VisLang: A Visual Language
Final Report

Bryant Eisenbach (UNI: bje2113)

Date: 08/14/2015

Contents

List of Figures

List of Tables

List of Code Listings

1 Introduction

VisLang is a block diagram language designed to allow fast and easy prototyping of programs for embedded processors. The language is created with a graphical editor in mind, and the core language is designed to be extensible so that any graphical editor can add additional elements or attributes for graphical display or other features.

1.1 Key Language Features

The language itself is based on the idea of blocks: small parts that can be grouped together into ever larger blocks and re-used across different programs. VisLang has a small group of fundamental (or atomic) blocks that will be understood by the VisLang compiler. Other blocks will be constructed as groupings of these atomic blocks, and can be referenced in other files. Libraries of useful function blocks can be constructed from these atomic blocks containing common parts such as timers, latches, etc. The ability to include blocks from libraries and other programs is a standard feature of the language.

The syntax of VisLang leverages standard XML, giving the language a well-formed and machine readable backbone. As noted previously, the point of leveraging XML is so that 3rd party programs can manipulate the file format in an easy way, and so that external programs can add additional elements (e.g. visual information for display) and attributes (e.g. location information) to the existing set of elements and attributes defined by the language. Those additional tags not included in the list of recognized elements/ attributes will be ignored by the compiler as a valid program is only defined as a series of well-connected blocks. All parts have a set of necessary attributes, and all connections require the source to exist. This creates a natural flow to interpreting the language, such that only functional errors should be raised by the compiler during compilation.

The VisLang Compiler will parse and check the specified input file and generate a viable C source file that can be used in combination with other generated and manually created C source files to combine into fully functional programs for embedded devices. Each generated file contains code that is completely reusable as the generated code has standard interfaces and does not rely on global definitions. Some manual coding will still be necessary to link into different types of embedded devices, the point is to create good intermediate code such that linking to I/O devices and dealing with the nuances of an arbitrary embedded device can be minimized.

2 Language Tutorial

Any well-formed VisLang program can be constructed with the following guidance. Firstly, the outer-most set of tags (the top of the XML tree or Top Block) must be a BLOCK element and should be given a name appropiate to the intended functionality (file name and block name do not have to match). Next, a selection of INPUT and OUTPUT elements should be chosen for that block that describe how it will interface with other blocks or C source files. When those have been chosen and given names and datatypes, it is now time to design the inner logic of the block being made.

Any collection of parts can be strung together from any input to any output. The first important caveat to this is that the datatype of each input and output must match to the corresponding connection being made. All parts in VisLang have an explicit or implicit datatype, and that must be matched up as suggested to avoid compilation errors. The second important caveat is that all calculations avoid referencing themselves. This means that when tracing from any output to any input, there is no instance of a calculation being used to define itself unless a MEM block has been placed to prevent an algebraic loop occurance.

Next comes the decision if there will be any references to external blocks. The user can specify the location of an external block in a file using the following syntax: "/path/to/file.vl|path|to|block". When using external blocks, it is important to match the names and datatypes of the inputs/outputs to that block when making connections to that block. Any incorrect types or names will be flagged as an error at compilation time. Any BLOCK in any VisLang file can be referenced, but each file must be compiled separately to avoid runtime errors when attempting to compile the target generated C files.

Finally, the design of the program could have grown to sufficient complexity where encapsulating that functionality into a separate block would be desireable. At that point, encapsulating all of the chosen parts into another BLOCK part would allow that part to be isolated from other parts in the program (different namespace), and that part can be referenced into another block in that program or any other.

There are a few specific things about some of the parts VisLang provides worth noting. All of the logic gate elements besides the NOT gate (AND, OR, etc.) and the SUMMER and PROD elements are defined as having two or more inputs. The way this works in practice is that each successive input increments the number after the word "input" e.g. "input1", "input2", "input3", etc. when making connections to these parts. If a number is skipped or the count does not start at 1, a compilation error will be raised. These parts are known as "binary recursive" parts because the operation involved will be applied to every input to the block in a recursive fashion e.g. (input1 op (input2 op (...))).

Most attributes for elements are defined with a string. The name attribute is common to every part in the VisLang language. The connection elements can reference these names when making connections between any two blocks. If an

element is an atomic part (any element besides BLOCK and REFERENCE), then linking a block input to that name is as easy as using that block's name. This is due to the fact that all of the atomic parts in VisLang are defined as only having one output, so there is no ambiguity. The other attributes have more explicit values that must match what is specified in the LRM.

When using the "ic" or "value" attributes, values they require are literals of the relevant datatype. Examples of literals for all VisLang types are below:

datatype	example literal(s)
boolean	false, true
single, double	1.000, -100., .000
integer	100, 0x20, 2x1011, 8x671, -120

Note booleans can only be false or true. Floating point quantities (single, double) can be a decimal quantity (with or without significant digits after the decimal point), or specified using a decimal point e.g. 10.600. Floating point literals can also be specified with a negative sign as well e.g. -1234.00. Integer quantities (e.g. uint8, uint16, uint32, uint64, int8, int16, int32, int64) can be specified as a decimal quantity (cannot have a decimal point), or using hex (e.g. 0x1A), binary (e.g.2x1010), or octal (e.g. 8x2462) representation. Only signed integer datatypes can have a negative sign in front.

2.1 Example Program

The following program illustrates some of the major features of the language. The program itself takes an Input (presumably on the target device) and starts a timer when the input is enabled. When the timer counts up to the target time, it will set the Output true and reset the timer, creating a pulse-blinking light with a period of 2 seconds.

Listing 1: Example Top Level

../example/timed-blinking-light.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="timed_blinking_light">
    <!-- This block denotes the contents of a program. Everything
         contained within (including file references would be compiled
         as a single binary. -->
    <vl:INPUT name="digital_input_1" datatype="boolean"/>
    <vl:CONSTANT name="time" datatype="single" value="2.000"/>
    <vl:NOT name="not_di_1">
        <vl:CONNECTION to="input" from="digital_input_1">
            <!-- Any unrecognized elements or attributes are ignored, e.g.
                 A GUI Program could specify the shape of the connection
                 here, but the compiler would ignore this attribute.  -->
            <shape>This will get ignored!</shape>
        </vl:CONNECTION>
```

```
    </vl:NOT>
    <!-- literal constants for booleans are "true" and "false"-->
    <vl:MEM name="count_expired_lp" ic="false" datatype="boolean">
    <!-- Memory block would store the state each pass of the variable
         specified by current_pass_value at the end of execution
         such that the last_pass_value can be used in the local scope
         without suffering from algebraic loops -->
        <vl:CONNECTION to="current"
            from="timer_instance_1|count_expired"/>
    <!-- The | operator on a name denotes an available connection -->
    </vl:MEM>
    <vl:OR name="reset_blink">
    <!-- OR, AND, etc. Gates can specify any number of inputs via
         incrementing the input specifiers "input1", "input2",
         "input3", etc. -->
        <vl:CONNECTION to="input1" from="not_di_1"/>
        <!-- Use the block name directly if it is an atomic part  -->
        <vl:CONNECTION to="input2" from="count_expired_lp"/>
    </vl:OR>
    <vl:REFERENCE name="timer_instance_1" ref="./timer.vl|timer">
        <!-- Reference block references a block in an external file as
             specified. File location is referenced relatively. All
             Inputs and Outputs of that block will be checked at compile
             time to match the connections made to the block. -->
        <vl:CONNECTION to="start" from="digital_input_1"/>
        <!-- Input and output connections to blocks are partially ambigious.
             However for a Connection to work, one and only one of "to" or
             "from" attributes must be an input/output of the part. -->
        <vl:CONNECTION to="reset" from="reset_blink"/>
        <vl:CONNECTION to="time" from="time"/>
    </vl:REFERENCE>
    <vl:OUTPUT name="digital_output_1" datatype="boolean">
    <!-- It is good practice to define outputs at the bottom of a document -->
        <vl:CONNECTION to="digital_output_1"
            from="timer_instance_1|count_expired"/>
        <!-- Any un-attached outputs to a block are optimized out, e.g.
             elapsed_time. All inputs are required -->
    </vl:OUTPUT>
</vl:BLOCK>
```

As noted, the above file contained a reference to another part called timer defined in timer.vs in the same directory. Any references must take place on a relative path to that file, and that reference must contain the same number of inputs specified by the target file inside the file referencing that part. The number of outputs need not match, but any outputs specified in the file referencing that part must also match what is available from the target file or an exception will be thrown. All other unused outputs will be disregarded. The following file displays the target file, complete with the relevant inputs and outputs as specified/required by the previous file.

Listing 2: Example Referenced Block

../example/timer.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="timer">
    <!-- The BLOCK element denotes a subsystem of parts -->
    <!-- All "parts" added by the user can use Inputs and/or
         Outputs for utilization elsewhere in project. The
         reference will search the path for that file -->
    <!-- All Inputs do not have to be used and will be optimized out -->
    <vl:INPUT name="start" datatype="boolean"/>
    <vl:INPUT name="reset" datatype="boolean"/>
    <vl:INPUT name="time" datatype="single"/>
    <!-- Constants can be defined as a seperate block as well -->
    <vl:CONSTANT name="zero_constant" datatype="single" value="0.000"/>
    <!-- The DT block puts out the difference in time between
         successive passes of program. In a Soft RTOS, this
         would be a variable number. In a Hard RTOS, this
         would be a constant number. Here, we are saying the
         module will run around 10Hz, or 100ms (0.1 s).
         The DT module needs an initializer to guess the value
         on the first pass, but will be updated every pass afterwards -->
    <vl:DT name="time_since_last_pass" ic="0.100"/>
    <vl:NOT name="count_not_expired">
        <vl:CONNECTION to="input" from="count_expired_lp"/>
    </vl:NOT>
    <vl:AND name="start_enb">
        <vl:CONNECTION to="input1" from="start"/>
        <vl:CONNECTION to="input2" from="count_not_expired"/>
    </vl:AND>
    <vl:IF name="increment_value" datatype="single">
    <!-- Control flow IF switch: If Control is true, execute
         True assignment, else execute False assignment -->
        <vl:CONNECTION to="control" from="start_enb"/>
        <vl:CONNECTION to="true" from="time_since_last_pass"/>
        <vl:CONNECTION to="false" from="zero_constant"/>
    </vl:IF>
    <vl:SUM name="summer" datatype="single">
    <!-- The summer will add all the inputs together. If you want
         add a negative number, use the NEG part to negate the
         signal before connecting to this part. -->
    <!-- Additionally, the PROD part exists for taking the PI
         product of a set of inputs, and the INV command for taking
         the recipicral of a number (divide by zero runtime error
         is partially mitgated, but unexpected operation may occur) -->
        <vl:CONNECTION to="input1" from="increment_value"/>
        <vl:CONNECTION to="input2" from="elapsed_time_lp"/>
    </vl:SUM>
    <vl:IF name="reset_switch" datatype="single">
        <vl:CONNECTION to="control" from="reset"/>
        <vl:CONNECTION to="true" from="zero_constant"/>
        <vl:CONNECTION to="false" from="summer"/>
    </vl:IF>
    <vl:COMPARE name="is_count_expired" datatype="single" operation=">=">
        <vl:CONNECTION to="lhs" from="elapsed_time"/>
        <vl:CONNECTION to="rhs" from="time"/>
    </vl:COMPARE>
```

```
    <vl:MEM name="elapsed_time_lp" datatype="single" ic="0.000" >
        <vl:CONNECTION to="current" from="elapsed_time"/>
    </vl:MEM>
    <vl:MEM name="count_expired_lp" datatype="boolean" ic="false">
        <vl:CONNECTION to="current" from="count_expired"/>
    </vl:MEM>
    <!-- All Outputs need to have a connection in the part,
         at least to a constant -->
    <vl:OUTPUT name="count_expired" datatype="boolean">
    <!-- Outputs of a subsystem need to have a connection specified -->
        <vl:CONNECTION to="count_expired" from="is_count_expired"/>
    </vl:OUTPUT>
    <vl:OUTPUT name="elapsed_time" datatype="single">
        <vl:CONNECTION to="elapsed_time" from="reset_switch"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

3 Language Reference Manual

3.1 Lexical Convention

VisLang uses standard XML syntax for its file specification. Several built-in elements, called Parts, are defined that make up the core of the language. The elements have a list of required attributes that must be defined using the correct type that attribute expects. Parts can be grouped into containers called Blocks using the BLOCK element, and that Block can be referenced internally or externally (using the REFERENCE element). All Parts have a name attribute that must be unique in the local scope of the block it is defined in and can be used as a named reference using CONNECTION elements to specify connections between different parts and blocks. Each project file must contain one, and only one, top level BLOCK element, but a block element can contain any number or level of additional block elements or built-in parts.

3.1.1 XML Elements and Attributes

Users of VisLang should be comfortable with how XML syntax works, but the following is a quick overview of how VisLang uses standard XML. XML elements are defined using the start tag " < " and end tag " > ". The element identifier immediately follows the beginning "<" character of a tag and is a valid Name matching any alphanumeric characters and the underscore character, completing at the next character of whitespace. XML elements can also contain attributes inside the tag after the tag identifier with white space following the identifer and seperating each additional attribute. The attributes are assigned a value using the syntax *attribute* = "*value*", where value can be anything representable without breaking the current line. The white space between each attribute assignment can include linebreaks, however that practice should be discouraged unless necessary to produce an easy to read document. Finally, XML elements need a way to describe when they are finished being defined, also known as "closing".
An element can be closed immediately using the "/" character before the end of the element tag (e.g. " < *element*/ > ") or with an additional tag with the "/" character following the " < " of that "closing tag" (e.g. " < *element* >< /*element* > "). All elements need to be closed in order to be considered well-formed and not raise an error. The second way of defining the closing tag means that the element can also contain inner elements. VisLang Block elements can contain any elements inside it including Parts and other Blocks. However, basic Parts cannot contain other Parts inside them, only ignored XML elements. All Parts and Blocks must contain their corresponding CONNECTION elements inside them if they are to be connected. Additionally, the CONNECTION element must be the first contained elements in a BLOCK definition (excluding comments).

3.1.2 Accepted Elements and Attributes

An XML namespace vl: was created such that only elements within the namespace are scanned. VisLang only accepts elements in the namespace and attributes inside those elemenbts to be used for compiling purposes, so by design all other elements and attributes not matching this set will be ignored. This decision allows developers to define programs that utilize VisLang as a base language.

3.1.3 Accepted Types

Attributes in vislang can only contain values matching the primative types accepted by the language. The scanning stage of the compiler will ignore any attributes whose values don't comply with this rule. This means that additional attributes to an accepted element can be defined, but VisLang will raise an error at in a later stage if a required attribute is missing. The list of accepted types for attribute values are as follows:

Table 2: Accepted Attributes

Type	Example	Regexp
name	"block_name"	[A-Za-z][A-Za-z0-9_]*
ref	"./path/to/file.vl"	("./" | "../"+ | "/") ([A-Za-z0-9-_.]* "/")* name ".vl"
to, from	"|block|block|etc"	("|" name)+
ic, value	-0x32Ab7f	true, false [+-]? ([0-9]+ "." [0-9]* | "." [0-9]+) [+-]? [0-9]+ [+-]? 0 [xX] [A-Fa-f0-9]+ [+-]? 0 [oO] [0-7]+ [+-]? 0 [bB] [0-1]+
datatype	uint32	boolean single double [u]?int(8|16|32)
operation	"=="	" > "," < ","! = " " == "," >= "," <= "

Note: value, ic accepts binary, hexidecimal, octal, and decimal coded signed and unsigned integers.

3.1.4 Comments

Although the usage scenario for VisLang is for developers to use the language specification as a baseline for further customizations, the VisLang compiler will

accept XML-style comments. XML style comments start with the tag "<!−−", and end with the tag "−− >". The scanner stage of the compiler will ignore any character between the opening and closing part of a comment. Comments are not nested by design in XML, and the VisLang compiler also does not accept nested comments. The W3C specification for XML states that comments also should never contain the "−−" string, or contain three dashes before the closing tag (e.g. "− − − >"), so it is suggested to follow that practice even though the VisLang compiler will accept any character between the comment start and end tags.
Additionally, the XML specification describes several special tags for the Prolog, Document Type Definitions, and CDATA. VisLang will accept and ignore these elements through the same mechanism as the comments. Therefore, any XML element starting with "<?" or "<![" and ending with "? >" and "] >" respectively will have all of it's contents ignored. Please follow the W3C XML specification for the full list of characters that should be avoided for this situation.

3.2 Built-In Parts

As discussed previously, VisLang has built-in Parts that are natively understood by the compiler. These parts have specific attributes and special properties that for using them, including a list of inputs that must be used.
Below is the list of standard elements supported by the language, and their required attributes:

Table 3: Accepted Elements

element	input(s)	output(s)	attributes
BLOCK	as defined	as defined	name
REFERENCE	as defined (external)	as defined (external)	name, ref
CONNECTION	none	none	to, from
INPUT	none	provides 'name'	name, datatype
OUTPUT	provides 'name'	none	name, datatype
CONSTANT	none	provides 'name'	name, datatype value
MEM	current	'name'	name, datatype ic
DT	none	'name'	name
NOT	input	'name'	name
AND	input# Note: # > 1	'name'	name
OR	input# Note: # > 1	'name'	name

NOR	input# Note: # > 1	'name'	name
NAND	input# Note: # > 1	'name'	name
XOR	input# Note: # > 1	'name'	name
IF	control, true, false	'name'	name, datatype
COMPARE	lhs, rhs	'name'	name, datatype operation
SUM	input# Note: # > 1	'name'	name, datatype
PROD	input# Note: # > 1	'name'	name, datatype
GAIN	input	'name'	name, datatype value
INV	input	'name'	name, datatype

3.3 Using Built-In Parts

3.3.1 Basic Language Elements

BLOCK: As noted prior, a BLOCK element is a container for other Blocks and/or Parts. The BLOCK element only has a single attribute "name" which is the identifier for that part. All of the valid elements contained within the BLOCK element is considered inside that block, therefore any connections made within that block between parts/blocks can reference any of the elements inside the block as connection points. A block does not need to have Inputs and Outputs defined. However, any Inputs or Outputs found directly inside that BLOCK element will be considered an input or output of that named element for use by other blocks above the named block, or inside other files through the REFERENCE element.

REFERENCE: The REFERENCE element is similar to the BLOCK element, however it has an additional attribute called "ref" that is a reference to a block contained within another file. The block referred to by REFERENCE is then used as if it were contained within the local program in the same way as the BLOCK element would. The REFERENCE element will need connections to any inputs that the referenced block had, again similar to as if that block were contained inside the local program.

CONNECTION: The CONNECTION element is special in that it does not by itself perform a function. The "to" and "from" attributes of this element refer to a connection between the output of one block or part and the input of another. CONNECTION elements must be contained inside

a block or part, and the "to" attribute must reference that block or part's inputs. The "from" attribute can reference the output of any block or part within the same level of the block that the "to" attribute refers to.

INPUT: The INPUT element is used as the input to a block element. It has a "name" attribute, which is an identifier that can be used in any connection at the current block level. The INPUT element does not have a connection inside it as it is considered a terminal for the block it is defined in. The "datatype" attribute refers to Datatype of that identifier. Datatype can either be a basic datatype (e.g. boolean, uint32, single, etc.) or it can be a reference to a structure type. Any connections made to the input must match its datatype to successfully compile.

CONSTANT: The CONSTANT element is also similar to the INPUT element except that it does not get used as an input to it's containing block. Instead, the CONSTANT element has a "value" attribute, which is a literal matching the type of the the element's "type" attribute. If the literal value does not match the definition of the above scanner regular expression for that type, an error will be thrown at compile time.

OUTPUT: The OUTPUT element is very similar to the INPUT element, the two differences are that it is considered a named output of the block it is contained in and that it requires a connection to be made inside it to a block. All of the rules relating to the attributes of INPUT block apply here as well. The "datatype" attribute of the OUTPUT element is where the compiler first begins it's type checking, so as it traces the connections made from the OUTPUT element all the way back to some INPUT element(s), the corresponding types must match between any intermediary Parts or Block Outputs.

3.3.2 Atomic Parts

Note: All Atomic Parts have a "name" attribute to use as an identifier for making connections to other parts. Unless otherwise specified, the default name for input to a signal input Part is "input" and the default name for an output is "output". All Parts are single output.

MEM: The memory block creates a unit-delayed signal that can be reused inside the current Block, usually to solve an algebraic loop concerning the connection of a block. The output value of this block will be the same value of the connection into the block, but only from the previous pass of the generated code. The "ic" attribute describes the value that the MEM element uses for the output on the very first pass of the generated code. The "datatype" attribute is required so that the element knows what the datatype is for it's input and output.

DT: The DT element only provides a signal output called "dt" which can be referenced and used as the delta time between passes of the generated code. This value will always be dynamically updated every pass to reflect the change in time natively. The DT Parts' output is a single precision floating point value.

NOT: The NOT Part provides the logical not of the input as it's output. It does not have any special attributes. The input and output type must be "boolean".

AND: The AND Part provides the logical and of two or more inputs as it's output. The AND Part is defined recursively in that it identifies each input and applies the same operation recursively on each input found. There must be two or more inputs for this operation to work however, or a compilation error will be given. The input and output type must be "boolean".

OR: The OR Part is defined the same as the AND Part, with the exception that the operation is the logical or of two or more inputs. The input and output type must be "boolean".

NOR: The NOR Part is defined the same as the AND Part, with the exception that the operation is the logical nor (not any) of two or more inputs. The input and output type must be "boolean".

NAND: The NAND Part is defined the same as the AND Part, with the exception that the operation is the logical nand (not all) of two or more inputs. The input and output type must be "boolean".

XOR: The OR Part is defined the same as the AND Part, with the exception that the operation is the logical xor (only one or the other) of two or more inputs. The recursive nauture of this definition means that the XOR gate with 3 or more inputs will set it's output true if an odd number of inputs are true. The input and output type must be "boolean".

IF: The IF Part has three defined inputs and performs a conditional operation to switch passing through to the output between two inputs. The "control" input must be a boolean type and is used to control the conditional operation. The "true" input is passed through to the output if the "control" input is set true, otherwise the "false" input is passed through. The "datatype" attribute is required so that the element knows what the datatype is for it's input and output.

COMPARE: The COMPARE Part has two inputs "rhs" and "lhs" and an "operation" attribute that evaulates the conditional statement "lhs operation rhs" and passes the result to the output. "lhs" and "rhs" must match datatype and cannot be the boolean datatype, and the operation

applied has the mathematical result expected. The "datatype" attribute is required so that the COMPARE element knows what the datatype is for it's inputs. It's output is type boolean.

SUM: The SUM Part is similar to the Gate Parts in that there are 2 or more inputs allowed and the function is defined recursively. However, the datatype allowed is either integer or floating point (all inputs must match type). The sum operation is defined as addition between the two or more inputs. Subtraction must take place using the GAIN Part (essentially unary negation) prior to the SUM Part, so that the recursive definition of this function can be used. If the result of the operation would have calculation returned an undefined result (e.g. outside of the bounds provided by the datatype), the result will be unhandled meaning care should be taken to ensure the result can never exceed those bounds.

PROD: The PROD Part is similar to the SUM part, with the only difference being it applies the multiplcation function recursively instead of addition. The same rules apply to the PROD Part as the SUM part otherwise. If division is required, the INV Part should be used prior to the PROD Part in order to invert the input for division.

GAIN: The GAIN Part is a unary operation that multiples the input by literal attribute "value" and returns it as the output of the Part. The input and output will match datatype, and the literal expression for "value" needs to match the datatype of the input in order not to raise an error while compiling.

INV: The INV Part is similar to the GAIN Part, except that the unary operation is inversion of the input's value e.g. division of 1 by that value. Division by zero is handled by outputing the maximum possible floating point value, so care must be taken to ensure the input value is never zero to avoid this behavior.

4 Project Plan

Since I was working on this project alone, there was more autonomy in creating the language. This actually led to be a bit of a problem as my initial ideas for what I wanted to accomplish were unrealistic and I was more willing to slide on the schedule I set for myself since there were no other group members to act in the project manager role to keep things on schedule, nor were any group members available to ensure that project goals were reasonable. Regardless, after an initial development period of over a month a working front end was developed leveraging the XML specification available online with the planned tags and attributes I had at the time. It was eventually decided that adding my own namespace for XML tags would be necessary to reduce the processing load in the scanner and parser section to work with other elements. This is right around the time the XML Abstract Syntax Tree was fully developed and work started on the backend of the compiler.
At this point, there was little testing in existance since I was just attempting to parse the example program, so testing had to be approached. It was decided I was going to leverage to bash testing script from the MicroC example language provided in class, and have additional python scripts be created leveraging the Ctypes module to test the functionality of each test case. Once this was decided, the first MWE test case was developed (the buffer test case) and more work was done to get that to pass. More complicated test cases led to a decision to add a complete block parsing algorithm in order to be able to produce a correctly formatted program for code generation. After some development, this algorithm allowed more test cases to pass and work to continue on integrating block group and referencing functionality. Once this was completed, the initial draft of VisLang was considered feature complete, and several other planned features were descoped due to time constraints on the project.

4.1 Software Development Environment

Development for the project took place entirely on an Asus Chromebook C720 using crouton to enable a full linux environment. The tools used for this project are listed below:

- ubuntu 14.04.2 LTS (Operating system environment)
- git 1.9.1 (source code, test, and documentation configuration management)
- vim 7.4.52 (general purpose text editor)
- ocaml 4.01.0 (including ocamlyacc and ocamllex)
- gcc 4.8.2 (compiling generated code)
- python 2.7.6 (scripting language for testing compiled C objects)

4.2 Project Timeline

- 2015-05-27 Decided on Simulink-like block language, using XML syntax
- 2015-05-31 Created example program
- 2015-06-12 Proposal Submitted
- 2015-06-21 First draft of scanner
- 2015-06-26 First draft of parser
- 2015-06-30 Scanner working for all attributes and tags
- 2015-07-04 LRM Submitted
- 2015-07-08 Parser working for new ast
- 2015-07-09 Added top level
- 2015-07-10 Moved errors to their own module
- 2015-07-14 XML ast working
- 2015-07-17 Integrated blockification function
- 2015-07-24 Removed interpreter
- 2015-07-27 Simplified blockification process
- 2015-07-28 Moved trace algorithm from blockify to it's own module
- 2015-08-05 Working Code Generation for all atomic parts
- 2015-08-12 Got blocks completely working end-to-end
- 2015-08-12 Updated blockification for Reference part
- 2015-08-13 Began working on paper
- 2015-08-14 Submitted paper
- 2015-08-15 Celebrated from Canada

4.3 Project Log

