FINSIM 12.0

User's Guide

SEPTEMBER 3, 2018

FINTRONIC USA, INC
1110 Oakridge Dr, Roseville California

Contents

1.0 Introduction

The FinSim Simulation Environment (Super-FinSim), consists of a Verilog 2001 compliant ompiler, a simulation builder, and a simulation kernel. The FinSimMath Simulation Environment consists of of Super-FinSim plus support for FinSimMath, which is an extension of Verilog for mathematical descriptions.

The purpose of the Verilog compiler is to (1) check the design for syntactic and semantic correctness, (2) generate code and elaboration data required to configure and program the simulation kernel according to the design description, and (3) optionally generate an intermediate format representation of the description for processing by other applications.

The purpose of the simulation builder is to compile the generated C code (if any) and link all the files that are necessary to build a simulator, i.e. the C object files corresponding to the Verilog source, the PLI object files and libraries and the simulation kernel library. The host C linker is invoked for this purpose.

The simulation kernel is the code that is common to the simulation of all Verilog/FinSimMath designs. Once configured and programmed, the simulation kernel becomes a simulator for a particular Verilog /FinSimMath design. The simulator in Super-FinSim can run in compiled, interpreted or a mix of compiled and interpreted modes. The simulator in FinSimMath can run only in compiled mode.

1.1 Purpose of this document

This document describes how to use the FinSim/FinSimMath Simulation Environment. Specifically, it describes:

a) Installation of FinSim/FinSimMath

b) How to use the compiler

c) How to use the simulation builder

d) How to use the simulation engine

e) Interactive commands

f) FinSimMath Reference Manual

g) How to interface FinSim/FinSimMath with other tools

h) How to interface with functions written in C/C++

i) How to perform Separate Compilation.

2.0 Installation

2.1 Installation Steps

The user must provide the MAC address of the server of the network on which FinSim is supposed to be executed. In response Fintronic will send a license file corresponding to the given MAC address. The license file must be pointed at by the environment variable RLM_LICENSE, which can be achieved, If using csh, by: setenv RLM_LICENSE full_path_of license_file_received_from_fintronic.

Also, the license file must be stored in the same directory as the rlm and fintronic daemons. These daemons can be downloaded as follows:

ftp fintronic.com

user name: support@fintronics.com

password: password provided by Fintronic

get fintronic

get rlm

The software can be downloaded as follows:

ftp fintronic.com

user name: download@fintronics.com

password: password provided by Fintronic

get finflot.tgz

cd finsim/finsim_version/linux64

get install.csh

get finsim.tar.gz

To complete the installation the following 7 steps must be performed:

1) create a directory in which you to store this release of the finsim software.

2) put in the newly created directory finflot.tgz, install.csh and finsim.tar.gz.

3) chmod 777 install.csh

4) install.csh

i) answer y to the first question provided that indeed you stored the license file along with rlm and fintronic and you have started rlm. You start rlm simply by invoking it.

ii) answer y to the second question provided that that is what you want.

5) gunzip finflot.tgz

6) tar-xvf finflot.tar

7) source <newly created dir>/env.script.csh

Note:

In the directory <newly_created_dir>/misc/demo contains different categories of examples.

The subdirectory demo_vp contains FinSimMath examples.

In the subdirectory demo_vp/filter one can find many examples similar to the one discussed in http://www.fintronic.com/finfilter.html. These are examples of IP generation taking advantage of the unique features of FinSimMath.

2.2 FinSim directory structure

The following subdirectories are created during the installation of -FinSim:

bin

Directory containing FinSim executable binaries.

lib

Directory containing FinSim runtime libraries.

obj

Directory containing FinSim object files.

env

Directory containing the FinSim environment script.

include

Directory containing FinSim header files.

demo

Directory containing FinSim demo designs.

2.3 Super-FinSim environment variables

Super-FinSim utilizes several environment variables to define its working environment. They are summarized in the table below as they would appear in a Linux environment.

FINTRONIC

Defines the root directory of FinSim,

FIN_OBJECT_PATH

Defines the location FinSim object files. Its default value is $FINTRONIC/obj/<compiler>.

FIN_INCLUDE_PATH

Defines the location of FinSim header files. Its default value is $FINTRONIC/include.

FIN_LIBRARY_PATH

Defines the location of FinSim library files. Its default value is $FINTRONIC/lib/<compiler>.

FINSYSPLIOBJ

Defines the location of the FinSim system PLI object file. Its default value is $FIN_OBJECT_PATH/verisim.o

FINTEMPDIR

Defines the working directory of a design relative to the directory in which the design is simulated. Most FinSim generated files are stored in the working directory. Its default value is fintemp.

3.0 How to use the compiler

3.1 Operations performed by the compiler

The compiler performs the following operations:

1.Finds syntactic and semantic errors in the design.

2.Generates the code necessary to configure and to program the simulation engine.

3.Generates elaboration data files to build the simulator data structures.

4.Generates a design file used to build the simulator.

5.Optionally generates debugging information for a source level debugger or source profiler.

6.Optionally writes the Intermediate Format corresponding to the design on disk.

3.2 Invoking the Verilog Compiler

The FinSim Verilog compiler has a fast and robust analyzer with an extensive error checking and recovery mechanism. In addition, the analyzer can optionally generate a number of warning messages flagging potential design errors such as accessing an array element out of bounds.

The FinSim Verilog compiler is invoked as follows:

finvc <option or source file> [<option or source file> ...]

Command files are also supported to facilitate invocation.

The desired simulation mode of FinSim, whether it be compiled, interpreted or a mixture of compiled and interpreted must be specified at compilation time. If no options are specified, FinSim will attempt to simulate the entire design in compiled mode if a license for compiled simulation is found. If not, the design will be simulated in the interpreted mode.

All compiler messages are stored in the log file 'finvc.log'.

3.2.1 Verilog Compiler Options

3.2.1.1 Library search control options

-ld <dir>: search for undeclared modules in the specified library directory

-y <dir>: same as -ld <dir>

-lf <file>: search for undeclared modules in the specified library file

-v <file>: same as -lf <file>

-le <str>: use the specified string as the file extension when searching for undeclared modules in library directories

+libext+<str>: same as -le <str>

-lsm <str>: the specified string describes the library search mechanism to use, it must be one of d (default), o (order) or r (rescan)

+liborder: same as -lsm o

+librescan: same as -lsm r

3.2.1.2 Include file search control options

-id <dir>: search for include files in the specified include directory

+incdir+<dir>: same as -id <dir>

3.2.1.3 Control file options

-cf <file>: take command line information from the specified command file

-f <file>: same as -cf <file>

3.2.1.4 Macro definition options

-dm <name>[=<value>]: define a macro with name <name> and value <value>

+define+<name>[=<value>]: same as -dm <name>[=<value>]

3.2.1.5 Warning suppression options

-nowarn <warning number> : suppress warning with the specified number

-imtm : suppress warning messages about inconsistent min:typ:max expressions that can be evaluated at compile time (e.g. 5:4:3)

-rmmo: suppress warning messages about range mismatches in mode and object declarations (e.g. input [0:7] a; wire [1:8] a;)

-fwrv: suppress warning messages about functions without a return value i.e. functions that do not assign to a register that has the same name as the function

-nprc: suppress warning messages about non positive replication counts in replicated concatenation expressions (e.g. {0 {r}}, {-1 {1'b1}})

-sav: suppress warning messages about slicing of supposedly atomic vectors (e.g. integer i; time t; initial i[10:9] = t[5:4];)

-cpd: suppress warning messages about cyclic parameter dependencies that may be created when parameters are overridden

-pot: suppress warning messages about parameters whose value is overridden two or more times

-ucp: suppress warning messages about ports that are left unconnected module/primitive instantiations

-aoi: suppress warning messages about modules that assign to their own input(s)

-mapl: suppress warning messages about ports that appear multiple times in the port list

-ues: suppress warning messages about unknown escape sequences encountered in strings

-rcel: suppress warning messages about case expressions or case labels that are of type real

-bss: suppress warning messages about bad strength specifications (e.g. supply0 (pull0, pull1) vcc = 1'b1;)

-bpsu: suppress warning messages about bad usages of parameters or specparams (e.g. using parameters within specify blocks, using specparams outside specify blocks, overriding of specparams

-swlv: suppress warning messages about specparams with list values (e.g. specparam s = (0, 1, 2, 3, 4, 5);), only the first value (0) is significant

-rdm: suppress warning messages about redefinitions of macros

-umnd: suppress warning messages about undefining of macros that are not defined

-bmn: suppress warning messages bad macro names in the command line (e.g +define+@#!!!++)

-icd: suppress warning messages about incorrect usage of the 'celldefine directive (e.g. nested 'celldefines, unmatched 'celldefines/'endcelldefines)

-ip: suppress warning messages about incorrect usage of the 'protect directive (e.g. unmatched 'protect/'endprotects)

-eti: suppress warning messages about extraneous tokens placed after a 'include "<file>" directive

-cee: suppress warning messages about constant event expressions (e.g. always @(25) a = b;)

-nseee: suppress warning messages about non scalar event expressions with edge specifiers (e.g. @(posedge a[0:7]))

-tne: suppress warning messages about triggered objects that are not events (e.g. real r; initial -> r;)

-tmpi: suppress warning messages about primitives with too many inputs

-bpte: suppress warning messages about bad entries in a primitive table (e.g. ? ? : 0)

-dreni: suppress warning messages about data or reference events in timing checks that are not module inputs

-wmiopp: suppress warning messages about width mismatches between inputs and outputs in a parallel path specification (e.g. i[0] => b[0:100] = 10;)

-wmodsep: suppress warning messages about width mismatches between the output and data source in an edge sensitive path specification (e.g. (posedge i[0] => (o[0] : c[0:2])) = 10;)

-nnsr: suppress warning messages about notify expressions in timing checks that are not scalar registers

-ustf: suppress warning messages about unknown system tasks/functions

-stfap: suppress warning messages about system tasks/function arguments that may cause problems

-ind: suppress warning messages about implicit net declarations

-bcwts: suppress warning messages about width specifications in based constants that are too small to hold the value of the constant (e.g. 2'hff)

-scw32: suppress warning messages about shift count expressions whose width is greater than 32

-aiw32: suppress warning messages about array index expressions whose width is greater than 32

-rcw32: suppress warning messages about repeat count expressions in repeat statements whose width is greater than 32

-mccw32: suppress warning messages about multiple concatenation expressions in which the width of the sub expression representing the replication count is greater than 32

-mcdw32: suppress warning messages about expressions representing multi channel descriptors whose width is greater than 32

-dw64: suppress warning messages about delay expressions whose width is greater than 64

-ncd: suppress warning messages about non constant delay expressions

-rd: suppress warning messages about delay expressions of type real

-nd: suppress warning messages about negative delay expressions

-xzd: suppress warning messages about delay expressions that contain x's or z's

-tmarg: suppress warning messages about type mismatches between actual and formal arguments of tasks/functions

-wmarg: suppress warning messages about width mismatches between actual and formal arguments of tasks/functions

-twmarg: same as "-tmarg -wmarg"

-tmass: suppress warning messages about type mismatches between the lhs and rhs of assignments

-wmass: suppress warning messages about width mismatches between the lhs and rhs of assignments

-twmass: same as "-tmass -wmass"

-tmgate: suppress warning messages about type mismatches between gate terminals and the expressions connected to them

-wmgate: suppress warning messages about width mismatches between gate terminals and the expressions connected to them

-twmgate: same as “-tmgate -wmgate”

-tmport: suppress warning messages about type mismatches between module/primitive ports and the expressions connected to them

-wmport: suppress warning messages about width mismatches between module/primitive ports and the expressions connected to them

-twmport: same as “-tmport -wmport”

-tmcel: suppress warning messages about type mismatches between case expressions and case labels

-wmcel: suppress warning messages about width mismatches between case expressions and case labels

-twmcel: same as “-tmcel -wmcel”

-tmexp: suppress warning messages about type mismatches (leading to type conversions) between operands of the following expressions (+, -, *, /, , <, <=, >, >=, ==, != , ===, !==, &, !, ^, ^~, ?:

-wmexp: suppress warning messages about width mismatches between operands of the following expressions (+, -, *, /, , <, <=, >, >=, ==, !=, ===, !==, &, !, ^, ^~, ?:

-twmexp: same as “-tmexp -wmexp”

-tm: same as “-tmarg -twmass -tmgate -tmport -tmcel -tmexp”

-wm: same as “-wmarg -wmass -wmgate -wmport -wmcel -wmexp”

-twm: same as “-twmarg -twmass -twmgate -twmport -twmcel -twmexp”

-eswis: generate warning messages about event statements with incomplete sensitivities e.g (@(a or b) d = a + b + c;), these messages are suppressed by default

-a: suppress all warning messages

3.2.1.6 Source encryption options

-encrypt: encrypt all code within ‘protect and ‘endprotect directives, an auto encryption option will also be added soon

+protect: same as -encrypt

3.2.1.7 Delay component selection options

-min: use the min component in min:typ:max expressions

+mindelays: same as -min

-typ: use the typ component in min:typ:max expressions, this is also the default

+typdelays: same as -typ

-max: use the max component in min:typ:max expressions

+maxdelays: same as -max

3.2.1.8 Delay mode selection options

+delay_mode_zero: use zero delay mode

+delay_mode_unit: use unit delay mode

+delay_mode_path: use path delay mode

+delay_mode_distributed : use distributed delay mode

3.2.1.9 License related options

-lic <file>: read license keys from the specified file rather than the one specified in the environment variable FIN_LICENSE_PATH or LM_LICENSE_FILE

-lic_verbose: run the license manager in verbose mode, useful for debugging license problems

-lic_type <100K|50K|25K|2K> : if you have licenses for different types of simulators and want to get a specific one

Interpretation/Compilation options

-comm <mod>: compile the specified module

March 1, 2012 16

-intm <mod>: interpret the specified module

-comf <file>: compile modules read from the specified file

-intf <file>: interpret modules read from the specified file

-comd <dir>: compile modules read from any file in the specified directory

-intd <dir>: interpret modules read from any file in the specified directory

-dsm <mode>: sets the default simulation mode to compile (-dsm com) or interpret (-dsm int), -dsm com is the default

-sysc <file>: considers that file contains SystemC modules. Some of these modules may be instantiated in Verilog modules. Multiple such options may be used in the same finvc invocation.

