f2_has_data = "", DIR = I
PORT f2_control_irq = "", DIR = I
PORT mbl1_clk = "", DIR = O, SIGIS = CLK, CLK_INPORT = sys_clk
PORT mbl1_rst = "", DIR = O, SIGIS = RST
END
```

XPS Synthesis Options - “etc/fast_runtime.opt”

```
FLOWTYPE = FPGA;
Program ngdbuild
-p <partname>;
-nt timestamp;
-bm <design>.bmm
<userdesign>;
-uc <design>.ucf;
<design>.ngd;
End Program ngdbuild
Program map
-o <design>_map.ncd;
-w;
-pr b;
-cm speed;
-global_opt speed;
-logic_opt on;
-lc auto;
-mt on;
-ol high;
-timing;
-detail;
<inputdir><design>.ngd;
<inputdir><design>.pcf;
END Program map
Program post_map_trce
-e 3;
-xml <design>_map.twx;
<inputdir><design>_map.ncd;
<inputdir><design>.pcf;
END Program post_map_trce
Program par
-w;
-ol high;
-mt 4;
<inputdir><design>_map.ncd;
<design>.ncd;
<inputdir><design>.pcf;
END Program par
Program post_par_trce
-e 3;
-xml <design>.twx;
<inputdir><design>.ncd;
<inputdir><design>.pcf;
END Program post_par_trce
```

XPS Resource Descriptions - “system.mhs”

```
PARAMETER VERSION = 2.1.0
PORT sys_clk_pin=sys_clk_pin, DIR=I, SIGIS=CLK, CLK_FREQ=100000000
PORT sys_rst_pin=sys_rst_pin, DIR=I, SIGIS=RST
PORT sys_rs232_rx1_pin=sys_rs232_rx1_pin, DIR=I
PORT sys_rs232_tx1_pin=sys_rs232_tx1_pin, DIR=O
BEGIN system_tld
    PARAMETER INSTANCE = tld
    PARAMETER HW_VER = 1.00.a
    PORT sys_clk = sys_clk_pin
    PORT sys_rst = sys_rst_pin
    PORT sys_rs232_rx1 = sys_rs232_rx1_pin
    PORT sys_rs232_tx1 = sys_rs232_tx1_pin
    PORT mbl1i_clk = mbl1i_clk
    PORT mbl1i_rst = mbl1i_rst
    PORT mbl1d_clk = mbl1d_clk
    PORT mbl1d_rst = mbl1d_rst
    PORT f0_fsl_clk = f0_fsl_clk
    PORT f0_sys_rst = f0_sys_rst
    PORT f0_fsl_rst = f0_fsl_rst
    PORT f0_full = f0_full
    PORT f0_has_data = f0_has_data
    PORT f0_control_irq = f0_control_irq
    BUS_INTERFACE MFSL_f0 = f0_fsl
    PORT f1_fsl_clk = f1_fsl_clk
    PORT f1_sys_rst = f1_sys_rst
    PORT f1_fsl_rst = f1_fsl_rst
    PORT f1_full = f1_full
    PORT f1_has_data = f1_has_data
    PORT f1_control_irq = f1_control_irq
    BUS_INTERFACE MFSL_f1 = f1_fsl
    PORT f2_fsl_clk = f2_fsl_clk
    PORT f2_sys_rst = f2_sys_rst
    PORT f2_fsl_rst = f2_fsl_rst
    PORT f2_full = f2_full
    PORT f2_has_data = f2_has_data
    PORT f2_control_irq = f2_control_irq
    BUS_INTERFACE SFSL_f2 = f2_fsl
    PORT mbl1_clk = mbl1_clk
    PORT mbl1_rst = mbl1_rst
END
BEGIN lmb_v10
    PARAMETER INSTANCE = mbl1i_lmb
    PARAMETER HW_VER = 2.00.a
    PORT LMB_Clk = mbl1i_clk
    PORT SYS_Rst = mbl1i_rst
END
BEGIN lmb_v10
    PARAMETER INSTANCE = mbl1d_lmb
    PARAMETER HW_VER = 2.00.a
    PORT LMB_Clk = mbl1d_clk
    PORT SYS_Rst = mbl1d_rst
END
BEGIN fsl_v20
    PARAMETER INSTANCE = f0_fsl
    PARAMETER HW_VER = 2.11.d
    PARAMETER C_EXT_RESET_HIGH = 1
    PARAMETER C_IMPL_STYLE = 0
    PARAMETER C_USE_CONTROL = 0
    PARAMETER C_FSL_DWIDTH = 32
    PARAMETER C_FSL_DEPTH = 1
    PARAMETER C_READ_CLOCK_PERIOD = 0
    PORT FSL_Clk = f0_fsl_clk
    PORT SYS_Rst = f0_sys_rst
    PORT FSL_Rst = f0_fsl_rst
    PORT FSL_Full = f0_full
    PORT FSL_Has_Data = f0_has_data
```

