# blooRTLs

Peter Burrows, Zhibo (Andy) Wan, Apurv Gaurav, Pinhong He
phb2114, zw2327, ag3596, ph2482
EE/CE Department, Columbia University

July 2, 2015

## Contents

# 1 Introduction

The Behavioral Language for Object-Oriented Register Transfer Level Specs (blooRTLs) is a high-level RTL desription language geared towards catalyzing the development, simulation, and synthesis of RTL specs.

While blooRTLs does not embody the structure of an established "object-oriented" language, it does use the notion of objects (as described in Section 3.4), and in a general sense of the "object-oriented" programming style, blooRTLs does contribute a modular approach to the RTL design domain.

In regards to simulation and synthesis of the RTL design, the source code of blooRTLs must be precompiled (as explained in Section 4) before it can be compiled to VHDL. The compiler assumes that a reasonable clock frequency is to be used for the desired RTL spec. As a result, the complexity and types of arithmetic operations that can be performed within a given cycle has been limited by the blooRTLs syntax. Just as an RTL implementation that uses a reasonable clock frequency (i.e. MHz range) cannot load a register with data that has propagated through a substantial number of gates within the same clock cyle, blooRTLs does not permit multiple arithmetic operations to be performed within the same assignment expression. The legal expressions are discussed further in Section 3.6.

Overall, the modular and syntactical styles of blooRTLs offer an efficient and effective means of developing syntehsizable bit manipulation methods and RTL algorithms. The following reference manual borrows the framework of Dennis Ritchie's *C Reference Manual* to afford readers a better understanding of blooRTLs [1].

# 2 Language Tutorial

blooRTLs was developed on Linux Ubuntu 14.04. The following section provides a tutorial on how to run programs with the precompiler. Although the precompiler does not print out VHDL (that is the job of the compiler), it is an effective means of testing an RTL algorithm or spec. Both the precomiler and compiler are discussed in Section 4

## 2.1 Running the Precompiler

To compile and run the precompiler execute the following:

0. Extract the `tar.gz` file
1. Navigate to the working directory in the terminal
2. Run the make file in the terminal (`$ make`)
3. Execute `$ ./blooRTLs<tutorial1.txt` (Note: `tutorial1.txt` contains the blooRTLs source code of Tutorial 1, which is decribed in the next subsection)

## 2.2 Tutorial Example 1: Addition of Objects

A quick tutorial on the language is provided through the following addition example:

Listing 1: Tutorial Example 1: Addition of Objects

```
BINMAP var {
        msb := [7];
        lsb := [0];
        middle2bits:= [4][3];
}

var := 10000001;

var.middle2bits := var.msb * var.lsb;

PRINT var middle2bits;
PRINT var;
```

In Tutorial Example 1, the terminology is described as follows.

- `var` is denoted as the *root variable*
- `msb`, `lsb`, and `middle2bits` are denoted as the `objects` of `var`.
- The `BINMAP` declaration, then, is used to declare the bit indices that the objects will represent.

Using the terminology, as described above, the `BINMAP var` declaration, for example, assigns object `lsb` to bit 0 of the root variable `var`. By the same principle, `msb` is assigned bit 7 of `var`, and `middle2bits` is assigned bit 4 of `var`. This concept is illustrated in the figure **??**

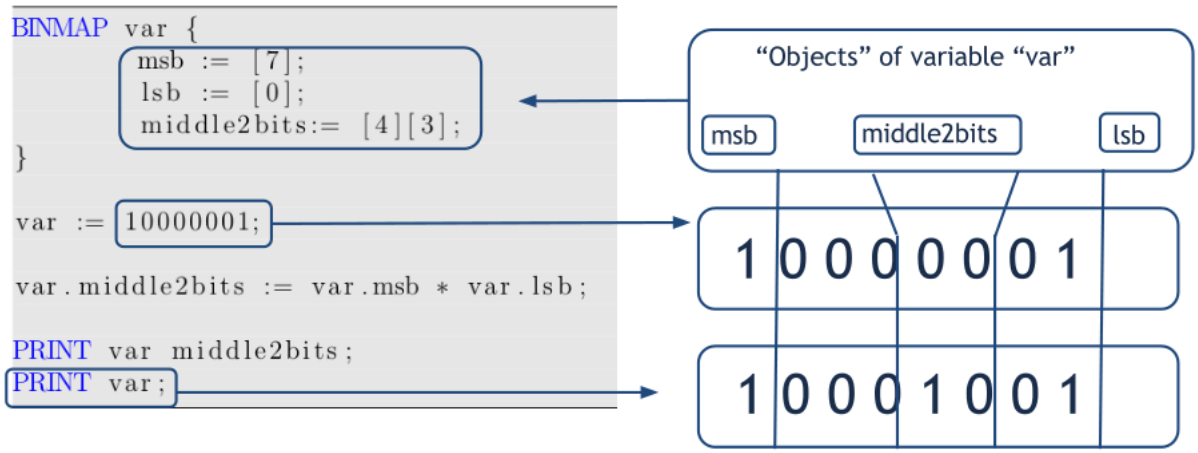


Figure 1: Objects of Tutorial Example 1

It is worth noting that `BINMAP` can be called multiple times throughout the program as long as it does not attempt to change the indicies of a previously assigned object. Additionally, `BINMAP` can be invoked on multiple root variables. In these cases, multiple root variables are permitted to use the same object identifiers because objects are unique root variables (i.e. `var.obj` and `var2.obj` are permissable if they were to be implemented into the same program).

Additionally, it is worth noting that whitespace within an index yields unexpected results. (i.e. `nibble0 := [3 ][ 2][ 1 ] [0 ]` is not the same as `nibble0 := [3][2][1][0]`, so it behooves users to avoid using the former representation).

The first assignment following `var`'s `BINMAP` declaration is a *load*. In particular, `var` is loaded with the bits `10000001`. When this occurs, the blooRTLs precompiler enters a dynamic check and assign routine to confirm that the overall assignment is valid. The checking routine for this assignment is described as follows.

- First, the precompiler checks if the indices of the previously declared objects fall within the dimensions of the root variable. For Tutorial Example 1, the precompiler asks if the indices of `lsb`, `msb`, and `middle2bits` fall within the indices of the root variable `var`.
- Upon the passing the frst check, the precompiler assigns the root variable with the bits, `10000001`.
- The dynamic assignments then occur. The precompiler iterates through the objects of the variable that was just updated (in this case `var` was just updated with `10000001`) and assigns new values to the objects whose indices fall within the updated variable. In the case of Tutorial Example 1, when `var` is assigned with `10000001`, `var.lsb` is assigned `1`, `var.msb` is assigned `1`, and `var.middle2bits` is assigned `00`.
- The same dynamic assinment routine would occur if an object (as opposed to a variable) were to be updated. If this were the case, the precompiler would iterate through all other objects in addition to the root variable and make assignment changes as necessary. However, Tutorial Example 1 does have any of these object assignments. (to refresh, an example object assignment would be `var.msb := 0`

It is worth noting that the blooRTLs language supports up to 32-bit assignments. Unexpected behavior occurs beyond when variables are assigned dimensions beyond this length.

It is also worth noting that the blooRTLs precompiler raises errors if overflow and underflow occur (i.e. if `var.lsb := 2d` [Note: 2d implies "2 as a decimal"], then `var.lsb` would overflow because it embodies a 1 bit number.

Running Example 1 with the precompiler outputs the following:

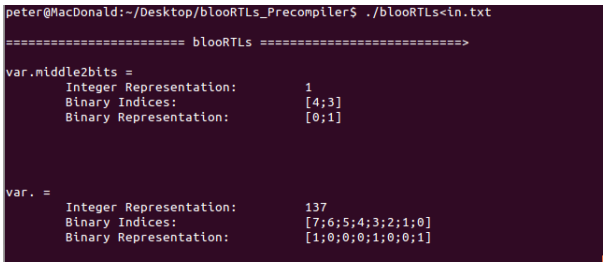


Figure 2: Executing Tutorial Example 1

Tutorial Example 2 is provided in the `tar.gz` file in addition to the Tutorial Example 1. The source is seen as follows:

Listing 2: Tutorial Example 2: DNA Sequencing

```
BINMAP var1 {
  nucleotide := [1][0];
}

var1 := 10011100000000100110011011110101;
elsedummy := 0;
adenosine := 000000;
cytosine := 000000;
thymine := 000000;
guanine := 000000;

REPEAT (
    IF (var1.nucleotide = 00) THEN (
      adenosine := adenosine + 1d;
    ) ELSE (elsedummy := 0;)

    IF (var1.nucleotide = 01) THEN (
      cytosine := cytosine + 1d;
    ) ELSE (elsedummy := 0;)

    IF (var1.nucleotide = 10) THEN (
      thymine := thymine + 1d;
    ) ELSE (
      guanine := guanine + 1d; )

    var1 := var1 >> 2d;
) UNTIL (var1 = 0d)

        PRINT adenosine;
        PRINT cytosine;
        PRINT thymine;
        PRINT guanine;
```

# 3 Language Reference Manual

## 3.1 Lexical Conventions

The lexical conventions implemented in blooRTLs consist of comments, identifiers, keywords, and constants.

### 3.1.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

### 3.1.2 Identifiers (Names)

An identifier is a sequence of letters and digits used to represent a register in the RTL design; all characters must be lower case.

### 3.1.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise (case-sensitive):

| | |
|---|---|
| `BINMAP` | `THEN` |
| `REPEAT` | `ELSE` |
| `UNTIL` | |
| `IF` | `PRINT` |

### 3.1.4 Constants

There are two types of constants, as follows:

- **Binary Constants**: A binary constant is a series of ones (1) and zeros (0).
- **Integer Constants**: An integer constant is a series of numerical digits (0-9), followed by the letter `d` (i.e. 4d) with no preceding sign character.

## 3.2 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in the `typewriter` style.

## 3.3 What's in a Name?

- blooRTLs suports one fundamental identifier type, which takes the form of the binary constant
- `BINMAP`: the `BINMAP` type creates objects, as defined in Section 3.4, from identifiers

## 3.4 Identifiers and Objects

Identifiers and Objects are related and defined, as follows:

- *identifier*: An identifier is interpreted as a binary constant that is assumed to be loaded into a region of storage within the RTL design (i.e. a register)
- *identifier.object*: An object is an indexed component of the Identifier.

Since identifiers are used to represent storage components within the data path of the RTL design, all identifiers are technically objects. However, identifiers and objects are not fully equivalent structures. When identifiers are segmented into smaller regions of storage, these resulting components are referred to as the objects. The means of partitioning identifiers into objects is discussed in Section 3.7.

