a binary manipulation language

*bloobRTLs*

Apurv Gaurav (ag3596)
Peter H Burrows (phb2114)
Pinhong He (ph2482)
Zhibo Wan (zw2327)
Motivation, Overview, and Tutorials
Introduction of the blooRTLs
  • Language Features
Project Architecture and keywords
  • Scanner, Parser, AST, VHDL
Test Suites
Summary and Lessons Learned
Demo!
Overview: Behavioral Language for Object-Oriented RTL Specs

- An RTL description language geared towards catalyzing the development, simulation, and synthesis of RTL specs
- “Object-Oriented” - but NOT in the traditional sense
- “Reasonably” fast clock frequency assumed (>MHz)
- Compiles down to Sequential VHDL
blooRTLs Tutorial

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

var := 10000001;

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

PRINT var middle2bits;
PRINT var;

"Objects" of variable "var"

<table>
<thead>
<tr>
<th>ms</th>
<th>middle2</th>
<th>lsb</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>bits</td>
<td></td>
</tr>
<tr>
<td>1000000001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10001001

peter@MacDonald:/Desktop/blooRTLs_Precompiler$ ./blooRTLs<in.txt

------------------------- blooRTLs -------------------------

var.middle2bits =
    Integer Representation: 1
    Binary Indices: [4;3]
    Binary Representation: [0;1]

var =
    Integer Representation: 137
    Binary Indices: [7;6;5;4;3;2;1;0]
    Binary Representation: [1;0;0;0;1;0;0;1]
Compiler Considerations for VHDL: The VHDL Libraries

- **Early 1990s** → **Synopsys** developed the arithmetic library with a user-friendly VHDL arithmetic syntax and packaged it into the IEEE library

- **Late 1990s** → **IEEE** developed and standardized the numeric library due to unexpected behavior across various toolkits that used the arithmetic library

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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```
Compiler Considerations for VHDL: The VHDL Libraries

**NOT STANDARDIZED**

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

- **Tradeoff:** The NUMERIC library is **MORE RELIABLE** for simulation and synthesis, however it is much **QUIRKIER**!
  - It does **NOT** raise an error for overflow/underflow
  - It does **NOT** permit arithmetic for vectors of varying lengths;
    however, there is a clever work-around

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

declaration
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
begin
  process (Resetn, Clock)
  begin
    if Resetn = '0' then
      state <= 'A';
    elsif (Clock EVENT AND Clock = '1') then
      case state is
        when 'A' =>
          if ( ... ) then
            -- some statements/code...
            state <= 'A';
          else
            state <= 'B';
          end if;
        when 'B' =>
          -- some statements/code..
          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;
The Bottom Line: the blooRTLs compiler performs VHDL “length inferencing” for you.

```vhdl
architecture rtl of example is
  -- signals declared here
  signal var : std_logic_vector (7 downto 0);
  signal nibble0_tmp : std_logic_vector (7 downto 0);
  signal one3 : std_logic_vector (3 downto 0) := "0001";

begin
  process (Resetn, Clock)
  begin
    if Resetn = '0' then
      state <= 'A';
    elsif (Clock EVENT AND Clock = '1') then
      case state is
      when 'A' =>
        nibble0_tmp <= std_logic_vector(
          one3_tmp * (var(7) & var(6) & var(5) & var(4)) -- nibble
          * ("0" & "0" & "0" & var(0)) ); -- lsb
        state <= 'B';
      when 'B' =>
        var(3) <= nibble0_tmp(3);
        var(2) <= nibble0_tmp(2);
        var(1) <= nibble0_tmp(1);
        var(0) <= nibble0_tmp(0);
        state <= 'A';
      when 'C' =>
        -- some statements/code..
      end case;
    end if;
  end process;

  -- push the desired logic to the output port.
  output0 <= var;
end rtl;
```
- Before compiling, the blooRTLs source code **MUST** be precompiled in order to:

  - Cache the bit vector indices given by the BINMAP
  - Check for arithmetic over/underflows

- In *Ocaml*, a map module was implemented to cache/log the values and indices of variables and objects…
**Precompiler: Ocaml environment**

