

Tutorial on FinSimMath(TM)

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

by Alec Stanculescu, PhD
Fintronic USA, Inc.

Foster City CA - October 30, 2008

FinSimMath(TM)

an extension of Verilog(R) for Mathematical Descriptions

OUTLINE

1. Introduction
2. Overview of FinSimMath
3. Basic FinSimMath
4. Hierarchical Expressions Evaluation
5. Cartesian and Polar Types
6. Operations on Multi-dimensional Arrays
7. Practical Exercises Written in FinSimMath
8. Concluding Remarks

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 supports a large number of mathematical system tasks, and provides access to information regarding the occurrence of overflow, underflow, maximum number of bits needed, and cumulative 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 precision 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 macros 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

```
$VpSetDefaultOptions (field1, 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

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

Assignments to objects of type integer or reg result in the object on the left hand side containing the same bit pattern 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.

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 without copying the data, as in the example 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.

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 vpadadd(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  $0 \leq j < SIZE$  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 operate in best conditions.

The values are chosen to be zero in this case. Other values may be better in other circumstances.

```
*****/
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 coefficients of Butterworth IIR LP filter 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 specified coefficients 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 acctableDistance \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
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 coefficients of Butterworth IIR LP filter
with passband: 0-500 Hz (assuming a sample
rate of 8000 samples /sec) effective order= 5
*****/
$display("Loading coefficients\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 operate
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 coefficients 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);
end

```

```
        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 partitions 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 sparse matrix

```

module top;
parameter SIZE = 2000;

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

initial begin

    $InitM(AR, ($I1 == $I2) ? 1 :
            ($I1 == 2*$I2) ? 7 : 0);
    /*invert it twice and compare */
    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 = 0.0000000000000001;
    $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
```

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.

True ESL design space exploration mandates runtime changes of formats and size of format fields.

Modeling adaptive systems that gracefully degrade is possible only with support for dynamic format changes.

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.