PRINCIPLES

OF

VERILOG PLI

PRINCIPLES

OF

VERILOG PLI

Swapnajit Mittra
Silicon Graphics Incorporated

SPRINGER-SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

Mittra, Swapnajit, 1969 -
Principles of Verilog PLI / Swapnajit Mittra.
p. cm.
ISBN 978-1-4613-7350-6 ISBN 978-1-4615-5161-4 (eBook)
DOI 10.1007/978-1-4615-5161-4

1. Verilog (Computer hardware description language) I. Title.
TK7885.7.M58 1999
621.39'2--dc21

99-18028
CIP

Copyright © 1999 by Springer Science+Business Media New York
Originally published by Kluwer Academic Publishers in 1999
Softcover reprint of the hardcover 1st edition 1999
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher, Springer-Science+Business Media, LLC

Printed on acid-free paper.

This printing is a digital duplication of the original edition.

This book is dedicated to all those who are eager to learn in a rapidly changing world, to those who teach and share knowledge, and to those who are determined to make a contribution through creative work and honesty.

Contents

PLI - A QUICK TOUR 1

1. WHAT IS A PLI ? 1
2. WHERE CAN I USE PLI ? 2
3. FOUR QUICK STEPS FOR WRITING A PLI ROUTINE 2
   3.1 Step 1: Including the header files 3
   3.2 Step 2: Function prototype and Variable declaration 3
   3.3 Step 3: The essential data structures 4
      3.3.1 veriuser_version_str 4
      3.3.2 endofcompile_routines 4
      3.3.3 err_intercept 5
      3.3.4 veriusertfs 5
      3.3.5 pli.tab file in VCS environment 6
   3.4 Step 4: *tf* routines 6
      3.4.1 The checktf routine 6
      3.4.2 The calltf routine 7
   3.5 Putting it all together 8
4. COMPILING AND LINKING A PLI ROUTINE 9
5. RESULTS 9
6. SELF CHECK 10
7. GETTING THE SIMULATION TIME 10
8. MODIFYING THE VALUE OF A REGISTER 11
   8.1 The *calltf* function 12
   8.2 Putting it all together 13
9. RUNNING AGAIN 15

10. SELF CHECK 16
11. SUMMARY 16

FUNDAMENTALS OF VERILOG PLI 17

1. INTRODUCTION 17
2. HOW PLI WORKS 19
2.1 Accessing information on variables 19
2.2 Accessing information on events 19
2.3 Accessing information and library functions 21
2.4 Integrating the PLI routine with the simulator 21
3. PROGRAM STRUCTURE OF A PLI ROUTINE 22
3.1 checktf 22
3.2 sizetf 23
3.3 calltf 24
3.4 misctf 24
4. PARAMETERS TO THE CONSTITUENT FUNCTIONS IN A PLI ROUTINE 25
4.1 *data* 26
4.2 *reason* and callback mechanism 27
4.2.1 reason_reactivate 28
4.2.2 reason_disable 28
4.2.3 reason_finish, reason_interactive 29
4.2.4 reason_synch, reason_rosynch 29
4.2.5 reason_paramvc 30
4.2.6 Other reasons 30
4.3 *paramvc* 31
5. INTEGRATING THE PLI ROUTINE INTO THE SIMULATOR 31
5.1 Interface in Verilog-XL 32
5.2 Interface in VCS 33
6. LINKING THE PLI ROUTINE WITH THE SIMULATOR 34
6.1 Linking with Verilog-XL 34
6.2 Linking with VCS 36
7. DIFFERENT TYPES OF LIBRARY FUNCTIONS 36
8. SUMMARY 37

UTILITY ROUTINES IN PLI1.0 39

1. WHAT ARE UTILITY ROUTINES ? 39
2. USING UTILITY ROUTINES 40
3. MODIFYING THE VALUE OF A DESIGN OBJECT 40

4. HANDLING DATA TYPES IN UTILITY ROUTINES 41
5. CLASSIFICATION OF UTILITY ROUTINES 41
5.1 Reading and modifying parameter values 43
5.1.1 Accessing 2-state values 43
5.1.2 Accessing 4-state values 45
5.1.3 Reading generic object value 48
5.1.4 Getting node information 52
5.2 Value change detection 56
5.2.1 Functions in this category 56
5.2.2 Purpose 57
5.2.3 Example 58
5.3 Simulation time 59
5.3.1 Getting simulation time 59
5.3.2 Timescale and precision of a module 60
5.3.3 Conversion between scaled delay and internal simulation time 60
5.3.4 Simulation synchronization 61
5.4 Displaying messages 63
5.4.1 Functions in this category 63
5.4.2 Purpose 63
5.4.3 Example 64
5.5 Instance pointer, module and scope instance name 64
5.5.1 Functions in this category 64
5.5.2 Purpose 65
5.5.3 Example 65
5.6 Saving information between calls 65
5.6.1 Functions in this category 65
5.6.2 Purpose 65
5.6.3 Example 66
5.7 Miscellaneous parameter related functions 66
5.7.1 Functions in this category 66
5.7.2 Purpose 66
5.7.3 Example 67
5.8 Stopping and finishing the simulation 67
5.8.1 Functions in this category 67
5.8.2 Purpose 67
5.8.3 Example 68
5.9 Getting plus arguments from the environment 68
5.9.1 Function in this category 68
5.9.2 Purpose 68
5.9.3 Example 68
5.10 Arithmetic and conversion functions 69
5.10.1 Functions in this category 69

5.10.2 Purpose 69
5.10.3 Example 70
5.11 Conversion between data types 70
5.11.1 Functions in this category 70
5.11.2 Purpose 71
5.11.3 Example 71
5.12 Save and restore operation 71
5.12.1 Functions in this category 71
5.12.2 Purpose 71
5.12.3 Example 72
6. USING INSTANCE POINTERS IN UTILITY ROUTINES 73
6.1 Using instance pointer 73
6.2 Functions for handling instance pointer 74
6.3 Functions which only work with an instance pointer 74
6.3.1 Functions in this category 74
6.3.2 Purpose 75
7. LIMITATIONS OF UTILITY ROUTINES 75
8. SUMMARY 76

ACCESS ROUTINES IN PLI1.0 77

1. WHAT ARE ACCESS ROUTINES ? 77
2. USING ACCESS ROUTINES IN A USER FUNCTION 78
3. HANDLE 79
4. CLASSIFICATIONS OF ACCESS ROUTINES 79
5. HANDLE ROUTINES 83
5.1 Getting a handle to a scope or object 83
5.1.1 Functions in this category 83
5.1.2 Purpose 84
5.1.3 Example 84
5.2 Getting a handle for the calling module and parent of an object 86
5.2.1 Functions in this category 86
5.2.2 Purpose 87
5.2.3 Example 87
5.3 Getting a handle to port and related objects 88
5.3.1 Functions in this category 88
5.3.2 Purpose 88
5.3.3 Example 89
5.4 Getting a handle to module path, datapath and related objects 90
5.4.1 Functions in this category 90
5.4.2 Purpose 91
5.4.3 Example 91

5.5 Getting a handle to timing check and related objects 93
5.5.1 Functions in this category 93
5.5.2 Purpose 94
5.5.3 Example 94
5.6 Getting a handle to simulated net and terminal 97
5.6.1 Functions in this category 97
5.6.2 Purpose 97
5.6.3 Example 97
5.7 Getting a handle to the parameters of a system call 98
5.7.1 Functions in this category 98
5.7.2 Purpose 98
5.7.3 Example 98
6. FETCH ROUTINES 99
6.1 Command line arguments 99
6.1.1 Functions in this category 99
6.1.2 Purpose 99
6.1.3 Example 99
6.2 Attribute, parameter, specparam 100
6.2.1 Functions in this category 100
6.2.2 Purpose 100
6.2.3 Example 101
6.3 Object name, type and location 103
6.3.1 Functions in this category 103
6.3.2 Purpose 104
6.3.3 Example 106
6.4 Delays 107
6.4.1 Functions in this category 107
6.4.2 Purpose 108
6.4.3 Example 110
6.5 Vector size and range 111
6.5.1 Functions in this category 111
6.5.2 Purpose 111
6.5.3 Example 111
6.6 Task/function arguments 112
6.6.1 Functions in this category 112
6.6.2 Purpose 113
6.6.3 Example 113
6.7 Timescale, precision 114
6.7.1 Functions in this category 114
6.7.2 Purpose 115
6.7.3 Example 115
6.8 Direction/value of a port 116
6.8.1 Functions in this category 116

6.8.2 Purpose 116
6.8.3 Example 116
6.9 Module path 118
6.9.1 Functions in this category 118
6.9.2 Purpose 118
6.9.3 Example 119
7. MODIFY ROUTINES 121
7.1 Delay values 121
7.1.1 Functions in this category 121
7.1.2 Purpose 122
7.1.3 Example 123
7.2 Logic values 128
7.2.1 Functions in this category 128
7.2.2 Purpose 128
7.2.3 Example 130
7.3 Pulse handling values 132
7.3.1 Functions in this category 132
7.3.2 Purpose 133
7.3.3 Example 133
8. VALUE CHANGE LINK (VCL) ROUTINES 134
8.1 Functions in this category 135
8.2 Purpose 135
8.3 Example 137
9. NEXT ROUTINES 141
9.1 Objects related to a module 142
9.1.1 Functions in this category 142
9.1.2 Purpose 143
9.1.3 Example 144
9.2 Objects related to the design hierarchy 145
9.2.1 Functions in this category 145
9.2.2 Purpose 146
9.2.3 Example 147
9.3 Objects related to driving or loading of a net 148
9.3.1 Functions in this category 148
9.3.2 Purpose 149
9.3.3 Example 149
9.4 Objects related to a vector 150
9.4.1 Function in this category 150
9.4.2 Purpose 150
9.4.3 Example 150
9.5 Objects related to a module path 152
9.5.1 Functions in this category 152
9.5.2 Purpose 152

