The MMAlpha Test Bench*

Patrice Quinton

February 24, 2010

Abstract

We describe a program to generate test bench automatically for MMAlpha programs. We also describe how to generate stimuli files, and how to generate a bit true simulator.

1 Introduction

In Section 2, we present how to generate a test bench. In Section 5, we present how to generate stimuli files. In Section 6, we explain how to simulate a program using a bit true simulation.

2 Test Bench Generation

The `VhdlTestBench` package contains a function that allows a VHDL test bench to be automatically generated for an Alpha program. This program must be in Alpha0 format, such that obtained after a successful utilization of the `a2v` command. This means that the current program (contained in the library variable, `$library`), has been translated into VHDL.

The function is called `vhdlTestBenchGen`. The test bench is produced in a file called `fileTB.vhd`, where `file` is the name of the system currently contained in the `sys` parameter of the `vhdlTestBenchGen` function[1].

The test bench file is written in the current directory of Mathematica. If such a file already exists, it is copied in a temporary file named `file_.TB.vhd` (a previous version of such a file is deleted, if any exists).

The test-bench that is produced must be used with stimuli files for each of the input variables of the Alpha program; it will produce one output file for each output variable of the Alphaprogram. How to produce these stimuli files is explained in Section 5 of this documentation.

The `vhdlTestBenchGen` function allows test benches to be generated for each one of the elements of the library. Recall that, after generating an alpha0 program, all modules produced are contained in an MMAlpha variable called `$library`. The current program is in variable `$result`, and it is usually the main program. Say this program is `fir` (corresponding to an initial `fir.alpha` Alpha program), then executing

```
vhdlTestBenchGen[ ]
```

*Version printed on March 11, 2010

[1]See the usage of this function.

will produced a test-bench named `firTB.vhd`. It is essential that the content of `$library` be consistent with the program for which the test bench is generated.

The `vhdlTestBenchGen` function checks various possible errors:

- The given program must be recognized as a cell, a module or a controller by MMALPHA (using the `checkCell`, `checkModule`, or `checkController` functions.
- The name of the given program must appear in the library contained in `$library`.
-

More can be found on this topic

3 Technicalities

- There is a provision for generating a VHDL file with the 1987 VHDL syntax (see the `s87` flag, directly in the code).
- The clock rate is 20 ns, and the clock initialization time is 30 ns.
- If there is already a `component` file for the program contained in `$result` (say for example, `fir.component`) in the directory, this component is used in the test bench, otherwise, a component is built on the fly (using the `buildComponent` function). Therefore, if there exists an inconsistency between the program and its component, an error may happen.

4 Meaning of the signals

Here is a description of some of the signals in the test bench produced.

- `clk_rate`:
- `clk_init`:
- `sig_Name`: a signal associated to ALPHA variable `Name`;
- `comp`: the name of the instanciated component;
- `ce`: clock enable signal;
- `rst_0`: initial value of the reset signal;
- `temp_buffer`:
- `temp_buffer_out`:
- `endstim`: a boolean variable that indicates the end of the stimulus file
- `good`: a boolean variable used to check the reading of a stimulus file
- `temp_buffer_Name`: a temporary buffer for ALPHA variable `Name`;

- `stim_file_Name`: the declaration of the stimulus file for variable `Name`:
- `stim_line_Name`: the declaration of the file line for variable `Name`:
- `timecounter`: time counter for the design;

5 Producing Stimuli

As already said, there exists a stimuli file for each input variable. If `inVar` is the name of an input variable of the ALPHAprogram, then its stimuli file is named `stimInVar.txt`. This file contains one value for each instant of time for which a value is required by the program.

Recall that, for any ALPHA variable, its *life time* is given by projecting its domain on the `t` index. Say this life interval is $10 \leq \mathtt{t} \leq 20$, then values will be required only for time instants between 10 and 20, therefore, the stimuli file contains 11 values.

Producing automatically stimuli files can be done either manually, or by using the `cGen` program, as described here.

