# Tutorial on FinSimMath(TM)

## (an extension of Verilog(R) for Mathematical Descriptions)

by Alec Stanculescu, PhD
Fintronic USA, Inc.

San Mateo CA - July 24, 2009

 

# FinSimMath(TM)

## an extension of Verilog(R) for Mathematical Descriptions

**OUTLINE**



 

# 1. Introduction

FinSimMath's creation was motivated by the need for having mathematical modeling within the Verilog language. This language was designed with the intent that (1) no explicit conversion functions are necessary, (2) runtime changes of formats including the number of bits of the various fields are supported, and (3) data in multi-dimensional arrays are easy to access globally.

FinSimMath suports a large number of mathematical system tasks, and provides access to information regarding the occurrence of overflow, underflow, maximum number of bits needed, and cummulative error.

# 2. Overview of FinSimMath

FinSimMath is an extension of the IEEE std 1364 Verilog language which supports also the types VpDescriptor, VpReg (for variable precision objects), VpCartesian, VpPolar, VpFCartesian, and VpFPolar types. Logical, Arithmetic and assignment operators are defined to operate on all combination of these types including on arrays and matrixes.

Objects of the variable precison types VpReg, VpCartesian, and VpPolar can have their formats (fixed or floating) and the sizes of the format fields modifiable at runtime. This allows for a tight loop in finding optimal formats and sizes of sub-fields, given various costs based on computation accuracy, overflow avoidance, quantization noise, power consumption (switching activity), or other resource constraints.

Global writing to and reading from multi-dimensional arrays are supported using positional system tasks for each range within the system tasks $InitM and $PrintM.

A general form of aliasing using positional system tasks for each dimension of a multi-dimensional array is introduced with the *View as* construct, enabling to declare multi-dimensional arrays that are contained within an already declared multi-dimensional array. Using this capability one can separate data from its actual location within a multi-dimensional array.

 

A rich mathematical environment is available based on a number of system functions and tasks, including: $VpSin, $VpCos, $VpTan, $VpCtan, $VpAsin, $VpAcos, $VpAtan, $VpActan, VpSinh, $VpCosh, $VpTanh, $VpCtanh, $VpAsinh, $VpAcosh, $VpAtanh, $VpActanh,$VpPow, $VpPow2, $VpLog, $VpLn, $VpAbs, $VpFloor, $VpHypot, $VpFft, $VpIfft, $VpDct, $VpIdct, $VpNormAbsMax, $VpNormAbsSum, $VpNorm-RMS, $VpDistAbsMax, $VpDistAbsSum, etc.

# 3. Basic FinSimMath

## 3.1 Declaring VP objects/data

VpReg is a predefined type. Objects of this type can have their formats modifiable at runtime.
The size of the *packed data* is the maximum size that the object can have during the simulation.

```
VpReg [0:511] mySecondReg;
```

 

## 3.2 Declaring VP descriptors

VpDescriptor is a predefined type. Objects of this type can store information regarding the format of data objects associated to it.

```
VpDescriptor mySecondDescriptor;
```

## 3.3 Associating VP descriptors to VP data

Before being used a VP data must be associated to a descriptor via a call to the system task

```
$VpAssociateDescriptorToData(data,
                             descriptor);
```

 

## 3.3 Semantics of the fields of VP descriptors

The macroes below are defined in finsimmath.h which can be included in any Verilog module to be simulated by FinSim.

**Field 1:** Size of integer part for fixed point or size of exponent plus one for floating point.
**Field 2:** Size of fractional part for fixed point or size of mantissa for floating point.
**Field 3:** Format options:

```
`define TWOS_COMPLEMENT 1
`define SIGN_MAGNITUDE 2
`define FLOATING 3
```

**Field 4:** Rounding options:

```
`define TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF 1
`define TO_NEAREST_INTEGER_IF_TIE_TO_PLUS_INF 2
`define TO_NEAREST_INTEGER_IF_TIE_TO_ZERO 3
`define JUST_TRUNCATE 4
`define TO_ZERO 5
`define TO_INF 6
`define TO_MINUS_INF 7
`define TO_PLUS_INF 8
```

**Field 5:** Overflow options:

```
`define SATURATION 1
`define NORMAL 2
'define WARNING 64
```

**Field 6:** Various flags:

```
`define REPORT_SPECIAL_CONDITION 64
```

 

## 3.4 Modifying fields of VP descriptors

Predefined values of fields of type VpDescriptor are declared in finsimmath.h which can be included in any FinSimMath model:

```
`include "finsimmath.h"
```

The size of the formats can be changed during the execution of the simulation, by using the system tasks

```
$VpAssociateDescriptorToData(data,
                                descriptor),
$VpSetDescriptorInfo(descriptor, field1,
field2, field3, field4, field5, field6),
```

**and**

```
$VpSetDefaultOptionsfield1, field2, field3,
field4, field5, field6).
```

## 3.5 Assigning Verilog expressions to VP objects.

Verilog expressions are evaluated and their values placed in the VP object according to the information present in the associated descriptor.

For example:

```
op1 = op2 + op3;
```

 

will perform the addition of the values in op2 and op3 and place the result in op1, where op1, op2, and op3 may have any scalar type: VpReg,  real, integer, literal real, literal integer.

The same expression will perform the addition in case op1, op2, and op3 are of any of the types: VpComplex, VpPolar, VpFComplex, VpFPolar.

The same expression will perform the addition in case op1, op2, and op3 are matrixes of compatible sizes where the elements can be either scalar or of the four Cartesian and Polar types.

## 3.6 Assigning VP objects to Verilog objects

Assignements to objects of type real result in the object of type real having a value as close as possible to the value being assigned.

Assignements to objects of type integer or reg result in the object on the left hand side containing the same bit patern as the object on the right hand side.

## 3.7 Displaying VP objects

The Verilog system tasks for displaying objects have been extended with the following format representations:

- `%y`: real number representation,
- `%k`: hex representation

 

- `%h`: binary representation

## 3.8 Logical and Arithmetic Operators

Verilog standard arithmetic operators `(+,-,*,/,**)` may be used in conjunction with variable precision objects (including cartesian and polar objects), as well as with multidimensional arrays of such objects, without the need of explicit conversion functions.