```
    PORT FSL_Control_IRQ = f0_control_irq
END
BEGIN fsl_v20
    PARAMETER INSTANCE = f1_fsl
    PARAMETER HW_VER = 2.11.d
    PARAMETER C_EXT_RESET_HIGH = 1
    PARAMETER C_IMPL_STYLE = 0
    PARAMETER C_USE_CONTROL = 0
    PARAMETER C_FSL_DWIDTH = 32
    PARAMETER C_FSL_DEPTH = 1
    PARAMETER C_READ_CLOCK_PERIOD = 0
    PORT FSL_Clk = f1_fsl_clk
    PORT SYS_Rst = f1_sys_rst
    PORT FSL_Rst = f1_fsl_rst
    PORT FSL_Full = f1_full
    PORT FSL_Has_Data = f1_has_data
    PORT FSL_Control_IRQ = f1_control_irq
END
BEGIN fsl_v20
    PARAMETER INSTANCE = f2_fsl
    PARAMETER HW_VER = 2.11.d
    PARAMETER C_EXT_RESET_HIGH = 1
    PARAMETER C_IMPL_STYLE = 0
    PARAMETER C_USE_CONTROL = 0
    PARAMETER C_FSL_DWIDTH = 32
    PARAMETER C_FSL_DEPTH = 1
    PARAMETER C_READ_CLOCK_PERIOD = 0
    PORT FSL_Clk = f2_fsl_clk
    PORT SYS_Rst = f2_sys_rst
    PORT FSL_Rst = f2_fsl_rst
    PORT FSL_Full = f2_full
    PORT FSL_Has_Data = f2_has_data
    PORT FSL_Control_IRQ = f2_control_irq
END
BEGIN bram_block
    PARAMETER INSTANCE = mbl1b_bram
    PARAMETER HW_VER = 1.00.a
    BUS_INTERFACE PORTA = mbl1b_lmba
    BUS_INTERFACE PORTB = mbl1b_lmbb
END
BEGIN lmb_bram_if_cntlr
    PARAMETER INSTANCE = mbl1b_lmba_cntlr
    PARAMETER HW_VER = 3.00.a
    PARAMETER C_BASEADDR = 0x00000000
    PARAMETER C_HIGHADDR = 0x00007fff
    BUS_INTERFACE SLMB = mbl1i_lmb
    BUS_INTERFACE BRAM_PORT = mbl1b_lmba
END
BEGIN lmb_bram_if_cntlr
    PARAMETER INSTANCE = mbl1b_lmbb_cntlr
    PARAMETER HW_VER = 3.00.a
    PARAMETER C_BASEADDR = 0x00000000
    PARAMETER C_HIGHADDR = 0x00007fff
    BUS_INTERFACE SLMB = mbl1d_lmb
    BUS_INTERFACE BRAM_PORT = mbl1b_lmbb
END
BEGIN microblaze
    PARAMETER INSTANCE = mbl1_microblaze
    PARAMETER HW_VER = 8.10.a
    PORT MB_RESET = mbl1_rst
    BUS_INTERFACE ILMB = mbl1i_lmb
    BUS_INTERFACE DLMB = mbl1d_lmb
    BUS_INTERFACE SFSL0 = f0_fsl
    BUS_INTERFACE SFSL1 = f1_fsl
    BUS_INTERFACE MFSL0 = f2_fsl
    PARAMETER C_FSL_LINKS = 2
END
```

Top-Level VHDL - "pcores/system_tld_v1_00_a/hdl/vhdl/system_tld.vhd"

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 use IEEE.NUMERIC_STD.ALL;
5
6 entity system_tld is
7 port(sys_clk : in std_logic;
8      sys_rst : in std_logic;
9      sys_rs232_rx1 : in std_logic;
10     sys_rs232_tx1 : out std_logic;
11     mbl1i_clk : out std_logic;
12     mbl1i_rst : out std_logic;
13     mbl1d_clk : out std_logic;
14     mbl1d_rst : out std_logic;
15     f0_fsl_clk : out std_logic;
16     f0_sys_rst : out std_logic;
17     f0_fsl_rst : in std_logic;
18     f0_fsl_m_clk : out std_logic;
19     f0_fsl_m_data : out std_logic_vector(31 downto 0);
20     f0_fsl_m_control : out std_logic;
21     f0_fsl_m_write : out std_logic;
22     f0_fsl_m_full : in std_logic;
23     f0_full : in std_logic;
24     f0_has_data : in std_logic;
25     f0_control_irq : in std_logic;
26     f1_fsl_clk : out std_logic;
27     f1_sys_rst : out std_logic;
28     f1_fsl_rst : in std_logic;
29     f1_fsl_m_clk : out std_logic;
30     f1_fsl_m_data : out std_logic_vector(31 downto 0);
31     f1_fsl_m_control : out std_logic;
32     f1_fsl_m_write : out std_logic;
33     f1_fsl_m_full : in std_logic;
34     f1_full : in std_logic;
35     f1_has_data : in std_logic;
36     f1_control_irq : in std_logic;
37     f2_fsl_clk : out std_logic;
38     f2_sys_rst : out std_logic;
39     f2_fsl_rst : in std_logic;
40     f2_fsl_s_clk : out std_logic;
41     f2_fsl_s_data : in std_logic_vector(31 downto 0);
42     f2_fsl_s_control : in std_logic;
43     f2_fsl_s_read : out std_logic;
44     f2_fsl_s_exists : in std_logic;
45     f2_full : in std_logic;
46     f2_has_data : in std_logic;
47     f2_control_irq : in std_logic;
48     mbl1_clk : out std_logic;
49     mbl1_rst : out std_logic);
50 end entity system_tld;
51
52 architecture str of system_tld is
53     signal f0s_ct : std_logic;
54     signal f0s_dt : std_logic_vector(31 downto 0);
55     signal f0s_ex : std_logic;
56     signal f0s_rd : std_logic;
57     signal f1s_ct : std_logic;
58     signal f1s_dt : std_logic_vector(31 downto 0);
59     signal f1s_ex : std_logic;
60     signal f1s_rd : std_logic;
61     signal f2m_ct : std_logic;
62     signal f2m_dt : std_logic_vector(31 downto 0);
63     signal f2m_fl : std_logic;
64     signal f2m_wr : std_logic;
65     signal helper125_0 : std_logic;
66     signal helper125_1 : std_logic_vector(7 downto 0);
```

```
67      signal helper125_2 : std_logic_vector(31 downto 0);
68      signal helper125_3 : std_logic;
69      signal helper125_4 : std_logic;
70      signal helper125_5 : std_logic_vector(31 downto 0);
71      signal helper125_6 : std_logic;
72      signal helper125_7 : std_logic;
73      signal helper125_8 : std_logic;
74      signal ren : std_logic;
75      signal s0 : std_logic;
76      signal s1 : std_logic;
77      signal s2 : std_logic;
78      signal s3 : std_logic;
79      signal s4 : std_logic;
80      signal s5 : std_logic;
81      signal s6 : std_logic;
82      signal s7 : std_logic;
83      signal s8 : std_logic;
84      signal tld_clks81_0 : std_logic;
85      signal tld_clks81_9 : std_logic;
86      signal uart102_1 : std_logic;
87      signal uart102_2 : std_logic;
88      signal uart102_3 : std_logic_vector(7 downto 0);
89  begin
90      mbl1i_clk<=tld_clks81_0; mbl1i_rst<=tld_clks81_9;
91      mbl1d_clk<=tld_clks81_0; mbl1d_rst<=tld_clks81_9;
92      f0_fsl_clk<=tld_clks81_0; f0_sys_rst<=tld_clks81_9;
93      f0_fsl_m_clk<=tld_clks81_0; f1_fsl_clk<=tld_clks81_0;
94      f1_sys_rst<=tld_clks81_9; f1_fsl_m_clk<=tld_clks81_0;
95      f2_fsl_clk<=tld_clks81_0; f2_sys_rst<=tld_clks81_9;
96      f2_fsl_s_clk<=tld_clks81_0; f0_fsl_m_data<=helper125_2;
97      f0_fsl_m_control<=helper125_3; f0_fsl_m_write<=helper125_4;
98      f1_fsl_m_data<=helper125_5; f1_fsl_m_control<=helper125_6;
99      f1_fsl_m_write<=helper125_7; f2_fsl_s_read<=helper125_8;
100     sys_rs232_tx1<=uart102_1; mbl1_clk<=tld_clks81_0;
101     mbl1_rst<=tld_clks81_9;
102
103     tld_clks81_inst : entity system_tld_v1_00_a.tld_clks
104     port map ( sys_clk => sys_clk, sys_rst => sys_rst,
105                clk0 => tld_clks81_0, clk90 => s0,
106                clk180 => s1, clk270 => s2,
107                clkdv => s3, clk2x => s4,
108                clk2x180 => s5, clkfx => s6,
109                clkfx180 => s7, rst => tld_clks81_9 );
110     uart102_inst : entity system_tld_v1_00_a.uart
111     port map ( clk => tld_clks81_0, rst => tld_clks81_9,
112                ten => helper125_0, trdy => s8,
113                tdat => helper125_1, tx => uart102_1,
114                ren => ren, rrdy => uart102_2,
115                rdat => uart102_3, rx => sys_rs232_rx1 );
116     helper125_inst : entity system_tld_v1_00_a.helper
117     port map ( clk => tld_clks81_0, rst => tld_clks81_9,
118                ten => helper125_0, tdat => helper125_1,
119                rrdy => uart102_2, rdat => uart102_3,
120                ifsl0_m_data => helper125_2,
121                ifsl0_m_ctl => helper125_3,
122                ifsl0_m_wr => helper125_4,
123                ifsl0_m_full => f0_fsl_m_full,
124                ifsl1_m_data => helper125_5,
125                ifsl1_m_ctl => helper125_6,
126                ifsl1_m_wr => helper125_7,
127                ifsl1_m_full => f1_fsl_m_full,
128                ifsl2_s_data => f2_fsl_s_data,
129                ifsl2_s_ctl => f2_fsl_s_control,
130                ifsl2_s_rd => helper125_8,
131                ifsl2_s_exists => f2_fsl_s_exists );
132 end architecture str;
```