9.5.3 Example 153
10. MISCELLANEOUS ROUTINES 154
10.1 Managing the acc routine environment 154
10.1.1 Functions in this category 154
10.1.2 Purpose 155
10.1.3 Example 158
10.2 Accessing all target objects simultaneously 159
10.2.1 Functions in this category 159
10.2.2 Purpose 159
10.2.3 Example 160
10.3 Comparing and matching objects 160
10.3.1 Functions in this category 160
10.3.2 Purpose 161
10.3.3 Example 162
11. USING ACCESS ROUTINES WITH VCS 162
12. SUMMARY 164

PUTTING PLI TO WORK 165

1. PROBLEM 1: GETTING INFORMATION FROM THE ENVIRONMENT 166
1.1 Approaching a solution 166
1.2 Program listing 167
1.3 Results 168
1.4 Self check 168
2. PROBLEM 2: CREATING A TEST BENCH 168
2.1 Approaching a solution 170
2.2 Program listing 171
2.3 Results 173
2.4 Self check 173
3. PROBLEM 3: A SIMPLE VALUE CHANGE DUMP TOOL 174
3.1 Approaching a solution 175
3.2 Program listing 177
3.3 Self Check 182
4. PROBLEM 4: A COVERAGE TOOL FOR GATE LEVEL NETLIST 182
4.1 Approaching a solution 184
4.2 Program listing 186
4.3 Self check 190
5. PROBLEM 5: COMMUNICATING BETWEEN VERILOG PROCESSES 191
5.1 Approaching a solution 194

5.2 Program listing 197
5.3 Self check 204
6. PROBLEM 6: VIRTUAL MEMORY MANAGEMENT 204
6.1 Approaching a solution 207
6.2 Program listing 208
6.3 Self check 216
7. PROBLEM 7: ASSIGNING A REGISTER VALUE FROM A FILE217
7.1 Approaching a solution 218
7.2 Program listing 219
7.2.1 calltf.c 219
7.2.2 lex_me.c 221
7.3 Self check 226
8. SUMMARY 226

WRITING BUS FUNCTIONAL MODEL 227

1. WHAT IS A BUS FUNCTIONAL MODEL ? 227
1.1 Behavioral / Functional Model 228
1.2 Register Transfer Level (RTL) Model 228
1.3 Structural Model 228
2. WHY USE A BFM ? 229
3. GENERAL METHODOLOGY FOR WRITING A BFM 230
4. WRITING A BUS FUNCTIONAL MODEL – A CROSSBAR SWITCH 231
4.1 Problem description 231
4.2 Algorithm for the crossbar 231
4.3 Implementation 232
4.4 Results 235
4.5 Self check 236
5. WRITING A BUS FUNCTIONAL MODEL – A RUDIMENTARY PROCESSOR 237
5.1 Problem description 238
5.2 Implementation overview 239
5.3 Forming the State Machine 240
5.4 Algorithm for the state machine 241
5.5 The algorithm for processor write 243
5.6 Algorithm for read 244
5.7 Implementation of the algorithm 245
5.8 Discussion of the code 252
5.8.1 Calling misctf routine 252
5.8.2 Declaring variables in misctf 252
5.8.3 Reasons 253

5.8.4 Reading the input file 253
5.8.5 Assigning a value to a named register 254
5.8.6 Future expansion of the instruction set 254
5.9 Self check 254
6. SUMMARY 256

VERILOG PROCEDURAL INTERFACE (PLI2.0) 257

1. WHY ANOTHER INTERFACE ? 257
2. ANATOMY OF A PLI ROUTINE USING VPI 258
2.1 User C functions 258
2.1.1 sizetf 259
2.1.2 calltf 259
2.1.3 compiletf 259
2.2 The registration function 259
2.3 The start-up array vlog_startup_routines[] 261
2.4 A simple example – $add_int() 261
2.5 Integrating user application with Verilog simulator 262
3. DATA ACCESS METHODOLOGY 262
3.1 Handle and its declaration 263
3.2 Associating a handle to an object 263
3.2.1 Relationship between two objects 264
3.2.2 Class and method 267
3.2.3 Tagged relationship 268
3.3 Getting handles to 'basic' objects 268
3.3.1 Getting handles to the top level modules 268
3.3.2 Getting handle to the parameters passed to the current system call 269
4. DESIGN OBJECTS – PROPERTIES AND RELATIONSHIPS 270
4.1 Module (vpiModule) 272
4.1.1 Category 272
4.1.2 Properties 272
4.1.3 Relationships 273
4.2 Scope (vpiScope) 274
4.2.1 Category 274
4.2.2 Constituent objects/classes 274
4.2.3 Relationship among the constituent objects 275
4.2.4 Properties 275
4.2.5 Relationships 275
4.3 Task and Function 275
4.3.1 Category 275
4.3.2 Constituent objects 276

4.3.3 Relationships between the constituent objects 276
4.3.4 Properties 276
4.3.5 Relationships 276
4.4 Task and Function call 276
4.4.1 Category 276
4.4.2 Constituent objects 277
4.4.3 Relationships among the constituent objects 277
4.4.4 Properties 277
4.4.5 Relationships 277
4.5 User-defined system call (vpiUserSystf) 278
4.5.1 Category 278
4.5.2 Property 278
4.5.3 Relationships 278
4.6 IO declaration (vpiIODecl) 278
4.6.1 Category 278
4.6.2 Properties 278
4.6.3 Relationships 279
4.7 Ports (vpiPort) 279
4.7.1 Category 279
4.7.2 Constituent objects 279
4.7.3 Relationships between the constituent objects 280
4.7.4 Properties 280
4.7.5 Relationships 281
4.8 Nets (vpiNet) 281
4.8.1 Category 281
4.8.2 Constituent objects 281
4.8.3 Relationship between the constituent objects 281
4.8.4 Properties 281
4.8.5 Relationships 283
4.9 Regs (vpiReg) 283
4.9.1 Category 283
4.9.2 Constituent objects 283
4.9.3 Relationships between the constituent objects 284
4.9.4 Properties 284
4.9.5 Relationships 284
4.10 Variables (vpiVariables) 285
4.10.1 Category 285
4.10.2 Constituent objects 285
4.10.3 Relationships among the constituent objects 285
4.10.4 Properties 285
4.10.5 Relationships 286
4.11 Variable-select (vpiVarSelect) 286
4.11.1 Category 286

4.11.2 Properties 286
4.11.3 Relationships 287
4.12 Named event (vpiNamedEvent) 287
4.12.1 Category 287
4.12.2 Properties 287
4.12.3 Relationships 287
4.13 Memory (vpiMemory) 288
4.13.1 Category 288
4.13.2 Properties 288
4.13.3 Relationships 288
4.14 Memory word (vpiMemoryWord) 288
4.14.1 Category 288
4.14.2 Properties 288
4.14.3 Relationships 289
4.15 Parameter (vpiParameter) 289
4.15.1 Category 289
4.15.2 Properties 289
4.15.3 Relationships 290
4.16 Specparam (vpiSpecParam) 290
4.16.1 Category 290
4.16.2 Properties 290
4.16.3 Relationships 291
4.17 Defparam (vpiDefParam) 291
4.17.1 Category 291
4.17.2 Properties 291
4.17.3 Relationships 291
4.18 Parameter assignment (vpiParamAssign) 292
4.18.1 Category 292
4.18.2 Properties 292
4.18.3 Relationships 292
4.19 Primitive (vpiPrimitive) 292
4.19.1 Category 292
4.19.2 Constituent objects 292
4.19.3 Relationships among the constituent objects 293
4.19.4 Properties 293
4.19.5 Relationships 294
4.20 Primitive terms (vpiPrimTerm) 294
4.20.1 Category 294
4.20.2 Properties 294
4.20.3 Relationships 295
4.21 UDP definition (vpiUdpDefn) 295
4.21.1 Category 295
4.21.2 Properties 295

4.21.3 Relationships 296
4.22 UDP table entry (vpiTableEntry) 296
4.22.1 Category 296
4.22.2 Properties 296
4.22.3 Relationships 296
4.23 Module path (vpiModPath) 297
4.23.1 Category 297
4.23.2 Properties 297
4.23.3 Relationships 298
4.24 Path term (vpiPathTerm) 298
4.24.1 Category 298
4.24.2 Properties 298
4.24.3 Relationships 299
4.25 Intermodule path (vpiInterModPath) 299
4.25.1 Category 299
4.25.2 Properties 299
4.25.3 Relationships 299
4.26 Timing check (vpiTchk) 300
4.26.1 Category 300
4.26.2 Properties 300
4.26.3 Relationships 300
4.27 Timing check term (vpiTchkTerm) 301
4.27.1 Category 301
4.27.2 Properties 301
4.27.3 Relationships 301
4.28 Continuous assignment (vpiContAssign) 302
4.28.1 Category 302
4.28.2 Properties 302
4.28.3 Relationships 302
4.29 Simple expression 302
4.29.1 Category 302
4.29.2 Constituent objects/classes 302
4.29.3 Relationships among the constituent parts 303
4.29.4 Properties 303
4.29.5 Relationships 303
4.30 Part select (vpiPartSelect) 303
4.30.1 Category 303
4.30.2 Properties 304
4.30.3 Relationships 304
4.31 Operation 304
4.31.1 Category 304
4.31.2 Properties 304
4.31.3 Relationships 304