-tp<module_name>: specifies a top-level module. If one top-level module is specified usually it is better to specify all top-level modules. Multiple such options may be used in the same finvc invocation.This option is mandatory in case generate statements occur in the Verilog code.

-irc: incrementally recompile the design, this option should be used when small changes are made to a design in which most or all modules are compiled, finvc determines the modules affected by the changes and interprets them to bypass the C compiler and linker

3.2.1.10 Compiler output options

-td <dir>: place generated files required for simulation in the specified directory, the default is ./fintemp

-pp: pretty print (decompile) the source (in file pp.out)

-d: same as -pp

-stb: print the symbol table (in file stb.out)

-ifb: print the intermediate format in binary (in file ifb.out)

-ifa: print the intermediate format in ascii (in file ifa.out)

3.2.1.11 Debugging options

+finvcc: generate information for Fintronic's code coverage tool

+vtdbg: generate information for Veritool's source level debugger

3.2.1.12 Optimization related options

-ol <0-11>: use the specified integer as the optimization level, a higher level indicates that more optimizations will be performed, the default is 1 (many optimizations are performed at level 1)

+fin_g2p_max_nd_inp+<num_inputs> : num_inputs may be between 7 and 12. The higher the number the more memory will be used, but the faster the simulaton may run. The simulation may also run slower in case the larger memory size affects the cache utilization.

-acc: use the acceleration algorithm in the simulator

-noacc: do not use the acceleration algorithm in the simulator

+symb_eval: necessary in order to process evaluation of symbolic expressions

+symb_proc: necessary in order to process symbolic transformations, such as $Dif, $Int, $Lap, $ILap

+move_TF_to_glbl: moves, if possible, tasks and functions to a global module named Fin$$Glbl

+no_inl_func: prevents inlining of functions

+inl_ct_func: inlines constant function calls, i.e. calls which will always produce the same result

+caxl: accelerate continuous assignments

+notimingchecks: ignore timing checks

+no_notifier: ignore notify registers in timing checks

+fin_no_ecs: do not use Fintronic's Enhanced Cycle Simulation, usage of this option will increase simulation time

+fullaccess: keep detailed information on all declared signals; this option might increase simulation time and memory consumption

-fastgate: use the fast gate algorithm in the simulator

+no_plusargs_substitution: does not optimize calls to $test$plusargs with constant string arguments at compile time allowing users to run the simulator multiple times with different plus args without having to recompile and build the simulator for each run, usage of this option will increase simulation time

3.2.1.13 Other options

-pli: generate pli information even if the user's source code does not have pli calls (needed for instance when used with Intergraph's Veriscope tool)

-des <str>: use the specified string as the name of the design

-ao: perform syntax and semantic analysis only, do not generate code and data for simulation

-c: same as -ao

-uc: convert all identifiers to upper case

-u: same as -uc

-ptab <file>: get information about pli tasks/functions from the specified ascii file instead of having to link user's pli object files to obtain this information

-nttfsm: reject constructs not supported by NTT's FSM tool

-log <file>: use the specified file as log file instead of the default finvc.log

-silent: suppress messages about whic source/library files are currently processed

-help: display all options

3.2.2 Precedence order for simulation mode options

The Verilog compiler determines the simulation mode of a module (compiled or interpreted) based on the invocation options '-comm <mod>', '-intm <mod>', '-comf <file>', '-intf <file>', '-comd <dir>', '-intd <dir>' and '-dsm <mode>'. The precedence order of these options are:

1.-comm <mod>, -intm <mod>

2.-comf <file>, -intf <file>

3.-comd <dir>, -intd <dir>

4.-dsm <mode>

For example, if the user types the following:

finvc -intf test.v -comm test

and the module ‘test’ is defined in the file ‘test.v’, then all modules in file ‘test.v’ will be interpreted except ‘test’ which will be compiled.

3.3 Files generated by the Verilog compiler finvc

The Verilog compiler generates ‘C’ code files, interpretation data files and elaboration data files for each design. The ‘C’ code, interpretation data and elaboration data files have the extension ‘.c’, ‘.i’ and ‘.edf’ respectively. The compiler also generates some other files needed at run time.

All files generated by the Verilog compiler are stored in the working directory defined in the environment variable FINTEMPDIR, the directory specified with the -td option or the directory fintemp created in the local directory.

3.4 Incremental recompilation

Incremental recompilation can be used when small changes are made to a design in which most of the modules are compiled. The compiler determines the modules affected by the changes and interprets them to bypass the C compiler and linker. The first time around, finvc should be invoked in the usual way. For subsequent runs, the option ‘-irc’ should be added to the other finvc options to take advantage of the incremental recompilation feature.

3.5 Separate compilation

Super-FinSim provides the facility of separately compiling parts of the Verilog hierarchy. This pre-compiled hierarchy can then be mixed with other Verilog sources to build a new design. This facility is extremely helpful for users who want to ship their IP to their customers but do not want them to access the Verilog source. Not only will the access to the source be denied using the regular `protect/`endprotect mechanism but the IP provider will only have to ship binary files which would make it virtually impossible to re-create the original Verilog code. Another useful application of separately compiled code is for users who add legacy code to their designs which has been tested and will not need to be modified.

3.5.1 Compiling a Verilog Design Hierarchy into object code for later reuse

A Verilog description can be separately compiled for later reuse by invoking the compiler, called finvc, with the special option +sepgen, followed by an invocation of finbuild, as follows:

#finvc +sepgen+<mymodel> <other options>

#finbuild

where <mymodel> is a name given by the user to this design. One of the uses of <mymodel> is to make any symbols in the separately compiled design not clash with similar symbols in the final design (which

instantiates the separately compiled design). After running these two steps, the temporary directory (fintemp by default) will contain the compiled `C' files, the interpretation data files, the elaboration data files as well as all other files needed for simulating this hierarchy. In addition it will contain a Verilog interface file called interface.v. This file contains the shell for the exported modules in the separately compiled design. Any of these modules can be instantiated in the final design.

finvc also assumes that everything in the separately compiled design except the interface for the top level module is protected. This assumption can be relaxed with the option (+fin_sep_unprotect).

3.5.2 Using a separately compiled hierarchy

In order to use a separately compiled Verilog design hierarchy as part of a new Verilog design hierarchy one must invoke finvc with the special option +sepuse, followed by finbuild and the invocation of the executable simulator, as follows:

finvc +sepuse+<directory name> <other options>

finbuild

TOP.sim

The directory name is the directory where all the files were generated in the compilation step. More than one such +sepuse options can be specified. In this case finvc treats the file interface.v in the separately compiled directory as a library file, so that any module that is used in the final design and cannot be found is searched in this file.

3.5.3 Restrictions

Restrictions on the separately compiled code

A module in the separately compiled design can be instantiated outside of the separately compiled design only if:

a) there are no external references anywhere in the separately compiled design (things like a.b.c)

b) none of its parameters or the parameters of the modules instantiated in the hierarchy below it are overwritten with more than one value

Restrictions on the code instantiating a separately compiled module

A design can instantiate modules from the separately compiled design if:

a) it does not overwrite the parameters of the separately compiled module

b) it doesn't make external references into the separately compiled design

c) its time precision is not finer than the time precision of the separately compiled design.

3.6 Calling user C tasks/functions in Super-FinSim without the PLI interface

FinSim allows the user to call functions written in the C language directly from within the Verilog code. The user has to provide one or more C header files with the prototypes of the C functions.

The arguments of C functions callable from FinSimMath can be real values (64 bits floating), characters (8 bits), short integers (16 bits), integers (32 bits), long integers (64 bits) pointers of the above mentioned types, and pointers of type simSIgnalPT corresponding to arrays (single or multi-dimensional) arguments passed to the FInSimMath call.

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 are generated by using + options: +Insert_dimension_info, +Insert_type_info, +Insert_view_info, +Insert_file_line.

The implicit arguments are:

a) number of original arguments (i.e. number of inputs plus 1, since there can be only one argument corresponding to the left hand side),

b) before each original argument the following implicit arguments are generated if the option +Insert_dimension_info is used.

i) number of dimensions

ii) for each dimension, the start index and the end index.

In case +Insert_type_info is used, the type of each original argument is provided as the first implicit argument associated to the given original argument. The type is of type int and provides the type of the array.

In case the +Insert_view_info is used, the view information of each original argument is inserted after the type information in case it exists and before the dimension related info.

The view information is of type long and provides a pointer to the indirection table for the given "view as" construct.

The 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) 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.

The semantics for calling C user functions and tasks are similar to the semantics for calling Verilog or PLI functions and tasks.

The header files providing the prototypes of the C functions are passed to finvc 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. The finc.h file is predefined and is available in the include directory, FINTRONIC/include

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.

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 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.

3.7 Using Mixed Verilog/SysytemC descriptions

3.7.1 Introduction

FinSim is integrated with the SystemC simulator. Please note that no representation is being hereby made with regards to the functionality or the quality of this simulator open source SystemC simulator.

Supported platforms: Linux 64 bit.

Supported versions of SystemC: SystemC 2.1

3.7.1 Instantiating SystemC modules in Verilog

Verilog modules that are empty and contain in their body:

(* foreign=SystemC *)

are considered to be SystemC modules.

3.7.2 Invoking finvc when there are SystemC modules involved

The option -sysc <file> must be provided to finvc, where <file> contains the SystemC description of all SystemC modules instantiated in the current Verilog simulation.

3.7.3 Rules to be observed by SystemC modules instantiated in Verilog:

i) The size and order of the ports of the Verilog prototype (description of ports) must match the size and order of the SystemC ports. The name of the ports in the Verilog prototype do not matter, but it is preferable for documentation purposes to have them being the same.

ii) SystemC modules shall have one bit ports of class sc_logic only. Class sc_bit is not supported at this time and class sc_lv shall not be used for vectors of one bit.

iii) SystemC modules shall have ports that are vectors of strictly more than one bit of class sc_lv only. Class sc_bv is not supported at this time.

iv) The size of SystemC port vectors shall not be more than 80,000 bits per port.

v) The Verilog prototype of SystemC modules shall have vectors declared in ascending order only.

3.7.4 Invoking TOP.sim or the name of the simulator

All the options available can be passed to the simulator. They will affect only Verilog modules. One can provide tracing information to the SystemC simulator by editing the file fintemp/mixed_sc_gen.cpp and inserting tracing-related calls in sc_main.

Note: An example of a simulation involving Verilog modules that instatiate SystemC modules is provided in the distribution under demo/demo_systemc.

3.7.5 Translating FinSimMath to SystemC

The translator handles one single FinSimMath/Verilog module. For complicated expressions, the translator translates them into a function call with an intuitive name for synthesis purposes and into co-simulation SystemC/FinSim for simulation purposes.

The translator is invoked by finvc +gen_sysc file_name.v and produces three files: fin_sysc.cpp, fin_sysc_top.h, and fin_sysc.v. Note that fin_sysc.cpp include fin_sysc_top.h. To run the generated using FimSim one must perform the following commands:

Finvc +FM –a –systemc fin_sysc.cpp fin_sysc.v

Finbuild

TOP.sim.

4.0 How to build the simulator

4.1 Operations performed by the simulation builder

Once the design has compiled successfully, a simulator can be built by invoking the simulation builder, finbuild. Simulators can be built to run stand-alone or integrated into a waveform display interface.

The process of building the simulator includes the compilation of the generated 'C' file(s), linking the object file(s) with the simulation kernel, appropriate waveform interface and PLI object files if any. Compilation of 'C' files can be performed on a single machine (default) or across a network of homogenous machines.

