A COMBINED LANGUAGE FOR HARDWARE AND SOFTWARE DESIGN

An Undergraduate Research Scholars Thesis

by

MICHAEL BASS

Submitted to Honors and Undergraduate Research
Texas A&M University
In partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Dr. Sunil P. Khatri

May 2015

Major: Electrical Engineering
Computer Science

TABLE OF CONTENTS

ABSTRACT

A Combined Language for Hardware and Software Design. (May 2015)

Michael Bass
Department of Electrical Engineering
Texas A&M University

Research Advisor: Dr. Sunil P. Khatri
Department of Electrical Engineering

Due to the nature of hardware and software, their respective design languages have evolved in isolation. Sophisticated languages and design environments exist for both hardware and software; however they remain distinct and independent, both syntactically and semantically. Hardware inherently operates in parallel and therefore hardware languages have always contained pragmas to handle parallelism, albeit explicitly. Software originally was envisioned with a serial execution paradigm. For several decades, there have been attempts to develop software tool-chains that allow parallel software execution. Significant research has been done on parallel software programming by extracting parallelism implicitly (automatically). This has generally yielded poor results, and programming paradigms that are very difficult to reason about and use in practice. In this thesis we will attempt to create a single language that can design either hardware or software. Our goal is to strive for a high level of efficiency as well as adoptability. This language provides a common syntax as well as a common semantic for designing both hardware and software. In this language, parallelism is explicitly expressed for both hardware and software. To achieve our goals we have designed a hardware translator and software translator that take the new language and translate it into Verilog [1] for hardware, and C++ [2] for software. We have tested the language against current hardware and software platforms with an array of algorithms and data sets. The result of this work could have a dramatic

impact in the digital design industry, and ultimately change the way digital design is done, allowing a merging of software and hardware design representations. This could be significant, because each of these design representations currently have billions of dollars invested in them, and are not mergeable.

ACKNOWLEDGEMENTS

I would like to acknowledge Monther Abusultan for creating the Verilog code for the Fast Fourier Transform and Matrix Multiplication algorithms. In addition I would like to acknowledge Monther for his continual assistance.

NOMENCLATURE

BCL	Bluespec Co-design Language	
CPU	Central Processing Unit	
FFT	Fast Fourier Transform	
FPGA	Field Programmable Gate Array	
HDL	Hardware Description LanguageLUT	Look Up Table
RAM	Random Access Memory	
RC++	Resulting C++	
RV	Resulting Verilog	

CHAPTER I

INTRODUCTION

Hardware and software both share a common goal to implement a digital design. In the most general sense, any digital design can be realized through either software or hardware. Technologically, hardware designs are generally more efficient for certain classes of designs, and software designs are more efficient for others. In practice, one or the other is chosen based on technology as well as economic factors. Some digital designs can be designed in either hardware or software, while some hardware designs are initially prototyped by means of software. Because of their differences, hardware and software design practices, principles, and design tools have evolved separately. Hardware developers use *hardware description languages (HDL's)* to model hardware logic, components, and modules. The most commonly used HDL's are Verilog [1] and VHDL [3]. Software developers use *programming languages* to describe a set of instructions to be executed on a processor. Commonly used software languages include C [4], C++ [2], C# [5], and Java [6]. These are only a few of the many programming languages that exist; the volume and diversity of programming languages far exceeds that of HDL's. It is our goal to unify hardware and software development into a single framework, with a single language which incorporates the semantics of both hardware and software.

Hardware is designed so that multiple components and modules can operate in parallel, while others operate serially. HDL's provide constructs and semantics to model hardware in a manner that this can be accomplished. HDL's typically use three semantic models: *structural (gate-level), dataflow (continuous), and behavioral (procedural).* Structural semantics allow a designer

to create an explicit gate-level description of their design. Dataflow semantics create a hardware description which is then transformed into physical logic elements by a *logic synthesis* tool. Behavioral semantics provide the highest level of abstraction, allowing the design to be expressed as a procedure, without any implied structure. These three semantic models provide the designer with a variety of design options, from explicit gate-level control to an abstract high-level design.

Software design has centered on *functions (procedures), object oriented design,* and more recently, *multithreading*. Originally software was created to run on a single processor with statements being evaluated sequentially. As software developed, object oriented languages became popular, providing semantics to further abstract software to a higher level. With the advancement of multicore processors, software began to support parallel programming using multithreading in languages such as C++, C#, and Java. Multithreading is a programming practice where a single *process* invokes several instruction *threads* that are capable of being executed on multiple processor cores. In this form of parallelism, multiple threads can access the same data in memory, but on the downside, it requires large amounts of synchronization, operations which make the code harder to write. In software, parallelism is expressed both implicitly as well as explicitly, depending on the design philosophy of the software platform.

Although hardware and software initially diverged in their design and practices, they have begun to slowly converge with advancements in HDL's, higher levels of abstractions in design languages for both hardware and software, improved parallel programming capabilities, and also event driven programming. Despite this slow convergence, several key issues remain, making

the convergence of hardware and software design platforms non-trivial. The concepts and tools developed in this thesis attempt to address these issues in a practical manner. It is our belief that the time is ripe for hardware and software design to be folded into a single uniform language that supports semantics for the design of both types of systems, without limiting either in practice.

Due to their separate evolution, the set of current hardware capabilities and the set of current software capabilities are not equivalent. In the uniform language, if the set of capabilities of either hardware or software were reduced, this would potentially limit the effectiveness and usability of the language. These side effects are neither desired nor acceptable. Instead, our philosophy is to implement the union of the two sets of capabilities. By taking this approach, both hardware and software will be enhanced instead of hindered. The final product is a language that is capable of being completely synthesized into hardware, as well as being completely compiled into software. In our approach, in other words, we do not limit the functionality of the hardware or the software, but instead support both in our language.

There has been limited progress in the field of combining hardware and software into a single unified language, and most of the research has been focused on *co-designed systems* [7] [8] [9]. In a co-designed system, a central processing unit (CPU) and a field programmable gate array (FPGA) are used in parallel to increase computational throughput. The CPU implements software, while the FPGA implements hardware. With this model, custom hardware can be created for the components of an application that are more demanding of time and resources (the key strength of hardware), and the more control-heavy components (the key strength of software) can be executed as usual on a CPU. While this model is powerful in principle, it must solve

difficult issues like scheduling, partitioning, and verification. In addition, co-design systems do not fully merge hardware and software design representations, but instead either reduce each representation to a smaller equivalent representation , or leave the two representations disjoint, with some functionality only available to one of the representations . Co-design solutions are available in two categories. The first category uses an existing software language, such as C, and tools that translate software into hardware that is capable of being ported to an FPGA. One example of this category is LegUp [7]. LegUp focuses on making hardware devices, such as FPGA's, easier for software programmers to use. To do this, LegUp partitions C code into two sets; one set will run on an FPGA, and the other set will run on a processor. These two sets operate in parallel to achieve the same result as the C code. By using a language that is already familiar to developers, this solution is potentially easier for the public to adopt. This solution has shown promise, however it is also limited by using C as its base language. This is because C inherently creates software, but lacks semantics to properly describe key aspects of hardware such as explicit parallelism, event driven models, and clocking. Also, the issue of partitioning the code and verifying the correct operation of the co-designed result is computationally highly difficult. Finally, such a paradigm needs to model and abstract hardware-software communication accurately. For these reasons, the resulting hardware partition of the co-design system is much less efficient than a solution created by an HDL. The second category has attempted co-design by creating a co-design language. Examples include the Bluespec Co-design Language (BCL) [8] from the Massachusetts Institute of Technology, as well as the Lime Language [9] from IBM. BCL uses atomic operations as the fundamental semantic instead of procedural statements. Lime is a Java based language designed to run in a co-design environment. Lime also supports existing Java code. These solutions have more potential,

however current attempts have produced limited success. These new languages step away from the familiarity of C, and introduce foreign and unfamiliar design constructs such as BCL's rule construct [8] and Lime's task and connect construct [9].

The language that we have created, Cv, has adopted the following design principles to avoid some of the problems of existing co-design languages:

- Create an all-hardware or all-software solution. This averts the problems related to co-design systems.
- Remove hardware or software syntactical elements that do not have meaning in the opposite design representation
- Explicit parallelism to give the designer full control over parallelism. This is based on the philosophy that the designer of the system is best equipped to make parallelism decisions.
- Syntactically simple event driven semantics.
- Use a syntax similar to C/C++ for high amounts of readability, teachability, and ensure a short learning curve. Since the Verilog syntax is also C-like, this choice is practical from a hardware design point of view as well.

In the remainder of this section, each of the above design principles is discussed in further detail.

Create an all-hardware or all-software solution

Cv approaches hardware and software design in a significantly different style, by giving the user the option to create either an all-hardware or all-software solution. Some co-design efforts target a primarily software audience, but most co-design solutions target embedded system applications

where hardware and software naturally work together. These solutions in turn neglect the majority of designers who create their design either only in hardware. Further, as discussed earlier, this requires addressing several technically challenging issues such as partitioning and verification, modeling of hardware-software communication, and difficulties modeling event-driven computation and clocking.

Remove hardware or software syntactical elements that do not have meaning in the opposite design representation

It is undesirable to have semantics and constructs that have meaning for only one of the two design representations. A finished design should be able to be executed in either design representation without modification. Therefore, hardware-only or software-only semantics would be intrusive to both the language and its purpose. For example, clocking in hardware triggers edge based flip flops, and therefore the flow of the hardware computation. However clocking does not translate into software where events are triggered by the processor clock. Another example (from software) is addressing variables that are stored at specific memory address locations. In hardware, variables are permanent physical bits that do not change location. Therefore addressing variables by their memory address does not translate to software. In subsequent enhancements of Cv, we may incorporate these elements.

Explicit parallelism to give the designer full control over parallelism

Parallel design is native to hardware and is becoming more common in software due to multithreading. Under the current multithreading paradigm, software parallelism experiences side effects and a large synchronization overhead. This is not true for hardware parallelism. In

hardware, parallel components cannot write to the same registers. We enforce the most abstract parallelism constraints in Cv. In Cv, multiple operations that are being computed in parallel cannot write a value to the same variable. However, multiple operations that are being computed in parallel can read from the same variable. Likewise, function calls can be made in parallel. Using the same semantic, function calls in parallel cannot write to the same variable, but function calls in parallel can read from the same variable. This allows the designer to explicitly describe a parallel structure in a manner that is agnostic to whether the final computation is being run in hardware or software.

Syntactically simple event driven semantics

Hardware and software both support event driven design models, however hardware natively supports event driven models through combinatorial logic and edge triggered flip-flops, while software uses event handlers and specialized classes. As event driven programming continues to grow in utility, it is only natural to combine these practices with well-developed hardware concepts into one uniform semantic. This is accomplished using Cv's event driven assignment. For example, in Cv, consider a variable (referred to as the *assigned variable*) that is assigned using the *event driven assignment*. When one of the variables in the right hand of the event driven assignment changes, the assigned variable is automatically updated using the new value of the right hand variable.

Use a syntax similar to C/C++ for high amounts of readability, teachability, and ensure a short learning curve

In order to achieve a high level of adoption, Cv has been designed to use a C/C++ like syntax so that both hardware and software designers will be able to quickly understand the language. Verilog, a very popular HDL, uses a C-like syntax. Our philosophy is that reading a design should not pose an additional burden to the designer, and therefore we have chosen to follow the C/C++ syntax style.

To test and demonstrate the functionality of Cv we have implemented a set of standard algorithms in Cv, Verilog, and C++. The resulting Cv hardware was compared to the Verilog hardware implementation in terms of FPGA look up tables (LUT's) used, and clock cycles required. The resulting Cv software was compared to the C++ implementation in terms of executable size, peak memory use, and execution time.

In the next chapter, we describe the different Cv constructs, and discuss how they are translated into software as well as hardware.

CHAPTER II

THE CV APPROACH

2.1 Overview

Cv is structured to take code describing a digital design and then either compile this code to software, or synthesize it into hardware. To do this the Cv compiler implements a *Cv Software Translator,* and a *Cv Hardware Translator.*

The Cv software translator translates the Cv code into an equivalent C++ representation. A *makefile* is then created to prepare the code for compilation. We use the G++ compiler [10] which provides the needed compiler, assembler, and linker. After these steps, the executable is available to be executed.

The Cv Hardware Translator translates the Cv code into an equivalent Verilog representation. The Verilog code is then imported into a *Xilinx ISE project* so that the design can be programmed onto an FPGA. The Xilinx ISE tools implement the needed synthesis, translate, map, place and route, and bitstream generation tools. After these steps, a bitstream is available to be programmed onto the FPGA.

These two paths are illustrated in Figure 1.

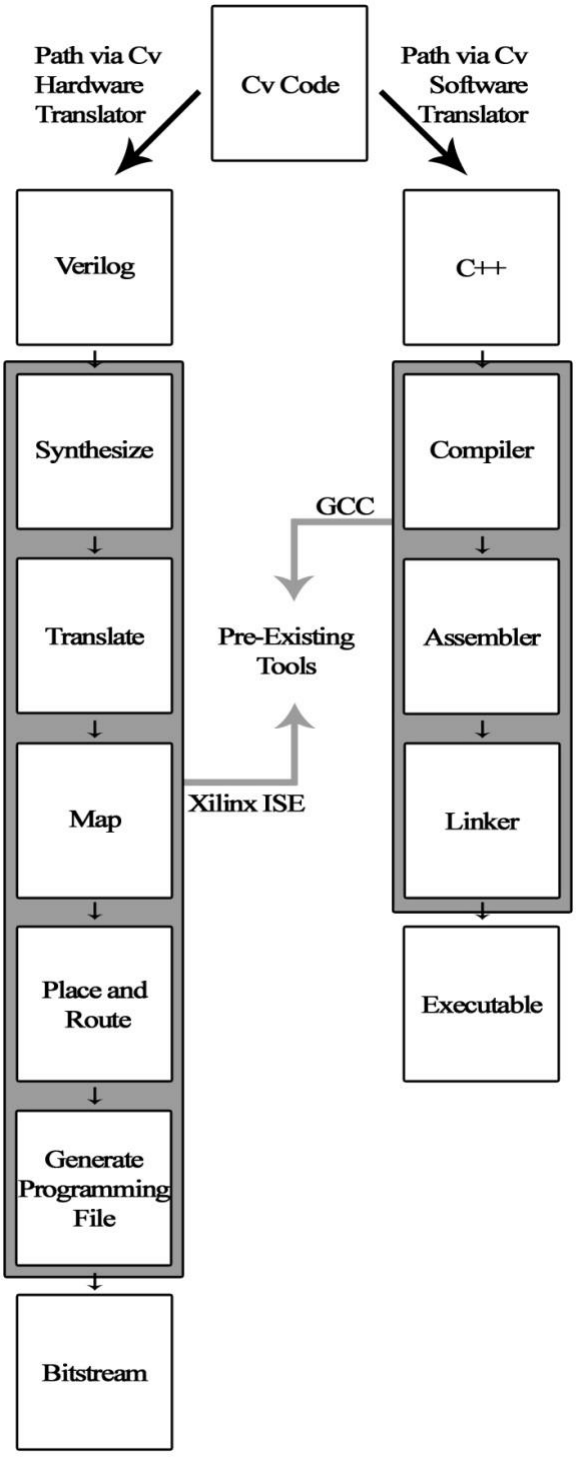

Figure 1.
Displays the separate branching paths of Cv, one into hardware, the other into software.

Because Cv strives to implement a single design representation that is capable of being executed on either software and hardware, each of the execution paths must be capable of implementing all features in the design representation. In order to combine the design representations of hardware and software, each representation must be modified to accommodate new design principles. Table 1 represents the semantic constructs that Cv implements.

Table 1.

Cv Constructs
(2.3) Functions
(2.4) Master Function
(2.5) Clocking
(2.6) Variables and Arrays
(2.7) CHard
(2.8) CvC and CvV Types
(2.9) Serial Statements
(2.10) Parallel Statements
(2.11) Non-Blocking Statements
(2.12) Event Driven Statements
(2.13) Function Call
(2.14) If Statement
(2.15) For Loop
(2.16) While Loop

(2.17) Return Statement
(2.18) Comments

List of the semantic constructs implemented in Cv. The numbers in parenthesis indicate the section for each item.

Next, the grammar for Cv is presented. Following this, each semantic construct presented in Table 1 is discussed. Examples for how to use each semantic construct are provided, as well as examples of how the constructs are translated into both C++ and Verilog.

2.2 Grammar

The grammar presented in Figure 2 describes the Cv language. From Sections 2.3 through 2.18, for each semantic construct (from Table 1) a description based on this grammar is provided first, followed by an example, then a description of how the Cv Software Translator handles the construct, and finally a description of how the Cv Hardware Translator handles the construct.

```
Program          ::= FunctionDecl+
FunctionDecl     ::=Formals ident ( Formals ) StmtBlock
Formals          ::=Variable +,|ε
VariableDecl     ::=Variable ;
Variable         ::=Type ident | Type ident [ intConstant ] |
                    Type ident [ intConstant ] [ intConstant ]
Type             ::=AdjustableType | double | bool | char
AdjustableType   ::= int | string | int < intConstant > | string < intConstant >
StmtBlock        ::={ VariableDecl* EventDriven* Stmt* }
Stmt             ::=<AssignExpr>; | IfStmt | WhileStmt | ForStmt | ReturnStmt |
                    PrallelStmt | StmtBlock
IfStmt           ::=if ( Expr ) Stmt  < else Stmt >
WhileStmt        ::=while ( Expr ) Stmt
ForStmt          ::=for ( <Expr>; Expr; <Expr> ) Stmt
ReturnStmt       ::= return ;
ParallelStmt     ::=Stmt || Stmt | ParallelStmt ||  Stmt
AssignExpr       ::=LValue = Expr | ( ident +, )  = Call | LValue <> NonCallExpr
EventDriven      ::= LValue ~ NonCallExpr ;
Expr             ::=Constant | LValue | Call | ( Expr ) | Expr + Expr | Expr – Expr |
                    Expr * Expr | Expr / Expr | Expr % Expr | - Expr | Expr < Expr |
                    Expr <= Expr | Expr > Expr | Expr >= Expr | Expr == Expr |
                    Expr != Expr | Expr & Expr | Expr | Expr | ! Expr | Expr >> Expr |
                    Expr << Expr
NonCallExpr      ::= Constant | LValue | (NonCallExpr) |
                    NonCallExpr + NonCallExpr |NonCallExpr – NonCallExpr |
                    NonCallExpr * NonCallExpr | NonCallExpr / NonCallExpr |
                    NonCallExpr % NonCallExpr | - NonCallExpr |
                    NonCallExpr < NonCallExpr |NonCallExpr <= NonCallExpr |
                    NonCallExpr > NonCallExpr | NonCallExpr >= NonCallExpr |
                    NonCallExpr == NonCallExpr | NonCallExpr != NonCallExpr |
                    NonCallExpr & NonCallExpr | NonCallExpr | NonCallExpr |
                    ! NonCallExpr | NonCallExpr << NonCallExpr |
                    NonCallExpr >> NonCallExpr
LValue           ::=ident | ident [ Expr ]
Call             ::=ident ( Actuals ).ident | ident ( Actuals )
Actuals          ::= Expr +, | ε
Constant         ::=intConstant | doubleConstant | boolConstant | stringConstant  |
                    charConstant
```

Figure 2.
Grammar of Cv.