**It’s a Map of Maps!**

<table>
<thead>
<tr>
<th>Keys (Variables)</th>
<th>Values (Maps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>“var” →</td>
<td>Keys (Objects) → Values (Tuples of int*int list * int list))</td>
</tr>
<tr>
<td></td>
<td>“” → (137, [7;6;5;4;3;2;1;0], [1;0;0;0;1;0;0;1],0)</td>
</tr>
<tr>
<td></td>
<td>“msb” → (1, [7], [1], 0)</td>
</tr>
<tr>
<td></td>
<td>“lsb” → (1, [0], [1], 0)</td>
</tr>
<tr>
<td></td>
<td>“middle2bits” → (1, [4;3], [0;1], 0)</td>
</tr>
</tbody>
</table>

```ocaml
# 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)
```

10
blooRTLs source code

Scanner

Parser/AST

Pre compiler

VHDL

Compiler

*Project Architecture*
* Variable Declaration var1
* Assign value for variables :=
* Basic operations: + - * =
* Binary shifting << >>
* BINMAP
* IF-THEN-ELSE, REPEAT-UNTIL
* PRINT

* Program Keywords
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'-'9']+'l' 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))
| +((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))
| +((int_of_char ind.[2] - 48)) \}
| +((int_of_char ind.[3] - 48)) \}
| +((int_of_char ind.[4] - 48)) \}
| eof \{ EOF \}
statement:
  BINMAP ID LCB objdecl RCB
  | expr
  | statement statement
  | PRINT ID ID SEMI
  | PRINT ID SEMI
  | IF LP expr EQU expr RP THEN
  |   LP statement RP ELSE
  |   LP statement RP
  | REPEAT LP statement RP
  | UNTIL LP expr EQU expr RP
    { Binmap ($2,$4) }
    { Expr ($1) }
    { Stmtseq ($1, $2) }
    { Print ($2,$3) }
    { Printvar ($2) }
    { Ifthen($3,$5,$9,$13) }
    { Repeat($3,$7,$9) }

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

expr:
  | BITS
  | DEC
  | ID ASSIGN expr SEMI
  | ID PERIOD ID ASSIGN expr SEMI
  | ID
  | ID PERIOD ID
  | expr PLUS expr
  | expr MINUS expr
  | expr TIMES expr
  | expr SHIFTR expr
  | expr SHIFTL expr
  | expr expr
    { Bits($1) }
    { Lit($1) }
    { AsnRoot($1,$3) }
    { AsnObj($1,$3,$5) }
    { IdenRoot($1) }
    { IdenObj($1,$3) }
    { Binop($1, Add, $3) }
    { Binop($1, Sub, $3) }
    { Binop($1, Mul, $3) }
    { Binop($1, Shiftr, $3) }
    { Binop($1, Shiftl, $3) }
    { Exprseq ($1,$2) }
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
* Using blooRTLs bit-mapping feature on sequential data, we can encode important sequential data, such as DNA, and be able to track genes
* In addition, DNA encoding can be optimized to use less space and digits
* Original Encoding:
  * A = 00, C = 01, T = 10, G = 11
* Huffman Encoding:
  * A = 0, T = 10, C = 101, G = 110
We will take a DNA sequence that has been encoded into binary numbers and count how many of each nucleotide there are.

Features Tested:
- BINMAP, If-Then-Else, Repeat-Until, PRINT, bit manipulation, (Switch)

Output:
14
6
8
7
Based on the nucleotide frequencies, we will re-encode the DNA code using the more efficient Huffman Algorithm.

Allots less bits to high freq info, more bits for low freq info.

Features Tested:
* BINMAP, If-Then-Else, Repeat-Until, PRINT, bit manipulation, (Switch)
* Although summer term is SHORT
* Better sense of how does Ocaml work and creating a compiler
* We learned to appreciate the complexity behind routine operations like loops and if-then statements that we take for granted in existing languages
* The levels of abstraction that exist between the programming language and machine code
* Computer Science is more than just coding

* Summary and Lessons Learned
Thank you
* BINMAP var1 {
  *   nibble:=[3][2][1][0];
  *
}  
* var1 := 10001110;
* var1.nibble := 0000;
* PRINT var1;