Compilation on a single machine can be performed sequentially or concurrently. In the latter case, the user has the option to specify how many compilation tasks can be launched at a time. This can speed up compilation time dramatically. However, more memory is utilized since there are more processes running. Concurrent compilation may not be faster than sequential compilation when the system does not have enough memory and/or there are too many processes running which overload the system and increase disk swapping activity.

Network compilation allows the simulation builder to spawn compilation tasks across many homogenous machines on the network. It is important that all machines specified have the same architecture. In the default mode, the simulation builder does not attempt to check whether the specified machine is valid or whether it has the same architecture as the others. To enable checking of the specified machines used for network compilation, one must specify the option '-hostchk'. This option must be specified before any machine name is specified.

Once the simulator is built, it is stored under the name:

<design name>.sim (for UNIX)

<design name>.exe (for Windows)

The simulation builder obtains the design name from either the default name "TOP" or from the user invocation option '-des <name>'. Its name is then compared with the design name stored in the design file. If they are the same, the simulator can be built. If one changes the default design name when invoking the Verilog compiler, it is also necessary to specify that design name when invoking the simulation builder. If the default design name is always used when invoking the compiler, then it is not necessary to specify the design name when invoking the simulation builder.

All messages generated by the simulation builder are stored in the log file 'finbuild.log'.

All messages generated by the host C compiler are stored in the log file 'compile.log'.

4.2 Invoking the simulation builder

The FinSim simulation builder, finbuild, is invoked as follows:

finbuild <build options>

4.2.1 Simulation builder options

-help: Display invocation options.

-pch: Use precompiled header files.

-static: Use the static library of FinSim.

-verbose: Display compilation command string.

-clean: Remove generated 'C' and object files.

-des <name>: Specify the name of the design.

-td <dir>: Specify the working directory.

-relpath: Use relative paths for include directories, object files etc.

-lic <file>: Process the licensing from a specified license file.

-lic_type <100K|50K|25K|2K> : Use the license for the specified product.

-driver: Generate driver file.

-vt: Link simulator with Veritool's PLI interface.

-H<host>: Add remote host for network compilation.

-C<compiler>: Specify the name of the compiler.

-L<linker>: Specify the name of the linker.

-O<level>: Specify optimization level for C compilation.

-hostchk: Check the validity of the hosts entered.

-concurrent <num>: Compile all modules concurrently (Unix only)

-seq: Compile all modules sequentially.

+<option>: Specify additional compilation option.

-um <name>: Use specified name instead of main(). For users who supply their own main() function.

4.2.2 Removing system files

Once the simulator is built, one can remove all the 'C' and object files by specifying the invocation option '-clean'. These files are not needed to run the simulation but are required if the incremental

recompilation feature will be used subsequently. One must never delete the elaboration data files prior to simulation because they are required by the simulator.

5.0 Building the PLI interface in FinSim

In order to support PLI, the compiler needs to obtain information about the user tasks and functions. Since this information is provided in the standard PLI table veriusertfs, a mechanism is required to link it into the compiler.

FinSim provides two methods of doing this. The first method is the use of a Fintronic PLI table. This is an ASCII text file containing information about the user PLI tasks and functions. This table can be created manually or it can be generated automatically. The second method is to build a custom compiler. This is achieved by linking all the user PLI object files with a custom compiler library.

A sample 'C' interface file, 'veriuser.c' is provided in the directory $FINTRONIC/include.

5.1 Using the Fintronic PLI table

The Fintronic PLI table can be created manually or automatically. The table consists of one entry for each PLI task or function. Each entry has the following format:

<NAME> <TYPE>

'NAME' is the name of the routine used in the Verilog source, e.g. $myplifunc

'TYPE' can be one of the following

task: Specify a user defined task.

func_real: Specify a user defined function returning a type real.

func_sized_[dd]: Specify a user defined function returning a value whose width is [dd], e.g. func_sized_32

Verilog style comments may be inserted anywhere within the table.

5.1.1 Creating the table manually

To construct the Fintronic PLI table manually from a standard PLI table, veriusertfs, perform the following steps:

1.Create equivalent entries in the Fintronic PLI table for all the entries found in the standard PLI table except for the very last one.

2.The NAME field in the Fintronic PLI table is obtained from the 7th element in the standard PLI table (tfname).

3.The TYPE file in the Fintronic PLI table is obtained from the 1st element in the standard PLI table (type) with the following conversion:

usertask -> task

userfunction -> func_sized_[dd]

userrealfunction -> func_real

Given a PLI definition below:

```
#include "veriuser.h"
#include "acc_user.h"
static int myfuncsize()
{
return(3);
}
static int myfunc()
{
io_printf("$my_func is called\n");
tf_putp(0, 1);
}
static int mytask()
{
io_printf("mytask is invoked.\n");
}
s_tfcell veriusertfs[] = {
/*
******************* Entry definition ************************
************************************************************
{type, data, checktf, sizetf, calltf, misctf, tfname, forwref},
*/
{usertask, 0, 0, 0, mytask, 0, "$mytask", 0},
{userfunction, 0, 0, myfuncsize, myfunc, 0, "$myfunc", 0},
/* all entry must be entered before this line */
{0, 0, 0, 0, 0, 0, 0, 0} /* this must be the last entry */
```


The Fintronic PLI table is as follows:

$mytask task

$myfunc func_sized_3

5.1.2 Creating the table automatically

The Fintronic PLI table can be built automatically by performing the following steps:

1. Define PLI object files in the environment variable FINUSERPLIOBJ. For example, if PLI routines are stored in the file veriuser.c and mypli.c, then the environment variable FINUSERPLIOBJ must be defined as follows:

setenv FINUSERPLIOBJ “veriuser.o mypli.o”

2. Define PLI static library files if any in the environment variable FINUSERPLILIB.

3. Define PLI dynamic libraries if any in the environment variable FINUSERPLIDLL.

4. Build the custom table generator

make -f $FINTRONIC/include/MakeTAB

5. Run the table generator

finvtab > <table>

If the PLI definition file, finpli.mak is used, step 4 must be run as follows:

make -f $FINTRONIC/include/MakeTABI

5.2 Building a custom compiler

The custom compiler can be built in 2 ways. The first method is the use of environment variables FINUSERPLIOBJ, FINUSERPLILIB, and FINUSERPLIDLL as shown below:

1.Define PLI object files in the environment variable FINUSERPLIOBJ. For example, if PLI routines are stored in the file veriuser.c and mypli.c, then the environment variable FINUSERPLIOBJ must be defined as follows:

setenv FINUSERPLIOBJ “veriuser.o mypli.o”

2.Define PLI static library files if any in the environment variable FINUSERPLILIB.

3.Define PLI dynamic libraries if any in the environment variable FINUSERPLIDLL.

4.Build the custom compiler

make -f $FINTRONIC/include/MakePLI

The second method uses a PLI definition file named ‘finpli.mak’. This file contains the definitions of FINUSERPLIOBJ, FINUSERPLILIB, and FINUSERPLIDLL similar to method 1. In addition, dependencies among the PLI source code and header files can be specified.

The custom compiler can then be built as follows:

make -f $FINTRONIC/include/MakePLII

The only difference between the makefile, ‘MakePLI’ and ‘MakePLII’ is that the latter includes the PLI definition file. The default rule to compile the PLI source code is defined in the makefile itself. If the PLI definition file is used, it is possible for one to override the default compilation rule.

The custom compiler is named ‘vc’ by default. This name can be changed by either modifying the makefile itself or by overriding the default compiler name on the command line as shown below:

make -f $FINTRONIC/include/MakePLI FINVC=<compiler name>

Note: It is not necessary to rebuild the custom compiler if only the body of the PLI code is changed. However, it is necessary to rebuild the custom compiler if the functional interface is changed such as the width of the value returned by a PLI function.

5.3 Building the simulator with PLI

Whenever PLI is utilized in the design, the simulation builder will obtain the information of all the PLI object

files from either the PLI definition file or the environment variables FINUSERPLIOBJ, FINUSERPLILIB and FINUSERPLIDLL. If the PLI definition file is found in the current path, the simulation builder will always use it first even if the PLI environment variables are also defined.

For an example on how to link in a PLI application, run the example in the demo_pli directory in the Super-FinSim distribution.

5.4 Using multiple veriusertfs tables

Super-FinSim allows the user to link in multiple PLI applications eachwith its own veriusertfs table without having to manually merge the tables. To do so, please follow these steps:

1. Link your PLI application into a shared library.

For gcc, you can do this by passing the -shared option:

gcc -shared pli1.o <other pli objects/libraries> -o pli.so

2. Call finvc with as many tab files as needed. Please note that you may choose to create only one tab file if you want but this is not necessary:

finvc -ptab pli1.tab -ptab pli2.tab <other -ptab options> ...

3. Set the variable FINUSERPLIDLL to include all the shared PLI libraries.

If you are setting this variable via the enviroment:

setenv FINUSERPLIDLL "pli1.so pli2.so <other shared PLI libraries>"

If you are using the finpli.mak file:

FINUSERPLIDLL = pli1.so pli2.so <other shared PLI libraries>

4. Run finbuild with your regular options:

finbuild <your options>

5. On Unix/Linux systems set the environment variable LD_LIBRARY_PATH to the paths where your shared PLI libraries reside:

setenv LD_LIBRARY_PATH <path to pli1.so>:<path to pli2.so>:<path to other shared PLI libraries>:$LD_LIBRARY_PATH

6. Run TOP.sim (or whatever the name for the simulator you chose) with the extra +veriuser+<name of shared PLI library> options:

TOP.sim +veriuser+pli1.so +veriuser+pli2.so <+veriuser+other shared PLI libraries> <other simulation options>

6.0 How to use the simulation engine

6.1 Operations performed by the simulation engine

The simulation engine performs two main operations:

1.Builds simulation objects: nets, registers, activities, etc.

2.Simulates a network of simulation objects that represents a particular design.

6.2 Invoking the simulator

The simulator is invoked as follows:

<design name>.sim <simulator options>

The simulator invocation options can be specified in any order. A few examples of simulator invocations are:

TOP.sim -i -nodriverchk

FDC.sim -r 5000 -script scriptfile

3.Simulates a network of simulation objects that represents a particular design.

4.Simulates a network of simulation objects that represents a particular design.

6.3 Simulator Options

-b: Build the simulation data structure only.

-r <time>: Run simulator for a specified time.

-i: Run simulator in interactive mode.

-delta <value>: Specify the maximum delta during a time cycle.

-deltastop: Interrupt the simulation if delta reached its maximum value.

-tr: Trace all signals.

-t <traceFile>: Trace signal specified in Fintronic Trace File.

-wave: Invoke real time waveform display.

-log <logFile>: Create simulation log file.

-key <keyFile>: Create simulation key file.

-script <scriptFile>: Read interactive command from a file instead of standard input.

-nolibcell: Disable cell instance from library.

-notimechk: Disable timing check.

-notimechkm <module>: Disable timing check on a specified module.

-notimechki <instance>: Disable timing check on a specified instance.

-nowarning: Suppress warning messages.

-nofillmemwarning: Suppress warning messages regarding over or underfilled user memories

-noannotate: Disable back-annotation through PLI or SDF.

-nopath_cond: Ignore the SDPD condition.

-path_edge: Enable the edge condition.

-nopulsemsg: Disable warning messages about pulse control errors.

-pulse_reject: Global path pulse control (0 - 100) : reject limit.

-pulse_error: Global path pulse control (0 - 100) : error limit.

-path_cond_edge: Do not ignore the edge condition if both SDPD and edge conditions are provided.

-fastgate: Speed up gate level simulation.

-fastca: Speed up continuous assignment.

-nodriverchk: Disable check for net having no driver to it

-ncols <num>: Specify the number of columns, <num> per line to be printed in waveform.

-sdfmsglevel <level>: Specify the level of sdf error message. The default value is 0.

-vectored_net: By default, do not expand a vector net.

-scalared_net: By default, always expand a vector net.

-verbose: Display debugging information about the simulator.

-msgx: Display simulation messages in the compatibility mode.

-c: OVIsim compatibility mode.

-acc: Simulate in accelerated mode.

-td <directory>: Specify the working directory of the simulator.

-lic <file>: Process the licensing from the specified license file.

-lic_type <100K|50K|25K|2K>: Use the license for the specified product.

-vmemd: Specify the directory in which the virtual memory file should be created.

-dumpd: Specify the directory in which the dump file should be created.

-help: Display all invocation options.

6.4 Simulation Modes

The simulator can be executed in either batch or interactive mode. If both simulation modes are specified, the interactive mode will be selected.

The simulator must never be invoked using the UNIX input redirection such as below:

TOP.sim < InputFile

Doing so will cause the simulator prompt to be printed in an infinite loop. Instead, input can be specified using a script file as described in the next section.

6.4.1 Batch Simulation

Batch simulation allows one to simulate a design without user interaction. The maximum duration of the simulation may be specified with the invocation option '-r <time>'. If the batch simulation time and the interactive mode option '-i' are not specified, the simulator will also run in batch mode. When this occurs, the simulator will run until the design has stabilized.