```
2015-08-14  Updated some of the old descriptions for things
2015-08-14  Cleaned up old comments in these files
2015-08-14  Managed to get code listings to appear the way required. Bonus:
            fixed bug with underscore display
2015-08-14  Shorted test case name so it fit in the table
2015-08-14  Redid sections for this part
2015-08-14  Updated tutorial file
2015-08-14  Cleaned up LRM. Made some minor modifications to other parts
2015-08-14  Updated test plan with table any other discussion
2015-08-14  Calling top makefile's clean rule to clean up after testing
2015-08-14  Removed generated C files for example from tracking. Added option
            to ignore tests with i-* prefix.
2015-08-14  More verbage
2015-08-14  Forgot to update this generated file too
2015-08-14  Massaged white space in generated code to pretty print for
            documentation
2015-08-14  Typo in label name
2015-08-14  Removed boxes around code listings. Added more TODO's
2015-08-14  Added C generated files for example into repo for documentation
            purposes. Removed C files from gitignore.
2015-08-14  Updated project plan and got pretty log working
2015-08-14  Updated so all filenames get read
2015-08-13  Added gibberish case to list of cases
2015-08-13  Added gibberish test case to show that VisLang is tolerant of
            random crap
2015-08-13  Added a rule to fix bug with no reduction possible if there is junk
            inside a tag without any other vl elements inside it
2015-08-13  Moved example files up a level now that simavr is gone
2015-08-13  Removed simavr submodule
2015-08-13  Removed simavr submodule
2015-08-13  Removed simavr submodule
2015-08-13  Added todo note
2015-08-13  Added some stuff to talk about, TODO tag
2015-08-13  Added test case listings. Rearranged appendix sections.
2015-08-13  Removed unnecessary code
2015-08-13  Modified timer test case to be a symbolic link to the example file
            instead of a separately maintained file
2015-08-13  Removed creating link to pyg file as it was causing more trouble
            that it's worth
2015-08-13  Finished with conclusion
2015-08-13  Updated git log print out so that it pretty prints only date and
            message, and limits output to 80 chars
2015-08-13  Removed unused attributes
2015-08-13  Added wrapfig package
2015-08-13  Completed architecture page
2015-08-13  Ignore generated C files in example
2015-08-13  No longer automatically re-creating pygment link. Seemed to stop
            allowing it to code
2015-08-13  Made some more updates. Added architecture figure
2015-08-13  completed intro
2015-08-13  Added statusing to all files for quick review. Added some stuff to
            project plan and conslusion
2015-08-13  Added date/time stamp to test log
2015-08-13  Added date and time stamp to log file
2015-08-13  Abandoned find/replace attempt as luacode wasn't working. Also gave
            subtitle to paper
2015-08-13  Now utilizing macro for appearance. Added makefile and test script,
            which isn't working
2015-08-13  Removed additional excess rules and non functional characters
2015-08-13  Removed unnecessary lines
2015-08-13  Added line to clean intermediates created from compiling final
            report
2015-08-13  Working on paper, added some macro and code listings
```

```
2015-08-13  Cleaned up line endings so they are 81 chars or less for pretty
            printing
2015-08-13  Added default rule
2015-08-12  Trying to get it to compile
2015-08-12  Initial version document
2015-08-12  Architecture belongs in report
2015-08-12  Turned proposal into introduction
2015-08-12  Updated for cleaning compiler directory too, and running make
            correctly in src folder
2015-08-12  Added rule to clean up pyg files from minted
2015-08-12  Updated dependancy file now that we're parsing files in blockify
2015-08-12  Updated autotest script because it was not accurately reporting
            errors when failure test cases passed compilation
2015-08-12  Added test to check that if a different type of attribute is in a
            connection, it will fail
2015-08-12  Added test case to show 'name' missing will fail parsing
2015-08-12  Updated test case to capture all of the operations allowed
2015-08-12  Reorganized a bit. Cleaned up comments and code. Added divide by
            zero protection for inverse operator
2015-08-12  Cleaned up comments and removed commented out dead code 'dead code'
2015-08-12  Updated so that vlcc will smartly write out code to file in only
            certain situations
2015-08-12  Removed unsupported tags and bitwise operator functionality
2015-08-12  Removed bitwise operator functionality
2015-08-12  Removed bitwise operator functionality
2015-08-12  Modified gates test case to show that 3 inputs can be handled
2015-08-12  Added xml header to all test files
2015-08-12  Added test case to test algebraic loop detection
2015-08-12  Added testing support for referenced files
2015-08-12  Added include file for reference blocks
2015-08-12  Incorrectly had hi instead of lo for case when input = 10
2015-08-12  Updated blockification such that reference models work okay
2015-08-12  Needed double backslash to avoid warning
2015-08-12  General cleanup. Fixed reference class. Added some additional
            information to errors
2015-08-12  Reworked test script to use vlcc with direct file I/O
2015-08-12  Added ability to call vlcc with or without direct file I/O
2015-08-12  Forgot that comparision objects require datatype
2015-08-12  Removed optimization module (that was doing nothing). Reorganized
            block object in prep for reference obj
2015-08-12  Removed all references to scope and size attributes. They will not
            be a part of the compiler from now on.
2015-08-12  Added new test case testing the reference block
2015-08-12  Moved existing check if pass only test cases to have prefix. Added
            testing for pass-only files
2015-08-12  Old files didn't have vl: namespace
2015-08-12  Added more conditional code generation for inputs, outputs, inner
            blocks of block
2015-08-12  Added extra space to code-gen trailer
2015-08-12  Needed to have GCC create 'Position-independant code suitable for a
            shared library' before compiling object file to a shared
            library
2015-08-12  Reworked so that operation was applied recursively. Ended up
            solving the problem with the disapearing inputs in the
            top block
2015-08-11  Made mistake specifying outputs. Added extra code to make generated
            code disappear for blocks with no outputs/inputs
2015-08-10  Attempt to integrate blocks-in-blocks
2015-08-10  Removed body printing for block objects because we don't want to
            print the body in the function definition, only in
            places it's used
2015-08-10  Removed print statement that was messsing things up
2015-08-10  Got it using compare function that was still there!
2015-08-10  bumbling with print statements
2015-08-10  removed commeneted out code
```