Verilog standard logical operators (>, >=, <, <=, ==, !=) may be used in conjunction with variable precision objects.

 

# 4. Hierarchical Evaluation of Expressions

Scalar objects may be used in hierarchical expressions. Subexpressions are evaluated in temporary VP objects. The evaluation is governed by the  default descriptor information, as well as by the descriptors of the operands where applicable. The descriptor of the operands are used in a way in which to minimize possible errors due to overflow or underflow.

FinSim does not yet support hierarchical expression with operands that are multi-dimensional arrays. Such expressions must be split into simple expressions, each having at most one operator.

 

# 5. Cartesian and Polar Types

## 5.1 VpCartesian

This type consists of two VP fields and objects of this type must be associated to a descriptor before usage. The two fields represent cartesian co-ordinates and are treated as such by the operators operating on them.

## 5.2 VpPolar

This type consists of two VP fields and objects of this type must be associated to a descriptor before usage. The two fields represent polar co-ordinates and are treated as such by the operators operating on them.

## 5.3 VpFCartesian

This type consists of two fields of type real which represent cartesian co-ordinates.

## 5.4 VpFPolar

This type consists of two fields of type real which represent polar co-ordinates.

 

## 5.5 Operations on types Cartesian and Polar +, -, *, /, **,==, !=

## 5.6 Mixing Cartesian and Polar operands in the same simple expression

```
myPolar = {1.0, $VpGetPi()};
myCart = {1.0, 1.0};
myCart = myCart + myPolar;
$display("myCart.Re = %y\n", myCart.Re);
```

will print: myCart.Re= 0.0

 

# 6. Multi-dimensional Arrays

## 6.1 + , – , * , / ,  **

These operators are defined on two dimensional arrays.

Usual constraints are placed on the sizes of each dimension:
a) for + and - the sizes of each dimension must be the same.
b) for * and / the size of the second dimension of the first operand must be equal to the size of the first dimension of the second operand.
c) No constraints are imposed on the operand of **, as both the inverse and the pseudo inverse operations are supported.

Note: Currently, FinSim has a limit of 4000 for the size of one dimension of a matrix, unless a call to $ToSparse(matrix) occurs at the beginning of an initial block. In such a case, FinSim works for matrices of up to 40,000,000 by 40,000,000.

 

# 6.2 Accessing copied data via position system tasks

This is achieved using the system task $InitM(myMem, value), where value stands for an expression in terms of system functions $I1 through $In with n being the number of dimensions of myMem. $In represents the index of the n-th dimension of the current location.

The effect of the call is that for all combinations of indexes myMem[$I1]..[$In] = value.

For complex operands (e.g. VpPolar) value stands for two arguments, one for each element of the complex object.

For example:

```
$InitM(myMem, oMem[$I2][$I1]);
$InitM(myPMem, pMem[$I2][$I1].Mag,
        Mem[$I2][$I1].Ang);
```

Will result in the two dimensional arrays myMem and myPMem receiving the data of the transposed of the two dimensional arrays oMem and pMem respectively.

 

## 6.3 Creating views of multi-dimensional data

A view declaration creates an object which when referenced represents data selected from another multi-dimensional array whithout copying the data, as in the exmaple below:

```
real myMem[0:SIZE-1][0:SIZE-1];
View real myView[0:SIZE-1][SIZE-1] as
myMem[$I2][$I1];
```

$I1, and $I2 in the View construct represent the position of each element within the view declaration (myView in this example).

As a result of the above View declaration any reference to myView or to any of its elements will get the transposed of myMem. However, the data is not copied and therefore any writing to myView will change myMem.

## 6.4 Displaying multi-dimensional data

This is achieved using $PrintM(myMem, format) where format stands for "%y" with y being the format in which the elements of myMem will be displayed.

## 6.5 Norm and Distance

FinSimMath supports the following norms:
- $VpNormAbsMax(matrix) - maximum absolute value of all elements

 

- $VpNormAbsSum(matrix) - sum of absolute values of all elements

- $VpNormRMS(matrix) - square root of sum of squares of each element divided by the number of elements

FinSimMath supports the following distances:
- $VpDistAbsMax(matrix) - the difference between the maximum absolute value of each matrix.

- $VpDistAbsSum(matrix) -sum of absolute differences between the corresponding values of two matrices

## 6.6 Sparse Matrices

Matrices can be declared sparse by calling $SpSparse(matrix) at the begining of an initial block. Operations on such matrices are a little slower than on regular matrices, however they can be much larger. Version 10_04_00 or higher support at least 40,000 by 40,000.

The following functions are support for accessing sparse matrices:

- $SpReadNextNzElemInLine(matrix, line, col, idx, value), where the inputs are matrix, line, and idx and the outputs are col, idx and val. To obtain the first element in a line one must set idx to -1. Upon execution idx will contain the handle to the next non-

 

zero element, col will contain the column of that element and val the value of that element.

- $SpReadNextNzElemInCol(matrix, line, col, value), where the inputs are matrix, line, and col and the outputs are line and value of teh next non-zero element. To obtain the first element in a column one has to set the variable line to -1.

- $SpNulifyLine(matrix, line) - sets to zero all elements of the line indicated by line.
- $SpNulifyCol(matrix, col) sets to zero all elements of the column indicated by col.

- $SpExchangeLine(matrix, line) - exchanges line indicted by line with line zero.

-$SpExchangeCol(matrix, col) - exchanges column indicated by col with column zero.

## 6.7 Associative matrices

A matrix can be declared to be associative by a call to $SpToAssociative(matrix) at the beginning of an initial block.
The following system tasks can accept as arguments associative matrices:

- $SpAssocGetNext(matrix, val, line, col, idx) - has as inputs matrix, val and idx and as outputs line, col and idx of the next element having the value val. In order to obtain the first element

 

which has the value val one must set idx to -1. The order in which the elements are obtained is the lower line first and if the lines are the same the lower column first.