G TPM Specification

Clock generator for Xilinx Devices

```
1  facet clockgen(CLKFX_MULTIPLY,CLKFX_DIVIDE::design natural;CLKDV_DIVIDE::design real;
2                sys_clk,sys_rst::input bit; clk0,clk90,clk180,clk270,clkdv,clk2x,clk2x180,
3                clkfx,clkfx180,rst::output bit) :: static is
4      irst,clkfb,clk0_dcm,clk90_dcm,clk180_dcm,clk270_dcm,clkdv_dcm,clk2x_dcm,
5      clk2x180_dcm,clkfx_dcm,clkfx180_dcm,sys_buf,locked::bit;
6  begin
7      add_library("unisim"); add_use("unisim.vcomponents.all");
8      dcm1: DCM("LOW","LOW",TRUE,FALSE,"1X","NONE","NONE","SAFE",FALSE,0,CLKFX_MULTIPLY,CLKFX_DIVIDE,
9                CLKDV_DIVIDE,10.0,"SYSTEM_SYNCHRONOUS",x"F0F0",not irst,sys_buf,clkfb,0,0,0,clk0_dcm,
10               clk90_dcm,clk180_dcm,clk270_dcm,clkdv_dcm,clk2x_dcm,clk2x180_dcm,clkfx_dcm,clkfx180_dcm,
11               _,locked,_);
12
13     bufg_rst:      BUFG(sys_rst,irst);          bufg_sysclk: BUFG(sys_clk,sys_buf);
14     bufg_clk0:     BUFG(clk0_dcm,clkfb);        bufg_clk90:  BUFG(clk90_dcm,clk90);
15     bufg_clk180:   BUFG(clk180_dcm,clk180);     bufg_clk270: BUFG(clk270_dcm,clk270);
16     bufg_clkdv:    BUFG(clkdv_dcm,clkdv);       bufg_clk2x:  BUFG(clk2x_dcm,clk2x);
17     bufg_clk2x180: BUFG(clk2x180_dcm,clk2x180); bufg_clkfx:  BUFG(clkfx_dcm,clkfx);
18     bufg_clkfx180: BUFG(clkfx180_dcm,clkfx180);
19     clk0 = clkfb; rst = not locked;
20 end facet;
```

FSL Helper for Multiply-Accumulate

```
1 use maccpkg;
2 facet helper(macc_cmd::input macc_command; macc_val::output macc_result;
3              a,b::output word(32); c::input word(32)) :: state_based is
4 begin
5     if macc_command_go(macc_cmd) then a'send(macc_command_value1(macc_cmd)) else true end if;
6     if macc_command_go(macc_cmd) then b'send(macc_command_value2(macc_cmd)) else true end if;
7     macc_val = if c'receive then MACCRES(1,c) else MACCRES(0,x"00000000") end if;
8 end facet;
```

Multiply-Accumulate Shared Package

```
1 package maccpkg :: static is
2     macc_command :: type is data
3         MACCCMD(macc_command_go::bit; macc_command_value1::word(32); macc_command_value2::word(32))
4     end data;
5
6     macc_result :: type is data
7         MACCRES(macc_result_done::bit; macc_result_value::word(32))
8     end data;
9 end package;
```

Multiply-Accumulate

```
1  facet macc(a,b::input word(32); o::output word(32)) :: state_based is
2      d1  :: bit initial is 0; acc :: word(32) initial is x"0000 0000";
3  begin
4      acc' = if a'receive and b'receive then
5              if a = x"0000 0000" and b = x"0000 0000" then x"0000 0000"
6              else acc + (a * b) end if
7             else acc end if;
8      d1' = (a'receive and b'receive);
9      if d1==1 then o'send(acc) else true end if;
10 end facet;
```

Find Maximum

```
use tpm;
facet findmax(cmd::input max_command; dout::output word(32)) :: state_based is
    mx::word(30) initial is 0; dl::bit initial is 0;
begin
    mx’ = if cmd’receive then case cmd is
              IDLE_MAX -> mx | INIT_MAX(v) -> v | UPDATE_MAX(v) -> if v > mx then v else mx end if
          end case else mx end if;
    dl’ = cmd’receive;
    if (dl==1) then dout’send(b”00” & mx) else true end if;
end facet;
```

PCR Register File Shared Package

```
package pcrpkg :: static is
    pcr_command :: type is data
      PCR_READ(pcr_read_addr::word(4)) | PCR_WRITE(pcr_write_addr::word(4);pcr_write_din::word(160))
    end data encoding is ”binary”, alignment is ”packed”;

    pcr_value :: type is word(160);
end package;
```

PCR Register File

```
use pcrpkg;
facet pcrrf(cmd::input pcr_command; dout::output pcr_value; isrst::output bit;
            rstlc,extlc::output word(8)) :: state_based is
    orst::bit; addr::word(4); orlc,oelc::word(8); outp::word(160);
    regs::array(16,word(160));
    resettable :: array( 16, bit ) is b”1000 0000 0000 0001”;
    reset_locale :: array( 16, word(8) ) is
        [x”80”,x”91”,x”B3”,x”A2”,x”E6”,x”F7”,x”D5”,x”C4”,x”4C”,x”5D”,x”7F”,x”6E”,x”2A”,x”3B”,x”19”,x”08”];
    extend_locale :: array( 16, word(8) ) is
        [x”00”,x”01”,x”02”,x”03”,x”04”,x”05”,x”06”,x”07”,x”08”,x”09”,x”0A”,x”0B”,x”0C”,x”0D”,x”0E”,x”0F”];
begin
    addr = case cmd is PCR_READ(addr) -> addr | PCR_WRITE(addr,din) -> addr | _ -> 0 end case;
    regs(addr)’ = case cmd is PCR_WRITE(addr,din) -> din | _ -> regs(addr) end case;
    outp’ = regs(addr);
    orst’ = resettable(addr);
    orlc’ = reset_locale(addr);
    oelc’ = extend_locale(addr);
    dout  = outp; isrst = orst; rstlc = orlc; extlc = oelc;
end facet;
```

SHA-1 Engine Shared Package

```
package shapkg :: static is
    sha_command :: type is data
        SHACMD(sha_command_start::bit;       sha_command_continue::bit; sha_command_finish::bit;
               sha_command_block::word(512); sha_command_size    ::word(7) )
    end data;

    sha_result :: type is data
        SHARES(sha_result_busy::bit; sha_result_valid::bit; sha_result_hash::word(160))
    end data;
end package;
```

Pseudo Random Number Generator

```
facet prng(en,sv::input word(32); rn::output word(32)) :: state_based is
    lf0 :: word(32) initial is x"B560AAEF";
    lf1 :: word(32) initial is x"F9082C6B";
    lf2 :: word(32) initial is x"3B0EAC69";
    lfb0,lfb1,lfb2,rcvd::bit; oval::word(32);
begin
    lfb0 = not (lf0(31) xor lf0(21) xor lf0(1) xor lf0(0));
    lfb1 = not (lf1(31) xor lf1(21) xor lf1(1) xor lf1(0));
    lfb2 = not (lf2(31) xor lf2(21) xor lf2(1) xor lf2(0));
    lf0' = if en'receive and (en/=x"00000000") then (lf0 sub [30,..0]) & lfb0
           elseif sv'receive then lf0 xor sv
           else lf0 end if;
    lf1' = if en'receive and (en/=x"00000000") and (lf0(0) == 1) then (lf1 sub [30,..0]) & lfb1
           elseif sv'receive then lf1 xor sv
           else lf1 end if;
    lf2' = if en'receive and (en/=x"00000000") and (lf0(0) == 0) then (lf2 sub [30,..0]) & lfb2
           elseif sv'receive then lf2 xor sv
           else lf2 end if;
    oval = if lf0(0) then lf1 else lf2 end if;
    if en'receive and (en==1) then rn'send(oval) else true end if;
end facet;
```