To better explain this program, assume we want to produce stimuli files for the following program `fir`:

```
system fir :   {K,N | 3<=K<=N-1}
               (x : {i | 0<=i<=N} of integer[S,16];
                w : {k | 1<=k<=K} of integer[S,16])
       returns (y : {i | K<=i<=N} of integer[S,32]);
var
  YLOC: {i,k | K<=i<=N; 0<=k<=K} of integer[S,32];
let
  YLOC[i,k] =
    case
      {| k = 0 }: 0[];
      {| k > 0 }: YLOC[i,k-1] + w[k]*x[i-k];
    esac;
  y[i] = YLOC[i,K];
tel;
```

We load this file, and we schedule it:

```
load[ "file.alpha" ];
scd[]
```

Then, we run the following program

```
cGen[ "file.c", { "K" -> 3, "N" -> 10 }, stimuli -> True  ]
```

This produces in file `file.c` a `C` program that simulates the operations of the ALPHA program. Compiling this program :

```
gcc file.c -o fir
```

produces a binary file `fir` that we can run

```
./fr
```

The effect of this execution is to ask the values of the input variables and to return, eventually, the values of the output variables. Here is what you see at the console:

```
x[0]?0
x[1]?1
x[2]?2
x[3]?3
x[4]?4
x[5]?5
x[6]?6
x[7]?7
x[8]?8
x[9]?9
x[10]?10
w[1]?1
w[2]?0
w[3]?0
y[3]=2
y[4]=3
y[5]=4
y[6]=5
y[7]=6
y[8]=7
y[9]=8
y[10]=9
```

The input chosen are `x[i] = i`, and `w[0] = 1`, `w[1] = w[2] = 0`, which gives the result shown.

As a side effect, this evaluation produces also 3 files: input files `stim_x.txt` and `stim_w.txt` where input values for x and w are recorded, and an output file `stim_y.txt`. Here is the `stim_x.txt` file:

```
x[0]=00000001
x[1]=00000002
x[2]=00000003
x[3]=00000004
x[4]=00000005
x[5]=00000006
x[6]=00000007
x[7]=00000008
x[8]=00000009
x[9]=0000000a
x[10]=00000000
```

Here is the `stim_w.txt` file:

```
w[1]=00000001
w[2]=00000000
w[3]=00000000
```

Here is the `stim_y.txt` file:

```
y[3]=00000003
y[4]=00000004
y[5]=00000005
y[6]=00000006
y[7]=00000007
y[8]=00000008
y[9]=00000009
y[10]=0000000a
```

The `cGen` function has several options. The `noPrint` option, when set to `True`, does not prints out the name of the inputs and outputs values (for example, `x[1]=00000001`, but only the result. The resulting files may then be used to be run with the test bench.

This example is presented in the `Fir` notebook in `mmalpha/demos/NOTEBOOKS/Fir`.

6 Bit True Simulation

Documentation to be done.

7 Generation of test files

The `VhdlTestBench` package contains now a new function, called `alpHardStim`, which allows stimuli files to be generated for the VHDL test bench. For each input `in`, it produces an input file called `alpHardStimin.txt`, and similarly for each output. This file contains, in the order given by the time variable, the command to be executed to generate...

8 The `alpha2mma` Package

There exists a prototype program that allows an ALPHA program to be translated into MATHEMATICA. This package is called `alpha2mma.m`. It is currently available in the directory

```
$MMALPHA/doc/Packages/Meta
```

as it is an example of using the `Meta` package.

The `alpha2mma.m` program is itself produced by the use of the `meta` command, applied to a description of the ALPHA syntax given in the meta file `alpha2mma.meta`. See in the notebook `meta.nb` located in the same directory how to call the translator.

Once the `alpha2mma.m` file has been produced, one loads it by the command

```
<<alpha2mma.m
```

Then to translate a program, say `test1.alpha` into a MATHEMATICA program, one does:

```
load[ "test1.alpha" ];
alpha2mma[]
```

(there exist a `debug` option, as usual).

This produces in the current directory a file called `test1.simul`, which contains definitions for the input, local and output variables.

Then, one loads this file:

```
<<test1.simul
```

and one calls the simul function :

```
simul["test1"]
```

This function has the only effect of entering the context `Alpha‘Simul‘test1‘` in which all definitions of the variables are available, as functions.