```
2015-08-10  got it to parse with the tuple list
2015-08-10  Working on block update
2015-08-09  Filtering out connection objects from blockification for inner
            blocks
2015-08-08  Removed checking for combo blockref/input because we are searching
            for blocks, not inputs
2015-08-08  Added simple block in block test program
2015-08-07  Added test case for timer
2015-08-07  Also need to exclude terminating blocks if they were in the current
            trace list as that is okay. Discovered by having a
            constant that was used in two places on the same path,
            which is only fine because it's a terminator and not
            used to calculate anything
2015-08-07  Added IF and DT parts
2015-08-07  Updated dt block to 100ms
2015-08-06  Added timer complicated test case from example file, also updated
            that example file
2015-08-06  Bad test result for inv of case 1: 1 not 0.2 is 1/1. Also,
            apparently -1*0 = -0 for ctypes, so added minus sign
2015-08-06  Bug in program such: used two instead of 4, thereby printing
            infinite for the divide by zero
2015-08-06  Commented out dead code parsing because it was printing to
            generated code
2015-08-06  Added gain and inverse blocks
2015-08-06  Added INV and GAIN parts to test case. Note: outputs are in
            reverse, and we avoid -4 to prevent div/0 fault
2015-08-06  Modified to have both addition and multiplication parts
2015-08-06  Reanmed addition module to also check out other math-y operations
2015-08-06  test case and results were in opposite order
2015-08-06  Corrected bug where blocks that terminate would be added to the
            list of objects multiple times if they were split off
            from each other because there was no check if they were
            in the priors list
2015-08-06  test case revealed not all gate parts were implemented
2015-08-06  Added gate logic test case
2015-08-06  Added failure test case utilities
2015-08-06  added vl: to the beginning of block
2015-08-06  made file ignore errors when removing intermediates
2015-08-06  Needed to modify python script a bit to proper parse through output
            structure and pretty-print any type properly
2015-08-05  'const' was a reserved keyword for C
2015-08-05  constant class print name was 'constant' not 'const'
2015-08-05  Added compare class. Enabled constant class to print itself as
            static in header
2015-08-05  Modified blockification for explicitly setting datatypes for parts
            that require it
2015-08-05  Modified test cases due to bugs
2015-08-05  Made booleans integer type for now
2015-08-05  Added datatype when necessary to disambiguiate for compiler
2015-08-05  Added datatype printout to io_part
2015-08-05  Removed datatype stub that was pointless
2015-08-05  Updated file such that error is raised when datatype is unset at
            compilation time.
2015-08-05  Applied reorganization such that atomic parts are referenced by
            name and not by reference to their outputs
2015-08-05  Incorrectly stated buffer instead of block name
2015-08-05  Had to reorganize such that you do not mention atomic parts by
            reference, only by name
2015-08-05  Got it working such that atomic parts print correct code.... almost
2015-08-05  Got the correct input names to appear
2015-08-05  Took care of little printing bug due to printing the body of input
            objects
2015-08-05  Switched order of dead code print for top because we already set
            the objects to match the pruned inner objects
2015-08-05  Solved bug with memory appearing twice. Terminate from trace when
```

```
            memory occurs only after verifying memory is an input to
            the trace. Additionally, when returning the trace list
            due to the part already occuring on the list, do not
            include that part in the trace list.
2015-08-05  Added printing in header for memory blocks
2015-08-05  Added dead code print
2015-08-05  Found bug where trace list was being built up in trace_split
            instead of trace function
2015-08-05  Some reorginzation, added comments, fixed bug with initializing
            trace_start for top
2015-08-05  Moved OR gate from before AND gate to after to visualize the
            algorithm correctly moving the OR gate where it needs to
            go in the list
2015-08-05  Moved get connection from external function to internal method of
            base class
2015-08-04  Made block inner objects work in the right order
2015-08-04  Removed scoping attribute
2015-08-04  Newline to trailer
2015-08-04  Restructuring of program flow for list of blocks
2015-08-04  Attempt to re-engineer blockification such that more fine grained
            control of blocks are found
2015-08-04  Turns out I didn't want to seperate these
2015-08-04  Setting inner objects before appending to list
2015-08-04  Redid function such that a list of blocks is returned instead
2015-08-04  working here...
2015-08-04  Modified memory class, added gates and other parts
2015-08-04  Slight reorganization
2015-08-04  Forgot scope in input
2015-08-03  Some errors with tests
2015-08-02  Updated regexps for all tags and attributes
2015-08-02  Added data sets for test cases
2015-08-02  Added some more tests
2015-08-01  Got testing to work
2015-08-01  Got helper code generation working
2015-08-01  Working on getting it to create test code
2015-08-01  No more interpreter
2015-08-01  Ignore generated C files
2015-08-01  Gave explicit datatype to top level ports
2015-07-30  Renamed bytecode module to blockparse
2015-07-30  Renamed bytecode module as it doesn't produce bytecode
2015-07-30  Reorganized code such that block tree gets produced by bytecode,
            and compile prints objects. Also added optimization
            layer.
2015-07-29  Managed to get algorithm working. Currently is not outputting
            inputs though
2015-07-29  Used wrong error function
2015-07-28  Added errors for using methods that trace algorithm should not try
            and do
2015-07-28  Added memory class
2015-07-28  Bytecode print function needed to correctly print class
2015-07-28  Moved trace algorithm from blockify
2015-07-28  Renamed bytecode methods
2015-07-28  Added missing virtual methods
2015-07-27  Simplified blockification process. Moving the trace function to
            bytecode production
2015-07-26  Updated dependancy file
2015-07-26  a little more reconfiguration. need to implement bytecode
2015-07-25  Updated such that block instantiation calls new block trace
            function
2015-07-25  Added input to block instantiation for blockify function to removed
            cyclic dependancy on that function to use it inside the
            block function for creating it's inner objects
2015-07-25  Updated dependancies for top level
2015-07-25  added dummy code for printing object for bytecode
2015-07-25  Modified top level for current structure of compiler
```

```
2015-07-25  dummy code for now
2015-07-25  Forced coercion of return type for blockify function to base class
2015-07-25  Needed to modify order of objects for compiling
2015-07-25  Added note to remind that there is some extra actions needed for
            blockifying a block
2015-07-25  Removed bot.ml
2015-07-25  Merged bot back into blockify. Decided to make block inner objects
            mutable and try it that way
2015-07-25  Modified clean rule to clean mli intermediates
2015-07-25  Still working on this one. does notgenerate mli file or compile
2015-07-25  Added input and output listing
2015-07-25  Forgot to rename parser include in scanner
2015-07-25  Added ability to generate interface files
2015-07-25  Added mli files to ignore list because I have no formal interfaces
2015-07-25  Deleting intermediate that was committed by mistake
2015-07-25  updated objectification functionality for new structure
2015-07-25  Updated dependancy file for new structure
2015-07-25  Updated parser for new approach
2015-07-25  Simplified method of doing scanner
2015-07-25  Renamed front end files to have prefix x, update intermediates
2015-07-25  Made type attribute into datatype
2015-07-25  Added xml namespaces for conflict management
2015-07-25  Renamed pre-parsing step to denote xml
2015-07-25  Reorg of files
2015-07-25  Continued attempt at making a working blockification method
2015-07-24  Updated dependancy file
2015-07-24  Removed interpreter
2015-07-24  added 'auto' to datatype
2015-07-24  Committed current updates. not building
2015-07-24  added the simplest possible test case
2015-07-17  Added semantic checking syntax tree
2015-07-17  Integrated blockify
2015-07-17  Added Block Parsing error fo blockify
2015-07-15  Typo
2015-07-14  Added option to run 'pass-*' tests to show that compilation simply
            worked
2015-07-14  Added option to clean up test intermediates
2015-07-14  Added test case to show unended blocks are caught
2015-07-14  Added test case to check that cascaded empty blocks work okay
2015-07-14  Added empty block for testing ast
2015-07-14  Finally got ast working by adding name attribute to the list of
            allowed values, which I forgot before
2015-07-14  Added check in parser to ensure tokens for open and close tag match
            when reduced using parser.
2015-07-13  Added test intermediates to ignore file
2015-07-13  Made master make file
2015-07-13  Made bash script executable
2015-07-13  Added first test and script copied from mc
2015-07-12  swapped incorrect tags for datatype and scope
2015-07-11  Reformatted how errors are processed for Parsing error to be
            correct
2015-07-11  redid dependency file for make
2015-07-11  didn't want to show anything after main compilation rule
2015-07-10  Made some of the make rules quiet
2015-07-10  Played with error messages a bit. Added ignore to remove compiler
            warning on returned type
2015-07-10  Moved errors to their own module
2015-07-09  Added vislang's top level
2015-07-09  Main compilation rule was only using first target, not all targets
2015-07-09  Added helper functions for debugging ast
2015-07-09  Made a typo on the source for the top level
2015-07-08  Removed test binaries
2015-07-08  Changed name of ast.mli to ast.ml so it can have code for
            reproducing the ast
2015-07-08  Decided against having seperate binaries for testing. Making
```

```
            special modes to dumb intermediate results to terminal
            in main vislang program.
2015-07-08  Updated scanner to work with explicit operator tokens. Updated
            parser such that open tag is one token
2015-07-08  Reconfigured parser and ast to get it to work!
2015-07-07  Created new rules for test builds. Added those builds to ignore
            list
2015-07-07  Updates for scanner so that it will work with parser. Also added
            CDATA style tags as a comment to skip those.
2015-07-04  Completed draft of language reference for homework
2015-07-04  Added nice to have section
2015-07-04  Added to do list
2015-07-04  changed reference attribute to ref to be shorter
2015-07-04  Redid connections to be inside parts and blocks
2015-07-03  Renamed block with reference attribute into it's own block
2015-07-03  Renamed IC attribute to a shorter name
2015-07-02  So close!
2015-07-02  Added secondary rule so that they don't get deleted
2015-07-02  Some silliness with pretty printing
2015-07-02  Valid dependency file NOT generated by ocamldep
2015-07-02  Updated makefile from template. Added dependency file, which sucks
2015-07-02  Merge branch 'xml'
2015-07-02  Saving draft for idea switch
2015-07-02  Added intermediate of parser output to ignore
2015-07-02  Added some extra notes
2015-07-02  Attempt at parsing more generically
2015-07-02  Modifcations made to scanner after discovering built-in int2str
            conversion functions work with the relevant prefixes
2015-07-01  Merged gitignore file and scanner from parser branch. Going to
            create a second parser branch
2015-07-01  Trying something new, so committing what I have so far
2015-07-01  Removed test build of scanner, added parser intermediates
2015-07-01  First draft attempt at ast
2015-06-30  Added scope and renamed connection to reference
2015-06-30  Added missing datatype attribute value
2015-06-30  Added a better comment
2015-06-30  Modification made for inner attrbiutes. Not perfect
2015-06-30  Now explicitly recursing back to top level tokenizing when an
            unfinished tag appears
2015-06-30  Made really good progress on scanner. Can now parse all tags and
            attributes with relevant values.
2015-06-29  Removed Program block, redid file extension to .vl from .vs,
            removed device level I/O, other syntax errors
2015-06-29  Some bugs with regex to do with literals and files, also bug to do
            with line counting for error function
2015-06-28  You need to explicitly tell Lexer a new line occured
2015-06-28  Managed to get error working
2015-06-28  Modified doc makefile and added pyg elements to ignore list
2015-06-28  Added simavr submodule along with readme for installation purposes
2015-06-26  Finally got parser to work for blocks!
2015-06-26  Syntax error. Type was not in quotes. Caught by lexxer!
2015-06-26  Still drafting. Added TODO with issue
2015-06-25  added some style guide info, etc.
2015-06-25  Managed to get it working, displays error on line 1
2015-06-21  Second attempt at writing fully functioning xml scanner
2015-06-21  Restructured makefile a bit
2015-06-21  Added ocaml intermmediate files to ignore
2015-06-21  First draft of scanner
2015-06-19  Added swap files from vim and intermeddiate file for scanner to
            ignore file
2015-06-19  Renamed extension to .vl from .vs
2015-06-17  Setup files for starting lexxer and parser
2015-06-12  Copied language reference table from proposal and added simulation
            components to it
2015-06-12  Final draft of proposal
```

```
2015-06-12  Updated scope and type to be string attributes
2015-06-12  Updated signal types to have size, made values into a string
2015-06-11  Added syntax chapter
2015-06-11  renamed some connections for brevity
2015-06-11  Reconfigured example such that connections are external to blocks,
            and now using the | operator to denote a member of a
            name
2015-06-10  Updated example to have referenced connections instead of explicit
2015-06-09  Added Makefile for building documentation
2015-06-09  Removed pdf and added to gitignore because we discovered how to
            change the output directory
2015-06-09  Updated listings to listingsutf8 to print special chars
2015-06-09  Updated PDF to try and get source files to work
2015-06-09  Updated proposal
2015-05-31  Moved example project to it's own directory to make room for other
            examples
2015-05-31  Added example program
```

5 VisLang Compiler Architecture

The architecture of VisLang has two distinct stages of operation from source file to target file. The scanning and parsing stages of the front end essentially implement read the XML elements of interest and skip through parsing any unrecognized tokens. After a correctly formed XML Object Tree has been formed, the next step is to translate that tree of XML Objects (an XML Object has a tag, a list of attributes and a list of inner objects, if any) into a block tree where each block can verify and access the necessary attributes it should have. Each object can also see the list of connections assigned to it when it was parsed, which is important when verifying the program is well-formed. That block tree is then taken and re-organized such that the inner objects of a block are in Static Single Assignment form, e.g. each block can be computed using the outputs of previous blocks in the list for that containing block.

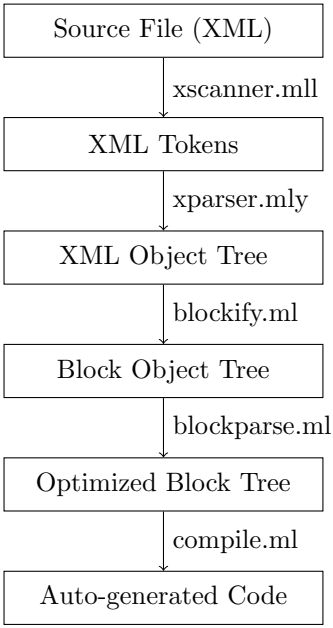

Figure 1: VLCC Architecture

In the process of reorganizing the inner blocks, the Block Parse algorithm will also perform the check that the inner blocks align (e.g. they call blocks that are properly assigned, they match in datatype, etc.) and that only inner blocks which are used to compute the output are in the calculation. Any blocks which do not align will raise an error (datatype mismatch, incorrectly attributes for that object, etc.), any blocks which reference other blocks in a circular fashion will raise an error (algebraic loop), and any blocks that are not necessary to compute an output will be optimized away. The end result is that the remaining optimized block tree is a suitable candidate to be directly translated into generated code as that generated code will have the property of minimal side-effects: all computations are computed either from inputs or derived from inputs.

6 Test Plan

The generated target code for Listing 1 and Listing 2 from the Tutorial are below:

Listing 3: Generated Code for Top Level

../example/timed-blinking-light.c

```c
#include <stdbool.h>
#include <stdint.h>
#include <float.h>
#include <math.h>

#include "./timer.c"

/* I/O Structures for block timed_blinking_light */
struct timed_blinking_light_in {
        bool digital_input_1;
};

struct timed_blinking_light_out {
        bool digital_output_1;
};

/* Initialize static variables */
static bool count_expired_lp = false;
static float_t time = 2.;

struct timed_blinking_light_out
/* Function def */ timed_blinking_light(struct timed_blinking_light_in inputs)
{
        /* Inputs for block timed_blinking_light */
        bool digital_input_1 = inputs.digital_input_1;

        /* Body for block timed_blinking_light */
        bool not_di_1 = !(digital_input_1);
        bool reset_blink = not_di_1 || count_expired_lp;
        struct timer_in timer_instance_1_inputs = {
                .time = time,
                .reset = reset_blink,
                .start = digital_input_1
        };
        struct timer_out timer_instance_1_outputs =
                timer(timer_instance_1_inputs);
        bool digital_output_1 = timer_instance_1_outputs.count_expired;
        count_expired_lp = timer_instance_1_outputs.count_expired;

        /* Outputs for block timed_blinking_light */
        struct timed_blinking_light_out outputs;
        outputs.digital_output_1 = digital_output_1;

        return outputs;
}

/* Generated using VLCC */
```

Listing 4: Generated Code for Referenced Block

../example/timer.c

```c
#include <stdbool.h>
#include <stdint.h>
#include <float.h>
#include <math.h>

/* I/O Structures for block timer */
struct timer_in {
        bool reset;
        bool start;
        float_t time;
};

struct timer_out {
        bool count_expired;
        float_t elapsed_time;
};

/* Initialize static variables */
static bool count_expired_lp = false;
static float_t elapsed_time_lp = 0.;
static float_t zero_constant = 0.;
static float_t time_since_last_pass = 0.1;

struct timer_out
/* Function def */ timer(struct timer_in inputs)
{
        /* Inputs for block timer */
        bool reset = inputs.reset;
        bool start = inputs.start;
        float_t time = inputs.time;

        /* Body for block timer */
        bool count_not_expired = !(count_expired_lp);
        bool start_enb = start && count_not_expired;
        float_t increment_value = (start_enb) ?
                (time_since_last_pass) :
                (zero_constant);
        float_t summer = increment_value + elapsed_time_lp;
        float_t reset_switch = (reset) ?
                (zero_constant) :
                (summer);
        float_t elapsed_time = reset_switch;
        bool is_count_expired = (elapsed_time >= time);
        bool count_expired = is_count_expired;
        elapsed_time_lp = elapsed_time;
        count_expired_lp = count_expired;

        /* Outputs for block timer */
        struct timer_out outputs;
        outputs.count_expired = count_expired;
```