6.4.2 Interactive Simulation

Interactive simulation allows the user to interact with the simulator under a command shell. The interface consists of a set of interactive commands which perform various functions such as running the simulator, setting breakpoints, displaying or modifying signal's value, etc. All the available interactive commands are described in Chapter 7 on page 34.

The simulator is started in the interactive mode only if the invocation option '-i' is specified. It has higher precedence over batch mode.

6.4.3 Using script files

For convenience, script files are supported to provide an alternate method of entering interactive commands. A script file can be specified whenever the simulator is invoked in the interactive mode. Since all interactive commands are saved in a simulation key file, 'finsim.key', it is possible to reproduce the last simulation run very easily.

6.4.4 The Save and Restart feature in Super-FinSim.

Super FinSim can save the state of the simulator and restart the simulator later on. Both save and restart operations are performed at very high speed (e.g. 5-6 seconds for saving and 2-3 seconds forrestarting simulation images of 256MB). This feature can be used both to recover after a hardware failure, or to bring the simulation in a certain state, save it and then restart it on many machines simultaneously in order to perform various tests with different stimuli starting from the desired state. This way one does

not need to repeat the part where the simulation is brought in the desired state. As an example, one may wish to save the state after the boot cycle is completed and then restart it in order to perform the various tests.

6.4.4.1 Saving a simulation.

FinSim allows the user to save the state of a simulation in one of two ways. For both cases the user first edits a command file with all the command line options for the simulator and runs the simulation using "-cf <file>"

a. Using the $save("suffix") system task.

The user inserts in the Verilog design at the desired time(s) calls to the system task $save. The simulation is saved in the file(s) finstate.<suffix> on Linux.

b. Using the interactive command save <suffix>

In the interactive mode the user issues the command save <suffix>. The simulation is saved in the file(s) finstate.<suffix> on Linux and <name of original simulator>.<suffix> on Solaris. To get to the interactive mode, one can either start the simulation in the interactive mode from the beginning with the "-i" option (specified in the command file), or can insert a $stop in the Verilog source

code or can type a CTRL-C while a batch simulation is running.

The following notes apply to saved simulations regardless of how they were saved except where noted.

The simulation state is always saved at the end of the current simulation time. If the design has PLI all misctf routines are invoked with reason 'reason_save'. All of the user's PLI data structures in memory are saved and restored automatically by FinSim however, the user is responsible for saving the state of any file/socket opened using PLI when the misctf routine is called with reason 'reason_save'.

In the interactive mode, if the <suffix> is omitted, the state will be saved in finstate.sav on Linux and <name of original simulator>.sav on Solaris. The system task $save requires a string as its only argument.

6.4.4.2 Restarting a saved simulation.

To restart a saved simulation, the user has to set the environment variable FIN_RESTART_ARGS to "+fin_restart+<suffix> -cf <file>" and call TOP.sim:

for csh/tcsh

#setenv FIN_RESTART_ARGS "+fin_restart+<suffix> -cf <file>"

#TOP.sim

for sh/bash

#set FIN_RESTART_ARGS="+fin_restart+<suffix> -cf <file>"

#export FIN_RESTART_ARGS

#TOP.sim

<suffix> is the suffix of the saved simulation and <file> contains all command line options for TOP.sim. FinSim allows the user to provide extra plus arguments when the simulation is restarted. This is useful for instance when the design is written such that it generates/applies different test vectors based on one or more plusargs provided at runtime. The user can save the state of the simulation when the initialization sequence is complete and restart the same saved simulation multiple times each with different plus arguments causing the test bench to generate/apply different testvectors for each run. Please note that in order for plus arguments to be evaluated at runtime, the design must be compiled with the option +no_plusargs_substitution to finvc. All plus arguments should be specified in <file>. All other arguments specified when the simulation is restarted will be ignored since the ones in the original run have already been evaluated and are therefore part of the saved image.

If the design has PLI all misctf routines are invoked with reason 'reason_restart'. All of the user's PLI data structures in memory are restored automatically by finsim, however the user is responsible for restoring the state of any file/socket opened using PLI in the initial run when his misctf routine is called with reason 'reason_restart'.

IMPORTANT: Please note that in order to run a new simulation, one has to unsetenv FIN_RESTART_ARGS to avoid restarting a saved one.

6.5 Starting a real time waveform display

Super-FinSim's simulator can be interfaced directly to a real time waveform display through a procedural waveform interface. While the simulator is running, value changes on traced signals are registered directly to the waveform display using inter-process communication.

6.6 Simulation output

In addition to the standard output, simulation results are stored in a log file whose default name is 'finsim.log'. The log file can be renamed with the option '-log <filename>' or by using the Verilog system task $log.

All interactive commands entered during the simulation are saved in a simulation key file whose default name is 'finsim.key'. The key file can be renamed with the option '-key <filename>' or by using the Verilog system task $key.

If the design uses SDF, a log file 'finsdf.log' is created to store the messages produced by the SDF compiler.

6.7 Interrupting the simulator

The simulator can be interrupted with 'Ctrl-C' or 'Ctrl-Z'. Once interrupted, the simulator enters the interrupted.

interactive mode. Most interactive commands can be executed in this mode except for 'run', 'monitor', 'force' and 'release'. The interactive command 'cont' can then be used to continue the simulator.

The simulator also enters the interrupted interactive mode whenever the Verilog system task $stop is executed.

To distinguish between the interactive mode and the interrupted interactive mode caused by Ctrl-C/Ctrl-Z or with the Verilog system task $stop, the simulation kernel displays the interactive prompt using the following convention:

Interactive Mode

Simulator prompt

interactive mode

scope>

interrupted interactive mode from Ctrl-<key>

scope[INTERRUPT]>

interrupted interactive mode from $stop

scope[STOP]>

6.8 Terminating the simulator

The simulator terminates when one of the following occurs:

1. The maximum simulation time is reached in batch mode.

2.The simulator has no more events to process.

3. The interactive command ‘quit’ or ‘$finish’ is entered in the interactive mode.

4. The system task $finish is executed from the Verilog source.

5. The user presses the keystroke ‘Ctrl-\’.

7.0 Super-FinSim Interactive commands

If the simulator is started in the interactive mode, a shell prompt is provided for entering interactive commands. All interactive commands must be entered in lower case. Arguments describing signal names must be entered using the appropriate case as specified in the source code. Other arguments can be entered in either lower or upper case.

An on-line help facility is available by entering ‘help’ at the interactive prompt.

7.1 List of interactive commands

$finish

Terminate the simulation

quit

Terminate the simulation.

build

Build the simulation data structures.

init

Build and initialize the simulation data structures.

run <time>

Simulate for the specified amount of time.

cont

Continue the simulation.

.

Same as cont

script <file>

Execute interactive commands from the specified script file.

pd

Display the simulation data structures.

readmemb <file> <memory>

Read binary data from a file into memory.

readmemh <file> <memory>

Read hexadecimal data from a file into memory.

info <signal>

Display information about a signal.

value <signal>

Display the value of a signal.

display(<format string>, signal)

March 1, 2012 41

display the value of a signal similar to $display.

force <signal> <dboh>

Force a signal permanently to a value

release <signal>

Deactivate a signal that was forced.

setenv <variable> <value>

Set the value of system environment variables.

printenv

Display the value of all system environment variables and the status of the simulator.

time

Display the current simulation time.

version

Display the version of the simulation kernel.

help <command>

Display an on-line help message for the specified interactive command.

cd <scope>

Change the current interactive scope.

ls

Display objects declared in the current scope.

plilist

Display all the registered user defined PLI functions/tasks

log [<file>]

Create a new log file or enable writing to an already open log file.

nolog

Disable writing to the log file.

key [<file>]

Create a new key file or enable writing to an already open key file.

nokey

Disable writing to the key file.

break <transition> <signal>

Create a break point on a signal value change. A signal transition can be one of the following: posedge, negedge, tox, toz, change.

break <mode> <time>

Create a time break point. Time break points can be either absolute or relative to the current time and they occur either at the beginning (abstimebf, reltimebf) or at the end of the time (abstimeaf or reltimeaf) cycle.

breaklist

Display list of current break points.

breakon [<break_point_number>...]

Enable the specified break points. If no argument is specified, all break points will be enabled.

breakoff [<break_point_number>...]

Disable the specified break points. If no argument is specified, all break points will be disabled.

breakclr [<break_point_number>...]

Remove the specified break points. If no argument is specified, all break points will be removed.

monitor <signal>...

Monitor the specified signals on the waveform display.

monitorall

Monitor all signals on the waveform display.

demonitor [<signal>...]

Terminate monitoring of specified signals on the waveform display.

demonitorall

Terminate monitoring of all signals on the waveform display.

monitoron [<signal>...]

Enable monitoring of all monitored signals on the waveform display

monitoroff [<signal>...]

Disable monitoring of all monitored signals on the waveform display.

history

Display command line history.

alias

Create or display command aliases.

readmem[bh] <file> <memory> [start_address [finish_address]]

Read values from the specified file into the specified memory. Start address and finish address are optional.

$dotask <name of task>

Execute a task defined in the Verilog source code

$<user PLI name>[(“argument string”)]

Execute a user PLI. The only argument allowed is a string.

save <suffix>

Saves the simulation state at the end of the current simulation time. The state is saved in one or more files each of which end with .<suffix>. If the suffix is not specified the string 'sav' is used as the suffix. If the design has PLI all misctf routines are invoked with reason 'reason_save'. All of the user's PLI data structures in memory are saved and restored automatically by finsim, however the user is responsible for saving the state of any file/socket opened using PLI when his misctf routine is called with reason 'reason_save'. Finsim permits the user to provide extra plus arguments when the simulation is restarted. This is useful because the design can be written so that it generates different test vectors based on one or more plus args provided at runtime. The user can save the state of the simulation when the initialization sequence is complete and restart the same saved simulation multiple times each with different plus arguments causing the test bench to generate different test vectors in each run.

7.2 Processing simulation data structures

This section describes the interactive commands that are used to process the simulation data structures.

7.2.1 Build

The interactive command ‘build’ is used to build the simulation data structures.

7.2.2 Init

The interactive command ‘init’ is used to initialize the simulation data structures. The simulation data structures will be built if they haven’t been built already.

7.3 Running the simulation

This section covers the interactive commands to run the simulator.

7.3.1 Run

The interactive command ‘run’ is used to run the simulator for a specific amount of time. The time unit associated with the value <time> is the smallest time precision in the design. Whenever ‘~’ is entered as the value of time, the simulator will run until the design stabilizes.

7.3.2 Cont

The interactive command ‘cont’ is used to resume simulation which may have been suspended by a user interrupt (Ctrl-C), execution of the system task $stop, or the encountering of a breakpoint.

7.4 Handling of simulation scope

When the simulator is interrupted, it displays the current scope. The default current scope when the simulation begins is the scope of the first root module.

7.4.1 Cd

The scope in the design hierarchy can be changed with the interactive command ‘cd’. Like its Unix counterpart, the symbol ‘.’ represents the current path. The symbol ‘..’ represents the parent path. The symbol ‘/’ when used by itself represents the root path. Otherwise, it is used as the hierarchy separator.

Currently, Super-FinSim only allows the scope to be changed one level up at a time. Thus the commands below are invalid:

> cd ../../

> cd ../abc

7.4.2 Ls

The interactive command ‘ls’ is used to display all objects declared in the current scope..

7.5 Querying of simulation objects

Object names in Super-FinSim are case-sensitive. Therefore, ‘abc’ and ‘ABC’ are considered to be two different objects. Super-FinSim allows the user to specify the name of a scope to commands that require signal names. Once located, the command is applied to all the signals declared in that scope.

For example, if an instance named ‘top.U00’ has two ports ‘A’ and ‘B’, one can display the value of these two ports as follows:

> value top.U00

This is equivalent as entering the following:

> value top.U00.A

> value top.U00.B

7.5.1 Info

The interactive command ‘info’ is used to display information about a signal, including its name, type, range, value, fanout etc.

7.5.2 Value

The interactive command ‘value’ is used to display the value of a signal. If the signal is not of type real, the value will be displayed in binary format. One can change the display format with the command ‘setenv’.

7.5.3 Force

The interactive command ‘force’ is used to permanently set a signal to a value during the simulation. The force command can be deactivated by executing the ‘release’ command.

The signal value entered in the interactive mode has the following DBOH format:

<width><base><value>

The first argument is a positive integer representing the width of the DBOH string. The second argument represents the base of the DBOH string. Valid DBOH bases are D, B, O, and H. The last argument represents the value of the DBOH string.

Here are some examples of valid DBOH string:

1B1

8HFE

10D25

7.5.4 Release

The interactive command ‘release’ is used to deactivate a signal that was forced.

7.6 Super-FinSim environment variables

The simulation kernel maintains a few simulation environment variables during the simulation. They are:

DBOH, ECHO

The simulation environment variable DBOH is used to specify the format used to display the value of a signal. The default format is binary (B|b). The user can specify other display formats including decimal (D|d), octal (O|o) or hexadecimal (H|h).

The simulation environment variable ECHO is used to determine whether the simulation output is displayed on the console or not. The possible values for ECHO are: {ON |OFF|on|off}, with ON being the default value.