To be more explicit, assume that the `test1.alpha` file looks as follows:

```
system test1   (a :  integer; b :  integer)
returns (c :  integer);
let
  c = a + b;
tel;
```

This program is translated into the following MATHEMATICA program:

```
Begin["Alpha‘Simul‘test1‘"];
Clear[ Alpha‘Simul‘test1‘a, Alpha‘Simul‘test1‘b, Alpha‘Simul‘test1‘c];

    (* Definition of c *)
    Alpha‘Simul‘test1‘c[f:(_Integer|_Symbol)...] := Plus[ a[f], b[f] ];
    Alpha‘Simul‘test1‘c[ ___ ] := Message[Alpha‘Simul‘test1‘c::params ];
End[];
```

As you can see, this program enters the special context `Alpha‘Simul‘test1‘`, then clears the variables `a`, `b` and `c` in this context, and defines a `c` function. The form of this function is quite special. It says basically that `c[ f ] := a[ f ] + b[ f ]`, where `f` is any sequence of integers or symbols. This allows the function to be used for any sequence of indices, a property that will be useful later on when one wants to evaluate a structured ALPHA program.

When one runs `simul[ "test1" ]`, one enters the (unique) simulation context of `test1`, where the definition of `c` are available. In this context, if we just evaluate `c[]`, we get the result `a[] + b[]`; in the same manned, we can evaluate any `c[ i1, i2, ..., ik ]` expression, where the `ik` are either integers or symbols of MATHEMATICA, to get the result `a[ i1, i2, ...  , ik ] + b[ i1, i2, ...  , ik]`.

Consider now a more involved example, where one wants to build a binary adder in ALPHA. The program is given in Fig. 1.

It contains two parts: a full adder (`FullAdder` program), and a `Plus` program, that allows one to build a binary adder of `W` bits by instanciating `W` times the full adder.

Translating this program into MATHEMATICA involves (currently) two steps. First, one translate the `FullAdder` program. To this end, assuming that the program is in the file `fulladder.alpha`, one loads this program, one select the `FullAdder` subsystem, and one translates it :

```
system FullAdder (A :  boolean;
                  B :  boolean;
                  Cin :  boolean)
       returns   (X :  boolean;
                  Cout :  boolean);
let
  X[] = A xor B xor Cin;
  Cout[] = A and B or A and Cin or B and Cin;
tel;

system Plus :  {W | 2<=W}
               (A : {b | 0<=b<=W} of boolean;
                B : {b | 0<=b<=W} of boolean)
       returns (S : {b | 0<=b<=W} of boolean);
var
  Cin : {b | 0<=b<=W-1} of boolean;
  Cout : {b | 0<=b<=W-1} of boolean;
  X : {b | 0<=b<=W-1} of boolean;
let
  use {b | 0<=b<=W-1} FullAdder[] (A, B, Cin) returns  (X, Cout) ;
  Cin[b] =
      case
        { | b=0} : False[];
        { | 1<=b} : Cout[b-1];
      esac;
  S[b] =
      case
        { | b<=W-1} : X[b];
        { | b=W} : Cout[W-1];
      esac;
tel;
```

Figure 1: The binary Plus program

```
load[ "fulladder.alpha" ];
getSystem[ "FullAdder" ];
alpha2mma[];
```

One does the same for the main program, `Plus`

```
getSystem[ "Plus" ];
alpha2mma[];
```

