FinSimMath,
as a model for HDLMath

IEC TC 91

October 21, 2015

Dr. Alec G. Stanculescu, Fintronic USA, Inc.

Contents

1. Introduction

This document begins by describing the requirements of HDLMath, a language for helping the design of ASIC or FPGA circuits implementing mathematical algorithms.

Next, this document describes FinSimMath, which currently is the only language that meets the requirements of HDLMath.

Next, this document describes examples of using FinSimMath.

This document contains portions of the FinSimMath documentation presented on www.fintronic.com. The IEC received permission from Fintronic USA, Inc. to copy and distribute this technical report which includes excerpts from Fintronic's documentation on FinSimMath,

2. Requirements of HDLMath

2.1. Support both Verilog and the description of mathematical algorithms.

The rationale for supporting Verilog is that it is the most used language for designing ASICs and FPGAs and it is a dual logo standard IEC/IEEE.

The rationale for supporting also the description of mathematical algorithms at a very high level is that a language intended to help with the implementation of mathematical algorithms into ASIC and FPGA circuits must support both levels of abstraction, i.e. the algorithms and their implementation in order to create a huge productivity increase compared to the case in which each level of abstraction is supported by a different language.

This is similar to the productivity increase introduced by Verilog when it supported for the first time both the gate and the RTL levels of abstraction, which allowed designers to design at the RTL level and to implement at the gate level. In a similar fashion, HDLMath will facilitate the design at the mathematical level and the implementation at the Verilog level.

2.2. Data Containers that must be Supported

2.2.1. Introduction

In this document the term “data container” refers to an object that can store numeric information. A data container has additional information associated to it which is provided at compile time and at run time in order to store and retrieve numerical values.

2.2.2. Data containers supported by Verilog

HDLMath shall support all data containers supported by Verilog. The rationale for this requirement is that HDLMath must support Verilog.

2.2.3. Variable precision data containers

Data containers with modifiable formats, size of fields, and rounding options are referred to as variable precision data containers.

These data containers must include at least (1) scalar containers named hereafter VpReg, (2) complex numbers in Cartesian co-ordinates, and (3) complex numbers in Polar coordinates.

Note that the high level support must be bit-accurate, i.e. the result of computations performed during simulation shall match the result produced by the actual hardware and not a result obtained by assuming that infinite resources are available.

The formats (floating or fixed point) of high level data containers, as well as the number of bits of their respective fields must be modifiable during the execution of the simulation.

The rationale for this requirement is that modifying formats and their respective fields during simulation allows for a more efficient design space exploration.

Note that this capability is protected by a US patent nr. 7,930,690 B1 and its owner, Alec Stanculescu, signed the IEC Patent Letter in which he expresses his intention to give reasonable terms for the use of the patent. This letter meets the IEC requirement for patents related to IEC standards.

2.3. Cartesian and Polar data containers

Cartesian and Polar data containers are pairs of numerical values having the meaning of Cartesian and Polar numbers, respectively. Such data containers must be supported both as pairs of Verilog real numbers and as pairs of variable precision data containers.

2.4. Exception handling (overflow, underflow).

Overflow and underflow exception handling must be supported.

The rationale behind this requirement is to help the design of exception handling, by providing mathematical level simulations that support exception handling.

2.5. Tracking of cumulative errors and peak number of bits used.

The rationale behind this requirement is to help the optimization of the implementation of mathematical algorithms by minimizing the errors and by using only the necessary number of bits that store data.

2.6. Support one and two-dimensional arrays

Support one and two-dimensional arrays of any kind of data container, including sparse arrays, as well as arithmetic and logical operations of any legal combination of data containers and/or arrays of data containers.

The rationale behind this requirement is that this capability allows the implementation of all mathematical algorithms, albeit not at the highest level, whereby high level system functions such as fft and cosine are described at a lower level only in terms of arithmetic operations.

2.6.1. Populating one and two dimensional arrays.

The rationale behind this requirement is that large arrays are difficult to populate by providing all the data by hand or from a file. It is sometimes very useful to have the data generated automatically in a declarative form.

2.6.2. Printing multiple values stored in one and two dimensional arrays.

The rationale behind this requirement is that it is useful to print two dimensional arrays without having to write two embedded for loops.

2.6.3. Accessing contiguously data that is stored non-contiguously.

The rationale behind this requirement is provide data to hardware subroutines without actually having to move the data into contiguous registers. The implementation of this construct would use multiplexors in order to bring the appropriate data at the appropriate place.

2.6.4. Displaying graphically values stored in one and two dimensional arrays.

The rationale behind this requirement is to help with the presentation of computation results.

2.7. Support for Sparse Arrrays and Matrices

The rationale behind this requirement is that in some cases arrays and matrices contain numerous null elements and processing them having this knowledge can be much more efficient.

2.8. Support Scalar, Cartesian and Polar Polynomials

The rationale behind this requirement is to support operations between polynomials at the highest mathematical level, i.e. using arithmetic operators. Polynomials must be supported with Scalar, Cartesian and Polar coefficients in both variable precision and real data containers.

2.9. Support bit level interfaces of high level mathematical descriptions.

The rationale behind this requirement is to make it easy to exchange high level descriptions with their low level implementation for fast simulations and get support from existing waveform viewers.

2.10. Support mixed numerical and symbolic computations.

The rationale behind this requirement is that all the necessary processing should be done in one execution and that the user shall not be burdened with passing data from one environment to another, such as from a symbolic environment to a numeric environment.

The implementation can involve performing symbolic execution using strings and at any moment evaluate a string in the current context of the simulation, as if the string would have been an expression in the numeric environment.

For an example, look at www.fintronic.com/eval_dif_fin_lap.html. During a simulation a symbolic expression is numerically evaluated based on the current values of its variables.

2.11. Support mathematical system functions and tasks.

There must be support for a large number of system functions, such as FFT, DFT, finding eigenvalues and eigenvectors, norms and distances, finding roots of polynomials.

The rationale for this requirement is that although all mathematical functionality can be written using the arithmetic operators available in FinSimMath such implementation would be approx. 10 times slower than the execution of the code generated by a C compiler out of a C code description, which is what is behind the calls to system functions and system tasks. In addition, providing such high level functions unburdens the designer from writing them.

2.12. Support User-defined tasks and function in pure C-code

Support for extending simulation functionality by having the capability to incorporate user defined C code execution in the simulation in a standard manner.

The rationale behind this requirement is that many design teams have their own mathematical libraries and nothing else can work as well for them. In such cases, such designers can use their own libraries inside the HDLMath environment.

Note that the C code implementing user defined tasks and functions callable from HDLMath code must have access to all HDLMath data containers.

2.13. Support information necessary for synthesis.

The rationale behind this requirement is that there is information that is needed for synthesis which is not necessary for simulation and which must be presented separate in order not to burden the users of the simulation tools.

However, for documentation purposes it is best if the synthesis information is completely presented in a textual form, so that it can be easily maintained during all phases of the design cycle.

Synthesis information can be of several kinds and each kind must be supported. The various kinds of synthesis information are described below.

2.13.1. Resources available

Resources available along with their cost, latency, geometrical parameters, number of bits, etc.

2.13.2. Address

Address in memory (including memory block) of given variables.

2.13.3. Register Variables

Specification of variables to be implemented as registers or memory elements.

2.13.4. Connectivity Information

Topological information consists of declaration of ports of memory blocks, inputs and outputs of busses, and the way they are connected.

2.13.5. Clock rates

Specification of sampling rates for input data, clock rates for clocks supplied to various memory blocks, as well as to computational resources.

3. FinSimMath Language Constructs

3.1. Introduction

This section describes FinSimMath features that support the various HDLMath requirements discussed above and provide a model for similar HDLMath constructs.

The FinSimMath features that extend Verilog are limited to the following categories: (1) additional keywords, (2) additional data containers, (3) additional formats that can be used in $monitor and $display, (4) assignments that can be present in initial and always blocks can be made to new kinds of data containers, (5) expressions involving the additional data containers, (6) additional system tasks and functions, and (7) implicit registers associated with variable precision data containers.

Due to the fact that Verilog is extended in such minor aspects, FinSimMath is an extension of several versions of Verilog, starting with Verilog 1995 (which is today supported by Verilog XL, and continuing with Verilog 2001, Verilog 2005, and System Verilog. This is a very important aspect because there are many more users of Verilog than users of System Verilog for example, as companies such as Xilinx provide only Verilog simulators to their customers and not System Verilog.

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. In addition FinSimMath supports user-defined C/C++ functions, thus allowing the user to utilize preferred mathematical functions inside the high level simulation environment associated to FinSimMath.

FinSimMath supports the types VpReg (for variable precision objects), VpComplex, VpPolar, VpFComplex, VpFPolar, types. Logical, Arithmetic and assignment operators are defined to operate on all combination of these types including on arrays and matrixes.

FinSimMath supports the type VpDescriptor, which is used to specify the format, size of fields, rounding and underflow and overflow options.

FinSimMath supports the types used to declare polynomials: VpPol, RealPol, FCartesianPol, FPolarPol, CartesianPol, and PolarPol. The above mentioned polynomials are treated as one dimensional arrays of data containers VpReg, real, FCartesian, FPolar, Cartesian, and Polar respectivel for all operations except in case both operands of an arithmetic expression are defined as polynomials, case in which the operation is considered an operation between polynomials.

Objects of variable precision types VpReg, VpComplex, 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 separate data from its location.

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, $Fft, $Ifft, $Dct, $Idct, etc.

3.2. Lexical Conventions

3.2.1. Introduction

Lexical conventions follow the lexical conventions of Verilog. The few additions are as follows:

i) Several additional keywords

ii) Implicit registers associated to registers declared as VpReg are automatically created. Their names are the concatenation of the name of the VpReg followed by underscore and followed by the name of the implicit register. See section on implicit registers.

3.2.2. Additional Keywords

The additional keywords are:

VpReg, VpDescriptor, VpCartesian, VpPolar, VpFCartesian, VpFPolar, VpPol, RealPol, FCartesianPol, FPolarPol, CartesianPol, PolarPol, view, as.

3.3. Data Containers

3.3.1. Introduction

Data Containers are objects that can store data in various formats. Such data containers include (1) the Verilog data containers, which are all of fixed number of bits, (2) some additional fixed precision data containers, and (3) variable precision data containers.

3.3.2. Verilog Data Containers

FinSimMath supports all data containers supported by Verilog, e.g. real, integer, wire, nets, and reg. These data containers have a value given either by the concatenation of their bits being interpreted as signed or unsigned integer, or for objects of type real, by the value assigned to them and the number of bits of their corresponding exponent and mantissa. The number of bits of the exponent and mantissa is implementation dependent and the user cannot modify it.

3.3.3. Non-Verilog Fixed Precision Data Containers

In addition to integer and real, FinSimMath supports complex numbers in Cartesian and Polar co-ordinates, VpFCartesian and VpFPolar respectively, where the two fields real/imaginary and magnitude/angle, respectively are represented by Verilog real, i.e. floating point, which in some implementation is a double floating point number with 52 bit mantissa and 11 bit exponent and 1 bit sign.

3.3.4. Variable Precision Data Containers

3.3.4.1. Introduction

Variable Precision Data Containers are data containers which have their formats and sizes of the associated fields modifiable during the execution of the simulation.

This section describes how rational numeric values are associated to registers declared as variable precision registers (referred hereafter as VP registers), and how those values are manipulated by a set of predefined functions, and overloaded operators in the Verilog language context.

FinSimMath supports variable-precision fixed-point and IEEE 754/854 radix 2 floating-point objects, functions, and math operators, using standard Verilog syntax, and custom Verilog semantic extensions. The predefined types VpReg and VpDescriptor are provided to declare VP registers and descriptors. The math operators +, -, *, **, and / can be applied to any combination of the following operands and results formats: arbitrary-precision fixed-point, arbitrary-precision floating-point, Verilog integer, Verilog real, Verilog register, and Verilog supported constants. Trigonometric and hyperbolic (direct and inverse) functions are supported for any precision. Power, logarithm, and square root operations are also available.

3.3.4.2. Values of VP registers

The values associated to VP registers are rational values of the form p/q where p is integer and q is an integer power of 2. The general form of the associated value is therefore:

$$p \cdot 2^{-k}$$

where both p and k are integers.

The value p is encoded using some of the bit values of the VP register. The encoding scheme for p is present in a descriptor that is associated to the VP register. That descriptor also contains all or part of the information about the value of the exponent k, whose value is in general given the difference between two terms k_fix and k_float. The value of k_fix depends only on information provided in the descriptor, it is not encoded in the bits of the VP register, and can be modified only by changing the descriptor. The value of k_float is encoded using the bits of the VP register and it is often changed during VP register handling.