7.6.1 Setenv

The interactive command ‘setenv’ is used to set Super-FinSim simulation environment variables. The syntax is follows:

setenv <ENVIRONMENT VARIABLE> <VALUE>

7.6.2 Printenv

The interactive command ‘printenv’ is used to display the value of all system environment variables as well as the simulator status.

7.7 Miscellaneous system facilities

Script

The interactive command ‘script’ is used to read and execute interactive commands from an ascii text file. There should be only one command per line in the script file. Since it is possible for one to invoke a script file within a script file, care must be taken to prevent recursive invocations.

Time

The interactive command ‘time’ displays the current simulation time.

$Finish

The interactive command ‘$finish’ is used to terminate the simulation. The user can also quit the simulation using the keystrokes ‘Ctrl-\’ or ‘Ctrl-D’.

Quit

Same as the interactive command ‘$finish’.

Log

The interactive command ‘log’ is used to create a new log file or enable writing to an already open log file. If a log file is currently open and the argument is not specified, that log file will be enabled for writing. Otherwise, the current log file is closed and a new log file will be created.

If there is no open log file and the argument is not specified, a new log file is created using the default name ‘finsim.log’. Otherwise, a new log file is created using the specified name.

Nolog

The interactive command ‘nolog’ is used to disable writing to the log file.

Key

The interactive command ‘key’ is used to create a new key file or enable writing to an already open key file. If a key file is currently open and the argument is not specified, that key file will be enabled for writing. Otherwise, the current key file is closed and a new key file will be created. If there is no open key file and the argument is not specified, a new key file is created using the default name ‘finsim.key’. Otherwise, a new key file is created using the specified name.

Nokey

The interactive command ‘nokey’ is used to disable writing to the key file.

Pd

The interactive command ‘pd’ is used to display simulation data structures.

7.8 Simulation Help Facility

Super-FinSim provides an on-line help facility similar to Unix man pages. This facility is provided through the interactive command ‘help’.

Help

The interactive command ‘help’ is used to display an on-line help message about an interactive command. The syntax is follows:

help [<command>]

If <command> is not specified, the list of all interactive commands will be displayed.

7.9 Command history

The interactive command ‘history’ is used to display the history of interactive commands. To execute the last command, enter the command ‘!!’. To execute the nth command, enter ‘!<n>’. When executing the command history, one can pass additional arguments if needed.

7.10 Command aliasing

Command aliases are used to create abbreviations of commonly used interactive commands. The interactive command ‘alias’ is used as follows:

alias Display the list of command aliases

alias <name> Display the value of the command alias <name>

alias <name> <value> Create an alias <name> with the value <value>.

When executing aliases, one can pass additional arguments if needed.

8.0 Support for FinSimMath

8.1 Introduction

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

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

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

Objects of the variable precision types VpReg, 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.

8.2 Variable Precision Fixed Point and Floating Point Support in FinSim

8.2.1 Introduction

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

FinSim 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 extensions1. Beginning with FinSim 10.0 support for the predefined types VpReg and VpDescriptor is also provided as a shorter way 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.

8.2.2 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: p2k.

The value p is allways encoded using some or all 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 manipulation.

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.

The conventions used to declare a VP register, specify the encoding of p, the value of k_fix, the encoding of k_float, the special values and other restrictions are presented in Section 8.5 on page 51. The set of available functions and overloaded operations are presented in detail in Section 8.6 on page 55. Conventions about evaluation of expressions containing VP register elements are presented inSection on page 65. Section on page 65 contains useful examples using the VP register features.

8.3 Specifying VP objects

8.3.1 Introduction

There two kinds of VP data containers: registers and wires. 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 mantisa), as well as information regarding rounding and overflow options. VP wires contain values but do not contain any information regarding format, rounding or overflow.

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

Step 1: Declare a VP descriptor

Step 2: Declare a VP register as data holder

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.

Registers are declared as VP register using the Verilog IEEE std 1364-2001 attribute construct. The attribute varprec is used, having 2 possible values: data and descriptor.

Examples:

(* varprec = data *) reg [0:511] in1;

VpReg [0:511] in1;

(* varprec = descriptor *) reg d1[0:1] d1;

VpDescriptor d1;

Objects marked with the varprec attribute set to data or declared of type 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 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.

VP wires do not have a descriptor associated to them. A VP wire cannot appear in an expression involving more than its name. VP wires are used in order to pass VP values through ports of modules. Wires that are not VP wires are treated as integers when participating in expressions that are assigned to a VP object or that contains a VP object. Therefore, in an assignment to a VP register such as

myVPreg = myWire;

myWire will be considered an integer, whereas if ti were a VP wire it would be considered to be of the same format and size as myVPreg as in the following example:

myVPreg = myVPwire;

It is illegal to have myVPwire declared with a size that is smaller than the necessary number of bits indicated by the descriptor of myVPreg.

Notes:

i) The size of the register 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 size of the descriptor register has no particular meaning, however SuperFinsim requires a size of at least two bits.

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

$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.

8.3.2 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>, <roundingOpt>, <overflowOpt>, <miscOpt>).

8.3.2.1 Setting Format and Sizes

The format field can have the following values:

1 - indicates two's complement

March 1, 2012 54

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 (refered 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 significative -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 significative -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 mantisa.

Notes:

i) negative sizes are not supported in version 8 and will be supported in version 9.

ii) sign magnitude format is not supported in version 8 and will be supported in version 9.

8.3.2.2 Setting Rounding Option

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

8.3.2.3 Setting Overflow Option

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 unrepresentable value divided by the maximum representable value plus one unit.

8.3.3 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 fileds: 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).

8.4 VP register manipulation

8.4.1 Simple Assignments to VP registers

8.4.1.1 Assigning Constants to VP registers

Integer literal constants can be assigned to VP registers as in the exmple below:

myVPreg = 23;

Real literal constants can be assigned to VP registers as in the example below:

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

However, note that real literals are first converted to the Verilog real (which in FinSim uses 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 transfomed 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 transfered 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 mantisa/fractional part or the exponent/integer part respectively would prevent the condition for underflowing or overflowing from occuring.

8.4.1.2 Assigning single VP register to VP register

The value stored in the VP register on the rhs will be transferred into the VP register on the lhs.

If the number of bits of the mantissa or fractional part of the VP register on the lhs are insufficient to store the value stored in the VP register on the rhs then rounding will occur according to the rounding option of the descriptor associated to the VP register on the lhs.

If the value stored in the VP register on the lhs is zero and the value stored in the VP register of the rhs 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 lhs would prevent the underflow condition from occuring.

If the value stored in the VP register on the rhs cannot be stored in the VP register on the lhs 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.

8.4.1.3 Assigning single VP wire to VP register

The number of bits of the VP 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 significative portion of the VP wire into the least significative 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.

Underflow or overflow conditions cannot occur during the execution of such an assignment.

8.4.2 Arithmetic Operators operating on VP registers

8.4.2.1 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.

8.4.2.2 Operators

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

List of unary arithmetic operators: +,-

8.4.2.3 Brief description

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 mantisa of the final result would have prevented the underflow condition from occuring.

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 iinteger part would have prevented the condition for overflow from occuring.

8.4.2.4 Example of use:

myVPreg = myVPr1 + myVPr2;

myVPreg = myVPr1 / myVPr2;

8.4.2.5 Restrictions on the power operator (x**a)

a) In case a > 0 x may have any value.

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

c) 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 a and x.

d) If a == 0 and x != 0 the result will be 0

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

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

g) All other usages will be prompted with an error mesage and the simulation will terminate.

8.4.2.6 Example of use of power operator

myVPreg = a ** x;

8.6.3 Logical Operators involving VP registers

The type of operands may be: literal integer, literal real, integer, reg, wire, VP register with two's complement format.

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

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

Note: in version 8_0_6 only == and != support all types of operands, whereas <, >, >=, <= do not support real literals or VP registers with floating format.

8.4.4 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 mantisa 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.

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

8.4.5 Trigonometric Direct and Inverse Functions

8.4.5.1 List of Functions

$VpSin, $VpCos, $VpTan, $VpCtan, $VpAsin, $VpAcos, $VpAtan, $VpActan

8.4.5.2 Domain of arguments and Range of returned values

These functions accept as argument an angle in radians (direct functions) or a real value. The argument can be in any of the following formats: VP reg twos complement, integer, real, literal integer, literal real, reg, wire.

The return value may be any VP register supported format.

The ideal result must be within 1 ulp of the correctly rounded result or 2**(-127) whichever is higher, unless either overflow or underflow occur.

TABLE 1.

Function	Input Domain	Range of values
$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)

8.4.6 Hyperbolic direct and Inverse Functions

8.4.6.1 List of Hyperbolic Functions

$VpSinh, $VpCosh, VpTanh, $VpCtanh, $VpAsinh, $VpAcosh, $VpAtanh, $VpActanh

8.6.6.2 Arguments and returned values

These functions accept as argument an angle in radians (direct functions) or a real value (inverse functions). The argument can be in any of the following formats: VP reg twos complement, integer, real, literal integer, literal real, reg, wire.

The return value may be any VP register supported format.

The ideal result must be within 1 ulp of the correctly rounded result or 2 **(-127) whichever is higher, unless either overflow or underflow occur.

TABLE 2.

Function	Input Domain	Range of Output Values
$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)

8.4.7 Functions returning universal constants

8.4.7.1 $E ($VpGetE has been deprecated)

Returns the value of e (i.e. 2.72...) with 128 bits for the fractional part or as much precision as fits in the vp register that is on the lhs.

8.4.7.2 $Pi ($VpGetPi has been deprecated)

Returns the value of Pi (i.e. 3.14...) with 128 bits for the fractional part or as much precision as fits in the vp register that is on the lhs.

8.4.7.3 $EM ($VpGetEM has been deprecated)

Returns the value of Euler-Mascheroni (i.e. 0.57...) with 17 digits of precision or as much precision as fits in the vp register that is on the lhs.

8.4.8 Logarithm and Exponential Functions

8.4.8.1 $VpLn

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

Example: $VpLn($VpGetE) = 1;

8.4.8.2 $VpExp

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

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

8.4.8.3 $VpSqrt

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

8.4.8.4 $VpLog

Returns the logarithm in base 10 of the input argument.

Example: $VpLog(100) == 2;

8.4.8.5 $VpPow

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

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

8.4.9 Other Functions accepting VP registers as operators

8.4.9.1 $VpPow2

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

8.4.9.2 $VpCeil

Returns the integer part of the value plus or minus one depending on whether the value is positive or negative.

8.4.9.3 $VpFloor

Returns the integer part of the value.

8.4.9.4 $VpGetExp

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

8.4.9.5 $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.

8.4.9.6 $VpGetMant

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

8.4.9.7 $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.

8.4.9.8 $VpAbs

Returns the absolute value of the argument.

8.4.9.9 $VpHypot

Accepts two arguments a and b, which can be literal integers, literal reals, integer, wire, Verilog registers, VP registers in two's complement or VP registers in floating point fomat.

The returned value is $VpSqrt(a*a + b*b).

8.4.10 Using Special Condition Signals/Flags of VP registers

8.4.10.1 Overflow

When an overflow occurs while assigning a value to a VP register named MyVPreg, an implicit register named MyVPreg.Overflow is set to a value equal to the number of bits which if added to the exponent or integer part of the VP register would have prevented the overflow from occurring if such a number exists or an arbitrary number greater than zero otherwise.

8.4.10.2 Underflow

When an underflow occurs while assigning a value to a VP register named MyVPreg, an implicit register named MyVPreg_Underflow is set to a value equal to the number of bits which if added to the mantissa or fractional part of the VP register would have prevented the underflow from occuring if such a number exists or an arbitrary number greater than zero otherwise.

8.4.11 Assigning VP registers to Verilog registers

The assignment is governed by normal verilog rules. The VP register is considered a verilog register for the purpose of such an assignment.

8.4.12 Assigning Verilog registers to VP registers

The assignment must be performed with the system function $VpCopyReg2Vp. Also, please note that in normal usage the bits stored in the register passed as argument to the system function must correspond to the bits resulting from an assignment of a VpReg having the same format and size of fields as the VpReg on the left hand side of the assignment of the system task. Only in such a case the result is easy to predict, e.g.:

```
reg [0:30] r;

VpReg [0:30] a;

VpReg [0:30]b;

VpDescriptor d;
```



```
initial begin
.......
$VpAssocDescriptorToData(a, d1);
$VpAssocDescriptorToData(b, d1);
a = 3.0;
r = a;
b = $VpCopyReg2Vp(r);
end
```

The value stored in b will be 3.0.

Restriction: note that in the current implementation of FinSimMath assinments of Verilog registers to VpReg using $VpCopyReg2Vp does not work in case the VpReg is an element of an array. The workaround is to use a temporary variable.

8.4.13 Assigning Verilog Real to Verilog registers

This assignment is governed by Verilog rules, namely the integer part of the value of the real is assigned to the Verilog register and it is truncated if not enough bits are available in the Verilog register.