SHA-1 Execution Engine Black Box Declaration (Implemented in Verilog)

```
facet interface sha1_exec(clk,reset,start::input bit; data_in::input word(32); load_in::input bit;
                          cv::input word(160); use_prev_cv::input bit; busy,out_valid::output bit;
                          cv_next::output word(160)) :: state_based is
end facet interface;
```

SHA-1 Engine

```
use shapkg;
facet sha(clk,rst::input bit; cmd::input sha_command; res::output sha_result) :: state_based is
    done,bsy,run,load,dowait,vld::bit; loadval::word(32); cv,cvout::word(160);
begin
    engine: sha1_exec(clk,rst,run,loadval,load,cv,0,bsy,done,cvout);
    padder: sha1_pad(clk,rst,sha_command_start(cmd),sha_command_continue(cmd),sha_command_finish(cmd),
                     bsy,done,sha_command_block(cmd),sha_command_size(cmd),dowait,vld,run,load,loadval);
    res = SHARES(dowait or bsy, vld, cvout);
    cv' = if clk'event and clk=1 then
            if rst=1 or sha_command_start(cmd) then x"67452301EFCDAB8998BADCFE10325476C3D2E1F0"
            elseif done=1 then cvout else cv end if
          else cv end if;
end facet;
```

TPM PCR Read Command

```
use tpm, pcrpkg, comms;
facet comm_pcrread(hdr     :: input comm_header;  res     :: output comm_result;
                   pcr_cmd :: output pcr_command; pcr_val :: input pcr_value) :: state_based is
    dena1, dena2 :: bit initial is 0;
begin
    dena1' = (comm_header_go(hdr) and comm_header_ena(hdr));
    dena2' = dena1;
    pcr_cmd = PCR_READ(comm_header_idat(hdr) sub [915,..912]);
    res = COMMRES(dena2,TPM_TAG_RSP_COMMAND & x"0000001e" & x"00000000" & pcr_val & x"00000000 00000000
                                                00000000 00000000 00000000 00000000 00000000 00000000
                                                00000000 00000000 00000000 00000000 00000000 00000000
                                                00000000 00000000 00000000 00000000 00000000 00000000
                                                00000000 00000000 00000000 00000000 0000");
end facet;
```

TPM Shared Package

```
1  package tpm :: static is
2      tpm_tag :: type is data
3          TPM_TAG_RQU_COMMAND | TPM_TAG_RQU_AUTH1_COMMAND | TPM_TAG_RQU_AUTH2_COMMAND |
4          TPM_TAG_RSP_COMMAND | TPM_TAG_RSP_AUTH1_COMMAND | TPM_TAG_RSP_AUTH2_COMMAND
5      end data;
6      tpm_ord :: type is data
7          TPM_ORD_SHA1COMPLETE | TPM_ORD_SHA1COMPLETEEXTEND | TPM_ORD_SHA1START | TPM_ORD_SHA1UPDATE |
8          TPM_ORD_PCRREAD      | TPM_ORD_EXTEND             | TPM_ORD_GETRANDOM | TPM_ORD_STIRRANDOM |
9          TPM_ORD_DOMACC       | TPM_ORD_DOMAX
10     end data;
11     tpm_err :: type is data
12         TPM_SUCCESS | TPM_AUTHFAIL | TPM_BADINDEX | TPM_BAD_PARAMETER
13     end data;
14     max_command :: type is data
15         IDLE_MAX | INIT_MAX(max_command_init_val::word(30)) | UPDATE_MAX(max_command_update_val::word(30))
16     end data;
17     tpm_command :: type is data
18         TPM_CMD_SHA1START | TPM_CMD_PCRREAD(pcrread_addr :: word(32)) |
19         TPM_CMD_SHA1UPDATE(sha1update_size::word(32); sha1update_data::word(512)) |
20         TPM_CMD_EXTEND(extend_addr :: word(32); extend_digest :: word(160))
21     end data;
22     tpm_size :: type is word(32);
23 end package;
```

RS-232 UART Shared Package

```
1  package uartconst() :: static is
2      STOP  :: bit is 1; START :: bit is 0; IDLE  :: bit is 1;
3      calculateBaud(clk :: integer; rate :: integer) :: integer is clk/(rate*16);
4  end package;
```

RS-232 UART

```
1  use uartconst;
2  facet uart(CLKRATE :: design integer; BAUDRATE :: design integer;
3             ten     :: input bit;      trdy     :: output bit; tdat :: input word(8);
4             tx      :: output bit;     ren      :: input bit;  rrdy :: output bit;
5             rdat    :: output word(8); rx       :: input bit ) :: state_based is
6      baud :: bit;
7  begin
8      baud' = upcounter(calculateBaud(CLKRATE,BAUDRATE));
9      rxinst: uart_rx( baud, ren, rrdy, rdat, rx );
10     txinst: uart_tx( baud, ten, trdy, tdat, tx );
11 end facet;
```

RS-232 UART Transmitter

```
1  use uartconst;
2  facet uart_rx(baud :: input bit;      ren :: input bit; rrdy :: output bit;
3                dout :: output word(8); rx  :: input bit) :: state_based is
4      dosht,dshf,busy,rcvd::bit initial is 0;
5      cntr  :: word(8) initial is b"00000000";
6      shreg :: word(10) initial is b"1111111111";
7  begin
8      busy'  = if (rx=START and busy=0) then 1 elseif (cntr=159) then 0 else busy end if;
9      cntr'  = named_upcounter(160, (busy or cntr=159) and baud);
10     recvd' = upcounter(10, dshf); shreg' = lshiftreg(rx, dosht);
11     dosht  = %((cntr sub [3,..0] == b"0111") and baud);
12     rrdy   = %(ren and recvd and (shreg(9) = START) and (shreg(0) = STOP));
13     dshf'  = dosht; dout   = shreg sub [1,..8];
14 end facet;
```

RS-232 UART Receiver

```
1  use uartconst;
2  facet uart_tx(baud :: input bit;     ena :: input bit; rdy :: output bit;
3                din  :: input word(8); tx  :: output bit) :: state_based is
4      bitnm :: word(4) initial is b"0000";
5      shreg :: word(10) initial is b"1111111111";
6      busy,next::bit;
7  begin
8      next'  = upcounter(16, baud and (ena or busy));
9      bitnm' = named_upcounter(11, ena or (baud and next and busy));
10     shreg' = rshiftreg(IDLE, next and baud, busy=0 and ena, STOP & din & START);
11     busy   = %(bitnm /= b"0000");
12     tx     = shreg(0);
13     rdy    = not busy;
14 end facet;
```

TPM Communication Shared Package

```
1  use tpm;
2  package comms :: static is
3      comm_header :: type is data
4          COMMHDR( comm_header_go   :: bit;     comm_header_ena  :: bit; comm_header_tag  :: tpm_tag;
5                   comm_header_size :: tpm_size; comm_header_idat :: word(1024) )
6      end data encoding is "binary", alignment is "packed";
7
8      comm_result :: type is data
9          COMMRES( comm_result_done :: bit; comm_result_odat :: word(1024) )
10     end data encoding is "binary", alignment is "packed";
11 end package;
```