2.3 Functions

Functions are the highest level construct that contain all other constructs. According to the grammar:

FunctionDecl ::= Formals **ident (** Formals **)** StmtBlock

Formals are a comma separated list of variable declarations (Section 2.4). Ident is the name of the function. The Formals before the function name are the outputs of the function, and the Formals within parenthesis are the inputs of the function. The StmtBlock after the inputs contains the code that the function will implement.

2.3.1 Example

```
//Cv Code
int output1, int output2 MyFunction ( int input1, int input2, int input3 )
{
	//empty function
}
```

Figure 3.
Example function declaration in Cv.

In Figure 3 `output1` and `output2` are the outputs of `MyFunction`, and both outputs are integers. `MyFunction` is the name of the function, and `input1`, `intput2`, and `input3` are the inputs of `MyFunction`. `MyFunction` does not have any code to implement. In general, the StmtBlock of `MyFunction` will utilize other constructs are discussed later in this chapter. One of the differences between Cv and common software languages is that the output variables are directly manipulable in Cv. Although return statements can be used in Cv, they are not required. When a

function has reached its end, the value of the outputs will be returned, based on the values that the outputs were set to in the function.

2.3.2 Cv Software Translator

When translating the function declaration to C++, the CV software translator will create a struct for the outputs, copy over the name of the function, and make a CvV type (Section 2.8) for each of the inputs.

```
//C++ Code
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
	CvV<int> output2;
};
CvFunctionStruct_MyFunction MyFunction(
	CvV<int> input1,
	CvV<int> input2,
	CvV<int> input3
) {
CvFunctionStruct_MyFunction returnValue;
return returnValue;
}
```

Figure 4.
Function declaration translated from Cv to C++.

It can be seen from Figure 4 that each of the outputs is placed in the struct `CvFunctionStruct_MyFunction`. In this case each of the outputs use the `CvV<int>` type (Section 2.8). The output struct is also created as a local variable of the function, and named `returnValue`. This allows for the function to directly set the outputs, as it will be shown in later examples. Once the function has reached the end of the function code, or a return statement, the function returns the `returnValue`. The inputs are also translated into `CvV` types.

2.3.3 Cv Hardware Translator

Given the example, the Cv Hardware Translator produces the following Verilog code.

```
//Verilog Code
module MyFunction( input clock, input reset, input start,
      input [31:0] input1,input [31:0] input2, input [31:0] input3,
      output reg done, output reg [31:0] output1, output reg [31:0] output2
);

endmodule
```

Figure 5.
Function declaration in Cv translated to Verilog.

Figure 5 demonstrates how functions in Cv are translated into Verilog. The inputs to the functions are created as input ports into the module, and the outputs of the functions are created as output ports of the module. In this example each of the inputs and outputs are 32 bits wide. This is because 32 is the default bit width for integers in Cv (Section 2.6). Additional inputs include `clock`, `reset`, and `start`. The additional output is `done`. These inputs and outputs are automatically created by the Cv hardware translator for every function.

- `clock`: This is the input for the clock, and controls clocking for all modules. Cv creates hardware that is driven off of a clock.
- `reset`: This input controls the reset for the system. Every module created has a reset input which, when applied, sets all variables to default values.
- `start`: This input indicates to the module to begin its computation.
- `done`: This output indicates to other modules that this module has finished its computation. When other modules see the done signal of this module they may begin to use the outputs of this module.

For functions that are not empty, their Cv code (between the end of the declarations of the ports to the `endmodule` statement) will be translated to Verilog code.

2.4 Master Function

In most software languages a main function is defined that indicates where the program is going to begin executing. In HDL's any module can be selected as the top module in the project. In Cv the *master function* is specified when the code is compiled. When translating Cv to C++, the master function is converted to the main function in C++. When translating Cv to Verilog, the master function is set as the top level module of the project. The master function is not an item in the grammar, but instead is decided at compile time.

2.4.1 Example

```
//Cv Code
int output1 MyFunction ( int input1 ){
      //empty function
}
```

Figure 6.
Example function in Cv to be compiled as the master function.

2.4.2 Cv Software Translator

When compiling Cv to software, the follow command will invoke the Cv compiler and instruct the compiler to compile `MyFunction` as the main function in software.

(Terminal Command) ./cvc < code.cv MyFunction –sw

In the statement above, "./cvc" invokes the Cv compiler. Next, "< code.cv" instructs the Cv compiler to compile the file "code.cv". The next option given is the name of the master function

in the “code.cv”. In this example that function is “MyFunction”. The final parameter given is “–sw”. This tells the Cv compiler to translate the given code into software.

2.4.3 Cv Hardware Translator

When compiling Cv to hardware, the follow command will invoke the Cv compiler and instruct the compiler to compile MyFunction as the top level module in hardware.

(Terminal Command) ./cvc < code.cv MyFunction –hw

In the statement above, “./cvc” invokes the Cv compiler. Next, “< code.cv” instructs the Cv compiler to translate the file “code.cv”. The next option given is the name of the function which should be treated as the top level module. In this example that function is “MyFunction”. The final parameter given is “–hw”. This instructs the Cv compiler to translate the given code into hardware.

2.5 Clocking

In hardware, one (and arguably the most popular) style of design involves triggering events to occur off of a clock edge. This is demonstrated in Verilog using *always* block that trigger at either the rising or falling edge of a clock. For example:

```
//Verilog Code
always@(posedge clock) begin
      a <= b + 1;
end
```

Figure 7.
Always block in Verilog.

In Cv there are no explicit clocking constructs. Instead when a function is translated to hardware, each function will either run in a single clock cycle, or multiple clock cycles (if the function requires CHard, as described in Section 2.7). When a function is translated to software, nothing special happens with respect to clocking, since the function does not have explicit clocking constructs to begin with.

Since clocking in Cv is therefore implicit, no examples are presented in this section, and no Cv Software Translator rules or Cv Hardware Translator rules are presented.

2.6 Variables and Arrays

This section focuses on variable declarations and how they are translated into C++ and Verilog. According to the grammar:

VariableDecl ::= Variable ;

Variable ::= Type **ident** | Type **ident [intConstant]**|Type **ident [intConstant] [intConstant]**

Type ::= AdjustableType | double | bool | char

AdjustableType ::= **int** | **string** | **int < intConstant >** | **string < intConstant >**

This reflects the types supported by Cv; integers, double precision floating point numbers, Boolean values, characters, and strings. The default bit widths are:

- 32 bits for an integer
- 64 bits for a double
- 1 bit for a bool
- 8 bits for a character

- 32 characters, or 256 bits, for a string.

In addition to the default values, the bit width of integers, and the number of characters in a string, can be specified by the user. According to the definition of *AdjustableType*, integers and strings can be declared with or without angle brackets. When there are no angle brackets, the bit width of an integer or string is the default bit width. When an integer is declared with angle brackets, then the bit width of the integer is the value given between the angle brackets. When a string is declared with angle brackets, then the number of characters in the string is equal to the number in angle brackets. Integers and strings declared with angle brackets are said to have a *specified size*. Arrays can also be declared using integers and strings with a specified size. Cv supports 1- and 2- dimensional arrays. Arrays must also be fixed size. The integer constant in brackets determines the size of the array.

We next present some examples. We first cover basic variable declarations, then variable declarations with a specific size, and finally array declarations. After this, we discuss the translation of various kinds of variable declarations into C++ and Verilog.

2.6.1 Basic Variable Declaration

To declare an `int` without a specified size:

```
//Cv Code
int myInt;
```

Figure 8.
Declaring an `int` variable in Cv.

Here `int` determines that the variable is an integer, and `myInt` is the name of the variable. Because there is no specified size, `myInt` is 32 bits. Next are examples creating variables of type `double`, `bool`, `char`, and `string`:

```
//Cv Code
double myDouble;
bool myBool;
char myChar;
string myString;
```

Figure 9.
Declaring different types of variables in Cv.

Here `myDouble` is 64 bits, `myBool` is 1 bit, `myChar` is 8 bits, and `myString` is 32 characters or 256 bits.

2.6.2 Variable Declaration with Specified Size

To declare an `int` with a specified size:

```
//Cv Code
int <50> myIntSize50;
```

Figure 10.
Declaring an integer with a specified size.

Here `int` determines that the variables is an integer, the 50 in angle brackets indicates that the variable has 50 bits, and `myIntSize50` is the name of the variable.

Using a specified size to declare a `string` can be accomplished as follows:

```
//Cv Code
```

```
string <100> myLongString;
```

Figure 11.
Declaring a `string` with a specified number of characters.

Here `string` determines that the variable is a string, the 100 in the angle brackets indicates that the variable has 100 characters, or 800 bits, and `myLongString` is the name of the variable.

2.6.3 Declaring Arrays

Creating an integer array with default integer bit width:

```
//Cv Code
int myIntArray [10];
```

Figure 12.
Declaring an array of integers in Cv.

Here, `myIntArray` is an array of 10 integers with 32 bits per integer. Arrays of the other types can be declared likewise.

```
//Cv Code
double myDoubleArray [20];
bool myBoolArray [10];
char myCharArray [4];
string myStringArray [15];
```

Figure 13.
Declaring arrays of different types in Cv.

In these declarations each array is of the Type specified, and each item in each array has the default bit width corresponding to its Type. To create an array with a specified size for integers and strings:

```
//Cv Code
int <64> myIntArraySize64 [20];
string <40> myStringArrayLarger [10];
```

Figure 14.
Declaring an array of integers and an array of strings with a specified size.

In `myIntArraySize64` there are 20 integers each with 64 bits. In `myStringArrayLarger` there are 10 strings each with 40 characters.

2.6.4 Example

```
//Cv Code
int output1 MyFunction (int input1) {
	int myInt;
	double myDouble;
	bool myBool;
	char myChar;
	string myString;

	int <16> myIntSmaller;
	string <64> myStringLarger;

	int myIntArray [20];
	double myDoubleArray [10];
	bool myBoolArray [5];
	char myCharArray [16];
	string myStringArray [8];

	int <64> myIntArraySized [10];
	string <16> myStringArraySized [20];
}
```

Figure 15.
A function containing variable declarations in Cv.

The function in Figure 15 will be used as an example of a function containing variable declarations. This subset of variables represents the different variables that can be used in Cv.

2.6.5 Cv Software Translator

```
//C++ Code
struct CvFunctionStruct_MyFunction {
      CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction( CvV<int> input1 ) {
      CvV<int> myInt;
      CvV<double> myDouble;
      CvV<bool> myBool;
      CvV<char> myChar;
      CvV<string> myString;

      CvV<int> myIntSmaller;
      CvV<string> myStringLarger;

      CvV<int> myIntArray [20];
      CvV<double> myDoubleArray [10];
      CvV<bool> myBoolArray [5];
      CvV<char> myCharArray [16];
      CvV<string> myStringArray [8];

      CvV<int> myIntArraySized [10];
      CvV<string> myStringArraySized [20];
}
```

Figure 16.
Translating variables within a function from Cv to C++.

Each variable uses the same name in C++ that was given in Cv. Integers and strings with a specified size use standard sizes in C++. Each variable is of the type *CvV* (Section 2.8)

2.6.6 Cv Hardware Translator

```
//Verilog Code
module MyFunction(input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] myInt;
reg [63:0] myDouble;
reg myBool;
reg [7:0] myChar;
reg [255:0] myString;

reg [15:0] myIntSmaller;
reg [63:0] myStringLarger;

reg [31:0] myIntArray [0:19];
reg [63:0] myDoubleArray [0:9];
```

```
reg myBoolArray [0:4];
reg [7:0] myCharArray [0:15];
reg [255:0] myStringArray [0:7];

reg [63:0] myIntArraySized [0:9];
reg [127:0] myStringArraySized [0:19];

endmodule
```

Figure 17.
Translating variable within a function from Cv to Verilog.

The above code reflects how each variable type is translated into Verilog. Each variable is made into a `reg` variable type in Verilog. The bit width of the variables is declared by the bracketed numbers on the left of each variable. If a variable does not have brackets indicating its size, the variable is 1 bit. Each of the bit widths matches the bit width given in the Cv code, and when there is no bit width specified in the Cv code, the default bit width is used. For the arrays, Cv creates a Verilog array. This can be seen by the bracketed numbers to the right of the variables.

2.7 CHard

Cv takes the design descriptions of software and hardware and creates a single design description. There are semantics of both software and hardware that do not have an identical meaning in the opposite domain. Therefore a hardware construct has been created to handle software semantics that are not compatible with hardware design. This new hardware construct is called *CHard*. CHard is not an item in the grammar, but instead is implemented by multiple elements of the grammar. Because CHard is designed to handle software semantics foreign to hardware, there is no implementation of CHard for the Cv Software Translator, it is only used for the Cv Hardware Translator.

2.7.1 Example that Requires CHard

The example below shows a for loop (Section 2.15) whose number of iterations is unknown at compile time.

```
//Cv Code
int output1 myFunction (int input1) {
      int i;

      output1 = 0;
      for(i  = 0; i < input1; i = i + 1)
            output1 = output1 + 1;
}
```

Figure 18.
Cv code that requires CHard.

In the code above a for loop is used, however the number of iterations of the for loop is unknown, and depends on the value of `input1`. This type of for loop, where the number of iterations are unknown at compile time, is unsupported in currents HDL's. Therefore CHard is used to realize this software semantic in hardware. The function is translated into Verilog as shown below. The constructs of CHard are then further explained.

2.7.2 Cv Hardware Translator code for Section 2.7.1

```
//Verilog Code
module myFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [3:0] functionCounter;

//flow block
always@(posedge clock) begin
      if(reset) begin
            output1 <= 0;
            i <= 0;
      end
      else if(start) begin
            output1 <= 0;
      end
```

```
      else if(functionCounter != 0) begin
            case(functionCounter)
                  1: i <= 0;
                  2: // do nothing, control block is testing i < input1
                  3: output1 <= output1 + 1;
                  4: i <= i + 1;
                  5: //do nothing, control block is testing i < input1
            endcase
      end
      else begin
            output1 <= output1;
            i <= i;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            functionCounter <= 0;
            done <= 0;
      end
      else if(start) begin
            functionCounter <= 1;
            done <= 0;
      end
      else if(functionCounter != 0) begin
            case(functionCounter)
                  2:
                        if(i < input) begin
                              functionCounter <= 3;
                        end
                        else
                              functionCounter <= 0;
                              done <= 1;
                        end
                  5:
                        if(i < input) begin
                              functionCounter <= 3;
                        end
                        else
                              functionCounter <= 0;
                              done <= 1;
                        end
                  default:
                        functionCounter <= functionCounter + 1;
                        done <= done;
            endcase
      end
      else begin
            functionCounter <= 0;
            done <= done;
      end
end
endmodule
```

Figure 19.
Translating Cv to Verilog with code containing CHard.

First the *flow block* is discussed, followed by the *control block*. The flow block implements the code that was given in the function. When `reset` is high, all local variables and outputs of the function are set to 0. The instructions required to be computed by Figure 19 are `output1 = 0`, `i = 0`, `i < input1`, and `i = i + 1`. The first instruction is executed when the `start` input is high, which sets `output1 = 0`. The remaining instructions are executed in the case statement which depends on the value of the *functionCounter*. This is how the flow block operates; the first statement in the code is executed when `start` is high, and the rest of the code is executed in the case statement. The next code that is encountered is the for loop.

The control block uses the `functionCounter` and `done` variables to control which statement is executed. The number of bits for the `functionCounter` is determined by the Cv compiler, so that the minimum number of bits may be used to account for all the instructions that must be executed. In this example a for loop must test `i < input1` before the for loop is run the first time, and it must retest this condition after each execution of the for loop. When the `functionCounter` value is 2 or 5, the flow block is not doing anything and the control block is testing the condition of the for loop. Once `i < input1` is no longer true, the execution would exit the for loop. Because the for loop is the last code in the function, once the for loop is finished, the function is finished. Therefore, once `i < input1` is no longer true, the `functionCounter` is set to 0, and `done` is set to 1. These two blocks execute in parallel, very similarly to how the code would execute on a processor.

One of the goals of Cv is to provide hardware that is efficient. Therefore, CHard is only created when it is required. If a designer uses constructs familiar to a hardware developer, then the Cv

compiler will not create CHard, and the resulting Verilog will not rely on a `functionCounter`. However, functions that do not require CHard will still have a `done` output. The `done` output of a function that does not require CHard is always high the clock cycle after the `start` input is high. Each of the constructs presented demonstrates cases when CHard is, and is not, created.

Next is an example of a function that does not require CHard. This function only does assignments, and therefore does not rely on software constructs foreign to hardware.

2.7.3 Example that Does Not Require CHard

```
//Cv Code
int output1 MyFunctionNoCHard( int input1 ) {
      int x;
      int y;
      x = input1*2;
      y = x - 3;
      output1 = y*4;
}
```

Figure 20.
Cv code that does not require CHard.

In Figure 20 only assignments are performed. HDL's provide semantics to do serial operations using *blocking statements*. Therefore, the code in Figure 20 can be translated using blocking statements, and not require CHard.