```
51          outputs.elapsed_time = elapsed_time;
52
53          return outputs;
54  }
55
56  /* Generated using VLCC */
```

The output program, when compiled using gcc, will be able to process the inputs provided by 'connecting' to that block and update it's outputs over time for every iteration of the program in the main loop. For programs without MEM or DT blocks, the resulting code has the property of being time-invariant, that is no matter how many times it is called or whatever the duration between calls are, it will produce the exact same result every time. The MEM element will remember a value between the last call and the current such that the resulting program loses that time invariance, but this operation allows the production of functionality such as states and transfer functions to be modeled using VisLang. The DT element is used when the amount of time between calls is important, but for a steady system this should never be an issue as it should stay relatively constant. This means programs using DT may or may not be almost time invariant, but that depends on the usage of the block.
To automate testing of VisLang programs, a shell script (Listing 14) was borrowed from the MicroC example language. The shell script looks at all of the files in a directory and processes them into 1 of 3 testing groups: test, pass, fail. The pass and fail testing groups simply looks to verify that the source file for such a test case either pass compilation (pass cases) or fails compilation (fail). In this way, specific compiler features that have to do with processing the input file (instead of the code generated) can be checked without further complication. The 'test' cases first verify that the source file can generate the target file, but additionally a functional check is provided through associated *.in and *.out files that are run against the target file.
The methodology for testing these cases involves additionally compiling the target files as source files for gcc, and turning the resulting object file into a shared library that can be interpreted through a testing script. The testing script is a python script that is generated using the -d option of vlcc which takes the *.in file and runs a while loop over each line of the file and produces what the output of the program would be for each timestep. The timestep is purposely never updated to ensure that a repeatable test environment exists. The output produced by the test script is then compared against the associated *.out file to see if any differences exist. If the two files match, then the test is determined to be passing.

6.1 Test Case List

The following is a list of the test cases used to verify the VisLang compiler produces correct code:

Table 4: Test Case Descriptions

Test Case	Description
Algebraic Loop Failure Case Source Code: Listing 15	Shows that an algebraic loop is caught
Bad Connection Failure Case Source Code: Listing 16	Shows that a badly specified connection is caught
Missing Attribute Failure Case Source Code: Listing 17	Shows that a missing attributes in a block is caught
Unended Block Failure Case Source Code: Listing 18	Shows that an unended block is caught
Empty Block Completion Case Source Code: Listing 20	Shows that an empty block compiles okay
Cascaded Blocks Completion Case Source Code: Listing 19	Shows that multiple empty blocks inside each other are okay
XML Tolerance Case Source Code: Listing 21	Shows that random XML and other input is okay between tags
Buffer Value Test Case Source Code: Listing 22	Shows proper operation of buffer block (O = I)
Buffer in Buffer Value Test Case Source Code: Listing 23	Shows that a block within a block works
Memory Block Test Case Source Code: Listing 28	Shows the memory block works okay
Comparision Operation Test Case Source Code: Listing 24	Shows all the comparision operations work
Logical Gate Test Case Source Code: Listing 25	Shows all the logical gates work
Math Operations Test Case Source Code: Listing 27	Shows all math blocks work okay
Reference Block Test Case Source Code: Listing 26	Shows a reference block works okay
SR Latch Complexity Test Case Source Code: Listing 29	Shows that a complex block (SR Latch) works
Timer Complexity Test Case Source Code: Listing 30	Shows that example (Timer block) works

7 Conclusion

The VisLang compiler was moderately a success because it lays the groundwork for future iterations of the program for use in a fully optimized environment as a replacement for developing embedded programs using proprietary IDEs or programming languages that are more difficult to understand. A variety of lessons where learned during the development of the program that will be detailed below. As a result of several of the lessons learned, suggestions for future development are also presented.

7.1 Lessons Learned

The original idea for VisLang was very ambitious: to make a general purpose embedded computing language in a visual format that could be used to develop programs for small embedded devices such as the Aruduino platform. Early on the in the project it was realized that this is much to ambitious of a goal because that would mean essentially replicating the avr C libraries in a language that was not meant for it. Instead, the scope of VisLang was first pared down such that VisLang instead generated C code instead of assembly so that it would be easy to link with the already feature- complete libraries that exist for the platform. Integration with the avr libraries remains untested at this time, but it is easy to show how VisLang-generated code could be easily integrated into while loop almost any embedded device utilizes to run code. The thought is that the specialty code needed to interface with the device is only a small portion of the overall code the user is interested in running, so such a tradeoff would be acceptable.

Another lesson that was learned a few weeks into development of VisLang was that testcase driven development would be required to move forward at an acceptable pace. Originally, the development philosophy was trying to implement all of the required features of the language at once, but even trying to get the simplest program (a buffer block, which passes input directly to output) was a difficult task and a philosophy change was needed. A test case for the buffer block was written and the compiler was made to work appropiately with that test case first before further test cases were developed and more functionality was implemented to meet those new cases. The benefits of this approach primarily are that these test cases are available for quick turnover later on to validate future changes to the code. This happened several times where a change made to satisfy one test case ultimately did, but broke several of the already completed test cases. Integrating test cases into development was one of the biggest lessons learned at first.

Finally, the last lesson that was learned was to complete more preliminary work before creating a specification for a language. Knowing how much would be possible as well as prototyping some of the features beforehand would have helped to write a much more sound specification to begin with so that scope reduction

and philosophy changes would be minimized.

7.2 Future Improvements

Several pieces of VisLang's original specification were descoped for the initial version of the compiler due to time constraints. Given future development time, most of these features would be required for VisLang to reach it's full potential as a general purpose prototyping and embedded controller language to match potential rivals such as Simulink and Modelica.

First off, a graphical interface for manipulating VisLang code would make development of programs in the language much easier, since that was the original intended use-case. Significant development would be necessary here, but thankfully true to the original design goals development to VisLang and any GUI environment that might use the language could happen mostly in parallel. Specific attention would need to be taken to overhaul VisLang's front end to make it truly resilient to non-VisLang XML elements. As it stands right now, VisLang supports ignoring additional attributes, but using attributes of the same name can confuse the parser which will throw an error. Either an alternative way to specify VisLang attributes would need to be attempted, the VL Compiler would need to be hardened against those attributes by more clever design of the front end, or a better methodology of specifying the XML would need to be investigated to satisfy this goal.

Arrays and Structures would be essential to truly allowing the language to prosper in all of its intended use cases. Originally, the Array features of VisLang would allow a user to create and pass around Arrays to inputs, enabling Function Language elements such as Filter, Reduce, and Map to be applied so duplicate functionality can be performed with minimal coding. This is important to larger embedded devices because they typically have redundant interfaces that require the exact same processing to each element. Additionally, digital busses can be arrays of packet structures that need the exact same processing where a language that operated on them in parallel would be able be more efficient in its operation. Structures would be used in a similar way, enabling I/O messages to be stripped apart and processed in a predictable way, or output messages to be created in a specified manner.

Of course, a block language like VisLang can always support more parts. The original specification for VisLang included several parts that were deemed unnecessary for the initial implementation of the compiler, so identifying and adding that functionality would be an obvious next step for the language. Adding the ability to encapsulate or link to arbitrary code would also be another possible design goal for VisLang as often it is necessary to have a calculation drive some action that interfaces with the embedded processor, such as servicing the watchdog timer or managing interrupts. This would be important if VisLang were to be used on larger projects.

A VLCC Source Code

Listing 5: Top Level

../src/vislang.ml

```
open Xscanner
open Xparser
open Blockify
open Blockparse
open Compile

type action = BlockTree | Compile | DebugCode
type rwfile = File | StdIO

let _ =
    let action = if Array.length Sys.argv > 1 then
        List.assoc Sys.argv.(1) [ ("-b", BlockTree);
                                  ("-c", Compile);
                                  ("-d", DebugCode)]
    else Compile
    and rwfile = if (Array.length Sys.argv > 2) then File else StdIO in

    let filein =
        match rwfile with
            File  -> (open_in Sys.argv.(2))
          | StdIO -> stdin
     in
    let lexbuf = Lexing.from_channel filein in
    let xml_tree = Xparser.xml_tree Xscanner.token lexbuf in
    let block_tree = Blockify.parse_xml_tree xml_tree in
    let program = Blockparse.block_parse block_tree in
    let listing =
        match action with
              BlockTree -> Blockparse.print_list program
            | Compile   -> Compile.translate program
            | DebugCode -> Compile.gen_debug_code program
    in let write_out_with_ext ext = output_string
                                        (open_out
                                            (Str.global_replace
                                                (Str.regexp "\\.vl")
                                                ext
                                                Sys.argv.(2)
                                            )
                                        )
    in match (rwfile, action) with
        (* Only print out to a new file if we are compiling or making debug
         * code with an input file, else print to screen if standard input
         * is used or we are printing the blocktree *)
           (File, Compile)   -> write_out_with_ext ".c"  listing
         | (File, DebugCode) -> write_out_with_ext ".py" listing
         | ( _ , _ )         -> print_string listing
```

Listing 6: XML Scanner

../src/xscanner.mll

```
{
    open Xparser
    open Errors
}
(* Main definitions for use below *)
let ws      = [' ' '\t']
let nl      = ['\r' '\n']
let tag     = ( "BLOCK"
              |   "REFERENCE"
              |   "INPUT"
              |   "OUTPUT"
              |   "CONSTANT"
              |   "MEM"
              |   "DT"
              |   "NOT"
              |   "AND"
              |   "OR"
              |   "NOR"
              |   "NAND"
              |   "XOR"
              |   "IF"
              |   "COMPARE"
              |   "SUM"
              |   "PROD"
              |   "GAIN"
              |   "INV"
              |   "CONNECTION"
              ) (* all accepted tags *)
let attr    = ( "name"
              |   "ref"
              |   "datatype"
              |   "to"
              |   "from"
              |   "ic"
              |   "operation"
              |   "value"
              ) (* all accepted attributes *)
let name    = ['A'-'Z' 'a'-'z']['A'-'Z' 'a'-'z' '0'-'9' '_']*
let datatype= ( "auto"
              |   "double" | "single"
              |   "boolean"
              |   'u'? "int" ("8" | "16" | "32") (* all integer types *)
             (*|   name (* for structs *)*)
             )
(* file names acceptable for referencing *)
let file    = ( ".." | ".")? ("/" ['A'-'Z' 'a'-'z' '0'-'9' '_' '-' '.']+ )+ ".vl"
(* Value literals. Used for CONSTANT, MEMORY, and GAIN blocks *)
let sign    = ( "+" | "-")
let boolean = ( "true" | "false")
let digit   = [ '0' - '9' ]
let flt_pt  = sign? ( digit+ "." digit* | "." digit+ )
let hex     = sign? '0' ['x' 'X'] ['A'-'F' 'a'-'f' '0'-'9']+
let oct     = sign? '0' ['o' 'O'] ['0'-'7']+
let bin     = sign? '0' ['b' 'B'] ['0' '1']+
```

```
55  let dec     = sign? digit+ (* Allow signed integers for any encoding *)
56
57  (* Main scanner step: search for elements, attributes, and comments *)
58  rule token =
59      parse
60      (* Comments: Search for any of the following ignored tag openings,
61       * then jump to rule for parsing an ignore anything inside it. *)
62          "<?"   | (* XML Declarators *)
63          "<!--" | (* XML Comments    *)
64          "<!["    (* DOCTYPE Markup  *)
65          as ctype                          { comm ctype lexbuf }
66      (* Elements: Scan for supported blocks and link to parsing stage.
67       * If an unsupported block is found, note it as information for
68       * compilation *)
69          | "<"  "vl:" (tag as t)          { O_ELEM( t )  }
70          | "</" "vl:" (tag as t) ">"      { C_ELEM( t )  }
71          | "/>"                           { E_ELEM       }
72          | ">"    (* No token required *) { token lexbuf }
73      (* Attributes: The following are tokens for different values
74       * attributes might take on. *)
75          | attr as a "="              { ATTR  ( a ) }
76          | "\"" (datatype as d) "\"" { DTYPE ( d ) }
77          (* note: names and files are allowed to have references *)
78          | "\"" (name as n) "\""?     { NAME  ( n ) }
79          | "\"" (file as f) "\""?     { FILE  ( f ) }
80          (* note: a reference always appears as a suffix to a name or file *)
81          | "|"  (name as r) "\""?     { REF   ( r ) }
82          (* Comparision Operators *)
83          | "\"" "==" "\""             { EQT }
84          | "\"" ">"  "\""             { GRT }
85          | "\"" "<"  "\""             { LST }
86          | "\"" ">=" "\""             { GEQ }
87          | "\"" "<=" "\""             { LEQ }
88          | "\"" "!=" "\""             { NEQ }
89          (* Literals *)
90          | "\"" (boolean as b) "\""  { BOOL  ( b ) }
91          | "\"" (flt_pt as f) "\""   { FLOAT ( f ) }
92          | "\"" (hex as h) "\""      { HEX   ( h ) }
93          | "\"" (dec as d) "\""      { DEC   ( d ) }
94          | "\"" (oct as o) "\""      { OCT   ( o ) }
95          | "\"" (bin as b) "\""      { BIN   ( b ) }
96      (* Extras: The following are tokens for other values *)
97          | ws                         { token lexbuf }
98          | nl                         { Lexing.new_line lexbuf;
99                                         token lexbuf }
100     (* This allows anything unsupported to be ignored *)
101         | _                          { token lexbuf }
102         | eof                        { EOF }
103 (* Comment sub-rule: search for matching comment tag.
104  * If a different comment tag type found, then continue,
105  * else return to token scanner.*)
106 and comm ctype =
107     parse "-->" { if ctype = "<!--" then token lexbuf else comm ctype lexbuf }
108         | "?>"  { if ctype = "<?"   then token lexbuf else comm ctype lexbuf }
109         | "]>"  { if ctype = "<!["  then token lexbuf else comm ctype lexbuf }
```

```
        | nl    { Lexing.new_line lexbuf;                    comm ctype lexbuf }
        | _     { (* Skip everything else *)                 comm ctype lexbuf }
```

Listing 7: XML Parser

../src/xparser.mly

```
%{
    open Xst
    open Errors
%}

%token E_ELEM EOF
%token <string> O_ELEM C_ELEM ATTR
%token <string> NAME FILE REF DTYPE
%token GRT LST EQT NEQ LEQ GEQ
%token <string> BOOL FLOAT HEX DEC OCT BIN

%left DTYPE NAME FILE REF
%left BOOL FLOAT HEX DEC OCT BIN
%left O_ELEM C_ELEM ELEM ATTR

%start xml_tree
%type <Xst.xml_obj>      xml_tree
%type <Xst.xml_obj list> xml_list
%type <Xst.xml_obj>      xml_obj

%%

xml_tree:
    xml_obj EOF { $1 }

xml_obj:
      O_ELEM attr_list E_ELEM           { { tagname     = $1 ;
                                            attributes  = $2 ;
                                            inner_objs  = [] } }
    | O_ELEM attr_list C_ELEM           { if $1 <> $3
                                          then xml_parse_error (3)
                                            ("Open/Close element mismatch. " ^
                                             "Element " ^ $1 ^ " <> " ^ $3)
                                          else
                                          { tagname     = $1 ;
                                            attributes  = $2 ;
                                            inner_objs  = [] } }
    | O_ELEM attr_list xml_list C_ELEM  { if $1 <> $4
                                          then xml_parse_error (4)
                                            ("Open/Close element mismatch. " ^
                                             "Element " ^ $1 ^ " <> " ^ $4)
                                          else
                                          { tagname     = $1 ;
                                            attributes  = $2 ;
                                            inner_objs  = $3 } }

xml_list:
```

```
      xml_obj           { [  $1  ] }
    | xml_list xml_obj  { $2 :: $1 }

attr_list:
      attr              { [  $1  ] }
    | attr_list attr    { $2 :: $1 }

attr:
      ATTR value        { { aname   = $1 ;
                            avalue  = $2 } }

value:
      ref               { Ref       ($1) }
    | NAME              { Name      ($1) }
    | literal           {           ($1) }
    | compopr           { Compopr   ($1) }
    | DTYPE             { Datatype  ($1) }

ref:
      FILE ref_list     { { reftype = "FILE" ;
                            refroot = $1     ;
                            reflist = $2     } }
    | NAME ref_list     { { reftype = "NAME" ;
                            refroot = $1     ;
                            reflist = $2     } }

ref_list:
      REF               { [  $1  ] }
    | ref_list REF      { $2 :: $1 }

literal:
      BOOL              { Bool  (bool_of_string  $1) }
    | FLOAT             { Float (float_of_string $1) }
    | HEX               { Int   (int_of_string   $1) }
    | DEC               { Int   (int_of_string   $1) }
    | OCT               { Int   (int_of_string   $1) }
    | BIN               { Int   (int_of_string   $1) }

compopr:
      GRT               { Grt }
    | LST               { Lst }
    | EQT               { Eqt }
    | NEQ               { Neq }
    | LEQ               { Leq }
    | GEQ               { Geq }
```

Listing 8: XML Syntax Tree

../src/xst.ml