If the descriptor contains all the information about the exponent k (meaning that k_float=0 at all times) the associated values are fixed point values, and the format is a fixed point format. Otherwise, if there is a field in the VP register which encodes k_float using the VP register bit values, the associated values are floating point values, and the format is a floating point format. Under special circumstances, some combination of bit values in the VP register represent special values that are not numeric values. Hereafter, when there is no possible confusion we will refer to the “VP register associated numeric value” as the “numeric value of the VP register”. A VP register can also be used as a regular Verilog register and assigned to registers and nets.

3.3.4.3. Specifying VP objects

VP registers contain values and have associated to them information regarding the format, number of bits used to by the various parts corresponding to the given format (e.g. exponent and mantissa), as well as information regarding rounding and overflow options.

The following four steps for providing information are required before using a VP register:

Step 1: Declare a VP descriptor

Step 2: Declare a VP register as data container

Step 3: Set the descriptor information

Step 4: Associate descriptor to data.

The only order constraints between the steps above are that step 4 should be performed after step 1 and step 2, and step 3 has to be performed after step 1 was performed during the execution of the simulation.

Examples of using Variable Precision Data Containers:

Step1: VpReg [0:511] in1;

Step2: VpDescriptor d1;

VpReg contain numerical values. The information regarding the format in which the numerical value is represented (i.e. the relation between the numerical value and the bit values of the VP register), as well as the information regarding the action to be taken in case overflow, or underflow occurs in an operation that assigns to the given VP register is stored in the descriptor that must be associated to any VP register.

Notes:

i) The size of the VpReg must be chosen such that during the entire simulation it exceeds the number of bits that are necessary to represent the VP register value.

ii) The descriptor has no size and multi-dimensional descriptors are not allowed.

A descriptor can be associated to any number of VP registers using the system task

Step3: $VpSetDescriptorInfo(<myVPdescriptor>,<size1>, <size2>, <format>, <rounding>, <overflow>, <misc>).

Step4: $VpAssociateDescriptorToData(myVPreg, myVPregDescriptor);

For each VP register there must be exactly one call associating to it a descriptor. This call must occur in the module in which the VP register is declared.

3.3.4.4. Setting the fields of the descriptor

The various fields of a descriptor are integers which can be can be modified at runtime any number of times using the system task $VpSetDescriptorInfo(<myVPdescriptor>,<size1>, <size2>, <format>, <rounding>, <overflow>, <misc>).

The format field can have the following values:

1 - indicates two's complement

2 - indicates sign magnitude

3 - indicates floating

4 - indicates floating with no denormals

In case the format is two's complement size1 and size2, if they are both positive, represent the number of bits of the integer part and the number of bits of the fractional part (referred to also as decimal part) respectively. It is illegal for both sizes to be negative. If one is negative the part to which it corresponds (integer or fractional) has zero bits representing it and the other part is represented by a number of bits equal to the sum of the absolute values of the two sizes, with the restriction that no information can be stored in the bits corresponding to the negative size which are located at the border to the other part (i.e. if the integer size is negative the most significant -size1 bits of the fractional part will not be used to store information even if an overflow must be reported. Similarly, in case size2 < 0 the least significant -size2 bits of the integer part will not contain any information even if an underflow must be reported.

In case the format is either floating or floating with no denormals the two sizes must be positive, with size1 representing the number of bits of the sign and the exponent and size2 representing the number of bits of the mantissa.

The rounding field can have the following values:

1 - indicates rounding to nearest integer, with approaching -infinity in case of a tie.

2- indicates rounding to nearest integer, with approaching +infinity in case of a tie.

3- indicates rounding to nearest integer, with approaching zero in case of a tie.

4 - indicates that a simple truncation will be performed

5 - indicates rounding to zero

6 - indicates rounding to +infinity for positive values and to -infinity for negative values

7 - indicates rounding to -infinity

8 - indicates rounding to +infinity

The overflow field can have the following values

1 - indicates saturation, i.e. in case of an overflow the value will keep the correct sign and the maximum possible value.

2 - indicates wrapping around, i.e. in case of an overflow the value will be the remainder of un-representable value divided by the maximum representable value plus one unit.

3.3.5. The Default Descriptor

The default descriptor contains the same information as any descriptor. There is no explicit default descriptor. The implicit default descriptor may have its various fields: size1, size2, format, rounding option, overflow option, underflow option set at runtime via the system task $VpSetDefaultDescriptorInfo.

The information stored in the default descriptor influences the values of the descriptors associated to temporary VP registers needed to evaluate complex expressions (e.g. involving more than one arithmetic operation).

3.3.6. Using variable precision and formats modifiable during simulation

A variable precision register has a descriptor associated to it. The values of the various fields of the descriptor can be modified during the simulation, making it possible to continue the simulation with a different format and/or different sizes of the fields of the format.

3.4. VP register handling

3.4.1. Assigning a constant to a VP register

Integer and real literal constants can be assigned to VP registers as in the examples below:

myVPreg = 23; myVPreg = 2.3; or myVPreg = 2.3e+0;

However, note that real literals are first converted to the Verilog real (which is usually the 64 bit double representation) and then converted to the format indicated by the descriptor. This may lead to a loss of information. In order to avoid any loss of precision, one can use the following:

myVPreg1 = 23;

myVpreg = myVPreg1 / 10;

The literal constant is transformed into a temporary VP register having a size such that as little data as possible is lost when placing the value of the temporary VP register into the left hand side VP register.

When the value of the temporary VP register is transferred into the left hand side of the assignment its underflow or overflow implicit signals may be set with the number of bits which if added to the mantissa/fractional part or the exponent/integer part respectively would prevent underflow or overflow from occurring.

3.4.2. Assigning a Verilog register to a VP register

The value of the non-VP register will be stored in the VP register, to the extent possible and any rounding will be taken care of according to the rounding option associated to the VP register. The overflow or underflow flags may be set as a result of such an assignment, similar to the case of assigning a constant to a VP register.

Note that in preserving the value of the right hand side into the left hand side one may have to change the bit pattern.

3.4.3. Assigning VP register to VP register

The value stored in the VP register on the right hand side of the assignment shall be transferred into the VP register on the left hand side.

If the number of bits of the mantissa or fractional part of the VP register on the left hand side are insufficient to store the value stored in the VP register on the right hand side then rounding shall occur according to the rounding option of the descriptor associated to the VP register on the left hand side

If the value stored in the VP register on the left hand side is zero and the value stored in the VP register of the right hand side is not zero then the underflow implicit register of the VP register on the lhs will be set to the number of bits which if added to the exponent or the fractional part of the VP register of the left hand side would prevent the underflow condition from occurring.

If the value stored in the VP register on the right hand side cannot be stored in the VP register on the left hand side because either the exponent or the integer part do not have enough bits then the overflow implicit register of the VP register on the lhs will be set to the number of bits which if added to the exponent or to the integer part of the VP register on the lhs would prevent the overflow condition from occurring.

3.4.4. Assigning a wire to a VP register

The number of bits of the wire must be at least as large as the number of bits necessary to represent any value in the format and sizes present in the descriptor of the VP register. The execution of the assignment will result in copying from the least significant portion of the wire into the least significant portion of VP register a number of n bits, where n is the sum of the two sizes present in the descriptor of the VP register, i.e. the number of bits necessary to represent any number in the format and with the sizes present in the descriptor of the VP register.

3.4.5. Assignments to non-VP objects

Any assignment of a VP register to a non-VP object will move the bit values representing the value of the VP object to the non VP object with the bits of the second part (fractional or mantissa depending on the format) being copied to the least significant part of the target. Any information related to the descriptor will not be passed to the non-VP object.

It is illegal to have the non-VP object declared with a size that is smaller than the necessary number of bits indicated by the descriptor of the VP object occurring on the right hand side.

Assignments in which VP registers are not referenced at all are governed by the rules of Verilog IEEE 1364-2001.

3.4.6. Variable precision polynomials

FinSimMath also supports data containers with variable precision for polynomials: VpPol, CartesianPol, PolarPol.

VpPol, CartesianPol and PolarPol are used to declare one-dimensional arrays of VpReg, Cartesian or Polar respectively, with the property that in case both operands of arithmetic operators +, -, *, / are declared as polynomials the operation undertaken is an operation between polynomials.

3.4.7. Arithmetic Operators operating on VP registers

3.4.7.1. Type of Operands

The type of operands may be: integer, reg, wire, VP register with two’s complement format, VP register with floating format, VP register with floating no denormals format.

3.4.7.2. Operators

List of binary arithmetic operators: +,-,*,/, **.

List of unary arithmetic operators: +,-

The operands are converted into VP registers if they are not VP registers already and then the operation is performed such that with the exception of division there is no loss of data in the result. In case of division only the n most significant bits of the fractional part are kept, where n

is number of bits of the fractional part of the final result plus three bits, which are used for rounding. The descriptor of the final result is obtained from the right hand side in case of expressions having only one operator or in a manner described later in this chapter for more complex expressions.

Once the operation is performed the value of the result is converted to the format and size of the final result.

The underflow implicit signal of the final result is set when the final result has the value zero while the result of the operation with as little loss of data as possible contained a non-zero value. The underflow signal, which is of type integer is set to the number of bits that if added to the fractional part or mantissa of the final result would have prevented the underflow condition from occurring.

The overflow implicit signal of the final result is set when the result of the operation has a value that cannot be stored in the final result because either the exponent (in case of a floating format of the final result) or the integer part (in case of a fixed point format of the final result) has an insufficient number of bits. The overflow implicit signal which is of type integer will be set to the number of bits which if added to the exponent of integer part would have prevented the condition for overflow from occurring.

Example of use:

myVPreg = myVPr1 + myVPr2;

myVPreg = myVPr1 / myVPr2;

3.4.7.3. Restrictions on the power operator (a**x)

In case the value returned by the power operator is to be stored in a scalar and not a polar or cartesian container then the following rules apply:

a) In case x < 0 a may only have a value of the form 1/2p+1, where p is an integer.

b) If a == 0 and x == 0 Super FinSim will arbitrarily report an overflow and will also produce a warning providing all the available information: file, line, values of x and a.

c) If x == 0 and a != 0 the result shall be 1

d) If x > 1 or x < -1 Overflow may be produced if a > 1 and x is large enough.

e) If x > 1 or x < -1 Underflow may be produced if a < -1 and -a is large enough.

Example using the power operator

myVPreg = x ** a;

3.4.8. Logical Operators involving VP registers

The type of operands may be: literal integer, literal real, integer, reg, wire, VP register.

The supported logical operators are: < (less than), > (greater than), <= (less or equal), >= (greater or equal), == (equal), != (not equal).

The expression returns a one bit which has the value of 1 in case the condition is met and returns 0 otherwise.

3.4.9. Displaying values of data containers

The $display and $monitor system tasks available in Verilog are extended to support the following additional formats:

%y: displays the value of VP registers with two's complement format with a decimal point separating the integer and fractional parts, e.g. 72.073, and VP registers with floating point formats with the same format as the display of Verilog reals, e.g. 2.5e-1 representing the same value as 0.25.

%k: displays the value of VP registers in binary format with the bits in the following order depending on the format indicated by the associated descriptor:

i) floating or floating without denormals: sign, exponent, mantissa, where sign is displayed as +/-, and exponent is separated from mantissa by a dot.

ii) two's complement: integer part, fractional part separated by a dot.

%p: displays the value of VP registers in hex format.

3.4.10. Cartesian and Polar Data Containers

Cartesian and Polar data containers can have their values as Verilog reals or as variable precision registers. The two fields of Polar data containers are .Mag and .Ang and the two fields of Cartesian data containers are .Re and .Im.

3.5. Arrays and matrices

3.5.1. Declaring arrays and matrices

The declaration of arrays and matrices follows either the Verilog syntax or for virtual arrays or matrices whose elements are not stored in the indicated structure but instead reside in other structures one can use the view-as construct.

3.5.2. Views of Arrays and Matrices

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.

3.5.3. Populating Arrays and Matrices

3.5.3.1. System Task $InitM(myMem, value)

One way to populate arrays and matrices is by 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:

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

VpPolar pMem[0:SIZE-1][0:SIZE-1];

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

$InitM(myMem, oMem[$I2][$I1]);

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

results in the two dimensional arrays myMem and myPMem containing the data of the transposed of the two dimensional arrays oMem and pMem, respectively.