2.7.4 Cv Hardware Translator

```
//Verilog Code
module MyFunctionNoCHard (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] x;
reg [31:0] y;

//flow block
always@(posedge clock) begin
      if(reset) begin
            x = 0;
            y = 0;
            output1 = 0;
      end
      else if( start ) begin
            x = input1*2;
            y = x - 3;
            output1 = y*4;
      end
      else begin
            x = x;
            y = y;
            output1 = output1;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
      end
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      end
end
endmodule
```

Figure 21.
Translating Cv to Verilog with code that does not contain CHard.

In Figure 21, the flow block and the control block still exist. However, the control block no longer contains the `functionCounter`. The `done` output is set by the control block to be high the clock cycle after the `start` input is high. In the flow block all assignments are done through blocking statement, contrary to the non-blocking statements used in the example that required CHard (Section 2.7.2). This is designed so that statements will be implemented sequentially. The

control block continues to use non-blocking statements for continuity. In the flow block, functions that do not require CHard have three sections; when `reset` is high, when `start` is high, and otherwise. When `reset` is high all local variables and outputs are set to 0. When `start` is high the instructions given by the user are implemented. When neither `reset` nor `start` are high, then the local variables and outputs retain their value. This allows other modules to then use these outputs in their computations.

2.8 CvC and CvV Types

Because Cv expands upon traditional software functionality, certain modifications must be made when translating Cv to C++. One of the greatest changes is the introduction of the *CvV* type. The CvV type is applied to all variables when the Cv Software Translator translates Cv code into C++. The CvV type allows variables to have the functionality of the event driven semantic, a semantic absent from C++. To implement the CvV type, two classes were created, the CvC class and the CvV class. These classes are presented below.

CvC Class

```
//C++ code
class CvC
{
public:
	CvC(void(*function)(CvC*))
		: _function(function) {
		_dependors = vector<CvC*>();
		_dependencies = vector<CvC*>();
	}
	void Update( ){
		_function(this);
		for(int i = 0; i < _dependors.size(); i++)
			_dependors[i]->Update( );
	}
	void AddDependency(CvC* cvC){
		cvC->AddDependor(this);
```

```
            _dependencies.push_back(cvC);
      }
      void AddDependor(CvC* cvC){
            _dependors.push_back(cvC);
      }
      void AddFunction( void(*function)(CvC*) ) {
            _function = function;
      }
      void RemoveFunction( ) {
            _function = NULL;
      }
      template<typename T>
      T Me( ){
            if(dynamic_cast<CvV<T>*>(this))
                  return dynamic_cast<CvV<T>*>(this)->Me();
            return 0;
      }
      template<typename T>
      void SetValue( T value ){
            if(dynamic_cast<CvV<T>*>(this))
                  dynamic_cast<CvV<T>*>(this)->SetValue(value);
      }
      template<typename T>
      void AfterParallelUpdate( ) {
            if(dynamic_cast<CvV<T>*>(this))
                  dynamic_cast<CvV<T>*>(this)->AfterParallelUpdate( );
      }
      vector<CvC*> GetDependencies( ){
            return _dependencies;
      }
protected:
      void(*_function)(CvC*);
      vector<CvC*> _dependors;
      vector<CvC*> _dependencies;
};
```

Figure 22.
C++ code of the CvC class.

CvV Class

```
//C++ Code
template<class T>
class CvV : public CvC
{
public:
      CvV( T value, void(*function)(CvC*) )
            : _value(value), _writeValue(value), CvC(function){ }
      CvV( )
            : _value(0), _writeValue(0), CvC(NULL){ }
      T& Me( ) { return _value; }
      void SetValue( T value ){ _value = value; }
      CvV( T value )
            : _value(value), _writeValue(value), CvC(NULL){ }
      CvV( const CvV<T>& cvV )
            : _value(cvV._value), _writeValue(cvV._writeValue),
```

```
        CvC(cvV function){ }
void AfterParallelUpdate() {
      _value = _writeValue;
}
CvV<T> operator=( T rightSide ){
      _value = rightSide;
      _writeValue = _value;
      for(int i = 0; i < _dependors.size(); i++)
            _dependors[i]->Update();
      return *this;
}
CvV<T> operator=( CvV<T> rightSide ){
      _value = rightSide.Me();
      _writeValue = _value;
      for(int i = 0; i < _dependors.size(); i++)
            _dependors[i]->Update();
      return *this;
}
CvV<T> operator^(T rightSide){
      _writeValue = rightSide;
      for(int i = 0; i < _dependors.size(); i++)
            _dependors[i]->Update();
      return *this;
}
CvV<T> operator^(CvV<T> rightSide){
      _writeValue = rightSide.Me();
      for(int i = 0; i < _dependors.size(); i++)
            _dependors[i]->Update();
      return *this;
}
CvV<T> operator+(T value){ return CvV<T>(_value + value, _function);}
CvV<T> operator+(CvV<T> value){
      return CvV<T>(_value +  value.Me(),_function);
}
CvV<T> operator-(T value){ return CvV<T>(_value - value, _function);}
CvV<T> operator-( CvV<T> value ){
      return CvV<T>(_value - value.Me(), function);
}
CvV<T> operator*( T value ){
      return CvV<T>(_value * value, _function);
}
CvV<T> operator*( CvV<T> value ){
      return CvV<T>(_value * value.Me(), function);
}
CvV<T> operator/( T value ){
      return CvV<T>(_value / value, _function);
}
CvV<T> operator/( CvV<T> value ){
      return CvV<T>(_value / value.Me(), _function);
}
CvV<T> operator%( T value ){return CvV<T>(_value % value,_function);}
CvV<T> operator%(CvV<T> value){
      return CvV<T>(_value%value.Me(),_function);
}
bool operator<( T value ){ return _value < value; }
bool operator<( CvV<T> value ){ return _value < value.Me(); }
bool operator<=( T value ){ return _value <= value; }
bool operator<=( CvV<T> value ){ return _value <= value.Me(); }
bool operator>( T value ){ return _value > value; }
bool operator>( CvV<T> value ){ return _value > value.Me(); }
bool operator>=( T value ){ return _value >= value; }
bool operator>=( CvV<T> value ){ return _value >= value.Me(); }
```

```
      bool operator==( T value ){ return _value == value; }
      bool operator==( CvV<T> value ){ return _value == value.Me(); }
      bool operator!=( T value ){ return _value != value; }
      bool operator!=( CvV<T> value ){ return _value != value.Me(); }

      template<class H>
      T ArrayAccess( CvV<H> index ){
            CvV<T>* cvPointer = this;
            for(int i = 0; i < index.Me(); i++)
                  cvPointer++;
            return cvPointer->Me();
      }
private:
      T _value;
      T _writeValue;
};
```

Figure 23.
C++ code for the CvV class.

These two classes are used to implement the event driven models and parallel models in C++. In the subsequent sections that use specific features of the CvC and CvV classes, the needed functionality is discussed.

Serial, Parallel, Non-Blocking, and Event Driven Statements

In Cv statements can be made serially, in parallel, as non-blocking assignment, or as event driven assignments. Each of these is described in detail in the following sections.

2.9 Serial Statements

Serial statement are reflected in the grammar by the definition of the StmtBlock:

StmtBlock ::= **{** VariableDecl* EventDriven* Stmt* **}**

Stmt* is a pointer to an array of statements. Statements occurring one after another are serial statements, and therefore are executed in the order they are written.

2.9.1 Example

```
//Cv Code
int output1 MyFunction(int input1) {
	int a;
	int b;

	a = 3*input1;
	b = a – 3;
	ouptut1 = b/2;
}
```

Figure 24.
Cv code using serial statements.

Figure 24 shows the assignments in series. The translated software and hardware must execute these statements in the same order.

2.9.2 Cv Software Translator

Serial statements have a very direct translation when translated into software. The result looks very similar to the given Cv code.

```
//C++ Code
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvV<int> a;
	CvV<int> b;
	CvFunctionStruct_MyFunction returnValue;

	a = 3*input1;
	b = a – 3;
	Value.output1 = b/2;
	return returnValue;
}
```

Figure 25.
Translating Cv to C++ for serial statements.

Because the = operator in the CvV class has been overridden, the translated code just needs to assign a value to the variable. This will set `_value` and `_writeValue` of the variable. The output variable is able to be directly assigned within the function. Once the function has reached the end of its code, the `returnValue` is returned, containing the output variable that has been properly set. The serial statements occur in order, as given in the Cv Code.

2.9.3 Cv Hardware Translator

When translating Cv to hardware, serial statements may or may not require CHard. If a function contains only a block of serial statements, then the code will not require CHard. However, if the code contains other constructs that requires CHard, then the serial statements will require CHard. For examples that require CHard, we refer the reader to the sections on the individual constructs that require CHard (Sections 2.10.6, 2.13.2.1, 2.13.2.2, 2.14.2.2, 2.14.2.4, 2.15.2.3, 2.16.2.1, and 2.17.3). In the example given in Figure 24, the function just contains a series of assignments. Therefore, this function does not contain any CHard constructs and does not require CHard.

```
//Verilog Code
module myFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] a;
reg [31:0] b;

//flow block
always@(posedge clock) begin
      if(reset) begin
            a = 0;
            b = 0;
            output1 = 0;
      end
      else if( start ) begin
            a = 3*input1;
            b = a - 3;
            output1 = b/2;
      end
      else begin
            a = a;
```

```
            b = b;
            output1 = output1;
      end
end
endmodule

//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
      end
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      end
end
endmodule
```

Figure 26.
Translating Cv to Verilog for serial statements.

In Figure 26, no CHard is created. All of the functionality is executed within one clock cycle as expected.

2.10 Parallel Statements

According to the grammar:

ParallelStmt ::= Stmt || Stmt | ParallelStmt || Stmt

Two or more statements are declared in parallel using the || symbol. In addition to this, multiple statements can be chained in parallel using the || symbol with successive statements. Statements in parallel are executed simultaneously. All variables on the right side of parallel assignments will use the value of the variable prior to entering the parallel section. Once all right hand

expressions have been evaluated, the variables on the left are assigned the results. A variable can only be assigned a value in one parallel statement.

```
//Cv Code
a = 2 + b;
|| b = 3 - a + c;
|| c = 3*b;
```

Figure 27.
Cv code showing parallel construct.

In Figure 27 the three statements are executed in parallel. This is declared using the parallel symbol, ||. It is read that the assignment to b is in parallel with the statement above it, the assignment to a. Also, the assignment to c is in parallel with the statement above it, the assignment to b, which is also in parallel to the assignment to a. If a is initially 3, b is initially 5, and c is initially 6, then after execution the resulting values for a, b, and c would be 7, 6, and 15 respectively. This is because neither a, nor b, nor c will change their value until all expressions are evaluated.

Two sets of parallel statement can be structured as follows to create parallel statements in series:

```
//Cv Code
//Parallel Statement Set 1
a = 2 + b;
|| b = 3 - a + c;
|| c = 3*b;
//Parallel Statement Set 2
a = b - 3;
|| b = a*2;
|| c = a + b;
```

Figure 28.
Cv code showing how the parallel construct can be used to place two sets of parallel statements in series.

First, the top three assignments will occur in parallel. Once these calculations are completed, the second set of three assignments occur in parallel. This is denoted by the break in the parallel symbols. The assignment to a in Parallel Statement Set 2 is not prefaced by a parallel symbol, and therefore indicates that it must take place after Parallel Statement Set 1.

Next, the examples from Figure 27 and Figure 28 are placed in functions, and it is shown how the examples are translated into software and hardware. The first example is presented, followed by its translation, and then the second example is presented, followed by its translation.

2.10.1 First Example

```
//Cv Code
int output1 MyFunction(int input1) {
      int a;
      int b;
      int c;

      a = 2 + b;
      || b = 3 - a + c;
      || c = 3*b;
}
```

Figure 29.
Cv code with parallel statements in a function.

Figure 29 has a function containing three statements that execute in parallel.

2.10.2 Cv Software Translator

```
/C++ Code
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvV<int> a;
	CvV<int> b;
	CvV<int> c;
	CvFunctionStruct_MyFunction returnValue;

	a ^ 2 + b;
	b ^ 3 - a + c;
	c ^ 3*b;
	a.AfterParallelUpdate( );
	b.AfterParallelUpdate( );
	c.AfterParallelUpdate( );
	return returnValue;
}
```

Figure 30.
Translating Cv to C++ for a function containing parallel statements.

In the translation in Figure 30 there are two main differences when compared to the serial statement example. The first is that the assignment is done with the caret (^) operator, and after the caret assignments each of the variables calls the *AfterParallelUpdate* function. The caret operator is used during parallel assignments to set the of each variable's `_writeValue` inside of the CvV class (Section 2.8). In Cv when variables are assigned values in parallel, all of the right hand expression must be evaluated before the left hand variables are updated. This is accomplished by every variable having a `_value` and a `_writeValue`. When an assignment is performed using the = operator in C++, both the `_value` and `_writeValue` are set in the CvV variable. However, when assignment is performed using the ^ operator, only the `_writeValue` is assigned. When the value of a variable is read, the `_value` of the CvV variable is what is read. Therefore, the ^ operator writes the assigned right hand value to the `_writeValue` of the variable and other statements read the `_value` of the variable. Once all of the parallel statements have executed, each variable that was assigned a value calls the AfterParallelUpdate function of the

CvV class. This function copies the value of `_writeValue` over to `_value`, and updates all *dependors* of the class. For more information on dependors, see the Event Driven section (Section 2.12). At the end of the function the `returnValue` is still returned, even though nothing was assigned to it.

One of the remaining goals of Cv is to use threads to execute parallel statements in software. Currently parallel statements are executed in series. In the future, parallel statements will be executed using threads for true parallelism.

2.10.3 Cv Hardware Translator

```
//Verilog Code
module myFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] a;
reg [31:0] b;
reg [31:0] c;

//flow block
always@(posedge clock) begin
      if(reset) begin
            a <= 0;
            b <= 0;
            c <= 0;
            output1 = 0;
      end
      else if(start) begin //parallel statements executed here
            a <= 2 + b;
            b <= 3 - a + c;
            c <= 3*b;
      end
      else begin
            a <= a;
            b <= b;
            c <= c;
            output1 = output1;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
```

```
            done <= 0;
      end
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      end
end
endmodule
```

Figure 31.
Translating Cv to Verilog for a function containing parallel statements.

Because the only statements in this function are parallel statements, the function does not require CHard. Instead, all of the parallel assignments happen simultaneously at the rising edge of the clock.

2.10.4 Second Example

```
//Cv Code
int output1 MyFunction(int input1) {
      int a;
      int b;
      int c;

      //Parallel Statement Set 1
      a = 2 + b;
      || b = 3 - a + c;
      || c = 3*b;
      //Parallel Statement Set 2
      a = b - 3;
      || b = a*2;
      || c = a + b;
}
```

Figure 32.
Cv code with two sets of parallel statements in series within a function.

In Figure 32 two sets of parallel statements are executed in series. Parallel Statement Set 1 is executed, and once it finishes, Parallel Statement Set 2 is executed. Both software and hardware must respect this order of operation.

2.10.5 Cv Software Translator

```
//C++ Code
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvV<int> a;
	CvV<int> b;
	CvV<int> c;
	CvFunctionStruct_MyFunction returnValue;

	//Parallel Statement Set 1
	a ^ 2 + b;
	b ^ 3 – a + c;
	c ^ 3*b;
	a.AfterParallelUpdate( );
	b.AfterParallelUpdate( );
	c.AfterParallelUpdate( );
	//Parallel Statement Set 2
	a ^ b - 3;
	b ^ a * 2;
	c ^ a + b;
	a.AfterParallelUpdate( );
	b.AfterParallelUpdate( );
	c.AfterParallelUpdate( );
	return returnValue;
}
```

Figure 33.
Translating Cv to C++ for a function containing two sets of parallel statements in series.

This example is very similar to the previous in terms of how the caret operator and AfterParallelUpdate functions work. Because this function has two sets of parallel statements in series, Parallel Statement Set 1 is executed and `a`, `b`, and `c` each calling AfterParallelUpdate. Next, Parallel Statement Set 2 is executed, followed by `a`, `b`, and `c` each calling AfterParallelUpdate.

2.10.6 Cv Hardware Translator

Unlike Section 2.10.3, which did not require CHard, this example will require CHard. This is because there are two sets of parallel statements performed in series. This means that the second set of parallel statements cannot be computed until after the first set of parallel statements has finished.

```
//Verilog Code
module myFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] a;
reg [31:0] b;
reg [31:0] c;
reg [1:0] functionCounter;

//flow block
always@(posedge clock) begin
      if(reset) begin
            a <= 0;
            b <= 0;
            c <= 0;
            output1 <= 0;
      end
      else if(start) begin
            //Parallel Statement Set 1
            a <= 2 + b;
            b <= 3 - a + c;
            c <= 3*b;
      end
      else if(functionCounter != 0) begin
            case(functionCounter)
                  //Parallel Statement Set 2
                  1: a <= b - 3;
                  b <= a * 2;
                  c <= a + b;
            endcase
      end
      else begin
            a <= a;
            b <= b;
            c <= c;
            output1 <= output1;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
            functionCounter <= 0;
      end
      else if(start) begin
            done <= 0;
            functionCounter <= 1;
      end
```

```
      else if(functionCounter != 0) begin
            case(functionCounter)
                  1: functionCounter <= 0;
                  done <= 1;
                  default: functionCounter <= functionCounter + 1;
            endcase
      end
      else begin
            functionCounter <= 0;
            done <= done;
      end
end
endmodule
```

Figure 34.
Translating Cv to Verilog for a function containing two sets of parallel statements in series.

In the Figure 34 there are two sets of parallel statements. Parallel Statement Set 1 is executed when `start` is high, and Parallel Statement Set 2 is executed when the `functionCounter` is 1. This means that the function takes two clock cycles to execute. Once the `functionCounter` is 1, the function has completed, and the `functionCounter` is set back to 0, and `done` is set high.

2.11 Non-Blocking Statements

In hardware design, *non-blocking* statements are used to assign values to variables in parallel at the edge of an event. For example:

```
//Verilog Code
always@(posedge clock) begin
      a <= b + 2;
      b <= a * 2;
end
```

Figure 35.
Verilog code using non-blocking statements.