```
(* Abstract Syntax Tree Definition *)
type copr = Grt | Lst | Eqt | Neq | Leq | Geq (* Comparison operators *)

type ref  = {
```

```
    reftype     :  string;
    refroot     :  string;
    reflist     :  string list;
}

type value =
      Ref       of ref              (* List of strings leading to a block      *)
    | Name      of string           (* Name of a block                         *)
    | Int       of int              (* Standard int type                       *)
    | Float     of float            (* Standard float type                     *)
    | Bool      of bool             (* Standard boolean type                   *)
    | Datatype  of string           (* datatype from set of types              *)
    | Compopr   of copr             (* Comparision operator                    *)

type attr = {
    aname       :  string;          (* Attribute Name                          *)
    avalue      :  value;           (* Attrbiute Value                         *)
}

type xml_obj = {
    tagname     :  string;          (* Block Name                              *)
    attributes  :  attr list;       (* Dictionary of attribute names and values*)
    inner_objs  :  xml_obj list;    (* List of contained XML objects
                                     * (can be empty) *)
}

(* Helper functions for printing XML AST *)
let string_of_comp_opr v = match v with
      Grt   -> ">"
    | Lst   -> "<"
    | Eqt   -> "=="
    | Neq   -> "!="
    | Leq   -> "<="
    | Geq   -> ">="

let string_of_ref (v) =
      v.refroot ^ "|" ^ String.concat "|" (v.reflist) ^
      " (" ^ v.reftype ^ " REF)"

let string_of_value value = match value with
      Ref       v -> string_of_ref v
    | Name      v -> v
    | Int       v -> string_of_int v
    | Float     v -> string_of_float v
    | Bool      v -> string_of_bool v
    | Datatype  v -> v
    | Compopr   v -> string_of_comp_opr v

let string_of_attr (a) =
    a.aname ^ ": " ^ string_of_value a.avalue

let rec string_of_xml (obj) =
    "Block: " ^ obj.tagname ^ "\n" ^
    "Attributes:\n-" ^
    (String.concat "\n-" (List.map string_of_attr obj.attributes)) ^
```

```
    if obj.inner_objs == []
    then "\n"
    else
    "\n\nChildren:\n" ^
    (String.concat "\n" (List.map string_of_xml obj.inner_objs)) ^
    "\nEnd of Children for: " ^ obj.tagname ^"\n"
```

Listing 9: XML Object to Block Object Converter

../src/blockify.ml

```
open Xst
open Errors
(* Helper functions for Object instantiaion *)
let get_attr attribute xml_obj =
    let attr = List.filter (fun x -> x.aname = attribute) xml_obj.attributes in
        match attr with
            []      -> object_error ("No attribute named " ^ attribute ^
                                     " in:\n" ^ (string_of_xml xml_obj))
          | [a]     -> a.avalue
          | _ :: _  -> object_error ("Too many attributes named " ^ attribute ^
                                     " in:\n" ^ (string_of_xml xml_obj))

let get_datatype dtype =
    match dtype with
        "boolean"   -> "bool"
      | "single"    -> "float_t"
      | _ as d      -> d ^ "_t" (* e.g. uint32_t, int8_t, etc. *)

let if_elements l printstr =
    if (List.length l) > 0
    then printstr
    else ""

(* Structure for returning input and output types *)
type interface = {
    name     : string;
    datatype : string;
}

(* virtual Base class all blocks inherit from. All methods here
 * will be utilized by upstream utilities *)
class virtual base xml_obj = object
    val name : string   = string_of_value (get_attr "name" xml_obj)
    method name         = name
    (* Block-specific functionality *)
    method virtual inputs       : interface list
    method virtual outputs      : interface list
    method virtual inner_objs   : base list
    (* Potentially dangerous, but only used in context of
     * getting inner objects first *)
    method virtual set_inputs   : interface list -> unit
    method virtual set_outputs  : interface list -> unit
    method virtual set_inner_objs : base list -> unit
```

```
44      (* Used for general purposes and to distinguish blocks *)
45      method virtual print_class  : string
46      method virtual print_obj    : string
47      (* Code generation functions *)
48      method virtual header       : string
49      method virtual body         : string
50      method virtual trailer      : string
51      (* Function used in trace algorithm in order to find
52       * connection from an input *)
53      method get_connection input_to =
54          let input_from = List.filter (fun x -> (get_attr "to" x) = Name input_to)
55              (List.filter (fun x -> x.tagname = "CONNECTION") xml_obj.inner_objs)
56           in match input_from with
57                 []      -> object_error
58                              ("No connections found for " ^
59                                  string_of_value (get_attr "name" xml_obj)
60                              )
61               | [cnx]   -> get_attr "from" cnx
62               | _ :: _  -> object_error
63                              ("Too many connections defined for " ^
64                                  string_of_value (get_attr "name" xml_obj)
65                              )
66  end;;
67
68  (* Intermediate class used by both block and reference classes *)
69  class virtual blk_or_ref blockify xml_obj = object (self)
70      inherit base xml_obj
71      val mutable virtual inner_objs : base list
72      method inner_objs = List.rev inner_objs
73      (* Get input/output objects inside this object *)
74      method inputs   = List.map
75          (fun x -> List.hd ((x :> base) #outputs))
76          (List.filter
77              (fun (x : base) -> ((x :> base) #print_class) = "input")
78              inner_objs
79          )
80      method outputs  = List.map
81          (fun x -> List.hd ((x :> base) #outputs))
82          (List.filter
83              (fun (x : base) -> ((x :> base) #print_class) = "output")
84              inner_objs
85          )
86      (* Since this object has a set of inputs we want to keep immutable
87       * use the following construct such that we can print what the body
88       * code needs without modifying the block's list of inputs/outputs *)
89      val mutable connected_inputs = []
90      method connected_inputs = connected_inputs
91      method set_inputs new_inputs = connected_inputs <- new_inputs
92      method set_outputs a = object_error (
93                                  "Should not set outputs of " ^
94                                  self#print_class ^ " object")
95      method input_type   = if_elements
96                              self#inputs
97                              ("struct " ^ self#func ^ "_in")
98      method output_type  = if_elements
```

```
                            self#outputs
                            ("struct " ^ self#func ^ "_out")
    method virtual func : string (* Used because block cannot have
                                    * a different name, but reference can *)
    method body       = if_elements (* Create code for setting input structure *)
                            self#inputs
                            (self#input_type ^ " " ^
                             self#name ^ "_inputs = " ^ "{\n\t\t" ^
                                (String.concat
                                    ",\n\t\t"
                                    (List.map
                                        (fun (x, y) -> "." ^ x.name ^
                                                       " = " ^ y.name
                                        )
                                        (List.combine
                                            self#inputs
                                            self#connected_inputs
                                        )
                                    )
                                ) ^ "\n\t};\n\t"
                            ) ^
                        if_elements (* Create code for setting output struct *)
                            self#outputs
                           (self#output_type ^ " " ^
                            self#name ^ "_outputs =\n\t\t") ^
                        self#func ^ "(" ^ (* function call *)
                        if_elements (* Only apply inputs if block has inputs *)
                            self#inputs
                            (self#name ^ "_inputs") ^
                        ");"
    method print_obj  = "\"" ^ self#print_class ^ "\": {\n" ^
                        "    \"name\":\"" ^ name ^ "\"\n" ^
                        "    \"inner_objs\": [\n      " ^
                        (String.concat "\n      "
                            (List.map
                                (fun (x : base) -> (x :> base) #print_obj)
                                self#inner_objs
                            )
                        ) ^ "\n    ]" ^
                        "\n}\n"
end;;

(* Block class: BLOCK tag, is a container for other blocks *)
class block blockify xml_obj = object (self)
    inherit blk_or_ref blockify xml_obj
    val mutable inner_objs = List.map
                             blockify
                             (List.filter
                                (fun x -> x.tagname <> "CONNECTION")
                                xml_obj.inner_objs
                             )
    method func = name
    method set_inner_objs new_inner_objs = inner_objs <- new_inner_objs
    method ref_blks = List.filter
            (fun (x : base) -> let c = ((x :> base) #print_class) in
```

```
                                c = "reference"
            )
            inner_objs
    method print_inc = if_elements
                            self#ref_blks
                            (String.concat
                                "\n"
                                (List.map
                                    (fun x -> (x :> base) #header)
                                    self#ref_blks
                                ) ^ "\n\n"
                            )
    method static_blks = List.filter
            (fun (x : base) -> let c = ((x :> base) #print_class) in
                                c = "memory"
                              || c = "constant"
                              || c = "dt"
            )
            inner_objs
    method print_static = if_elements
                            self#static_blks
                            ("/* Initialize static variables */\n" ^
                             String.concat
                                "\n"
                                (List.map
                                    (fun x -> (x :> base) #header)
                                    self#static_blks
                                ) ^ "\n\n"
                            )
    method print_class  = "block"
    method input_struct = if_elements
                            self#inputs
                            (self#input_type ^ " {\n\t" ^
                                (String.concat ";\n\t"
                                    (List.map
                                    (fun x -> (get_datatype x.datatype) ^
                                              " " ^ x.name
                                    )
                                    self#inputs)
                                ) ^ ";\n};\n\n"
                            )
    method output_struct = if_elements
                            self#outputs
                            (self#output_type ^ " {\n\t" ^
                                (String.concat ";\n\t"
                                    (List.map
                                    (fun x -> (get_datatype x.datatype) ^
                                              " " ^ x.name
                                    )
                                    self#outputs)
                                ) ^ ";\n};\n\n"
                            )
    method header     = (* Include statements for referenced files*)
                        self#print_inc ^
                        (* Structure definition for block *)
```

```
209                         if_elements
210                             (self# inputs @ self#outputs)
211                             ("/* I/O Structures for block " ^ name ^ " */\n") ^
212                         self#input_struct ^
213                         self#output_struct ^
214                         (* Initialize static constants and parameters *)
215                         self#print_static ^
216                         (* Function definition *)
217                         (let out_struct = self#output_type in
218                           if out_struct <> ""
219                           then out_struct
220                           else "void") ^
221                         "\n/* Function def */ " ^ name ^ "(" ^
222                         (let in_struct = self#input_type in
223                         if in_struct <> ""
224                         then in_struct ^ " inputs"
225                         else "") ^
226                         ")\n{\n" ^
227                         (* Unpack inputs *)
228                         (let input_blk = String.concat "\n\t"
229                             (List.map
230                             (fun x -> (get_datatype x.datatype) ^ " " ^
231                                 x.name ^ " = inputs." ^ x.name ^ ";"
232                             )
233                             self#inputs) in
234                         if input_blk <> ""
235                         then "\t/* Inputs for block " ^ name ^
236                              " */\n\t" ^ input_blk ^ "\n\n"
237                         else "") ^
238                         (* Code for inner objects in SSA form *)
239                         if_elements
240                             self#inner_objs
241                             ("\t/* Body for block " ^ name ^ " */\n\t" ^
242                             (String.concat "\n\t"
243                                 (List.map
244                                     (fun x -> (x :> base) #body)
245                                     (* Skip parts block takes care of *)
246                                     (List.filter
247                                         (fun x -> let c =
248                                                     (x :> base) #print_class
249                                                   in
250                                                    not (   c = "input"
251                                                         || c = "dt"
252                                                         || c = "constant"
253                                                        )
254                                         )
255                                         self#inner_objs
256                                     )
257                                 )
258                             ) ^ "\n\n")
259
260     method trailer    = (* Pack up outputs *)
261                         if_elements
262                             self#outputs
263                             ("\t/* Outputs for block " ^ name ^" */\n\t" ^
```

```
                            self#output_type ^ " outputs;\n\t" ^
                            (String.concat ";\n\t"
                                (List.map
                                (fun x -> "outputs." ^ x.name ^ " = " ^ x.name)
                                self#outputs)
                            ) ^ ";\n\n" ^
                            (* terminate function *)
                            "\treturn outputs;") ^
                        "\n}\n"
end;;

(* Parse referenced file for the referenced block and return it for down below *)
let get_file xml_obj =
    let r = (get_attr "ref" xml_obj)
         in match r with
              Ref r -> if r.reftype = "FILE"
                       then r.refroot
                       else object_error "Ref object only supports " ^
                                         "file references"
            | _     -> object_error "Incorrect Type for filename"

(* Get the referenced block in the right file for the given reference object *)
let get_ref_blk xml_obj =
    let rec get_inner_blk blk_list xml_obj =
        match blk_list with
            [] -> xml_obj
          | hd :: tl -> begin
                        let new_xml_obj =
                                (List.filter
                                    (fun x -> string_of_value
                                                (get_attr "name" x) = hd)
                                     (List.filter
                                        (fun x -> x.tagname <> "CONNECTION")
                                        (xml_obj :: xml_obj.inner_objs)
                                     )
                                )
                         in if (List.length new_xml_obj) <> 1
                            then object_error ("Did not find exactly one " ^
                                               "referenced block")
                            else get_inner_blk tl (List.hd new_xml_obj)
                        end
     in let file = get_file xml_obj
     in let xml_obj = (Xparser.xml_tree Xscanner.token
                                (Lexing.from_channel (open_in file) )
                       ) (* Have to parse referenced
                            * file to get block *)
        and blk_list =
            let r = (get_attr "ref" xml_obj)
                in match r with
                       Ref r -> r.reflist
                     | _     -> object_error "Incorrect Type for block ref"
         in get_inner_blk blk_list xml_obj

(* Reference class: REFERENCE tag, references a block in another file *)
class reference blockify xml_obj = object (self)
```

```
319     inherit blk_or_ref blockify xml_obj
320     method func = string_of_value (get_attr "name" (get_ref_blk xml_obj))
321     val mutable inner_objs   = List.map
322                                  blockify
323                                  (List.filter
324                                     (fun x -> x.tagname <> "CONNECTION")
325                                     (get_ref_blk xml_obj).inner_objs
326                                  )
327     method set_inner_objs new_inner_objs = object_error
328                              ("Should not try to set inner objects of " ^
329                               self#print_class ^ " object: " ^ self#name ^ "")
330     method print_class  = "reference"
331     method header       = let vlfile = (get_file xml_obj)
332                           in let cfile = (Str.global_replace
333                                             (Str.regexp "\\.vl")
334                                             ".c"
335                                             vlfile
336                                          )
337                           in "#include \"" ^ cfile ^"\""
338     method trailer      = ""
339 end;;
340
341 (* virtual I/O Part class: do all I/O Part attributes and checking *)
342 class virtual io_part xml_obj = object (self)
343     inherit base xml_obj
344     method inner_objs = object_error
345                              ("Should not try to access inner objects of " ^
346                               self#print_class ^ " object: " ^ self#name ^ "")
347     method set_inner_objs new_inner_objs = object_error
348                              ("Should not try to set inner objects of " ^
349                               self#print_class ^ " object: " ^ self#name ^ "")
350     val datatype = string_of_value (get_attr "datatype" xml_obj)
351     method datatype = datatype
352     val mutable inputs   = [{ name = string_of_value
353                                        (get_attr "name" xml_obj);
354                               datatype = string_of_value
355                                        (get_attr "datatype" xml_obj)
356                             }]
357     method inputs = inputs
358     method set_inputs new_inputs = inputs <- new_inputs
359     method outputs  = [{ name = self#name; datatype = self#datatype }]
360     method set_outputs a = object_error (
361                                "Should not set outputs of " ^
362                                self#print_class ^ " object")
363     method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
364                           "\"name\":\"" ^ name ^ "\", " ^
365                           "\"datatype\":\"" ^ datatype ^ "\", " ^
366                           " }"
367     method header     = ""
368     method body       = ""
369     method trailer    = ""
370 end;;
371
372 (* Input class: INPUT tag*)
373 class input xml_obj = object (self)
```