## 3.5 Conversions

blooRTLs has one conversion operator:

### 3.5.1 The decimal operator (d)

`d` is shorthand for decimal. It's implementation is as follows:

```
var := 9d;
/* var holds the binary constant value of 1001 */
```

## 3.6 Expressions

### 3.6.1 Primary Expression

*primary-expression :*
*identifier*
*constant*
*(expression)*

- An *identifier* is a primary expression, provided it has been suitably declared as discussed in Section 3.7.
- A *constant* is either a binary or an integer constant, as specified in Section 3.1.4.
- An *(expression)* is an expression that takes the highest precedence.

### 3.6.2 Unary Expression

Unary expressions group left to right and are listed as follows

*expression* `+` *expression* : binary constant add
*expression* `-` *expression* : binary constant subtract
*expression* `*` *expression* : binary constant multiply
*expression* `>>` *expression* : binary constant shift right
*expression* `<<` *expression* : binary constant shift left

### 3.6.3 Assignment

*identifier* `:=` *expression*

### 3.6.4 Relational

*expression* `=` *expression* : returns boolean `true` if equal

## 3.7 Declarations

Declarations take the general form indicated, as follows:

*declaration:*
*decl-specifiers declarator-list*

*decl-specifiers:*
`BINMAP`
*expression*

*declarator-list:*
*declarator*
*declarator, declarator-list*

*declarator:*
`identifier`
`identifier.object`
`declarator( )`
`(declarator)`

### 3.7.1 BINMAP

The `BINMAP` *decl-specifier* creates an object within an identifier, which enables usage of the `identifier.object` *declarator*. This idea is illustrated in the following example:

```
/* Within the identifier ‘‘var’’, BINMAP is used to declare an
   object called ‘‘LSB’’ that indexes bit location 0, and ‘‘MSB,’’
    which indexes bit location 4 */
BINMAP var{
        lsb := [0];
        msb := [4];
}

var := 1000;
var.lsb := 1;
var.msb := 0;
/* var now holds the binary value 0001 */
```

## 3.8 Statements

Statements are executed in sequence.

### 3.8.1 Expression statement

Most statements are expression statements, which have the following form:

*expression;*

Usually expression statements are assignments or function calls.

### 3.8.2 Compound statement

The use of several statements, where one is expected, is noted as follows:

*compound-statement:*
    *statement*

*statement-list:*
    *statement*
    *statement statement-list*

### 3.8.3 Conditional statement

The two forms of the conditional statement are listed as follows:

`if` ( *expression* ) *statement*
`if` ( *expression* ) *statement* `else` *statement*

### 3.8.4 repeat until statement

The `repeat until` takes the following form:

`repeat` ( *expression* ) `until` ( *expression* )

## 3.9 Scope

blooRTLs source text must be kept in a single file.

# 4 Project Plan

Disclaimer: the compiler is not working, however, the following section takes into account various considerations for the compiler's implementation.

## 4.1 VHDL Library Considerations

When VHDL began, the main logic type was called the `std_logic_vector`, which embodies a sequence of bits that hold no numerical value. As a result, in the early 1990s, Synopsys, an ASIC design tool vendor, developed three libraries that could assign numerical values to the `std_logic_vector` with the hope of easing the arithmetic syntax in VHDL. The libraries are listed as follows:

Listing 3: Non-standardized VHDL libraries by Synopsys

```
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_signed.all;
```

Although these libraries ease the aritmetic syntax of VHDL, these libraries are mistaken to have been standardized by the IEEE. As a result, the various tools and platforms offered by vedors interpret these libraries differently in simulation and synthesis.

The better option is to use the IEEE's standardized libraries as follows:

Listing 4: Standardized IEEE VHDL libraries

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

Because these libraries have been standardized, they perform (for the most part) identically regardless of vendor. However, interpreting blooRTLs into VHDL is not trivial. The following arithmetic examples in the numeric library raise errors when compiled.

Listing 5: Error: Adding with different lengths

```
-- sum is length 1 downto 0
-- a is length 1
-- b is length 1
sum <= std_logic_vector( ("0" & a) + ("0" & b) );
```

Listing 6: Error: Multiplying with different lengths

```
-- product is length 4 downto 0
--    (must be twice the length of its operands)
-- a is length 1
-- b is length 1
product <= std_logic_vector( ("0" & a) * ("0" & b) );
```

Oddly, the following two scenarios work

Listing 7: Success: Adding with different lengths

```
-- sum is length 1 downto 0
-- a is length 2
-- b is length 1
...
sum <= std_logic_vector( a + ("0" & b) );
```

Listing 8: Success: Multiplying with different lengths

```
-- product is length 3 downto 0
--    (must be twice the length of its operands)
-- a is length 2
-- b is length 1
product <= std_logic_vector( a * ("0" & b) );
```

But these two successes do not cover all cases. In order to cover all cases, a clever trick was implemented.

Listing 9: Clever "dummy" Trick to cover all multiplication cases

```
-- product is length 3 downto 0
-- one_2 := "01"
-- a is length 1
-- b is length 1
product <= std_logic_vector( one_2 * ("0" & a) * ("0" & b) );
```

Listing 10: Clever "dummy" Trick to cover all sum cases

```
-- sum is length 1 downto 0
-- zero_2 := "00"
-- a is length 1
-- b is length 1
sum <= std_logic_vector( zero_2 + ("0" & a) + ("0" & b) );
```

Obviously, the `zero_2` and `one_2` can be replaced by `zero_N` and `one_N` depending on the situation. While this is quirky on behalf of the VHDL compiler, this trick must be used when necessary, considering the reliability of the numeric library at the simulation and synthesis stages.

## 4.2 Example: Compiling a simple blooRTLs program

The following diagram walks through the compiling of a blooRTLs program:

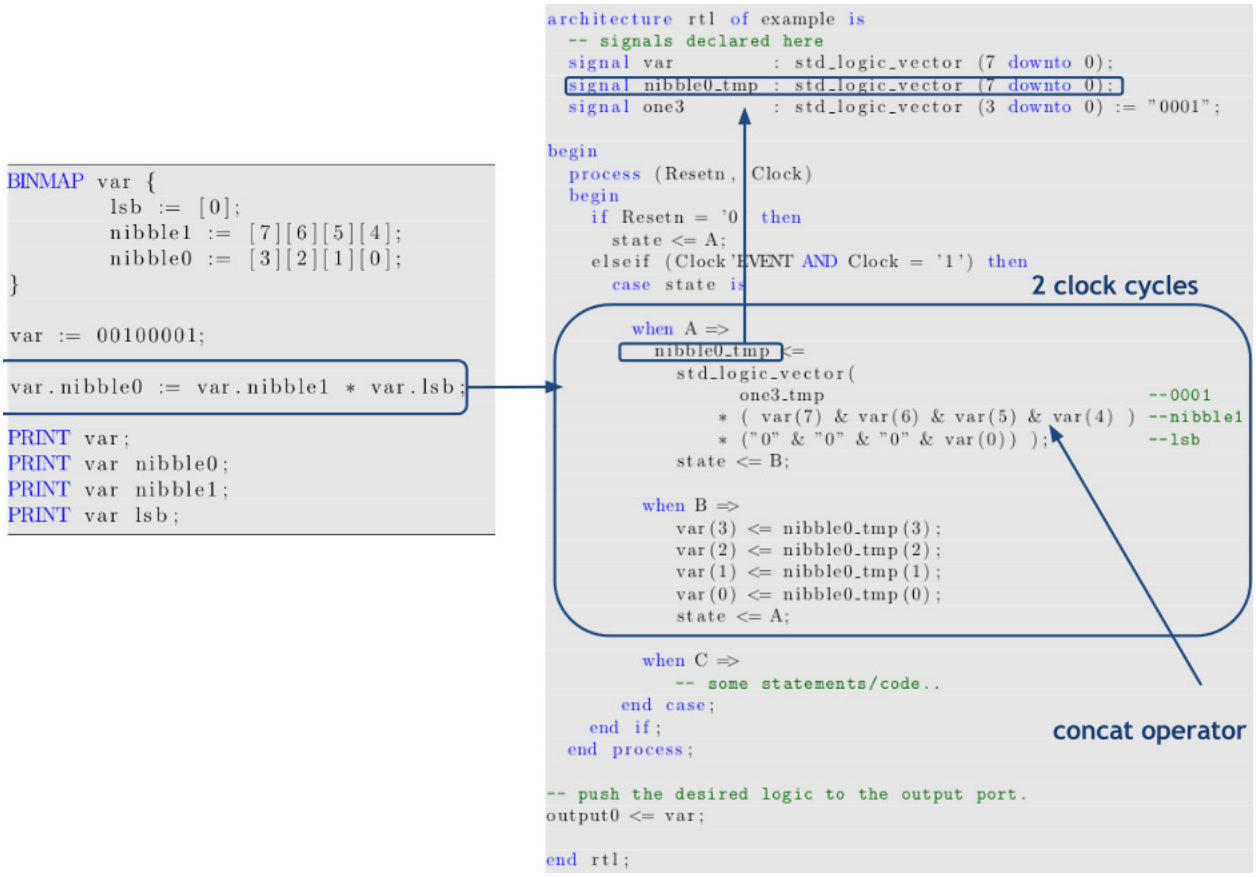


Figure 3: Compiling a simple blooRTLs program

## 4.3 Sequential VHDL Consideration

Listing 11: Sequential VHDL Framework

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity moore is
  port (
    Clock       : in STD_LOGIC;
    Resetn      : in STD_LOGIC;

    -- declare all inputs/outputs & vector length
    -- Arbitrary examples to illustrate syntax:
    input0      : in std_logic_vector (7 downto 0);
    input1      : in std_logic;          -- 1 bit
    output0 : out std_logic_vector(2 downto 0));

end moore;

architecture rtl of moore is
  type State_type is (A,B,C); --Declare RTL states
  signal state: State_type;

  -- declare all signals & vector lengths
  -- Arbitrary examples to illustrate syntax:
  signal signal0 : std_logic_vector (5 downto 0);
  signal signal1 : std_logic_vector (2 downto 0) := "000";
  signal signal2: std_logic := '1';

begin
  process (Resetn, Clock)
  begin
    if Resetn = '0' then
      state <= A;
    elsif (Clock'EVENT AND Clock = '1') then
      case state is

        --'A' State
        when A =>
          if ( ... ) then
            -- some statements/code...
            state <= A;
          else
```

```
            state <= B;
          end if;

         --'B' State
         when B =>
            -- some statements/code..