3.5.3.2. System Task $Diag

Another way of populating a two dimensional array is by using system task $Diag(myMem, l, c, val). The two dimensional array will have the value 1.0 on the first diagonal and, in case both l and c are different from zero, the locations (i,j) inside the matrix having the property that l*I = c*j will have the value val.

3.5.4. Printing arrays and matrices

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. In addition to the formats supported by Verilog FinSimMath supports formats to be used for variable precision registers: %k similar to %b for Verilog registers, %p similar to %h for Verilog registers, and %y similar to %e for Verilog reals.

3.5.5. Displaying graphical information stored in matrices - $Flot

The system task $Flot produces a text file which can be displayed by Flot. In order for the $Flot system task to work under the FinSim simulator one has to download finfloat.tgz from Fintronic's ftp site and unzip it and untar it in the directory pointed by the environment variable FINTRONIC.

It can display several curves on the same image and supports zoom in and out. $Flot accepts the following arguments:

1) Name of the result file.

2) Nr of different curves to be plotted on one image.

3) The distance between the projections on the first dimension of two consecutive points.

4) Title to be displayed as the header of the image.

5) Label of the first dimension

6) Label of the second dimension

7) Two-dimensional array of values to be plotted. Each row represents one curve to be plotted.

8) The remaining arguments represent the labels of the different curves to be plotted. FinSim version 10_05_67 supports only up to eight different curves on one image.

An example of usage of $Flot is given below:

$Flot("test.html", 2, h/2, "Tennisball (0.057kg, 0.032m) Force pushing the wall)", "Time (ms)", "Total Force(N)", 0, nr_slices, ar_f_total, "10m/s", "30m/s");

3.6. Sparse arrays and matrices

As in Verilog, FinSimMath supports multi-dimensional arrays. Unlike in Verilog arithmetic operators can operate on various combinations of operands including multi-dimensional arrays. Any array can be considered as sparse fi there is call to $ToSparse having the given array as argument. Such a call uses a different implementation for reading and writing to the array which more effective in case many of the elements are zero, but which will be inefficient in case that most elements are non-zero.

3.7. Polynomials

3.7.1. Introduction

Polynomials are arrays of coefficients which can be of any kind of data container. These arrays are special in that if both operands of an arithmetic operations are declared as polynomials the arithmetic operation performed is the operation corresponding to polynomials.

The coefficients are stored from left to right in decreasing order of the corresponding power. The last coefficient is stored at the rightmost position of the array.

3.7.2. Declaration of Polynomials

Polynomials are declared using keywords indicating the type of array, i.e. RealPol, FCartesianPol, FPolarPol. VpPol, CartesianPol and PolarPol corresponding to the underlying data containers of the array, namely Real, VpFCartesian, VpFPolar, VpReg, VpCartesian, and VpPolar respectively.

3.7.3. Error reporting

3.7.3.1. In case the output of the polynomial operation does not fit in the space associated to the output data container an error is issued.

3.7.3.2. In case all coefficients of the divisor in a divide operation are null an error is issued.

3.8. Structural description at bit-accurate mathematical level

3.8.1. Structural descriptions

Structural descriptions remain strictly as in Verilog. Module instances are interconnected with Verilog constructs. Data in ports can be converted to variable precision data containers for processing within initial or always blocks using the system function $VpCopyReg2Vp described in the chapter System Tasks and Functions.

3.8.2. Bit-accurate models

Processing units such as adders and multipliers can be described at the mathematical level.By using data in variable precision data containers the results are bit-accurate. The bit-accurate results computed at the mathematical level can be assigned to the bit level output ports without any conversion function.

The data upon which to perform the mathematical level bit-accurate computations can be obtained from input ports using the system function $VpCopyReg2Vp. Note that the variable precision register where the result of $VpCopyRegToVp is stored must have an appropriate descriptor associated to it, i.e. with appropriate format, sizes of fields, rounding options, etc.

3.9. Behavioral Description

3.9.1. Introduction

The assignments that can be placed in an initial or always block, as well as the data containers on which such assignments can operate are being extended from their Verilog semantics.

3.9.2. Assignments

Both blocking and non-blocking assignments are extended from their Verilog definition.

Assignments consist of an expression that is placed on the right hand side of the assignment operator, which is being evaluated according to the rules of Verilog and the result placed in the data container indicated on the left hand side of the assignments operator.

The extension to Verilog consists in the kind of data containers that can be placed on the left and on the right side of the assignment operator.

In case the right hand side has the same number of scalar elements as the left hand side the assignment is performed element to element. In case the data containers are of the same kind and the same format (i.e. floating or fixed point) the assignment is performed as such, otherwise

an implicit conversion takes place in case the elements are of a different kind, e.g the left hand side has elements declared as VpCartesian and the right hand side has elements declared as VpPolar.

Even in case the elements are of the same kind, the format and the size of the various fields can differ. In such case an implicit conversion takes place at simulation time.

Note that in some cases the number of elements of the right side is determined by the dimensions of the left hand side. For example, in case the right hand side is a multiplication of one-dimensional arrays, if the left hand side is a one-dimensional array, then it must be of the same size as the two arrays on the right hand side and the multiplication is performed on corresponding elements. If however, the left hand side is just one scalar element then the multiplication is interpreted to mean the scalar product of the two arrays.

When a scalar is assigned to a Cartesian or Polar data container, it is assumed that the scalar is written to the first field (.Re or .Mag respectively) and that the value zero is assigned to the second field.

3.9.3. Expressions

All Verilog expressions are legal in FinSimMath. The extension consists in that operators can operate on operands having a wider variety of dimensions and place results in prescribed structures for each legal combination of operands.

In case the number of dimensions and size of each dimension for both operands are legal, and the left hand side has a number of dimensions and sizes of each dimension that are compatible with the two operands then the implicit conversion (Polar to Cartesian, Cartesian to Polar, as well as format and size related conversions) are performed before the placement of the data.

All Verilog operators are legal in FinSimMath. The operators that are extended are the four arithmetic operators and the power operator, i.e. +,-,*, /, and ** respectively.

The power operator applied to a matrix with value of the power being -1indicates matrix inversion or pseudo inversion, depending on the size of the two dimensions of the matrix.

Multiplication of two two-dimensional arrays necessitates that one of the following cases occur:

a) the arrays can have dimensions of the same size and the result is placed in a two dimensional array of the same sizes,

b) the arrays are of sizes n x m and m x n with the result being a two dimensional array of sizes n x n.

c) the arrays are one dimensional of the same size and the result is either a scalar (in case of the scalar product) or a one dimensional array of the same size as the operands (in case of multiplication element by element).

3.9.4. Implicit registers for exception handling

During the placement of data into variable precision elements as a result of an assignment some extra information is collected by the simulator: peak number of integer bits used, cumulative error, number of decimal bits lost, underflow and overflow. This extra information is stored in implicit registers associated to the VpReg declaration and can be referenced as name_PeakNrOfIntBitsUsed, name_CumulativeError, name_NrOfDecBitLost, name_Underflow , name_Overflow, where name is the name of the VpReg.

3.9.5. finsimmath.h include file

This file contains declarations that are common to most FinSimMath descriptions and must be included in many circumstances. In case it is needed and is missing the compiler will complain.

3.10. System Tasks and Functions

3.10.1. Introduction

System tasks and functions provide implementations of tasks and functions that are more efficient than the ones that can be written without using C code.

FinSimMath system tasks and functions begin with the characters $Vp. In the absence of any statement to the contrary the arguments of FinSimMath system tasks and functions can be of any kind of data container. In case FinSim's implementation of FinSimMath imposes any temporary restrictions, such restrictions shall be reported when such restricted usage is encountered.

3.10.2. Norms and Distances

Norms and Distances are system functions returning a real value having the property of being a norm or a distance, respectively.

3.10.2.1. $VpDistAbsMax(M1, M2)

M1 and M2 are matrices having the same number of elements.

This system function returns the maximum of the absolute values of the differences between all elements of the two matrices having the same indexes.

3.10.2.2. $VpDistAbsSum(M1, M2)

M1 and M2 are matrices having the same number of elements.

This system function returns the sum of the absolute values of the differences between all elements of the two matrices having the same indexes.

3.10.2.3. $VpNormAbsMax(M1)

This system function is a norm that returns the maximum absolute value of the all elements of the matrix M1.

3.10.2.4. $VpNormAbsSum(M1)

This system function is a norm that returns the sum of the absolute values of the all elements of the matrix M1.

3.10.2.5. $VpNormAbsRMS(M1)

This system function is a norm that returns the square root of the sum of the power of two of all elements of the matrix M1.

3.10.2.6. $VpAbs

This system function is a norm that returns the absolute value of its argument.

3.10.2.7. $VpHypot

This system function is a distance that returns the square root of the sum of the squares of the two arguments.

3.10.2.8. $VpFloor

This system function is a norm that returns the integer part of the value contained in the argument.

3.10.2.9. $VpCeil

This system function is a norm that returns the value of its argument in case it is an integer, the integer part of the argument plus one in case its argument is positive and the integer part minus one in case its argument is negative.

3.10.3. Support for sparse matrices

3.10.3.1. $ToSparse

This system task accepts as argument a one or two-dimensional array of any kind of supported data container.

The effect of the call is that the internal storage of the array will be modified to make it optimal for the case in which most of the individual data containers have the value zero. After the call the array shall be treated as a sparse array.

3.10.3.2. $SpReadNextNxElemInLine

This system function returns 1 in case it finds the next non-zero element on a given line of the sparse array. It uses the following arguments:

1) A two-dimensional sparse array,
2) An integer indicating the line,
3) An integer indicating the column. This argument is updated during the call to the next non-zero element on the given line.
4) An integer representing some internal information which is useful in optimizing the speed of retrieving the next non-zero element on the given column.
5) A container of the type of the containers in the sparse array which will contain the next non-zero element.

Note: for the first element on a given line the fourth argument must have the value -1, and the value of the third argument does not matter. After the call the third argument will contain the value of the column in which the non-zero element can be found in case one has been found.

3.10.3.3. $SpReadNextNzElemInCol

This system function returns 1 in case it finds the next non-zero element on a given column of the sparse array. It uses the following arguments:

1) A two-dimensional sparse array,

2) An integer indicating the line,
3) An integer indicating the column. This argument is updated during the call to the next non-zero element on the given line.
4) An integer representing some internal information which is useful in optimizing the speed of retrieving the next non-zero element on the given column.
5) A container of the type of the containers in the sparse array which will contain the next non-zero element.

Note: for the first element on a given line, the line must be initialized to the proper value and the fourth argument must have the value -1. The value of the third argument does not matter.

After the call, the third argument will contain the value of the column in which the non-zero element can be found in case it exists.

The example below shows how sparse matrices can be used in FinSimMath.

This example inverts a simple sparse matrice of 4,000,000 x 4,000,000 elements of type real twice, uses two norm system functions to measure it and displays all non-zero values on one line and one column.

```
module top;
  parameter integer size = 4,000,000;
  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, max, sum;

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

    /* initializing matrice to be inverted*/
    $Diag(MReal1, 2, 1, 7.0);

     /*inverting matrix */
     MRInv = MReal1 **(-1);
     MRInv = MRInv **(-1);
     lin = 4*size/10;
     $display("displaying all non-zero values on line %d\n", lin);
     idx = -1;
     found = $SpReadNextNzElemInLine(MRInv, lin, col, idx, r);
     while (found) begin
        $display("MRInv[%d][%d]=%e\n", lin, col, r);
        found = $SpReadNextNzElemInLine(MRInv, lin, col, idx, r);
      end

      col = 2*size/10;
      $display("displaying all non-zero values on column %d\n", col);
      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 and distances*********\n");
    max = $VpNormAbsMax(MRInv);
    sum = $VpNormAbsSum(MRInv);
    $display("max=%e, sum=%e\n", max, sum);
  end
endmodule // top
```

3.10.4. Conversion between Verilog and FinSimMath-specific data containers

3.10.4.1. Variable precision semantics of Verilog declarations

Data in net, wire, and reg are considered with their Verilog semantics when participating in an expression. In order to be considered with variable precision semantics they must be assigned to a variable precision register using the system function $VpCopyReg2Vp. The call to the system function $VpCopyRegToVp must be applied individually to each pair of scalar data containers.

3.10.4.2. Assignment to Verilog objects

The conversion from variable precision scalars to Verilog registers requires just a simple assignment and no system function call is required. Such assignments place in the Verilog data structure the bits as they codify the value in the particular register. Their meaning can be recovered only after they are assigned to a VpReg using a call to $VpCopyReg2Vp, in a context in which the same data of the original descriptor apply.