```
374     inherit io_part xml_obj as super
375     method inputs   = object_error "Should never access inputs of input obj"
376     method set_inputs  a = object_error
377                                ("Should not set inputs of " ^
378                                self#print_class ^ " object")
379     method print_class  = "input"
380 end;;
381
382 (* Output class: OUTPUT tag *)
383 class output xml_obj = object (self)
384     inherit io_part xml_obj as super
385     method print_class  = "output"
386     method body       = get_datatype (List.hd self#outputs).datatype ^ " " ^
387                         self#name ^ " = " ^
388                         (List.hd self#inputs).name ^ ";"
389 end;;
390
391 (* Constant class: CONSTANT tag*)
392 class constant xml_obj = object (self)
393     inherit input xml_obj (* A constant acts like an input, except it has
394                              * a value and doesn't interact with block I/O *)
395     val value    = string_of_value (get_attr "value"    xml_obj)
396     method value = value
397     method header     = (* overriden for block#header*)
398                         "static " ^ (get_datatype self#datatype) ^ " " ^
399                         self#name ^ " = " ^ value ^ ";"
400     method print_class  = "constant"
401     method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
402                           "\"name\":\"" ^ name ^ "\", " ^
403                           "\"value\":\"" ^ value ^ "\", " ^
404                           " }"
405 end;;
406
407 (* DT class: starts as ic, gets updated each pass as delta t in code exec *)
408 class dt xml_obj = object (self)
409     inherit base xml_obj
410     method inner_objs = object_error
411                           ("Should not try to access inner objects of " ^
412                            self#print_class ^ " object: " ^ self#name ^ "")
413     method set_inner_objs new_inner_objs = object_error
414                           ("Should not try to set inner objects of " ^
415                            self#print_class ^ " object: " ^ self#name ^ "")
416     method inputs   = object_error "Should never access inputs of dt obj"
417     method set_inputs  a = object_error
418                                ("Should not set inputs of " ^
419                                self#print_class ^ " object")
420     method outputs  = [{ name = self#name; datatype = "single" }]
421     method set_outputs a = object_error (
422                                "Should not set outputs of " ^
423                                self#print_class ^ " object")
424     method datatype = "single"
425     val init_cond       = string_of_value (get_attr "ic" xml_obj)
426     method header     = "static " ^ (get_datatype self#datatype) ^ " " ^
427                         self#name ^ " = " ^ init_cond ^ ";"
428     method body         = ""
```

```
    method trailer      = ""
    method print_class  = "dt"
    method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
                          "\"name\":\"" ^ name ^ "\", " ^
                          "\"ic\":\"" ^ init_cond ^ "\", " ^
                          " }"
end;;

(* All other parts inherit from this one *)
class virtual part xml_obj = object (self)
    inherit base xml_obj
    method inner_objs = object_error
                          ("Should not try to access inner objects of " ^
                           self#print_class ^ " object: " ^ self#name ^ "")
    method set_inner_objs new_inner_objs = object_error
                          ("Should not try to set inner objects of " ^
                           self#print_class ^ " object: " ^ self#name ^ "")
    val virtual mutable inputs  : interface list
    method inputs  = inputs
    method set_inputs new_inputs = inputs <- new_inputs
    val virtual mutable outputs : interface list
    method outputs = outputs
    method set_outputs new_outputs = outputs <- new_outputs
    method virtual body         : string
    method header     = ""
    method trailer    = ""
end;;

(* Memory class: MEM tag*)
class memory xml_obj = object (self)
    inherit part xml_obj
    val init_cond       = string_of_value (get_attr "ic" xml_obj)
    val mutable inputs   = [{ name = "current"; datatype = "auto" }]
    val mutable outputs  = [{ name = "stored"; datatype = "auto" }]
    val datatype = string_of_value (get_attr "datatype" xml_obj)
    method datatype = datatype
    method init_cond = init_cond
    method print_class  = "memory"
    method print_obj    = "\"memory\": { " ^
                          "\"name\":\"" ^ name ^ "\", " ^
                          "\"init_cond\":" ^ init_cond ^ "\"" ^
                          " }"
    method header     = (* overriden for block#header*)
                        "static " ^ (get_datatype self#datatype) ^ " " ^
                        self#name ^ " = " ^ init_cond ^ ";"
    method body         = self#name ^ " = " ^
                          (List.hd inputs).name ^ ";"
end;;

(* NOT Gate Part class: unary NOT operation *)
class not_gate xml_obj = object (self)
    inherit part xml_obj
    val mutable inputs  = [{ name = "input"; datatype = "boolean" }]
    val mutable outputs = [{ name = "output"; datatype = "boolean" }]
    method print_class  = "not"
```

```
    method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
                          "\"name\":\"" ^ name ^ "\", " ^
                          "\"operation\":\"!\" }"
    method body         = (get_datatype (List.hd outputs).datatype) ^ " " ^
                          self#name ^ " = !(" ^
                          (List.hd inputs).name ^ ");"
end;;

(* Helper functions for binary operation parts, which can have an arbitrary
 * number of inputs, so long as there is at least 2. *)
let get_num_connections xml_obj =
    let inputs = List.filter
                 (fun x -> x.tagname = "CONNECTION")
                 xml_obj.inner_objs
     in List.length inputs

let get_cnx_list xml_obj set_type=
    let num_cnx = get_num_connections xml_obj
     in
        let rec create_cnx_list num_cnx cnx_list =
            let idx = (num_cnx - (List.length cnx_list))
         in let idx_name = "input" ^ (string_of_int idx)
         in match idx with
                0 -> cnx_list
              | _ -> let cnx_list =
                          {name = idx_name; datatype = set_type} :: cnx_list
                     in create_cnx_list num_cnx cnx_list
     in create_cnx_list num_cnx []
    (* inputs for binop parts are named input1 through inputN
    * and the operation will be applied on all elements *)

(* virtual Binary Operation class: do all binary attributes and checking *)
class virtual binop_part xml_obj = object (self)
    inherit part xml_obj
    val virtual operation : string
    method operation = operation
    method virtual datatype : string
    method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
                          "\"name\":\"" ^ name ^ "\", " ^
                          "\"operation\":\"" ^ self#operation ^ "\" }"
    method body         = (get_datatype self#datatype) ^ " " ^
                          self#name ^ " = " ^ String.concat
                                (" " ^ self#operation ^ " ")
                                (List.map
                                    (fun x -> x.name)
                                    self#inputs
                                ) ^
                          ";"
end;;

(* intermediate class to explicitly set datatype for gate parts *)
class virtual gate xml_obj = object
    inherit binop_part xml_obj as super
    val datatype = "boolean"
    method datatype = datatype
```

```
539     val mutable inputs  = get_cnx_list xml_obj "boolean"
540     val mutable outputs = [{ name = "output"; datatype = "boolean" }]
541 end;;
542 
543 (* OR gate: inherits from binary_gate_part, logical OR operation *)
544 class or_gate xml_obj = object (self)
545     inherit gate xml_obj
546     val operation = "||"
547     method print_class = "or"
548 end;;
549 
550 (* AND gate: inherits from binary_gate_part, logical AND operation *)
551 class and_gate xml_obj = object (self)
552     inherit gate xml_obj
553     val operation = "&&"
554     method print_class = "and"
555 end;;
556 
557 (* NOR gate: inherits from binary_gate_part, logical NOR operation *)
558 class nor_gate xml_obj = object (self)
559     inherit gate xml_obj as super
560     val operation = "" (* overriden body, operation is AND of NOT-ed inputs *)
561     method print_class = "nor"
562     method body         = (get_datatype self#datatype) ^ " " ^
563                           self#name ^ " = !(" ^ String.concat
564                                 (") && !(")
565                                 (List.map
566                                     (fun x -> x.name)
567                                     self#inputs
568                                 ) ^
569                           ");"
570 end;;
571 
572 (* NAND gate: inherits from binary_gate_part, logical NAND operation *)
573 class nand_gate xml_obj = object (self)
574     inherit gate xml_obj
575     val operation = "" (* overriden body, operation is OR of NOT-ed inputs *)
576     method print_class = "nand"
577     method body         = (get_datatype self#datatype) ^ " " ^
578                           self#name ^ " = !(" ^ String.concat
579                                 (") || !(")
580                                 (List.map
581                                     (fun x -> x.name)
582                                     self#inputs
583                                 ) ^
584                           ");"
585 end;;
586 
587 (* XOR gate: inherits from binary_gate_part, logical XOR operation *)
588 class xor_gate xml_obj = object (self)
589     inherit gate xml_obj as super
590     val operation = "" (* overriden body, operation is NEQ of each input *)
591     method print_class = "xor"
592     method body         = (get_datatype self#datatype) ^ " " ^
593                           self#name ^ " = (" ^ String.concat
```

```
                                (") != (")
                                (List.map
                                    (fun x -> x.name)
                                    self#inputs
                                ) ^
                            ");"
end;;

(* Summation point: inherits from binop_part, addition operation *)
class sum xml_obj = object (self)
    inherit binop_part xml_obj
    val operation = "+"
    val datatype = string_of_value (get_attr "datatype" xml_obj)
    method datatype = datatype
    method print_class = "sum"
    val mutable inputs  = get_cnx_list xml_obj "auto"
    val mutable outputs = [{ name = "output"; datatype = "auto" }]
end;;

(* Production point: inherits from binop_part, multiplication operation *)
class prod xml_obj = object (self)
    inherit binop_part xml_obj
    val operation = "*"
    val datatype = string_of_value (get_attr "datatype" xml_obj)
    method datatype = datatype
    method print_class = "prod"
    val mutable inputs  = get_cnx_list xml_obj "auto"
    val mutable outputs = [{ name = "output"; datatype = "auto" }]
end;;

(* GAIN Part class: unary multiplication operation *)
class gain xml_obj = object (self)
    inherit part xml_obj
    val mutable inputs  = [{ name = "input"; datatype = "auto" }]
    val mutable outputs = [{ name = "output"; datatype = "auto" }]
    val datatype = string_of_value (get_attr "datatype" xml_obj)
    method datatype = datatype
    val value    = string_of_value (get_attr "value"    xml_obj)
    method value = value
    method print_class  = "gain"
    method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
                          "\"name\":\"" ^ name ^ "\", " ^
                          "\"datatype\":\"" ^ datatype ^ "\", " ^
                          "\"value\":\"" ^ value ^ "\" }"
    method body         = (get_datatype datatype) ^ " " ^
                          self#name ^ " = " ^ value ^ " * " ^
                          (List.hd inputs).name ^ ";"
end;;

(* INV Part class: unary inversion/division operation *)
class inv xml_obj = object (self)
    inherit part xml_obj
    val mutable inputs  = [{ name = "input"; datatype = "auto" }]
    val mutable outputs = [{ name = "output"; datatype = "auto" }]
    val datatype = string_of_value (get_attr "datatype" xml_obj)
```

```
649     method datatype = datatype
650     method print_class  = "inv"
651     method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
652                           "\"name\":\"" ^ name ^ "\", " ^
653                           "\"datatype\":\"" ^ datatype ^ "\" }"
654     method body         = let input = (List.hd inputs).name in
655                           (get_datatype datatype) ^ " " ^
656                           self#name ^ " = " ^
657                           (* Divide by zero protection *)
658                           "(abs(" ^ input ^ ") >= FLT_MIN) ?\n\t\t" ^
659                           "(1 / ( " ^ input ^ " )) : (0.000f);"
660 end;;
661
662 (* Compare Part: compares two inputs using operation *)
663 class compare xml_obj = object (self)
664     inherit part xml_obj
665     val operation = string_of_value (get_attr "operation" xml_obj)
666     val datatype = string_of_value (get_attr "datatype" xml_obj)
667     method datatype = datatype
668     method print_class = "compare"
669     val mutable inputs  = [{ name = "lhs"; datatype = "auto" };
670                            { name = "rhs"; datatype = "auto" }]
671     val mutable outputs = [{ name = "output"; datatype = "boolean" }]
672     method body         = (get_datatype (List.hd outputs).datatype) ^ " " ^
673                           self#name ^ " = (" ^
674                             String.concat
675                                 ( " " ^ operation ^ " ")
676                                 (List.map (fun x -> x.name) self#inputs)
677                           ^ ");"
678     method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
679                           "\"name\":\"" ^ name ^ "\", " ^
680                           "\"datatype\":\"" ^ datatype ^ "\", " ^
681                           "\"operation\":\"" ^ operation ^ "\" }"
682 end;;
683
684 (* If part: if control is true, pass true input, else false input *)
685 class if_sw xml_obj = object (self)
686     inherit part xml_obj
687     val datatype = string_of_value (get_attr "datatype" xml_obj)
688     method datatype = datatype
689     method print_class = "if"
690     val mutable inputs  = [{ name = "control"; datatype = "boolean" };
691                            { name = "true"; datatype = "auto" };
692                            { name = "false"; datatype = "auto" }]
693     val mutable outputs = [{ name = "output"; datatype = "auto" }]
694     method body         = (get_datatype datatype) ^ " " ^
695                           self#name ^ " = (" ^ (List.nth self#inputs 0).name ^
696                           ") ?\n\t\t(" ^ (List.nth self#inputs 1).name ^
697                           ") :\n\t\t(" ^ (List.nth self#inputs 2).name ^
698                           ");"
699     method print_obj    = "\"" ^ self#print_class ^ "\": { " ^
700                           "\"name\":\"" ^ name ^ "\", " ^
701                           "\"datatype\":\"" ^ datatype ^ "\" }"
702 end;;
703
```

```
(* Main block management functions *)
(* Blockify goes through and matches the tagname to the appropiate object *)
let rec blockify xml_obj =
    match xml_obj.tagname with
          "BLOCK"       -> (new block blockify xml_obj :> base)
        | "REFERENCE"   -> (new reference blockify xml_obj :> base)
          (* Note: passing blockify into block/ref instantiation because they
           * can't see at compile time what the function blockify refers to *)
        | "INPUT"       -> (new input       xml_obj :> base)
        | "OUTPUT"      -> (new output      xml_obj :> base)
        | "CONSTANT"    -> (new constant    xml_obj :> base)
        | "DT"          -> (new dt          xml_obj :> base)
        | "MEM"         -> (new memory      xml_obj :> base)
        | "NOT"         -> (new not_gate    xml_obj :> base)
        | "AND"         -> (new and_gate    xml_obj :> base)
        | "OR"          -> (new or_gate     xml_obj :> base)
        | "NAND"        -> (new nand_gate   xml_obj :> base)
        | "NOR"         -> (new nor_gate    xml_obj :> base)
        | "XOR"         -> (new xor_gate    xml_obj :> base)
        | "SUM"         -> (new sum         xml_obj :> base)
        | "PROD"        -> (new prod        xml_obj :> base)
        | "GAIN"        -> (new gain        xml_obj :> base)
        | "INV"         -> (new inv         xml_obj :> base)
        | "COMPARE"     -> (new compare     xml_obj :> base)
        | "IF"          -> (new if_sw       xml_obj :> base)
        (* CONNECTION blocks are not supported by this operation.
         * See get_connection above *)
        | _ as name -> object_error ("Tag " ^ name ^ " not supported.")

(* Main caller function simply to protect against top level blocks not being
 * of type BLOCK *)
let parse_xml_tree xml_obj =
    match xml_obj.tagname with
          "BLOCK"   -> blockify xml_obj
        | _ as name -> object_error
                          ("Tag " ^ name ^ " cannot be top level block")
```

Listing 10: Block Object Ordering and Optimization

../src/blockparse.ml

```
open Blockify
open Errors
open Xst

let print_list program = String.concat "\n\n"
    (List.map (fun x -> (x :> base) #print_obj) program)

(* Block Parse intelligently traces through the objects inside a block from
 * output to input and finds an appropiate path through the block such
 * that when the code is extracted from the order obtained here,
 * the program is consistent and no runtimes issues occur. *)
let rec block_parse top =
    (* Algorithm:
```

```
14      * The block trace algorithm will get the list of outputs from the
15      * current block level, and recursively traverse the current object
16      * list by finding the connection made from each input (starting at
17      * the output), and tracing it back to it's last output. The recursion
18      * will continue until either: an input is found (terminate that branch),
19      * a memory block is found (terminate that branch, and add memory's input
20      * to the list of traversals), a traversal is made to an object on the list
21      * of priors (terminate branch), or an algebraic loop is detected (raise
22      * error if the next traversal is already in the list of traversals made).
23      * At the termination of a traversal for an output, all of the objects
24      * detected are consistent and the entire list of objects is added to the
25      * list of priors. This process continues until all output and memory
26      * blocks successfully traverse back to inputs or priors branches. *)
27    let rec trace block_list prior_list trace_list current =
28        let compare_obj n = (fun x -> (x :> base) #name = n)in
29        match ((current :> base) #print_class) with
30            "input"
31          | "constant"
32          | "dt"        -> if List.exists (compare_obj current#name) prior_list
33                           || List.exists (compare_obj current#name) trace_list
34                           (* If terminating block exists in EITHER
35                            * list, exclude *)
36                           then trace_list
37                           else current :: trace_list
38          | _ as blk    ->
39            (* If current object exists in the current trace loop,
40             * this means there's a cyclic reference in the trace that
41             * will not be possible to escape, e.g. algebraic loop *)
42            if List.exists (compare_obj current#name) trace_list
43            then object_error (blk ^ ": " ^ ((current :> base) #name) ^
44                               " is in an algebraic loop...")
45            (* If current object exists on the list of priors, that means
46             * that value is already computed and will not need to be
47             * computed again. *)
48            else if List.exists (compare_obj current#name) prior_list
49                 then trace_list
50                 (* Default case: kick off trace for each connected input
51                  * in current object's list of inputs *)
52                 else if (blk = "memory") && ((List.length trace_list) > 0)
53                      then trace_list (* Terminate trace at memory block
54                                       * if one is found as an input *)
55                 else
56                  (* First find and verify all inputs connected to current
57                   * block, matching them to the relevant blocks for further
58                   * recursion. Next, set names of current blocks to the
59                   * outputs of those blocks correctly such that they can
60                   * be printed correctly in SSA form without error.
61                   * Note: need to handle blocks (function calls) separetely
62                   * using the REF type so that SSA works.
63                   * Note: In order to link current block to inputs, we
64                   * need to replace input names for current block with
65                   * the output names of the corresponding parts. E.g.
66                   * block name for basic parts and structured defs
67                   * for block and reference function calls. *)
68                    let (new_inputs, input_names) =
```