         --'C' State
         when C =>
            -- some statements/code..
       end case;
    end if;
  end process;

-- push the desired logic to the output port. Arbitrary example
output0 <= "001" when state = C else "000";

end rtl;
```

## 4.4 Precompiler

When using the VHDL numeric library, the VHDL compiler does not detect overflows, nor does it detect underflows. Hence the need for a precompiler.

The goal of the precompiler was to cache and log the vector lengths of all the variables and objects. The basic architecture of this "database" takes the following form:

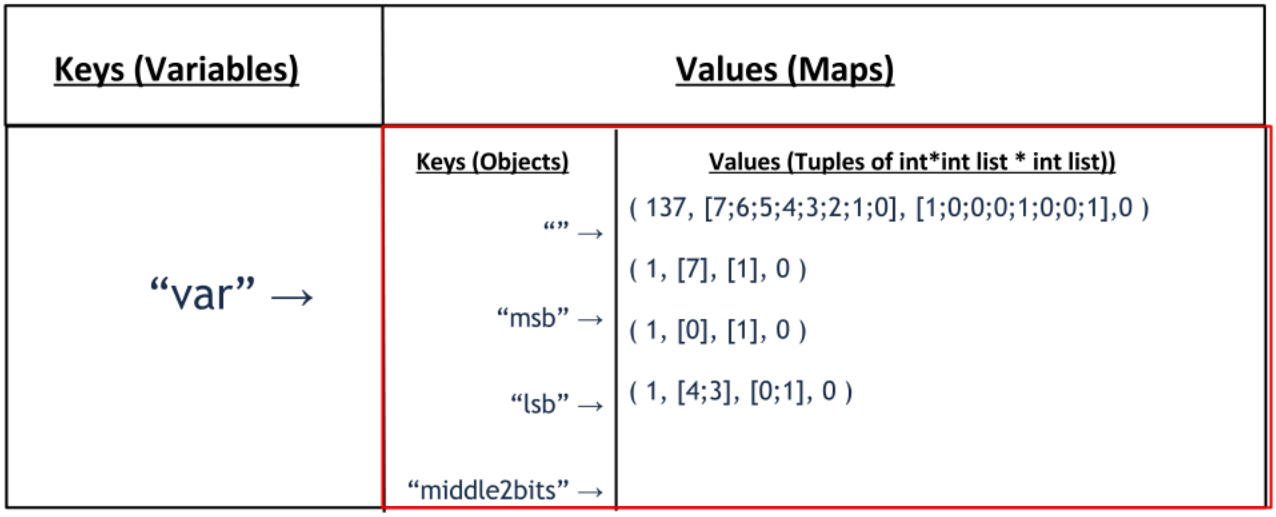


Figure 4: Ocaml "Database" of Vars and Objs

```
# let binmap = mapUpdate "var" "" ( 137, [7;6;5;4;3;2;1;0], [1;0;0;0;1;0;0;1],0 ) binmap;;
val binmap : (int * int list * int list * int) NameMap.t NameMap.t = <abstr>
# NameMap.find "" (NameMap.find "var" binmap);;
- : int * int list * int list * int =
(137, [7; 6; 5; 4; 3; 2; 1; 0], [1; 0; 0; 0; 1; 0; 0; 1], 0)
#
```

Figure 5:

# 4. Project Plan

## 4.1 Planning, Specification, and Development

All members of our team contributed to all parts of the project equally according to their responsibilities. Originally, because we understand the summer term is short, we plan to meet every time after class, every Friday or even on weekends if necessary. However, after a few weeks, after we divided roles, we figure out that it would be more efficient to work individually and frequently update progress to group members via email and cellphone.

## 4.2 Programming Style and Development Environment Used

Each group member uses different computer system environment, some of us use Linux for faster coding and some other use Windows for development. All of us use OCaml, OCamllex, and OCamlyacc for design and develop our pre-compiler. Two of members in our group are familiar with VHDL, and the other two are willing to learn VHDL while developing the project.

## 4.3 Team responsibilities

| | |
|---|---|
| **Peter Burrows** | Designer, actively contribute to all processes |
| **Apurv Gaurav** | Test Designer, Debug |
| **Pinhong He** | Scanner ,Parser, AST |
| **Zhibo Wan** | Manager, Lexical Analysis |

## 4.4 Project Timeline

| | |
|---|---|
| **6/3/15** | Proposal Submitted |
| **6/8/15** | Scanner |
| **6/10/15** | Parser |
| **6/11/15** | AST |
| **6/14/15** | Pre-complier done |
| **6/30/15** | Test, Debug |
| **7/2/15** | Final report, slices submitted |

What I learnt:

**Pinhong He:** I really enjoy learning from scratch to how to actually design and develop a compiler. Although I was totally unfamiliar with OCaml and it took me a while to understand it, however I really like it after I see how it works and helps to develop our language compiler. For the teamwork, I like working with my teammates, although some of us have time conflicts and a busy schedule, but it all worked out at the end. It is a pity that we could not finish what we expected at the beginning, but if we got more time, I believe our language will be useful!

**Zhibo Wan** learned:

Computer Science is more than just coding. And this project “forced” us to jump out of the mainstream code such as C++ , Java, python, and let me have a big picture of how programing language , as a whole, looks like.  Ocmal is not that bad,  it can do some work amazingly efficient, especially recursion function.

# Test Cases

The blooRTLs compiler accepts a blooRTLs program and generates a VHDL program (.v output file and the test bench file). Our system test suite takes a list of simple, representative blooRTLs programs and compiles those programs into their runtime output and translates the programs to the appropriate VHDL files (the actual .v output file and the test bench file). These VHDL output files and their corresponding test bench are then run to produce a waveform. If the VHDL program files are correct, the runtime output and generated waveform will match with the expected waveform. Hence, we are system testing our compiler after integrating all its parts together.

I have divided the test cases as follows. The arithmetic test cases individually test the various operations of our program, from basic addition, subtraction, and multiplication to shifting. The binary map test cases will illustrate one of blooRTLs key features, the ability to map a binary sequence into its bits and the ability to manipulate those bits. Finally, our feature test case will combine the various features of the language into a real life application - mapping and encoding our DNA!

1. **Arithmetic Test Cases**

**AddBinary:**

**Operation:**

5 + 4 = 9; Note that here, I created a third register var3 to hold the sum of var1 + var2. Creating a third register allows blooRTLs to pad var1 and var2 with an extra "zero" on its MSB so that we don't get an overflow, since 9 is $(1001)_2$ which is one extra bit than either var1, which is $(101)_2$ or var2, which is $(100)_2$.

**blooRTLs:**

```
var1 := 5d;
var2 := 4d;
var3 := var1 + var2;
```

**VHDL:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity AddBinary is
        Port (Num1 : in STD_LOGIC_VECTOR (4 downto 0) := "00000"
                Num2 : in STD_LOGIC_VECTOR (4 downto 0) := "00000"
                Sum : out STD_LOGIC_VECTOR (4 downto 0));
end AddBinary;
architecture Structure of AddBinary is
begin
Sum <= Num1 + Num2;
end Structure;
```

**Testbench:**

blooRTLs enters in 0101 for A and 0100 for B, automatically padding A and B with an extra zero

**AddBinary - Overflow Error Detection:**

**Operation:**

5 + 4 = 9; Instead of creating a third register var3 to hold the sum of var1 + var2 as in the previous test case, I will simply let var1 = var1 + var2. This will cause an overflow error, since the sum, 9, is (1001), which is one more bit than what var1 could store. Since var1

since the sum, 9, is $(1001)_2$ which is one more bit than what var1 could store. Since var1 was originally created to store just three bits, for $(101)_2$.Were it not for blooRTL to point out the error, VHDL would run anyway producing a resulting sum that would be truncated to not reflect the fourth digit. In this blooRTLs is able to inform the user of semantic errors in a code that VHDL could not do.

**blooRTLs:**

```
var1 :=5d;
var2 := 4d;
var1 := var1 + var2;
PRINT var1;
```

**SubtractBinary:**

**Operation:**

5 - 4 = 1; Since I am subtracting a smaller number from a larger one, I know that the number of bits required to store the difference is less than or equal to the number of bits to store either 5 or 4. Thus, I can simply do var1 = var1 - var2.

**blooRTLs:**

```
var1 := 5d;
var2 := 4d;
var1 := var1 - var2;
PRINT var1;
```

**VHDL:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;;
entity SubtractBinary is
        Port (Num1 : in STD_LOGIC_VECTOR (3 downto 0) := “0000”;
                Num2 : in STD_LOGIC_VECTOR (3 downto 0) := “0000”;
                Diff : out STD_LOGIC_VECTOR (3 downto 0));
end SubtractBinary;
architecture Structure of SubtractBinary is
begin
Diff <= Num1 - Num2;
end Structure;
```

**Testbench:**

I will enter in 101 for A and 100 for B

**Multiplication:**

**Operation:**

5 * 4 =; I will have a third variable, var3 store the product. var3 will automatically have double the number of bits as the factors.

**blooRTLs:**

```
var1 = 5d;
var2 = 4d;
var3 = var1 * var2;
PRINT var3;
```

**VHDL:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Multiplication is
Port (Num1 : in STD_LOGIC_VECTOR (4 downto 0) := “00000”;
        Num2: in STD_LOGIC_VECTOR  (4 downto 0) := “00000”;
                Product: in STD_LOGIC_VECTOR  (4 downto 0))
end Multiplication;
architecture Structure of Multiplication is
```

architecture Structure of Multiplication is
begin
Product <= Num1 * Num2;
end Structure;

**Testbench:**

I will enter in 00101 for A and 00100 for B

**Multiplication Overflow Error:**

**Operation:**

It is important to note that once variables are created in blooRTLs, the variables will always maintain the same number of bits as when it was created. Thus, if I store 5 in var1, and 4 in var 2, both var1 and var2 will contain 3 bits each.

**blooRTLs:**

```
var1 = 5d;
var2 = 4d;
var1 = var1 * var2;
PRINT var1;
```

**Shifting a sequence left**

**Operation:** Taking the sequence 10001100 and shifting it 3 bits to the left would give us 01100000. Zeroes are automatically padded in to maintain the bit length.

**blooRTs:**

```
var1 := 10001100
var1 := var1<<3d;
PRINT var1;
```

**Shifting a sequence right**

**Operation:** Taking the sequence 10001100 and shifting it 3 bits to the right would give us 00010001. Zeroes are automatically padded in to maintain the bit length.

**blooRTs:**

```
var1 := 10001100
PRINT var1;
var1 := var1>>3d;
PRINT var1;
```

**Up Counter**

**Operation:**

This blooRTLs program will count up from 0 to 15 with a change in number every x seconds. The blooRTLs language is designed such that each line of the code takes x seconds sequentially, but with the INPUT and OUTPUT lines being read first. Thus, in the following code, the number display changes to display + 1, every x seconds.

**blooRTLs:**

```
INPUT datastream := 0000;
display := datastream;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
```

```
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
display := display + 1d;
OUTPUT outstream := display
```

**Down Counter**

**Operation:**

This is like the Up Counter except the program counts down.