If one is interested in storing the exact sequence of bits of the representation of the value stored in the Verilog real, one can use the system function $VpFPkCopyFl2Reg or $VpFCopyFl2Reg. The first one is more efficient but the returned value cannot currently be displayed. It can only be used by $VpFPkCopyReg2Fl. The result returned by $VpFCopyFl2Reg can be displayed by the standard Verilog mechanisms ($display, $monitor, etc.). A result obtained via $VpFPkCopyReg2Fl must be converted back to a real by using the function $VpFPkCopyFl2Reg and a result obtained via $VpFCopyFl2Reg must be converted back to a real by using the function $VpFCopyReg2Fl.

8.4.14 Displaying VP register values

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

%y: displays the value of VP registers with twos 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 depnding on the fomat 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.

8.4.15 I/O of VP registers

I/O is not supported in FinSim 8.x for VP registers. However, by using assignments to and from Verilog registers and using Verilog 2001 I/O for Verilog registers, I/O can be performed for VP registers, albeit in an indirect way.

8.4.16 Plotting data

The Super FinSim distribution contains in demo/demo_vp as an example fft_plot.v which performs the fft of an input vector and produces a text file which can be displayed by ptplot from UC Berkeley and it represents the magnitude vs frequency curve of the fft result.

8.4.15.1 $VpPtPlot

One can also use the system task $VpPtPlot which produces a text file which can be displayed by ptplot. It can display

several curves on hte same image. $VpPtPlot accepts the following arguments:

1) Name of the result file.

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

3) the double of the distance between the projection on the first dimension of two consecutive point.

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

5) The total span of the the first dimension, i.e. the difference betoween the fist coordinates of the last and first points to be represented.

6) Label of the first dimension

7) Label of the second dimension

8) two dimensional array of values to be plotted. Each row represents one curve to be plotted.

9) The remaining arguments represent the labels of the different curves to be plotted. FinSim version 10_05_29 supports only up to three different curves on one image.

An example of usage of $VpPtPlot is given below:

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

8.4.15.2 $Flot

One can also use the system task $Flot which produces a text file which can be displayed by Flot. In order to work 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 projection 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");

8.5 Cartesian and Polar types

These predefined types are added to Verilog by the FinSimMath extension. Objects of these types may be elements of multi-dimensional arrays, may be operands of arithmetic operators and may be arguments of various system functions and tasks. Any reference to $I represents a reference to a complex number in cartesian representation having its .Re field equal to zero and its .Im field equal to one.

8.5.1 Type VpComplex

This type consists of two fields of type VP register named "Re" and "Im". Objects of this type must be associated to a descriptor before being used. The two fields represent cartesian co-ordinates and are treated as such by the operators operating on them as well as by the system tasks and functions to which they are provided as arguments.

Objects of this type have associated underflow and overflow registers, which are updated at the time new values are placed in them.

8.5.2 Type VpPolar

This type consists of two fields of type VP register named “Mag” and “Ang”. Objects of this type must be associated to a descriptor before being used. The two fields represent polar co-ordinates and are treated as such by the operators operating on them as well as by the system tasks and functions to which they are provided as arguments.

Objects of this type have associated underflow and overflow registers, which are updated at the time new values are placed in them.

8.5.3 Type VpFComplex

This type consists of two fields of type real named “Re” and “Im”. The two fields represent cartesian co-ordinates and are treated as such by the operators operating on them as well as by the system tasks and functions to which they are provided as arguments.

8.5.4 Type VpFPolar

This type consists of two fields of type real named “Mag” and “Ang”. The two fields represent polar co-ordinates and are treated as such by the operators operating on them as well as by the system tasks and functions to which they are provided as arguments.

8.5.5 Operators on Cartesian and Polar types

The following operators are supported in conjunction with Cartesian and Polar types: +, -, *, /, **, =., == and !=.

Any Cartesian or Polar type may be assigned to any other cartesian or polar type without using explicit conversion functions. The operator = is used for assignment.

Expressions involving Cartesian or Polar types may be part of hierarchical expressions. contain at most one operator and may contain only operands that are of type Cartesian or Polar. Other kind of expressions must be broken down into simple expression having the property described above.

8.6 Operations on Multi-dimensional arrays

Multi-dimensional arrays may have elements of one of the following types: integer, reg, real, VpReg, VpComplex, VpPolar, VpFComplex, and VpFPolar.

8.6.1 Populating Multi-dimensional arrays with values

Multi-dimensional may have values placed in them via: (1) element by element assignments, (2) assignments of aggregated values, (3) assignment of another multi-dimensional array with compatible elements and exactly the same number of dimensions and the same size on each dimension and (4) the $InitM system task.

8.6.1.1 Element by element assignment

This can be achieved using standard Verilog constructs for loops and multi-dimensional array reference. For referencing Cartesian or Polar fields, System Verilog syntax is used. For example:

```
VpFPolar myFP[SIZE-1:0][SIZE-1:0];

for (i = 0; i < SIZE; i = i + 1)

begin

myFP[i]j].Mag = 1.0;

myFP[i][j].Ang = 0.0;

end
```

8.6.1.2 Assignment of aggregated values

This can be achieved using standard System Verilog construct for aggregate assignment. For example:

```
VpFPolar myFP[SIZE-1:0][SIZE-1:0];

myFP = {{1.0, 0.0}, ...{1.0, 0.0}}
```

where "..." represents the repetion of {1.0, 0.0} as necessary to cover the entire matrix.

8.6.1.3 Assignment of another multi-dimensional array

This can be achieved using an assignment statement between two multi-dimensional arrays having the same number of dimensions and same size on each dimension and having compatible types. Types are compatible automatic conversions are supported between them.

Types real and VpReg are compatible with each other and types VpComplex, VpPolar, VpFComplex, and VpFPolar types are compatible with each other.

An example of assignment of simple identifiers representing compatible multi-dimensional arrays is provided below:

```
VpComplex myC[SIZE-1:0][SIZE-1:0];

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

...

myP = myC;
```


8.6.1.4 $InitM system task

The $InitM system tasks accepts two or three arguments, depending on whether the elements of the multi-dimensional array are scalars or of a Cartesian or Polar type. The first argument is the name of the multi-dimensional array to be populated. For example:

VpComplex myC_Mat[SIZE-1:0][SIZE-1:0];

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

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

$InitM(myP, 1.0, 0.0);

$InitM(myC, 1.0, 0.0);

$InitM(myR, 1.0);

Note that the second and third arguments represent any valid scalar expressions. These expressions may also contain the allowable positional system functions $I1, $I2, $I3, $I4, and $I5 each corresponding to the corresponding dimension of the multi-dimensional array.

The dimensions are counted from the left in the declaration of the multi-dimensional array and the counting starts with 1.

A positional system function is allowable if the corresponding multi-dimensional array has the necessary number of dimensions.

Each positional system function returns the current index on the given dimension at the time of evaluation of the expression. Therefore the myR[3][4] will receive the value of the expression where any occurrence of $I1 is replaced by 3 and of $I2 by 4.

The example below places in myRT the transposed of myR:

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

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

$InitM(myRT, myR[$I2][$I1]);

8.6.2 Viewing elements of a multi-dimensional array as part of a different structure

8.6.2.1 The View - as declaration that can be used for both reading and writing

The syntax of the View declarations that may be used for both reading and writing of objects is

View <multi_dim_array_declaration> as

<multi-dim_arrray_selection>

The expressions representing the index in each of the dimensions of the <multi-dim_arrray_selection> may contain positional system functions. These functions are replaced with the corresponding indexes in the references to the view.

For example:

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

View real myRT[SIZE-1:0][SIZE-1:0] as myR[$I2][$I1];

Any reference to myRT[i][j] refers to myR[exp1][exp2] where exp1 and exp2 are the results of evaluating the expressions in the view declaration with $I1 replaced by i and $I2 replaced by j.

8.6.2.2 The View - as declaration that can be used only for reading

If the View declaration is used only for reading then the

<multi-dim_arrray_selection> may be replaced with a general expression. Therefore, The syntax of the View declarations that may be used for reading is:

View <multi_dim_array_declaration> as

<expression_with_positional_system_tasks>

This feature is first supported in version 10.0.

8.6.3 Displaying Multi-dimensional Arrays

8.6.3.1 $display and $monitor system tasks

Each scalar element of a multi-dimensional array may be displayed with the $monitor and $display system tasks.

Multi-dimensional arrays may be displayed globally using the $PrintM system task. The first argument of this task is the name of the multi-dimensional array to be displayed. The other argument indicates in quotes the format in which each field of the elements of the multi-dimensional array are to be displayed.

Note: this mechanism works only for multi-dimensional arrays having as elements scalars or a record of scalars, which is the case for the currently supported types: real, reg, integer, VpReg (VP Register), VpComplex, VpFComplex, VpPolar, and VpFPolar.

Eaxample:

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

$PrintM(myR_Mat, “%e”);

8.6.3.3 $PrintLine and $PrintCol

A line or a column of a two dimensional array may be displayed globally using the $PrintLine or $PrintCol system tasks, respectively. The first argument of these tasks is the name of the two-dimensional array. The other argument indicates the line or column respectively. The format is not selectable. For elements of type real it is %e and for elements of type VpReg it is %y;

The type of elements can be: real, reg, integer, VpReg (VP Register), VpComplex, VpFComplex, VpPolar, and VpFPolar. For FinSim versions prior to version 11.0 only two-dimensional arrays of type VpReg and type real may be provided as first argument to these functions.

Eaxample:

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

$PrintLine(myR_Mat, 1);

8.6.3.2 $DispM system task

Multi-dimensional arrays may be displayed globally using the $DispM system task. The first argument of this task is the name of the multi-dimensional array to be displayed. The other argument indicates in quotes the format in which each field of the elements of the multi-dimensional array are to be displayed. If the second argument is omitted “%e” is assumed. In versions prior to 11.0 only one argument can be provided to $DispM and it has to be a matrix with elements of type real.

Eaxample:

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

$DispM(myR_Mat);

8.6.4 Norms and Distances

Norms and Distances are system functions returning a real value.

8.6.4.1 $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.

8.6.4.2 $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.

This system function returns the maximum absolute value of the all elements of the matrix M1.

8.6.4.4 $VpNormAbsSum(M1);

This system function returns the sum of the absolute values of the all elements of the matrix M1.

8.6.4.5 $VpNormAbsRMS(M1);

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

8.6.5 Sparse Matrices

A two-dimensional array may be declared at runtime as a sparse matrix, using the system task $ToSparse. This task accepts as argument the name of a two dimensional array.

Sparse matrices store zeroes more efficiently and operations such as: storing, arithmetic operations, printing are performed faster.

Sparse matrices can be passed as aruments to $PrintLine and $PrintCol.

Sparse matrices can be passed as arguments to system tasks that perform Norms and Distances.

Sparse Matrices can have their non-zero elements read more efficiently using the boolean system functions $SpReadNextNzElemInLine and $SpReadNextNzElemInCol.

These functions return true if they could find a non-zero element positioned after the current element.

Example:

found = $SpReadNextNzElemInLine(M, lin, col, idx, value); where M is a sparse matrix, lin and col are the line and column of the current element, idx is an handle associated with a certain element in the sparse matrix and value is the value of the next element. Note that col may change value during the call since lin/val before the call point to a certain element and after the call lin/val point to the next non-zero element on the same line.

found = $SpReadNextNzElemInCol(M, lin, col, value); where M is a sparse matrix, lin and col are the line and column of the current element and value is the value of the next element. Note that lin may change value during the call since lin/val before the call point to a certain element and after the call lin/val point to the next non-zero element on the same column.

Arithmetic operations can be performed on sparse matrices, provided that all operands are sparse matrices.

8.6.6 Fast Fourier Transform: $VpFft and $VpIfft

These tasks perform the Fast Fourier 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.

Example:

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”);

8.6.7 Discrete Cosine Transform: $VpDct and $VpIdct

These tasks perform the Discrete 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 DCT 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 consecutive in the object or the view being passed as argument.

Example:

View real myR_even[SIZE/2-1:0]as myR[2*$I1];

$VpDct(myR_even,0, SIZE/2);

$PrintM(myR_even, “e”);

8.6.8 Linear Differential Equations

8.6.8.1 $VpLODE

The i-th equation of a system of linear differential equations can be represented as:

Cin*Yi(n)+ ...Ci0*Yi(0) = Fi, where Cin is the coefficient of the n-th derivative of the i-th variable, and Fi is the i-th external function.

Linear Differential equations can be solved using the system task $VpLODE, which accepts the following parameters:

1) maximum order of derivatives appearing in hte equations to be solved.

2) number of equations to be solved

3) double of the distance between two consecutive values of the solution of the equation.

4) two dimensional array storing the solution, where the first dimension must be larger than the number of equations. Note that the original value must be provided before the call to $VpLODE.

5) two dimensional array storing the coefficients of the system of equations.

6) Two dimensional array storing the external functions.