TPM Get Random Number Command

```
1  use tpm, comms;
2  facet comm_getrandom( hdr :: input comm_header; res :: output comm_result;
3                        en  :: output bit;        val :: input word(32) ) :: state_based is
4      rsize::word(32); dne::bit;
5  begin
6      rsize = comm_header_idat(hdr) sub [943,..912];
7      en = (comm_header_go(hdr) and comm_header_ena(hdr));
8      dne' = (comm_header_go(hdr) and comm_header_ena(hdr));
9      res = COMMRES(dne, TPM_TAG_RSP_COMMAND & x"0000 000E 0000 0000 0000 0004" & val & x"0000 0000 0000 0000
10                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
11                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
12                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000");
13 end facet;
```

TPM Stir Random Number Generator Command

```
1  use tpm, comms;
2  facet comm_stirrandom(hdr::input comm_header;res::output comm_result; st::output bit;
3                        sv ::output word(32);  val::input word(32)) :: state_based is
4      rsize :: word(32);
5  begin
6      rsize = comm_header_idat(hdr) sub [943,..912];
7      st = (comm_header_go(hdr) and comm_header_ena(hdr));
8      sv = comm_header_idat(hdr) sub [911,..880];
9      res = COMMRES(comm_header_go(hdr), TPM_TAG_RSP_COMMAND & x"0000 000a 0000 0000 0000 0000 0000 0000 0000
10                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
11                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
12                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000");
13 end facet;
```

TPM Find Maximum Auxiliary Command

```
use tpm, comms;
facet comm_fmax(hdr :: input comm_header; res  :: output comm_result; cmd :: output max_command;
                go  :: output bit;        done :: input bit;          val :: input word(32))
                    :: state_based is
    ddne :: bit initial is 0;
begin
    cmd = case hdr is
                 COMMHDR(go,ena,tg,sz,idat) -> if go and ena then
                     if (idat sub [943,..912]) = x"00000000" then INIT_MAX(idat sub [911,..882])
                     else UPDATE_MAX(idat sub [911,..882]) end if
                   else IDLE_MAX end if
               | _ -> IDLE_MAX
          end case;

    go = case hdr is COMMHDR(g,e,_,_,_) -> g and e | _ -> 0 end case;
    ddne' = done;

    res = COMMRES(ddne, TPM_TAG_RSP_COMMAND & x"0000 000e 0000 0000" & val & x"0000 0000 0000 0000 0000
                                              0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
                                              0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
                                              0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
                                              0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
                                              0000 0000 0000 0000 0000 0000 0000 0000");
end facet;
```

TPM SHA-1 Start Command

```
use tpm, comms, shapkg;
facet comm_sha1start(hdr :: input comm_header;  res :: output comm_result;
                     cmd :: output sha_command; val :: input sha_result) :: state_based is
begin
    cmd = SHACMD(comm_header_go(hdr) and comm_header_ena(hdr), 0, 0, x"0000 0000 0000 0000 0000 0000
                0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
                0000 0000 0000 0000 0000 0000 0000 0000 0000", b"0000000");

    res = COMMRES(comm_header_go(hdr),TPM_TAG_RSP_COMMAND & x"0000000a" & x"00000000" & x"0000 0000
                 0000 0000 0000 0000 0000 0000 0000 0000" & x"00000000 00000000 00000000 00000000
                 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
                 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
                 00000000 00000000 0000");
end facet;
```

TPM SHA-1 Continue Command

```
use tpm, comms, shapkg;
facet comm_sha1contn(hdr :: input comm_header;  res :: output comm_result; cmd :: output sha_command;
                     val :: input sha_result) :: state_based is
    dvalid, rvalid :: bit initial is 0;
begin
    cmd = SHACMD( 0, comm_header_go(hdr) and comm_header_ena(hdr), 0,
                 comm_header_idat(hdr) sub [911,..400], comm_header_idat(hdr) sub [918,..912] );

    dvalid' = sha_result_valid(val);
    rvalid = ((sha_result_valid(val) xor dvalid) and sha_result_valid(val));

    res = COMMRES(rvalid, TPM_TAG_RSP_COMMAND & x"0000000a" & x"00000000" & x"0000 0000 0000 0000 0000
                 0000 0000 0000 0000 0000" & x"00000000 00000000 00000000 00000000 00000000 00000000
                 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
                 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000");
end facet;
```

TPM Multiply-Accumulate Auxiliary Command

```
1 use tpm, comms, maccpkg;
2 facet comm_macc(hdr      :: input comm_header;   res      :: output comm_result;
3                 macc_cmd :: output macc_command; macc_val :: input macc_result) :: state_based is
4 begin
5     macc_cmd = MACCCMD( comm_header_go(hdr) and comm_header_ena(hdr), comm_header_idat(hdr) sub [943,..912],
6                         comm_header_idat(hdr) sub [911,..880] );
7     res = case macc_val is
8               MACCRES(dne,val) -> COMMRES(dne,TPM_TAG_RSP_COMMAND & x"0000 000e 0000 0000" & val & x"0000
9                      0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
10                     0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
11                     0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
12                     0000 0000 0000 0000 0000")
13            | _ -> COMMRES(0,TPM_TAG_RSP_COMMAND & x"0000 000e 0000 0000" & x"0000 0000 0000 0000 0000
14                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
15                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
16                   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
17                   0000 0000 0000")
18          end case;
19 end facet;
```

TPM Command Receiver

```
1 use tpm;
2 facet comm_recv(ren  :: output bit; rrdy :: input bit;      rdat :: input word(8);
3                 rcvd :: output bit; tag  :: output tpm_tag; size  :: output tpm_size;
4                 ord  :: output tpm_ord; ocmd  :: output word(1024)) :: state_based is
5     cmd,rcmd::word(1024); cdone,ddone,fdone::bit initial is 0;
6 begin
7     ren   = 1; ocmd   = rcmd;       ddone' = cdone; fdone' = ddone; rcvd = fdone;
8     tag   = rcmd sub [1023,..1008]; size   = rcmd sub [1007,..976]; ord  = rcmd sub [975,..944];
9     cmd'  = lshiftreg(rdat, rrdy);  cdone' = upcounter(128, rrdy);
10    rcmd' = if ddone then cmd else rcmd end if;
11 end facet;
```

TPM PCR Extend Command

```
1 use tpm, pcrpkg, shapkg, comms;
2 facet comm_extend(hdr    ::input comm_header; res    ::output comm_result; pcr_cmd::output pcr_command;
3                   pcr_val::input pcr_value;   sha_cmd::output sha_command; sha_vl ::input sha_result)
4                          :: state_based is
5     stype :: type is data IDLE | RDRF | SHA1 end data;
6     state :: stype initial is IDLE;
7 begin
8     state' = case state is
9                   IDLE -> if comm_header_go(hdr) and comm_header_ena(hdr) then RDRF else IDLE end if
10                | RDRF -> SHA1
11                | SHA1 -> if sha_result_valid(sha_vl) then IDLE else SHA1 end if
12             end case;
13    pcr_cmd = if isSHA1(state) and sha_result_valid(sha_vl)
14              then PCR_WRITE(comm_header_idat(hdr) sub [915,..912],sha_result_hash(sha_vl))
15              else PCR_READ(0) end if;
16    sha_cmd = SHACMD( comm_header_go(hdr), 0, isRDRF(state), pcr_val &
17                      (comm_header_idat(hdr) sub [911,..752]) &
18                      x"00000000000000000000000000000000000000000000000", b"0101000" );
19    res = COMMRES(isSHA1(state) and sha_result_valid(sha_vl),TPM_TAG_RSP_COMMAND & x"0000001e" &
20          x"00000000" & sha_result_hash(sha_vl) & x"00000000 00000000 00000000 00000000 00000000
21            00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
22            00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
23            00000000 0000");
24 end facet;
```