3.10.4.3. Copying bits of Verilog real to Verilog reg.

The conversion from Verilog real data containers into Verilog reg data containers is performed using system function $VpFCopyFl2Reg on each individual pair of scalar data containers involved.

3.10.4.4. Converting bits of a Verilog reg into a Verilog real

The conversion from Verilog reg to Verilog real data containers is performed using system function $VpFCopyReg2Fl on each individual pair of data containers involved.

3.10.4.5. $VpGetExp

Accepts as input a vp register in floating point format and returns the exponent into a normal Verilog register with sufficient bits.

3.10.4.6. $VpSetExp

Accepts as input a normal Verilog register, checks that the lhs is a vp register with floating point format and sets the value of the exponent of the lhs to the value of the input.

3.10.4.7. $VpGetMant

Accepts as input a vp register in floating point format and returns the mantissa into a normal Verilog register with sufficient bits.

3.10.4.8. $VpSetMant

Accepts as input a normal Verilog register, checks that the lhs is a vp register with floating point format and sets the value of the mantissa of the lhs to the value of the input.

3.10.5. Trigonometric and Hyperbolic functions

Function call	Input range	Output Range
$VpSin	(-inf : +inf)	[-1 : +1]
$VpCos	(-inf : +inf)	(-1 : +1]
$VpTan	(-inf : +inf)	(-inf:+inf)
$VpCtan	(-inf : +inf)	(-inf:+inf)
$VpAsin	(-1 : +1)	[-pi/2 : pi]/2]
$VpAcos	(-1 : +1)	0 : pi]
$VpAtan	(-inf : +inf)	[-pi/2 : pi/2]
$VpActan	(-inf : +inf)	[-pi/2:0) U (0:pi/2)
$VpSinh	(-inf : +inf)	(-inf : +inf)
$VpCosh	[1 : +inf)	[1 :+inf)
$VpTanh	(-inf : +inf)	(-1 : +1)
$VpCtanh	(-inf : 0) U (0 : +inf)	(-inf : -1) U (1 : +inf)
$VpAsinh	(-inf : +inf)	(-inf : +inf)
$VpAcosh	[1 : +inf)	[0 : +inf)
$VpAtanh	(-1 : +1)	(-inf : +inf)
$VpACtanh	(-inf : -1) U (1 : +inf)	(-inf : 0) U (0 : +inf)

3.10.6. Exponential and logarithmic functions

3.10.6.1. $VpLn

Returns the logarithm in base e (natural logarithm) in the format and precision of the lhs.

Example: $VpLn($VpGetE) = 1;

3.10.6.2. $VpExp

Returns e**x where x is the argument passed as input, with as much precision as it can be stored in the lhs.

Example: $VpExp($VpLn($VpGetE())) == $VpGetE();

3.10.6.3. $VpSqrt

Example: $VpSqrt(a*a) == a;

3.10.6.4. $VpLog

Returns the logarithm in base 10 of the input argument.

Example: $VpLog(100) == 2;

3.10.6.5. $VpPow

This function has two arguments a and x and returns a**x.

Example: $VpPow(-10000000, 1.0/7.0) == -10.0;

3.10.6.6. $VpPow2

Accepts one argument, a, and returns 2**a. It is more efficient than using 2**a.

3.10.7. Support for polynomials

3.10.7.1. $Roots

This system function accepts as argument an array of values representing the coefficients of a polynomial and returns the roots of the polynomial.

3.10.7.2. $Poly

This system function accepts as argument an array of values representing the roots of a polynomial and returns an array of values representing the coefficients of the polynomial having the roots provided as input.

3.10.7.3. $Poleval

This system function accepts as inputs an array of scalar values representing the coefficients of a polynomial and a value at which the polynomial shall be evaluated and returns the value of the polynomial.

3.10.8. Fourier Transforms

3.10.8.1. $VpFft, and $VpIfft

These tasks perform the Fast Fourier Transform and its inverse, respectively.

They each have three arguments.

The first argument is the name of a one dimensional array of reals upon which the transformation is performed in place.

The second argument is the first address within the one dimensional array.

The third argument is the number of consecutive elements that are used during the FFT transformation.

Note that the first argument may be a view declaration and hence the elements need not be actually consecutive in memory. They must be only consecutive in the object or the view being passed as argument.

Example of FFT using view:

```
real      myR[SIZE-1:0];
View real myR_even[SIZE/2-1:0]as myR[2*$I1];
$VpFft(myR_even,0, SIZE/2);
$PrintM(myR_even, “e”);
```

3.10.8.2. $VpDct, $VpIdct

These tasks perform the Digital Cosine Transform and its inverse, respectively.

The first argument is the name of a one dimensional array of reals upon which the transformation is performed in place.

The second argument is the first address within the one dimensional array.

The third argument is the number of consecutive elements that are used during the FFT transformation.

Note that the first argument may be a view declaration and hence the elements need not be actually consecutive in memory. They must be only consecutive in the object or the view being passed as argument.

3.10.9. Support for automatic control

3.10.9.1. $Rank

This function returns the rank of matrix provided as argument.

3.10.9.2. $Charpol

This function returns the characteristic polynomial of the matrix provided as argument.

3.10.9.3. $Eig

This function returns the eigenvalues corresponding to the first argument (a matrix) and possibly returns the eigenvectors in case a second argument is provided in which the eigenvectors will be stored.

This example below shows how to find the eigenvalues and eigenvectors of a matrix and how to test that indeed the eigenvectors corresponding to given eigenvalues are correct.

Example using $Eig.

```
module top;
`include
parameter size = 3;
 real p[0:size];
 VpFCartesian eval[0:size-1], evct[0:size-1][0:size-1];
 VpFCartesian A[0:size-1][0:size-1], I[0:size-1][0:size-1], LI[0:size-1][0:size-1];
 integer i, j, k, r;
```

```
 VpFCartesian B[0:size-1][0:size-1];
 VpFCartesian P[0:size-1][0:0];
 real norm;
 view VpFCartesian evctk[0:size-1][0:0] as evct[$I1][k];
 VpFCartesian l;

  initial begin
   /* initialize matrix A */
   A[0][0].Re = 1;
   A[0][0].Im = 1;
   A[0][1].Re = -1;
   A[0][1].Im = -1;
   A[0][2].Re = 2;
   A[0][2].Im = 2;

   A[1][0].Re = 0;
   A[1][0].Im = 0;
   A[1][1].Re = 0;
   A[1][1].Im = 1;
   A[1][2].Re = 2;
   A[1][2].Im = 0;

   A[2][0].Re = 0;
   A[2][0].Im = 0;
   A[2][1].Re = -1;
   A[2][1].Im = 0;
   A[2][2].Re = 3;
   A[2][2].Im = 1;

   $PrintM(A, "%e");

   /* compute eigenvalues and eigenvectors of matrix A */
   eval = $Eig(A, evct);
   $PrintM(eval, "%e");
   $PrintM(evct, "%e");

   /* check correctness */
   $InitM(I, ($I1 == $I2) ? 1 : 0, 0);
   k = 0;
   i = 0;
   while (i < size) begin
//   get next distinct root
     if (i < (size-1)) begin
      while (($VpAbs(eval[i].Re-eval[i+1].Re) < 0.0000001) &&
            ($VpAbs(eval[i].Im-eval[i+1].Im) < 0.0000001)) begin
        i = i + 1;
       end
     end

     l = eval[i];
     $display("\n\nTesting root %d: Re=%e Im=%e\n", i, l.Re, l.Im);
     LI = l*I;
```

```
    B = A - LI;
    r = $Rank(B);
    $display("Rank(B) = %d\n", r);

//  check each eigenvalue against its corresponding eigenvectors
    $display("There are %d eigenvectors\n", size-r);
    for (j = 0; j < size-r; j++) begin
      P = B*evctk;
      norm = $VpNormAbsMax(P);
      if (norm > 0.0001) $display("ERROR: for evect %d norm is %e\n", k, norm);
      else  $display("OK: for evect %d norm is %e\n", k, norm);
      k = k + 1;
    end
    i = i + 1;
  end
  $display("Test completed\n");
  end
endmodule // top
```

3.10.9.4. $LSim

This system function solves a system of linear differential equations. If the system is described by

A x X = B, and
C x X = D, where x is an n by 1 array representing the state (e.g. position and velocity in mechanical systems), u is a scalar representing the input (e.g. a force or torque in mechanical systems), and y is a scalar representing the output. The matrices A (n by n), B (n by 1), and C (1 by n) determine the relationships between the state and input and output variables,
then y = $LSim(A, B, C, D, u, t0, dt, nr_samples, x) solves the system, where the input and output variables are declared as described below

real A[0:size-1][0:size-1], B[0:size-1][0:m-1], C[0:p-1][0:size-1], D[0:p-1][0:m-1];
real x[0:size-1][0:nr_samples-1];
real y[0:p-1][0:nr_samples-1];, where p is the number of outputs and m is the number of inputs.

3.10.9.5. $Place

This system function computes a matrix K which will change the poles of the linear system A, B, C, D, to some given values for the system A-B*K, B, C, D. The syntax of this function call is K = $Place(A, B, poles), where A and be are as defined in the $LSim description, and poles is an array of Cartesian numbers containing the desired values of the poles.

3.10.10. Support for mixed numeric/symbolic computation

Support for mixed numeric and symbolic computations allows to perform numeric computations, store the results in data containers and then perform symbolic computations resulting in strings representing expressions that can be evaluated in the context of the data containers referenced by the resulting string.

An example of using mixed numeric and symbolic computation can be found on www.fintronic.com/dif_int_lap.html.

3.10.10.1. $Eval

This system function performs the numeric evaluation of a string, provided that it corresponds to a legal FinSimMath expression. The evaluation will be based on the current values of the variables participating in the expression.

3.10.10.2. $Dif

This function returns a string corresponding to the symbolic differentiation of a source string. The first argument is the number of differentiations applied consecutively and the second and last argument is the source string.

3.10.10.3. $Int

This function returns a string corresponding to the symbolic integration of a source string. The first argument is the number of integrations applied consecutively and the second and last argument is the source string.

3.10.10.4. $Lap

This function returns the Laplace transform of the expression provided as argument.

3.10.10.5. $ILap

This function returns the inverse Laplace transform of the symbolic expression (string) provided as argument.

3.10.11. Functions returning universal constants

3.10.11.1. $E

$E returns the value of e, i.e. 2.72…, with 128 bit of the fractional part or as many bits which fit in the register to which $E is assigned, whichever is lower.

3.10.11.2. $Pi

$Pi returns the value of pi, i.e. 3.14.., with 128 bit of the fractional part or as many bits which fit in the register to which $E is assigned, whichever is lower.

3.10.11.3. $EM

$EM returns the value of Euler-Mascheroni, i.e. 0.57……, with 128 bit of the fractional part or as many bits which fit in the register to which $E is assigned, whichever is lower.

3.10.12. Support User-defined System Tasks and Functions

User-defined system tasks and functions can be created using: 1) the Verilog/FinSimMath task and function mechanism, 2) the PLI mechanism, which allows to also check the validity of the arguments passed, or 3) for enhanced speed of execution, the C/C++ interface supported.

3.10.12.1. Creating Tasks and Functions using PLI

FinSim, the simulator supporting the FinSimMath extension of Verilog supports various ways of creating user-defined System Tasks and Functions based on PLI. In general simulators support various versions of PLI and FinSimMath can work with any such versions. What is important from the standpoint of using FinSimMath, is that one can use PLI to create tasks and fiunctions that can be invoked in a FinSimMath description. The advantage of using PLI is that one can check the validity of the arguments.

3.10.12.2. Creating Tasks and Functions using the supported C/C++ interface

An example of C code callable from FinSimMath is presented in chapter 5.

C/C++ functions can be called directly from within the Verilog/FinSimMath code. The user has to provide one or more C header files with the prototypes of the C functions.
How the header files and the executable code corresponding to the c-code are passed to the FinSimMath compiler is not part of the FinSimMath language, but is tool dependent. What

must be supported, however, is the capability to access the data stored in every possible data container of the FinSimMath language, including scalars, arrays and matrices, views of arrays and matrices, and sparse matrices.

3.10.12.2.1. Formal and Actual Arguments of C functions callable from FinSimMath

The arguments of C functions callable from FinSimMath can be characters (8 bits), short integers (16 bits), integers (32 bits), long integers (64 bits) and pointers of the above mentioned types. It is assumed that all pointers in the interface correspond to outputs that are going to be written inside the C functions. All other arguments are assumed to be inputs to the C functions.