4.32 Constant (vpiConstant) 305
4.32.1 Category 305
4.32.2 Properties 305
4.32.3 Relationships 305
4.33 Expression (vpiExpr) 305
4.33.1 Category 305
4.33.2 Constituent objects/classes 305
4.33.3 Relationships among the constituent objects 306
4.33.4 Properties 306
4.33.5 Relationships 306
4.34 Event statement (vpiEventStmt) 306
4.34.1 Category 306
4.34.2 Properties 306
4.34.3 Relationship 306
4.35 Assignment (vpiAssignment) 307
4.35.1 Category 307
4.35.2 Properties 307
4.35.3 Relationships 307
4.36 Delay control (vpiDelayControl) 307
4.36.1 Category 307
4.36.2 Properties 308
4.36.3 Relationships 308
4.37 Event control (vpiEventControl) 308
4.37.1 Category 308
4.37.2 Properties 308
4.37.3 Relationships 308
4.38 Repeat control (vpiRepeatControl) 309
4.38.1 Category 309
4.38.2 Properties 309
4.38.3 Relationships 309
4.39 While, Repeat and Wait statements 309
4.39.1 Category 309
4.39.2 Constituent objects 309
4.39.3 Relationship among the objects 310
4.39.4 Properties 310
4.39.5 Relationship 310
4.40 For statement (vpiFor) 310
4.40.1 Category 310
4.40.2 Properties 310
4.40.3 Relationships 311
4.41 Forever statement (vpiForever) 311
4.41.1 Category 311
4.41.2 Properties 311

4.41.3 Relationship 311
4.42 If and If-Else statements 311
4.42.1 Category 311
4.42.2 Constituent objects 312
4.42.3 Relationship among the constituent objects 312
4.42.4 Properties 312
4.42.5 Relationship 312
4.43 Case statement (vpiCase) 312
4.43.1 Category 312
4.43.2 Properties 312
4.43.3 Relationships 313
4.44 Case item (vpiCaseItem) 313
4.44.1 Category 313
4.44.2 Properties 313
4.44.3 Relationships 313
4.45 Force and procedural continuous assignment 314
4.45.1 Category 314
4.45.2 Constituent objects 314
4.45.3 Relationship between the constituent objects 314
4.45.4 Properties 314
4.45.5 Relationships 314
4.46 Release and Deassign statement 314
4.46.1 Category 314
4.46.2 Constituent objects 315
4.46.3 Relationship between the constituent objects 315
4.46.4 Properties 315
4.46.5 Relationships 315
4.47 Disable statement (vpiDisable) 315
4.47.1 Category 315
4.47.2 Properties 315
4.47.3 Relationship 316
4.48 Atomic statement 316
4.48.1 Category 316
4.48.2 Constituent objects/classes 316
4.48.3 Relationships among the constituent objects/classes 316
4.48.4 Properties 317
4.48.5 Relationships 317
4.49 Block statement 317
4.49.1 Category 317
4.49.2 Constituent objects 317
4.49.3 Relationships among the constituent objects 317
4.49.4 Properties 317
4.49.5 Relationships 317

4.50 Statement (vpiStmt) 318
4.50.1 Category 318
4.50.2 Constituent objects/classes 318
4.50.3 Relationship among the constituent objects/classes 318
4.50.4 Properties 318
4.50.5 Relationships 318
4.51 Process (vpiProcess) 318
4.51.1 Category 318
4.51.2 Constituent objects 318
4.51.3 Relationships among the object 318
4.51.4 Properties 319
4.51.5 Relationships 319
4.52 Callback 319
4.52.1 Category 319
4.52.2 Property 319
4.52.3 Relationships 319
4.53 Time queue (vpiTimeQueue) 320
4.53.1 Category 320
4.53.2 Property 320
4.53.3 Relationship 320
4.54 Summary 320

LIBRARY FUNCTIONS IN VPI 321

1. DIFFERENT CLASSES OF VPI FUNCTIONS 321
2. FUNCTIONS IN VPI 322
2.1 Getting a handle to an object from its properties 322
2.1.1 Functions in this category 322
2.1.2 Purpose 323
2.1.3 Example 323
2.2 Accessing properties of an object 323
2.2.1 Reading the properties of an object 323
2.2.2 Reading/modifying object delays 324
2.2.3 Reading/modifying simulation value of object 327
2.3 Getting the current simulation time 330
2.3.1 Function in this category 330
2.3.2 Purpose 330
2.3.3 Example 330
2.4 Simulation related callbacks 331
2.4.1 Functions in this category 331
2.4.2 Purpose 331
2.4.3 Example 334

2.5 Registering and getting information about a user-defined system call 335
2.5.1 Functions in this category 335
2.5.2 Purpose 335
2.5.3 Example 336
2.6 Traversing relationships 336
2.6.1 Functions in this category 336
2.6.2 Purpose 336
2.6.3 Example 337
2.7 File related activities 337
2.7.1 Functions in this category 337
2.7.2 Purpose 337
2.7.3 Example 338
2.8 Miscellaneous functions 339
2.8.1 Displaying messages 339
2.8.2 Comparing objects 339
2.8.3 Getting error information 340
2.8.4 Product invocation information 341
2.8.5 Freeing allocated memory 342
3. USING VPI: AN X-DETECTOR 343
3.1 Problem description 343
3.2 Program listing 343
3.3 Self Check 347
4. USING VPI: DISPLAYING DRIVERS 348
4.1 Problem description 348
4.2 Program listing 348
4.3 Self Check 352
5. FUTURE DIRECTION 352
6. SUMMARY 353

Foreword

Programming Language Interface (PLI) has shaped Verilog HDL usage in many different ways. Although the concept of extending an application through a programming interface (API) is not a new one, Verilog HDL pioneered the field in applying the concept to hardware designs.

Even after 20 years of HDL usage in the industry at large, the hardware engineers remain skeptical of their ability to use a programming language. The power of combining a hardware description language with a programming language such as C or C++ has been proven. A number of applications ranging from waveform viewers and debugging environments to delay calculators and co-verification products have been created and successfully used. Yet, the capability of extending Verilog HDL based design environment has been limited to only a few experienced hardware designers who dare to dabble equally in their software skills.

I had my first encounter with PLI in 1986-87 when the size and power of Sun-3 workstations, though considered leading-edge at the time, were quite inadequate for ASIC and board simulations for the next generation hardware. Due to our familiarity with remote procedure calls (RPC), our team decided to explore the possibility of building a distributed simulation environment. We started with less than 30 PLI routines, and requested a number of additional routines that eventually made their way into supported products. The experiment allowed us to build considerably large simulations than what was possible on a single machine, though performance gain was never significant. Since then, I have used PLI extensively for various debug, verification and analysis purposes.

From those early days, PLI has grown, matured and transformed itself into a part of IEEE standard. It used to be just a collection of *tf* utility routines, which was later augmented with *acc* netlist traversal routines. Later, a more powerful and concise form of *vpi* routines, also called PLI 2.0, were created.

Swapnajit has put in a significant effort to collate PLI related information from several sources. He has explained the concepts, routines and their applications in a very organized and easy to understand manner. The binding methods for different simulators have been described to make sure that your learning curve is short no matter which simulator you use. Each PLI routine is explained in great detail with its purpose, input arguments, return values, and an example or two showing proper usage. You will find the "gotcha" sections the most useful as you start writing your own applications.

The examples chapter, my personal favorite, is where the readers will find the most value. The author has taken a range of examples from simple and straight forward to quite complex. For each example, a clear problem statement is presented, a logical approach to solving the problem at had is discussed, and the code is built in small fragments. Then, the entire program is presented (code listing) and the results from actual simulation run (execution) are shown. The inter-process communication and the virtual memory management examples would appeal to even the most competent programmers among the hardware designers.

If you understand the problem statement and the approach to solving the problem, you will enjoy the extensions suggested in the "Self Check" section - test your understanding right there. There is likely to be something for everyone and every situation!

It is encouraging to see an engineer with full-time project responsibilities has taken time to share wealth of his experience with the rest of us.

Good luck in your quest for building more powerful design environments with Verilog HDL.

Yatin Trivedi
San Jose

Preface

The last decade has seen a dramatic change in the way hardware components and systems are designed. In the early eighties, most of the digital integrated circuits were relatively simple and were designed using schematic-based software. The situation started changing rapidly in the mid-eighties once the now famous Moore's law took over the semiconductor technology. While this opened a new horizon for hardware designers, soon it was apparent that schematic-based design entry methodology would not be able to sustain this exponential growth in the design complexity, primarily for two reasons. First, as the number of individual modules in a design went up drastically, they became unmanageable in a schematic-based design environment. Also, it was becoming increasingly clear that such an environment would not be able to provide a suitable design verification process required for the complex systems.

Primitive versions of proprietary hardware description languages (HDL) used to exist even earlier, but they were viewed mostly as peripheral utilities bundled with their parent schematic capture tools. With the tremendous growth in design complexity, these HDLs took the center stage and slowly but certainly started replacing the schematic capture tools. This shift in design methodology started a new era in digital design. A new set of HDLs was born which were rich in features and easier to use than their predecessors.

As all of us are aware today, among these new genres of languages, only Verilog and VHDL survived the race. Even today, the debate between die-hard fans of these two languages claiming superiority of one to the other is still very much alive. An often-sited advantage of Verilog over VHDL is Verilog's Programming Language Interface (PLI) mechanism. VHDL, in spite of its robust design, has no such application program interface (API) and the talk of a standardized API for VHDL is still in a nascent state.

The primary inspiration for writing this book came from numerous postings on the internet usenet group *comp.lang.verilog* asking about the basic PLI methodology. Although several books are available on Verilog, until now the vendor's manuals and the standards have been the only source of information on PLI. This, in my view is one of the main reasons why many hardware design engineers still perceive PLI as a 'software thing' and shy away from this powerful feature of Verilog.

Taking a different approach from the manuals and standards, this book has been written keeping in mind the real life problems that one faces in using PLI. Also, conscious effort was made to keep the C programs simple, modular and easy to understand. This has been done to dispel the fear that you need to be a software guru to use PLI.

PREREQUISITES

This book assumes that you have a good command over both C and Verilog. You should also have preliminary knowledge of data structures, algorithms and program development.