In this example the values of `a` and `b` are set in parallel. The right hand expressions are evaluated using the previous values of `a` and `b`, and `a` and `b` are not assigned new values until both right hand expressions are evaluated. This same semantic is used in Cv. In Cv, all non-blocking

statements are collected and evaluated prior to the rest of the function. Also, in Cv, it is illegal to use the values of variables that are assigned with non-blocking statements, in assignments that are not non-blocking statements. According to the Cv grammar:

AssignExpr ::= LValue = Expr | **(ident** +, **)** = Call | LValue <> NonCallExpr

The non-blocking assignment in Cv is the assignment that uses the <> operator.

Example

```
//Cv Code
int output1 MyFunction(int input1){
      int a;
      int b;
      int c;

      a <> input1*2;
      b = input1 - 3;
      c <> input1 - 4;
      output1 <> c + 3;
}
```

Figure 36.
Cv code containing non-blocking statements.

In this example there are four assignments. Variables `a`, `c`, and `output1` are assigned using non-blocking assignments, and the variable `b` is assigned using a normal assignment. The non-blocking assignments will be gathered and executed in parallel. Once the non-blocking statements have been executed, the remaining code, in this case the assignment to variable `b`, will be executed.

Cv Software Translator

```
//C++ Code
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvFunctionStruct_MyFunction returnValue;
	CvV<int> a;
	CvV<int> b;
	CvV<int> c;

	//non-blocking assignment
	a ^ input1 * 2;
	c ^ input1 - 4;
	returnValue.output1 ^ c + 3;
	a.AfterParallelUpdate( );
	c.AfterParallelUpdate( );
	returnValue.output1.AfterParallelUpdate( );

	//serial statement
	b = input1 - 3;

	return returnValue;
}
```

Figure 37.
Translating Cv to C++ for a function containing non-blocking statements.

In Figure 37 all of the non-blocking statements are collected and executed first. The non-blocking assignment are executed using the caret operator. This operator evaluates right hand and stores the value in the `_writeValue` member of the CvV class (Section 2.8), leaving the `_value` member of the CvV class available to be read without change. Once all assignments are done using the caret operator, `a`, `c`, and `returnValue.output1` each call the AfterParallelUpdate function from the CvV class to copy the `_writeValue` member over to the `_value` member. Evaluating non-blocking statements in software is identical to evaluating parallel statements (Section 2.10). The difference is that all non-blocking statements are executed before the rest of the code.

Cv Hardware Translator

```
//Verilog Code
module myFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] a;
reg [31:0] b;
reg [31:0] c;

//non-blocking block
always@(posedge clock) begin
      if(reset) begin
            a <= 0;
            c <= 0;
            output1 <= 0;
      end
      else if(start) begin
            a <= input1 * 2;
            c <= input1 - 4;
            output1 <= c + 3;
      end
      else begin
            a <= a;
            c <= c;
            output1 <= output1;
      end
end

//flow block
always@(posedge clock) begin
      if(reset) begin
            b = 0;
      end
      else if(start) begin
            b = input1 - 3;
      end
      else begin
            b = b;
      end
end

always@(posedge clock) begin
      if(reset) begin
            done <= 0;
      end
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      end
end
endmodule
```

Figure 38.
Translating Cv to Verilog for a function containing non-blocking statements.

Non-blocking statements are native to Verilog, and therefore the Cv Hardware Translator uses the Verilog non-blocking semantic. A third always block is made for non-blocking statements, and all non-blocking assignments are placed in the new always block. Non-blocking statements never force a function to require CHard.

2.12 Event Driven Statements

According to the grammar:

EventDriven ::= LValue **~** EventDrivenExpr **;**

The LValue is either a variable or a computation accessing an element of an array. The tilde operator (~) declares that this is an *event driven* assignment. In an event driven assignment, anytime one of the parameters on the right side of the tilde changes, the LValue on the left is automatically updated. For example, if `a` and `b` are both integers, and we have the statement:

```
a ~ b + 4;
```

then if `b` is equal to 6, then `a` is equal to 10. If something changes the value of `b` to 8, then `a` is automatically changed to 12. The operations that are legal for the right side of an event driven statement are: addition, subtraction, multiplication, division, modulus, negation, less than, greater than, less than or equal to, greater than or equal to, equal to, not equal to, and, or, left shift, and right shift. The only expression that is not allowed to be used in an event driven statement is a function call.

2.12.1 Example

```
//Cv Code
int a MyFunction (int b, int c){
      a ~ b + c;
}
```

Figure 39.
Cv code containing event driven statements.

In this example a is event driven assigned to b + c. Any time that b or c change, a will automatically be updated.

2.12.2 Cv Software Translator

The event driven semantic is accomplished in software by the CvC and CvV classes. Within the CvC class there are 2 members:

```
//C++ code
vector<CvC*> _dependors;
vector<CvC*> _dependencies;
```

Figure 40.
Private members of the CvC class in C++. These members are used to implement the event driven construct in software.

Because each variable in Cv is translated into a CvV, which is inherited from the CvC class, each variable has dependors and dependencies. Given the event driven example:

```
//Cv Code
a ~ b + c;
```

Figure 41.
An event driven statement in Cv.

For variable a, variables b and c would be added to a's dependencies vector. For variables b and c, a would be added to their dependors vector. Every CvC, and therefore CvV, also has a member of that class that is a pointer to a function that takes a pointer to a CvC. For discussion this function will be called the variable's Update Function. The Update Function is what updates the left hand variable when one of the right hand variables is changed.

Next, the overridden = operator is discussed. When a CvV variable is assigned a new value each of its dependors call the dependor's Update Function. This is what enables a change to variables b and c to automatically update variable a. Once b or c has changed, a, which is one of b and c's dependors, will call its Update Function. Variable a's Update Function is created when the Cv Software Translator is compiling the event driven statement. Each event driven statement creates a unique function. For this example, the function would be:

```
//C++ Code
void CvEventFunction_1(CvC* cvc) {
      Cvc* b = cvc->GetDependencies( )[0];
      Cvc* c = cvc->GetDependencies( )[1];
      cvc->SetValue(b->Me<int>( ) + c ->Me<int>( ));
}
```

Figure 42.
C++ functions generated to execute the event driven statement.

Because this is the Update Function for the variable a, when CvEventFunction_1 is called, the CvC* argument will point at variable a. When a calls its Update Function (CvEventFunction_1), variables b and c will get selected out of a's dependencies, added together, and set as a's _value using SetValue;

The example function would be translated as follows:

```
//C++ Code
void CvEventFunction_1(CvC* cvc) {
      Cvc* b = cvc->GetDependencies( )[0];
      Cvc* c = cvc->GetDependencies( )[1];
      cvc->SetValue(b->Me<int>( ) + c ->Me<int>( ));
}
struct CvFunctionStruct_MyFunction {
      CvV<int> a;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> b, CvV<int> c) {
      CvFunctionStruct_MyFunction returnValue;
      returnValue.a.AddDependency(b);
      returnValue.a.AddDependency(c);
      b.AddDependor(a);
      c.AddDependor(a);
      a.AddFunction(CvEventFunction_1);

      return returnValue;
}
```

Figure 43.
Translating Cv to C++ for event driven statements.

In Figure 43, `a` adds `b` and `c` as its dependors. Then `b` and `c` each add `a` as a dependency. This is how different variables set their dependors and dependencies.

2.12.3 Cv Hardware Translator

Verilog has built in event driven semantics using the *assign* operator. Therefore this is what Cv will use to translate the Cv code.

```
//Verilog Code
module MyFunction(input clock, input reset, input start, input [31:0] b,
      input [31:0] c, output reg done, output wire [31:0] a);

assign a = b + c;
endmodule
```

Figure 44.
Translating Cv to Verilog for event driven statements.

The above code uses the `assign` statement in Verilog to implement the event driven semantic in Cv. Although outputs are normally `reg` variables after being translated from Cv, `a` has been changed to a `wire`. This is because Verilog requires that the left hand side of an `assign` statement is a `wire`. The Cv compiler knows how Verilog types need to be declared in order to be comply with Verilog rules.

2.13 Function Calls

Function calls are integral to software languages, and are closely related to module instantiations in Verilog. Because the two are closely related, module instantiations can be used in hardware to implement function calls. According to the grammar:

Call ::= **ident (** Actuals **).ident** | **ident (** Actuals **)**

AssignExpr ::= LValue **=** Expr | **(ident** +, **) =** Call | LValue <> Expr

The first definition (**ident**(Actuals).**ident**) is used to target individual outputs of a function, while the second definition (**ident**(Actuals))is used to set multiple variables to the output of the function call. To set multiple variables to the outputs of a function call, the syntax for the second definition of the AssignExpr (**(ident** +, **) =** Call) is used.

The Example section of Section 2.13 has been omitted. Because there are many special cases for the Cv Hardware Translator that differ from the Cv Software Translator, all examples are presented in Section 2.13.1 (Cv Software Translator) and Section 2.13.2 (Cv Hardware Translator).

2.13.1 Cv Software Translator

When translating Cv function calls to C++, the two look very similar. There are two cases to consider: targeting an individual output of a function, and setting multiple outputs of a function.

2.13.1.1 Targeting an Individual Output

Each functions is translated to return a struct, and elements of a struct are accessed using the dot notation. Because of this, the C++ syntax to access a value within a struct that has been returned matches the Cv syntax for accessing an output of a function.

Example:

```
//Cv Code
int a FunctionToBeCalled(int y){
      a = y * 2;
}
int output1 MyFunction(int input1){
      output1 = FunctionToBeCalled(input1).a + 3;
}
```

Figure 45.
Cv code containing a function call.

In Figure 45, `output1` is being assigned the value of the output `a` from `FunctionToBeCalled` plus 3. To get the value of a specific output from a function call, the function call is followed by "." and the output variable's name. In Figure 45, the output variable `a` from `FunctionToBeCalled` is selected using this method.

Translated into C++:

```
//C++ Code
struct CvFunctionStruct_FunctionToBeCalled {
	CvV<int> a;
};
CvFunctionStruct_FunctionToBeCalled FunctionToBeCalled(CvV<int> y) {
	CvFunctionStruct_FunctionToBeCalled returnValue;
	returnValue.a = y * 2;
	return returnValue;
}
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvFunctionStruct_MyFunction returnValue;
	returnValue.output1 = FunctionToBeCalled(input1).a + 3;
	return returnValue;
}
```

Figure 46.
Translating Cv to C++ for code containing a function call.

As demonstrated above, the function call looks very similar to the Cv code. The function call is made and is provided `input1` as the argument to `FunctionToBeCalled`. Because `FunctionToBeCalled` returns a struct, containing a `CvV<int> a`, which is the desired output, this can select this using `.a` syntax.

2.13.1.2 Setting Multiple Outputs

To set multiple outputs of a function, the function call cannot be an inline call and must be a separate assignment. The following syntax is an example of assigning multiple outputs of a function to variables.

```
//Cv Code
(a, b, c) = CallSomeFunction(x);
```

Figure 47.
An example of Cv code calling a function and setting multiple outputs.

Here `CallSomeFunction` returns three variables, whose types would match `a`, `b`, and `c`. The variable `a` would be assigned to the first output of `CallSomeFunction`, `b` would be assigned to the second output, and `c` would be assigned to the third. The number of variables being assigned must match the number of outputs of the function.

Example:

```
//Cv Code
int a, int b FunctionToBeCalled(int y){
	a = y * 2;
	b = y - 3;
}
int output1 MyFunction(int input1){
	int c;
	int d;

	(c, d) = FunctionToBeCalled(input1);
	output1 = c + d;
}
```

Figure 48.
Cv code calling a function and setting multiple outputs.

Translated into C++:

```
//C++ Code
struct CvFunctionStruct_FunctionToBeCalled {
	CvV<int> a;
	CvV<int> b;
};
CvFunctionStruct_FunctionToBeCalled FunctionToBeCalled( CvV<int> y ) {
	CvFunctionStruct_FunctionToBeCalled returnValue;
	returnValue.a = y * 2;
	returnValue.b = y - 3;
	return returnValue;
}
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction( CvV<int> input1 ) {
	CvFunctionStruct_MyFunction returnValue;
	CvV<int> c;
	CvV<int> d;

	CvFunctionStruct_FunctionToBeCalled FunctionToBeCalled_1 =
```

```
                FunctionToBeCalled(input1);
        c = FunctionToBeCalled_1.a;
        d = FunctionToBeCalled_1.b;

        returnValue.output1 = c + d;
        return returnValue;
}
```

Figure 49.
Translating Cv to C++ for a function call that returns multiple outputs.

When translated into C++, a struct that the called function returns is created and set equal to the output of the called function. Then, the variables assigned to the outputs of the function are assigned in order to the variables inside the struct. This can be seen where `c` and `d` are assigned to variables `a` and `b` inside the `FunctionToBeCalled_1`.

2.13.2 Cv Hardware Translator

When translating Cv function calls into Verilog three scenarios exist: function calls that occur inline with other statements, function calls in series with other statements, and function calls that occur purely in parallel.

2.13.2.1 In Line Function Calls

It is very common in software to see function calls made inline as a part of an assignment. For example:

```
//C++ Code
int x = 3 + CallToFunction(x);
```

Figure 50.
Showing an example of a function call in C++.

Therefore, Cv also supports inline function calls. This section presents two examples. The first example uses one inline call. The second examples uses two inline calls in the same line.

Example 1:

```
//Cv Code
int a MyFunctionToBeCalled( int y ){
	a = y*2 + 3;
}
int output1 MyFunction( int input1 ){
	int myInt;
	myInt = 4;
	output1 = myInt + input1 + MyFunctionToBeCalled(input1).a;
}
```

Figure 51.
Cv code containing an inline function call.

Figure 51 provides two functions, where `MyFunction` calls `MyFunctionToBeCalled` in the assignment of the variable `x`. Because Cv allows functions to have more than one output, the call must attach the variable output from the called function, using the "." notation. This can be seen in Figure 51 by selecting output variable `a` from `MyFunctionToBeCalled` using the `.a` syntax after the function call.

Translated into Verilog:

```
//Verilog Code
module MyFunctionToBeCalled(input clock, input reset, input start,
	input [31:0] y, output reg done, output reg [31:0] a);

//flow block
always@(posedge clock) begin
	if(reset) begin
		a = 0;
	end
	else if(start) begin
		a = y*2 + 3;
```

```
	end
	else begin
		a = a;
	end
end

//control block
always@(posedge clock) begin
	if(reset) begin
		done <= 0;
	end
	else if(start) begin
		done <= 1;
	end
	else begin
		done <= done;
	end
end
endmodule

module MyFunction(input clock, input reset, input start,
	input [31:0] input1, output reg done, output reg [31:0] output1);

//instantiate the module to be called
reg startMyFunctionToBeCalled_1;
wire doneMyFunctionToBeCalled_1;
wire [31:0] myFunctionToBeCalled_1_a;
MyFunctionToBeCalled myFunctionToBeCalled_1(
	.clock(clock),
	.reset(reset),
	.start(startMyFunctionToBeCalled_1),
	.y(input1),
	.done(doneMyFunctionToBeCalled_1),
	.a(myFunctionToBeCalled_1_a)
);

reg [1:0] functionCounter;
reg [31:0] myInt;

//flow block
always@(posedge clock) begin
	if(reset) begin
		myInt <= 0;
		startMyFunctionToBeCalled_1 <= 0;
		output <= 0;
	end
	else if(start) begin
		myInt <= 4;
	end
	else if(functionCounter != 0) begin
		case(functionCounter)
			1: startMyFunctionToBeCalled_1 <= 1;
			2: startMyFunctionToBeCalled_1 <= 0;
			3: output1 <= myInt + input1 + MyFunctionToBeCalled_1_a;
		endcase
	end
	else begin
		myInt <= myInt;
		startMyFunctionToBeCalled_1 <= startMyFunctionToBeCalled_1;
		output <= output;
	end
end
```

```
//control block
always@(posedge clock) begin
      if(reset) begin
            functionCounter <= 0;
            done <= 0;
      end
      else if(start) begin
            functionCounter <= 1;
            done <= 0;
      end
      else if(functionCounter != 0) begin
            2:
            if(doneMyFunctionToBeCalled_1) begin
                  functionCounter <= 3;
            end
            else begin
                  functionCounter <= 2;
            end
            3: functionCounter <= 0;
            done <= 1;
            default: functionCounter <= functionCounter + 1;
      end
      else begin
            functionCounter <= 0;
            done <= done;
      end
end
endmodule
```

Figure 52.
Translating Cv to Verilog for an inline function call.

In Figure 52, first `MyFunctionToBeCalled` is translated into a module. `MyFunctionToBeCalled` does not require CHard, as can be seen by the lack of a `functionCounter` in `MyFunctionToBeCalled` . Then `MyFunction` is translated into a module. While translating `MyFunction` the Cv compiler recognizes that there is an inline function call in `MyFunction`. The module for `MyFunctionToBeCalled` is instantiated and indexed. This can be seen in Figure 52 where the instantiation for `MyFunctionToBeCalled` is `myFunctionToBeCalled_1`, where 1 is the index. Indexing instantiations allows for multiple function calls to be made to the same function, and each instantiation will receive a different index. Because `MyFunction` uses an inline function call, `MyFunction` will automatically require CHard. The result of `MyFunctionToBeCalled` must have finished before `MyFunction` can use the result of

MyFunctionToBeCalled . Therefore, in one clock cycle the start signal for MyFunctionToBeCalled will be raised high. The next clock cycle the start signal for MyFunctionToBeCalled is set low. This can be seen in Figure 52 when startMyFunctionToBeCalled_1 is set high on the clock cycle when the functionCounter is 1, and set low when the functionCounter is 2. Next, the control block does not allow the function to proceed until the done signal from MyFunctionToBeCalled is high. This can be seen in Figure 52 in the control block when the functionCounter is 2. If doneMyFunctionToBeCalled_1 is high then the functionCounter increments to 3, otherwise the functionCounter remains at 2, and the next clock cycle the function will check again to see if doneMyFunctionToBeCalled_1 is high. Once MyFunctionToBeCalled is done, the result is used in the next clock cycle in the computation. This is seen in Figure 52 in the flow block when the functionCounter is 3, output1 <= myInt + input1 + MyFunctionToBeCalled_1_a.