The formal arguments of C functions callable from FinSimMath are either corresponding to actual arguments that are going to be passed at invocation or to actual arguments that are being implicitly passed at invocation. For each actual argument that is an array there are a number of optional implicit arguments that are automatically inserted by the FinSimMath compiler just before the array.

The implicit optional arguments, in order, are: type, view, and a number of triplets providing size, start index and end index of each supported dimension. The number of triplets is specified by the finvc compiler option +Insert_dimensions=n.

The type is of type int and provides the type of the array. It is inserted if the finvc invocation uses the option +Insert_type_Info.

The view information is of type long and provides a pointer to the indirection table for the given **view as** construct. It is inserted if the finvc invocation includes the option +Insert_view_info.

Options to insert apply to all actual arguments that are arrays. In case some arrays do not have the arguments to insert (e.g. +Insert_view_info is used, but the array is not a view, or +Insert_dimensions=2 is used but the array has only one dimension) then the missing arguments to insert will contain the value zero.

C functions that return a value which is an array will have the output appended to the list of arguments and will be preceded by all the appropriate implicit arguments.

3.10.12.2.2. Body of C functions callable from FinSimMath

Formal arguments that are of type long and that correspond to actual arguments which are arrays must be recast into a pointer of type simSignalPT. From this pointer one can access all the data within the array as shown in the example in section 5.3.2.

3.10.12.2.3. Environment variable related to C code invoked from FinSimMath

The object files containing the user C functions can be specified either in the file finpli.mak in the variable FINUSERCOBJ:

FINUSERCOBJ = example.o

or via the environment variable with the same name:

setenv FINUSERCOBJ example.o

More than one object files can be specified. If used, the file finpli.mak has to be in the local directory where finbuild, the linker of the FinSimMath compiler, is called.

If the specified object file does not exist, finbuild will attempt to compile it using a default compilation rule that calls the C compiler on the corresponding .c file.

3.10.12.2.4. Finvc invocation related to C code callable from FinSimMath

In FinSim, the header files providing the prototypes of the C functions are passed to the finvc compiler with the -ch <name of header file> option. This option can be specified any number of times if more than one header file is required. Note that the header files must be self-sufficient (as all well written header files ought to be), i.e. if a header file uses things defined in another header file then the 2nd header file should be included in the 1st header file. If any of the header files is in a different directory, the user can specify the include directory by using the +incdir option the same way as for Verilog header files.

4. Supplemental Synthesis Information

4.1. Introduction

This document addresses mainly the simulation aspects of FinSimMath. The synthesis aspects are part of a different standard candidate, FinSynthMath, which supports the description of (1) the synthesizable part of the FinSimMath description, i.e. the descriptor related info is omitted, (2) resources available for the synthesized circuit, (3) binding information of data containers, (4) topological information of the circuit, (5) Clock rates. This document addresses only FinSimMath and only briefly mentions FinSynthMath, with the purpose of highlighting the kind of information needed for synthesis and the relationship of such information to the simulation information.

In the near future translators from FinSynthMath to the input of commercially available high level synthesis can easily be developed. Also, performing high level synthesis on FinSynthMath directly is not only possible, but it is easier than from SystemC for example, due to the fact that FinSimMath provides information regarding the size and formats of data containers whereas SystemC does not and this info must be provided as companion information.

4.2. Resource file

This file contains information regarding resources available along with their cost, latency, geometrical parameters, number of bits, etc.

4.3. Binding information of data containers

The binding information of data containers consists of (1) the address in memory (including memory block) of given variables, (2) specification of which variables shall be implemented as registers, (3) specification of slices of busses to which a given variable can be connected.

4.4. Topological information of the circuit

The topological information of the circuit consists of (1) memory ports, and (2) description of buses and their inputs and outputs.

4.5. Clock rates

In case an input to the circuit being synthesized is a clock, it's rate may need to be provided in order to synthesize the circuit. The name of the clock must match the name of the input to which it corresponds.

4.6. Path to Synthesis from FinSimMath

In order to synthesize a circuit from an HDL description there is a need to provide additional synthesis information, since the HDLs are generally supporting only simulation.

The FinSimMath language has more information needed by synthesis than any other HDL. In addition to what Verilog, VHDL and SystemC have to provide and which FinSimMath does support since it is an extension of Verilog, FinSimMath also supports information regarding the format,(e.g. floating point, fixed point, two's complement), as well as the size of the various fields, e.g. exponent, mantissa, etc. Evidently, there still is a need for additional synthesis information even in case of FinSimMath, since it is preferable to limit FinSimMath to only constructs needed for simulation.

Having established that FinSImMath is the most complete language for implementing mathematical algorithms into ASICs we discussed how FinSimMath is supported by the tools provided by Fintronic in order to better understand how such a language can be implemented.

Since we discuss FinSim, as an implementation reference for FinSimMath, it is important to discuss also what the path to synthesis from FinSimMath is. The potential for developing the best synthesis program is there because FInSimMath provides more information, as we presented earlier.

Aside from the potential to develop a synthesis program it is interesting to know whether there are tools that can support a path to synthesis from FinSimMath at this time. For this purpose Fintronic supports:

1) A translator from FinSimMath to SystemC, which can be used as input for synthesis programs such as Cynthesizer from Cadence and

2) Mixed SystemC/FinSimMath simulations, to better help the interoperability between SystemC and FinSimMath.

5. Example of C code callable from FinSimMath

5.1. Introduction

This example shows how the function tf2ssc can be implemented in C code and can be invoked from within FinSimMath Code.

Given two polynomials a, b representing the transfer function b/a, the function tf2ssc generates the matrices A, B, C, and D corresponding to the controllable canonical form associated to the given transfer function specified by b/a.

Section 5.2 shows the .h file. Section 5.3 shows the .c file. Section 5.5 shows the pli.mak file. Section 5.4 shows the .v file where the tf2sc function is invoked and Section 5.6 shows the invocation of finvc with all the necessary options.

The detailed theory behind the tf2sc function this is explained in http://en.wikipedia.org/wiki/State-space_representation.

The actual algorithm for constructing the canonical controllable state space representation is available in http://lpsa.swarthmore.edu/Representations/SysRepTransformations/TF2SS.html

The actual example of how it works in MatLab is provided in http://www.mathworks.com/help/signal/ref/tf2ss.html. Note that one must to click on “expand all” in order to see the example. Please note that the tf2sc.c implementation provided assumes that the transfer function is reduced, namely the nominator and the denominator do not share common roots. Some additional coding is necessary to perform the reduction, using $Roots and $Poly.

5.2. The header file: lib.h

```
long tf2ssc(long file, int line,
            int sz1b, int st1b, int end1b,
              int sz2b, int st2b, int end2b, long b,
              int sz1a, int st1a, int end1a,
              int sz2a, int st2a, int end2a,  long a,
              int sz1SS, int st1SS, int end1SS,
              int sz2SS, int st2SS, int end2SS, long SS);
```

5.3. The c code file: lib.c

5.3.1. Type declarations

```
#include "stdio.h"
typedef struct _simDefaultT {
   unsigned char funcId;
   unsigned char zeroDriverVal;
   unsigned char resolution;
   unsigned char v;
} simDefaultT, *simDefaultPT;
typedef struct _simVmemT
{
   int      startI;
```

```
    int      endI;
    unsigned status;              /*new, old*/
    union{
       char      *memoryP;
       long int   fileOffset;
    } p;
} simVmemT, *simVmemPT;
typedef struct _simMemT
{
    union{
       simVmemPT    vP;
       char        *memoryP;
    } p;
    unsigned int     msb1d; /* msb of the address */
    unsigned int     lsb1d; /* lsb of the address */
    unsigned int     msb2d; /* msb of the word */
    unsigned int     lsb2d; /* lsb of the word */
    unsigned int     bCnt;
    int              flag; /*memory status: corruption*/
} simMemT, *simMemPT;
typedef union
{
  simMemPT               memP;
  char *csp;
} simSignalMiscT, *simSignalMiscPT;
typedef struct _simSignalT
{
  char *p0P;
  unsigned              flag;
  unsigned              flag2;
  char *p1P;
  char *p2P;
  char *p3P;
  simDefaultT           init;
  char *p4P;
  simSignalMiscT misc;
  char *p5P;
  char *p6P;
  char *p7P;
} simSignalT, *simSignalPT;
```

5.3.2. Code for body of C functions

```
long tf2ssc(long file, int line,
          int sz1b, int st1b, int end1b,
          int sz2b, int st2b, int end2b,
          long b,
          int sz1a, int st1a, int end1a,
          int sz2a, int st2a, int end2a,
          long a,
          int sz1SS, int st1SS, int end1SS,
          int sz2SS, int st2SS, int end2SS,
```

```
        long SS)
{
  /* declaration of internal objects needed */
  int i, j;
  char *(*aMP), *(*bMP), *(*SSMP);
  double tmpRe, *A, *B, *C, *D, *aP, *bP, *SSP;
  simSignalPT saP, sbP, sSSP, fileP;
  static int idx;

  /* extract file pointer and line number from arguments provided */
  fileP = (simSignalPT) file;
  /* recasting of arguments */
  saP = (simSignalPT) a;
  sbP = (simSignalPT) b;
  sSSP = (simSignalPT) SS;

  /* extracting memory addresses */
  aMP = (char **)saP->misc.memP->p.memoryP;
  bMP = (char **)sbP->misc.memP->p.memoryP;
  SSMP = (char **)sSSP->misc.memP->p.memoryP;

  /* Allocating space for denominator and nominator of transfer function */
  aP = (double *)calloc(sz1a, sizeof(double));
  bP = (double *)calloc(sz1b * sz2b, sizeof(double));

  /* read b, i.e. coefficients of numerator of transfer function  */
  for (i = 0; i < sz2b; i++) {
     for (j = 0; j < sz1b; j++) {
           simVpGet_r(bMP, sz1b*i + j, &tmpRe, 0, sz1b*sz2b-1);
           bP[sz1b*i + j] = tmpRe;
     }
  }

  /* read a, i.e. coefficients of denominator of transfer function */
  if (sz2a == 0) {
    for (i = 0; i < sz1a; i++) {
      simVpGet_r(aMP, i, &tmpRe, 0, sz1a-1);
      aP[i] = tmpRe;
    }
  }
  else {
    printf(" Error in file = %s, line = %d: denominator of tf2ssc must be a  one-dimensional
array\n",
          fileP, line);
    finExit(2);
  }
  if (sz2SS - sz1a > 0) {
    printf(" Error in file = %s, line = %d: tf2ss can handle only one input.\n",
          fileP, line);
    finExit(2);
  }
  if (sz2b - sz1a != 0) {
```

```
   printf(" Error in file = %s, line = %d: denominator and nominator given to tf2ss as arguments must have one dimension of the same size.\n",
          fileP, line);
   finExit(2);
  }
  /* from arrays aP and bP compute A, B, C, D in the controllable cannonical form */
  /* Allocate internal space for A, B, C, and D */
  A = (double *)calloc((sz1a-1) * (sz1a-1), sizeof(double));
  B = (double *)calloc((sz1a-1)*(sz1SS-sz1a+1), sizeof(double));
  C = (double *)calloc((sz1SS-sz1a+1)*(sz1a-1), sizeof(double));
  D = (double *)calloc((sz1SS-sz1a+1)*(sz2SS-sz1a+1), sizeof(double));

  /* compute A */
  for (i = 0; i < sz1a-1; i++) {
    for (j = 0; j < sz1a-1; j++) {
      if (i == 0) {
        A[j] = -aP[j+1]/aP[0];
      }
      else if (i == j+1) {
          A[i*(sz1a-1)+j] = 1;
      }
      else {
            A[i*(sz1a-1)+j] = 0;
      }
    }
  }

  /* compute B */
  for (i = 0; i < sz2b-1; i++) {
    B[i] = 0;
  }
  B[0] = 1;

  /* compute C */
  for (i = 0; i < sz1b; i++) {
    for (j = 0; j < sz1a-1; j++) {
      C[i*(sz1a-1) + sz2b-2-j] = bP[i*sz2b + sz2b-1-j]/aP[0] - aP[sz2b-1-j]*bP[0];
    }
  }

  /* compute D */
  for (i = 0; i < sz1b; i++) {
      D[i] = bP[sz1b-i];
  }

  /* write A, B,C, and D into output */
  for (i = 0; i < sz1SS; i++) {
      for (j = 0; j < sz2SS; j++) {
        if ((j < sz1a-1) && (i < sz1a-1)) {
                /* write A */
                tmpRe = A[i*(sz1a-1)+j];
                simVpPlace_r(SSMP, sz2SS*i+j, &tmpRe, 0, sz1SS*sz2SS-1);
```

```
        }
        else if ((j >= sz1a-1) && (i < sz1a-1)) {
                /* write B */
                tmpRe = B[i];
                simVpPlace_r(SSMP, sz2SS*i+j, &tmpRe, 0, sz1SS*sz2SS-1);
        }
          else if ((j < sz1a-1) && (i >= sz1a-1)) {
                /* write C */
                tmpRe = C[(i-(sz1a-1))*(sz1a-1)+j];
                simVpPlace_r(SSMP, sz2SS*i+j, &tmpRe, 0, sz1SS*sz2SS-1);
        }
        else {
                /* write D */
            tmpRe = D[(i-sz1a+1)*(sz1a-1)+j-sz1a+1];
                simVpPlace_r(SSMP, sz2SS*i+j, &tmpRe, 0, sz1SS*sz2SS-1);
     }
    }
  }
  free(aP);  free(bP);  free(A);  free(B);  free(C);  free(D);
}
```