DESCRIPTION OF THE CHAPTERS

Conceptually, this book can be divided into three parts. Chapters 1 and 2 introduce the basic PLI methodology. Chapters 3 to 6 give the structural and syntactical details of functions in PLI1.0 including their applications. Chapter 7 and 8 are dedicated to the use of the new PLI2.0 or VPI methodology.

In their classic *The C Programming Language*, Brian Kernighan and Dennis Ritchie pointed out that "...the way to learn a new language is to write programs in it." Even though it is not a language, it applies equally well to PLI methodology also. This book starts with a tutorial in Chapter 1 giving the essential structure of a PLI routine and the basic set of library functions. This will give the reader a jump-start for writing simple applications.

Chapter 2 discusses the basic mechanisms of interfacing a PLI application with a Verilog program. It starts with an explanation of the basic data structures involved and gradually goes into the details of more advanced features of PLI, such as reasons and callback.

The main backbone of the PLI methodology is its sets of library functions. The first of these sets, the utility routines, is discussed in Chapter 3.

Chapter 4 discusses the access routines, the second generation of PLI functions.

Chapter 5 shows how to employ the library functions, discussed in the previous two chapters, for real life applications.

Today hardware models play an important role in design verification. The importance of hardware models is evident from the fact that many vendors have introduced behavioral models as separate commercial products. PLI is the pivotal element in this new arena. This special application of PLI, often termed as Bus Functional Modeling, is the subject of Chapter 6.

Chapter 7 introduces the basic concepts of Verilog Procedural Interface (VPI) or PLI2.0. It covers the data access methodology and different types of relationship.

Library functions of VPI are discussed in Chapter 8. Also included are a few examples of using these functions.

Throughout the book, exercises are provided at the end of a topic wherever required. I hope that it will intrigue the reader to explore the subject related to that topic further.

All updates and corrections to the text of this book can be found on the URL http://www.angelfire.com/ca/verilog/book_update.html.

NAMING CONVENTION

This book primarily follows the naming conventions adopted by IEEE Std.1364-1995 and OVI PLI reference manual. However, there are places where the conventional names have been replaced by new terms defined inside this book.

One example of such cases is the use of the term *system call* in place of *user-defined task and function*. In this book, the name *system call* is used for describing *any* PLI application, whether it is vendor supplied or user-defined. The standard vendor-supplied applications (such as `$display`, `$monitor`, etc.) are termed as *built-in system calls*, whereas custom-made user applications are called *user-defined system calls*. This has been done to emphasize that there is virtually no difference between these two types of applications as far as interfacing to the simulator is concerned.

Similarly, the term *PLI routine* is used to denote a user program which forms one or more applications.

Also, following the common convention in C, I preferred to use the term *function* instead of *routine* more often for a C function defined as part of the PLI library.

USE OF FONTS

The entire typesetting of this book has been done using Microsoft Word. Times New Roman font of point size 11 has been used for normal text. All new terms and keywords are mentioned in *italics* on their first occurrence. All program listings and terms from the code are mentioned using fixed width `Courier New` of point size 9.

ACKNOWLEDGMENT

So many people have offered their valuable suggestions and time on this project that I am afraid I might be missing any name here. First, my thanks to Meeta Gupta Bhaduri of HaL Computer Systems and Vinay Bhasin of CadWorx Inc. for completing the colossal task of reading the entire manuscript and contributing criticisms, ideas and corrections. Rajesh Bawankule of Comit Systems Inc. gave me an interesting set of problems which are used in different places in the book. Yatin Trivedi of Seva Technology provided some important suggestions regarding the examples in Chapter 5. Ravindra Divekar of Philips Semiconductor helped me in testing many of the examples. I also owe much to all the people who sent suggestions on comp.lang.verilog in reply to my postings.

I also remain indebted to Dave Kelf of Cadence Design Systems and Dave Burow, Marie Ann White and her team in Synopsys Inc. for granting copyright permission for their respective software tools.

I started the preliminary work for this book during my days with WIPRO Infotech Ltd. in India. Starting from what was intended to be a tutorial on PLI for colleagues, it took three years for it to transform itself into a book. During this time, I have also worked in HaL Computer Systems and Silicon Graphics Inc. I am grateful to all my colleagues in these three organizations for their help and cooperation.

It was also a pleasure to work with Carl Harris and his colleagues at Kluwer Academic Press. I am thankful to them for their help and patience.

Finally, my thanks to all those who shared their expertise during the photo-typesetting process, including Steve "Code Complete" McConnell, the people at Microsoft Press and the great folks of *comp.fonts*.

Swapnajit Mittra
Campbell, California

Chapter 1

PLI - A QUICK TOUR

Let us begin with a quick introduction to Programming Language Interface (PLI) of Verilog HDL. In this chapter, we will go through the essentials of PLI mechanism without getting bogged down by too much of details. Our aim of this chapter would be to acquire a workable knowledge of PLI so that we can start writing useful routines at the earliest. We will take an example of a very basic PLI activity - reading the design database. Further, we will modify it to explore some other basic features of PLI. We will end the chapter with a review of the features covered in this chapter and how they form the basis for moving to the details of PLI.

1. WHAT IS A PLI ?

Programming Language Interface (PLI) is a mechanism to provide an Application Program Interface (API) to Verilog HDL. Essentially it is a mechanism to invoke a C function from a Verilog code. The construct which invokes a PLI routine in a Verilog code is usually called a *system call*, if it is built-in to the simulator, and a *user-defined task or function*, if it is written by the user. Since, the essential mechanism for PLI remains the same in both the cases, throughout this book, we will use the term system call to indicate both. The common system calls built-in to most Verilog simulators are `$display`, `$monitor`, `$finish` etc.

S. Mittra, *Principles of Verilog PLI*
© Kluwer Academic Publishers 1999

2. WHERE CAN I USE PLI ?

As the above examples of built-in PLI routines indicate, PLI is primarily used for doing such things which would not have been possible otherwise using Verilog syntax. For example, while IEEE Std. 1364-1995 Verilog has a way of doing file write (`$fwrite`, which is another built-in system call written using PLI), it does not have any such predefined construct for reading a register value directly from a file. It is possible to use PLI to write a system call which does this.

Some other common examples where PLI is the only way of achieving the desired results are:

- Writing functional models
- Delay calculations
- Getting design information - for example, there is no Verilog construct which gives the name of the parent of the current module in the design hierarchy.

3. FOUR QUICK STEPS FOR WRITING A PLI ROUTINE

The file containing a PLI routine, by convention, is called `veriuser.c`. Although this name can be changed, this is the default name assumed by the *vconfig* tool while generating the compilation script in Verilog-XL[1] environment (We will discuss about this later in this chapter). For our discussion, we will assume that our PLI routine will be kept in the file `veriuser.c`.

To illustrate the basic steps for creating a PLI routine, we will take the following problem. This problem is much simplified compared to an actual 'real life' problem solved using PLI. However, this example will show the basic steps that build up a PLI routine.

Problem description: Create a system call `$print_reg` which takes a register as parameter and prints its value. (This is a simplified version of `$display`).

As an example, a Verilog code like this:

```
module my_module;
reg  [3:0] r1;
initial  begin
      r1 = 4'ha;
```

[1] Verilog-XL™ is a registered trademark of Cadence Design Systems, Inc.

```
#100 $print_reg(r1);
#100 r1 = 4'h3;
#100 $print_reg(r1);
#100 $finish;
endmodule
```

Should produce the values 10 and 3 at times 100 and 300.

To create a PLI routine which will do this, we will break up the task into four steps. These steps are given below.

3.1 Step 1: Including the header files

The file `veriuser.c` must start with the following line(s)

```
#include <veriuser.h>
#include <vxl_veriuser.h>
```

in a Verilog-XL environment and

```
#include <vcsuser.h>
```

in a VCS[2] environment.

The header file(s) contains the most basic data-structure of the Verilog-PLI that the program will use later on.

3.2 Step 2: Function prototype and Variable declaration

This part of the program contains the declarations for all variables and function prototypes that the PLI routine uses. In the present case, as explained in the next step, it will be as shown below.

```
int my_calltf(), my_checktf();
```

If the functions are in separate files, they should be declared as external functions.

```
extern int my_calltf(), my_checktf();
```

[2] The VCS simulator is a copyrighted works of Synopsys, Inc. and VCS™ is a trademark of Synopsys, Inc.

A typical PLI routine, just as any other C program, may need few other housekeeping variables.

3.3 Step 3: The essential data structures

There are a number of data structures that must be defined in a PLI routine. These are the variables through which Verilog simulator communicates with the C code. Although Open Verilog International (OVI), an organization for promoting and standardizing Verilog language, recommends only one mandatory data structure (`veriusertfs`), the exact number and syntax of these data structures vary from simulator to simulator. For example, Verilog-XL from Cadence Design Systems and Polaris from Avant! require four such data structures and functions for any PLI routine to work, whereas VCS from Synopsys needs none of them, but uses a separate input file instead. Going by the popularity of Verilog-XL, we are going to mention all four of them here (Section 3.3.1 to 3.3.4) *in the order they should appear in the program.* The data structure required for VCS will be given in Section 3.3.5.