**blooRTLs:**

```
INPUT datastream := 0000;
var := datastream;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
var := var - 1;
OUTPUT outstream := var
```

## II. Binary Map Cases

**BINMAP - Changing the indices of a binary string**

**Operation:** We take a binary number, 10001110, map the first four indices as 'nibble' using BINMAP, and change those bits so that they are 0000. The output is 10000000, as expected

**blooRTLS:**

```
BINMAP var1 {
  nibble:=[3][2][1][0];
}
var1 := 10001110;
var1.nibble := 0000;
```

**BINMAP - Left-to-Right Sequence Reader**

**Operation:** Using the fact that each line of the blooRTL, excluding the Input and Output declaration, takes x seconds and flows sequentially, we can build a sequence reader so that the sequence is "read" one or more bits at a time. Here, the program takes the sequence 10001110 and reads it two bits at a time from the left to right. Thus, the program displays 10 the first x seconds, 00 the next x seconds, 11 the next seconds, and 10 the final x seconds.

**blooRTL:**

```
INPUT datastream := 10001110;
BINMAP sequence {
  display:=[7][6];
}
```

```
}
sequence := datastream;
sequence.display := sequence.display << 2;
sequence.display := sequence.display << 2;
sequence.display := sequence.display << 2;
OUTPUT outstream := sequence.display;
```

**BINMAP - Left-to-Right Sequence Reader Using Repeat-Until Loop**

**Operation:** We will once again build a program that takes the sequence 10001110 and reads it two bits at a time from the left to right. However, instead of relying on the computer's time per line to display the count, we will use the repeat-until loop with each iteration printing two bits.

**blooRTL:**

```
BINMAP sequence {
  display:=[7][6];
}
sequence := 10001110;
PRINT sequence.display;

REPEAT
(    sequence.display := sequence.display << 2;
   PRINT sequence.display;
)
UNTIL (sequence = 00000000)
```

**BINMAP - Overflow Error Detection:**

**Operation:**

We take a binary number, 10001110, map the first four indices as 'nibble' using BINMAP, and try to change those bits so that they are 10000. However, since 10000 is five bits instead of the four allotted for 'nibble', the compiler will notify the user of an overflow

**blooRTLs:**

```
BINMAP var1 {
  nibble := [3][2][1][0];
}
var1 := 10001110;
var1.nibble := 10000;
```

**BINMAP - Floating Point Multiplication:**

**Operation:**

Computers often convert large numbers into floating point form to perform multiplication with less running time. The following program demonstrates the ease with which blooRTL can be used to convert two integers into floating point using BINMAP, and then be multiplied. The decimal equivalent of the two floating point numbers are 1224 and 45. Their product, 55080, will be shown in floating point form as 01000111010101110010100000000000.

**blooRTL:**

```
fp1 := 01000100100110010000000000000000; // decimal equivalent = 1224
fp2 := 01000010001101000000000000000000; // decimal equivalent = 45
/*
Floating Point Numbers is of the form (+/- 1.Mantissa * 2^exponent), with sign = 0 for
positive and sign = 1 for negative. As an example, fp1's sign is 0, its exponent 10001001,
and Mantissa as 00110010000000000000000. This translates to the number 1224 in
decimal representation
*/
```

```
BINMAP var1 {
   sign := [32];
   exponent := [31][30][29]..[23];
   mantissa := [22][21]..[0];
}
BINMAP var2 {
   sign := [32];
   exponent := [31][30][29]..[23];
   mantissa := [22][21]..[0];
}
BINMAP out {
   sign := [32];
   exponent := [31][30][29]..[23];
   mantissa := [22][21]..[0];
}

out := BINARY (0) // In order to Initialize the product variable
var1: = fp1;
var2: = fp2;
out.exponent := var1.exponent + var2.exponent;
out.sign := var1.sign XOR var2.sign;
PRINT out;
```

III. **Argumentative Test Cases**

**Up Counter Using Repeat-Until Loop**

**Operation:**

Once again the blooRTLs program will count up from 0 to 15. However, instead of using the delay of reading each line to display the count, we will use a repeat-until loop.

**blooRTLs:**

```
data := 0000
REPEAT
(
        data := data + 1d;
        PRINT data;
)
UNTIL (data = 1111)
```

**Down Counter Using Repeat-Until Loop**

**Operation:**

Once again the blooRTLs program will count up from 0 to 15. However, instead of using the delay of reading each line to display the count, we will use a repeat-until loop.

**blooRTLs:**

```
data := 1111
REPEAT
(
        data := data - 1d;
        PRINT data;
)
UNTIL (data = 0000)
```

**If-Then-Else Switch statement:**

**Operation:**

Here we will design a situation where a binary number is read. Then using If-Then-Else statements, we design cases for what the number could be. Please note that since an 'Else' is compulsory in an If-Then-Else statement, we will use a dummy variable to fill in an

is compulsory in an If-Then-Else statement, we will use a dummy variable to fill in an 'Else' condition when we do not nee

**blooRTLs:**

```
var1 := 01;
zero := 0;
one := 1;
two := 2;
three := 3;
elsedummy := 0;
IF (var1 = 00) THEN
(PRINT zero;) ELSE (elsedummy := 0;)
IF (var1 = 01) THEN
(PRINT one;) ELSE (elsedummy := 0;)
IF (var1 = 10) THEN
(PRINT two;) ELSE (elsedummy := 0;)
IF (var1 = 11) THEN
(PRINT three;) ELSE (elsedummy := 0;)
```

**The One's Counter**

**Operation:**

We will count the number of 1's in a sequence by using an if-then statement with counter embedded inside a repeat-until loop

**blooRTLs:**

```
BINMAP var1 {
  lsb := [0];
}
var1 := 10001110;
counter := 000
REPEAT (
IF (var.lsb = 1) THEN
        (
    counter = counter + 1d;
        var1 := var1 >> 1d;
        )
)
ELSE
    (var1 := var1 >> 1d;)
)
UNTIL (var1 = 00000000)
PRINT counter;
```

**Sequence Detection (Overlapping):**

**Operation:**

Given the sequence 1101110101, detect 1011

**blooRTLs**

```
BINMAP var1 {
  detectbits := [3][2][1][0];
}
var1 := 1101110101;
counter := 000;
REPEAT (
        IF (detectbits = 1011)
        THEN (
                counter := counter + 1d;
            var1 := var1 >> 1d;
                )
        ELSE (var1 := var1 >> 1d;)
        )
```

```
        )
UNTIL (var1 = 0d)
PRINT counter;
```

## IIII. **Feature Test Case**

**DNA Nucleotide Counter**

**Operation:**

This is a part of our featured test case. DNA uses four nucleotides - Adenosine (A), Cytosine (C), Thymine (T), and Guanine (G) in building a sequence that is our genetic code. We will take a DNA sequence that has been encoded into binary numbers and count how many of each nucleotide there are. Adenosine is encoded as 00, Cytosine as 01, Thymine as 10, and Guanine as 11.

**blooRTLs**

```
BINMAP var1 {
 nucleotide := [1][0];
}
var1 := 0110011011110101;
elsedummy := 0;
adenosine := 000000;
cytosine := 000000;
thymine := 000000;
guanine := 000000;
REPEAT (
  IF (var1.nucleotide = 00) THEN (
    adenosine := adenosine + 1d;
  ) ELSE (elsedummy := 0;)
  IF (var1.nucleotide = 01) THEN (
    cytosine := cytosine + 1d;
  ) ELSE (elsedummy := 0;)
  IF (var1.nucleotide = 10) THEN (
    thymine := thymine + 1d;
  ) ELSE (
    guanine := guanine + 1d; )
   var1 := var1 >> 2d;
) UNTIL (var1 = 0d)
  PRINT adenosine;
  PRINT cytosine;
  PRINT thymine;
  PRINT guanine
```

**Optimized DNA Encoder:**

**Operation:**

In this program, we will use the results of the previous program, the count of each nucleotide in a DNA sequence to re-encode the DNA. The re-encoding is based on the famous Huffman algorithm, which allows for more efficient encoding, minimizing the number of bits to store the same information. In order to do that, I will encode the nucleotide with the highest frequency with ‘0’, the second highest frequency with ‘10’ the third highest frequency with ‘110’ and the lowest frequency nucleotide with ‘111’. For purposes of this test case, let’s assume the highest to lowest frequency of nucleotides were cytosine, adenosine, thymine, guanine.

**blooRTLs:**

```
BINMAP oldseq {
  nucleotide := [1][0];
}
oldseq := 0110011011110101;
BINMAP newseq {
 cbits := [16];
```

```
 tbits := [17][16];
 gbits := [18][17][16];
 abits := [18][17][16];
}
newseq := 0d;
seqlength := 0d;
elsedummy := 0;
REPEAT (
        IF (oldseq.nucleotide = 01)
        THEN (
                newseq.cbits := 0;
           oldseq := oldseq >> 2d;
           newseq := newseq >> 1d;
           seqlength := seqlength + 1d;
                )
        ELSE (elsedummy := 0;)

        IF (oldseq.nucleotide = 10)
        THEN (
           newseq.tbits := 10;
           oldseq := oldseq >> 2d;
           newseq := newseq >> 2d;
           seqlength := seqlength + 2d;
           )
        ELSE (elsedummy := 0;)

        IF (var1.nucleotide = 00)
        THEN (
           newseq.abits := 110;
           oldseq := oldseq >> 2d;
           newseq := newseq >> 3d;
           seqlength := seqlength + 3d;
           )
        ELSE
           (
           newseq.gbits := 111;
           oldseq := oldseq >> 2d;
           newseq := newseq >> 3d;
           seqlength := seqlength + 3d;
           )
UNTIL (var1 = 0d)
PRINT newseq;
PRINT seqlength;
```

# References

[1] Ritchie, Dennis M. *C Reference Manual*, Bell Telephone Laboratories, Murray Hill, New Jersey 07974 1973.