5.4.FinSimMath file invoking function tf2ssc written in C code: tf2ss.v

```
module top;
`include "finsimmath.h"
parameter Nx = 2; /*nr of states */
parameter integer Ny = 2; /*nr of outputs*/
parameter [0:31]Nu = 1; /*nr of inputs */
real a[0:2], b[0:1][0:2];
real M[0:Nx+Ny-1][0:Nx+Nu-1];
view real A[0:Nx-1][0:Nx-1] as M[$I1][$I2];
view real B[0:Nx-1][0:Nu-1] as M[$I1][2];
view real C[0:Ny-1][0:Nx-1] as M[Nx+$I1][$I2];
view real D[0:Ny-1][0:Nu-1] as M[Nx+$I1][Nx+$I2];
initial begin
  a = {1.0, 0.4, 1.0};
  b = {0.0, 2.0, 3.0, 1.0, 2.0, 1.0};
  M = tf2ssc(b, a);
  $PrintM(a, "%e");
  $PrintM(b, "%e");
  $PrintM(A, "%e");
  $PrintM(B, "%e");
  $PrintM(C, "%e");
  $PrintM(D, "%e");
end
endmodule
```

5.5.The finpli.mak file

```
FINUSERCOBJ = lib.o
```

5.6.The invocation of finvc for this example

```
Finvc +FM +Insert_dimensions=2 +Insert_file_line –ch lib.h tf2ss.v
```

6. Example of FinSimMath Test Bench

6.1.Introduction

This example shows some of FinSimMath's capabilities in developing test benches. An FIR filter is described at the structural level along with its test bench. This circuit was automatically generated using FinFilter. The generated code that is presented in the following subsections, consists of stimulus generation, amplitude response computation, instantiation of device under test, supplying stimulus to the device under test, test bench controller (i.e. code that model the starting and ending of the filtering operation), computation and display of input/output spectrum, display of input output waveforms, using mixed level assertions to compare results, library of components used, computational unit of device under test and device under test.

6.2.Stimulus Generation

```
‘timescale 10fs / 10fs
module idata_gen (clk, init, data_reg, valid);
 input clk, init;
 output data_reg, valid;
 ‘include "finsimmath.h"
parameter SIZE = 1024;
 reg[0 : 27] data_reg;
 real delta;
 reg valid;
 integer j;
 VpDescriptor d1;
 VpReg[0 : 27] data;
 initial begin
 $VpSetDescriptorInfo(d1, 6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
 $VpSetDefaultOptions(6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
 $VpAssocDescrToData(data, d1);
 j = 0;
 valid = 0;
 delta = 2*$Pi;
 delta = delta/SIZE;
 end
 always @(negedge clk)
 valid = 0;
 always @(posedge clk)
 if (init)
 begin
 j = 0;
 data_reg = 0;
 valid = 0;
 end
 else
 begin
 data = $VpSin(153.600000*delta*j);
 data_reg = data;
 if (j<SIZE)
 begin
 valid = 1;
 j = j+1;
 end
 else
 begin
 valid = 0;
```

```
 end
 end
endmodule

module data_gen (clk, init, data_reg, valid);
 input clk, init;
 output data_reg, valid;
 ‘include "finsimmath.h"
parameter SIZE = 1024;
 reg[0 : 27] data_reg;
 real delta;
 reg valid;
 integer j;
 VpDescriptor d1;
 VpReg[0 : 27] data;
 VpReg[0 : 27] noise;
 initial begin
     $VpSetDescriptorInfo(d1, 6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
     $VpSetDefaultOptions(6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
     $VpAssocDescrToData(data, d1);
     $VpAssocDescrToData(noise, d1);
     j = 0;
     valid = 0;
     delta = 2*$Pi;
     delta = delta/SIZE;
 end

always @(negedge clk)
   valid = 0;

always @(posedge clk)
   if (init)
   begin
      j = 0;
      data_reg = 0;
      valid = 0;
   end
   else
   begin
       data = $VpSin(153.600000*delta*j);
       noise = $VpSin(460.800000*delta*j);
       data = data+noise;
       data_reg = data;
       if (j<SIZE)
       begin
          valid = 1;
          j = j+1;
       end
       else
       begin
           valid = 0;
       end
   end
endmodule
```

6.3.Top level module of Test Bench

6.3.1. Test Bench declarations

```
module top ();
    ‘include "finsimmath.h"
    parameter SIZE = 1024;
    parameter real srate = 2.000000;
    parameter real orate = 2.000000;
    parameter real irate = 1.000000;Page 3
    parameter real mrate = 8.000000;
    parameter time Tmclk = 6250.000000;
    parameter D1 = 1;
    parameter Np = 2;
    parameter SZ = 28;
    reg[0 : 111] data_ww;
    reg[0 : 27] data, data_i;
    reg[0 : 1] pack;
    reg init_clk, sampling_clk, output_clk, internal_clk, filterRead, dg_init,
    idg_init, fir_init, fir_init_1, fir_init_2, fir_init_3,filter_input_valid,
   record_filter, b, to_filter;
   VpDescriptor d1;
   VpReg[0 : 27] vp_sum;
   VpReg[0 : 27] vp_term;
   VpReg[0 : 27] sampledIn[0:SIZE-1];
    VpReg[0 : 27] filteredOut[0:SIZE-1];
    VpReg[0 : 27] idealOut[0:SIZE-1];
    VpReg[0 : 27] vp;
    VpReg[0 : 27] vp1;
    VpFPolar polar;
    VpFComplex H, jomega, ejomega, ejomegak, c_term;
    VpFComplex inSpectrum[0:SIZE-1];
    VpFComplex outSpectrum[0:SIZE-1];
    reg[0 : 55] out_r;
    wire [0:27] data_w, idata_w;
    wire [0:2*SZ-1] out;
    wire [0:Np*SZ-1] data_ww_0, data_ww_1, data_to_filter;
    real y[0:2][0:SIZE-1], step, omega, M, T, l10, h, distance, sum, fact, Hr, t1;
   integer i, i_in, j, k, p, dec1, dec2, scnt, ocnt, icnt, D0, dstate, F, K, R, m;
   reg[0 : 27] mem[0 : 30];
```

6.3.2. Clock Generation

```
assign #(Tmclk) m_clk = (~m_clk)&&init_clk;

always @(negedge m_clk)
begin
      scnt = (scnt+1)%2;
      if (scnt==0)
         sampling_clk = ~sampling_clk;
      ocnt = (ocnt+1)%2;
      if (ocnt==0)
         output_clk = ~output_clk;
      icnt = (icnt+1)%4;
      if (icnt==0)
         internal_clk = ~internal_clk;
end
```

6.3.3. Amplitude Response Computation

```
task ComputeGainC;Page 4
begin
    step = (srate/SIZE)/2;
    l10 = $VpLn(10.0);
    omega = 0;
    for (j = 1; (j < SIZE); j = j+1)
    begin
        vp = $VpCopyReg2Vp(mem[fir.Na]);
        Hr = vp*fact;
        for (i = 1; (i <= 15); i = i+1)
        begin
        vp = $VpCopyReg2Vp(mem[fir.Na-i]);
        t1 = vp*fact;
        t1 = t1*2*$VpCos(omega*i*T);
        Hr = Hr+t1;
        end
        Hr = $VpAbs(Hr);
        y[0][j] = 20*$VpLn(Hr)/l10;
        omega = omega + step;
    end
end
endtask
```

6.3.4. Instantiation of Device Under Test

```
fir fir(internal_clk, ~filter_input_valid, data_to_filter, ready, out, filter_valid);
```

6.3.5. Instantiation of Modules generating Stimulus

```
data_gen #(SIZE) data_gen(sampling_clk, dg_init||fir_init, data_w, dg_valid);
idata_gen #(SIZE) idata_gen(sampling_clk, dg_init||fir_init, idata_w, idg_valid);
```

6.3.6. Supplying Stimulus to the Device under Test

```
always @(negedge sampling_clk)
begin
    if (dg_valid)
    begin
        if (i_in < SIZE)
        begin
            vp = $VpCopyReg2Vp(data_w);
            sampledIn[i_in] = vp;
            vp = $VpCopyReg2Vp(idata_w);
            idealOut[i_in] = vp;
        end
        if (pack == Np-1)
        begin
            data_ww[(b*Np*SZ+(Np-1-pack)*SZ)+:SZ] = data_w;
            pack = 0;
            if (b == 1)
            b <= 1'b0;
            else
            b <= 1'b1;
            filter_input_valid <= 1'b1;
        end
```

```
        else
        begin
            data_ww[(b*Np*SZ+(Np-1-pack)*SZ)+:SZ] = data_w;
            pack = pack+1;
        end
        i_in = i_in+1;
    end
end

always @(posedge ready)
begin
    dg_init = 1'b0;
    fir_init_1 <= 1'b0;
end

always @(posedge internal_clk)
begin
    if (!fir_init_1)
    fir_init_2 <= 1'b0;
    if (!fir_init_2)
    fir_init_3 <= 1'b0;
    if (!fir_init_3)
    fir_init <= 1'b0;
end

always @(posedge filter_valid)
begin
    if (filter_input_valid)
    begin
        record_filter = 1'b1;
        filterRead = 1'b0;
        out_r = out;
    end
end
```

6.3.7. Getting the results from the Device under Test

```
always @(posedge output_clk)
begin
  if ( filter_input_valid && (record_filter || (filter_valid && !filterRead)))
    begin
        record_filter = 1'b0;
        filterRead = 1'b1;
        K = 2;
        R = 1;
        for (p = 0; (p < K); p = p+1)
        begin
            data = out_r[p*SZ+:SZ];
            vp = $VpCopyReg2Vp(data);
            vp1 = fact*vp;
            filteredOut[k] = vp1;
            if (k<SIZE-1-Np)
            begin
                k = k+1;
                end
                else
                begin
                fir_init_3 = 1'b1;
                fir_init_2 = 1'b1;
```

```
                fir_init_1 = 1'b1;Page 6
                fir_init = 1'b1;
                init_clk = 1'b0;
            end
            R = D1;
        end
    end
end
```