TPM Response Transmitter

```
1  use tpm;
2  facet comm_tran(ten  :: output bit; trdy :: input bit; tdat :: output word(8);
3                  send :: input bit;  res  :: input word(1024)) :: state_based is
4      tcnt::word(8) initial is 0; tshft::word(1024) initial is 0; dtr,drdy,rdy::bit initial is 0;
5  begin
6      tcnt'  = named_upcounter(129, send or rdy);
7      tshft' = lshiftreg(x"00", rdy, send, res);
8      drdy'  = trdy;
9      rdy    = %((drdy xor trdy) and trdy = 1);
10     dtr'   = %(send or (tcnt/=x"00" and rdy));
11     ten    = dtr;
12     tdat   = tshft sub [1023,..1016];
13 end facet;
```

TPM SHA-1 Finish Command

```
1  use tpm, comms, pcrpkg, shapkg;
2  facet comm_sha1finsh(hdr :: input comm_header;  res :: output comm_result; ext :: input bit;
3                       cmd :: output sha_command; val :: input sha_result;   pcm :: output pcr_command;
4                       pvl :: input pcr_value) :: state_based is
5      stype :: type is data IDLE | RDRF | SHA1 end data;
6      state :: stype initial is IDLE;
7
8      shaen::bit; shasz::word(7); shadt::word(512); rsz::word(32); svl,rsh,rpc::word(160);
9      dvalid, evalid, rvalid :: bit initial is 0;
10 begin
11     cmd = SHACMD(evalid, 0, shaen, shadt, shasz );
12     shaen = if isIDLE(state) then comm_header_go(hdr) and comm_header_ena(hdr) else isRDRF(state) end if;
13
14     shadt = if isIDLE(state) then
15               if ext then comm_header_idat(hdr) sub [879,..368]
16               else comm_header_idat(hdr) sub [911,..400] end if
17             else pvl & svl & x"000000000000000000000000000000000000000000000000" end if;
18
19     shasz = if isIDLE(state) then
20               if ext then comm_header_idat(hdr) sub [886,..880]
21               else comm_header_idat(hdr) sub [918,..912] end if
22             else b"0101000" end if;
23
24     svl' = if isIDLE(state) and ext and evalid then sha_result_hash(val) else svl end if;
25
26     state' = case state is
27                  IDLE -> if comm_header_ena(hdr) and evalid and ext then RDRF else IDLE end if
28                | RDRF -> SHA1
29                | SHA1 -> if sha_result_valid(val) then IDLE else SHA1 end if
30              end case;
31
32     pcm = if isSHA1(state) and sha_result_valid(val)
33           then PCR_WRITE(comm_header_idat(hdr) sub [915,..912],sha_result_hash(val))
34           else PCR_READ(0) end if;
35
36     dvalid' = sha_result_valid(val);
37     evalid  = (comm_header_ena(hdr) and ((sha_result_valid(val) xor dvalid) and sha_result_valid(val)));
38
39     rvalid = if ext then isSHA1(state) and sha_result_valid(val) else evalid end if;
40     rsz = if ext then x"00000032" else x"0000001e" end if;
41     rsh = if ext then svl else sha_result_hash(val) end if;
42     rpc = if ext then sha_result_hash(val) else x"00000000 00000000 00000000 00000000 00000000" end if;
43
44     res = COMMRES(rvalid, TPM_TAG_RSP_COMMAND & rsz & x"00000000" & rsh & rpc &
45                  x"00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
46                    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
47                    00000000 0000");
48 end facet;
```

TPM Communication

```
1  use tpm, pcrpkg, maccpkg, shapkg, comms;
2  facet comm(macc_cmd :: output macc_command; macc_val :: input macc_result;  fmax_cmd :: output max_command;
3             fmax_val :: input word(32);      sha_cmd  :: output sha_command; sha_val  :: input sha_result;
4             pcr_cmd  :: output pcr_command;  pcr_val  :: input pcr_value;    pcr_en   :: output bit;
5             rng_en :: output word(32); rng_sv :: output word(32); rng_val :: input word(32);
6             ten  :: output bit; trdy :: input bit; tdat :: output word(8); ren  :: output bit;
7             rrdy :: input bit;  rdat :: input word(8)) :: state_based is
8    UNKNOWN::word(1024) is TPM_TAG_RSP_COMMAND & x"0000 000a 0000 000a 0000 0000 0000 0000 0000 0000 0000
9                0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
10               0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
11               0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000";
12   tag::tpm_tag; size::tpm_size; ord::tpm_ord; rnsv,rnvl,fmvl::word(32); cmd,tres::word(1024);
13   rcvd,tsend,rnen,rnst,fmdn,fmgo::bit; fmcm::max_command;
14   result,macc_res,fmax_res,extend_res,sha1st_res,sha1ct_res,
15   sha1fn_res,pcrread_res,getrn_res,strrn_res::comm_result;
16   pcrread_pcr_cmd,extend_pcr_cmd,sha1fn_pcr_cmd::pcr_command;
17   extend_sha_cmd,sha1st_sha_cmd,sha1ct_sha_cmd,sha1fn_sha_cmd::sha_command;
18 begin
19   recvr: comm_recv(ren,rrdy,rdat,rcvd,tag,size,ord,cmd);
20   tsmtr: comm_tran(ten,trdy,tdat,tsend,tres);
21   pcrrd: comm_pcrread(COMMHDR(rcvd,ord=TPM_ORD_PCRREAD,tag,size,cmd),pcrread_res,pcrread_pcr_cmd,pcr_val);
22   getrn: comm_getrandom(COMMHDR(rcvd,ord=TPM_ORD_GETRANDOM,tag,size,cmd),getrn_res,rnen,rnvl);
23   strrn: comm_stirrandom(COMMHDR(rcvd,ord=TPM_ORD_STIRRANDOM,tag,size,cmd),strrn_res,rnst,rnsv,rnvl);
24   maccc: comm_macc(COMMHDR(rcvd,ord=TPM_ORD_DOMACC,tag,size,cmd),macc_res,macc_cmd,macc_val);
25   fmaxx: comm_fmax(COMMHDR(rcvd,ord=TPM_ORD_DOMAX,tag,size,cmd),fmax_res,fmcm,fmgo,fmdn,fmvl);
26   ext: comm_extend(COMMHDR(rcvd,ord=TPM_ORD_EXTEND,tag,size,cmd),extend_res,
27                    extend_pcr_cmd,pcr_val,extend_sha_cmd,sha_val);
28   sha1st: comm_sha1start(COMMHDR(rcvd,ord=TPM_ORD_SHA1START,tag,size,cmd),
29                          sha1st_res,sha1st_sha_cmd,sha_val);
30   sha1ct: comm_sha1contn(COMMHDR(rcvd,ord=TPM_ORD_SHA1UPDATE,tag,size,cmd),
31                          sha1ct_res,sha1ct_sha_cmd,sha_val );
32   sha1fn: comm_sha1finsh(COMMHDR(rcvd,ord=TPM_ORD_SHA1COMPLETE,tag,size,cmd),sha1fn_res,
33                          ord=TPM_ORD_SHA1COMPLETEEXTEND,sha1fn_sha_cmd,sha_val,sha1fn_pcr_cmd,pcr_val);
34
35   fmdn  = fmax_val'receive;
36   fmvl' = if fmax_val'receive then fmax_val else fmvl end if;
37   if fmgo then fmax_cmd'send(fmcm) else true end if;
38   rnvl' = if rng_val'receive then rng_val else rnvl end if;
39   tsend = comm_result_done(result);
40   tres = comm_result_odat(result);
41
42   if (rcvd==1) and (ord==TPM_ORD_STIRRANDOM) then rng_sv'send(rnsv) else true end if;
43   if (rcvd==1) and (ord==TPM_ORD_GETRANDOM) then rng_en'send(b"00000000 00000000 00000000 0000000"&rnen)
44   else true end if;
45
46   result = if ord=TPM_ORD_PCRREAD then pcrread_res elseif ord=TPM_ORD_EXTEND then extend_res
47            elseif ord=TPM_ORD_SHA1START then sha1st_res elseif ord=TPM_ORD_SHA1UPDATE then sha1ct_res
48            elseif ord=TPM_ORD_GETRANDOM then getrn_res elseif ord=TPM_ORD_STIRRANDOM then strrn_res
49            elseif ord=TPM_ORD_DOMACC then macc_res elseif ord=TPM_ORD_DOMAX then fmax_res
50            elseif ord=TPM_ORD_SHA1COMPLETE then sha1fn_res
51            elseif ord=TPM_ORD_SHA1COMPLETEEXTEND then sha1fn_res
52            else COMMRES(rcvd,UNKNOWN) end if;
53
54   pcr_cmd = if ord=TPM_ORD_PCRREAD then pcrread_pcr_cmd elseif ord=TPM_ORD_EXTEND then extend_pcr_cmd
55             elseif ord=TPM_ORD_SHA1COMPLETE then sha1fn_pcr_cmd
56             elseif ord=TPM_ORD_SHA1COMPLETEEXTEND then sha1fn_pcr_cmd
57             else PCR_READ(0) end if;
58   pcr_en  = if ord=TPM_ORD_PCRREAD or ord=TPM_ORD_EXTEND then 1 else 0 end if;
59
60   sha_cmd = if ord=TPM_ORD_EXTEND then extend_sha_cmd elseif ord=TPM_ORD_SHA1START then sha1st_sha_cmd
61             elseif ord=TPM_ORD_SHA1UPDATE then sha1ct_sha_cmd
62             elseif ord=TPM_ORD_SHA1COMPLETE then sha1fn_sha_cmd
63             else SHACMD(0,0,0,x"0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
64                       0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
65                       0000 0000",b"0000000") end if;
66 end facet;
```