Then, one loads both programs :

```
<<"Plus.simul"
<<"FullAdder.simul"
```

which creates definitions for the variables

A The Test Bench for the Fir Filter

```
--------------------------------------------------------------------------------
-- VHDL test bench file for system fir
-- Generated with vhdlTestBench.m at 2/25/2010
--------------------------------------------------------------------------------
library ieee;
USE ieee.std_logic_textio.all;
USE std.textio.all;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE ieee.numeric_std.all;

USE work.all;
USE work.types.all;
USE work.definition.max;
USE work.definition.min;

ENTITY testbench_fir is
END testbench_fir;

ARCHITECTURE behavioural OF testbench_fir IS

COMPONENT fir
PORT(
  clk: IN STD_LOGIC;
  CE : IN STD_LOGIC;
  Rst : IN STD_LOGIC;
  x : IN  SIGNED (15 DOWNTO 0);
  w : IN  SIGNED (15 DOWNTO 0);
  y : OUT  SIGNED (31 DOWNTO 0)
);
END COMPONENT;

-- Design independent signals

  -- Integers to and from tested component

  SIGNAL rst_0 : std_logic := '0';
  SIGNAL clk : std_logic := '0';
  SIGNAL ce : std_logic := '0';

  CONSTANT clk_rate : TIME := 20 ns;

  CONSTANT clk_init : TIME := 50 ns;
```

```
---- Design dependent signals
  -- Inputs
  SIGNAL sig_x :  SIGNED (15 DOWNTO 0);
  SIGNAL sig_w :  SIGNED (15 DOWNTO 0);
  -- Outputs
  SIGNAL sig_y :  SIGNED (31 DOWNTO 0);


BEGIN

---- Instantiation of the components

  comp : fir PORT MAP( clk => clk, ce => ce, rst => rst_0, x => sig_x, w => sig_w, y => sig_y )

-- clock, clock enable and reset generation

  ce <= '1' AFTER clk_rate;
  rst_0 <= '1' AFTER clk_init;
  clk <= NOT clk AFTER clk_rate;

-- Process start
stimuli : PROCESS( clk, rst_0 )

-- Design independent variables
  VARIABLE temp_buffer:        STD_LOGIC_VECTOR (31 DOWNTO 0);
  VARIABLE temp_buffer_out: STD_LOGIC_VECTOR (31 DOWNTO 0);
  VARIABLE endstim : BOOLEAN := FALSE; -- end of stimulation
  VARIABLE good : BOOLEAN; -- check current read in stimuli file
  VARIABLE i,i1,i2,i3,i4 : INTEGER; -- loop counter
  VARIABLE j : INTEGER; -- loop counter
  CONSTANT space : STRING := "  ";  -- Blank string

---- Design dependent variables
  --Inputs
  VARIABLE x :  SIGNED (15 DOWNTO 0);
  VARIABLE w :  SIGNED (15 DOWNTO 0);
 --outputs
  VARIABLE y :  SIGNED (31 DOWNTO 0);

---- Design dependent Buffers
  -- Inputs
  VARIABLE temp_buffer_x : STD_LOGIC_VECTOR (31 DOWNTO 0) := (OTHERS => '0');
  VARIABLE temp_buffer_w : STD_LOGIC_VECTOR (31 DOWNTO 0) := (OTHERS => '0');
 --outputs
  VARIABLE temp_buffer_y : STD_LOGIC_VECTOR (31 DOWNTO 0) := (OTHERS => '0');
```

```
---- Design dependent file declaration
  --Inputs
  FILE stim_file_x :text OPEN READ_MODE IS "stim_x.txt";
  FILE stim_file_w :text OPEN READ_MODE IS "stim_w.txt";
  --Outputs
  FILE stim_file_y :TEXT OPEN WRITE_MODE IS "stim_y.txt";

---- Design dependent line declaration
  --Inputs
  VARIABLE stim_line_x : LINE ;
  VARIABLE stim_line_w : LINE ;
  --Outputs
  VARIABLE stim_line_y : LINE ;


  VARIABLE timecounter : INTEGER := -1; -- indicate the step t.
  -- Initialisation is design dependent

BEGIN
  IF rst_0 = '0' THEN
  -- Signal initialisation


  ELSIF clk'EVENT AND clk='1' THEN
---- Reading stimuli files

  IF (timecounter >= 0) AND (timecounter <= 100) THEN
  FOR i IN 0 TO 100 LOOP
    READLINE(stim_file_x, stim_line_x);
    HREAD(stim_line_x, temp_buffer_x, good);
    x := SIGNED(temp_buffer_x  (15 DOWNTO 0));
   ASSERT good REPORT "read text i/o read error" SEVERITY ERROR;
  END LOOP;
  END IF;

  IF (timecounter >= 1) AND (timecounter <= 3) THEN
  FOR k IN 1 TO 3 LOOP
    READLINE(stim_file_w, stim_line_w);
    HREAD(stim_line_w, temp_buffer_w, good);
    w := SIGNED(temp_buffer_w  (15 DOWNTO 0));
   ASSERT good REPORT "read text i/o read error" SEVERITY ERROR;
  END LOOP;
  END IF;
```

```
-- Affectation to signals

  sig_x <= x;
  sig_w <= w;

  y := sig_y;


-- Writing stimuli to files

  IF (timecounter >= 4) AND (timecounter <= 100) THEN
  FOR i IN 3 TO 100 LOOP
    temp_buffer_y  (31 DOWNTO 0) := std_logic_vector(y());
    HWRITE(stim_line_y, temp_buffer_y);
    WRITELINE( stim_file_y , stim_line_y);
  END LOOP;
  END IF;


-- End of process, increment of timecounter

  timecounter := timecounter+1;

  ASSERT NOT endstim REPORT "end of stimuli file. Stop the rtl!" SEVERITY ERROR;
  endstim := ENDFILE(stim_file_x) OR ENDFILE(stim_file_w) ;

  -- severity error does not stop the simulation, whether failure does!
  -- But the process has to continue until the end of the calculation

  ASSERT timecounter <= 101
-- May be upBound ?
  REPORT " Failure asked on purpose, normaly result is written in stim_y.txt "
  SEVERITY FAILURE;
  END IF;

END PROCESS;

END BEHAVIOURAL;
```