Example 2:

If multiple function calls are made in the same inline statement, then each function is started at the same time. The control block waits to see the done signal from each module started before proceeding.

```
//Cv Code
int a MyFunctionToBeCalled1(int y){
      a = y*2 + 3;
}
int b MyFunctionToBeCalled2(int z){
      b = z/5 + 4;
}
int output1 MyFunction(int input1){
      output1 =  MyFunctionToBeCalled1(input1).b +
MyFunctionToBeCalled2(input1).a;
}
```

Figure 53.

Cv code containing inline function calls being added together.

Translated into Verilog:

```
//Verilog Code
module MyFunctionToBeCalled1(input clock, input reset, input start,
      input [31:0] y, output reg done, output reg [31:0] a);

//flow block
always@(posedge clock) begin
      if(reset) begin
            a = 0;
      end
      else if(start) begin
            a = y*2 + 3;
      end
      else begin
            a = a;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
      end
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      end
end
endmodule

module MyFunctionToBeCalled2(input clock, input reset, input start,
      input [31:0] z, output reg done, output reg [31:0] b);

//flow block
always@(posedge clock) begin
      if(reset) begin
            b = 0;
      end
      else if(start) begin
            b = z/5 +4;
      end
      else begin
            b = b;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
      end
      else if(start) begin
```

```
            done <= 1;
      end
      else begin
            done <= done;
      end
end
endmodule

module MyFunction(input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

//instantiate the module to be called
reg startMyFunctionToBeCalled1_1;
wire doneMyFunctionToBeCalled1_1;
wire [31:0] MyFunctionToBeCalled1_1_a;
reg startMyFunctionToBeCalled2_1;
wire doneMyFunctionToBeCalled2_1;
wire [31:0] myFunctionToBeCalled2_1_b;
MyFunctionToBeCalled MyFunctionToBeCalled1_1(
      .clock(clock),
      .reset(reset),
      .start(startMyFunctionToBeCalled1_1),
      .y(input1),
      .done(doneMyFunctionToBeCalled1_1),
      .a(myFunctionToBeCalled1_1_a)
);
MyFunctionToBeCalled myFunctionToBeCalled2_1(
      .clock(clock),
      .reset(reset),
      .start(startMyFunctionToBeCalled2_1),
      .z(input1),
      .done(doneMyFunctionToBeCalled2_1),
      .b(myFunctionToBeCalled2_1_b)
);

reg [1:0] functionCounter;

//flow block
always@(posedge clock) begin
      if(reset) begin
            myInt <= 0;
            startMyFunctionToBeCalled1_1 <= 0;
            startMyFunctionToBeCalled2_1 <= 0;
            output <= 0;
      end
      else if(start) begin
            startMyFunctionToBeCalled1_1 <= 1;
            startMyFunctionToBeCalled2_1 <= 1;
      end
      else if(functionCounter != 0) begin
            case(functionCounter)
                  1: startMyFunctionToBeCalled1_1 <= 0;
                  startMyFunctionToBeCalled2_1 <= 0;
                  2: output1 <= MyFunctionToBeCalled1_1_a +
                                          MyFunctionToBeCalled2_1_b;

            endcase
      end
      else begin
            startMyFunctionToBeCalled1_1 <= startMyFunctionToBeCalled1_1;
            startMyFunctionToBeCalled2_1 <= startMyFunctionToBeCalled2_1;
            output <= output;
```

```
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            functionCounter <= 0;
            done <= 0;
      end
      else if(start) begin
            functionCounter <= 1;
            done <= 0;
      end
      else if(functionCounter != 0) begin
            1:
            if(doneMyFunctionToBeCalled1_1 & doneMyFunctionToBeCalled2_1)
            begin
                  functionCounter <= 2;
            end
            else begin
                  functionCounter <= 1;
            end
            2: functionCounter <= 0;
            done <= 1;
            default: functionCounter <= functionCounter + 1;
      end
      else begin
            functionCounter <= 0;
            done <= done;
      end
end
endmodule
```

Figure 54.
Translating Cv to Verilog for code containing 2 inline functions being added together.

In Figure 54, in `MyFunction` there are two function calls made within the same line. When `MyFunction` is translated into Verilog `startMyFunctionToBeCalled1_1` and `startMyFunctionToBeCalled2_1` are both raised high during the same clock cycle. This can be seen in Figure 54 when `startMyFunctionToBeCalled1_1` and `startMyFunctionToBeCalled2_1` are set high when the `start` is high. In addition to this, the control block does not continue with execution until both `doneMyFunctionToBeCalled1_1` and `doneMyFunctionToBeCalled2_1` are both high. This can be seen in Figure 54 in the control block of `MyFunction` when the `functionCounter` is 1. This allows for functions to be called that require different numbers of clock cycles.

2.13.2.2 Function Calls in Series

This section covers function calls that are in series with other statements, but are not made inline with other computations.

Example:

```
//Cv Code
int a MyFunctionToBeCalled(int y){
	a = y*2 + 3;
}
int output1 MyFunction(int input1){
	int myInt;
	myInt = 4; //line 1
	output1 = MyFunctionToBeCalled(input1).a; //line 2
	output1 = myInt + input1 + output1; //line 3
}
```

Figure 55.
Cv code containing a function call in series with other statements.

Translated into Verilog:

```
//Verilog Code
module MyFunctionToBeCalled(input clock, input reset, input start,
	input [31:0] y, output reg done, output reg [31:0] a);

//flow block
always@(posedge clock) begin
	if(reset) begin
		a = 0;
	end
	else if(start) begin
		a = y*2 + 3;
	end
	else begin
		a = a;
	end
end

//control block
always@(posedge clock) begin
	if(reset) begin
		done <= 0;
	end
```

```
	else if(start) begin
		done <= 1;
	end
	else begin
		done <= done;
	end
end
endmodule

module MyFunction(input clock, input reset, input start,
	input [31:0] input1, output reg done, output reg [31:0] output1);

//instantiate the module to be called
reg startMyFunctionToBeCalled_1;
wire doneMyFunctionToBeCalled_1;
wire [31:0] myFunctionToBeCalled_1_a;
MyFunctionToBeCalled myFunctionToBeCalled_1(
	.clock(clock),
	.reset(reset),
	.start(startMyFunctionToBeCalled_1),
	.y(input1),
	.done(doneMyFunctionToBeCalled_1),
	.a(myFunctionToBeCalled_1_a)
);
reg [2:0] functionCounter;
reg [31:0] myInt;

//flow block
always@(posedge clock) begin
	if(reset) begin
		myInt <= 0;
		startMyFunctionToBeCalled_1 <= 0;
		output <= 0;
	end
	else if(start) begin
		myInt <= 4;
	end
	else if(functionCounter != 0) begin
		case(functionCounter)
			1: startMyFunctionToBeCalled_1 <= 1;
			2: startMyFunctionToBeCalled_1 <= 0;
			3: output1 <= MyFunctionToBeCalled_1_a;
			4: output1 <= myInt + input1 + output1;
		endcase
	end
	else begin
		myInt <= myInt;
		startMyFunctionToBeCalled_1 <= startMyFunctionToBeCalled_1;
		output <= output;
	end
end

//control block
always@(posedge clock) begin
	if(reset) begin
		functionCounter <= 0;
		done <= 0;
	end
	else if(start) begin
		functionCounter <= 1;
		done <= 0;
	end
```

```
      else if(functionCounter != 0) begin
            2:
            if(doneMyFunctionToBeCalled_1) begin
                  functionCounter <= 3;
            end
            else begin
                  functionCounter <= 2;
            end
            4: functionCounter <= 0;
            done <= 1;
            default: functionCounter <= functionCounter + 1;
      end
      else begin
            functionCounter <= 0;
            done <= done;
      end
end
endmodule
```

Figure 56.
Translating Cv to Verilog for a function call in series with other statements.

This example is very similar to the example in Section 2.13.2.1. In Figure 55 the function call in line 2 is made by itself (not inline), however it is in series with the line 1 and line 3. Therefore, the function call must be made after line 1, but before line 3. This will require the Cv compiler to use CHard. The only difference in the translated Verilog is in the flow block when the `functionCounter` is 3. Here `output1` is set to the result of `MyFunctionToBeCalled`. Then, when the `functionCounter` is 4, the addition is performed. This is an accordance to how the code was written in Cv.

2.13.2.3 Function Calls in Parallel

In hardware, or Verilog, when modules are instantiated within one another, they operate in parallel. Therefore, in Cv there is a way to call functions such that the functions behave in the same manner that is traditional to hardware developers. When function calls are made in parallel then they can be translated without creating CHard.

Example:

```
//Cv Code
int a MyFunctionToBeCalled(int y){
      a = y*2 + 3;
}
int output1 MyFunction(int input1){
      int outputOfFunction;
      outputOfFunction = MyFunctionToBeCalled(input1).a;
      || output1 = input1 + outputOfFunction;
}
```

Figure 57.
Cv code calling a function in parallel with other statements.

Translated into Verilog:

```
//Verilog Code
module MyFunctionToBeCalled(input clock, input reset, input start,
      input [31:0] y, output reg done, output reg [31:0] a);

//flow block
always@(posedge clock) begin
      if(reset) begin
            a = 0;
      end
      else if(start) begin
            a = y*2 + 3;
      end
      else begin
            a = a;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
      end
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      end
end
endmodule

module MyFunction(input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

//instantiate the module to be called
wire doneMyFunctionToBeCalled_1;
```

```
wire [31:0] outputOfFunction;
MyFunctionToBeCalled myFunctionToBeCalled_1(
      .clock(clock),
      .reset(reset),
      .start(start),
      .y(input1),
      .done(doneMyFunctionToBeCalled_1),
      .a(outputOfFunction)
);

//flow block
always@(posedge clock) begin
      if(reset) begin
            output1 = 0;
      end
      else if(start) begin
            output1 = input1 + outputOfFunction;
      end
      else begin
            output1 = output1;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
      end
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      end
end
```

Figure 58.
Translating Cv to Verilog for code containing function calls in parallel with other code.

In the example above (Figure 58) there are several things different than the examples before Fig 58 examples. In this discussion `MyFunction` is the focus. `MyFunctionToBeCalled` is identical to the previous examples in the section. The `start` input for the instantiation of `MyFunctionToBeCalled`, is the same `start` input to `MyFunction`. This is because `MyFunctionToBeCalled` needs to start at the same time as the code in MyFunction. Also, in the Cv code in Figure 57, `outputOfFunction` is assigned to `MyFunctionToBeCalled(input1).a`. Therefore in this situation `outputOfFunction` is defined as a `wire`, and is the output `a` in

`MyFunctionToBeCalled`. It can also be seen in the Figure 58 that CHard is not created. Using this style of design, hardware developers can create hardware similarly to methods used today.

2.14 If Statement

If statements are conditional expressions that execute a desired branch based on a given expression. According to the grammar:

IfStmt ::= **if (** Expr **)** Stmt < **else** Stmt >

The if statement is the keyword *if*, followed by *a test* expression to evaluate in parenthesis, and then a statement (or a statement block) to execute if the test expression is true. Optionally an *else* condition can follow. It is common to see the else followed by another if statement, making a chain of if, else if, else if, etc.

The example section of if statement has been omitted. Because there are many special cases for the Cv Hardware Translator that differ from the Cv Software Translator, all examples are presented in the Cv Software Translator (Section 2.14.1) and Cv Hardware Translator (Section 2.14.2).

2.14.1 Cv Software Translator

Like other constructs, the translated C++ code highly resembles the Cv code. The test condition must be manipulated so that it return a C++ bool (Boolean value), and not a CvV<bool>.

Example:

```
//Cv Code
int output1 MyFunction(int input1) {
	int a;

	if(input1 == 2)
		a = 3;
	else if(input1 == 3) {
		a = 4;
		a = 5 * a;
	}
	else
		a = 6;
	output1 = a * 9;
}
```

Figure 59.
Cv code containing if statements.

In Figure 59, the `input1` is tested for values 2 and 3. The integer `a` is set to a certain value depending on the value of `input1`. `Output1` then uses `a` to calculate its value.

Translated into C++:

```
//C++ Code
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvFunctionStruct_MyFunction returnValue;
	CvV<int> a;

	if(CvV<bool>(input1 == 2).Me( )) {
		a = 3;
	}
	else if(CvV<bool>(input1 == 3).Me( )) {
		a = 4;
		a = 5 * a;
	}
	else {
		a = 6;
	}

	returnValue.output1 = a * 9;

	return returnValue;
}
```

Figure 60.

Translating Cv to C++ for code containing if statements.

In the translation it can be seen that the structure of the if statements is almost identical to the Cv code. In the test of each if statement the expression is wrapped by, `CvV<bool>(` test expression `).Me( )`. This is because comparing a CvV variable (Section 2.8) returns another CvV variable, instead of a C++ bool. C++ requires that the test expression of an if statement is a C++ bool. Therefore, first the expression is transformed into a `CvV<bool>`, and second the `Me` function is accessed. The `Me` function for any CvV variable returns the `_value` member. Because the variable has set the template type to bool, the `Me` function will return the Boolean result of the test expression.

2.14.2 Cv Hardware Translator

The Cv Hardware Translator must be concerned with the following scenarios given an if statement: If statements that require CHard, if statements that don't require CHard, if statements containing non-blocking statements, and when the if statement contains a call in the test expression. These 4 scenarios are presented below with examples.

2.14.2.1 If Statement Does not Require CHard

The first scenario to consider is when the function does not require CHard.

Example:

```
//Cv Code
int output1 MyFunction (int input1) {
      int myInt;
```

```
	if(input1 == 2)
		myInt = 6;
	else if(input1 == 3)
		myInt = 10;
	else
		myInt = 3;
	output1 = 4*myInt;
}
```

Figure 61.
Cv code containing if statement that do not require CHard.

This code uses if statements in series with other assignments. However, there are no loops involved, so the code does not require CHard.

Translating to Verilog:

```
//Verilog Code
module myFunction (input clock, input reset, input start,
	input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] myInt;

//flow block
always@(posedge clock) begin
	if(reset) begin
		myInt = 0;
	end
	else if(start) begin
		if(input1 == 2) begin
			myInt = 6;
		end
		else if(input1 == 3) begin
			myInt = 10;
		end
		else begin
			myInt = 3;
		end
		output1 = 4*myInt;
	end
	else begin
		myInt = myInt;
		output1 = output1;
	end
end

//control block
always@(posedge clock) begin
	if(reset)
		done <= 0;
	end
```

```
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      done
end

endmodule
```

Figure 62.
Translating Cv to Verilog for code containing if statements that do not require CHard.

`MyFunction` does not require CHard. As stated in the CHard section, all of the code to be implemented from the user is placed in the flow block, and is executed when `start` is high. It can be seen how there is a direct translation between the code Figure 61, and the resulting Verilog in Figure 62.

2.14.2.2 If Statement Requires CHard

Next an if statement is presented that requires CHard. In this if statement a function call is used.

Example:

```
//Cv Code
int a CalledFunction(int y)
{
        //some code
}

int output1 MyFunction ( int input1 ) {
      int myInt;
      if(input1 == 2)
            myInt = 6;
      else if(input1 == 3)
            myInt = CalledFunction(input1).a;
      else
            myInt = 3;
      output1 = 4*myInt;
}
```

Figure 63.

Cv code containing if statements. Because one of the if statements contains a function call, the code requires CHard.

Because `CalledFunction` occurs in the if statement, and is in series with the last statement in the function, this function requires CHard.

Translated into Verilog:

```
//Verilog Code
module MyFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [3:0] functionCounter;
reg [31:0] myInt;
reg startCalledFunction_1;
wire doneCalledFunction_1;
wire [31:0] calledFunction_1_a;
CalledFunction calledFunction_1(
      .clock(clock),
      .reset(reset),
      .start(startCalledFunction_1),
      .y(input1),
      .done(doneCalledFunction_1),
      .a(calledFunction_1_a)
);

//flow block
always@(posedge clock) begin
      if(reset) begin
            myInt <= 0;
            startCalledFunction_1 <= 0;
            output1 <= 0;
      end
      else if(start) begin
            //nothing will happen here, control block is deciding which
            //branch to take
      end
      else if(functionCounter != 0) begin
            case(functionCounter)
                  1: myInt <= 6;
                  2: startCalledFunction_1 <= 1;
                  3: startCalledFunction_1 <= 0;
                  4: myInt <= calledFunction_1_a;
                  5: myInt <= 3;
                  6: output1 <= myInt*4;
            endcase
      end
      else begin
            myInt <= myInt;
            startCalledFunction_1 <= startCalledFunction_1;
            output1 <= output1;
```

```
	end
end

//control block
always@(posedge clock) begin
	if(reset) begin
		done <= 0;
		functionCounter <= 0;
	end
	else if(start) begin
		done <= 0;
		if(input1 == 2) begin
			functionCounter <= 1;
		end
		else if(input1 == 3)
			functionCounter <= 2;
		end
		else begin
			functionCounter <= 5;
		end
	end
	else if(functionCounter != 0) begin
		case(functionCounter)
			1: functionCounter <= 6;
			3: if(doneCalledFunction_1) begin
				functionCounter <= 4;
			end
			else begin
				functionCounter <= 3;
			end
			4: functionCounter <= 6;
			5: functionCounter <= 6;
			6: functionCounter <= 0;
			done <= 1;
		endcase
	end
	else begin
		functionCounter <= 0;
		done <= done;
	end
end
```

Figure 64.
Translating Cv to Verilog for if statements containing code that requires CHard.

In Figure 64 attention should be paid to how instructions are placed within the flow block, as well as the jumps made in the control block. In the flow block, when `start` is high the flow block does nothing. This is because when a function requires CHard the `if` Statement uses one clock cycle to determine which branch of the `if` Statement to take. After this, when the `functionCounter` is not equal to 0, all of the instructions within all `if` and `else if` blocks are laid out sequentially. This is because the control block will when instructions get executed.