SHA-1 Padding

```
1 facet sha1_pad(clk,rst,start,continue,finish,busy,done::input bit; blck::input word(512); size::input word(7);
2               dowait,valid,run,load::output bit; loadval::output word(32)) :: state_based is
3     last,padding,rpadding,ipadding,dpadding,firstpad,extra,dbusy,cansend,rdone::bit;
4     rawbytes,padbytes,rembytes::word(7); outshift :: word(512);
5     ldshift,rnshift::word(16); andmask,ormask,outraw::word(32); bitsize,length::word(64);
6 begin
7     load = ldshift(15); run = rnshift(15); dowait = extra; valid = rdone;
8     rdone' = if clk'event and clk=1 then
9                if rst=1 or start=1 or continue=1 or finish=1 then 0
10               elseif done=1 and extra=0 then 1 else rdone end if
11             else rdone end if;
12    loadval = if ldshift(15) = 1 then (outraw and andmask) or ormask else 0 end if;
13    bitsize = b"0000000000000000000000000000000000000000000000000000000" & size & b"000";
14
15    length' = if clk'event and clk=1 then
16                if rst=1 or start=1 then 0
17                elseif continue=1 or finish=1 then length + bitsize
18                else length end if
19              else length end if;
20    outshift' = if clk'event and clk=1 then
21                  if continue=1 or finish=1 then blck
22                  else (outshift sub [479,..0]) & x"0000 0000" end if
23                else outshift end if;
24    outraw = outshift sub [511,..480];
25
26    cansend = %((busy xor dbusy) and busy=0);
27    dbusy' = if clk'event and clk=1 then busy else dbusy end if;
28
29    ldshift = if clk'event and clk=1 then
30                if continue=1 or finish=1 or (extra=1 and cansend=1) then b"1111 1111 1111 1111"
31                else (ldshift sub [14,..0]) & b"0" end if
32              else ldshift end if;
33    rnshift = if clk'event and clk=1 then
34                if continue=1 or finish=1 or (extra=1 and cansend=1) then b"0000 0000 0000 0001"
35                else (rnshift sub [14,..0]) & b"0" end if
36              else rnshift end if;
37
38    rembytes = if clk'event and clk=1 then
39                 if rst=1 or start=1 then 0
40                 elseif continue=1 or finish=1 then size
41                 elseif rembytes > 3 then rembytes - 4
42                 else 0 end if
43               else rembytes end if;
44    last' = if clk'event and clk=1 then
45              if start=1 then 0
46              elseif finish=1 then 1
47              else last end if
48            else last end if;
49
50    ipadding = %((ldshift(15)=1 and last=1 and rembytes<4) or (extra=1 and dbusy=1 and rembytes/=0));
51    rpadding' = if clk'event and clk=1 then
52                  if start=1 or continue=1 or finish=1 then 0
53                  elseif ipadding=1 then 1
54                  else rpadding end if
55                else rpadding end if;
56    padding = %(ipadding or rpadding);
57
58    firstpad = %((padding xor dpadding) and padding=1);
59    dpadding' = if clk'event and clk=1 then padding else dpadding end if;
60    extra' = if clk'event and clk=1 then
61               if rst=1 or start=1 or (extra=1 and cansend=1) then 0
62               elseif finish=1 and (size >= 56) then 1
63               else extra end if
64             else extra end if;
65
66    rawbytes = if finish=1 then b"1000000"-(b"0"&(size sub [5,..0]))
```

```
67                else b"1000000"-(b"0"&(length sub [8,..3])) end if;
68     padbytes = if clk'event and clk=1 then
69                  if padding=1 and ldshift(15)=1 then padbytes - (4-rembytes)
70                  elseif rawbytes<9 and padding=0 then rawbytes + b"1000000"
71                  elseif padding=0 then rawbytes
72                  else padbytes end if
73                else padbytes end if;
74
75     ormask  = if firstpad then
76                 if rembytes=0 then b"1000 0000 0000 0000 0000 0000 0000 0000"
77                 elseif rembytes=1 then b"0000 0000 1000 0000 0000 0000 0000 0000"
78                 elseif rembytes=2 then b"0000 0000 0000 0000 1000 0000 0000 0000"
79                 elseif rembytes=3 then b"0000 0000 0000 0000 0000 0000 1000 0000"
80                 else b"0000 0000 0000 0000 0000 0000 0000 0000" end if
81               elseif padbytes=8 then length sub [63,..32]
82               elseif padbytes=4 then length sub [31,..0]
83               else b"0000 0000 0000 0000 0000 0000 0000 0000" end if;
84     andmask = if rembytes=0 then b"0000 0000 0000 0000 0000 0000 0000 0000"
85               elseif rembytes=1 then b"1111 1111 0000 0000 0000 0000 0000 0000"
86               elseif rembytes=2 then b"1111 1111 1111 1111 0000 0000 0000 0000"
87               elseif rembytes=3 then b"1111 1111 1111 1111 1111 1111 0000 0000"
88               else b"1111 1111 1111 1111 1111 1111 1111 1111" end if;
89 end facet;
```