3.3.1 veriuser_version_str

This character pointer can be used to indicate a version of the compiled Verilog for your reference. It can also be set to NULL string. The string appears at the standard output and also goes to the Verilog log file when you run your compiled Verilog. A typical example of using it is shown below.

```
char *veriuser_version_str ="\n\
============================\n\
 VERILOG PLI FIRST_ATTEMPT   \n\
       ($print_reg)          \n\
============================\n";
```

3.3.2 endofcompile_routines

This data structure, as the name suggests, can be used for declaring functions that will be invoked at the end of the simulation. The present example does not need it. Nonetheless there has to be a default definition as shown below.

```
int (*endofcompile_routines[])() ={0};
```

3.3.3 err_intercept

This function, which returns either true or false (i.e. a boolean type), can be used to enhance the error detection mechanism of the code and can be ignored for small applications. A typical example of this routine is shown below.

```
bool err_intercept(level, facility, code)
int level; char * facility; char *code;
{ return (true); }
```

3.3.4 veriusertfs

The main interaction between the PLI routine and the Verilog simulator is done through a table. The Verilog simulator looks at this table and figures out the properties that the system call corresponding to this PLI routine would be associated with. This table is called `veriusertfs`. Note that, it may *not* be called by any other name as the simulator will not recognize that. Each element of `veriusertfs` has a distinct function and is very important to achieve the overall objective of writing the PLI routine.

The number of elements in `veriusertfs` is same as that of the number of user-defined system calls, plus one for the last entry, which is a mandatory *{0}*.

In the present case, the `veriusertfs` array should look like:

```
s_tfcell veriusertfs[] =
{
  /*** Template for an entry :
  {usertask | userfunction,
   data,
   checktf(),
   sizetf(),
   calltf(),
   misctf(),
   "$tfname"
   },
   ***/
{usertask, 0, my_checktf, 0,
 my_calltf, 0, "$print_reg"},
{0} /* Final entry must be zero */
}
```

In the above data-structure, `my_checktf` and `my_calltf` are the names of the two functions that we will use to implement the system call `$print_reg`. `my_checktf` is the routine which checks the validity of the passed parameters etc. and `my_calltf` perform the main task of the system call. These names are arbitrary and could have been something else. The first entry `usertask` indicates that the system call does not return anything and is equivalent to `procedure` in Pascal or `function` returning `void` in C. Note that:

- The relative positions of these names in `veriusertfs` are very important - for example `my_checktf` (or any other name) *must* be the 3rd element, as long as we want to use it as a checking function.
- Any function that we do not use (e.g. misctf in our example. The objective of providing so many functions will be discussed in Chapter 2) should be replaced by a '0'.
- We need to define a separate entry in `veriusertfs` for each system call in case there are more than one.

3.3.5 pli.tab file in VCS environment

In VCS environment, instead of defining a table in the program to indicate which functions are being used, this information is written in a separate file, usually called `pli.tab` (once again the name could be anything). The contents of this file, in the present example of `$print_reg`, is as shown below.

```
$print_reg check=my_checktf  call=my_calltf
```

3.4 Step 4: *tf* routines

3.4.1 The checktf routine

The checktf routine, as mentioned before, checks the validity of the passed parameters. It is a good practice to check two things in a checktf routine.

- whether the total number of parameters are same as what is expected and
- whether each parameter is of the required type.

For example, in our case, we expect only one parameter to be passed and it must be a register type. This is done in the following function `my_checktf()`.

```
int my_checktf()
{
  if (tf_nump() != 1) {
    tf_error("$print_reg:");
```

```
      tf_error("Usage:$print_reg(register_name);");
    }
    if (tf_typep(1) != tf_readwrite) {
       tf_error ("$print_reg:");
      tf_error ("The argument must be a register type\n");
     }
}
```

The functions starting with `tf_` are the library functions and commonly known as *utility routines*. The library functions that we have used in the above function and their usages are given below.

`tf_nump()`: Determines how many parameters you are passing.

`tf_typep()`: Determines the type of the parameter by its position in the system call (in our case `$print_reg`) in the Verilog code. Thus `tf_typep(1)` gives the type of the first parameter, `tf_typep(2)` would give the type of the second parameter etc. In case the parameter does not exist (like `tf_typep(2)` in our case), `tf_typep()` returns error. In the present example, we expect register to be passed, the type should be `tf_readwrite`. In case of a wire, the type is `tf_readonly`. Note that, to avoid error condition checking, it is a good idea to check the number of parameters first, followed by their types.

`tf_error()`: Prints an error message and signals to the simulator to increment its error count.

These library functions and constants are defined in the header files that we included at the top.

3.4.2 The calltf routine

The calltf function is the heart of the PLI routine. It usually contains the main body of the PLI code. In our case, it should read the value of the register and then print it. The following code shows how to do that.

```
int  my_calltf()
{
  io_printf("$print_reg : Value of the reg=%d",tf_getp(1));
}
```

The library functions used above and their usages are given below.

`io_printf()`: Does the same thing what `printf()` does in normal C programming - printing the value on standard output. Additionally, it prints the same in the Verilog log file.

`tf_getp()`: Gets the integer value of the register.

In real life routines, the calltf function is the most complicated and often runs for several hundred lines.

3.5 Putting it all together

Now that we have created the two main functions needed for our present example, let us see how to put them together in one place. The following program listing shows how to implement `$print_reg` in a Verilog-XL environment.

```
#include <veriuser.h>
#include <vxl_veriuser.h>

int my_calltf(), my_checktf();

char *veriuser_version_str ="\n\
============================\n\
 VERILOG PLI FIRST_ATTEMPT   \n\
       ($print_reg)          \n\
============================\n";

int (*endofcompile_routines[])() ={0};

bool err_intercept(level, facility, code)
int level; char * facility; char *code;
{ return (true); }

s_tfcell veriusertfs[] =
{
  {usertask, 0, my_checktf, 0, my_calltf, 0,
  "$print_reg"},
 {0} /* Final entry must be zero */
};

int my_checktf()
{
  if (tf_nump() != 1) {
    tf_error ("$print_reg:");
     tf_error ("Usage : $print_reg(register_name);\n");
   }
  if (tf_typep(1) != tf_readwrite){
    tf_error ("$print_reg:");
```

```
        tf_error ("The argument must be a register type\n");
      }
  }

  void  my_calltf(){
    io_printf("$print_reg : Value of the reg=%d",
              tf_getp(1));
  }
```

4. COMPILING AND LINKING A PLI ROUTINE

Once a PLI routine is written and the corresponding system call is instantiated in a Verilog program, the next task is to make the Verilog simulator understand the existence of the call and properly execute it. This is done by compiling the PLI routine and then linking it with the existing binary of the simulator.

Although the integration of the PLI routine with the Verilog binary can be done manually by running C-compiler and merging the object files, for convenience it is usually done through a shell script. In Verilog-XL, a program called `vconfig` generates this scripts, which takes the name `cr_vlog` by default. While generating this script, the program `vconfig` asks the name of the compiled Verilog that you prefer. It also asks whether to include model libraries (some of which are pre-developed PLI routines) from standard vendors. For most of these questions, the default answer (if you press return without entering anything) is good enough unless you need a customized environment. At the end, `vconfig` asks for the path for your `veriuser.c` file(s). Once the script `cr_vlog` is generated, just by running it one can get the new Verilog simulator binary which would recognize the user defined system call.

The compilation process for VCS will be discussed in Section 8. Also, we will take up the topic of linking more elaborately in Section 6 of Chapter 2.

5. RESULTS

A sample run of the compiled Verilog having our PLI routine produces the following output with Verilog-XL simulator.

```
=============================
 VERILOG PLI FIRST_ATTEMPT
       ($print_reg)
=============================
Compiling source file "test.v"
Highest level modules:
my_module
$print_reg: Value of the reg=10
$print_reg: Value of the reg=3
L9 "test.v": $finish at simulation time 400
15 simulation events
CPU time: 0.2 secs to compile + 0.1 secs to link + 0.0
secs in simulation
End of VERILOG-XL 2.2.1 Jun 1, 1997 22:44:02
```

6. SELF CHECK

– Repeat this problem by replacing the register by a wire. The system call should read and print the value of the wire. Run the PLI routine on a net which is `tri1`, rather than `wire`. Check its value when there is no active driver of the net.

7. GETTING THE SIMULATION TIME

Although the PLI routine in the previous example worked as expected, the output of the program shows a serious weakness of the routine – there is no way to figure out the simulation time when the register changed the value. With multiple invocation of this routine (as in this case) this problem becomes even worse.

This can be solved using another utility function `tf_gettime()` which tells the current time of the simulation. This has been done in a new revision of our calltf routine below by making minor modification to the previous one. Also for convenience, the value of the register is displayed in hexadecimal rather than decimal.

```
void  my_calltf(){
  io_printf("$print_reg: Value of the reg=%h at time=%d\n",
            tf_getp(1), tf_gettime());
}
```

As shown in the routine, `tf_gettime()` returns the current simulation time and it does not need any input parameter. A sample run of the modified version of our routine give the following output.

```
=============================
 VERILOG PLI FIRST_ATTEMPT
       ($print_reg)
=============================
Compiling source file "test.v"
Highest level modules:
my_module
$print_reg: Value of the reg=a at time=100
$print_reg: Value of the reg=3 at time=300
L9 "test.v": $finish at simulation time 400
15 simulation events
CPU time: 0.2 secs to compile + 0.1 secs to link + 0.0
secs in simulation
End of VERILOG-XL 2.2.1 Jun 1, 1997 23:44:02
```

8. MODIFYING THE VALUE OF A REGISTER

If reading the design information is one reason why PLI is used, the other reason would be to modify this information in the design database. The following example shows how it is done.

Problem description: Create a system call `$invert` to print the value of a register (just like what was done in the last example). Now bit-wise invert the contents of the register and print this updated value too. (Or in other words, at the end the register will have one's complement of its current contents.)

There are more than one ways to obtain the inverted value. The algorithm described below may not be the smartest as far as doing one's complement is concerned, but it would give more insight into the PLI mechanism and library functions.

Step 1: Read the contents of the register is *as a string*;
Step 2: Make the following modifications on the string:
Flip all 1s in the string to 0s and all 0s to 1s;
All Zs become X and Xs, if any, remain intact.
Step 3: Put back the modified string into the register.