```
69                          List.split
70                              (List.map
71                                  (fun x -> let ref = current#get_connection x.name
72                                       in match ref with
73                                              Name n -> ({
74                                                              name = n;
75                                                              datatype = x.datatype
76                                                         },
77                                                         n)
78                                            (* When a reference is found, assume
79                                             * the function call completed and we
80                                             * are extracting the relevant output
81                                             * to that block here. *)
82                                            | Ref r  ->
83                                              if r.reftype = "NAME"
84                                              then if ((List.length r.reflist) = 1)
85                                                 then let cnx = (List.hd r.reflist)
86                                                      in ({ name = r.refroot ^
87                                                                   "_outputs." ^
88                                                                   cnx;
89                                                            datatype = x.datatype
90                                                          }, r.refroot)
91                                                 else object_error
92                                                    ("Cannot reference more " ^
93                                                    "than 1 deep for blocks")
94                                              else object_error
95                                                  ("FILE reference type " ^
96                                                  "not supported for ref " ^
97                                                      (string_of_ref r)
98                                                  )
99                                            | _ as attr -> object_error
100                                                 ("Attribute " ^
101                                                  (string_of_value attr) ^
102                                                  " not supported.")
103                                 )
104                                 ((current :> base) #inputs)
105                             )
106                     (* Compute the list of inputs to the current block
107                      * to split path and continue traversal *)
108                     in let input_list =
109                                 (List.map
110                                     (fun x ->
111                                         (List.find
112                                             (compare_obj x)
113                                             block_list
114                                         )
115                                     )
116                                     input_names
117                                 )
118                 in ((current :> base) #set_inputs new_inputs);
119                    let trace_list = current :: trace_list
120                 in trace_split block_list prior_list trace_list input_list
121 (* for each input of a block, trace out the list from that point on *)
122 and trace_split block_list prior_list trace_list input_list =
123     match input_list with
```

```
            []          -> trace_list
          | hd :: tl    -> let trace_list =
                                (trace block_list prior_list trace_list hd)
                           in trace_split block_list prior_list trace_list tl

    (* trace_start function: this function is the wrapper used to call the
     * inner trace algorithm. It recurses through the list of start objects,
     * applying the trace algorithm for each object, then appending the result
     * to the list of priors for the next recursion *)
     in
     let rec trace_start block_list prior_list start_list =
        match start_list with
            []       -> List.rev prior_list (* reverse list here because we were
                                               * traversing backwards above *)
          | hd :: tl -> let prior_list = prior_list @
                            (trace block_list prior_list [] hd)
                        in trace_start block_list prior_list tl

    (* start_list: the list of objects in the top block used to prime the trace
     * algorithm. All outputs and memory blocks are added to the start list
     * because they are the termination of the code the block will generate *)
     in
    let inner_objs obj = (obj :> base) #inner_objs
     in
    let start_list obj =
       (List.filter
            (fun x -> (x :> base) #print_class = "output")
            (inner_objs obj)
       )
     @ (List.filter
            (fun x -> (x :> base) #print_class = "memory")
            (inner_objs obj)
       )
     in
    (* Perform the same mutation operations for any inner blocks of top.
     * Note: at this point, if an inner object was not used, it should not appear
     * in the code for top below. *)
    let inner_block_list = List.filter
                           (fun x-> (x :> base) #print_class = "block")
                           (inner_objs top)
     in
    (* Perform the trace operation and re-set the inner objects of top with the
     * result. Also print objects that will be removed. *)
    let new_inner_objs = (trace_start (inner_objs top) [] (start_list top))
     in
    top#set_inner_objs new_inner_objs;
    (* Return a list of blocks with properly configured inner objects
     * to be used for compilation. Note: we reverse the list here so that
     * inner_blks are first to be compiled. *)
    List.rev (top :: List.flatten (List.map block_parse inner_block_list))
```

Listing 11: Code Generation

../src/compile.ml

```
open Blockify
open Blockparse

let translate program =
    (* Print standard libraries required *)
      "#include <stdbool.h>\n"
    ^ "#include <stdint.h>\n"
    ^ "#include <float.h>\n"
    ^ "#include <math.h>\n"
    ^ "\n"
    (* Print print the code for each block in the program using the optimized and
     * ordered inner blocks in the body code method for each *)
    ^ String.concat "\n\n" (List.map
                              (fun x -> let obj = (x :> base) in
                                  obj#header ^ obj#trailer
                              )
                              program
                           )
    ^ "\n/* Generated using VLCC */\n"
(* Generate python script for processing in files and sending it through the
 * compiled binary and printing the results as it is running *)
let gen_debug_code program =
    let top = ((List.hd (List.rev program)) :> base) in
        let name    = top#name in
        let inputs  = top#inputs in
        let outputs = top#outputs in
        let ctypes  = List.map
                        (fun x -> x.name ^ "\", "
                         ^  match x.datatype with
                              "uint8"  -> "c_uint8"
                            | "uint16" -> "c_uint16"
                            | "uint32" -> "c_uint32"
                            | "int8"   -> "c_int8"
                            | "int16"  -> "c_int16"
                            | "int32"  -> "c_int32"
                            | "single" -> "c_float"
                            | "double" -> "c_double"
                            | "boolean"-> "c_byte" (* Assume uint8 *)
                            | _ -> failwith "unassigned value"
                        )
         in
           "import sys\n"
         ^ "import ctypes\n"
         ^ "from ctypes import *\n"
         ^ "lib = cdll.LoadLibrary('./test-" ^ name ^ ".so')\n"
         ^ "class " ^ name ^ "_inputs(Structure):\n"
         ^ "    _fields_ = [(\""
         ^ (String.concat "), (\"" (ctypes inputs)) ^ ")]\n"
         ^ "    \n"
         ^ "class " ^ name ^ "_outputs(Structure):\n"
         ^ "    _fields_ = [(\""
         ^ (String.concat "), (\"" (ctypes outputs)) ^ ")]\n"
         ^ "    \n"
         ^ "lib." ^ name ^ ".restype = " ^ name ^ "_outputs\n"
```

```
            ^ "with open(sys.argv[1]) as f:\n"
            ^ "    for line in f:\n"
            ^ "        listargs = line.strip('\\n').split(',')\n"
            ^ "        inputs = " ^ name ^ "_inputs("
            ^     (String.concat
                      ", "
                      (List.mapi
                          (fun i x -> (
                              match x.datatype with
                                  "uint8"
                                | "uint16"
                                | "uint32"
                                | "int8"
                                | "int16"
                                | "int32"  -> "int"
                                | "single"
                                | "double" -> "float"
                                | "boolean"-> "int" (* Assume uint8 *)
                                | _ -> failwith "unassigned value"
                              )
                              ^ "(listargs[" ^ string_of_int(i) ^ "])"
                          )
                          inputs
                      )
                  )
                  ^ ")\n"
            ^ "        outputs = lib." ^ name ^ "(inputs)\n"
            ^ "        print ','.join(["
            ^     (String.concat
                      ", "
                      (List.map
                          (fun x -> "\"" ^ (
                              match x.datatype with
                                  "uint8"
                                | "uint16"
                                | "uint32"
                                | "int8"
                                | "int16"
                                | "int32"  -> "%d"
                                | "single"
                                | "double" -> "%.3f"
                                | "boolean"-> "%d" (* Assume uint8 *)
                                | _ -> failwith "unassigned value"
                              )
                              ^ "\" % outputs." ^ x.name
                          )
                          outputs
                      )
                  )
                  ^ "])"
```

Listing 12: VisLang Errors

../src/errors.ml

```
open Lexing
open Parsing
open Xst

(* Define errors *)
let issue msg start finish  =
        Printf.sprintf "(line %d: char %d..%d): %s"
                (start.pos_lnum)
                (start.pos_cnum - start.pos_bol)
                (finish.pos_cnum - finish.pos_bol)
                msg
exception XML_Error of string
let xml_error lexbuf = raise
                (XML_Error
                    (issue
                        ("Badly Formatted XML: " ^ (Lexing.lexeme lexbuf))
                        (Lexing.lexeme_start_p lexbuf)
                        (Lexing.lexeme_end_p lexbuf)
                    )
                )
let xml_warning lexbuf = ignore
                    (issue
                        ("Warning -- Skipping XML: " ^ (Lexing.lexeme lexbuf))
                        (Lexing.lexeme_start_p lexbuf)
                        (Lexing.lexeme_end_p lexbuf)
                    )
exception XML_Parse_Error of string
let xml_parse_error nterm msg = raise
                (XML_Parse_Error
                    (issue
                        ("Badly Formatted XML: " ^ msg)
                        (rhs_start_pos nterm)
                        (rhs_end_pos nterm)
                    )
                )

exception Block_Error of string
let block_error blk msg = raise
                (Block_Error
                    (msg ^ " for block:\n" ^ Xst.string_of_xml blk)
                )

let object_error msg = raise (Block_Error (msg) )
```

B VLCC Utilities

Listing 13: Automated Build Script

../src/Makefile

```
.DEFAULT_GOAL := vlcc

```

```
   OCAMLC=ocamlc
 OCAMLOPT=ocamlopt
 OCAMLDEP=ocamldep
 OCAMLLEX=ocamllex
OCAMLYACC=ocamlyacc

# main compilation
.SECONDARY:
MAIN_OBJS = xst.cmo errors.cmo xscanner.cmo xparser.cmo \
                        blockify.cmo blockparse.cmo compile.cmo vislang.cmo
vlcc : $(MAIN_OBJS)
        @echo "$(OCAMLC) -o vlcc"
        @$(OCAMLC) -o $@ str.cma $^

# Lexxer rules
%.ml : %.mll
        $(OCAMLLEX) -q $<

# Parser rules
%.ml %.mli : %.mly
        $(OCAMLYACC)    $<

# Common rules
.SUFFIXES: .ml .mli .cmo .cmi .cmx

.ml.cmo:
        $(OCAMLC) -c $<

.mli.cmi:
        $(OCAMLC) -c $<

.ml.cmx:
        $(OCAMLOPT) -c $<

clean:
        rm -f vlcc
        rm -f xscanner.ml xparser.ml xparser.mli
        rm -f *.cm[iox]

# Dependencies
depend:
        $(OCAMLDEP) $(INCLUDES) *.mli *.ml > .depend

include .depend
```

Listing 14: Automated Testing Script

../test/run_tests.sh

```
#!/bin/sh

VLCC="../src/vlcc"
GCC="gcc"
PYTHON="python"
```

```
6
7   # Set time limit for all operations
8   ulimit -t 30
9
10  globallog="../testall.log"
11  rm -f $globallog
12  error=0
13  globalerror=0
14
15  keep=0
16
17  Usage() {
18      echo "Usage: testall.sh [options] [.vl files]"
19      echo "-k    Keep intermediate files"
20      echo "-h    Print this help"
21      exit 1
22  }
23
24  SignalError() {
25      echo "FAILED"
26      error=1
27      echo "  $1"
28  }
29
30  # Compare <outfile> <reffile> <difffile>
31  # Compares the outfile with reffile.
32  # Differences, if any, written to difffile
33  Compare() {
34      generatedfiles="$generatedfiles $3"
35      echo diff -b $1 $2 ">" $3 1>&2
36      diff -b "$1" "$2" > "$3" 2>&1 || {
37          SignalError "$1 differs"
38          echo "FAILED $1 differs from $2" 1>&2
39      }
40  }
41
42  # Run <args>
43  # Report the command, run it, and report any errors
44  Run() {
45      echo $* 1>&2
46      eval $* || {
47          SignalError "failure: $*"
48          return 1
49      }
50  }
51
52  Check() {
53      error=0
54      basename=`echo $1 | sed 's/.*\\///
55                               s/.vl//'`
56      reffile=`echo $1 | sed 's/.vl$//'`
57      basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
58
59      echo -n "$basename..."
60
```

```
61      echo 1>&2
62      echo "###### Testing $basename" 1>&2
63
64      generatedfiles=""
65
66      generatedfiles="$generatedfiles ${basename}.c" &&
67      Run "$VLCC" "-c" $1 &&
68      referencedfiles="$(cat ${basename}.c | grep '#include \".*\.c\"' |
69          sed 's/\#include *\"//' | sed 's/\.c\"/.c/')" &&
70      generatedfiles="$generatedfiles $referencedfiles" &&
71      for file in $referencedfiles; do
72          Run "$VLCC" "-c" "${file%.c}.vl";
73      done &&
74      generatedfiles="$generatedfiles ${basename}.o" &&
75      Run "$GCC" "-c -fPIC" ${basename}.c &&
76      generatedfiles="$generatedfiles ${basename}.so" &&
77      Run "$GCC" "-shared -o" ${basename}.so ${basename}.o &&
78      generatedfiles="$generatedfiles ${basename}.py" &&
79      Run "$VLCC" "-d" $1 &&
80      generatedfiles="$generatedfiles ${basename}.c.out" &&
81      Run "$PYTHON" ${basename}.py ${basename}.in > ${basename}.c.out &&
82      Compare ${basename}.c.out ${reffile}.out ${basename}.c.diff
83
84      # Report the status and clean up the generated files
85
86      if [ $error -eq 0 ] ; then
87          if [ $keep -eq 0 ] ; then
88              rm -f $generatedfiles
89          fi
90          echo "OK"
91          echo "###### SUCCESS" 1>&2
92      else
93          echo "###### FAILED" 1>&2
94          globalerror=$error
95      fi
96  }
97
98  CheckPass() {
99      error=0
100     basename=`echo $1 | sed 's/.*\\///
101                             s/.vl//'`
102     reffile=`echo $1 | sed 's/.vl$//'`
103     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
104
105     echo -n "$basename..."
106
107     echo 1>&2
108     echo "###### Testing $basename" 1>&2
109
110     generatedfiles=""
111     # Basically check if we can compile all of it,
112     # then stop short of any testing
113     generatedfiles="$generatedfiles ${basename}.c" &&
114     Run "$VLCC" "-c" $1 &&
115     generatedfiles="$generatedfiles ${basename}.o" &&
```

```
116     Run "$GCC" "-c -fPIC" ${basename}.c &&
117
118     # Report the status and clean up the generated files
119
120     if [ $error -eq 0 ] ; then
121         if [ $keep -eq 0 ] ; then
122             rm -f $generatedfiles
123         fi
124         echo "OK"
125         echo "###### SUCCESS" 1>&2
126     else
127         echo "###### FAILED" 1>&2
128         globalerror=$error
129     fi
130 }
131
132 SignalPass() {
133     if [ $error -eq 1 ] ; then
134         echo "OK"
135         error=0
136     fi
137 }
138
139 # RunFail <args>
140 # Report the command, run it, and report any errors
141 RunFail() {
142     echo $* 1>&2
143     eval $* && {
144         SignalError "uncaught: $*"
145         return 1
146     } || {
147         SignalPass
148         return 0
149     }
150 }
151
152 CheckFail() {
153     error=1
154     basename=‘echo $1 | sed ’s/.*\\///
155                             s/.vl//’‘
156     reffile=‘echo $1 | sed ’s/.vl$//’‘
157     basedir="‘echo $1 | sed ’s/\/[^\/]*$//’‘/."
158
159     echo -n "$basename..."
160
161     echo 1>&2
162     echo "###### Testing $basename" 1>&2
163
164     RunFail "$VLCC" "-c" $1
165
166     # Report the status and clean up the generated files
167     if [ $error -eq 0 ] ; then
168         if [ $keep -eq 0 ] ; then
169             rm -f $generatedfiles
170         fi
```