Because there is a function call, `startCalledFunction_1` must be raised high, and then `MyFunction` must wait until `doneCalledFunction_1` is high to continue. In the control block it is shown how the branches are chosen. First, when `start` is high, the control block evaluates the if statement, and determines the appropriate branch to take. If `input1` is equal to 2, then the `functionCounter` is assigned 1. Else, if `input1` is 3, then the `functionCounter` is assigned 2. If neither of these two test expressions are true, then the `functionCounter` is assigned 5. By assigning the `functionCounter` to 1, the instruction within the first `if` statement is executed. Else, if the `functionCounter` is assigned to 2, then the code from the `else if` statement is executed. In this branch, the function call is made by raising `startCalledFunction_1`. `MyFunction` waits until `doneCalledFunction_1` if high to continue. If the functionCounter is assigned to 5, then the code for the `else` block is executed. When any of the branches are done executing, the `functionCounter` is assigned to 6, which is the first statement after the end of the if statement. This can be seen in Figure 64 when the `functionCounter` is 1, 4, or 5, the `functionCounter` is assigned to the value 6. This will prevent `MyFunction` from executing statements in other branches of the if statement.

2.14.2.3 Non-Blocking Statements

It is common in hardware design to use if statements containing non-blocking statements (Section 2.11). Therefore, in Cv this same design philosophy is supported. Normally all non-blocking assignments are extracted and placed together in an always block. However, with if statements, the entire if statement must be extracted and placed in the non-blocking block.

Example:

```
//Cv Code
int output1 MyFunction(int input1) {
	int myInt;

	if(input1 == 2)
		myInt <> 3;
	else
		myInt <> 4;
	output1 <> myInt*2;
}
```

Figure 65.
Cv code containing if statements that contain non-blocking statements.

Translated into Verilog:

```
//Verilog Code
module myFunction (input clock, input reset, input start,
	input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] myInt;
//non-blocking block
always@(posedge clock) begin
	if(reset) begin
		myInt <= 0;
		output1 <= 0;
	end
	else if(start) begin
		if(input1 == 2) begin
			myInt <= 3;
		end
		else begin
			myInt <= 4;
		end
		output1 <= myInt*2;
	end
	else begin
		myInt <= myInt;
		output1 <= output1;
	end
end

//control block
always@(posedge clock) begin
	if(reset) begin
		done <= 0;
	end
	else if(start) begin
		done <= 1;
	end
	else begin
		done <= done;
	end
end
```

```
endmodule
```

Figure 66.
Translating Cv to Verilog for code containing if statements that contain non-blocking statements.

In Figure 66 the entire if statement is extracted and placed in the non-blocking block. There is no code to put in the flow block, so the flow block is omitted. Because this function only contains non-blocking statement, it does not require CHard, and therefore does not depend on a `functionCounter`.

2.14.2.4 Calls Within If Statement Test Expression

The next situation that must be considered is when there is a function call within the test expression of the if statement.

Example:

```
//Cv Code
int a CalledFunction1(int y)
{
        //some code
}

int b CalledFunction2(int z)
{
        //some code
}

int output1 MyFunction(int input1) {
      int myInt;

      if(CallFunction1(input1).a == 2)
            myInt = 3;
      else if(CallFunction2(input1).b == 4)
            myInt = 4;
      else
            myInt = 5;
      output1 = myInt*2;
}
```

Figure 67.

Cv code containing if statements that use function calls in their test expressions.

In Figure 67 the `if` and `else if` Statements each contain a function call. Therefore, the values of the outputs of `CalledFunction1 and CalledFunction2` must be determined before the appropriate branch from the if statement can be selected. Function calls within the an if statement's test expression always require CHard.

Translated into Verilog:

```
//Verilog Code
module myFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [3:0] functionCounter;
reg [31:0] myInt;
reg startCalledFunction1_1;
wire doneCalledFunction1_1;
wire [31:0] calledFunction1_1_a;
reg startCalledFunction2_1;
wire doneCalledFunction2_1;
wire [31:0] calledFunction2_1_b;
CalledFunction1 calledFunction1_1(
      .clock(clock),
      .reset(reset),
      .start(startCalledFunction1_1),
      .y(input1),
      .done(doneCalledFunction1_1),
      .a(calledFunction1_1_a)
);
CalledFunction2 calledFunction2_1(
      .clock(clock),
      .reset(reset),
      .start(startCalledFunction2_1),
      .z(input1),
      .done(doneCalledFunction2_1),
      .b(calledFunction2_1_b)
);

//flow block
always@(posedge clock) begin
      if(reset) begin
            output1 <= 0;
            myInt <= 0;
            startCalledFunction1_1 <= 0;
            startCalledFunction2_1 <= 0;
      end
      else if(start) begin
            startCalledFunction1_1 <= 1;
```

```
            startCalledFunction2_1 <= 1;
      end
      else if(functionCounter != 0) begin
            case(functionCounter)
                  1: startCalledFunction1_1 <= 0;
                  startCalledFunction2_1 <=0;
                  2: myInt <= 3;
                  3: myInt <= 4;
                  4: myInt <= 5;
                  5: output1 = myInt*2;
            endcase
      end
      else begin
            output1 <= output1;
            myInt <= myInt;
            startCalledFunction1_1 <= startCalledFunction1_1;
            startCalledFunction2_1 <= startCalledFunction2_1;
      end
end
//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
            functionCounter <= 0;
      end
      else if(start) begin
            done <= 0;
            functionCounter <= 1;
      end
      else if(functionCounter != 0) begin
            case(functionCounter)
                  1: if(doneCalledFunction1_1 & doneCalledFunction2_1)
                  begin
                        if(calledFunction1_1_a == 2) begin
                              functionCounter <= 2;
                        end
                        else if(calledFunction2_1_b == 4) begin
                               functionCounter <= 3;
                        end
                        else begin
                              functionCounter <= 4;
                        end
                  end
                  else begin
                        functionCounter <= 1;
                  end
                  2: functionCounter <= 5;
                  3: functionCounter <= 5;
                  4: functionCounter <= 5;
                  5: functionCounter <= 0;
                  done <= 1;
                  default: functionCounter <= functionCounter + 1;
            endcase
      end
      else begin
            functionCounter <= 0;
            done <= done;
      end
end
endmodule
```

Figure 68.

Translating Cv to Verilog for code containing if statements that use function calls in their test expressions.

In the above translation the flow block initially starts `CalledFunction1` and `CalledFunction2`. The flow block then waits until `doneCalledFunction1_1` and `doneCalledFunction2_1` are high. In the control block, once `doneCalledFunction1_1` and `doneCalledFunction2_1` are high, the if statement determines which branch to take. If the first branch is taken then the `functionCounter` is assigned 2. If the second branch is taken the `functionCounter` is assigned 3. If the last branch is taken the `functionCounter` is assigned 4. Because there is only one statement per branch, the `functionCounter` is assigned 5 when the `functionCounter` equals 2, 3, or 4. This shows how the proper branch is selected, and when the selected branch has finished executing, the statement after the if statement is executed (`output1 = myInt*2`).

2.15 For Loop

For loops allow a section of code to be run multiple times. According to the grammar:

ForStmt ::= **for (** <Expr>; Expr; <Expr> **)** Stmt

For loops have four elements; initialize (*init*), *test* expression, *step*, and *statement*. The statement can be a single line statement or a statement block. Below Example 1 displays a for loop with a single line statement, and Example 2 displays a for loop with a statement block. In the initialize step, a loop index may be set to an initial value. The initialize step is not required. The test step is a Boolean expression that is evaluated each time the for loop is run. The first time the for loop is run, the initialize statement will be executed, followed by evaluating the test. The step is the

statement to be executed at the completion of the for loop. Once all of the computations within the for loop are completed, the step statement is executed. Then the test is reevaluated. If the test is true, then the code in the for loop will be run again. If the test if false, then the function will resume at the end of the for loop.

```
//Pseudo Code
//Example 1
for(init; test; step)
      single line statement

//Example 2
for(init; test; step) {
      statement block
}
```

Figure 69.
Examples of for loops using a single line statement with no braces, and a statement block contained in braces.

The example section of the for loop section has been omitted. Because there are many special cases for the Cv Hardware Translator that differ from the Cv Software Translator, all examples are presented in the Cv Software Translator and Cv Hardware Translator sections.

2.15.1 Cv Software Translator

The Cv Hardware Translator must focus on the difference between a static and dynamic for loop, (see Section 2.15.2), however the Cv Software Translator does not translate static and dynamic for loops differently.

Example:

```
//Cv Code
int output1 MyFunction(int input1){
	int i;

	for(i = 0; i < input1; i = i + 1){
		output1 = output1 + 2*i;
	}
}
```

Figure 70.
Cv code containing a for loop.

Translated into C++:

```
//C++ Code
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvFunctionStruct_MyFunction returnValue;
	CvV<int> i;

	for( i = 0; CvV<bool>(i < input1).Me( ); i = i + 1) {
		returnValue.output1 = returnVariable.output1 + 2 * i;
	}

	return returnValue;
}
```

Figure 71.
Translating Cv to C++ for code containing for loops.

In Figure 71 the for loop is created in C++, and the only syntactically change is in the test of the for loop. Similar to the test in the if statement (Section 2.14), the test expression is wrapped as a CvV<bool>, and then the `Me` function returns the C++ bool for the expression. The for loop will execute, setting the value of `output1` in the `returnValue` , and at the end of `MyFunction` , the `returnValue` will be returned.

2.15.2 Cv Hardware Translator

For hardware there are two cases to consider with for loops; dynamic and static for loops. Dynamic for loops do not have a set number of iterations, and can vary given the current operating parameters. Static for loops have a set number of iterations, and do not vary with the current operating parameters. Dynamic for loops require CHard, while static for loops can be translated into Verilog for loops, and do not require CHard. Figure 72 has an example of a dynamic for loop and an example of a static for loop.

```
//Pseudo Code
//Static For Loop
for(i = 0; i < 20; i = i + 1) {
      //code
}

//Dynamic For Loop
for(i = 0; i < input1; i = i + 1){
      //code
}
```

Figure 72.
Pseudo code showing the difference between the static and dynamic for loops. The static for loop goes from i = 0 to 19, and this can be determined at compile time. The dynamic for loop does not contain a guaranteed upper bound.

If the index of the loop is adjusted within the for loop, then the translation will require CHard (Section 2.15.2.1). The static for loop (Section 2.15.2.2) starts with `i` equal to 0, and loops until `i` is greater than or equal to 20. It can be determined at compile time that this for loop will run twenty times, and it will not require CHard. In the dynamic for loop (Section 2.15.2.3) the test is `i < input1`. At compile time it cannot be determined what the value of `input1` will be. Therefore this for loop is dynamic, and will require CHard.

2.15.2.1 Adjusting Index Within For Loop

```
//Cv Code
for(i = 0; i < 20; i = i + 1) {
      //code
      i = SomeFunction(i).output1;
      //code
}
```

Figure 73.
Cv code containing a for loop that would appear to be static. However, because the loop adjusts the index variable, this loop will be treated as dynamic and require CHard.

In Figure 73 the index variable `i` is altered within the function by setting `i` equal to `output1` of `SomeFunction`. This will require CHard. If elements of either the initialize, step, test ,or statement contain constructs that require CHard, such as function calls, then the otherwise static for loop will require CHard. Next, examples of static and dynamic for loops within functions are given.

2.15.2.2 Static For Loop

Example:

```
//Cv Code
int output1 MyFunctionStaticForLoop (int input1) {
      int i;

      output1 = 0;
      for(i  = 0; i < 10; i = i + 1)
            output1 = output1 + 1;
}
```

Figure 74.
Cv code containing a static for loop.

In this example a static for loop is used to iterate over the code ten times.

Translated into Verilog:

```
//Verilog Code
module MyFunctionStaticForLoop (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg [31:0] i;

//flow block
always@(posedge clock) begin
      if(reset) begin
            output1 = 0;
            i = 0;
      end
      else if(start) begin
            output1 = 0;
            for(i  = 0; i < 10; i = i + 1) begin
                  output1 = output1 + 1;
            end
      end
      else begin
            output1 = output1;
            i = i;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            done <= 0;
      end
      else if(start) begin
            done <= 1;
      end
      else begin
            done <= done;
      end
end
endmodule
```

Figure 75.
Translating Cv to Verilog for code containing a static for loop.

Verilog supports static for loops, so this construct is used. In Verilog static for loops are translated into multiple instances of the hardware described. If the desire is to loop over a set of code twenty times, Verilog will create and link twenty different instances of the hardware described within the for loop. Because the for loop is static, and does not contain any constructs that require CHard, `MyFunctionStaticForLoop` does not require CHard. This is reflected in the absence of the `functionCounter`, and will be done one clock cycle after `start` is high.

2.15.2.3 Dynamic For Loop

Example:

```
//Cv Code
int output1 MyFunctionDynamicForLoop (int input1) {
	int i;

	output1 = 0;
	for(i  = 0; i < input1; i = i + 1)
		output1 = output1 + 1;
}
```

Figure 76.
Cv code containing a dynamic for loop.

Translated into Verilog:

```
//Verilog Code
module MyFunctionDynamicForLoop (input clock, input reset, input start,
	input [31:0] input1, output reg done, output reg [31:0] output1);

reg [3:0] functionCounter;

//flow block
always@(posedge clock) begin
	if(reset) begin
		output1 <= 0;
		i <= 0;
	end
	else if(start) begin
		output1 <= 0;
	end
	else if(functionCounter != 0) begin
		case(functionCounter)
			1: i <= 0;
			2: // do nothing, control block is testing i < input1
			3: output1 <= output1 + 1;
			4: i <= i + 1;
			5: //do nothing, control block is testing i < input1
		endcase
	end
	else begin
		output1 <= output1;
		i <= i;
	end
end

//control block
always@(posedge clock) begin
```

```
	if(reset) begin
		functionCounter <= 0;
		done <= 0;
	end
	else if(start) begin
		functionCounter <= 1;
		done <= 0;
	end
	else if(functionCounter != 0) begin
		case(functionCounter)
			2:
				if(i < input) begin
					functionCounter <= 3;
				end
				else
					functionCounter <= 0;
					done <= 1;
				end
			5:
				if(i < input) begin
					functionCounter <= 3;
				end
				else
					functionCounter <= 0;
					done <= 1;
				end
			default:
				functionCounter <= functionCounter + 1;
				done <= done;
		endcase
	end
	else begin
		functionCounter <= 0;
		done <= done;
	end
end
endmodule
```

Figure 77.
Translating Cv to Verilog for code containing a dynamic for loop.

In the for loop the instructions must be executed in this order: initialize followed by repeating test, statement, step, until the test expression is false. In the flow block the statement are laid out sequentially and the control of which statement is executed is handled in the control block. When the `functionCounter` is 2 and 5 the flow block does nothing. When the `functionCounter` is 2 and 5 the control block is testing the test expression to determine if the for loop should be run again. If the test expression is true then the `functionCounter` is set to 3, the first statement in the for loop. Once the for loop has finished, when the `functionCounter` is 4, the flow block

executes the step (`i <= i + 1`). Because there is only a for block in this function, when the `functionCounter` is 5 and the test expression is false, then `MyFunctionDynamicForLoop` is done and `done` is set high.

2.16 While Loop

A *while loop* is used to continually loop over a set of code until the test expression is false. According to the grammar:

WhileStmt ::= **while (** Expr **)** Stmt

While loops have two elements; *test* expression and *statement*. The statement can be a single line statement, or a statement block.

```
//Pseudo Code
//Example 1
while(test)
      single line statement

//Example 2
while(test) {
      statement block
}
```

Figure 78.
Examples of while loops using a single line statement with no braces, and a statement block contained in braces.

In Example 1 of Figure 78 a while loop has a test expression in parenthesis and a single line statement following the while statement. The single line is the only statement that will executed for the while loop. In Example 2 of Figure 78 a statement block (within braces) is used. To include multiple statements to be looped over a statement block must be used. All code within the braces will be executed as long as the test expression is true.

The example section of the while loop section has been omitted. Because there are many special cases for the Cv Hardware Translator that differ from the Cv Software Translator, all examples are presented in the Cv Software Translator (section 2.16.1) and Cv Hardware Translator (Section 2.16.2).

2.16.1 Cv Software Translator

The while loop, much like the for loop, translates very closely to C++, only wrapping the test expression in a CvV<bool>.

Example:

```
//Cv Code
int output1 MyFunction(int input1){
	output1 = 0;
	while(output1 < input1)
		output1 = 2*output1 + input1;
}
```

Figure 79.
Cv code containing a while loop.

Translated into C++:

```
//C++ Code
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvFunctionStruct_MyFunction returnValue;

	returnValue.output1 = 0;
	while(CvV<bool>(output1 < input1).Me( )) {
		returnValue.output1 = 2*returnValue.output1 + input1;
	}
	return returnValue;
```

```
}
```

Figure 80.
Translating Cv to C++ for code containing a while loop.

In Figure 80 the while loop is created and the test expression (`output1 < input1`) is wrapped in a CvV<bool>, which accesses the Boolean value of the test expression through the `Me` function.

2.16.2 Cv Hardware Translator

To translate a while loop into hardware two situations must be considered: when the statement of the while loop contains constructs that require CHard, and when the statement of the while loop does not contain constructs that require CHard. Each of these cases are presented in the examples that follow.

2.16.2.1 While Loops Containing Constructs that Require CHard

In Figure 81 `MyFunction` contains a while loop, and a function call within the while loop. When there is a function call within a while loop the function will automatically require CHard.

Example:

```
//Cv Code
int a CalledFunction(int y)
{
        //some code
}

int ouptut1 MyFunction( int input1 ) {
      output1 = 0;
      while(output1 < 10)
            output1 = output1 + CalledFunction(input1).a;
}
```

Figure 81.

Cv code containing a while loop. This while loop contains a function call, and therefore requires CHard.