7) Two dimensional array storing the derivatives of the solution. Note that this is a view of a three dimensional array: one dimension for equations, one dimension for the order of the derivative, and one dimension for the values of each derivative. However, the system task $VpLODE accepts a two dimensional array, so this two dimensional array will store in groups of n-1 lines the n-1 derivatives of the of each variable of the solution.

8.6.8.2 Finding polinomial given the roots

pol = Poly(roots); Note that the argument must be of a complex type and the result an array of scalars real or variable precision.

8.6.8.3 Finding roots given a polinomial

r = $Roots(pol);

8.6.8.4 Finding characteristic polynomial of a square matrix

pol = $CharPol(M);

8.6.8.5 Finding the rank of a matrix

rank = $Rank(M);

8.6.8.6 Finding the eigenvalues of a matrix

eval = $Eig(M);

8.6.8.7 Finding the eigenvalues and eigenvectors of a matrix

eval = $Eig(M, evect);

8.6.8.8 Finding the feedback matrix K that places the poles of the closed loop system as indicated by the complex vector r.

K = $Place(A, B, r);

8.6.8.9 Finding the solution of a system of linear system of differential equations of order one. The result is placed in y and the state variables are in x.

y = $LSim(A, B, C, D, u, t0, dt, x);

8.6.8.10 Finding the dc-gain of a linear system

dcg = $dcgain(A, B, C, D);

8.6.9 Numeric Differentiation and Integration

8.6.9.1 Numeric Differentiation: $VDif(1-DM, sampling step, nr of samples, 1-DMRes)

The numeric integration function accepts the following arguments:

1) a one dimensional array storing the values of the input function at the various sampling points.

2) the value of the distance between two consecutive sampling points.

3) nr of sampling points

4) a one-dimensional array containing the result.

8.6.9.2 Numeric Integration: $VInt(1-DM, sampling step, nr of samples, 1-DMRes)

The numeric integration function accepts the following arguments:

1) a one dimensional array storing the values of the input function at the various sampling points.

2) the value of the distance between two consecutive sampling points.

3) nr of sampling points

4) a one-dimensional array containing the result.

8.6.10 Symbolic Computation

8.6.10.1 Symbolic Differentiation: resExpr = $Dif(n, symbExpr, “name”);

The symbolic differentiation function returns a symbolic expression and accepts the following arguments:

1) number of times differentiation will be applied to the input symbolic expression and to the interim result in order to obtain the final result. For example $Dif(3,“sin(x)”, “x”) will return the third derivative with respect to x of sin(x).

8.6.10.2 Symbolic Integration: resExpr = Int(n, symbExpr, “name”);

The symbolic integration function returns a symbolic expression and accepts the following arguments:

1) number of times differentiation will be applied to the input symbolic expression and to the interim result in order

to obtain the final result. For example $Dif(3,“sin(x)”, “x”) will return the third derivative with respect to x of sin(x).

This system function performs the Laplace Transform on the exression stored in the second argument with respect to the variable whose name is stored in the third argument. The first argument indicates the number of times the Laplace Transform shall be applied. The Laplace transform is represneted as a function of “s”.

8.6.10.4 Inverse Laplace Transform: resExpr = $ILap(n, symbExpr, “name”);

This system function performs the Laplace Transform on the exression stored in the second argument with respect to the variable whose name is stored in the third argument. The first argument indicates the number of times the Laplace Transform shall be applied. The Laplace transform is represented as a function of “s”.

8.6.10.5 Evaluation of Symbolic Expression: $Eval(symbExpr, value)

The symbolic expression stored in the first argument is evaluated for the current values of the objects being referenced within the expression, and the result is stored in the second argument, which may not be an expression or a literal..

8.6.10.6 Multi-Evaluation of Symbolic Expression: $EvalArray(symbExpr, “x”, increment, nrSteps, value)

The symbolic expression stored in symbExpr is evaluated for the current value of the variable whose name is provided by the second argument as well as for each subsequent value of this variable obtained by adding its value to the value of the third argument. Each result of the evaluation is stored at the next index starting with index zero. There will be as many evaluations as indicated by the fourth argument

8.6.10.6 1-D Symbolic Extraction: $FindExpr1(1-DM, step, “name”, dist, n, resExpr)

This system task returns in the sixth argument the symbolic expression, in terms of the variable having the name provided by the third argument, which approximates the values stored in the first argument, where consecutive values correspond to an increment provided by the second variable. The fifth argument indicates how many samples starting with index zero shall be considered within hte array 1-DM.

Given:

$FindExpr1(1-DM, step, “name”, dist, n, resExpr);

$EvalArray(resExpr, “name”, step, n, 1-DMRes);

d = $VpDistAbsMax(1-DM, 1-DMRes);

It must be that either d < dist (the fourth argument of system task $FindExpr) or the the sixth argument of the same task must return a null string.

8.6.10.7 2-D Symbolic Extraction: $FindExpr2(2-DM, step1, step2, “n1”, “n2”, dist, n1, n2, resExpr)

Similar to $FindExpr1.

8.6.10.8 2-D Symbolic Extraction: $FindExpr3(3-DM, step1, step2, step3, “n1”, “n2”, “n3”, dist, n1, n2, n3, resExpr)

Similar to $FindExpr1.

8.7 Generation of Gate-level models

Various Generators of gate-level models can supported on the FinSimMath simulation platform, because:

1) it is easy to provide bit-accurate FinSimMath models for the gte-level descriptions and thus allow for an easy configuration change for each instance from FinSimMath to gate level and vice-versa.

2) all interfaces are modeled at the Verilog bit level and conversion functions are provided to convert between Verilog bit level, Verilog real type and variable precision objects: $VpFCopyFl2Reg, $VpFCopyReg2Fl, $VpCopyReg2Vp.

8.7.1 FIR Filter Generation

8.7.1.1 Invocation

The FIR filter generation is performed invoking:

finvc -cf fir_spec.cf

where the command file fir_spec has the following syntax:

fir_spec := -a <name>.v +FIR +Pass +Synch

+attenuation=<value> +ripple=<value>

+frequency_unit=<funit> +pa=<value>

+p1=<value> +p2=<value> +pb=<value>

+pm=<value> +mrate=<value> +srate=<value>

+irate=<value> +orate=<value>

+tb_f1=<value> +tb_f2=<value>

+nr_samples=<value> +Res=<rname>.res

<timescale>

As a result of the invocation a file named <name>GTL<rname>.v will be produced. This file will contain the gate-level representation of the generated filter, the test bench and the high level bit accurate models of resources used by the generated filter, such as the multiply accumulate module.

8.7.1.2 Description of the resource file

The resource file indicated by +Res=<rname>.res is a special case of the resource file used for Math2GTL Synthesis and has the following syntax:

primitives: string, integer

file ::= {declarations | mac_decl}

declarations ::=

mac_decl ::= mac <mac_name>; <op_body> end

mac_name ::= mac_<type>_<sz1>_<sz2>

type ::= fl | fx

sz1 ::= integer /*size of first field of data container*/

sz2 ::= integer /*size of second field of data container*/

fl ::= string /*floating point*/

fx ::= string /*fix point as two's complement*/

declarations ::= declare {decl_assign} enddeclare

decl_assign ::= nr_mem_assign|nr_op_assign|nr_targets_assign

nr_mem_assign ::= nrMemories = 0;

nr_op_assign ::= nrOperators = 1;

nr_targets_assign ::= nrTargets = 1;

<op_body> ::= {op_assign}

op_assign ::= format = string; sz1 = value; sz2 = value;

rounding = string, overflow = string;

word_width = value; latency = value;

nrInst = value;

8.7.1.3 Description of the Filter Specification

+FIR: is mandatory and indicates the type of filter to be generated. Based on which type of filter is to be generated different specification items must be provided.

+Pass: indicates whether the specification is for pass band or for stop band.

+attenuation: indicates in dB the minimum attenuation for the stop band.

+ripple: indicates in dB the maximum ripple of the pass band.

+frequency unit: indicates which unit is used for specifying the various frequencies. All use the same unit. The unit can be one of the following: GHz, Gr/s, MHz, Mr/s, KHz, Kr/s, Hz, r/s.

In case +Pass is specified the meaning of +pa, +p1, +p2, +pb, and +pm are as follows:

+pa: indicates the right bound of the left stop band.

+p1: indicates the left bound of the pass band.

+p2: indicates the right bound of the pass band

+pb: indicates the left bound of the right stop band.

In case +Pass is not specified the meanings are chnaged such that +p1 and +p2 represent the letf and respectively the right bound of the stop band, and +pa and +pb represent the right and respectively left bound of the left and respectively right pass bands.

+pm: indicates the maximum frequency of interest. Typically this is sampling_rate/2 and should not exceed this limit.

+mrate: indicates the frequency of the root clock from which all clocks can be derived.

+srate: indicates the sampling rate.

+irate: indicates the frequency of the internal clock.

+orate: indicates the frequency of the output clock.

+tb_f1: indicates the frequency of the ideal output which will also participate additively to the input.

+tb_f2: indicates the frequency of the signal to be added to the signal of frequency tb_f1 in order to produce the input signal.

+nr_samples: indicates the number of samples tha twill be considered in hte simulation.

+Res: indicates the name of the resource file.

-timescale: is Verilog timescale directive, e.g. timescale 1ps/ 1ps, to be used in the simulation. The timescale will be made consistent with the frequency unit and the frequency values used.

8.7.1.4 Description of the generated code

The generated code consists of:

- module top: test bench

- module fir: filter described at structural level

- module pmac: described at structural level contains several multiply accumulators working simultaneously and some logic that decides when the pmac should process and put output results at the multiplexed output of the filter.

- module circ_cnt: circular up counter described at RTL level

-module idxgen: circular down counter described at RTL level

- module cmp1: comparator of bits described at structural level.

- module mem_a: RTL model of the memory containing the coefficients.

- module mem_in: structural model of a single write port/multiple read ports, using instances of module mem_rw.

-module mem_rw: RTL model of a single port write single port read memory.

- module data_gen: FinSimMath description of the module producing the input signal as specified by tb_f1 and tb_f2.

module idata_gen: FinSimMath description of the module producing the ideal output as specified by tb_f1.

module mac: FinSimMath bit accurate description at the mathematical level of the multiply accumulate module used in one or more instances.

8.7.1.5 Description of the Generated Test Bench

The Test Bench consists of:

- code that buffers the input to the filter and serializes the output of the filter, if necessary. This code is combined with code that handles the decimation of the output.

- code that computes at the mathematical level the computations performed at the gate-level by the filter and compares the results, issueing error messages if necessary.

- code that produces graphical representations of: i/o spectrum, i/o values, and amplitude gain.

9.0 Tour of the Super-FinSim design environment

9.1 Running Super-FinSim in pure interpreted mode

In the pure interpreted mode, all modules are interpreted. To run Super-FinSim in the interpreted mode, the compiler must be invoked as follows:

finvc -dsm int <other compiler options>

9.2 Running Super-FinSim in mixed mode

The compiler allows many options to specify mixed mode simulation. The options are:

-dsm <mode>: Set the default simulation mode to compile (-dsm com) or interpret (-dsm int). The default simulation mode is compiled.

-intm <modulename>: Interpret the specified module.

-comm <modulename>: Compile the specified module.

-intf <file>: Interpret modules read from the specified file.

-comf <file>: Compile modules read from the specified file.

-intd <directory>: Interpret modules read from any file in the specified directory.

-comd <directory>: Compile modules read from any file in the specified directory.

9.3 Running FinSim in accelerated mode

For the fastest simulation speed, the compiler should be invoked with the optimization level 11, '-ol 11'. Other general invocation options of the simulator that can speed up the overall simulation time are '-nodriverchk', '-nowarning' and '-notimechk'. A side affect of disabling timing check with the option '-notimechk' is that timing check information will not be accessible through PLI or SDF. For an even greater performance advantage, timing checks can be disabled at compile time with the option +notimingchecks. Furthermore, entire specify blocks can be disabled at compile time with +nospecify.

9.4 General simulation tips

1.Behavioral designs/modules should be compiled and structural designs should be interpreted.

2.Modules providing stimulus via large initial blocks should be interpreted instead of compiled.

3.Designs that run for a short period of time should be interpreted, particularly if the time it takes to build the simulator is comparable to the actual simulation time.

4.Designs that have no timing violations run faster if the option '+notimingchecks' is specified at compile time.

5. If the C compiler fails on a module due to the limitations of the compiler, one can interpret that module instead.

6. Simulations running at the highest optimization level (-ol 11) may lead to slightly different results. This is mainly due to the difference in the order of evaluation.

7.For high degree of compatibility with the OVIsim simulator, invoke the simulator with the option '-c'.

10.0 The Graphical User Interface for Super-FinSim

To start the GUI run fingui from the command prompt.

You can now open an existing design or create a new one. A design is a collection of Verilog source files and the options for the compile, build and simulate operations of Super-FinSim. It could also contains some other files used in your simulation: C or object files for your PLI or text files for comments.

To create a new design, go to Design | New design. By default the name of a new design is untitled. To change the name of the design go to Design | Save As and specify the name of the design. You can use also Design | Save to save any changes you made in an opened design. To open an existing design already saved use Design | Open.