8.1 The *calltf* function

The checktf function in this case will not be different from the earlier one as in both the cases the number and the type of the input parameter is same. The following code shows the `invert_calltf`, the calltf function for the `$invert` routine.

```
int invert_calltf() {
char *val_string;
/* The string which temporarily
   holds the value of the reg */

    /* Step 1 of the algorithm */

    val_string = (char *) malloc(tf_sizep(1)+1);
    /* +1 to accommodate the null char at the end */
    strcpy(val_string, tf_strgetp(1, 'b'));

    /* Step 2 of the algorithm */

    move('1', '2', val_string);
    move('0', '1', val_string);
    move('z', 'x', val_string);
    move('2', '0', val_string);

    /* Step 3 of the algorithm */

    io_printf("$invert: %s --> %s at time %d\n",
              tf_strgetp(1, 'b'), val_string,
              tf_gettime());
    tf_strdelputp(1, tf_sizep(1), 'b', val_string, 0, 0);
}
```

The library functions used above are explained below.

`tf_strgetp()`: Reads the contents of the register as a string, just like `tf_getp()` reads it as a decimal. In the above example, `tf_strgetp(1, 'b')` reads the contents of the first parameter in binary format ('h' or 'o' would have read it in hexadecimal or octal format). Then it is copied to a string `val_string`.

`tf_strdelputp()` : Writes value to parameter from string value specification. This function takes a number of arguments. In order of their position, they are : the parameter index number (the relative position of the

parameters, starting with 1 for the leftmost), size of the string whose value is going to be put in (which, in the present case, should be same as the size of the register), the encoding radix, the actual string and two other delay related arguments, which we chose to ignore by passing 0s for them. (Although it will not be used in the present example, it is worth noting that, there is a simpler decimal counterpart of `tf_strdelputp()` too. It is called `tf_putp()`).

One important point to remember here is that the contents of only certain objects can be changed or overwritten from a PLI routine. Any attempt to change a variable which can not be put on the left hand side of a procedural assignment in Verilog, will result in an error. In general, you can modify the contents of a variable only if its type is `tf_readwrite`. (This has been checked in the checktf function `my_checktf()`).

`move()`, a function yet to be defined, would replace each occurrence of the first parameter (a character) by the second parameter (another character) in the third parameter (a string). In the present case, a series of calls to `move()` would change the 0s to 1s and vice versa. Note that, once 0s are converted to 1s, we need to distinguish between them and the original 1s in the string. To solve this, at first all 1s are changed to 2 – an invalid value for binary logic, but a work around for us. At the end, all 2s are converted back to 0s.

The program completes its task by putting back the inverted value back into the register using `tf_strdelputp()` function.

Here is the code for `move()`.

```
void move(from, to, in_str)
char from, to, *in_str;
{
  int i=0;
  while(*(in_str+i)) {
    if (*(in_str+i) == from)
      *(in_str+i) = to;
    i++;
  }
}
```

8.2 Putting it all together

The complete listing for the entire PLI routine is given below. Note that this time it has been customized for VCS environment.

```
#include "vcsuser.h"

int invert_calltf(), my_checktf();
```

```
void move();

int my_checktf() {
  if (tf_nump() != 1) {
    tf_error("$invert:\n");
    tf_error("Usage: $invert(register_name);\n");
  }
  if (tf_typep(1) != tf_readwrite)
    tf_error ("$invert: Arg 1 must be of type reg\n");
}

int invert_calltf() {
char *val_string;
/* The string which temporarily holds the
   value of the reg */

  /* Step 1 of the algorithm */

  val_string = (char *) malloc(tf_sizep(1)+1);
  /* +1 to accommodate the null char
     at the end */
  strcpy(val_string, tf_strgetp(1, 'b'));

  /* Step 2 of the algorithm */

  move('1', '2', val_string);
  move('0', '1', val_string);
  move('z', 'x', val_string);
  move('2', '0', val_string);

  /* Step 3 of the algorithm */

  io_printf("$invert: %s --> %s at time %d\n",
            tf_strgetp(1, 'b'), val_string, tf_gettime());
  tf_strdelputp(1, tf_sizep(1), 'b', val_string, 0, 0);
}

void move(from, to, in_str)
char from, to, *in_str;
{
  int i=0;
  while(*(in_str+i)) {
```

```
      if (*(in_str+i) == from)
        *(in_str+i) = to;
      i++;
    }
  }
```

9. RUNNING AGAIN

A sample Verilog program containing the call `$invert` is shown below.

```
module mymodule;
reg [7:0] r1;
initial begin
      r1 = 8'hf;
#100  $invert(r1); // Invert the contents of r1
#100  $finish;
end
```

As mentioned earlier, in VCS environment the functions associated with a PLI routine are listed in a file. Although this file can have any name, it is usually called `pli.tab`. This file is equivalent of `veriusertfs[]` structure in Verilog-XL environment. The contents of this file in the present example is given below.

```
$invert  call=invert_calltf check=my_checktf
```

Assuming the above program is saved in the file `test.v`, an executable binary `pli_invert` is generated by giving the following command.

```
$ vcs -o pli_invert -P pli.tab test.v veriuser.c
```

Executing `pli_invert`, we get the following result.

```
Contains Chronologic Simulation proprietary information.
Oct 21 00:05 1997

$invert: Modifying the contents from 00001111 to 11110000
at time 100
$finish at simulation time                    200
   V C S   S i m u l a t i o n   R e p o r t
Time: 200
```

```
CPU Time:   0.067 seconds; Data structure size: 0.0Mb
Tue Oct 21 00:05:16 1997
```

10. SELF CHECK

- Modify the PLI routine `$invert()` to create a new system call `$twos_complement()`, which takes a register as its argument and replaces its contents by the two's complement of its present value. Remember, you should issue an error message and abort, if any bit of the register is Z or X. Repeat the problem again, so that the modified value of the register appears only 10 time units after the current simulation cycle.

11. SUMMARY

In this chapter, we have learnt the basic structure of a PLI routine and mechanism of linking it to build a custom version of Verilog simulator. We have also learnt how to read, convert and modify the values of elementary components of a design database in Verilog using PLI mechanism. And also how to read the simulation time. These are the basic and most often-needed tasks of a PLI routine.

Compared to the rest of the universe that PLI offers, however, this is really a small part. In later chapters we will see how PLI can be used to access other design information not related to the basic component – name of the current module and its parent, or controlling the path-delays, for example.

Also, we have so far used only the so-called 'utility routines' provided in PLI1.0. There is a complete different set of routines, known as *access routines*, which is often recommended for more uniform interfacing. We will learn about it in Chapter 4.

Chapter 2

FUNDAMENTALS OF VERILOG PLI

We have seen the general structure of a PLI routine in Chapter 1. In this chapter we will go into the details of each component of a PLI routine. This includes the discussion about all functions that build up a PLI routine and the set of pre-defined parameters passed to these functions. We will also discuss how the interfacing between the simulator and a user-defined call is done in different types of simulators.

1. INTRODUCTION

Most modern programming languages are designed to solve problems of a specific kind. For example, languages such as Perl or AWK provide constructs to make text processing easier, while C++ and Smalltalk focus on object oriented programming. But this specialized approach also makes it difficult to program a task in a language meant for another language. This is where *Programming Language Interface* or PLI comes into picture. In general, PLI refers to the mechanism that makes the resources of programs written in one language accessible to programs using another language. This allows the user to partition his program in two parts written in different languages and run them together. Usually the second language is chosen such that this process enhances the overall versatility of the first language as a whole. Often the two languages are selected from different programming paradigms. For instance, many database languages use, what is known as, '3GL interface' allowing users to write programs in C or C++ to manipulate the data structures defined inside them.

S. Mittra, *Principles of Verilog PLI*
© Kluwer Academic Publishers 1999

Similarly, Verilog, a hardware description language, often needs features of a standard procedural language. For example, standard Verilog does not have any concept of dynamic memory allocation for variables, which is a common methodology in procedural languages. However, in order to make programs written in Verilog more efficient in memory consumption, having such a mechanism would be really helpful.

One way of handling this would be to add the missing feature to Verilog. This is not a desirable solution though – borrowing features from another language may tend to dilute the primary focus of Verilog itself.

A more convenient solution will be to provide an interface to another language like C, which already has these features.

A PLI routine in Verilog can be thought of as an equivalent of a library function, defined using C, rather than Verilog itself. Many of these routines are needed so frequently that by default they are included into most popular Verilog simulators. Common examples of such routines or *built-in system calls* are `$monitor`, `$display`, `$showvars`, etc.

Apart from the built-in system calls, it is also possible for a user to define his own PLI routine and call it from his Verilog program. Such a *User defined system call* can be used to perform any unique task intended by the user. As an example, there are standard system calls available for writing to a file in Verilog (`$fdisplay`, `$fwrite`). But it does not have any such call for reading values from a file into a register. In this case, the user defined call can be customized to do something specific to the user's requirement, such as reading the value as a hexadecimal with zeroes inserted in the most significant empty bits.

Over the years, PLI has grown to evolve into different versions. There are two distinct standardized and well-accepted versions of PLI, called PLI1.0 and PLI2.0. These versions have been standardized by IEEE Std. 1364-1995 as well as by Open Verilog International (OVI). The libraries and facilities provided by the Verilog vendors are usually supersets of these standards. Although the underlying principles are same for both these versions, the libraries and the mechanism work completely independently. Most vendors at this point of time support only PLI1.0.

In this chapter we are going to focus on the PLI1.0 mechanism. PLI2.0 will be discussed in Chapter 7.

2. HOW PLI WORKS

To know how PLI mechanism works we need to know how a digital logic simulator works internally. When a Verilog program is simulated, the simulator generates data-structures and databases derived from the Verilog program. These data-structures can be broadly divided into two categories:

- Data-structures meant for variables (reg, wire etc.) of the Verilog program
- Data-structures for simulating the events defined by the Verilog program

Using certain rules/guidelines, Verilog PLI can be used to access information from both categories.