```
        echo "###### SUCCESS" 1>&2
    else
        echo "###### FAILED" 1>&2
        globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
        k) # Keep intermediate files
            keep=1
        ;;
        h) # Help
            Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="./fail-*.vl ./pass-*.vl ./test-*.vl"
fi


for file in $files
do
    case $file in
        *test-*)
            Check $file 2>> $globallog
        ;;
        *fail-*)
            CheckFail $file 2>> $globallog
        ;;
        *pass-*)
            CheckPass $file 2>> $globallog
        ;;
        *)
            echo "unknown file type $file"
            globalerror=1
        ;;
    esac
    # Date and Time stamp for user log
    echo "Test completed at $(date '+%H:%M:%S on %m/%d/%y')" 1>> $globallog
done

exit $globalerror
```

C VLCC Test Cases

Listing 15: Algebraic Loop Failure Case

../test/fail-algebraic_loop.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="set_reset_latch">
    <vl:INPUT name="set" datatype="boolean"/>
    <vl:INPUT name="reset" datatype="boolean"/>
    <vl:NOT name="not_reset">
        <vl:CONNECTION to="input" from="reset"/>
    </vl:NOT>
    <vl:AND name="latch_and_not_reset">
        <vl:CONNECTION to="input1" from="set_or_mem"/>
        <vl:CONNECTION to="input2" from="not_reset"/>
    </vl:AND>
    <vl:OR name="set_or_mem">
        <vl:CONNECTION to="input1" from="set"/>
        <vl:CONNECTION to="input2" from="latch"/>
    </vl:OR>
    <vl:MEM name="latch_lp" datatype="boolean" ic="false">
        <vl:CONNECTION to="current" from="latch_and_not_reset"/>
    </vl:MEM>
    <vl:OUTPUT name="latch" datatype="boolean">
        <vl:CONNECTION to="latch" from="latch_and_not_reset"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 16: Bad Connection Failure Case

../test/fail-bad_connection.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="buffer">
    <vl:INPUT name="in" datatype="uint32"/>
    <vl:OUTPUT name="out" datatype="uint32">
        <vl:CONNECTION to="out" from="0x32"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 17: Missing Attribute Failure Case

../test/fail-missing_attribute.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK>
    <vl:INPUT name="in" datatype="uint32"/>
    <vl:OUTPUT name="out" datatype="uint32">
        <vl:CONNECTION to="out" from="in"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 18: Unended Block Failure Case

../test/fail-unended_block.vl

```
<vl:BLOCK name="empty_block">
```

Listing 19: Cascaded Blocks Completion Case

../test/pass-cascaded_empty_blocks.vl

```
<vl:BLOCK name="empty_block">
    <vl:BLOCK name="empty_block1"/>
    <vl:BLOCK name="empty_block2"/>
    <vl:BLOCK name="empty_block3">
        <vl:BLOCK name="empty_block4"/>
    </vl:BLOCK>
    <vl:BLOCK name="empty_block5">
        <vl:BLOCK name="empty_block6"/>
        <vl:BLOCK name="empty_block7"/>
    </vl:BLOCK>
    <vl:BLOCK name="empty_block8"/>
    <vl:BLOCK name="empty_block9"/>
</vl:BLOCK>
```

Listing 20: Empty Block Completion Case

../test/pass-empty_block.vl

```
<vl:BLOCK name="empty_block"/>
```

Listing 21: XML Tolerance Case

../test/pass-gibberish.vl

```
<?xml version="1.0" encoding="UTF-8"?>
sfdghsdf
sfgnsfjhs
fj
rtsr
thntr<Sgadfgsfg?>fsgfg<!>
</Sfdghsdths>
<vl:BLOCK name="buffer">
    gibberish name fgfgavava datatype  fgdfablah
    <vl:INPUT name="in" datatype="uint32"/>
    <vl:OUTPUT name="out" datatype="uint32">
        sfdgadfnagt
        CONNECTION " blah"
        <vl:CONNECTION to="out" from="in"/>
        OUTPUT dfgdfger
        more gibberish
    </vl:OUTPUT>
    this is all gibberish!
</vl:BLOCK>
```

Listing 22: Buffer Value Test Case

../test/test-buffer.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="buffer">
    <vl:INPUT name="in" datatype="uint32"/>
    <vl:OUTPUT name="out" datatype="uint32">
        <vl:CONNECTION to="out" from="in"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 23: Buffer in Buffer Value Test Case

../test/test-buffer_in_buffer.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="buffer_in_buffer">
    <vl:INPUT name="in" datatype="uint32"/>
    <vl:BLOCK name="buffer">
        <!-- 'from' in connection is scoped external
             of block, 'to' is internal -->
        <vl:CONNECTION to="in" from="in"/>
        <vl:INPUT name="in" datatype="uint32"/>
        <vl:OUTPUT name="out" datatype="uint32">
            <vl:CONNECTION to="out" from="in"/>
        </vl:OUTPUT>
    </vl:BLOCK>
    <vl:OUTPUT name="out" datatype="uint32">
        <vl:CONNECTION to="out" from="buffer|out"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 24: Comparision Operation Test Case

../test/test-compare.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="compare">
    <vl:INPUT name="in" datatype="single"/>
    <vl:CONSTANT name="ten" datatype="single" value="10.000"/>
    <vl:COMPARE name="grt" datatype="single" operation=">">
        <vl:CONNECTION to="lhs" from="in"/>
        <vl:CONNECTION to="rhs" from="ten"/>
    </vl:COMPARE>
    <vl:OUTPUT name="grt_out" datatype="boolean">
        <vl:CONNECTION to="grt_out" from="grt"/>
    </vl:OUTPUT>
    <vl:COMPARE name="lst" datatype="single" operation="<">
        <vl:CONNECTION to="lhs" from="in"/>
        <vl:CONNECTION to="rhs" from="ten"/>
```

```
    </vl:COMPARE>
    <vl:OUTPUT name="lst_out" datatype="boolean">
        <vl:CONNECTION to="lst_out" from="lst"/>
    </vl:OUTPUT>
    <vl:COMPARE name="geq" datatype="single" operation=">=">
        <vl:CONNECTION to="lhs" from="in"/>
        <vl:CONNECTION to="rhs" from="ten"/>
    </vl:COMPARE>
    <vl:OUTPUT name="geq_out" datatype="boolean">
        <vl:CONNECTION to="geq_out" from="geq"/>
    </vl:OUTPUT>
    <vl:COMPARE name="leq" datatype="single" operation="<=">
        <vl:CONNECTION to="lhs" from="in"/>
        <vl:CONNECTION to="rhs" from="ten"/>
    </vl:COMPARE>
    <vl:OUTPUT name="leq_out" datatype="boolean">
        <vl:CONNECTION to="leq_out" from="leq"/>
    </vl:OUTPUT>
    <vl:COMPARE name="eq" datatype="single" operation="==">
        <vl:CONNECTION to="lhs" from="in"/>
        <vl:CONNECTION to="rhs" from="ten"/>
    </vl:COMPARE>
    <vl:OUTPUT name="eq_out" datatype="boolean">
        <vl:CONNECTION to="eq_out" from="eq"/>
    </vl:OUTPUT>
    <vl:COMPARE name="neq" datatype="single" operation="!=">
        <vl:CONNECTION to="lhs" from="in"/>
        <vl:CONNECTION to="rhs" from="ten"/>
    </vl:COMPARE>
    <vl:OUTPUT name="neq_out" datatype="boolean">
        <vl:CONNECTION to="neq_out" from="neq"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 25: Logical Gate Test Case

../test/test-gates.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="gates">
    <vl:INPUT name="in1" datatype="boolean"/>
    <vl:INPUT name="in2" datatype="boolean"/>
    <vl:NOT name="not">
        <vl:CONNECTION to="input" from="in1"/>
    </vl:NOT>
    <vl:OUTPUT name="not_gate" datatype="boolean">
        <vl:CONNECTION to="not_gate" from="not"/>
    </vl:OUTPUT>
    <vl:OR name="or">
        <vl:CONNECTION to="input1" from="in1"/>
        <vl:CONNECTION to="input2" from="in2"/>
    </vl:OR>
    <vl:OUTPUT name="or_gate" datatype="boolean">
        <vl:CONNECTION to="or_gate" from="or"/>
```

```
    </vl:OUTPUT>
    <vl:AND name="and">
        <vl:CONNECTION to="input1" from="in1"/>
        <vl:CONNECTION to="input2" from="in2"/>
    </vl:AND>
    <vl:OUTPUT name="and_gate" datatype="boolean">
        <vl:CONNECTION to="and_gate" from="and"/>
    </vl:OUTPUT>
    <vl:NOR name="nor">
        <vl:CONNECTION to="input1" from="in1"/>
        <vl:CONNECTION to="input2" from="in2"/>
        <vl:CONNECTION to="input3" from="and"/>
    </vl:NOR>
    <vl:OUTPUT name="nor_gate" datatype="boolean">
        <vl:CONNECTION to="nor_gate" from="nor"/>
    </vl:OUTPUT>
    <vl:NAND name="nand">
        <vl:CONNECTION to="input1" from="in1"/>
        <vl:CONNECTION to="input2" from="in2"/>
        <vl:CONNECTION to="input3" from="not"/>
    </vl:NAND>
    <vl:OUTPUT name="nand_gate" datatype="boolean">
        <vl:CONNECTION to="nand_gate" from="nand"/>
    </vl:OUTPUT>
    <vl:XOR name="xor">
        <vl:CONNECTION to="input1" from="in1"/>
        <vl:CONNECTION to="input2" from="in2"/>
        <vl:CONNECTION to="input3" from="nor"/>
    </vl:XOR>
    <vl:OUTPUT name="xor_gate" datatype="boolean">
        <vl:CONNECTION to="xor_gate" from="xor"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 26: Reference Block Test Case

../test/test-hysteresis_sw.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="hysteresis_sw">
    <vl:INPUT name="in" datatype="single"/>
    <vl:CONSTANT name="hi" datatype="single" value="20.000"/>
    <vl:CONSTANT name="lo" datatype="single" value="10.000"/>
    <vl:COMPARE name="hi_cmp" datatype="single" operation=">=">
        <vl:CONNECTION to="lhs" from="in"/>
        <vl:CONNECTION to="rhs" from="hi"/>
    </vl:COMPARE>
    <vl:COMPARE name="lo_cmp" datatype="single" operation="<=">
        <vl:CONNECTION to="lhs" from="in"/>
        <vl:CONNECTION to="rhs" from="lo"/>
    </vl:COMPARE>
‘   <vl:REFERENCE name="sr_latch"
            ref="./test-set_reset_latch.vl|set_reset_latch">
        <vl:CONNECTION to="set" from="hi_cmp"/>
```

```
        <vl:CONNECTION to="reset" from="lo_cmp"/>
    </vl:REFERENCE>
    <vl:OUTPUT name="out" datatype="boolean">
        <vl:CONNECTION to="out" from="sr_latch|latch"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 27: Math Operations Test Case

../test/test-math_constant.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="math_constant">
    <vl:INPUT name="in" datatype="single"/>
    <vl:CONSTANT name="two" datatype="single" value="2"/>

    <vl:SUM name="summer" datatype="single">
        <vl:CONNECTION to="input1" from="in"/>
        <vl:CONNECTION to="input2" from="two"/>
    </vl:SUM>
    <vl:OUTPUT name="sum_out" datatype="single">
        <vl:CONNECTION to="sum_out" from="summer"/>
    </vl:OUTPUT>

    <vl:PROD name="mult" datatype="single">
        <vl:CONNECTION to="input1" from="in"/>
        <vl:CONNECTION to="input2" from="two"/>
    </vl:PROD>
    <vl:OUTPUT name="mult_out" datatype="single">
        <vl:CONNECTION to="mult_out" from="mult"/>
    </vl:OUTPUT>

    <vl:GAIN name="gain" datatype="single" value="-1.000">
        <vl:CONNECTION to="input" from="in"/>
    </vl:GAIN>
    <vl:OUTPUT name="gain_out" datatype="single">
        <vl:CONNECTION to="gain_out" from="gain"/>
    </vl:OUTPUT>

    <!-- Add four to input to prevent DIV/0 fault -->
    <vl:CONSTANT name="four" datatype="single" value="4"/>
    <vl:SUM name="summer2" datatype="single">
        <vl:CONNECTION to="input1" from="in"/>
        <vl:CONNECTION to="input2" from="four"/>
    </vl:SUM>
    <vl:INV name="inv" datatype="single">
        <vl:CONNECTION to="input" from="summer2"/>
    </vl:INV>
    <vl:OUTPUT name="inv_out" datatype="single">
        <vl:CONNECTION to="inv_out" from="inv"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 28: Memory Block Test Case

../test/test-memory.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="memory">
    <vl:INPUT name="in" datatype="uint32"/>
    <vl:MEM name="mem" datatype="uint32" ic="0x0">
        <vl:CONNECTION to="current" from="in"/>
    </vl:MEM>
    <vl:OUTPUT name="out" datatype="uint32">
        <vl:CONNECTION to="out" from="mem"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 29: SR Latch Complexity Test Case

../test/test-set_reset_latch.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="set_reset_latch">
    <vl:INPUT name="set" datatype="boolean"/>
    <vl:INPUT name="reset" datatype="boolean"/>
    <vl:NOT name="not_reset">
        <vl:CONNECTION to="input" from="reset"/>
    </vl:NOT>
    <vl:AND name="latch_and_not_reset">
        <vl:CONNECTION to="input1" from="set_or_mem"/>
        <vl:CONNECTION to="input2" from="not_reset"/>
    </vl:AND>
    <vl:OR name="set_or_mem">
        <vl:CONNECTION to="input1" from="set"/>
        <vl:CONNECTION to="input2" from="latch_lp"/>
    </vl:OR>
    <vl:MEM name="latch_lp" datatype="boolean" ic="false">
        <vl:CONNECTION to="current" from="latch_and_not_reset"/>
    </vl:MEM>
    <vl:OUTPUT name="latch" datatype="boolean">
        <vl:CONNECTION to="latch" from="latch_and_not_reset"/>
    </vl:OUTPUT>
</vl:BLOCK>
```

Listing 30: Timer Complexity Test Case

../test/test-timer.vl

```
<?xml version="1.0" encoding="UTF-8"?>
<vl:BLOCK name="timer">
    <!-- The BLOCK element denotes a subsystem of parts -->
    <!-- All "parts" added by the user can use Inputs and/or
         Outputs for utilization elsewhere in project. The
         reference will search the path for that file -->
    <!-- All Inputs do not have to be used and will be optimized out -->
```

```
    <vl:INPUT name="start" datatype="boolean"/>
    <vl:INPUT name="reset" datatype="boolean"/>
    <vl:INPUT name="time" datatype="single"/>
    <!-- Constants can be defined as a seperate block as well -->
    <vl:CONSTANT name="zero_constant" datatype="single" value="0.000"/>
    <!-- The DT block puts out the difference in time between
         successive passes of program. In a Soft RTOS, this
         would be a variable number. In a Hard RTOS, this
         would be a constant number. Here, we are saying the
         module will run around 10Hz, or 100ms (0.1 s).
         The DT module needs an initializer to guess the value
         on the first pass, but will be updated every pass afterwards -->
    <vl:DT name="time_since_last_pass" ic="0.100"/>
    <vl:NOT name="count_not_expired">
        <vl:CONNECTION to="input" from="count_expired_lp"/>
    </vl:NOT>
    <vl:AND name="start_enb">
        <vl:CONNECTION to="input1" from="start"/>
        <vl:CONNECTION to="input2" from="count_not_expired"/>
    </vl:AND>
    <vl:IF name="increment_value" datatype="single">
    <!-- Control flow IF switch: If Control is true, execute
         True assignment, else execute False assignment -->
        <vl:CONNECTION to="control" from="start_enb"/>
        <vl:CONNECTION to="true" from="time_since_last_pass"/>
        <vl:CONNECTION to="false" from="zero_constant"/>
    </vl:IF>
    <vl:SUM name="summer" datatype="single">
    <!-- The summer will add all the inputs together. If you want
         add a negative number, use the NEG part to negate the
         signal before connecting to this part. -->
    <!-- Additionally, the PROD part exists for taking the PI
         product of a set of inputs, and the INV command for taking
         the recipicral of a number (divide by zero runtime error
         is partially mitgated, but unexpected operation may occur) -->
        <vl:CONNECTION to="input1" from="increment_value"/>
        <vl:CONNECTION to="input2" from="elapsed_time_lp"/>
    </vl:SUM>
    <vl:IF name="reset_switch" datatype="single">
        <vl:CONNECTION to="control" from="reset"/>
        <vl:CONNECTION to="true" from="zero_constant"/>
        <vl:CONNECTION to="false" from="summer"/>
    </vl:IF>
    <vl:COMPARE name="is_count_expired" datatype="single" operation=">=">
        <vl:CONNECTION to="lhs" from="elapsed_time"/>
        <vl:CONNECTION to="rhs" from="time"/>
    </vl:COMPARE>
    <vl:MEM name="elapsed_time_lp" datatype="single" ic="0.000" >
        <vl:CONNECTION to="current" from="elapsed_time"/>
    </vl:MEM>
    <vl:MEM name="count_expired_lp" datatype="boolean" ic="false">
        <vl:CONNECTION to="current" from="count_expired"/>
    </vl:MEM>
    <!-- All Outputs need to have a connection in the part,
         at least to a constant -->
```

```
    <vl:OUTPUT name="count_expired" datatype="boolean">
    <!-- Outputs of a subsystem need to have a connection specified -->
        <vl:CONNECTION to="count_expired" from="is_count_expired"/>
    </vl:OUTPUT>
    <vl:OUTPUT name="elapsed_time" datatype="single">
        <vl:CONNECTION to="elapsed_time" from="reset_switch"/>
    </vl:OUTPUT>
</vl:BLOCK>
```