The list of files which are part of the current design is displayed on the left side of the main window. To add a new file to the design, go to Design | Insert file and select a file. To remove a file from the design use Design | Delete file.

To open a source file from your design double-click with the left button of your mouse. It is possible also to open any text file selecting File | Open. If you want to save the changes you made in a source file, use File | Save or File | Save As. It is possible also to create a new source file from File | New.

In order to run a simulation from GUI, you need to execute the same steps as in the batch mode: compile (similar with finvc) from Run | Compile, build (similar with finbuild) from Run | Build and run (similar with running TOP.sim) from Run | Simulate. The same commands can be executed by selecting the buttons from the toolbar of the main window. Before running any of these commands you can set the options from Options | Compile, Options | Build or Options | Simulate. An easier way to set the options and closer to the batch mode is to go to Options | Compile | Custom, Options | Build | Custom or Options | Simulate | Custom.

When you run the compile or build command, a new window is displayed with all messages reported by this command. To close the window select Exit button from the top of it.

If there are no errors reported by compile or build commands, you can execute the simulation from Run | Simulate or pressing the button from the toolbar. A new window will be displayed and all messages reported by the simulation will be reported in this window. If the simulation stops in the interactive mode (for example by executing the Verilog $stop statement), you can enter valid interactive commands from the combobox located at the bottom of the window and pressing Send command. If you want to cancel the simulation just pres Quit.

The simulation can be run in the interactive debug mode by selecting Run | Debug or Debug. The debug mode is slower than the default mode and is possible only if you run the compile command with the option +srcdbg. Here are some features of the debug mode:

•the simulation can be stopped by using the Stop button. The simulation will then run in interactive mode.

•select Source debug for source level debugging. In this mode, use Next to go to the next Verilog code line to be executed. Also you can add source breakpoints: double-click in the left list box of the window.

•add time breakpoints: go to Breakpoints | Add time breakpoint. The time can be absolute (from the beginning of the simulation) or relative (from the current time).

•add signal change breakpoints: go to Breakpoints | Add signal breakpoint.

•delete a breakpoint: go to Breakpoints | Breakpoints list, select the breakpoint and press the Delete button.

It is possible to visualize the code coverage of your simulation if you compiled the design with the option +finvcc. You can have access to this feature selecting select Run | Code coverage after you completed the simulation. A new window will be displayed and you can select the source Verilog file for which you want to view the coverage and also the type of the report you want to obtain (“less than” or “greater than” a particular value.

Note: GUI for FinSim is available only for the Windows, Linux and Solaris platforms.

10.1 FinSim status codes and errors

All FinSim programs return a status code upon execution. A status code of 0 indicates that the command executed successfully. A non-zero status value indicates the execution failed. The table below illustrates all FinSim status codes:

Status code

Status meaning

0

Execution was successful with zero or more warnings.

1

Execution failed due to an I/O error.

2

Execution failed due to a memory error.

3

Execution failed due to a user error.

4

Execution failed due to a fatal error.

5

Execution failed due to an internal error.

6

Execution failed due to a user warning.

7

Execution failed due to a license error.

10.2 Syntactic errors

Syntactic errors in a Verilog description are errors that violate the BNF grammar of Verilog. Error messages identify the source file, the line and the column of the offending character or word and provide information regarding any possible correction.

10.3 Semantic errors

Semantic errors in a Verilog description are errors that do not violate the BNF grammar, but the meaning of the language. Error messages clearly indicate the nature of the problem.

10.4 Simulation errors

Simulation errors are run-time errors encountered by the simulation kernel. These errors consist of design errors or errors made by the user during the simulation session.

10.5 Compiler Internal errors

Compiler internal errors should never be encountered by the user of the simulator. However, if this ever happens, such messages should be reported to Fintronic USA, Inc., along with the corresponding file, as user's bug reports (UBRs).

10.6 Simulation Internal errors

Simulation internal errors should never be encountered by the user of the simulator. However, if this ever happens, such messages should be reported to Fintronic USA, Inc., along with the corresponding file, as user's bug reports (UBRs).

11.0 Running FinSim with Code Coverage

11.1 Introduction

Code Coverage is a package of tools which allow the user to observe how many times each behavioral statement is executed during simulation.

Before starting synthesis, it is important to ensure that each line of behavior has been executed at least once.

To use the code coverage mechanism in FinSim, a proper license must be ordered from Fintronic or its distributors.

To activate the code coverage mechanism, the FinSim Verilog compiler (finvc) must be invoked with the option +finvcc, as follows:

finvc <option or source file> +finvcc [<option or source file>...]

After compiling, linking and simulation, a file design.fil will be created in the fintemp directory. This file contains information about the code coverage in binary form.

11.2 Code Coverage Information

The utility vccdump displays all information contained in design.fil. This information tells how many times each basic block was executed and also very important, which blocks were never executed. By default, vccdump reads the information from fintemp\design.fil. To read information from another file, run

vccdump -f <filename>

where <filename> is the full path to another *.fil file.

The utility vccmerge merges two or more code coverage files (*.fil) from many runs of the same design. The syntax is:

vccmerge file1 file2 .. filen [-o file_result]

where file1, file2..., filen are the files to be merged and file_result is the result file.

By default the result file is design.fil.

11.3 Display the Code Coverage Information

The utility vccdisplay transforms each Verilog source file that is part of the simulation that produced the code coverage information into a corresponding file with the extension .pro.

Each .pro file is located in the same directory as the corresponding Verilog file, has the same prefix (name without extension), and contains the same Verilog description. In addition to the Verilog description, the .pro file contains annotated code coverage information, indicating how many times each line has been executed.

If different portions of the same line have been executed a different number of times then there will be more numbers in front of that line.

To reduce the number of occurrences of lines with multiple execution numbers, the user is invited to run the collection of code coverage information (finvc +finvcc) on the pretty

printed version of the Verilog code. The pretty printed version of the Verilog code can be obtained by running

finvc -pp <other compiler options>

The pretty printed version of the Verilog code will be produced in the file pp.out or

finvc -ppf <pretty printed file name> <other compiler options>

to get the pretty printed version in a specified file.

11.4 Graphical User Interface for Code Coverage

To access the visual interface for code coverage run vccgui. To open a design created from fingui, use File | Open design or Open design button. It assumes the design was compiled (with '+finvcc' option), built and simulated.

After a design is opened, load a source file from the design list. In the list boxes from the left will be displayed how many times each basic block was executed. You can select the basic blocks which were executed less/more than a specific value. To do this, go to View | Select. To unselect all lines which were selected, go to View | Unselect all.

To display the code coverage information for the entire design, select Command | VccDump. This command is the same as vccdump from the command prompt.

To merge two code coverage result files select Command | Merge vcc files.

Note: GUI for Code Coverage is available only for the Windows and Linux platforms.

12.0 Running FinSim with third party tools

12.1 Running Super-FinSim with Specman

These instructions apply to FinSim version 4.7.26 or higher.

The files needed for linking FinSim with Specman are:

specman.tab TAB file to let finvc know about Specman's PLIs

specmanpli.com.makinclude file for running Specman in compiled mode; specifies the Specman object files that need to be linked in

specmanpli.int.mak include file for running Specman in interpreted mode; specifies the Specman object files that need to be linked in

All these files can be found in the include directory of the Super-FinSim distribution (the directory specified in the FIN_INCLUDE_PATH environment variable).

In order to have Finsim working with Specview, you need to replace the file "misc.tcl" from your Specman installation with the one in this directory. Just do the following:

mv $SPECMAN_HOME/tcl/specman/misc.tcl $SPECMAN_HOME/tcl/specman/misc.tcl.old

cp $FIN_INCLUDE_PATH/misc.tcl $SPECMAN_HOME/tcl/specman

(Currently this is the misc.tcl from Specman 4.0.2. If you have a different version of Specman, please contact Fintronic Customer Support to get an updated misc.tcl for your release. This will be corrected by Verisity R&D in the near future to be taken care of automatically.)

12.1.1 Verilog Compilation

finvc -ptab $FIN_INCLUDE_PATH/specman.tab +fullaccess <your other options>

12.1.2 Building the simulator.

•Run Specman in compiled mode.

Copy the file specmanpli.com.mak into finpli.mak in the local directory where you run finbuild:

cp $FIN_INCLUDE_PATH/specmanpli.com.mak finpli.mak

Edit the file finpli.mak and add to the FINUSERPLIOBJ variable all the .o files (or shared libraries) generated by Specman's sn_compile.sh. Please note that the Specman supplied $(SPECMAN_HOME)/src/veriuser.c file will be compiled into $(SPECMAN_HOME)/src/veriuser.o. If you prefer this object file to reside in a different directory, please make a copy of $(SPECMAN_HOME)/src/veriuser.c into that directory and modify finpli.mak correspondingly.

Build the simulator:

finbuild -verbose

•Run Specman in interpreted mode.

Copy the file specmanpli.int.mak into finpli.mak in the local directory where you run finbuild.

cp $FIN_INCLUDE_PATH/specmanpli.int.mak finpli.mak

Please note that the Specman supplied $(SPECMAN_HOME)/src/veriuser.c file will be compiled into $(SPECMAN_HOME)/src/veriuser.o. If you prefer this object file to reside in a different directory, please make a copy of $(SPECMAN_HOME)/src/veriuser.c into that directory and modify finpli.mak correspondingly.

For Solaris, please replace linux with solaris for all variables in the finpli.mak file and change FINUSERPLIDLL to FINUSERPLIDLL=-ldl

Build the simulator:

finbuild -verbose

12.1.3 Running the simulation

You can now use Specman with FinSim in the same way you would do it with any other simulator:

•Interactive mode

In interactive mode you must run the simulator with the flag that requires to enter interactive mode immediately:

TOP.sim -i

Once you get the command prompt you can issue $sn commands:

specman > $sn("test")

and then run the simulation

specman > run ~

specman > quit

•Batch mode

In batch mode you have 2 alternatives:

If you have a command file (or script file) you may add $sn("test") there before 'run' and then run

TOP.sim -script <your command file name>

or, you may use Specman pre-commands in one of the few possible ways:

setenv SPECMAN_PRE_COMMANDS "test"

specview -p "test" <your normal simulation command>

specsim -p "test" <your normal simulation command>

12.2 Running Super-FinSim with Debussy

The FinSim family of simulators now support a direct, high performance interface to Debussy so there is no need to link in any PLI. Simply place your $fsdb<...> calls in the Verilog source code and run the simulation as you would normally do. To run FinSim in interactive mode under the Debussy source level debugger, first replace the Debussy resource file debussy.rc you are currently using with the one downloaded from our website.

Add the following options to finvc:

+srcdbg -interactive

and the following options to finbuild:

-debussy_interactive

Run Debussy, select FinSim as your simulator and follow Debussy's instructions on how to run a simulation in the interactive mode.

12.3 Running Super-FinSim with Undertow

Under normal operation, Super-FinSim can generate a VCD dump file that can be viewed with Undertow. This is accomplished with the Verilog system task $dumpvars. However, VCD dump files tend to be large and inefficient. Instead, the user may wish to use the Optimizing Tool in Undertow to generate a more compact VCD file with the PLI routine $vtDump. In addition, connectivity data can be generated with the PLI routine $utConnectivity. With the connectivity data, Undertow can display net driver(s) during post simulation analysis. For complete information of all the PLI routines, refer to the documentation provided with Undertow.

In order to use Undertow’s PLI interface, copy the pre-defined PLI configuration files into the working directory. The file undertow.tab contains the Fintronic PLI table used by the compiler. The PLI definition file undertow.mak is used to build the simulator.

cp $FIN_INCLUDE_PATH/undertow.tab .

cp $FIN_INCLUDE_PATH/undertow.mak finpli.mak

Next invoke the compiler as follows:

finvc -ptab undertow.tab ....

13.0 Super-FinSim Implementation Notes

This chapter presents information pertaining to Fintronic's implementation of Verilog

13.1 Unsupported system tasks/functions

The Verilog system tasks/functions not supported yet are:

$input, $list, $incsave, $reset_count, $reset_value, $reset, $scope

All routines ass

13.2 Default files

13.2.1 Default VCD dump file

In the Super-FinSim Simulation Environment, the default VCD file is named 'finsim.dmp'.

13.2.2 Default simulation log file

In the Super-FinSim Simulation Environment, the default simulation log file is named 'finsim.log'.

13.2.3 Default simulation key file

In the Super-FinSim Simulation Environment, the default simulation key file is named 'finsim.key'.

13.2.4 Default SDF log file

In the Super-FinSim Simulation Environment, the default SDF log file is named 'finsdf.log'.

13.3 Super-FinSim system limitations

Super-FinSim limits the maximum value of simulation time to a 64-bit unsigned integer and limits the maximum bit width of an expression to be 1Mbits.

13.4 Limitations of the host 'C' compiler

Some large behavioral modules may lead to the generation of a C file that the C compiler cannot handle. Whenever a particular module cannot be compiled, it can be interpreted instead using the compiler option '-intm <module name>'.