[2] http://parallelpoints.com/numbers-in-vhdl/

[3] http://www-micro.deis.unibo.it/ drossi/Dida02/lezioni/IEEE_Standard_Packages.pdf

# A Appendix

Listing 12: scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| ">>" { SHIFTR }
| "<<" { SHIFTL }
| ":=" { ASSIGN }
| "=" { EQU }
| ';' { SEMI }
| '(' { LP }
| ')' { RP }
| '{' { LCB }
| '}' { RCB }
| '.' { PERIOD }
| ['0' - '9']+'d' as decimal { DEC (decimal) }
| ['0' - '1']+ as bits { BITS (bits) }
| ['a'-'z']['a'-'z' '0'-'9']+ as lxm { ID(lxm) }
| "PRINT" { PRINT }
| "INPUT" { INPUT }
| "REPEAT" { REPEAT }
| "UNTIL"  { UNTIL }
| "IF"           { IF }
| "THEN"         { THEN }
| "ELSE"         { ELSE }
| "BINMAP" { BINMAP }

| '['['0'-'9']']' as ind
        { IND(int_of_char ind.[1] - 48) }
| '['['0'-'9']['0'-'9']']' as ind
        { IND((10*(int_of_char ind.[1] - 48))
                        +((int_of_char ind.[2] - 48)) ) }
| '['['0'-'9']['0'-'9']['0'-'9']']' as ind
        { IND((100*(int_of_char ind.[1] - 48))
                        +(10*(int_of_char ind.[2] - 48))
                        +(int_of_char ind.[3] - 48) ) }
| '['['0'-'9']['0'-'9']['0'-'9']['0'-'9']']' as ind
        { IND((1000*(int_of_char ind.[1] - 48))
                  +(100*(int_of_char ind.[2] - 48))
                   +(10*(int_of_char ind.[3] - 48))
                    +((int_of_char ind.[4] - 48)) ) }
| '['['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']']' as ind
```

```
        { IND((10000*(int_of_char ind.[1] - 48))
                  +(1000*(int_of_char ind.[2] - 48))
                   +(100*(int_of_char ind.[3] - 48))
                    +(10*(int_of_char ind.[4] - 48))
                        +((int_of_char ind.[5] - 48)) ) }
| '['['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']']' as ind
        { IND((100000*(int_of_char ind.[1] - 48))
                  +(10000*(int_of_char ind.[2] - 48))
                   +(1000*(int_of_char ind.[3] - 48))
                    +(100*(int_of_char ind.[4] - 48))
                        +(10*(int_of_char ind.[5] - 48))
                        +((int_of_char ind.[6] - 48)) ) }
| '['['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']']'
    as ind
        { IND((1000000*(int_of_char ind.[1] - 48))
                  +(100000*(int_of_char ind.[2] - 48))
                   +(10000*(int_of_char ind.[3] - 48))
                    +(1000*(int_of_char ind.[4] - 48))
                        +(100*(int_of_char ind.[5] - 48))
                        +(10*(int_of_char ind.[6] - 48))
                        +((int_of_char ind.[7] - 48)) ) }
|
   '['['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']']'
    as ind
        { IND((10000000*(int_of_char ind.[1] - 48))
                  +(1000000*(int_of_char ind.[2] - 48))
                   +(100000*(int_of_char ind.[3] - 48))
                    +(10000*(int_of_char ind.[4] - 48))
                        +(1000*(int_of_char ind.[5] - 48))
                        +(100*(int_of_char ind.[6] - 48))
                        +(10*(int_of_char ind.[7] - 48))
                        +((int_of_char ind.[8] - 48)) ) }
|
   '['['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['
    as ind
        { IND((100000000*(int_of_char ind.[1] - 48))
                  +(10000000*(int_of_char ind.[2] - 48))
                   +(1000000*(int_of_char ind.[3] - 48))
                    +(100000*(int_of_char ind.[4] - 48))
                        +(10000*(int_of_char ind.[5] - 48))
                        +(1000*(int_of_char ind.[6] - 48))
                        +(100*(int_of_char ind.[7] - 48))
                        +(10*(int_of_char ind.[8] - 48))
                        +((int_of_char ind.[9] - 48)) ) }
|
   '['['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['
```

```
        as ind
        { IND( (1000000000*(int_of_char ind.[1] - 48))
                  +(100000000*(int_of_char ind.[2] - 48))
                  +(10000000*(int_of_char ind.[3] - 48))
                  +(1000000*(int_of_char ind.[4] - 48))
                  +(100000*(int_of_char ind.[5] - 48))
                  +(10000*(int_of_char ind.[6] - 48))
                  +(1000*(int_of_char ind.[7] - 48))
                  +(100*(int_of_char ind.[8] - 48))
                  +(10*(int_of_char ind.[9] - 48))
                  +((int_of_char ind.[10] - 48)) ) }
|
   '['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['0'-'9']['
        as ind
        { IND( (10000000000*(int_of_char ind.[1] - 48))
                  +(1000000000*(int_of_char ind.[2] - 48))
                  +(100000000*(int_of_char ind.[3] - 48))
                  +(10000000*(int_of_char ind.[4] - 48))
                  +(1000000*(int_of_char ind.[5] - 48))
                  +(100000*(int_of_char ind.[6] - 48))
                  +(10000*(int_of_char ind.[7] - 48))
                  +(1000*(int_of_char ind.[8] - 48))
                  +(100*(int_of_char ind.[9] - 48))
                  +(10*(int_of_char ind.[10] - 48))
                  +((int_of_char ind.[11] - 48)) ) }
| eof { EOF }
```

Listing 13: ast.mli

```
type operator = Add | Sub | Mul | Shiftr | Shiftl

type expr =
    Lit of string
  | Bits of string (* binary string *)
  | AsnRoot of string * expr
  | AsnObj of string * string * expr
  | IdenRoot of string
  | IdenObj of string * string
  | Binop of expr * operator * expr
  | Exprseq of expr * expr
```

```
type objdecl =
        Objdeclseq of string * objdecl * objdecl
  | Objmap of string * objdecl
  | Indices of int
  |     Indseq of int * objdecl


type statement  =
    Binmap of string * objdecl
  | Expr of expr
  | Stmtseq of statement * statement
  | Print of string * string
  | Printvar of string
  | Ifthen of expr * expr * statement * statement
  | Repeat of statement * expr * expr
```

```
(*

| objdecl SEMI objdecl                          { Objdeclseq($1,$3) }



| expr SEMI expr                                 { Seq($1, $3) }
| INPUT expr SEMI INPUT expr    { Seq($2, $5) }
| expr SEMI                                      { Expr($1) }
| INPUT expr SEMI                                { Expr($2) }


  | Seq of expr * expr
  | Expr of expr
  | Binop of expr * operator * expr
```

```
| expr PLUS   expr                                { Binop($1, Add, $3) }
| expr MINUS  expr                                { Binop($1, Sub, $3) }
| expr TIMES  expr                                { Binop($1, Mul, $3) }
| expr DIVIDE expr                                { Binop($1, Div, $3) }




type expr =
    Literal of int
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
*)
```

Listing 14: parser.mly

```
%{ open Ast %}

%token PLUS MINUS TIMES SEMI ASSIGN INPUT EOF LP RP PERIOD BINMAP LCB
   RCB SHIFTR SHIFTL PRINT EQU IF THEN ELSE REPEAT UNTIL
%token <int>  LITERAL IND VARIABLE
%token <string> ID BITS DEC

%left SEMI
%right ASSIGN INPUT
%left PLUS MINUS
%left TIMES DIVIDE

%start statement
%type < Ast.statement> statement
%type < Ast.objdecl> objdecl
%type < Ast.expr> expr

%%
```

```
statement:
  BINMAP ID LCB objdecl RCB                          { Binmap ($2,$4) }
| expr                                                                { Expr
   ($1) }
| statement statement                                { Stmtseq ($1, $2) }
| PRINT ID ID SEMI                                          { Print ($2,$3)
    }
| PRINT ID SEMI                                             { Printvar ($2)
    }
| IF LP expr EQU expr RP THEN
       LP statement RP ELSE
       LP statement RP                                      { Ifthen($3,$5,
          $9,$13) }
| REPEAT LP statement RP
  UNTIL LP expr EQU expr RP                          { Repeat($3,$7,$9) }


objdecl:
  ID ASSIGN objdecl SEMI                             { Objmap($1,$3)
            }
| ID ASSIGN objdecl SEMI objdecl    { Objdeclseq($1,$3,$5) }
| IND                                                               {
   Indices($1)          }
| IND objdecl                                                { Indseq($1,$2)
           }

expr:
|  BITS                                                             { Bits(
   $1) }
| DEC                                                               { Lit(
   $1) }
| ID ASSIGN expr SEMI                                { AsnRoot($1,$3) }
| ID PERIOD ID ASSIGN expr SEMI         { AsnObj($1,$3,$5) }
| ID                                                                {
   IdenRoot($1) }
| ID PERIOD ID                                               { IdenObj($1,$3
   ) }
| expr PLUS   expr                                           { Binop($1, Add
   , $3) }
| expr MINUS  expr                                           { Binop($1, Sub
   , $3) }
| expr TIMES  expr                                           { Binop($1, Mul
   , $3) }
| expr SHIFTR  expr                                  { Binop($1, Shiftr, $3)
    }
| expr SHIFTL  expr                                          { Binop($1,
   Shiftl, $3) }
```

```
| expr expr                                                    {
   Exprseq ($1,$2) }
```

Listing 15: blooRTLs.ml

```
open Ast

(* TITLE:       THE PRECOMPILER *)
(* AUTHOR:      PETER BURROWS   *)
(* DATE:        7/2/15                  *)


(*=================  THE ENVIRONMENT  =================*)

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

let binmap = NameMap.empty;;

let objects = NameMap.empty;;



(*
        ------------------------------------------------
        NOTE: binmap takes the following module form ->
                        (string [variables] * module [objects])

        NOTE: objects take the following tuple form ->
                        (string [object], (int [value] * int array [bit
                            indices])

        ------------------------------------------------ *)

(* Creating/Updating the environment: *)

(* Initialize mapUpdate:
        - Note: this is a place holder for other global functions to
           initialize
        - mapUpdate is defined later *)
let mapUpdate = (fun var obj tuplVal map ->
                if (NameMap.mem var map) then
                        let objects = NameMap.find var map in
                        let objects = NameMap.add obj tuplVal objects
                           in
                        let map = NameMap.add var objects map in map
```

```
                else
                        let objects = NameMap.add obj tuplVal objects
                           in
                        let map = NameMap.add var objects map in map);;



(* Subfunctions to retrieve the value and bit indices of an object *)
let gettuple = (fun var obj map -> NameMap.find obj (NameMap.find var
   map) );;
let getintVal = (fun (w,x,y,z) -> w);;
let getBitIndices = (fun (w,x,y,z) -> x);;
let getBinlist = (fun (w,x,y,z) -> y);;
let getSpecialvar = (fun (w,x,y,z) -> z);;




(* ------------------------------updateBinlist
   --------------------------------*)
(* Update object 'a' given the new value of new object 'newb' and
   output the new 'a' *)
let getInd = (fun (x,y) -> x);;
let getBit = (fun (x,y) -> y);;

let rec comp a b inc =
        match a with
          [] -> []
        | hd::tl ->
                if ( (getInd (hd)) = (getInd (List.nth b inc) ) ) then
                        (List.nth b inc) :: comp tl b (inc) else (hd)::
                           comp tl b (inc);;

let rec compall a b inc =
        if inc > -1 then compall (comp a b inc) b (inc-1) else a;;

let updateBinlist a newb =
        compall a newb ( (List.length newb) - 1);;