6.3.8. Test Bench Controller

```
initial begin
/* Dumping signals for debugging with waveform display*/
    $dumpvars(1);

    /* associating variable precision registers to their descriptors */
    $VpSetDescriptorInfo(d1, 6, 22, 1, 'JUST_TRUNCATE, 1, 1);
    $VpSetDefaultOptions(6, 22, 1, 'JUST_TRUNCATE, 1, 1);
    $VpAssocDescrToData(vp, d1);
    $VpAssocDescrToData(vp1, d1);
    $VpAssocDescrToData(vp_sum, d1);
    $VpAssocDescrToData(vp_term, d1);
    $VpAssocDescrToData(sampledIn, d1);
    $VpAssocDescrToData(idealOut, d1);
    $VpAssocDescrToData(filteredOut, d1);

    /*preparing constant expressions used in multiple places */
    T = (2*$Pi)/srate;
    mem[0] = 28'b0000000000000000000000000000;
    mem[1] = 28'b1111111101001100110111000101;
    mem[2] = 28'b1111111100111111000101001010;
    mem[3] = 28'b1111111111001110101010011011;
    mem[4] = 28'b0000000000110101011100101001;
    mem[5] = 28'b0000000000000000000000000000;
    mem[6] = 28'b1111111110111110001101111001;
    mem[7] = 28'b0000000001001010000000011000;
    mem[8] = 28'b0000000101100110010001110111;
    mem[9] = 28'b0000000110100001111111100000;
    mem[10] = 28'b0000000000000000000000000000;
    mem[11] = 28'b1111110110001101000000101111;
    mem[12] = 28'b1111110010111100000000111111;
    mem[13] = 28'b1111111011010111111110011111;
    mem[14] = 28'b0000001001010000000011000011;
    mem[15] = 28'b0000010000000000000000000000;
    mem[16] = 28'b0000001001010000000011000011;
    mem[17] = 28'b1111111011010111111110011111;
    mem[18] = 28'b1111110010111100000000111111;
    mem[19] = 28'b1111110110001101000000101111;
    mem[20] = 28'b0000000000000000000000000000;
    mem[21] = 28'b0000000110100001111111100000;
    mem[22] = 28'b0000000101100110010001110111;
    mem[23] = 28'b0000000001001010000000011000;
    mem[24] = 28'b1111111110111110001101111001;
    mem[25] = 28'b0000000000000000000000000000;
    mem[26] = 28'b0000000000110101011100101001;
    mem[27] = 28'b1111111111001110101010011011;
    mem[28] = 28'b1111111100111111000101001010;
    mem[29] = 28'b1111111101001100110111000101;
    mem[30] = 28'b0000000000000000000000000000;
```

```
sum = 0;
for (i = 0; (i <= fir.Na); i = i+1)
begin
    vp = $VpCopyReg2Vp(mem[i]);
    $display("coef[%d]=%y %k\n", i, vp, vp) ;
end
fact = 2.000000e-01;

/* preparing the start of the filtering activity */
dg_init = 1'b1;
icnt = 0;
scnt = 0;
ocnt = 0;
sampling_clk = 1'b0;
output_clk = 1'b0;
internal_clk = 1'b0;
fir_init_3 = 1'b1;
fir_init_2 = 1'b1;
fir_init_1 = 1'b1;
fir_init = 1'b1;
record_filter = 1'b0;
filterRead = 1'b1;
filter_input_valid = 1'b0;
pack = 0;
b = 1'b0;
to_filter = 1'b1;
k = 0;
dec2 = 0;
i_in = 0;
init_clk = 1'b0;
#(200000.000000);
init_clk = 1'b1;
#(200000.000000);
```

6.3.9. Computation and Display of Amplitude Response

```
@(negedge init_clk);
ComputeGainC;
step = (srate/SIZE)/2;
$Flot("Ex1_Gain.html", 1, step, "Amplitude Gain", "frequency (GHz)",
"Gain (dB)", 0, SIZE-1, y, "amplitude");
step = (srate/SIZE);
```

6.3.10. Computation and Display of Input/Output Spectrum

```
$InitM(inSpectrum, sampledIn[$I1], 0);
$VpFft(inSpectrum, 0, SIZE-1);
$InitM(outSpectrum, filteredOut[$I1], 0);
$VpFft(outSpectrum, 0, SIZE-1);
for (j = 0; j < (SIZE/2); j = j + 1)
begin
    polar = inSpectrum[j];
    y[0][j] = polar.Mag;
    polar = outSpectrum[j];
    y[1][j] = polar.Mag;
end
$Flot("Ex1_ioSpectrum.html", 2, step, "Input-Output Spectrum", "frequency
(GHz)", "Amplitude ", 0, (SIZE/2)-1, y, "inSpectrum", "outSpectrum");
```

6.3.11. Display Input/Output Waveforms

```
for (j = 0; j < SIZE; j = j + 1)
begin
    y[0][j] = filteredOut[j];
    y[1][j] = sampledIn[j];
    y[2][j] = idealOut[j];
end
step = 5.000000e+04;
$Flot("Ex1_Input_Output.html", 3, step, "Filtered output vs input",
"time (10 fs)", "Amplitude (multiple of arbitrary constant)", 0, SIZE-1, y,
"out", "in", "iout");
```

6.3.12. Compute and Display Distances

```
distance = $VpDistAbsSum(filteredOut, idealOut)/SIZE;
$display("distance between filtered out and ideal output = %e\n", distance) ;
distance = $VpDistAbsSum(sampledIn, idealOut)/SIZE;
$display("distance between sampled input and ideal output = %e\n", distance) ;
```

6.3.13. Use of Mixed Level Assertions to compare Results

```
R = D1;
for (j = 0; (j < SIZE+1-2*Np); j = j+Np)
for (k = 0; (k < Np); k = k+R)
begin
    vp_sum = 0;
    for (i = 0; (i < 31); i = i+1)
        if (j+k-i >= 0)
        begin
            vp = $VpCopyReg2Vp(mem[i]);
            vp_term = vp*sampledIn[j+k-i];
            vp_sum = vp_sum+vp_term;
        end
    vp_sum = vp_sum * fact;
    vp = vp_sum - filteredOut[j+k];
    vp_sum = $VpAbs(vp);
    sum = vp_sum;
    if (sum > 3.814697e-06)
    begin
        vp = filteredOut[j+k];
        $display("Error: filteredOut[%d]=%y, dif=%f, %y, %k\n", j+k, vp, sum, vp_sum,
        vp_sum);
    end
end
end

/* resources needed to serialize the input data coming at the sampling rate which is higher in
this case than the internal clck frequency of the filterdata to the filter */
not not_to_filter(nto_filter, to_filter);
bufif1 bufif1_0[0:55](data_to_filter, data_ww[0:Np*SZ-1], b);
bufif1 bufif1_1[0:55](data_to_filter, data_ww[Np*SZ:2*Np*SZ-1], ~b);
endmodule
```

6.4. Library of Elementary Modules

```
module jkff (clk, init, j, k, out, nout);
      input clk, init, j, k;
      output out, nout;
      wire clk, init, j, k;
      reg out, nout, data;
      always @(negedge clk)
      begin
      if (init)
      data = 1'b0;
      else
      if (j && k)
      data = ~data;
      else
      if (j)
      data = 1'b1;
      else
      if (k)
      data = 1'b0;
      out <= data;
      nout <= ~data;
      end
     endmodule

     module mem_in (clk, en, data, d_2_3, d_4_5, d_6_7, d_8_9, d_10_11, d_12_13,
     d_14_15, d_16_17, d_18_19, d_20_21, d_22_23, d_24_25, d_26_27, d_28_29,
     d_30_31);
      input clk, en, data;
      output d_2_3, d_4_5, d_6_7, d_8_9, d_10_11, d_12_13, d_14_15, d_16_17,
     d_18_19, d_20_21, d_22_23, d_24_25, d_26_27, d_28_29, d_30_31;
      wire clk;
      wire[0 : 55] data;
      reg[0 : 55] d_2_3, d_4_5, d_6_7, d_8_9, d_10_11, d_12_13, d_14_15, d_16_17,
     d_18_19, d_20_21, d_22_23, d_24_25, d_26_27, d_28_29, d_30_31;
      reg[0 : 27] mem[0 : 31];
      reg[0 : 5] i;
      initial begin
      for (i = 0; (i < 32); i = (i + 1))
      begin
      mem[i] = 0;
      end
      end
      always @(posedge clk)
      begin
      if (en)
      begin
      for (i = 0; (i < 29); i = (i + 1))
      begin
      mem[31-i] = mem[31-i-2];
      end
      mem[2] = data[0:27];
      mem[3] = data[28:55];
      mem[1] = mem[31];
      mem[0] = mem[30];
      end
      d_2_3 = {mem[2], mem[3]};
      d_4_5 = {mem[4], mem[5]};
      d_6_7 = {mem[6], mem[7]};
```

```
        d_8_9 = {mem[8], mem[9]};
        d_10_11 = {mem[10], mem[11]};
        d_12_13 = {mem[12], mem[13]};
        d_14_15 = {mem[14], mem[15]};
        d_16_17 = {mem[16], mem[17]};
        d_18_19 = {mem[18], mem[19]};
        d_20_21 = {mem[20], mem[21]};
        d_22_23 = {mem[22], mem[23]};
        d_24_25 = {mem[24], mem[25]};
        d_26_27 = {mem[26], mem[27]};
        d_28_29 = {mem[28], mem[29]};
        d_30_31 = {mem[30], mem[31]};
    end
endmodule

 module sub_fx_6_22 (clk, init, op1_reg, op2_reg, op3_reg, v_reg);
      input clk, init, op1_reg, op2_reg;
      output op3_reg, v_reg;
      ‘include "finsimmath.h"
      wire clk, init;
      wire [0:27]op1_reg, op2_reg;
      reg v_reg;
      reg[0 : 27] op3_reg;
      VpDescriptor d1;
      VpReg[0 : 27] op1;
      VpReg[0 : 27] op2;
      VpReg[0 : 27] op3;
      initial begin
          $VpSetDescriptorInfo(d1, 6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
          $VpSetDefaultOptions(6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
          $VpAssocDescrToData(op1, d1);
          $VpAssocDescrToData(op2, d1);
          $VpAssocDescrToData(op3, d1);
          v_reg <= 0;
      end

      always @(negedge clk)
      if (init)
      begin
      v_reg <= 0;
      op3_reg <= 0;
      end
      else
      if (init===0)
      begin
      op1 = $VpCopyReg2Vp(op1_reg);
      op2 = $VpCopyReg2Vp(op2_reg);
      op3 = op1-op2;
      op3_reg <= op3;
      v_reg <= 1’b1;
      end
  endmodule

  module add_fx_6_22 (clk, init, op1_reg, op2_reg, op3_reg, v_reg);
       input clk, init, op1_reg, op2_reg;
       output op3_reg, v_reg;
         ‘include "finsimmath.h"
         wire clk, init;
         wire [0:27]op1_reg, op2_reg;
         reg v_reg;
```

```
 reg[0 : 27] op3_reg;
 VpDescriptor d1;
 VpReg[0 : 27] op1;Page 11
 VpReg[0 : 27] op2;
 VpReg[0 : 27] op3;
 initial begin
    $VpSetDescriptorInfo(d1, 6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
    $VpSetDefaultOptions(6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
    $VpAssocDescrToData(op1, d1);
    $VpAssocDescrToData(op2, d1);
    $VpAssocDescrToData(op3, d1);
    v_reg <= 0;
 end

 always @(negedge clk)
 if (init)
   begin
      v_reg <= 0;
      op3_reg <= 0;
    end
    else
    if (init===0)
       begin
          op1 = $VpCopyReg2Vp(op1_reg);
          op2 = $VpCopyReg2Vp(op2_reg);
          op3 = op1+op2;
          op3_reg <= op3;
          v_reg <= 1’b1;
       end
endmodule

module mlt_fx_6_22 (clk, init, op1_reg, op2_reg, op3_reg, v_reg);
    input clk, init, op1_reg, op2_reg;
    output op3_reg, v_reg;
    ‘include "finsimmath.h"
    wire clk, init;
    wire [0:27]op1_reg, op2_reg;
    reg v_reg;
    reg[0 : 27] op3_reg;
    VpDescriptor d1;
    VpReg[0 : 27] op1;
    VpReg[0 : 27] op2;
    VpReg[0 : 27] op3;
    initial begin
       $VpSetDescriptorInfo(d1, 6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
       $VpSetDefaultOptions(6, 22, 1, ‘JUST_TRUNCATE, 1, 1);
       $VpAssocDescrToData(op1, d1);
       $VpAssocDescrToData(op2, d1);
       $VpAssocDescrToData(op3, d1);
       v_reg <= 0;
    end

    always @(negedge clk)
    if (init)
    begin
        v_reg <= 0;
        op3_reg <= 0;
        end
        else
        if (init===0)
```

```
        begin
        op1 = $VpCopyReg2Vp(op1_reg);
        op2 = $VpCopyReg2Vp(op2_reg);
        op3 = op1*op2;
        op3_reg <= op3;
        v_reg <= 1'b1;
    end
endmodule


module dl1_28 (clk, init, op1_reg, op3_reg, v_reg);
     input clk, init, op1_reg;
     output op3_reg, v_reg;
     'include "finsimmath.h"
     wire clk, init;
     wire [0:27]op1_reg;
     reg v_reg;
     reg[0 : 27] op3_reg;
     VpDescriptor d1;
     VpReg[0 : 27] op1;
     VpReg[0 : 27] op3;
     initial begin
        $VpSetDescriptorInfo(d1, 6, 22, 1, 'JUST_TRUNCATE, 1, 1);
        $VpSetDefaultOptions(6, 22, 1, 'JUST_TRUNCATE, 1, 1);
        $VpAssocDescrToData(op1, d1);
        $VpAssocDescrToData(op3, d1);
        v_reg <= 0;
     end
     always @(negedge clk)
     if (init)
     begin
         v_reg <= 0;
         op3_reg <= 0;
     end
     else
     if (init===0)
     begin
         op1 = $VpCopyReg2Vp(op1_reg);
         op3 = op1;
         op3_reg <= op3;
         v_reg <= 1'b1;
     end
endmodule
```

