Proposal: `blooRTLs` - The Behavioral Language for Object-Oriented Register Transfer Level Specs

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

June 3, 2015

1 Motivation

While computer engineers may freely choose their preferred methodologies and tools to help catalyze the Register Transfer Level (RTL) design process, the initial construction of the behavioral spec is inevitable. Behavioral specs have been written in various forms, which have consisted of prose, pseudo code, and block diagrams. Often, the eccentric form of these specs results in confusion among a team of engineers with diverse skill sets and varying academic/industrial backgrounds. Ultimately, it is the goal of blooRTLs to both universalize and facilitate the development of an RTL behavioral spec.

2 Language Usage

At its core, blooRTLs is a bit manipulation language. The language is aimed to provide users with the means to understand, design, validate, debug, and scale an RTL behavioral spec, although its usage is not limited to this domain (as will become evident later in this section).

2.1 Example "One's Counter" RTL Spec Design using blooRTLs

In order to afford readers a better understanding of blooRTLs language, its usage is best seen through a simple example:

- *Verbal Description of a "One's Counter" RTL Spec*: Given a 32-bit binary number placed on "input" bus, count the number of 1 bits in this number, and place result on the "Output" bus

- *blooRTLs program to represent a "One's Counter" Algorithm*:

Listing 1: Simple one's counter algorithm implemented in blooRTLs

```
// Declare variables
ocount := 0;       //ocount (i.e. one's count) loaded with 0
data := load(9);   //data is loaded with #9d, (Data:= 1001;)

// Map the least significant bit in Data
binmap data {
  LSB = 0; //declares the LSB to be at bit position 0 of data
} //data.LSB can be called

// Main loop
repeat
  if (data.LSB = 1){
    ocount := ocount + 1; //increment 1's count
  }
    data := data >> 1;     //shift right by 1
until (Data = 0);          //Loop body allows early exit

// Print Ocount to screen
print "bin:ocount=" ocount "\n";
print "int:ocount=" to_int(ocount);
```

- The output screen would print the format (bin, int), the variable (ocount), and the value held by the variable (#10b, #2d):

```
>>> bin : ocount = 10
>>> int : ocount = 2
```

- The remaining steps of the RTL design flow are appended for curious readers (They were adopted from Steven Nowick's notes in his Advanced Logic Design course and include contributions from Daniel Gajski). From Nowick's notes, one can observe the advantages blooRTLs to the RTL design flow.

- From the previous example, it also becomes evident that blooRTLs only permits variables to be stored as the binary type. However, it is worth mentioning that these variables can be loaded (and displayed) as floating points (i.e. `x := load(1.9)`), integers (i.e. `x :=load(8)`), and ASCII strings (i.e. `x := load(``hello")`). The input type to load need not be specified; blooRTLs adjusts memory sizes of the variables accordingly.

- *A Note on Hardware Descriptive Languages (HDLs):* In opposition to blooRTLs, HDLs are syntactically rigid as they preallocate datapath resources based on the user's code. In the RTL design space, the purpose of blooRTLs is to verify a behavioral spec before the combinational components and registers are selected. In turn, blooRTLs grants designers the flexibility to allocate data path resources accordingly.

2.2 Other Uses

While the notion of facilitating a language that verifies behavioral RTL specs helped conceive the idea of blooRTLs, the `binmap` "class" inherently permits the language to extend into other domains.

- One such domain is encryption. Take, for example, a user who wishes to encrypt data at the bit level using his own encryption method and send it. In particular, assume the user wants to use alternating bits to send the first character and the remaining alternating bits to send the second character. His or her program might look something like this:

Listing 2: Arbitrary Bit Level Encryption Example

```
//Attach a sockets library
#include blooRTLs_sockets.h

senddata := 0;  //senddata is loaded with #0b

// Encryption map: declares the bit positions that will hold
// the 1st and 2nd char values
binmap senddata {
  first_char = 14:12:10:8:6:4:2:0;
  second_char = 15:13:11:9:7:5:3:1;
} // Note that ':' concatanates the bit positions

  //load 1st char bit positions with "a":
senddata.first_char := load("a");
  //load 2nd char bit positions with "b":
senddata.second_char:= load("b");

//senddata holds binary value of "a" and "b" intertwined:
//  "a" = 01100001 in ASCII
//  "b" = 01100010 in ASCII
//  => senddata = 0011110000000010

// assuming a socket was set up, send the data:
socketsend(senddata); //senddata sent to another user
```

- Another useful call is the `function` method, which permits scalability for all binary operations. An example containing two functions is seen below:

Listing 3: Arbitrary Bit Level Encryption Example

```
// Declare variables
data0 := load(40);  //data0 is loaded with #40d
data1 := 10;  //data1 is loaded with #10b, or #2d
datasum := 0;

// Main
datasum := add(data0,data1);
print_datasum();

// Functions
function sum = add(a,b){
  sum := a + b;
}  // the funtion returns the sum given args a,b

function [] = print_datasum(){
        // datasum Ocount to screen
  print "bin:datasum=" datasum "\n";
  print "int:datasum=" to_int(datasum);
  print "str:datasum=" to_ascii(datasum);
}  // the funtion prints data sum as described
```

```
>>> bin:datasum = 00101010
>>> int:datasum = 42
>>> str:datasum = *
```