Translated into Verilog:

```
//Verilog Code
module MyFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg startCalledFunction_1;
wire doneCalledFunction_1;
wire [31:0] calledFunction_1_a;
CalledFunction calledFunction_1(
      .clock(clock),
      .reset(reset),
      .start(startCalledFunction_1),
      .y(input1),
      .done(doneCalledFunction_1),
      .a(calledFunction_1_a)
);

reg [2:0] functionCounter;

//flow block
always@(posedge clock) begin
      if(reset) begin
            output1 <= 0;
            startCalledFunction_1 <= 0;
      end
      else if(start) begin
            output1 <= 0;
      end
      else if(functionCounter != 0) begin
            case(functionCounter)
                  1: //test the test condition in the control block
                  2: startCalledFunction_1 <= 1;
                  3: startCalledFunction_1 <= 0;
                  4: output1 <= output1 + calledFunction_1_a;
            endcase
      end
      else begin
            output1 <= output1;
            startCalledFunction_1 <= startCalledFunction_1;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            functionCounter <= 0;
            done <= 0;
      end
      else if(start) begin
            functionCounter <= 1;
            done <= 0;
      end
      else if(functionCounter != 0) begin
```

```
            case(functionCounter)
                  1: if(output1 < 10) begin
                        functionCounter <= 2;
                  end
                  else begin
                        functionCounter <= 0;
                        done <= 1;
                  end
                  3: if(doneCalledFunction_1) begin
                        functionCounter <= 4;
                  end
                  else begin
                        functionCounter <= 3;
                  end
                  4: functionCounter <= 1;
                  default: functionCounter <= functionCounter + 1;

            endcase
      end
      else begin
            functionCounter <= 0;
            done <= done;
      end
end
endmodule
```

Figure 82.
Translating Cv to Verilog for code that contains a while loop. The while loop contains a function call, and therefore requires CHard.

In Figure 82 the computations to consider are setting `output1` to 0, `output1 < 10`, the inline function call to `CallFunction`, and `output1 = output1 + CalledFunction(input1).a`. These statements are observed in the flow block of `MyFunction`. When `start` is high `output1` is set to 0. Once the `functionCounter` is 1, `output1` is tested in the control block to see if `output1` is less than 10. When the `functionCounter` is 1, the flow block does nothing. When the `functionCounter` is 1, if the test expression is true the while loop is run again. The `startCallFunction_1` is set high when `functionCounter` is 2, and `startCallFunction_1` is set low when the functionCounter is 3. When the `functionCounter` is 3, the control block tests to see if `doneCallFunction` is high. If `doneCallFunction` is high, then the `functionCounter` is set to 4 to execute the addition and assignment. When the `functionCounter` is 4, the `functionCounter` is set to 1, to evaluate the test expression of the while loop. This process will

continue until the test expression is false. When the test expression is false, the `functionCounter` will be set to 0, and `done` will be set high.

2.16.2.2 While Loops Containing Constructs that do not Require CHard

In Figure 83 `MyFunction` contains a while loop that does not contain any constructs that require CHard. In the control block the test expression from the while statement is evaluated, and once the test expression is false `done` is set high. Also, there is an additional signal, `inProgress`, that goes high when `start` is high, and stays high until the test expression is false. If there were serial statements before or after the while loop then `MyFunction` would require CHard.

Example:

```
//Cv Code
int ouptut1 MyFunction( int input1 ) {
      while(output1 < 10)
            output1 = output1 + input1;
}
```

Figure 83.
Cv code containing a while loop. The while loop does not contain any constructs that require CHard, and therefore does not require CHard.

Translated into Verilog:

```
//Verilog Code
module MyFunction (input clock, input reset, input start,
      input [31:0] input1, output reg done, output reg [31:0] output1);

reg inProgress;

//flow block
always@(posedge clock) begin
      if(reset) begin
            output1 = 0;
      end
```

```
      else if(start) begin
            output1 = 0;
      end
      else if(inProgress) begin
            output1 = output1 + input1;
      end
      else begin
            output1 = output1;
      end
end

//control block
always@(posedge clock) begin
      if(reset) begin
            inProgress <= 0;
            done <= 0;

      end
      else if(start | inProgress) begin
            if(output1 < 10) begin
                  inProgress <= 1;
                  done <= 0;
            end
            else begin
                  inProgress <= 0;
                  done <= 1;
            end
      end
      else begin
            inProgress <= 0;
            done <= done;
      end
end
endmodule
```

Figure 84.
Translating Cv to Verilog for code containing a while loop. The while loop does not contain any constructs that require CHard, and therefore does not require CHard.

2.17 Return Statement

Software languages commonly use *return* statements to return a value from a function. Hardware languages do not use return statements and instead set the outputs of a module directly. In Cv the outputs of a function are set directly, and a return statement can be used if needed. A function will automatically be finished when it reaches the end of its code, but a return statement can be used to make a function complete earlier.

In contrast to software languages, in Cv a return statement is not followed by a value to return. Instead a return statement is always just a return, as seen below.

```
//Cv Code
return;
```

Figure 85.
A return statement in Cv. In Cv return statements do not have an argument.

When a `return` is encountered, the function exits and returns the current values that the outputs have been set to.

2.17.1 Example

```
//Cv Code
int output1 MyFunction(int input1){
	if( input1 == 1 )
		output1 = 2;
	else {
		output1 = 3;
		return;
	}

	output1 = output1 * 2;
}
```

Figure 86.
Cv code containing a return statement.

In Figure 86 a return statement is used to exit the function within the else clause of the if statement.

2.17.2 Cv Software Translator

```
//C++ Code
```

```
struct CvFunctionStruct_MyFunction {
	CvV<int> output1;
};
CvFunctionStruct_MyFunction MyFunction(CvV<int> input1) {
	CvFunctionStruct_MyFunction returnValue;
	if( CvV<bool>(input1 == 1).Me( ) ){
		returnValue.output1 = 2;
	}
	else {
		returnValue.output1 = 3;
		return returnValue;
	}
	returnValue.output1 = returnValue.output1 * 2;
	return returnValue;
}
```

Figure 87.
Translating Cv to C++ for code containing a return statement.

When translating to C++, `return` statements are replaced by returning the `returnValue`. This makes the function return the values of the outputs that have already been set.

2.17.3 Cv Hardware Translator

When a return statement is present, the Cv Hardware Translator will require CHard. This is because the return statement is handled by altering the control block.

```
//Verilog Code
module MyFunctionStaticForLoop (input clock, input reset, input start,
	input [31:0] input1, output reg done, output reg [31:0] output1);

reg[2:0] functionCounter;
//flow block
always@(posedge clock) begin
	if(reset) begin
		output1 <= 0;
	end
	else if(start) begin
		//do nothing, control block is evaluating if statement
	end
	else if(functionCounter != 0) begin
		case(functionCounter)
			1: output1 = 2;
			2: output1 = 3;
			3: //do nothing, the control block is returning
			4: output1 = output1 * 2;
		endcase
	end
	else begin
```

```
            output1 <= output1;
      end
end
//control block
always@(posedge clock) begin
      if(reset) begin
            functionCounter <= 0;
            done <= 0;
      end
      else if(start) begin
            if( input1 == 1) begin
                  functionCounter <= 1;
            end
            else begin
                  functionCounter <= 2;
            end
            done <= 0;
      end
      else if(functionCounter != 0) begin
            1: functionCounter <= 4;
            2: functionCounter <= 3;
            3: functionCounter <= 0; //return statement being executed here
            done <= 1;
            4: functionCounter <= 0;
            done <= 1;
            default: functionCounter <= functionCounter + 1;
      end
      else begin
            functionCounter <= 0;
            done <= done;
      end
end
endmodule
```

Figure 88.
Translating Cv to Verilog for a function containing a return statement.

When the `functionCounter` is equal to 3, the `return` statement would be executed. Although it is not the end of the function code, the `done` output is set high, and the `functionCounter` is set to 0.

2.18 Comments

Comments are text in the code that do not execute any instructions, but inform the user of intended meaning. Therefore this section presents examples, but nothing on the Cv Software Translator, or CV Hardware Translator.

2.18.1 Example

In Cv there are two styles of comments, line comments and block comments. Line comments use the "//" symbol and comment out the remainder of the line. For example:

```
//Cv Code
x = 5; //assigning 5 to x
```

Figure 89.
Cv code using a line comment. The "assigning 5 to x" will not be executed.

In this example `x = 5` will be executed, however the `//assigning 5 to x` will not, and is just text.

Block comments use the pair of symbols /* and */ to denote the start and stop of a comment block. Anything contained in within the two symbols is a comment and will not be executed. For example:

```
//Cv Code
x = 5;
y = 6;
/*
a = 2;
b = 3;
c = 7;
*/
```

Figure 90.
Cv code using a block comment. The text between "/*" and "*/" will not be executed.

In the code above 5 will be assigned to x, and 6 will be assigned to y. The other lines will not be executed.

Assignments to `a`, `b`, and `c` occur within a comment block. These commands will not be executed, and `a`, `b`, and `c` will not be assigned values.

CHAPTER III

EXPERIMENTAL EVALUATION

To assess the functionality of Cv, several representative algorithms have been selected as a basis for comparison. These algorithms were then written in three languages -- Cv, C++, and Verilog. Because Cv can be compiled into software or synthesized into hardware, the Cv software was compared to the C++ implementation, while the Cv hardware was compared to the Verilog implementation. These results have been compared on different figures of merit for the hardware and software implementations because the two platforms naturally have different ways that they are evaluated. Software results have been compared in terms of executable size, peak memory usage, and execution time. Hardware results have been compared based on FPGA area utilization (number of LUT's) and the number of clock cycles required to complete the computation. Power consumption is often proportionate to the number of LUT's, or gates, and therefore the number of LUT's can be used as a proxy for the expected power consumption.

In Section 3.1 we present the representative algorithms used, along with pseudo code and a brief description for each algorithm. Section 3.2 will present the results and comparisons. In Section 3.2 the Cv implementation of each algorithm is presented, followed by its software and hardware results.

3.1 Selected Test Algorithms

3.1.1 Fast Fourier Transform

The Fourier Transform was initially used to transform a signal from the time domain to the frequency domain. The Fourier Transform has been applied to many engineering problems such as electromagnetics, audio, and image processing [11]. The Fast Fourier Transform (FFT) is a digital implementation that takes advantage of the symmetry between roots of –1 to compute the Fourier Transform faster. Pseudo code describing the FFT is shown in Figure 91.

```
arrayOut FFT(numberOfElements, arrayIn)
{
      //divide phase
      if(numberOfElements == 1)
            return arrayIn[0]
      evenFFT = FFT(numberOfElements/2, even elements of arrayIn)
      oddFFT = FFT(numberOfElements/2, odd elements of arrayIn)

      //conquer phase
      for( i = 0 to numberOfElements/2 - 1)
      {
            complexSinusoid = e^(2*pi*k*i/numberOfElements)
            arrayOut[i] = evenFFT[i] + complexSinusoid*oddFFT[i]
            arrayOut[i + numberOfElements/2] = evenFFT[i]-
                  complexSinusoid*oddFFT[i]
      }
      return arrayOut;
}
```

Figure 91.
Pseudo code showing the Fast Fourier Transform.

The FFT is a recursive algorithm that uses a divide and conquer approach. In the divide phase, `arrayIn` is recursively divided into two separate arrays, one array contains the even elements of `arrayIn` , and the other array contains the odd elements of `arrayIn` . These two arrays are recursively further divided. The even elements of an array are the elements with an array index of 2*n*, where *n* is greater than or equal to 0 and is an integer. The odd elements of an array are the elements with an array index of 2*n* – 1, where *n* is greater than or equal to 1 and is an integer. In

the conquer phase, the array is recursively recombined while multiplying by roots of –1. The root of –1 is calculated by the complex exponential. The even and odd arrays are then recombined into a single array, using the `complexSinusoid`. The FFT is completed by recursively recombining each even and odd array.

3.1.2 Insertion Sort

In computer science sorting a list of elements is an algorithm that is used very often. There are many different sorting algorithms, both recursive and non-recursive. Insertion Sort is a non-recursive algorithm that begins on the left hand side of an array (array[0]), and progresses to the right hand side of the array (array[array size – 1]). Each element of the array is continually swapped with its left neighbor until the element's left neighbor is less than the element. Pseudo code describing Insertion Sort is shown below.

```
arrayOut InsertionSort(numberOfElements, arrayIn)
{
      for(i = 1 to numberOfElements – 1)
      {
            j = i;
            while(data[j] < data[j-1] & j > 0)
            {
                  temp = data[j]
                  data[j] = data[j-1]
                  data[j-1] = temp
                  j = j - 1
            }
      }
      return arrayOut
}
```

Figure 92.
Pseudo code showing the Insertion Sort.

3.1.3 Matrix Multiplication

Matrix operations are widely used in many applications, including circuits, quantum mechanics, three dimensional computer images, seismic surveys, and robotics [12]. Multiplying two *n* x *n* matrices results in an *n* x *n* matrix. Pseudo code describing Matrix Multiplication is shown in Figure 93.

```
matrixOut MatrixMultiplication(dimension, matrixA, matrixB)
{
      for(i = 0 to dimension)
      {
            for(j = 0 to dimension)
            {
                  for(k = 0 to dimension)
                  {
                        matrixOut[i][j] = matrixOut[i][j] + matrixA[i][k] *
                              matrix[k][j]
                  }
            }
      }
      Return matrixOut
}
```

Figure 93.
Pseudo code for Matrix Multiplication.

There are several different matrix multiplication algorithms that are used to optimized matrix multiplication. The `MatrixMultiplication` algorithm shown is known as the brute force method. The algorithm does not use any optimizations, and sets matrixOut[i][j] equal to the dot product of row *i* of matrixA and column *j* of matrixB.

3.1.4 Hash Table Algorithm

A hash table is a data structure that is used to efficiently store and retrieve data. The hash table uses different *buckets* to store data. Each piece of data to be stored has a key-value pair. The hash table uses a hash function on the `key` to determine which bucket to store the key-value pair. Pseudo code describing storing data in a hash table is shown in Figure 94.

```
LinkedList < LinkedList* > hashTable
Void StoreData(key, value)
{
      hashKey = HashFunction(key)
      hashTable[hashKey].Add(key, value)
}
```

Figure 94.
Pseudo code for storing data in a Hash Table.

The hash table in this example is represented by the `LinkedList` `hashTable`, whose elements each point to a different `LinkedList`. First a `hashKey` is generated from the given `key`. The `hashKey` indicates which `LinkedList` to use from the `hashTable`. The given key-value pair is then added to the `LinkedList` selected.

3.2 Results

In this section the Cv implementation and results for each algorithm are presented. For each algorithm, first the Cv code is presented, followed by the software results, and lastly the hardware results. The C++ code was compiled using G++ [10] and executed on Ubuntu [13]. The Verilog code was synthesized using Xilinx ISE [14] and simulated using Xilinx ISE Simulator ISIM [15]. The Verilog was synthesized using the Xilinx Virtex 6 XC6VLX75T as the device. In the first two examples, we compared that the results from Cv (for hardware as well as software compilation) against the results obtained by compiling the Verilog code and the C++ code respectively. In the last two examples, Verilog and C++ code were not compared.

3.2.1 Fast Fourier Transform

3.2.1.1 Cv Implementation

FFT is inherently a recursive algorithm. Cv currently does not support recursion, and so we had to construct a non-recursive FFT algorithm. In addition to this there are no library functions defined for sine, cosine, exponents, and logarithms in Cv. Each of these functions were implemented as Cv functions. Sine and cosine were implemented as a look up table made using if statements. The Cv sine and cosine functions take integer arguments instead of decimal numbers. This is because the functions are only defined for certain angles. The decimal angle in radians would be equal to the integer angle times pi, then divided by 180. Exponents were implemented in a function with a for loop to perform repeated multiplications. Logarithms were implemented in a function with a for loop to perform repeated divisions.

The FFT algorithm is implemented in the function `CvFFT` . `CvFFT` calculates the FFT for an 8 sample signal. The FFT could be reconfigured for a larger number of samples by changing the size of the arrays and the `numberOfSamples` variable. Initially the `inputArray` is rearranged in the same style as in the recursive implementation. Arrays that are suffixed with `Hold` are used to perform calculations on elements that have not yet been modified. Second, the array is rearranged in the same style as in the recursive implementation. The elements of the array are multiplied by complex exponentials while recombining.

```
double result Sin(int angle)
{
      if(angle == 0)
            result = 0.0;
      else if(angle == 1)
            result = 0.38268343;
      else if(angle == 2)
```

```
            result = 0.70710678;
      else if(angle == 3)
            result = 0.92387953;
      else if(angle == 4)
            result = 1.0;
      else if(angle == 5)
            result = 0.92387953;
      else if(angle == 6)
            result = 0.70710678;
      else if(angle == 7)
            result = 0.38268343;
}

double result Cos(int angle)
{
      if(angle == 0.0)
            result = 1.0;
      else if(angle == 1)
            result = 0.92387953;
      else if(angle == 2)
            result = 0.70710678;
      else if(angle == 3)
            result = 0.38268343;
      else if(angle == 4)
            result = 0.0;
      else if(angle == 5)
            result = - 0.38268343;
      else if(angle == 6)
            result = - 0.70710678;
      else if(angle == 7)
            result = - 0.92387953;
}

int result Pow(int base, int exponent)
{
      int i;
      if(exponent == 0)
            result = 1;
      else
      {
            result = base;
            for(i = 0; i < exponent - 1; i = i + 1)
                  result = result * base;
      }
}

int result Log(int base, int argument)
{
      int i;
      int test;
      result = 1;
      test = argument;
      while(test != base)
      {
            test = test / base;
            result = result + 1;
      }
}

double realResult[8], double imaginaryResult[8] CvFFT(double inputArray[8])
{
      int numberOfSamples;
```

```
	int i;
	int j;
	int k;
	int length;
	int blockSize;
	int repetitions;
	int index1;
	int index2;
	double original[8];
	double modified[8];
	double modifiedHold[8];
	double realResultHold[8];
	double imaginaryResultHold[8];
	double sin;
	double cos;
	int angle;
	numberOfSamples = 8;
	original = inputArray;

	modified = original;
	modifiedHold = modified;

		// divide
	length = Log(2, numberOfSamples).result;
	for (i = length; i > 0; i = i - 1)
	{
		blockSize = Pow(2, i).result;
		repetitions = numberOfSamples / blockSize;
		for(j = 0; j < repetitions; j = j + 1)
		{
			for (k = 0; k < blockSize / 2; k = k + 1)
			{
				index1 = j * blockSize + k;
				index2 = j * blockSize + k * 2;
				modified[index1] = modifiedHold[index2];
			}
			for (k = 0; k < blockSize / 2; k = k + 1)
			{
				index1 = blockSize / 2 + j * blockSize + k;
				index2 = j * blockSize + k * 2 + 1;
				modified[index1] = modifiedHold[index2];
			}
		}
		modifiedHold = modified;
	}
	realResult = modified;
	realResultHold = realResult;

	//conquer
	for (i = 1; i <= length; i = i + 1)
	{
		blockSize = Pow(2, i).result;
		repetitions = numberOfSamples / blockSize;
		for(j = 0; j < repetitions; j = j + 1)
		{
			for (k = 0; k < blockSize / 2; k = k + 1)
			{
				angle = k * 16 / blockSize;
				sin = Sin(angle).result;
				cos = Cos(angle).result;

				index1 = j * blockSize + k;
```

```
                        index2= j * blockSize + k + blockSize/2;

                        realResult[index1] = realResultHold[index1]
                                + realResultHold[index2]*cos
                                + imaginaryResultHold[index2]*sin;

                        imaginaryResult[index1] =
                                imaginaryResultHold[index1] -
                                realResultHold[index2]*sin +
                                imaginaryResultHold[index2]*cos;


                        realResult[index2] = realResultHold[index1]
                                - realResultHold[index2]*cos
                                - imaginaryResultHold[index2]*sin;

                        imaginaryResult[index2] =
                                imaginaryResultHold[index1]
                                + realResultHold[index2]*sin
                                - imaginaryResultHold[index2]*cos;
                    }
                }
                realResultHold = realResult;
                imaginaryResultHold = imaginaryResult;
            }
}
```

Figure 95.
Cv implementation of the FFT and associated functions.