## 6.8 Solving Differential Equations

Support for solving differential equations is provided by the system task $VpLODE(order, nrEq, h, nr_pts_per_ct_coef+1, x_ct, coef, Fe_ct, y_ct, ressymb), where:
- order indicates the order of the differential equation, i.e. the highest derivative that is involved,
- nrEq indicates the number of equations,
- h is the double of the sampling period
- nr_pts_per_ct_coef is the size of the sampled data,
- x_ct is a two dimensional array which contains the solution after the execution of the system task, whereas the array x_ct[0] contains the initial conditions,
- coef is an array which contains the value of the coefficients of the equation,
- Fe_ct is a two dimensional array containing the samples of the values that are independent of the functions to be found,
- y_ct is a two dimensional array containing after the execution of the system task the first derivative of the solution in case the order is two or a three dimensional array containing the the first and subsequent derivatives till order-1. Note that before the call, the initial conditions have be be provided in y_ct[0],
- ressymb is a an array which contains the symbolic values if they exist.

 

 

# 7. Practical Exercises Written in FinSimMath

These examples are running on  Super FinSim version 10_0_0 or subsequent versions.

FinSimMath may be usable in the future in conjunction with other standard compliant Verilog or SystemVerilog simulators.

 

# 7.1 Example of Verilog modules exchanging VP values

Instances of modules may exchange values either via external references or via ports. The example below shows how module top instantiates a module VpAdd and passes to it two operands to be added.

Note that the passing of vp data via ports is done while making sure that the data objects in both the instantiating module and the instantiated module use descriptors with the same formats and same size for the corresponding fields of the formats.

 

```
module vpadd(in1w, in2w, out);
input in1w;
input in2w;
output out;
(* varprec = data *)
wire [0:511] in1w;
(* varprec = data *)
wire [0:511] in2w;
(* varprec = data *)
wire [0:511] out;
VpDescriptor d1;
VpReg [0:511] in1;
VpReg [0:511] in2;
VpReg [0:511] outR;

initial begin
  $VpSetDescriptorInfo(d1, 256, 96,`TWOS_COMPLEMENT,
  `TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
  `SATURATION, 1);
  $VpAssocDescrToData(in1, d1);
  $VpAssocDescrToData(in2, d1);
  $VpAssocDescrToData(outR, d1);
end
assign out = outR;
always @(in1w or in2w)
begin
in1 = in1w;
in2 = in2w;
outR = in1 + in2;
end
endmodule
```

 

```
module top;
VpReg [0:511] in1;
VpReg [0:511] in2;
VpDescriptor d1;
(* varprec = data *)
wire [0:511] w;

vpadd add1(in1, in2, w);

initial begin
  $VpSetDescriptorInfo(d1, 256, 96,`TWOS_COMPLEMENT,
  `TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
  `SATURATION, 1);
  $VpAssocDescrToData(in1, d1);
  $VpAssocDescrToData(in2, d1);
#10;

in1 = 2;
in2 = 3;

#2;
in2 = w;
$display("in2 = %y\n", in2);

end
endmodule
```

 

# 7.2 Butterworth LP IIR order 5 filter using operands of type VpFCartesian.

The type VpFCartesian consists of two fields of type real making the execution faster than when using objects of types whose formats can be modified at run time, as in the example in 7.3.

The error is measured as a vector distance between the output and the ideal output. The frequency spectrum of the output is displayed in both cartesian and polar coordinates.

This step is typically used to make sure that the algorithm works properly.

 

```
module top;

parameter SIZE = 32 * 32;
parameter ORDER= 6;

VpFCartesian in[-ORDER+1:SIZE-1], out[-
ORDER+1:SIZE-1], idealOut[0:SIZE-1];
VpFPolar in_polar[0:SIZE-1], polar_s;

real a [0:ORDER-1];
real b [0:ORDER-1];
real t[0:ORDER-1], s[0:ORDER-3];

real delta;

integer i, j, k;
real distance;

initial begin

/**********************************
1. Load input in in.Re and load ideal output
in idealOut.Re
 Notation:
 a) sampling_rate: time passed between loading
consecutive values in in.Re.
 b) SIZE: number of samples
 c) delta: 2*Pi/SIZE is a constant chosen such
that values $VpSin(n*delta*j)
 with 0 <= j < SIZE represents a sinusoid as
function of time with frequency freq =
((sampling_rate/2)/SIZE) * n, in other words
```

 

```
within the time span of the collection of all
SIZE samples, there are n complete periods of
the sinusoid.
2. Initialize ORDER number of values of the
history of in and out for the filter to oper-
ate in best conditions.

The values are chosen to be zero in this case.
Other values may be better in other circum-
stances.
**********************/
delta = (2*$VpGetPi()) / SIZE;
$InitM(in, (($I1 <= 0) ?
  0.0 : $VpSin(delta * $I1) +
        $VpSin((SIZE/4)*delta*$I1)/10), 0.0);
$InitM(idealOut, $VpSin(delta * $I1)/10, 0.0);
$InitM(out, 0.0, 0.0);


/************************************
3. Load coeficients of Butterworth IIR LP fil-
ter with passband: 0-500 Hz (assuming a sample
rate of 8000 samples /sec) effective order= 5
*****************************/

a = { 1.6411125E-4, 8.205562E-4, 0.0016411124,
0.0016411124, 8.205562E-4, 1.6411125E-4 };
b = { 1.0, -3.7314737, 5.693888, -4.420512,
1.7411026, -0.277753 };
/**********************************
4. Perform filtering according to the speci-
fied coeficients and initial values of histo-
```

 

```
ries of in and out
**********************************/
for (k=0; k < SIZE; k = k+1)
begin
  t[ORDER-1] = a[ORDER-1] *
               in[k-ORDER+1].Re;
  for (j = ORDER-2; j >= 0; j = j - 1)
    begin
      t[j] = a[j] * in[k-j].Re + t[j+1];
    end
  s[ORDER-3] = -b[ORDER-1]*
     out[k-ORDER+1].Re - b[ORDER-2] *
     out[k-ORDER+2].Re;
  for (j = ORDER-4; j >= 0; j = j - 1)
  begin
    s[j] = s[j+1] - b[j+1] * out[k-j-1].Re;
  end
  out[k].Re = t[0] + s[0];
end
/**************************************
5. Display sampled values - in[].Re
**************************************/
for (j = 0; j < SIZE; j = j + 1)
begin
  $display("sampled value[j]=%e\n", in[j].Re);
end

/*********************************************
6. Display ideal output values - idealOut
*********************************************/
for (j = 0; j < SIZE; j = j + 1)
```