6.5.Computational Unit of Device under Test

```
module tu (clk, init, in12_1, in11_1, in10_1, in9_1, in8_1, in7_1, in6_1,
  in5_1, in4_1, in3_1, in2_1, in1_1, in0_0, in1_0, in2_0, in3_0, in4_0, in5_0,
  in6_0, in7_0, in8_0, in9_0, in10_0, in11_0, in12_0, data_out, valid_out);
    input clk, init, in0_0, in1_0, in2_0, in3_0, in4_0, in5_0, in6_0, in7_0,
   in8_0, in9_0, in10_0, in11_0, in12_0, in12_1, in11_1, in10_1, in9_1, in8_1,
   in7_1, in6_1, in5_1, in4_1, in3_1, in2_1, in1_1;
    output data_out, valid_out;
    wire clk, init, valid_out;
    wire[0 : 27] in0, in1, in2, in3, in4, in5, in6, in7, in8, in9, in10, in11,
   in12, in0_0, in1_0, in2_0, in3_0, in4_0, in5_0, in6_0, in7_0, in8_0, in9_0,
   in10_0, in11_0, in12_0, in12_1, in11_1, in10_1, in9_1, in8_1, in7_1, in6_1,
   in5_1, in4_1, in3_1, in2_1, in1_1, out_0_0, out_1_0, out_2_0, out_3_0,
```

```
    out_4_0, out_5_0, out_6_0, out_7_0, out_8_0, out_9_0, out_10_0, out_11_0,
    out_12_0, out_0_1, out_1_1, out_2_1, out_3_1, out_4_1, out_5_1, out_6_1,
    out_7_1, out_8_1, out_9_1, out_10_1, out_11_1, out_12_1, out_0_2, out_1_2,
    out_2_2, out_3_2, out_4_2, out_5_2, out_6_2, out_7_2, out_0_3, out_1_3,
    out_2_3, out_3_3, out_0_4, out_1_4, data_out;
     jkff ff1(clk, init, ninit, init, ninit1, init1);
     jkff ff2(clk, init, ninit1, init1, ninit2, init2);
     jkff ff3(clk, init, ninit2, init2, ninit3, init3);
     jkff ff4(clk, init, ninit3, init3, ninit4, init4);
     jkff ff5(clk, init, ninit4, init4, ninit5, init5);
     jkff ff6(clk, init, ninit5, init5, ninit6, init6);
     dl1_28 pdelay1_0(clk, init, in0_0, in0, );
     add_fx_6_22 padd_1(clk, init, in1_0, in1_1, in1, );
     add_fx_6_22 padd_2(clk, init, in2_0, in2_1, in2, );
     add_fx_6_22 padd_3(clk, init, in3_0, in3_1, in3, );
     add_fx_6_22 padd_4(clk, init, in4_0, in4_1, in4, );
     add_fx_6_22 padd_5(clk, init, in5_0, in5_1, in5, );
     add_fx_6_22 padd_6(clk, init, in6_0, in6_1, in6, );
     add_fx_6_22 padd_7(clk, init, in7_0, in7_1, in7, );
     add_fx_6_22 padd_8(clk, init, in8_0, in8_1, in8, );
     add_fx_6_22 padd_9(clk, init, in9_0, in9_1, in9, );
     add_fx_6_22 padd_10(clk, init, in10_0, in10_1, in10, );
     add_fx_6_22 padd_11(clk, init, in11_0, in11_1, in11, );
     add_fx_6_22 padd_12(clk, init, in12_0, in12_1, in12, );
     dl1_28 pair_0(clk, init1, in0, out_0_1);
     mlt_fx_6_22 mlt_1(clk, init1, 28'b0000001001010000000011000011, in1, out_1_1, v_1_1);
     mlt_fx_6_22 mlt_2(clk, init1, 28'b1111111011010111111110011111, in2, out_2_1, v_2_1);
     mlt_fx_6_22 mlt_3(clk, init1, 28'b1111110010111100000000111111, in3, out_3_1, v_3_1);
     mlt_fx_6_22 mlt_4(clk, init1, 28'b1111110110001101000000101111, in4, out_4_1, v_4_1);
     mlt_fx_6_22 mlt_5(clk, init1, 28'b0000000110100001111111100000, in5, out_5_1, v_5_1);
     mlt_fx_6_22 mlt_6(clk, init1, 28'b0000000101100110010001110111, in6, out_6_1, v_6_1);
     mlt_fx_6_22 mlt_7(clk, init1, 28'b0000000001001010000000011000, in7, out_7_1, v_7_1);
     mlt_fx_6_22 mlt_8(clk, init1, 28'b1111111110111110001101111001, in8, out_8_1, v_8_1);
     mlt_fx_6_22 mlt_9(clk, init1, 28'b0000000000110101110100101001, in9, out_9_1, v_9_1);
     mlt_fx_6_22  mlt_10(clk,  init1,  28'b1111111111001110101010011011,  in10,  out_10_1,
    v_10_1);
     mlt_fx_6_22  mlt_11(clk,  init1,  28'b1111111100111111000101001010,  in11,  out_11_1,
    v_11_1);
     mlt_fx_6_22  mlt_12(clk,  init1,  28'b1111111101001100110111000101,  in12,  out_12_1,
    v_12_1);
     add_fx_6_22 add_0_2(clk, init2, out_0_1, out_1_1, out_0_2);
     add_fx_6_22 add_1_2(clk, init2, out_2_1, out_3_1, out_1_2);
     add_fx_6_22 add_2_2(clk, init2, out_4_1, out_5_1, out_2_2);
     add_fx_6_22 add_3_2(clk, init2, out_6_1, out_7_1, out_3_2);
     add_fx_6_22 add_4_2(clk, init2, out_8_1, out_9_1, out_4_2);
     add_fx_6_22 add_5_2(clk, init2, out_10_1, out_11_1, out_5_2);
     dl1_28 del_12_2(clk, init2, out_12_1, out_6_2);
     add_fx_6_22 add_0_3(clk, init3, out_0_2, out_1_2, out_0_3);
     add_fx_6_22 add_1_3(clk, init3, out_2_2, out_3_2, out_1_3);
     add_fx_6_22 add_2_3(clk, init3, out_4_2, out_5_2, out_2_3);
     dl1_28 dl1_3_3(clk, init3, out_6_2, out_3_3, );
     add_fx_6_22 add_0_4(clk, init4, out_0_3, out_1_3, out_0_4);
     add_fx_6_22 add_1_4(clk, init4, out_2_3, out_3_3, out_1_4);
     add_fx_6_22 add_out(clk, init5, out_0_4, out_1_4, data_out, v_out);
     and and_valid(valid_out, clk, v_out);
     not n1(ninit, init);
endmodule
```

6.6. Device under Test

```
‘timescale 10fs / 10fs
‘timescale 10fs / 10fs
module fir (clk, init, d_1_0, ready, data_out, valid_out);
        input clk, init, d_1_0;
        output ready, data_out, valid_out;
        parameter Na = 15;
        wire clk, init, ready, valid_out;
        wire[0 : 55] data_out;
        wire[0 : 55] d_in, d_1_0, d_2_3, d_4_5, d_6_7, d_8_9, d_10_11, d_12_13,
       d_14_15, d_16_17, d_18_19, d_20_21, d_22_23, d_24_25, d_26_27, d_28_29,
       d_30_31;
        mem_in mem_in(clk, ninit, d_1_0, d_2_3, d_4_5, d_6_7, d_8_9, d_10_11,
       d_12_13, d_14_15, d_16_17, d_18_19, d_20_21, d_22_23, d_24_25, d_26_27,
       d_28_29, d_30_31);
        tu tu_0(clk, init, d_2_3[0:27], d_2_3[28:55], d_4_5[0:27], d_4_5[28:55], d_6_7[28:55],
       d_8_9[0:27], d_8_9[28:55], d_10_11[0:27], d_12_13[0:27], d_12_13[28:55],
       d_14_15[0:27], d_14_15[28:55], d_16_17[0:27], d_16_17[28:55], d_18_19[0:27]
       , d_18_19[28:55], d_20_21[0:27, d_22_23[0:27], d_22_23[28:55], d_24_25[0:27]
       , d_24_25[28:55], d_26_27[28:55], d_28_29[0:27], d_28_29[28:55], d_30_31[0:27]
       , data_out[0:27], valid_out);
        tu tu_1(clk, init, d_1_0[28:55], d_2_3[0:27], d_2_3[28:55], d_4_5[0:27], d_6_7[0:27]
       , d_6_7[28:55], d_8_9[0:27], d_8_9[28:55], d_10_11[28:55], d_12_13[0:27]
       , d_12_13[28:55], d_14_15[0:27], d_14_15[28:55], d_16_17[0:27], d_16_17[28:55]
       , d_18_19[0:27], d_18_19[28:55], d_20_21[28:55], d_22_23[0:27], d_22_23[28:55]
       , d_24_25[0:27], d_26_27[0:27], d_26_27[28:55], d_28_29[0:27], d_28_29[28:55]
       , data_out[28:55], valid_out);
        not n1(ninit, init);
        jkff ffi1(clk, 1’b0, 1’b1, 1’b0, ready1, nready1);
        jkff ffi2(clk, 1’b0, ready1, nready1, ready2, nready2);
        jkff ffr(clk, 1’b0, ready2, nready2, ready, nready);
endmodule
```

7. Comments on the Test Bench presented in chapter 6

7.1. Introduction

FinSimMath supports the conversion of mathematical algorithms into structural level Verilog also by supporting the development of the corresponding test benches. Such test benches can: (1) use mixed mathematical and Verilog stimulus generation and assertions, and (2) analyse and display information using math-level constructs.

7.2. Mixed Mathematical and Verilog Stimulus Generation

Section 6.2 contains the code of two modules that produce stimulus.

The first one, module idata_gen generates the samples of the ideal output sampled at the sample rate desired by the user. The ideal output is a single frequency waveform labelled by the user as ideal because if supplied as input to the filter it is supposed to show up at the output of the filter unaffected by the filtering operation.

The second one, module data_gen, generates the samples of the input waveform which is the sum of two single frequency waveforms, ideal output and noise. Both frequencies have been selected by the user to be consistent with the characteristic of the Filter, namely the ideal output being preserved and the noise being attenuated by the filter

The generation of the samples requires the usage of the system function $VpSin, which computes the sine function.

Note that the values of the samples are computed in variable precision registers which then are assigned to Verilog registers in order to be passed as outputs of the module.

7.3. Bit accurate mathematical-level models of computational units

Section 6.4 contains bit accurate mathematical-level models of multipliers and adders. Such models result in simulations that are 1000x faster than simulations involving the corresponding gate-level implementation.

7.4. Mixed Mathematical and Verilog assertions

Sub-section 6.3.13 contains the code that computes the filtering operations at the mathematical level (a scalar product) and compares the results with the results obtained from the generated filter. Note that the mathematical operations are performed using the exact number of bits used in the actual implementation. Of course it is important for the assertions to be correct, and in particular to ensure that the values of the high level mathematical computation are compared with the appropriate values of the gate-level implementation. Mixed level assertions provide a substantial help in debugging.

7.5. Analyzing and displaying information using math-level constructs

Sub-section 6.3.3 contains the code that computes the amplitude response of the filter. Note that the computations are performed using the exact number of bits used in the actual implementations, by using coefficients stored in variable precision registers. Therefore the computed amplitude response will reflect the number of bits used. In order to get the value of a

Verilog register stored in a variable precision register, one must use an explicit conversion function, as is done in:

vp = $VpCopyReg2Vp(mem[fir.Na]);, where mem is a Verilog array of coefficients and vp is a VpReg, having a descriptor associated to it. The conversion is necessary because a simple assignment of a Verilog register to a variable precision register will put in the variable precision register the value contained in the Verilog register (which is a signed or unsigned integer) and not copy the bit pattern as the conversion function does.

The amplitude response computation uses $VpCos and $VpLn, which compute cosine and natural logarithm, respectively.

The code invoking the computation of the amplitude response and the displaying of the amplitude response can be found in sub-section 6.3.9. Note that the amplitude response is computed taking into account the number of bits used by the filter and it is not the amplitude response of an ideal filter having infinite resources.

The input/output spectrum is computed using the $VpFft system task which performs the fft transform and the corresponding code can be found in sub-section 6.3.10.