3.2.1.2 Software Results

Table 2.

Fast Fourier Transform	Cv Software	C++	Cv/C++
Executable Size	68 KB	9 KB	7.55
Peak Memory Use	1064 KB	1064 KB	1
Execution Time	10.468 seconds	0.035 seconds	299.09

Software results for the FFT.

The C++ implementation was written using the same non recursive algorithm style as the Cv implementation. This allows the two algorithms to be compared. The Cv implementation resulted in a larger executable and longer execution time. However, the peak memory used was equal for

both implementations. The execution time was determined by running the FFT algorithm on a random eight element array 10,000 times.

3.2.1.3 Hardware Results

Table 3.

Fast Fourier Transform	Cv Hardware	Verilog	Cv/Verilog
Number of LUT's	13,141	5,413	2.427
Clock Cycles	3188	70	45.54

Hardware results for the FFT.

The Verilog implementation was done using the Xilinx IP Core Generator. The IP Core Generated algorithm provides a well refined algorithm implementation for comparison that is highly optimized for the Xilinx device. Because of this, it is not surprising that the Verilog implementation with IP Core Generator produces better results. The Verilog implementation used fewer LUT's and fewer clock cycles than the Cv implementation.

3.2.2 Matrix Multiplication

3.2.2.1 Cv Implementation

There are several highly refined matrix multiplication algorithms, and many of these refined algorithms use recursive approaches. With Cv we are not trying to test the effectiveness of an algorithm, but instead the effectiveness of the Cv language and the correctness of the translation tools. Therefore, the Cv implementation is a brute force matrix multiplication algorithm that loops through the mathematical definition of matrix multiplication.

```
int result[5][5] MatrixMultiplication (int a[5][5], int b[5][5])
{
	int i;
	int j;
	int k;
	int size;

	size = 5;

	for(i = 0; i < size; i = i + 1)
	{
		for(j = 0; j <size; j = j + 1)
		{
			for(k = 0; k < size; k = k + 1)
			{
				result[i][j] = result[i][j] + (a[i][k] * b[k][j]);
			}
		}
	}
}
```

Figure 96.
Cv implementation of Matrix Multiplication.

3.2.2.2 Software Results

Table 4.

Matrix Multiplication	Cv Software	C++	Cv/C++
Executable Size	37 KB	9 KB	4.111
Peak Memory Use	1064 KB	824 KB	1.291
Execution Time	7.802 seconds	0.024 seconds	325.083

Software result for the Matrix Multiplication.

The C++ implementation was written using the same brute force method as the Cv implementation. This will allow the two results to be closely compared. The C++ executable is smaller than the Cv executable. The C++ implementation also used less memory and took less time to execute. In the FFT and Matrix Multiplication the C++ implementation had a smaller

executable size, and less run time. However, in the FFT the Cv and C++ implementations had the same peak memory usage. In contrast, the C++ implementation of the Matrix Multiplication used about 30% less memory. The execution time was determined by running the Matrix Multiplication algorithm on two random 5 x 5 element matrices 10,000 times.

3.2.2.3 Hardware Results

Table 5.

Matrix Multiplication	Cv Hardware	Verilog	Cv/Verilog
Number of LUT's	3,360	2191	1.536
Clock Cycles	501	18	27.833

Hardware results for the Matrix Multiplication.

The hardware implementation was done using systolic matrix multiplication. This algorithm is more refined than the brute force method, and therefore it is expected to produce better results. The Verilog implementation required fewer LUT's and clock cycles. This comparison holds true for both the FFT and the Matrix Multiplication.

The Cv FFT and Matrix Multiplication implementations were compared to the C++ and Verilog implementations.

In addition to these algorithms, we have also written Insertion Sort and a Hash Table in Cv. The statistics for these functions are presented for Cv, but are not compared to C++ and Verilog implementations.

3.2.3 Insertion Sort

3.2.3.1 Cv Implementation

The sorting algorithm selected to use was Insertion Sort. There are many sorting algorithms, many of which are recursive algorithms. Insertion Sort is not a recursive algorithm. Insertion Sort works by sorting a subset of the array, and continually increasing the subset until it is the full array. The Insertion Sort was implemented on a sixteen element array. This implementation could easily be extended to larger arrays by changing the parameters in the code.

```
int a[16] InsertionSort(int d[16])
{
	int i;
	int j;
	int temp;
	int data[16];

	data = d;
	for(i = 1; i < 16; i = i + 1)
	{
		j = i;
		while(data[j] < data[j-1] & j > 0)
		{
			temp = data[j];
			data[j] = data[j-1];
			data[j-1] = temp;
			j = j - 1;
		}
	}

	a = data;
}
```

Figure 97.
Cv implementation of Insertion Sort.

3.2.3.2 Software Results

Table 6.

Insertion Sort	Cv Software
Executable Size	38 KB
Peak Memory Use	1064 KB
Execution Time	2.762 seconds

Software results for Insertion Sort.

The execution time was determined by running the Insertion Sort algorithm on a random sixteen element array 10,000 times.

3.2.3.3 Hardware Results

Table 7.

Insertion Sort	Cv Hardware
Number of LUT's	2180
Clock Cycles	351.6

Hardware results for Insertion Sort.

The number of clock cycles is an average of sorting ten different sixteen element data sets.

3.2.4 Hash Table

3.2.4.1 Cv Implementation

Hash Tables are efficient methods to store and retrieve data. We have implemented the storing of data in a Hash Table. Retrieving data is more trivial and similar to searching any list. In our Cv

implementation we use four main pieces of data: bucketStart, hashData, keys, and pointers. HashData holds the data that is stored. The hash function used is the remainder when the key is divided by eight. Each piece of data is always stored in the rightmost available location of hashData. The elements of bucketStart hold the array index of hashData for the first element of each bucket. Pointers holds the indices of each element for the next element in the bucket. Keys stores the respective key for each value stored in hashData.

```
int n1 HashTable(int key, int value)
{
	int bucketStart [8];
	int hashData [100];
	int keys [100];
	int pointers [100];

	int nextAvailable;
	int hashKey;
	int position;
	int i;
	int loops;

	loops = 100;

	hashKey = key%8;
	position = bucketStart[hashKey];
	if(nextAvailable == 0)
	{
		nextAvailable = 1;
	}
	if(position == 0)
	{
		hashData[nextAvailable] = value;
		keys[nextAvailable] = key;
		bucketStart[hashKey] = nextAvailable;
		nextAvailable = nextAvailable + 1;
	}
	else
	{
		for(i = 0; i < loops; i = i + 1)
		{
			if(pointers[position] != 0)
			{
				position = pointers[position];
			}
			else
			{
				i = loops;
			}
		}

		hashData[nextAvailable] = value;
```

```
            keys[nextAvailable] = key;
            pointers[position] = nextAvailable;
            nextAvailable = nextAvailable + 1;
        }
}
```

Figure 98.
Cv Implementation of the Hash Table.

3.2.4.2 Software Results

Table 8.

Hash Table	Cv Software
Executable Size	66 KB
Peak Memory Use	1064 KB (2916)
Execution Time	33.594 seconds

Software results for the Hash Table.

The execution time was determined by running the Hash Table algorithm on a random 99 key-value pairs 10,000 times.

3.2.4.3 Hardware Results

Table 9.

Hash Table	Cv Hardware
Number of LUT's	67,013
Clock Cycles	47.08

Hardware results for the Hash Table.

The hardware results reflect running the Hash Table algorithm on a 99 random key-value pairs. The number of clock cycles is the average number of clock cycles to store one element in the

hash table. As the number of elements increases, the number of clock cycles needed to store one element increases. The number of LUT's required is much greater than the other example algorithms. This increase in LUT's is due to the large arrays used to store data. To reduce the number of LUT's, the number of bits per integer could be reduced.

CHAPTER IV

FUTURE WORK

In this thesis, we have presented the philosophy and semantics of Cv, for both hardware and software use. We have been able to implement and demonstrate Cv on a variety of algorithms that find common use in industrial practice. Our preliminary results demonstrate correctly translating Cv code into hardware and software. As Cv undergoes further development, we anticipate several potential improvements. Cv also has several applications beyond the currently envisioned use of Cv as a language that is compiled into *only* software or *only* hardware. The remainder of this section first discusses improvements to be made to Cv, followed by future applications of Cv.

4.1 Improvements

There are several improvements that can be made to Cv. Some of these improvements are based on an analysis of the results we have gathered, while others are general improvements that have been discovered while developing Cv. We next discuss improvements based upon the current results, followed by a discussion on general improvements.

4.1.1 Improvements Based on Current Results

In each case that was compared against C++ and Verilog, the Cv implementation took longer and required more resources. Given that Cv adds a new level of abstraction, a decrease in optimality

is not surprising. The two key areas that Cv performed the worst in were execution time in software, and number of clock cycles in hardware.

To analyze the cause of the Cv software implementation being slower than the C++ implementation, it was initially suspected that the CvC and CvV special classes were increasing (slowing) software execution time. When translating Cv to C++, we will call the C++ that is produced the *resulting C++* (RC++). By manually altering the RC++, all CvV variables were removed, and changed to their corresponding C++ data types. After this alteration, the RC++ code was found to execute at the same speed as the C++ implementation. The next step was to determine why the CvV class was slowing down the Cv software. It was determined that the amount of casting in the CvC and CvV classes were slowing down the Cv software. The casting is due to using C++ templates to create a high amount of flexibility. To improve this bottleneck, the CvC and CvV classes would need to narrow their focus to the basic data types, and not be a template class with template functions. It is conjectured that this would improve the speed of the RC++ code significantly.

The second area where Cv performed poorly was the number of clock cycles required in hardware. When translating Cv to Verilog, we will call the Verilog that is produced the *resulting Verilog* (RV). The functions that were used to test Cv all required CHard so that the number of clock cycles could be compared. When translating Cv to Verilog, function calls, double operations, and loops require multiple clock cycles. However, all other single line statements take a single clock cycle to execute. Because *all* statements are given a single clock cycle, the number of clock cycles required is increased beyond what is required. Often several small

instructions could be combined to execute in a single clock cycle. By better analyzing the user's code, the Cv translator can improve the number of statements that are executed in a clock cycle. This can possibly be implemented as a post-processing step which operates on the RV code as well. It is anticipated that this change will improve the number of clock cycles needed to execute hardware obtained from Cv.

4.1.2 General Improvements

With the current implementation of Cv there are software and hardware features that are not currently supported. A few popular features that should be supported in the future include: object oriented design, pointers, and recursion.

Software and hardware both already support object oriented designs. In software languages such as C++, C#, and Java, objects can be constructed out of classes. Each class has data values that belong to it, and functions it can perform. Verilog has modules that contain data values, and can perform operations on the data values as well. The problem with Cv is that in Verilog, a separate module is created for each function call. Therefore, a function call to the same function creates multiple instantiations of a module. By calling a function several times, unneeded hardware is created, which increases resource utilization. By switching to an object oriented model where the user creates each object manually, the user will have control over how many objects and modules are instantiated in hardware.

Pointers and recursion are two popular features in software languages that have not yet been implemented in Cv. Pointers have the ability to change the variable they point to, and often it

cannot be determined at compile time what variable the pointer is pointing to. In addition to this, loops in software make it possible to allocate an unlimited number of objects with pointers. Hardware does not have the ability to create more resources when needed. Therefore this makes pointers a challenge to implement in hardware. Recursive functions, if implemented in Cv, are also challenging to translate to Verilog. A recursive function in software is a function that makes a call to itself. The challenging aspect of recursive functions is that it cannot be determined at compile time how many times a function will be called. In addition to this, each time a recursive function is called, more resources must be allocated. Similar to pointers, hardware cannot create more resources when needed, making recursive functions that allocate more resources challenging to implement. Pointers and recursive functions in hardware are areas of interest that we hope to introduce in a future version of Cv.

To support both pointers and recursion in hardware, we envision the use of a scratchpad memory, whose contents may be used to allocate additional resources as required. This is a subject of future work.

4.2 Future Applications

Due to the unique philosophy of Cv compared to current software and hardware languages, Cv is better prepared to be adapted to new platforms being developed. The two subjects that we will focus on are co-design systems, and object architect systems.

4.2.1 Co-design Systems

Currently Cv focuses on using a single piece of code, and being able to translate the entire code to either software, or hardware. However, there is great value in the possibility of translating *part* of the user's code to software, and the remainder of the user's code to hardware. These two parts, working together in a parallel software and hardware environment, have the potential to increase speed and efficiency of digital systems. By analyzing the user's code, the part of the code that is computationally intensive can be translated into hardware and run on an FPGA, while the rest of the code can be translated to software and run on the CPU. Parts of the Cv code that are natively matched for hardware (or software) can be compiled into hardware (software). This would alleviate a key restriction of our current Cv implementation. Some research groups have been doing research on co-design systems, and it is likely that Cv will have an impact in this area.

Currently large technology companies such as Intel, IBM, and Microsoft are designing hardware boards and systems that use a CPU and FPGA on the same platform [16] [17] [18]. These parallel architectures have been used to accelerate specific applications, and have been used primarily in enterprise-class server systems. In order for these platforms to succeed in the enterprise-class server applications as well as in consumer applications, tools will need to exist that allow developers to easily take advantage of these architectures. Currently programming languages do not adequately describe hardware, and HDL's do not properly instruct software. Instead, a language that can create software and hardware from a single code base will be ideal to power these systems. This is a key area where Cv can be enhanced, to meet the needs of a changing computing environment. By enabling Cv to translate the computationally intense code to hardware, and the rest of the code to software (or translating the parts of the code that are

natively matched for hardware (software) to hardware (software)), we would be able to impact a rapidly developing industrial computing platform which could be the future of the computing industry in the years ahead.

4.2.2 Object Architect Systems

A new architecture that is being researched is called the *object architecture*. In the object architecture, all "items" (program or data) associated with the computer are objects located in the cloud. Therefore the only thing the computer needs is a CPU capable of manipulating these objects, RAM, and other hardware components. This model removes the hard drive, and places a heavy focus on the cloud, but fails to greatly reduce the other hardware.

A modification to object architecture is to use an FPGA in conjunction with the CPU, and use the FPGA to execute the other hardware functions. With this model a new obstacle is incurred; files no longer have hardware that can interpret their structure. Using Cv, each type of file can carry with it the associated code to describe how to interpret the file. Cv eliminates the need for large amounts of specific hardware, and allows each type of file to dynamically create its needed hardware on the FPGA.

REFERENCES

[1] J. Bhasker, *A Verilog HDL Primer*. Star Galaxy Pub, Ed. 2: 1999

[2] B. Stroustrup, *Programming Principles and Practice Using C++*. Addison-Wesley: 2008.

[3] P. J. Ashenden, *The Designer's Guide to VHDL*. Elsevier Science, Ed. 3: 2008.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice Hall, Ed. 2: 1988.

[5] J. Albahari and B. Albahari, *C# in a Nutshell: The Definitive Reference*. O'Reilly Media, Ed. 5: 2012.

[6] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*. Addison-Wesley, Ed. 4: 2006

[7] *LegUp Documentation*, Release 3.0, University of Toronto, 2013

[8] N. Dave, "A Unified Model for Hardware/Software Co-design", Ph. D. dissertation, Dept. Elect. Eng. and Comp. Sci., Massachusetts Institute of Technology, 2011

[9] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures," IBM Research, 2010

[10] *Using the GNU Compiler Collection*, GCC version 4.7.4, GNU Press, Boston, MA

[11] "Q: What is a Fourier transform? What is it used for?," http://www.askamathematician.com/2012/09/q-what-is-a-fourier-transform-what-is-it-used-for/

[12] "Application of Matrices in Real Life," http://www.edurite.com/kbase/application-of-matrices-in-real-life

[13] “Ubuntu,” http://www.ubuntu.com/

[14] *ISE Design Suite 13.4: Release Notes Guide*, v13.4, Xilinx, 2012

[15] *ISim User Guide*, v14.1, Xilinx, 2012

[16] N. Carter, “Call for Proposals: Intel-Altera Heterogeneous Architecture Research Platform Program,” http://www.sigarch.org/2015/01/17/call-for-proposals-intel-altera-heterogeneous-architecture-research-platform-program/

[17] T. P. Morgan, “IBM Accelerates Power8 Clusters With GPUs, FPGAs, And Flash, ” http://www.enterprisetech.com/2014/10/02/ibm-accelerates-power8-clusters-gpus-fpgas-flash/

[18] “Catapult,” http://research.microsoft.com/en-us/projects/catapult/