(* ---------------------------lists2tuplelist
   --------------------------------*)
(* Merge the binary indices list and binary list to a combined tuple
   list
        i.e. [1;2;3] [4;5;6] -> [(1,4);(2,5);(3,6)] *)
        (* NOTE: l1 and l2 must be the same length *)
let rec lists2tuplelist l1 l2 =
```

```
        match l1 with
          [] -> []
        | hd::tl -> (hd,List.hd l2):: lists2tuplelist tl (List.tl l2);;


(* ------------------------------tuplelist2list
   --------------------------------*)
(* Convert the tuple lists to lists *)
let rec tuplelist2list l ind =
        match l with
          [] -> []
        | hd::tl -> if ind = 0 then (fun (x,y) -> x) hd ::
           tuplelist2list tl ind
                              else if ind = 1 then (fun (x,y) -> y)
                                 hd :: tuplelist2list tl ind
                              else [];;


(* ============================== TYPE CONVERSIONS
   ==================================*)
(* ===== binstr2binlist: Convert binary string to binary int list:

        i.e. "110010" -> [1;1;0;0;1;0]



let binstr2binlist s =
        let str2charlist s =
                let rec exp i l =
                if i < 0 then l else exp (i - 1) (s.[i] :: l) in
                exp (String.length s - 1) [] in
        List.map (fun a -> (int_of_char a) - 48) (str2charlist s);;

(* ===== binlist2int : Convert binary int list to integer:

        i.e. [1;0;0;0] -> 8


let binlist2int binlist =
  let converted_binlist = List.map float_of_int binlist in
  let helper converted_binlist =
        let rec eval i = function
           []    -> []
        | hd::tl -> if i < 0. then tl else (hd *. 2. ** i) :: eval (i
           -. 1.) tl in
        eval (float_of_int(List.length converted_binlist - 1))
           converted_binlist in
```

```
  int_of_float (List.fold_left (fun s e -> s +. e) 0. (helper
     converted_binlist) );;


(* ===== int2binlist: Convert int to a binary list

        i.e. 18 -> [1;0;0;1;0]


let int2binlist intVal =
let intVal = float_of_int intVal in
  let rec eval l intVal =
        let quot = (intVal) /. 2. in
        let flquot = floor(quot) in
        let r = 2. *. (quot -. flquot) in
          if quot < 1. then r::l else r::eval l flquot in
  let l = eval [] intVal in
let l = List.rev (l) in
List.map int_of_float l;;
(* ====================== END OF TYPE CONVERSIONS
   ==================================*)



(* =========== CONSTRUCTED SHIFT OPERATORS *)
(* ===== shiftR: i.e shiftR [1;1;1] 1 -> [0;1;1]        *)
let shiftR l byX =
let rec helper fill l byX =
        match l with
          [] -> fill
        | hd::tl ->
                if (byX > 0) then (
                        helper (0::fill) tl (byX-1)
                ) else
                        fill@(List.rev(l)) in
helper [] (List.rev(l)) byX;;

(* ===== shiftL: i.e shiftL [1;1;1] 1 -> [1;1;0]        *)
let shiftL l byX =
let rec helper fill l byX =
        match l with
          [] -> fill
        | hd::tl ->
                if (byX > 0) then (
                        helper (0::fill) tl (byX-1)
                ) else
                        (l)@fill in
```

```
helper [] (l) byX;;



(* ------------------------ mapUpdate ----------------------------*)
exception Failure of string;;


(* ===== updater :
        - the final subfunction executed by mapUpdate
        - updater updates the values of all objects within a root
           variable,
                given a chage in the root variable, or one of the
                   objects *)

        (* ===== gettuplelist - retrieves the indices and binary list
           from the environment *)
        let gettuplelist var obj map =
          lists2tuplelist
                (getBitIndices (gettuple var obj map)) (getBinlist (
                   gettuple var obj map));;

(* ===== updaterhelper: subfunction called within updater *)
let rec updaterhelper var newObj objNames map =
        match objNames with
          [] -> map
        | hd::tl -> let old = gettuplelist var hd map in
                                let old = updateBinlist old newObj in
                  let map = mapUpdate var hd
                        ( binlist2int (tuplelist2list old 1),
                          (tuplelist2list old 0),
                                (tuplelist2list old 1),
                                  (getSpecialvar(gettuple var hd map))
                                    )
                  map in updaterhelper var newObj tl map;;

(* ===== updater *)
let updater var newObj map =
        updaterhelper var newObj (List.rev(NameMap.fold (fun s c out ->
            s::out)
                                                                (NameMap.find
                                                                   var map) [])
                                                                 ) map;;
```

```
(* ===== check: called within mapUpdateHelper1 (which is declared below
   )
        - throw failures when new objects exceed previously declared
           bounds/indices *)
let check var obj tuplVal map =

  (* general check to make sure that the objects do not use duplicate
     indices *)
  let rec duplicates l =
  match l with
    [] -> []
  | hd::tl -> hd::(duplicates (List.filter (fun x -> x <> hd) tl)) in
  let out = (duplicates(getBitIndices(tuplVal))) in
        if ( (List.length out) < (List.length (getBitIndices(tuplVal)))
            ) then (
                raise (Failure ((var)^"."^(obj)^"␣uses␣duplicate␣
                   indices") )
    ) else [];

  (* general check to make sure that the objects have identical indices
      *)
  if ( (obj <> "") && (NameMap.mem var map) ) then (

                (* compare tupl lengths *)
                if ( (List.length (getBitIndices(tuplVal))) <
                        (List.length (getBinlist(tuplVal))) ) then (
                    raise (Failure ((var)^"."^(obj)^"␣overflow"))
                ) else [];

                if (NameMap.mem obj (NameMap.find var map)) then (
                        if ( (getBitIndices(gettuple var obj map)) <>
                                (getBitIndices(tuplVal)) ) then (
                               raise (Failure ((var)^"."^(obj)^"␣
                                  changed␣its␣previous␣bounds"))
                        ) else []
                ) else []
  ) else [] ;

  (* if the variable has already been declared and if it is the root *)
  if ( (NameMap.mem var map) && (obj = "") ) then (

                (* if the root has already been declared,
                        make sure the new declaration does not exceed
                           its previous value.. *)
                if ((NameMap.mem "" (NameMap.find var map)) &&
                ( (List.nth (getBitIndices(gettuple var "" map)) 0) <
                  (List.nth (getBitIndices(tuplVal)) 0)) ) then (
```

```
                    raise (Failure ((var)^" exceeds its previous
                       bounds"))

              (* ...else if the new root has less indices than its
                 previous value.. *)
              ) else if ((NameMap.mem "" (NameMap.find var map)) &&
              ( (List.nth (getBitIndices(gettuple var "" map)) 0) >
                (List.nth (getBitIndices(tuplVal)) 0)) ) then (
                    (* do nothing -> the root indices will be
                       reformatted in mapUpdate's 1st check *)
                    []

              (* ..else compare new root to existing objects (if any)
                  *)
              ) else (
              let rec outer objname =
              match objname with
                [] -> []
              | hd::tl ->
                (* compare root to all of the indices of all of the
                   objects *)
                let rec compareObjs ind =
                        if ((ind > -1) && ( (List.nth (getBitIndices(
                           tuplVal)) 0) <
                              (List.nth (getBitIndices(gettuple var
                                 hd map)) ind) ) ) then (
                                    raise (Failure ((hd)^" exceeds
                                       the bounds of "^(var)))
                        ) else if (ind > -1) then ( compareObjs (ind
                           -1)
                        ) else outer tl in
                compareObjs ((List.length (getBitIndices (gettuple
                   var hd map)))-1) in
              outer (List.rev(NameMap.fold (fun s c out -> s::out) (
                 NameMap.find var map) [])) )

(* if the variable has already been declared and the object is not
   the root,
       but the root does exist *)
) else if ( (NameMap.mem var map) && ( NameMap.mem "" (NameMap.find
   var map)) ) then (
              let rec compareObj2root newobjIndices  =
              match newobjIndices with
                [] -> []
              | hd::tl -> if ( hd > (List.nth (getBitIndices(gettuple
                   var "" map)) 0) ) then (
                                              raise(   Failure ((obj)
```

```
                                                    ^" exceeds the
                                                    bounds of "^(var)))
                                          ) else compareObj2root tl in
                compareObj2root (getBitIndices(tuplVal))

  (* if the variable has not been declared *)
  ) else [];;



(* ===== mapUpdateHelper 1: called within mapUpdateHelper0, which is
   seen below *)
let rec mapUpdateHelper1 = (fun var obj tuplVal map ->
  (* Check the bounds of the new object *)
  let _ = check var obj tuplVal map in

  if (NameMap.mem var map) then (
        let objects = NameMap.find var map in
        (* If objects exists *)
        if (NameMap.mem obj objects) then (

          (* diff MISMATCH procedure left pads zeros to maintain the
             binary array, as needed *)
          let diff = ((List.length (getBinlist(NameMap.find obj objects
             )) ) -
                    (List.length (getBinlist(tuplVal)))) in
            if (diff < 0) then
                      raise (Failure ((var)^"."^(obj)^" length
                         mismatch/overflow") )
            else if (diff > 0) then ( (* concat based on diff and add
               indices *)
                  let rec append l n =
                        if n = 0 then l else 0::append l (n-1) in
                        let appended = append (getBinlist(tuplVal))
                           diff in
                        let objects = NameMap.add obj
                                ( binlist2int(appended),
                                        (getBitIndices(NameMap.find obj
                                            objects)),
                                         (appended),
                                                (getSpecialvar(NameMap.
                                                   find obj objects)) )
                                                    objects in
                        let map = NameMap.add var objects map in map)
                    (* populate the map with the modified object *)
              else
                        let objects = NameMap.add obj tuplVal objects
```

```
                            in
                        let map = NameMap.add var objects map in map)

        (* else load a new variable and object/root *)
        else (
                let objects = NameMap.add obj tuplVal objects in
                let map = NameMap.add var objects map in map))

  (* else load a new variable and object/root *)
  else(
        let objects = NameMap.add obj tuplVal objects in
        let map = NameMap.add var objects map in map));;


(* ===== mapUpdateHelper0: first subfunction called by mapUpdate *)
let mapUpdateHelper0 var obj tuplVal map =
  (* First Check:
          - reformat the root variable if its new indices are smaller
              than its previous indices *)
  if ( (obj ="") && (NameMap.mem var map) && (NameMap.mem "" (NameMap.
     find var map)) &&
         ( (List.nth (getBitIndices(gettuple var "" map)) 0) >=
           (List.nth (getBitIndices(tuplVal)) 0)) ) then (