 

```
begin
  $display("ideal output[%d]=%e\n", j,
           idealOut[j].Re);
end

/************************************
7. Display filtered values - out
************************************/
for (j = 0; j < SIZE; j = j + 1)
begin
  $display("filtered output[%d]=%e\n", j,
           out[j].Re);
end
```

 

```
/*********************************
8. Compute distance between filtered output
and ideal output vectors
*********************************/
distance = $VpDistAbsSum(out, idealOut)/SIZE;
$display("Mean distance between filtered  out
and ideal out samples = %e\n", distance);
distance = $VpDistAbsMax(out, idealOut);
$display("Maximum distance between filtered
out and ideal out samples = %e\n", distance);

/*********************************
9.Display frequency spectrum of input/sampled
 values - in
*************************/
$VpFft(in, 0, SIZE-1);
for (j = 0; j < SIZE/2; j = j + 1)
begin
  $display("in[%d] Re=%e, Im=%e\n", j,
in[j].Re, in[j].Im);
end
$display("finished display of freq dom of
input\n");
```

 

```
/************************************
10.a Display magnitude and phase of spectrum
of input/sampled values using array assignment
with implicit conversion from cartesian to
polar coordinates
************************************/
in_polar = in;
for (j = 0; j < SIZE/2; j = j + 1)
begin
  $display("in_polar[%d] Mag=%e, Ang=%e\n",
      j, in_polar[j].Mag, in_polar[j].Ang);
end

/************************************
10.b Display magnitude and phase of spectrum
of input/sampled values using assignment to
scalar with implicit conversion from cartesian
to polar coordinates
************************************/
for (j = 0; j < SIZE/2; j = j + 1)
begin
   polar_s = in[j];
  $display("polar_s[%d] Mag=%e, Ang=%e\n", j,
   polar_s.Mag, polar_s.Ang);
end
```

 

```
/*********************************
11. Perform $VpIfft on the content of in vec-
tor. The result must be close to the sampled
input. This is just a check for the accuracy
of $VpFft and $VpIfft system tasks for the
given number of bits used (precision)
*****************************************/
$VpIfft(in, 0, SIZE-1);
for (j = 0; j < SIZE/2; j = j + 1)
begin
  $display("should be close to in[%d] Re=%e,
  Im=%e\n", j, in[j].Re, in[j].Im);
end

/*********************************
12. Display frequency spectrum of ideal
    output - idealOut
*********************************/
$VpFft(idealOut, 0, SIZE-1);
for (j = 0; j < SIZE/2; j = j + 1)
begin
  $display("idealOut[%d] Re=%e, Im=%e\n",
        j, idealOut[j].Re, idealOut[j].Im);
end
```

 

```
/**********************
13. Display frequency spectrum of filtered
output - out
***************************/
$VpFft(out, 0, SIZE-1);
for (j = 0; j < SIZE/2; j = j + 1)
begin
$display("out[%d] Re=%e, Im=%e\n", j,
out[j].Re, out[j].Im);
end
end /*initial*/
endmodule
```

 

# 7.3 Butterworth LP IIR order 5 filter using VP objects of fixed and floating point formats.

The size of the fields of the formats are changed at runtime in order to find an acceptable solution.

This example uses FinSimMath's VpCartesian and VpPolar scalar and vector types in conjunction with $VpSin, $VpFft, $VpIfft to demonstrate the implementation of a low pas Butterworth filter and other DSP processing.

 

```
module top;

`include "finsimmath.h"
parameter SIZE = 32 * 32;
parameter ORDER= 6;

VpCartesian in[-ORDER+1:SIZE-1], out[-
ORDER+1:SIZE-1], idealOut[0:SIZE-1];
VpPolar in_polar[0:SIZE-1], polar_s;

VpReg [0:511] tmp;
VpReg [0:511] a [0:ORDER-1];
VpReg [0:511] b [0:ORDER-1];
VpReg [0:1] d1;
VpReg [0:511] t[0:ORDER-1];
VpReg [0:511] s[0:ORDER-3];

real acceptableDistance;
integer notDone,j,k, sizeInt, sizeDec, format;
real delta, dist;

initial begin
$VpAssocDescrToData(s, d1);
$VpAssocDescrToData(t, d1);
$VpAssocDescrToData(in_polar, d1);
$VpAssocDescrToData(polar_s, d1);
$VpAssocDescrToData(a, d1);
$VpAssocDescrToData(b, d1);
$VpAssocDescrToData(in, d1);
$VpAssocDescrToData(out, d1);
$VpAssocDescrToData(idealOut, d1);
```

 

```
/**********************************
Because the there is a distorsion in phase due
to a delay between the filtered output and the
ideal output, the distance depends on the num-
ber of samples per time unit, becoming smaller
with more samples.
**********************************/
if (SIZE == 1024) acceptableDistance = 0.032;
else if (SIZE == 4096)
   acceptableDistance = 0.012;
else begin
  $display(" acceptableDistance is not yet
known for SIZE=%d. Use operands of type real
to determine accptableDistance \n", SIZE);
end

for (format = 0; format < 2;
     format = format + 1)
begin
  if (format == 0)
  begin
    $display("Try Floating point\n");
    sizeInt = 7;
    sizeDec = 14;
  end
  else
  begin
    $display("Try Two's complement\n");
    sizeInt = 7;
    sizeDec = 14;
  end
notDone = 1;
```