2.1 Accessing information on variables

As an example of how a simulator stores the information about a variable, let us take the case of a `reg` type variable. A register in Verilog can have values '1', '0', 'X' and 'Z'. Inside the simulator each register bit is represented by two integers, called the real field and the control field. A {real field, control field } combination of '00' means logic '0' , '10' means '1', '11' is 'X' and '10' represents 'Z'.

This means, in case we want to assign a value to the register, there are two possible ways of doing it. First, we can do a procedural assignment inside the Verilog program. Alternately, as one can guess from the discussion above and this is where the essence of PLI is, we can as well assign the required values to the real field and the control field of the register using a C program. Or in other words, without taking the help of Verilog language as such, it is possible to modify the design database that the simulator generates.

2.2 Accessing information on events

Once a Verilog program is submitted to the simulator for execution, it is translated into a sequence of events scheduled over the range of the simulation time. This sequence, known as the *event queue*, determines the order of execution of these events during the simulation. To illustrate how an event queue works, let us consider the following simple Verilog program.

```
module test;
reg r1, r2;
```

```
initial  begin
   r1 = 1'bz;              ...event1
   #100
   r1 = 1'b1;              ...event2
   r2 = 1'b0;              ...event3
   #100
   $finish;                ...event4
end

endmodule
```

The events generated from this Verilog program are marked against their corresponding lines in the above program. The event queue can be thought of as a two dimensional mesh of events and time. The events in our program are shown in the following diagram. At time 0, there is only one event corresponding to the assignment of the register r1. Similarly, at time 200, the program stops with the event4 associated with `$finish`. At time 100, there are two events related to the assignments of the registers r1 and r2. Note that, *in general* the order of execution of the events at the same simulation time slot (such as event2 and event3 in this example) are arbitrary and depends on several factors including event optimization by the simulator. It is also worth mentioning here that besides the events generated from the user Verilog program, the actual event queue may have many other simulator generated events (for example, all registers are initialized to x and real type variables to 0.0 at time 0).

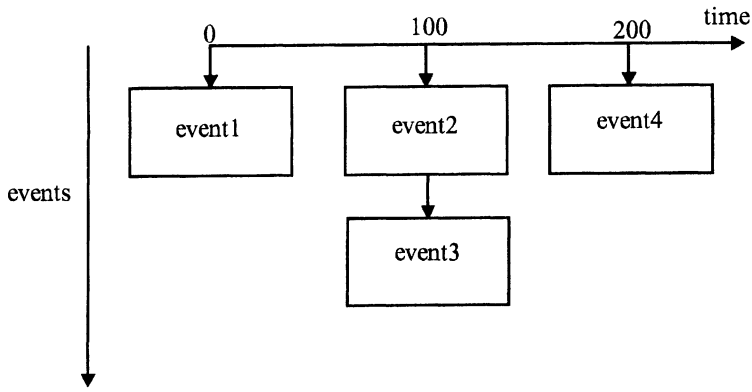

Figure 1. Event queue for our simple Verilog program

The event queue is generated initially from the input Verilog code. As the simulation progresses, new events are generated or previously scheduled events are canceled modifying the initial event queue.

Using standard library functions of PLI, it is possible to schedule an event explicitly at a future simulation time. This feature may be used to show the delay associated with modifying the value of an object in the design database. It can also be utilized to schedule an event at regular interval – each time the event scheduling itself for the next time after its own execution.

2.3 Accessing information and library functions

At this point, we must remember that there are several other data-structures inside the simulator and not all of them are accessible even using a PLI routine. For instance, in order to change the value of a register, we can not directly assign the values to the real and control fields of the register in the simulator database. This would ruin the data coherency inside the database giving erroneous results during the simulation. Instead, there are functions that allow us to access and, if required, to modify the values in the internal data-structure. These functions, available from standard PLI libraries, make sure that data coherency is maintained throughout the design database. The user is supposed to create the PLI routine using these library functions whenever he wants to access the design database. There are two library files associated with PLI1.0, commonly called `veriuser.h` and `acc_user.h`. The library function for PLI2.0 is called `vpi_user.h`.

2.4 Integrating the PLI routine with the simulator

Once the PLI routine is written, it can be called from a Verilog program by associating it with a system call. The name of the system call must start with '$', e.g. `$my_routine`.

The next step is to let the Verilog simulator know about the existence of the system call. This is done (as described in Chapter 1) using a table defined inside the PLI routine. In Verilog-XL, it is done through a pre-defined structure called `veriusertfs`, whereas VCS uses a file to read in the entries. We will study the fields of this tabular structure in details later in this chapter (Section 5). However, it is worth mentioning here that in this structure, besides other details, we need to specify the name of the system call and the functions associated with it.

In Verilog-XL, to simulate a program that uses the corresponding system call, a set of files are compiled and linked together to produce a compiled binary for the customized Verilog simulator. This set contains the original file containing the Verilog simulator binaries, the file containing the tabular structure `veriusertfs` (and certain other optional definitions of pre-defined data types) and the file(s) containing the actual PLI routine.

Notice that this whole task is independent of the Verilog program that utilizes this system call.

During the simulation, if the simulator comes across any system call, it first checks its database of built-in routines. If the system call is not found there, next the simulator refers to the user-defined list. Then, in turn, it invokes the compiled routines that define the activity associated with the system call.

VCS has a different approach for compiling PLI routine. In VCS, the input Verilog program is translated to C and compiled along with the PLI C routines to produce a single object file. This file is integrated with other files of the simulation environment during execution. We will discuss the details of this in Section 6.

3. PROGRAM STRUCTURE OF A PLI ROUTINE

A PLI routine corresponding to a system call consists of several different functions provided by the user. Although the actual tasks of these functions are completely user-defined, their overall functionalities should follow a set of guidelines (This is equivalent to `main()` in C, which is user-defined, but mandatory). As we will soon discover, each of these functions is planned to do a very well defined part of the entire job. Note that, each of the pre-defined functions, such as checktf, calltf etc., is invoked by the simulator itself. It is forbidden for these functions to invoke each other. For example, the calltf function can not call the checktf function etc. In this section, we will discuss the significance of using each of these functions. The following are the different functions that constitute a PLI routine.

3.1 checktf

As mentioned earlier, a system call is equivalent to a library function in C. A system call may take a set of parameters similar to a C function. But, unlike a C function, where the number of parameters and their data types are checked during compilation, there is no such mechanism in Verilog. The

only way to check this would be to define a function within the PLI routine which will do this explicitly. (An equivalent example of this in a C program would be to check the number and the values of command line arguments.)

These checks are incorporated in a checktf function. The actual name of the function can be anything, as long as it is passed to the `veriusertfs` structure in Verilog-XL environment or the tabular interface in VCS environment.

As these are the very first things to check before proceeding any further, checktf is the first function to be executed during a system call. This also allows the user to include any initialization inside checktf if required.

3.2 sizetf

Depending on whether a PLI routine returns some value at the end of the execution or not, it is termed as a *user-defined function* or a *user-defined task* respectively. Let us consider a function `foo` which returns n-bit value. Inside the Verilog program it is instantiated as shown below.

```
reg [n:0] result;
  result = $foo(<parameters>); // Instantiation 1
```

or may be as

```
some_module u1 ($foo(<parameters>),
                <other parameters>);
                                // Instantiation 2
```

Note that, as the return size is unknown to the Verilog simulator, there has to be a mechanism to pass this information to it from the PLI routine. The Verilog compiler can not determine this automatically as there may be multiple return points (for example, after checking the parameters in checktf, if anything is wrong, it might immediately return a single bit error flag). Also, the overhead of the Verilog compiler would be too much just to handle this dynamically.

In Verilog-XL, this problem is solved by using a sizetf function for user-defined functions. This function returns the size of the returned data structure in bits. Since tasks do not return a value, a sizetf function is not required for tasks.

Another interesting point to note here is that, if we are using the function to do something like the 'Instantiation 1' above, we could as well define it as a task as shown below.

```
$foo(result, <parameters>);
```

and modify the value of the `result` inside the calltf function (calltf function will be discussed in the next section). The choice between using a function or to do the chore inside a PLI routine depends on the complexity of the task. Also, keep in mind that this technique will not work in cases like the 'Instantiation 2' shown above.

VCS environment does not require a sizetf function as the return size is directly specified in the interface table to be discussed in Section 5.2.

In absence of a sizetf function, the default return size of a user-defined function is 32-bit.

3.3 calltf

calltf is similar to the function `main()` in a C program. It is the central program control unit of a PLI routine. Most of the actual task that the PLI routine is supposed to do is performed in this function. Calltf can further call a series of other functions that execute different sub-parts of the job.

3.4 misctf

As the name suggests, this function in a PLI routine performs a number of miscellaneous jobs. This function can be used to do any housekeeping jobs, such as, freeing up memory allocated to some variable or closing a file etc.

However, as explained below, there is an important distinction between misctf and the rest of the functions in the way they are executed in a PLI routine.

Remember that each of these functions can be invoked only by the simulator. While other pre-defined functions, such as checktf or calltf, are called only once per instantiation of the system call, misctf function can be called multiple number of times by the simulator. A misctf function can be asked to do different tasks depending on the condition for which it is called. As we will see in the next section, the real strength of a misctf function lies in its use with these conditions, commonly known as *reasons*.

4. PARAMETERS TO THE CONSTITUENT FUNCTIONS IN A PLI ROUTINE

A system call may have zero or more number of parameters depending on its use. For example,

```
$time; // Takes no parameter
$print_sim_time(); // Takes a null parameter
$print_reg(my_reg); // Takes one parameter
```

During the execution of a system call, the parameters passed to its constituent functions, such as calltf or misctf, are *completely different* from the parameters passed to the system call. In contrast to the actual system call, only a fixed number of pre-defined parameters can be passed to these functions. Since all constituent functions are invoked by the simulator, there may be two different types of parameters passed to them:

- A constant parameter which the user (i.e. the person writing the PLI routine) asks the simulator to pass to the functions when they are invoked (Note again, this is different from the parameters passed to the system call itself). This parameter is known as *data*.