TPM Top-Level Design

```
1  CLKRATE :: integer is 100000000;
2  BDRATE :: integer is 115200;
3
4  tag2bits(tag :: tpm_tag) :: word(16) is
5      case tag is
6            TPM_TAG_RQU_COMMAND       -> x"00C1" | TPM_TAG_RQU_AUTH1_COMMAND -> x"00C2"
7          | TPM_TAG_RQU_AUTH2_COMMAND -> x"00C3" | TPM_TAG_RSP_COMMAND       -> x"00C4"
8          | TPM_TAG_RSP_AUTH1_COMMAND -> x"00C5" | TPM_TAG_RSP_AUTH2_COMMAND -> x"00C6"
9      end case;
10
11 ord2bits(ord :: tpm_ord) :: word(32) is
12     case ord is
13           TPM_ORD_SHA1COMPLETE -> x"000000A2" | TPM_ORD_SHA1COMPLETEEXTEND -> x"000000A3"
14         | TPM_ORD_SHA1START    -> x"000000A0" | TPM_ORD_SHA1UPDATE         -> x"000000A1"
15         | TPM_ORD_PCRREAD      -> x"00000015" | TPM_ORD_EXTEND             -> x"00000014"
16         | TPM_ORD_GETRANDOM    -> x"00000046" | TPM_ORD_STIRRANDOM         -> x"00000047"
17         | TPM_ORD_DOMACC       -> x"000000FF" | TPM_ORD_DOMAX              -> x"000000FE"
18     end case;
19
20 max2bits(mx :: max_command) :: (word(4),word(32),word(32),bit) is
21     case mx is
22           IDLE_MAX      -> (x"1",x"00000000",x"00000001",1)
23         | INIT_MAX(a)   -> (x"2",a,x"AAAAAAAA",0)
24         | UPDATE_MAX(a) -> (x"3",x"55555555",a,1)
25     end case;
26
27 use tpm; use pcrpkg; use shapkg; use maccpkg;
28 facet tld :: system is
29     sys_clk,sys_rst,clk,mclk,rst,rx,tx,ren,ten,rrdy,trdy::bit;
30     macc_a,macc_b,macc_acc,mout,prng_en,prng_sv,prng_val::word(32);
31     rdat,tdat::word(8);
32     macc_cmd::macc_command; macc_val::macc_result;
33     pcr_cmd::pcr_command; pcr_val::pcr_value; pcr_en::bit;
34     sha_cmd::sha_command; sha_val::sha_result;
35     mcmd :: max_command;
36 begin
37     fpga: genesys( sys_clk, sys_rst, rx, tx );
38     mbl1: mblaze( mclk, rst );
39     mbl1'target = fpga;
40
```

```
41    mbl1_bram: bram_memory( 8192 );
42    mbl1_imem: lmb_v10(mclk,rst);
43    mbl1_dmem: lmb_v10(mclk,rst);
44    mbl1_imem.master = [mbl1.ilmb];
45    mbl1_dmem.master = [mbl1.dlmb];
46    mbl1_imem.slave = [mmport(mbl1_bram.lmba,x"00000000",x"00007FFF")];
47    mbl1_dmem.slave = [mmport(mbl1_bram.lmbb,x"00000000",x"00007FFF")];
48    mdm1: mdm(rst); mdm1.master = [mbl1.debug];
49
50    communications_link("fsl", macc_a, clk, rst);   communications_link("fsl", macc_b, clk, rst);
51    communications_link("fsl", macc_acc, clk, rst); communications_link("fsl", prng_en, clk, rst);
52    communications_link("fsl", prng_sv, clk, rst);  communications_link("fsl", prng_val, clk, rst);
53
54    clks: clockgen(3,2,2.0,sys_clk,sys_rst,clk,_,_,_,_,_,_,_,_,rst);  clks'target = fpga;
55    mclks: clockgen(3,2,2.0,sys_clk,sys_rst,mclk,_,_,_, _,_,_,_,_,_); mclks'target = fpga;
56    clk'sigis      = "CLK";   mclk'sigis      = "CLK";   rst'sigis = "RST";
57    clk'clk_inport = sys_clk; mclk'clk_inport = sys_clk;
58
59    iuart:   uart( CLKRATE, BDRATE, ten, trdy, tdat, tx, ren, rrdy, rdat, rx ); iuart'target = fpga;
60    active_high_reset(iuart, rst);
61    rising_edge_clock(iuart, clk);
62
63    pcrs: pcrrf( pcr_cmd, pcr_val, _, _, _ ); pcrs'target = fpga;
64    refine_type(pcrs, pcr_command, "binary", "packed");
65    active_high_enable(pcrs, pcr_en);
66    rising_edge_clock(pcrs, clk);
67
68    sha1:  sha( clk, rst, sha_cmd, sha_val ); sha1'target = fpga;
69    refine_type(sha1, sha_command, "binary", "packed");
70    refine_type(sha1, sha_result, "binary", "packed");
71
72    rng: prng(prng_en,prng_sv,prng_val); rng'target = fpga;
73    active_high_reset(rng, rst);
74    rising_edge_clock(rng, clk);
75
76    icom:  comm(macc_cmd,macc_val,mcmd,mout,sha_cmd,sha_val,pcr_cmd,pcr_val,pcr_en,prng_en,prng_sv,
77                prng_val,ten,trdy,tdat,ren,rrdy,rdat); icom'target = fpga;
78    refine_type(icom, max_command, "binary", "packed");
79    refine_type(icom, tpm_tag, tag2bits);
80    refine_type(icom, tpm_ord, ord2bits);
81    refine_type(icom, pcr_command, "binary", "packed");
82    refine_type(icom, macc_command, "binary", "packed");
83    refine_type(icom, macc_result, "binary", "packed");
84    refine_type(icom, sha_command, "binary", "packed");
85    refine_type(icom, sha_result, "binary", "packed");
86    refine_type(pcr_cmd, pcr_command, "binary", "packed");
87    refine_type(macc_cmd, macc_command, "binary", "packed");
88    refine_type(macc_val, macc_result, "binary", "packed");
89    refine_type(sha_cmd, sha_command, "binary", "packed");
90    refine_type(sha_val, sha_result, "binary", "packed");
91    active_high_reset(icom, rst);
92    rising_edge_clock(icom, clk);
93
94    hlpr: helper( macc_cmd, macc_val, macc_a, macc_b, macc_acc ); hlpr'target = fpga;
95    refine_type(hlpr, macc_command, "binary", "packed");
96    refine_type(hlpr, macc_result, "binary", "packed");
97    active_high_reset(hlpr, rst);
98    rising_edge_clock(hlpr, clk);
99
100   mcc: macc(macc_a, macc_b, macc_acc); mcc'target = mbl1;
101   fmax: findmax( mcmd, mout ); fmax'target = fpga;
102   refine_type(fmax, max_command, "binary", "packed");
103   refine_type(mcmd, max_command, "binary", "packed");
104   active_high_reset(fmax, rst);
105   rising_edge_clock(fmax, clk);
106   communications_link("fsl", mcmd, clk, rst); communications_link("fsl", mout, clk, rst);
107 end facet;
```