Appendix - Nowick's RTL design flow

A general RTL design flow has been provided by Steven Nowick. Using excerpts from Daniel Gajski, Nowick's RTL design process fully guides an engineer in the pursuit of transforming an algorithm into an RTL architecture comprised of data path blocks synced to a control unit. The design flow is summarized in the following steps:

0. Verbal Description of the Algorithm (behavioral)
1. Psuedo-Code of the Algorithm (behavioral) [Note: this step was replaced by blooRTLs]
2. Write RTL Specification: Generalized Algorithmic State Machine (ASM)
3. Allocate (Select) Datapath Blocks (and set harwired inputs + other optimizations)
4. Identify Status Signals
5. Draw Final Microarchitecture (RTL)
6. Derive Controller Specification = Control ASM Specification
7. Generate Symbolic State Diagram = Control FSM Specification
8. Synthesize Controller (FSM)

Write RTL Specification: Generalized Algorithmic State Machine (ASM)

After the blooRTLs "one's counter" program runs as desired, the next step of Nowick's RTL design flow calls for the blooRTLs syntax to be super imposed onto a Gajskiian generalized ASM. Extending the "one's counter" example, the resulting ASM would appear as follows:

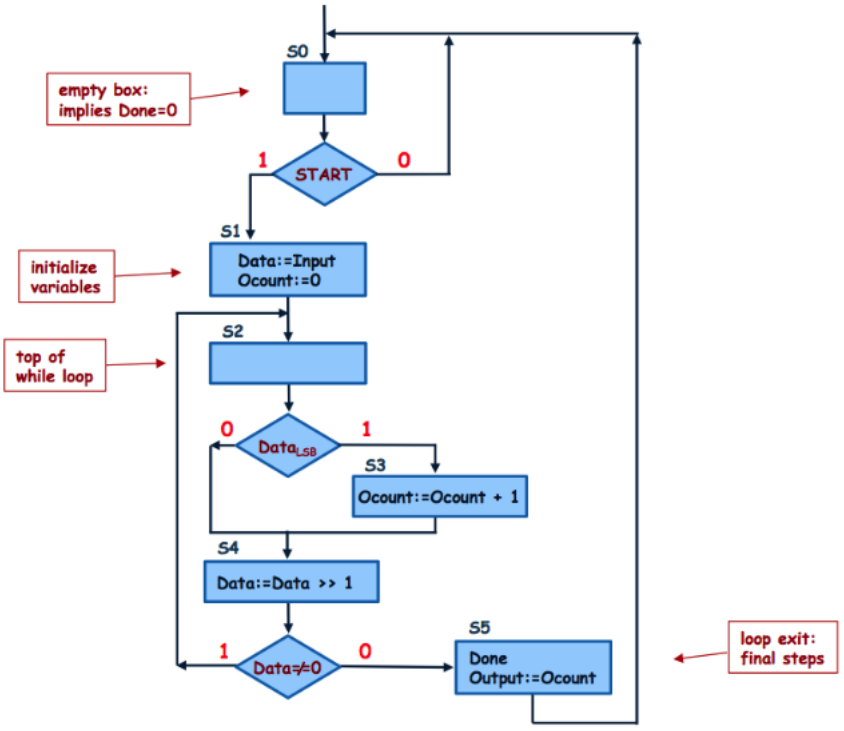

Figure 1: "One's Counter" ASM

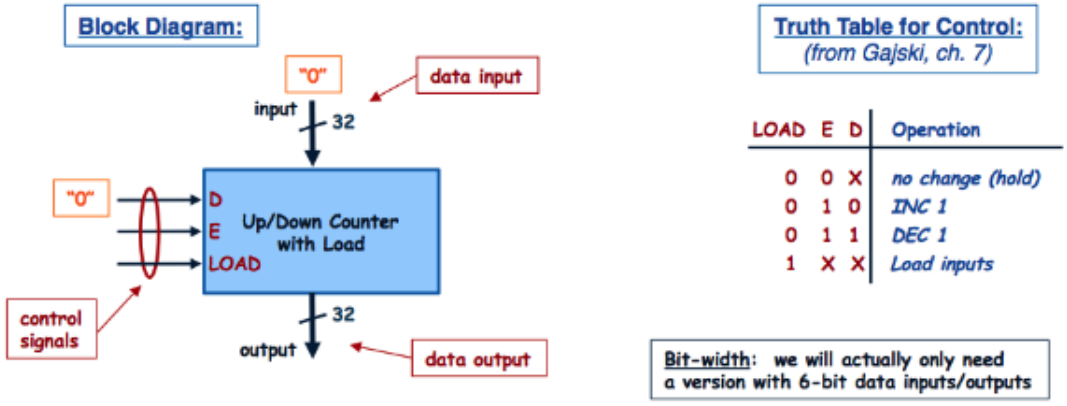

Figure 2: Resource Allocation: Example Data Path Block for 'Data'

Overall, it was the blooRTLs program that hastened the development of the ASM. The remaining steps of the RTL design process heavily rely on this ASM. Each block of the ASM embodies a state. The data path resources must be allocated to accomodate each state (example seen above). Based on the resource allocation, the controller plane can be developed, and ultimately, as one progresses through Nowick's RTL flow, the design can be fully synthesized.