 

```
while (notDone)
begin
  if (format == `FLOATING)
  begin
    $VpSetDefaultOptions(sizeInt,
      sizeDec, `FLOATING,
     `TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
     `SATURATION, 1);
    $VpSetDescriptorInfo(d1, sizeInt,
      sizeDec,`FLOATING,
      `TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
      `SATURATION, 1);
   end
   else
   begin
     $VpSetDefaultOptions(sizeInt, sizeDec,
                          `TWOS_COMPLEMENT,
     `TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
     `SATURATION, 1);
      $VpSetDescriptorInfo(d1, sizeInt,
      sizeDec, `TWOS_COMPLEMENT,
     `TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
     `SATURATION, 1);
end
$display("Trying sizeInt=%d, sizeDec=%d\n",
sizeInt, sizeDec);
```

 

```
/**********************************
1. Load input in in.Re and load ideal output
in idealOut.Re
 Notation:
a) sampling_rate: time passed between loading
consecutive values in in.Re.
b) SIZE: number of samples
c) delta: 2*Pi/SIZE is a constant chosen such
that values $VpSin(n*delta*j) with 0 <= j <
SIZE represents a sinusoid as function of time
with frequency
freq = ((sampling_rate/2)/SIZE) * n, in other
words within the time span of the collection
of all SIZE samples, there are n complete
periods of the sinusoid.
************************/

delta = (2*$VpGetPi()) / SIZE;
for (j = 0; j < SIZE; j = j + 1)
begin
  in[j].Re = $VpSin(delta * j) +
       $VpSin((SIZE/4)*delta*j)/10.0;
       idealOut[j].Re = $VpSin(delta * j);
  in[j].Im = 0.0;
  idealOut[j].Im = 0.0;
end
```

 

```
/**********************************
2. load coeficients of Butterworth IIR LP fil-
ter with passband: 0-500 Hz (assuming a sample
rate of 8000 samples /sec) effective order= 5
***********************************/
$display("Loading coeficients\n");
a[0] = 0.00016411125; b[0] = 1.0;
a[1] = 0.0008205562; b[1] = -3.7314737;
a[2] = 0.0016411124; b[2] = 5.693888;
a[3] = 0.0016411124; b[3] = -4.420512;
a[4] = 0.0008205562; b[4] = 1.7411026;
a[5] = 0.00016411125; b[5] = -0.277753;

/**********************************
3. Initialize ORDER number of values of the
history of in and out for the filter to oper-
ate in best conditions. The values are chosen
to be zero in this case. Other values may be
better in other circumstances.
***********************************/
$display("Initialize History\n");
for (j = 0; j < ORDER; j = j + 1)
begin
in[j-ORDER+1].Re = 0.0;
out[j-ORDER+1].Re = 0.0;
end
```

 

```
/*********************************
4. Perform filtering according to the speci-
fied coeficients and initial values of histo-
ries of in and out
*********************************/
$display("Perform Filtering\n");
for (k=0; k < SIZE; k = k+1)
begin
  t[ORDER-1] = a[ORDER-1] * in[k-ORDER+1].Re;
  for (j = ORDER-2; j >= 0; j = j - 1)
    t[j] = a[j] * in[k-j].Re + t[j+1];
    s[ORDER-3] = -b[ORDER-1]*
   out[k-ORDER+1].Re - b[ORDER-2] *
   out[k-ORDER+2].Re;
   for (j = ORDER-4; j >= 0; j = j - 1)
     s[j] = s[j+1] - b[j+1] * out[k-j-1].Re;
     out[k].Re = t[0] + s[0];
  end

  for (j = 0; j < SIZE; j = j + 1)
    $display("filtered output[%d]=%y\n", j,
            out[j].Re);

  $display("Compute Distance\n");
  dist = $VpDistAbsSum(out, idealOut)/SIZE;
 $display("distance between filtered out and
ideal output = %e\n", dist);
  if (dist > acceptableDistance)
  begin
    $display("For sizeDec = %d the distance is
%e, while acceptable is %e\n",
      sizeDec, dist, acceptableDistance);
```

 

```
    sizeDec = sizeDec + 1;
  end
  else
  begin
    $display("sizeInt = %d\n sizeDec = %d\n
lead to a distance of %e <= acceptable dis-
tance of %e", sizeInt,sizeDec, dist,
             acceptableDistance);
     notDone = 0;
  end
end
end
end
endmodule
```

 

# 7.4 Performing Fft and Ifft transforms

```
module top;
parameter SIZE = 1024 * 1024;
integer k;
real delta;
VpFCartesian xformFC [0:SIZE - 1];
initial begin
#1;
$InitM(xformFC, (($I1==3) ? 7.0 : 0.0), 0.0);
$display("xformFC[3].Re=%e\n",xformFC[3].Re);
for (k = 0; k < 1; k = k + 1)
begin
  $VpFft(xformFC, 0, SIZE-1);
  $VpIfft(xformFC, 0, SIZE-1);
end
/*$PrintM(xformFC, "%e");*/
$display("xformFC[3].Re=%e\n",xformFC[3].Re);
end
endmodule
```

 

# 7.5 Partitioning for Multi-threaded processing

Example of using the *View as* construct in order to write code that is independent of the actual location of the data, within a multi-dimensional array. One application is the coding of multi-threaded video processing, where the code ought to remain unchanged when the number of partions change.

 

```
module top;
parameter W = 2;
parameter SIZE = 8;

/* main data */
real Orig[SIZE-1:0][SIZE-1:0];

/* copied partition of main data */
real M3[SIZE/2+1:0][SIZE/2+1:0];

/* sliding windows into the copied partitions
   enable writing code that is independent of the
   actual location of the data                  */
view real VM3[W:0][W:0] as
  M3[VM3_base1+$I1][VM3_base2+$I2];
integer VM3_base1, VM3_base2;

/* view for writing data back into Orig */
view real V3M[SIZE/2-1:0][SIZE/2-1:0] as
  Orig[M3_base1+$I1][M3_base3+$I2];
integer M3_base1, M3_base3;

initial begin
  $InitM(Orig, ($I1*10000+$I2));
end

/* example of processing partition M3, using the
   sliding window VM3                           */
initial begin
#20;
  /* copy from the appropriate partition in Orig
into M3. Set to 0 data located out of the range
of the original matrix. */
$InitM(M3, (($I1 == SIZE/2+1) ||
```