B The Stimuli Generator for the Fir Filter

This program was generated with the default value of the `noPrint` option.

```
/* Generated: 25/2/2010 at 14:58:58 */
/* Code generated by MMAlpha code generator version 0.2.6 (02/02/2001 11:35) FQ */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>




#define min(a,b) ((a) > (b) ? (b) : (a))
#define max(a,b) ((a) > (b) ? (a) : (b))
#define power(a, i) ((a)^(i))
static int rfloor (int a, int b) {
  assert (b>0);
  return ((a<0) ? ((a+1)/b)-1 : a/b);
}
static int rceil (int a, int b) {
  assert (b>0);
  return ((a>0) ?  ((a-1)/b)+1 : a/b);
}

void fir(short* _x, short* _w, int* _y) {
  /* Aliases for all variables */
#define x(i) _x[(i)]
#define w(k) _w[(-1 + (k))]
#define y(i) _y[(-3 + (i))]
#define YLOC(t, p) _YLOC[(-12 + 4*(p) + (t))]
#define outy(t, p) y(p)

  /* Allocate memory for local variables */
  int * _YLOC = (int *) malloc(sizeof(int)*(32));


  /* Loop variables */
  int i;
  int p;
  int t;

  /* A few loops */
  t = 0;
```

```
  for (p = 3; p <= 10; ++p) {
    YLOC(t, p) = 0;
  }
  for (t = 1; t <= 3; ++t) {
    for (p = 3; p <= 10; ++p) {
      YLOC(t, p) = (YLOC((-1 + t), p) + (w(t) * x((p + -t))));
    }
  }
  t = 4;
  for (p = 3; p <= 10; ++p) {
    outy(t, p) = YLOC(3, p);
  }

  /* Clean up local variables' memory */
  /* Commented out because it was crashing at run time*/
  /*
  free(_YLOC);

  */
  /* And finally undef aliases */
#undef x
#undef w
#undef y
#undef YLOC
#undef outy
}

int main (void) {
#define x(i) _x[(i)]
#define w(k) _w[(-1 + (k))]
#define y(i) _y[(-3 + (i))]
  short * _x = (short *) malloc(sizeof(short)*(11));
  short * _w = (short *) malloc(sizeof(short)*(3));
  int * _y = (int *) malloc(sizeof(int)*(8));
  int i;
  int k;
   FILE   *file_x=fopen("stim_x.txt","w");
   FILE   *file_w=fopen("stim_w.txt","w");
   FILE   *file_y=fopen("stim_y.txt","w");
  for (i = 0; i <= 10; ++i) {
    fprintf(stdout, "x[%i]?", i);
    fscanf(stdin, "%i", &(x(i)));
    fprintf(file_x,"x[%i]=", i);
    fprintf(file_x,"%.8x\n", (x(i)));
  }
```

```
  for (k = 1; k <= 3; ++k) {
    fprintf(stdout, "w[%i]?", k);
    fscanf(stdin, "%i", &(w(k)));
    fprintf(file_w,"w[%i]=", k);
    fprintf(file_w,"%.8x\n", (w(k)));
  }
  fir(_x, _w, _y);
  for (i = 3; i <= 10; ++i) {
    fprintf(stdout, "y[%i]=%i\n", i, y(i));
    fprintf(file_y,"y[%i]=%.8x\n", i, y(i));
  }
  fclose(file_x);
  fclose(file_w);
  fclose(file_y);
  free(_x);
  free(_w);
  free(_y);
#undef x
#undef w
#undef y
   exit(0);
}
```