                (* compare new root indices with the old *)
                let rec compareRootInds newRoot ind =
                match newRoot with
                  [] -> []
                | hd::tl -> if (hd = (List.nth
                                        (List.rev(getBitIndices
                                           (gettuple var "" map
                                           ))) ind)) then (
                                        compareRootInds tl (ind
                                           +1)
                                ) else (
                                        raise (Failure ((var)^"
                                           's indices have
                                           changed")) ) in
                compareRootInds (List.rev (getBitIndices(tuplVal))) 0;

                        (* then update the map using prior root indices
                            *)
                        (mapUpdateHelper1 var obj
                                ( getintVal(tuplVal),
                                        getBitIndices(gettuple var ""
                                           map),
                                        getBinlist(tuplVal),
```

```
                                                        getSpecialvar(
                                                           tuplVal) )
                                                           map)
        ) else mapUpdateHelper1 var obj tuplVal map;;



(* ===== mapUpdate *)
let mapUpdate var obj tuplVal map =
        let map = mapUpdateHelper0 var obj tuplVal map in
        let map = updater var (gettuplelist var obj map) map in
        map;;
(* --------------------END OF mapUpdate ----------------------------*)




(* ================== Additional global functions
   ====================== *)

(* ===== mkIndices -> makes an int list given a length
        i.e. mkIndices [4;2;3] (List.length [4;2;3])
                        -> [2;1;0]] *)
let rec mkIndices l len =
                if (len > -1) then (
                        len::mkIndices l (len-1)
                ) else l;;

(* print_list -> used for debugging *)
let rec print_list = function
                          [] -> print_endline "]"
                        | e::l -> print_int e ;
                                        if (List.length l = 0) then
                                           print_string ""
                                        else print_string ";" ;
                                           print_list l;;

(* print_string_list -> used for debugging *)
let rec print_string_list = function
                          [] -> print_endline "]"
                        | e::l -> print_string e ;
                                        if (List.length l = 0) then
                                           print_string ""
                                        else print_string ";" ;
                                           print_string_list l;;

(* checktotaloperations ->
```

```
        Check length of the IR vhdl string list. If too many assignment
            operations
        (i.e. var := a+b+c), then raise an error *)
let checktotaloperations = (fun appendvhdl ->
  let checktotaloperations =
        if ( (List.length (appendvhdl)) > 8 ) then (
          let rev = List.rev (appendvhdl) in
          let var = List.hd (rev) in
          let obj = List.hd (List.tl rev) in
        raise (Failure ("Too many arithmetic operators assigned to "^(
           var)^"."^(obj)) )
                ) else [] in checktotaloperations);;


(* print -> i.e. print var obj -> prints to screen *)
        let print var obj binmap = [];
                print_endline "\n";
                print_string var; print_string "."; print_string obj;
                   print_endline " =";
        print_string "\tInteger Representation:\t\t";
                        print_endline (string_of_int (getintVal(
                           gettuple var obj binmap)));
                print_string "\tBinary Indices:\t\t\t";
                        print_string "[";
                        print_list (getBitIndices(gettuple var obj
                           binmap));
                print_string "\tBinary Representation:\t\t";
                        print_string "[";
                        print_list (getBinlist(gettuple var obj binmap)
                           );
                (*print_string "\tType (In/Out/Signal):\t\t";
                        print_endline (string_of_int (getSpecialvar(
                           gettuple var obj binmap)));*)
                print_endline "\n";;

(* ================== END OF Additional global functions
   ======================*)



(* INITIALIZE THE VHDL IR FOR EVALUATION *)
print_string "\n========================= blooRTLs
   ========================>";;
let vhdl = NameMap.empty;;
let vhdlcount = "0";;
let vhdl = NameMap.add vhdlcount ("" ::[]) vhdl;;
```

```
(* vhdl incrementer *)
let increment =(function count -> string_of_int((int_of_string count) +
    1));;

(*
   ------------------------------------------------------------------------
   *)
(* ---------------------------- EVALUATION
   ---------------------------- *)
(*
   ------------------------------------------------------------------------
   *)
(* ==== evaluates BINMAP statements *)
let rec evalBinmap var obj indices binmap = (function
        Objdeclseq(obj,decl1,decl2) -> ( (*print_string "\nObjdeclseq(
           obj,decl1,decl2) ->";*)
          let var, obj, indices, binmap = evalBinmap var obj indices
             binmap decl1 in
          let var, obj, indices, binmap = evalBinmap var "" [] binmap
             decl2 in
          var, obj, indices, binmap )

  | Objmap(obj,decl) -> ( (*print_string "\nObjmap(obj,decl) -> ";*)
          let var, obj, indices, binmap = evalBinmap var obj indices
             binmap decl in
          var, obj, indices, binmap )

  |     Indseq(ind,decl) -> ( (*print_string "\n\tIndexSeq(i,decl) ->
     ";*)
          let var, obj, indices, binmap = evalBinmap var obj (ind::
             indices) binmap decl in
          var, obj, indices, binmap )

  |     Indices(ind) -> ( (* print_string "Index(i) -> "; *)
        let ind = ind::indices in
          let dummybits = List.map (fun x -> x-x) ind in
          let tupl = (0,List.rev(ind),dummybits,0) in

          (* if the variable already exits... *)
          if ((NameMap.mem var binmap) &&
                        (NameMap.mem "" ((NameMap.find var binmap))))
                           then (
                (* retrieve all values from the root variable... *)
                let originaltupl = gettuple var "" binmap in
                  (* then, update the new object indices to the map...
                     *)
                  let binmap = mapUpdate var obj tupl binmap in
```

```
                        (* finally, restore original values of the root
                            to the map *)
                        let binmap = mapUpdate var "" originaltupl
                           binmap in
        (* return: *)
         var, obj, indices, binmap

          (* otherwise, initialize the object loading the dummy bit
             values *)
          ) else (
                let binmap = mapUpdate var obj tupl binmap in

        (* return: *)
          var, obj, indices, binmap ) )

)

(* ==== evaluates Arithmetic Expressions/Assignments *)
let rec eval var obj binmap vhdl vhdlcount = (function
(* all types return (tuplVal, map) *)

        Bits(b) -> ( (* print_string "LoadBits(b) -> ";*)
          let binlist = binstr2binlist b in
                let x = binlist2int binlist in
                  let bitIndices = mkIndices [] ((String.length b)-1)
                     in

                        (* APPEND VHDL *)
                let appendvhdl = NameMap.find vhdlcount vhdl in
                let appendvhdl = b::"BITS"::appendvhdl in
                let vhdl = NameMap.add vhdlcount appendvhdl vhdl in

                var, obj, vhdl,vhdlcount,(x,bitIndices,binlist,-1),
                   binmap )

  | Lit(x) -> (         (* print_string "LoadInt(i) -> ";*)

        (* convert ['0'- '9']+'d' to integer *)
        let intx =
                let removeD s =
                        let rec helper s i out =
                                if ( i < 0 ) then (
                                  out
                                ) else  ( helper s (i-1) (out + ( (
                                   int_of_char s.[i]) - 48)*
                                  int_of_float(10.**(float_of_int(((
                                     String.length s) - 2) - i)))) ) in
```

```
                    helper s (((String.length s) - 2)) 0 in
            removeD x in


            (* APPEND VHDL *)
            let appendvhdl = NameMap.find vhdlcount vhdl in
            let appendvhdl = (string_of_int intx)::"INT"::
               appendvhdl in
            let vhdl = NameMap.add vhdlcount appendvhdl vhdl in

        (* return the tupl and binmap *)
            var, obj, vhdl,vhdlcount,(intx,[],(int2binlist intx)
               ,-1), binmap)

| IdenRoot(var) -> (  (*print_string "IdenRoot(var) -> "; *)

    if ( (NameMap.mem var binmap) && (NameMap.mem "" (NameMap.find
       var binmap)) ) then (
            let newintVal = getintVal (gettuple var "" binmap) in
            let newBitIndices = getBitIndices (gettuple var ""
               binmap) in
            let newBinlist = getBinlist (gettuple var "" binmap) in
            let newSpecialVar = getSpecialvar (gettuple var ""
               binmap) in

                    (* APPEND VHDL *)
            let appendvhdl = NameMap.find vhdlcount vhdl in
            let appendvhdl = ""::var::appendvhdl in
            let vhdl = NameMap.add vhdlcount appendvhdl vhdl in

    var, obj, vhdl,vhdlcount, (newintVal,newBitIndices,newBinlist,
       newSpecialVar), binmap
    (* if iden not found, then raise an error! *)
    ) else (raise (Failure ((var)^" has no value"))) )

| IdenObj(var, obj) -> (      (*print_string "IdenObj(var,obj) -> ";
   *)

    if ( (NameMap.mem var binmap) && (NameMap.mem obj (NameMap.find
        var binmap)) ) then (
            let newintVal = getintVal (gettuple var obj binmap) in
            let newBitIndices = getBitIndices (gettuple var obj
               binmap) in
            let newBinlist = getBinlist (gettuple var obj binmap)
               in
            let newSpecialVar = getSpecialvar (gettuple var obj
               binmap) in
```

```
                    (* APPEND VHDL *)
                let appendvhdl = NameMap.find vhdlcount vhdl in
                let appendvhdl = obj::var::appendvhdl in
                let vhdl = NameMap.add vhdlcount appendvhdl vhdl in

        var, obj, vhdl,vhdlcount,(newintVal,newBitIndices,newBinlist,
           newSpecialVar), binmap

        (* if iden not found, then raise an error! *)
        ) else (raise (Failure ((var)^"."^(obj)^" has no value")))  )


| AsnRoot(var,e) -> (
                (* Retrive the root variable *)
                let asnvar = var in
                let asnobj = "" in


                (* APPEND PREVIOUS VHDL LINE *)
                let appendvhdl = NameMap.find vhdlcount vhdl in
                let appendvhdl = ";"::appendvhdl in
                (* Check length of of the string list! If too many
                   operations
                        (i.e. var := a+b+c),then raise an error!!
                      let checklen = checktotaloperations appendvhdl
                         in *)

        let vhdl = NameMap.add vhdlcount appendvhdl vhdl in
                (* Used for debugging
                print_string "\n\t+ IR LINE "; print_string vhdlcount;
                   print_string ": [";
                print_string_list (List.rev (NameMap.find vhdlcount
                   vhdl));

        print_string "\nAsnRoot (var,exp) -> ";*)

        (* BEGIN APPENDING A NEW VHDL LINE *)
        (* first increment the vhdl line counter *)
        let vhdlcount = increment vhdlcount in
        let appendvhdl = "<="::""::var::[] in
                let vhdl = NameMap.add vhdlcount appendvhdl vhdl in


        let var, obj, vhdl,vhdlcount, tupl_rx,binmap = eval asnvar
           asnobj binmap vhdl vhdlcount e in
```