 

```
            ($I2 == SIZE/2+1))? 0 :
        Orig[SIZE/2-1+$I1][SIZE/2-1+$I2]);
  $InitM(M3, Orig[M3_base1+$I1][M3_base2+$I2]);
  $PrintM(M3, "%e");

/* Set the base of the sliding window  */
  VM3_base1 = 2;
  VM3_base2 = 2;

/* modify data in M3 */
VM3[W][W] = 99.0;

/* set the base of V3M for writing into Orig */
  M3_base1 = SIZE/2;
  M3_base3 = SIZE/2;

/* write into Orig via the view V3M */
$InitM(V3M, M3[$I1+1][$I2+1]);

$PrintM(Orig, "%e");
 end
endmodule
```

 

# 7.6 Finding the inverse of a matrix

```
module top;
parameter SIZE = 16;

real AR[SIZE-1:0][SIZE-1:0];
integer mone;

initial begin
  /* populate AR into a Pascal Matrix */
  $InitM(AR, (($I1 == 0) ? 1 :
        (($I2 == 0) ? 1 : (AR[$I1-1][$I2] +
                           AR[$I1][$I2-1]))));
  $PrintM(AR, "%e");

  /*compute the inverse twice */
  AR = AR**(-1);
  $PrintM(AR, "%e");
  AR = AR**(-1);
  $PrintM(AR, "%e");
end
endmodule
```

 

# 7.7 Finding the inverse of a large matrix of type real

This example works on FinSim 10_0_6 and subsequent versions.

This example shows how to invert  matrices of 4000x4000 elements of type real. The matrix is inverted twice and three values are checked to see that they remained unchanged after the two inversions.

```
module top;

parameter SIZE = 4000;

real AR[0:SIZE-1][SIZE-1:0];

real IR[SIZE-1:0][SIZE-1:0];

real r;

initial begin

  $InitM(AR, ($I1 == $I2) ? 1 : ($I1 == 2*$I2)
? 7 : 0);

   IR = AR**(-1);

  IR = IR ** (-1);

  $display("IR[%d][%d]= %e\n",16,8,IR[16][8]);

  $display("IR[%d][%d]= %e\n",16,9,IR[16][9]);

  $display("IR[%d][%d]= %e\n", 200, 100,
IR[200][100]);

end
endmodule
```

 

# 7.8 Finding the pseudo inverse of a matrix

```
module top;

real Q[3:0][2:0];
real S[3:0][0:0];
real P[2:0][0:0];

  initial begin
    Q = {1.0, 1.0, 1.0,
         1.0, 2.0, 1.0,
         1.0, 1.0, 2.0,
         2.0, 1.0, 1.0};
    S = {6.0, 8.0, 9.0, 7.0};

    $PrintM(Q, "%e");
    $PrintM(S, "%e");

    P = S/Q;

    $PrintM(P,"%e");
  end
endmodule
```

 

# 7.9 Checking Special Conditions

```
module top;
`include "finsimmath.h"
VpReg [0:511] in1;
VpReg [0:511] in2;
VpReg [0:511] out;
VpDescriptor d1, d2;

initial begin
$VpSetDescriptorInfo(d1, 150, 96,
`TWOS_COMPLEMENT,
`TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
                    `SATURATION, 1);
$VpSetDescriptorInfo(d2, 20, 10,
`TWOS_COMPLEMENT,
`TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF+
   `WARNING,
`SATURATION+`WARNING, 1);

$VpSetDefaultOptions(256, 96,
`TWOS_COMPLEMENT,
`TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
`SATURATION, 1);

$VpAssocDescrToData(in1, d1);
$VpAssocDescrToData(in2, d1);
$VpAssocDescrToData(out, d2);

#10;
```

 

```
/*overflow at assignment to smaller integer
part */
  in1 = 2323;
  in2 = in1 **10;
  $display("in2 = %k\n", in2);
  out = in2;
  $display("out = %k\n", out);

#10;
  /* underflow at assignment to smaller frac-
tional size */
  in2 = 0.00000000000001;
  $display("in2 = %k\n", in2);

  out = in2;
  $display("out = %k\n", out);
end

always @(out_Overflow)
begin
   $display($time,,"out: Overflow = %d\n",
out_Overflow);
end

always @(out_Underflow)
begin
   $display($time,,"out: Underflow = %d\n",
out_Underflow);
end

always @(out_PeakNrOfIntBitsUsed)
begin
```

 

```
   $display($time,,"out: PeakNrOfIntBitsUsed =
%d\n", out_PeakNrOfIntBitsUsed);
end

always @(out_NrOfDecBitsLost)
begin
   $display($time,,"out: NrOfDecBitsLost =
%d\n", out_NrOfDecBitsLost);
end

endmodule
```

 

# 7.10 Fast Autocorrelation

This example shows how to perform fast autocorrelation on two vectors of type VpFComplex. Objects of this type are complex numbers in cartesian coordinates, with fields of type real.

```
module top;
  parameter SIZE = 1024;
  VpFCartesian t1[2*SIZE -1:0],
               t2[2*SIZE -1:0],
             prod[2*SIZE -1:0];
  integer j;
  initial begin
    #1;
    $InitM(t1,
           ($I1 < SIZE-1) ? 0 :
                            $I1-SIZE+1, 0);
    $PrintM(t1, "%e");
    $InitM(t2,
           ($I1 < SIZE) ? 0 : 2*SIZE-$I1, 0);
    $PrintM(t2, "%e");
    $VpFft(t1, 0, 2*SIZE-1);
    $VpFft(t2, 0, 2*SIZE-1);
    for (j = 0; j < 2*SIZE; j = j + 1) begin
      prod[j] = t1[j] * t2[j];
    end
    $VpIfft(prod, 0, 2*SIZE-1);
    $PrintM(prod, "%e");
  end
endmodule
```

 

# 7.11 Inverting a 4,000,000 by 4,000,000 sparse matrix in Fin-SimMath

This example works on FinSim 10_05_33 and subsequent versions.

This example shows how to invert sparse matrices of 4000000x4000000 elements of type real.

The matrix is inverted twice and all non-zero values on one line and one column are displayed to see that they remained unchanged after the two inversions.

On a single 32 bit Pentium 1800MHz processor this example run in less than 100 seconds. Note that this is the simplest matrix to invert. Nevertheless, this example shows that large matrices can be handled.

Also note that lines and columns of sparse matrices can be displayed with system tasks $PrintLine and $PrintCol, as well using system tasks `SpReadNextNzElemInLine` and `SpReadNext-NzElemInCol`.

```
module top;

parameter integer size = 4000000;

real MReal1 [size-1 : 0][size-1 : 0];

real MRInv [size-1 : 0][size-1 : 0];

integer found, lin, col, idx;

integer i;

real r;
```

 

```
initial begin
  /* declaring sparse matrices */
  $ToSparse(MReal1);
  $ToSparse(MRInv);

  /* initializing matrice */
  for (i = 0; i < size; i++)
    begin
      MReal1[i][i] = 1;
       if ((2*i < size) && (i != 0)) begin
          MReal1[2*i][i] = 7.0;
       end
    end
  /*inverting twice */
  MRInv = MReal1 **(-1);
  MRInv = MRInv ** (-1);

  $display("********displaying all non-zero
values on one line**********\n");
$PrintLine(MRInv, 4*size/10);
  $display("********displaying all non-zero
values on one line one element at a
time**********\n");
```

 

```
  idx = -1;
  found = $SpReadNextNzElemInLine(MRInv,
4*size/10, col, idx, r);
  while (found) begin
    $display("MRInv[%d][%d]=%e\n", 4*size/10,
col, r);
    found = $SpReadNextNzElemInLine(MRInv,
4*size/10, col, idx, r);
  end

  $display("********displaying all non-zero
values on one column*********\n");
$PrintCol(MRInv, 2*size/10);
  $display("********displaying all non-zero
values on one column one element at a
time***********\n");
  col = 2*size/10;
  lin = -1;
  found = $SpReadNextNzElemInCol(MRInv, lin,
col, r);
  while (found) begin
    $display("MRInv[%d][%d]=%e\n", lin, col,
r);
    found = $SpReadNextNzElemInCol(MRInv, lin,
```

 

```
col, r);
  end
  $display("********displaying norms
*********\n");
  max = $VpNormAbsMax(MRInv);
  sum = $VpNormAbsSum(MRInv);
  $display("max=%e, sum=%e\n", max, sum);
end
endmodule
```

 

# 7.12 Solving a non-linear differential equation in FinSimMath

This example shows how a differential equation with variable coefficients can be solved in extended in Verilog. The equation models the force of a tennis ball hitting a wall by using extensively the work presented by S.J. Haake, M.J. Carre and S.R. Goodwill at the University of Shefield, Dept. of Mech. Eng.

This example works on FinSim 10_01_31 and subsequent versions.

This example models the force of a tennis ball hitting a wall. The main model is that of a spring-mass system governed by the formula: m*y(2) + c*y(1) + k*y = 0, with y = 0 and y(1) = speed of ball hitting the wall. There are three forces that push against the wall:

1) the spring force, spring_force is due to the ball being compressed, with spring_force = k*y.

2) the damper force, damper_force is due to the viscuosity of the ball with damper_force = c*y(1)

3) the flux force, flux_force is due to the portion of the ball that loses it's speed during a small time slice. This flux_force is m0 * speed / delta_time, where m0 = m*s1/s2, where s1 is approx. the surface of a cillinder having as base the flattened portion of the ball and as hight the current speed y0 * delta_time and s2 is the surface of the ball, which makes

flux_force = y0*y0*m* $VpSqrt(2*r*x0-x0*x0)/(2*r*r), where y0 is the speed at a particular time, m is the total mass of

 

the ball, r is the radious of the ball, and x0 is the position of the center of the ball with respect to its original position when the impact began.

The algorithm solves the problem for speeds of 10m/s and 30m/s in a loop. In an inner loop the algorithm adjusts the non-linear coefficients and solves the equation for a number of steps with constant coeficients. It then performs again the same steps for the new value of the loop variable (named slice), which automatically adjusts which portion of the data is provided to the differential eq. solver via the "view..as" mechanism. Note that by using the "view..as" mechanism no data transfer is needed during the computation, the data being computed "in place".

The results are displayed via the $PrintM task as well as via the $VpPtPlot task.

```
module top;

parameter real alpha = 1.65;

parameter real r_total_time = 0.005;/*
seconds */

parameter nr_pts_per_ct_coef = 10;

parameter nr_slices_per_sec = 10000;

parameter nr_pts_per_sec =
nr_pts_per_ct_coef * nr_slices_per_sec;

parameter integer Size =
r_total_time*nr_pts_per_sec;

parameter real r_size = Size;
```

 

```
parameter real h = 2*r_total_time/r_size;/
* set double of sampling period */

parameter real m = 0.057;/* kg */

parameter real A = 16000000;/* N/m**2 */

parameter real k0= 21000; /*N/m*/

parameter real B = 3500; /* Ns/m */

parameter real r = 0.032; /*m (radius of
ball)*/

parameter order = 2;

parameter nrEq = 1;

localparam integer nr_slices =
nr_slices_per_sec * r_total_time;


real Fe[0:0][0 : Size], x[0:0][0 : Size],
y[0:0][0 : Size];

reg [0 : 3199] ressymb[0 : 0];

real coef[0 : 0][0 : 2];


view real x_ct [0:0][0:nr_pts_per_ct_coef]
    as x[0][$I2+slice*nr_pts_per_ct_coef];

view real Fe_ct
[0:0][0:nr_pts_per_ct_coef]
    as
```

 

```
Fe[0][$I2+slice*nr_pts_per_ct_coef];
view real y_ct [0:0][0:nr_pts_per_ct_coef]
    as y[0][$I2+slice*nr_pts_per_ct_coef];

real ar_f_spring [0:nr_slices];
real ar_f_damper [0:nr_slices];
real ar_f_flux [0:nr_slices];
real ar_f_total [0:2][0:nr_slices];

integer i, j, isZero, total_time;
integer slice = 0;
real k, c, x0, y0, v0;

initial begin
  #1;
  coef[0][0] = m;
  $InitM(Fe, 0.0);
  for (j = 0; j < 2; j = j + 1)
  begin
    /* find total force for two speeds: v0
= 10m/s and v0 = 30m/s */
    v0 = (j == 0) ? 10 : 30;
```