```
      let newbitIndices =
              (* get the bit indices from the map if they exist *)
              if ( (NameMap.mem var binmap) &&
                       (NameMap.mem "" (NameMap.find var binmap)) )
                          then (
              getBitIndices(gettuple var "" binmap)
       (* if no indices list exists, make a list *)
      ) else ( mkIndices [] (List.length (getBinlist(tupl_rx))-1) )
         in

      (* making the new tupl (slightly different than in the AsnObj
         procedure) *)
      let newtupl =
        ((getintVal(tupl_rx)),(newbitIndices),getBinlist(tupl_rx),0)
          in

      (* NEED TO INDICATE WHETHER OR NOT THE ROOT IS A SPECIAL VAR?
         *)
      (* update the binmap with the new variable and tuple *)
      let binmap = mapUpdate var "" newtupl binmap in

      (* return*)
      var, obj, vhdl, vhdlcount, newtupl, binmap)

| AsnObj(var,obj,e) -> (

              (* APPEND PREVIOUS VHDL LINE *)
              let appendvhdl = NameMap.find vhdlcount vhdl in
              let appendvhdl = ";"::appendvhdl in
              (* Check length of of the string list! If too many
                 operations
                       (i.e. var := a+b+c),then raise an error!!
                     let checklen = checktotaloperations appendvhdl
                        in *)

              let vhdl = NameMap.add vhdlcount appendvhdl vhdl in
              (* Used for debugging *)
              (*
              print_string "\n\t+ IR LINE "; print_string vhdlcount;
                 print_string ": [";
              print_string_list (List.rev (NameMap.find vhdlcount
                 vhdl));*)

              (* print_string "\nAsnObj(var,obj,exp) -> ";*)
      (* BEGIN APPENDING A NEW VHDL LINE *)
      (* first increment the vhdl line counter *)
```

```
        let vhdlcount = increment vhdlcount in
        let appendvhdl = "<=" :: obj :: var :: [] in
                let vhdl = NameMap.add vhdlcount appendvhdl vhdl in

        let var, obj, vhdl,vhdlcount, tupl_rx,binmap = eval var obj
           binmap vhdl vhdlcount e in

        let newbitIndices =
                (* get the bit indices from the map *)
                if ( (NameMap.mem var binmap) &&
                        (NameMap.mem obj (NameMap.find var binmap)) )
                           then (
                getBitIndices(gettuple var obj binmap)
         (* if no indices list exists, throw failure *)
        ) else ( raise (Failure ((var)^"."^(obj)^" has not been
           declared")) )  in

        (* NOTE: NEED TO CONVERT ID INT TO BINLIST BEFORE UPDATING!
                OTHERWISE OVERFLOW!! *)
                let newintVal = getintVal (tupl_rx) in
                let newBinlist = int2binlist newintVal in
                let newtupl = (newintVal,(newbitIndices),newBinlist,0)
                   in

        (* update the binmap with the new variable and tuple *)
        let binmap = mapUpdate var obj newtupl binmap in


        (* return *)
        var, obj, vhdl, vhdlcount, newtupl, binmap)


  | Binop(e1, op, e2) -> ( (* print_endline "\nBinop(e1,op,e2) -> "; *)

        (* Retrieve the length of the variable and object being
           assigned *)
        let asnvar = var in
        let asnobj = obj in
(*
        print_endline "VAR";
        print_endline var;
        print_endline obj; *)


        let var1, obj1, vhdl, vhdlcount, rxtupl1, binmap = eval var obj
            binmap vhdl vhdlcount e1 in
```

```
                (* Need to APPEND op to VHDL *)

        let var2, obj2, vhdl, vhdlcount, rxtupl2, binmap =
                eval var obj binmap vhdl vhdlcount e2 in

        let newintVal =
          match op with
          (* check lengths before executing the operation *)
            Add -> (
                        getintVal(rxtupl1) + getintVal(rxtupl2) )
          | Mul -> (
                        getintVal(rxtupl1) * getintVal(rxtupl2) )
          | Sub -> (
                        let checksub =
                          if (getintVal(rxtupl1) < getintVal(rxtupl2) )
                              then (
                                  raise (Failure ((var)^"."^(obj)^"
                                     subtraction underflow"))
                          ) else ( getintVal(rxtupl1) - getintVal(
                             rxtupl2) ) in checksub
                )
          | Shiftr -> ( let shiftedVal = shiftR (getBinlist(rxtupl1)) (
             getintVal(rxtupl2)) in
                                ( binlist2int shiftedVal) )
          | Shiftl -> ( let shiftedVal = shiftL (getBinlist(rxtupl1)) (
             getintVal(rxtupl2)) in
                                ( binlist2int shiftedVal)
                )   in




        let newtupl = (newintVal,[],int2binlist(newintVal),-1) in
          var, obj, vhdl, vhdlcount, newtupl, binmap)


  | Exprseq(e1,e2) ->( (* print_string "\nExprseq(exp1,exp2) -> "; *)
          let var, obj, vhdl,vhdlcount, tupl_rx, binmap = eval var obj
             binmap vhdl vhdlcount e1 in
          let var, obj, vhdl,vhdlcount, tupl_rx, binmap = eval var obj
             binmap vhdl vhdlcount e2 in
                var, obj, vhdl,vhdlcount, tupl_rx, binmap )
)
```

```
(* ==== Reads in the statements from the input text and handles them
   accordingly *)
let rec evalstatements var obj tuplrx binmap vhdl vhdlcount = (function
        Binmap(var,objdecl) -> ( (*print_string "\n\n\n------ BINMAP(
           var,objdecl) ------> ";*)
          let var,obj,indices,binmap = evalBinmap var "" [] binmap
             objdecl in
          var, obj, tuplrx, vhdl, binmap )

  |     Expr(e) -> (
        (* print_string "\n\n\n----------- EXPRS(exp) -----------> ";*)
                let var, obj, vhdl,vhdlcount, tuplrx ,binmap = eval var
                    obj binmap vhdl vhdlcount e in

                (* APPEND PREVIOUS VHDL LINE *)
                let appendvhdl = NameMap.find vhdlcount vhdl in
                let appendvhdl = ";"::appendvhdl in

                (* Check length of of the string list! If too many
                   operations
                        (i.e. var := a+b+c),then raise an error!!
                       let checklen = checktotaloperations appendvhdl
                          in *)
                let vhdl = NameMap.add vhdlcount appendvhdl vhdl in

                (* Used for debugging *)
                (* print_string "\n\t+ IR LINE "; print_string
                   vhdlcount; print_string ": [";
                print_string_list (List.rev (NameMap.find vhdlcount
                   vhdl));
                print_endline "\n"; *)

                (* Update the binmap with the received tuple *)
                        let rev = List.rev (appendvhdl) in
                        let var = List.hd (rev) in
                        let obj = List.hd (List.tl rev) in
                let binmap = mapUpdate var obj tuplrx binmap in

                        var, obj, tuplrx, vhdl, binmap )

  | Stmtseq(st1,st2) ->( (* print_string "\n\nSTMTSEQ(stmt1,stmt2) ->
    "; *)

                let var, obj, tuplrx, vhdl, binmap =
                        evalstatements var obj tuplrx binmap vhdl
                           vhdlcount st1 in
```

```
            let var, obj, tuplrx, vhdl, binmap =
                    evalstatements var obj tuplrx binmap vhdl
                       vhdlcount st2 in
            var, obj, tuplrx, vhdl, binmap )

| Print(var,obj) -> (print var obj binmap;
            var, obj, tuplrx, vhdl, binmap)
| Printvar(var) -> (print var "" binmap;
            var, obj, tuplrx, vhdl, binmap)

| Ifthen(condition1,condition2,ifstmt,elsestmt) -> (
            (* APPEND DUMMY VHDL? *)

      let var, obj, vhdl,vhdlcount, tuplrx1 ,binmap =
            eval var obj binmap vhdl vhdlcount condition1 in
      let var, obj, vhdl,vhdlcount, tuplrx2 ,binmap =
            eval var obj binmap vhdl vhdlcount condition2 in

      let var, obj, tuplrx, vhdl,binmap  =
            if (getintVal(tuplrx1) = getintVal(tuplrx2)) then (
                  evalstatements var obj tuplrx binmap vhdl
                     vhdlcount ifstmt
            ) else (evalstatements var obj tuplrx binmap vhdl
               vhdlcount elsestmt) in

            var, obj, tuplrx, vhdl, binmap)

| Repeat(repeatstmt,condition1,condition2) -> (

      let var, obj, vhdl,vhdlcount, tuplrx1 ,binmap =
            eval var obj binmap vhdl vhdlcount condition1 in
      let var, obj, vhdl,vhdlcount, tuplrx2 ,binmap =
            eval var obj binmap vhdl vhdlcount condition2 in

      let cond1 = getintVal(tuplrx1) in
      let cond2 = getintVal(tuplrx2) in



      let binmap =
        let rec loop binmap var obj vhdl vhdlcount repeatstmt cond1
           cond2 out =


                  if (cond1 = cond2) then (
                        binmap
                  ) else (
```

```
                                let var, obj, tuplrx, vhdl, binmap =
                                        evalstatements var obj tuplrx
                                           binmap vhdl vhdlcount
                                           repeatstmt in

                                let _, _, _, _, tuplrx2, _ =
                                        eval var obj binmap vhdl
                                           vhdlcount condition1 in

                                let _, _, _, _, tuplrx1, _ =
                                        eval var obj binmap vhdl
                                           vhdlcount condition2 in

                                let cond1 = getintVal(tuplrx1) in
                                let cond2 = getintVal(tuplrx2) in

                                (* let intUpdate = getintVal (gettuple
                                   var obj binmap)
                                let intUpdate = getintVal (tuplrx) in

                                print_string "\nVAL var:\t";
                                print_int (getintVal ( gettuple "var"
                                   "" binmap ));

                                print_string "\nVAL var2:\t";
                                print_int (intUpdate);

                                (* check binary map *)
                                let binmap = mapUpdate var obj tuplrx
                                   binmap in*)

                        loop binmap var obj vhdl (increment vhdlcount)
                           repeatstmt cond1 cond2 out ) in
                loop binmap var obj vhdl vhdlcount repeatstmt cond1
                   cond2 [] in

        var, obj, tuplrx, vhdl, binmap

        )

)



let _ =
```

```
let lexbuf = Lexing.from_channel stdin in
let statement = Parser.statement Scanner.token lexbuf in

let var,obj,tuplrx, vhdl, binmap =
      evalstatements "" "" (0,[],[],0) binmap vhdl vhdlcount
         statement in
      let var = "var1" in
      let obj = "" in []
```