 

```
    x[0][0] = 0;
    y[0][0] = v0;
    isZero = 0;/* kludge for nulifying
results for when the model does not apply*/
    for (slice = 0; slice <= nr_slices;
slice = slice + 1)
    begin
      x0 = (x_ct[0][0] > 0) ? x_ct[0][0] :
-x_ct[0][0];
      y0 = y_ct[0][0];
      c = 4.0*B*x0*(2.0*r-x0);
      k = (slice < 2) ? 120000 : (k0 +
A*(x0**alpha));

      coef[0][1] = c;
      coef[0][2] = k;
      ar_f_spring[slice]= x0*k;
      ar_f_damper[slice] = y0*c;
      if ((y0 > 0) && (x0 > 0)) begin
           ar_f_flux[slice] = y0*y0*m*
                              $VpSqrt(2*r*x0-
x0*x0)/(2*r*r);
      end
```

 

```
      else  ar_f_flux[slice] = 0;

      ar_f_total[j][slice] =
ar_f_spring[slice] +

ar_f_damper[slice] +

ar_f_flux[slice];
      ar_f_total[j][slice] = (isZero) ? 0:
((ar_f_total[j][slice]>0)?ar_f_total[j][sl
ice]:0);
      if ((slice != 0) &&
(ar_f_total[j][slice] < 15
           /* after this value the spring
model no longer applies*/ )) begin
        isZero = 1;
      end
      /* call dif eq solver. The array x
will contain the solution and the
       array y will contain the first
derivative of x.
       Note that the initial conditions
have been already placed in x[0]
       and y[0].
```

 

```
        */
       $VpLODE(order, nrEq, h,
nr_pts_per_ct_coef+1,
                x_ct, coef, Fe_ct, y_ct,
ressymb);
    end
    ar_f_spring[nr_slices]=
x_ct[0][nr_slices]*k;
    ar_f_damper[nr_slices] =
y_ct[0][nr_slices]*c;

    ar_f_total[j][nr_slices] =
ar_f_spring[nr_slices] +

ar_f_damper[nr_slices] +

ar_f_flux[nr_slices];
    ar_f_total[j][nr_slices] =
(ar_f_total[j][nr_slices]>0)?ar_f_total[j]
[nr_slices]:0;
  end
  $PrintM(ar_f_total,"%e");
  total_time = r_total_time*1000;
  $VpPtPlot("standalonePlotMLSample.txt",
```

 

```
2, h,
            "Tennisball (0.057kg, 0.032m)
Force pushing the wall)", total_time,
            "Time (ms)", "Total Force(N)",
0, nr_slices, ar_f_total,
            "10m/s", "30m/s");
 end
endmodule
```

 

# 7.13 Mixed Symbolic and Numeric Computations

This example works on FinSim 10_05_28 and subsequent versions. It shows symbolic expression evaluation, differentiation, integration, and Laplace Transform.

```
module top;
`include "finsimmath.h"
   VpReg [0:200]r;
   VpReg [0:200]val_vp;
   VpDescriptor d1;
   real val;
   reg [0:30000] symbExpr1;
   reg [0:30000] symbExpr2;
   real x;

initial begin
   $VpSetDescriptorInfo(d1, 5, 7, `FLOATING,

`TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
                          `SATURATION+`WARNING,
1);
   $VpSetDefaultOptions(5, 7, `FLOATING,
```

 

```
`TO_NEAREST_INTEGER_IF_TIE_TO_MINUS_INF,
                          `SATURATION+`WARNING,
1);

   $VpAssocDescrToData(r, d1);
   $VpAssocDescrToData(val_vp, d1);

   r = $VpGetPi()/6;
   x = $VpGetPi()/6;

   symbExpr1 = "$VpSin(r*x)";
   $Eval(symbExpr1, val);
   $display("$Eval(%0s) = %e, for x = %e, r =
%y\n", symbExpr1, val, x, r);

   symbExpr2 = $Dif(3, symbExpr1, "x");
   $Eval(symbExpr2, val_vp);
   $display("*** SDif(SDif(SDif(%0s))) is %0s
and its value for x = %e is %y\n",
            symbExpr1, symbExpr2, x, val_vp);

   symbExpr2 = $Int(1, symbExpr1, "x");
   $Eval(symbExpr2, val_vp);
```

 

```
   $display("$SInt(%0s) = %0s and its value
for x = %e is = %y\n",
            symbExpr1, symbExpr2, x, val_vp);

   symbExpr2 = $Lap(1, symbExpr1, "x");
   $display("$LaplaceT(%0s) = %0s\n",
symbExpr1, symbExpr2);

   symbExpr1 = "x**4 * $VpSin(x)";
   symbExpr2 = $Lap(1,symbExpr1, "x");
   $display("*** Lap(%0s) is %0s \n",
symbExpr1, symbExpr2);
end
endmodule // top
```

 

# 8. Concluding Remarks

FinSimMath is an extension of the Verilog IEEE 1364 language having, as described in chapter 8 of FinSim's User's Guide available at www.fintronic.com (click on Support, FAQ, download FinSim's Users Guide).

In this era of globalization, when teams from different parts of the world co-operate on the same project it is more cost effective to have the ESL,  RTL and Gate level descriptions done in the same environment and even in the same language.

Modeling adaptive systems that gracefully degrade is possible only with support for dynamic format changes. True ESL design space exploration mandates runtime changes of formats and size of format fields.

FinSimMath supports the modeling at the mathematical level (differential equations, matricial calculus, FFT/IFFT, autocorrelation, mixed symbolic and numeric computations, etc.) both of the circuit to be designed and of the environments in which the models of such circuits must be verified.

FinSim already supports a large subset of FinSimMath and Fintronic USA intends to provide FinSimMath support also in conjunction with other standard compliant Verilog/SystemVerilog simulators.

 

 