Parameters which the simulator passes to the function indicating various states of the simulation. There are two such simulator generated parameters - *reason* and *paramvc*. Note that

- all of them are read-only and can not be altered from any of the functions.
- these parameters are optional; for example, even if you do not pass any parameter to calltf or misctf, the function still remains syntactically correct.
- when used, only a few specific combinations of these parameters are allowed.

The following table shows these valid combinations and their order in a function definition. In this example myFunc() is a function representing any of the checktf, calltf or misctf.

Table 1. Valid combinations for parameters.

Valid combination	Example
(no parameters)	int myFunc() { ... }
data, reason	int myFunc(int data, int reason) { }
data, reason, paramvc	int myFunc(int data, int reason, int paramvc) { }

Each of these parameters is discussed next.

4.1 *data*

Before describing what data is, let us consider the following problem: We would like to develop two system calls `$print_regb` and `$print_regh` to print the value of a register in binary and hexadecimal form respectively.

One obvious way of doing this would be to write them separately. But, looking into our program in Chapter 1, it is easy to figure out that, except for the part of the calltf which prints out the value of the register (essentially the `io_printf()` statement), rest of the code for these two applications would be identical. It makes more sense to share the same checktf and calltf between the applications, but inside the calltf to arrange for some mechanism to distinguish between the system calls `$print_regb` and `$print_regh`.

The data parameter can be used to associate a particular value with these two different system calls, when they activate the same set of functions. For example, we can pass 0 and 1 as data for `$print_regb` and `$print_regh` respectively – or more sensibly, since data can be any integer, 'b' for `$print_regb` and 'h' for `$print_regh`. A modified version of the function `my_calltf()` of Chapter 1 does exactly that by introducing a *switch* statement as shown below. It assumes a user-defined function `to_binary()` to return the binary value of the register as a string. (Remember, reason must be passed along with data as shown in Table 1).

```
int my_calltf(data, reason)
int data, reason;
{
   switch (data) {
   case 'b' :
      io_printf("$print_regb:\n");
      io_printf("%d: register=%s", tf_gettime(),
                to_binary(tf_getp(1) ));
   break;

   case 'h' :
      io_printf("$print_regh:\n");
      io_printf("%d: register=%h", tf_gettime(),
                tf_getp(1));
   break;

   default :
```

```
        tf_error("Trouble!!! You should not be here ...");
    }
}
```

Although we have used the data parameter here, we do not know yet how to assign a value to it. In Section 5, we will see how a data is actually passed to a function.

4.2 *reason* and callback mechanism

If data shows which system call caused the activation of a function, reason, as the name suggests, tells the reason behind it.

Reason is another integer parameter passed to the functions in a PLI routine. But, unlike data, reason can have only a pre-defined set of values given in `veriuser.h`. The value of reason is set by the simulator and the user has no control on it. For example, inside checktf, sizetf and calltf routines, the values of reason are predefined integers `reason_checktf`, `reason_sizetf` and `reason_calltf` respectively. But the primary strength of reason as a parameter is manifested when it is used inside misctf function. Depending on the context in which the misctf is called, the value of reason varies widely and can be used for a whole range of applications.

Perhaps the most important of them is the callback mechanism. Callback for an API application is defined as a mechanism to schedule a call to a specified function whenever a certain event occurs. Depending on this calling event, callback may be divided into the following categories.

- Simulation event related (e.g. whenever a net or its driver changes value etc.)
- Simulation action or feature related – when certain simulator activity occurs (e.g. end of compilation or end of save or restart).
- Simulation time related (e.g. scheduling an event at the end of the current event queue or at a future simulation time).

As a callback occurs, the misctf function is called with a reason indicating the type of callback. Actions corresponding to each value of the reason can be defined inside the misctf using a switch statement on reason. We will discuss the reasons associated with different types of callbacks in a short while.

The actual set of supported values may vary from simulator to simulator. Following are the standard values of reason as recommended by the IEEE Std. 1364-1995.

4.2.1 reason_reactivate

This value is assigned to reason during a previously scheduled callback of the misctf function at a particular simulation time.

This is done by using the function `tf_setdelay(n)`, where n is an integer. A call to `tf_setdelay(n)` from any function schedules the misctf routine n time units after the current simulation time. Such a call from the calltf function at simulation time t would schedule the execution of misctf to a time (t+n), whereas the same call inside the misctf would start a series of recursive calls to itself at an interval of n time units. In both cases, the value of reason passed to the misctf function is `reason_reactivate`.

This feature can be used to model, for example, a synchronous state machine by activating the misctf function periodically at an interval same as that of the clock period.

4.2.2 reason_disable

As the execution of a PLI routine may extend over a period of simulation time, it may so happen that the Verilog block which instantiates the corresponding system call might have been disabled.

As an example consider the following Verilog module: at the positive edge of each clock it processes the contents of an internal register using `$process_reg` system call. Also, if there is any rising edge of the input signal `interrupt` it disables the block which calls `$process_reg` and calls a task called `interrupt_subroutine`.

```
module reason_disable_example(clk, interrupt);
input clk, interrupt;
reg r1;
always @(posedge clk)
begin : main_job
    $process_reg(r1);
end
always @(posedge interrupt)
begin
    disable main_job;
    interrupt_subroutine();
end
task interrupt_subroutine;
begin
```

```
    /* whatever */
end
endtask
endmodule
```

In the case shown above, with the block `main_job` disabled, the misctf associated with `$process_reg` will be called with a reason `reason_disable`.

4.2.3 reason_finish, reason_interactive

In many cases, a PLI routine may extract the necessary information from the design database over a period of simulation time and the final calculation is done only at the end of the simulation. For example, a PLI routine `$power` which computes the avarage power consumed by a module, needs to monitor all the nets inside that module throughout the length of the simulation. The average power can be computed only at the end of the simulation. The misctf function is invoked with a reason `reason_finish` at the end of the simulation to handle this kind of situation.

Similarly, when the simulation enters the interactive mode after encountering `$stop()`, the misctf function is called with a reason `reason_interactive`. This can be used, for instance, to compute average power consumed by a module upto that point in simulation.

4.2.4 reason_synch, reason_rosynch

To illustrate the objective of these reasons, let us once again take the example of Chapter 1, where we changed the contents of a register. Let us further assume that the PLI routine modifies the contents of the register to `'b0`. Now consider a situation where the main Verilog code has another assignment changing the value of the same register to `'b1` at the same simulation time slot. At the end of that slot, the contents of the register could be either `'b0` or `'b1`, depending on how the two assignments are scheduled. Verilog does not guarantee any specific value for this kind of *race condition*, as the order of these assignments will depend on the nature of code optimization performed by the simulator.

Now suppose, we want the final value of the register at the end of the simulation time to be same as what we forced from our PLI routine. In such a case, we must schedule the assignment made by the PLI routine at the end of the event queue at that particular simulation time. To schedule an event

generated by the PLI routine at the end of the event queue at a particular time slot, the utility function `tf_synchronize()` is used. This function invokes misctf function at the end of the current simulation time slot with a reason `reason_synch`.

The utility function `tf_rosynchronize()` is similar to `tf_synchronize()` in operation, but it further guarantees that no new event would be generated for later simulation time slot. The reason associated with `tf_rosynchronize()` is `reason_rosynch`. Although this function provides more control over the scheduling, this also means calling a function which might add an event in the same time slot, such as, `tf_strdelputp()` and `tf_setdelay()`, is illegal when the reason is `reason_rosynch`.

4.2.5 reason_paramvc

This is another powerful use of the reason argument. `reason_paramvc` is associated with the utility function `tf_asynchon()`. Once `tf_asynchon()` is called, whenever there is a change in value in any of the arguments passed to the system call, the misctf function will be invoked with a reason `reason_paramvc`. The effect of `tf_asynchon()` can be disabled by calling `tf_asynchoff()`.

An example of its usage would be to implement a system call similar to the built-in `$monitor`, which lists the values of all arguments, whenever any of them changes.

4.2.6 Other reasons

IEEE Std 1364-1995 also recommends the following values of reason. They are seldom used and not supported by all simulators.

reason_endofcompile : At the end of the compilation but before starting the simulation.

reason_paramdrc : Callback due to change in value of the driver of any argument passed to the system call.

reason_force, reason_release : Associated with force and release of any argument to the system call.

The following values are associated with a built-in system call:

reason_scope : associated with `$scope()`.

reason_startofsave, reason_save : assigned to reason at the beginning and the completion of `$save()` respectively.

reason_restart : assigned to reason while executing `$restart()`.

reason_reset, reason_endofreset : associated with the beginning and the end of execution of `$reset()`.

4.3 *paramvc*

The third parameter passed to the function misctf is called *paramvc*. This is a special purpose integer parameter used along with the function `tf_asynchon()`. As mentioned earlier, this function enables misctf to be called whenever a parameter value changes. In such a call, paramvc indicates the position of the changed parameter in the argument list (the left-most parameter being number 1). Using in association with a reason `reason_paramvc`, this parameter can be used to model any asynchronous event, such as an interrupt or trap mechanism.

5. INTEGRATING THE PLI ROUTINE INTO THE SIMULATOR

Once a PLI routine is written, it is invoked from the Verilog program by instantiating the corresponding system call. The name of a system call must start with a '$', e.g., `$my_routine()`.

During compilation, if the tool comes across a term starting with '$', it first checks whether the term is a built-in system call. If it does not find the term in the list, then the tool consults its internal database to find out if it is a user-defined system call. Once the term is confirmed to be a user-defined system call, the simulator executes the associated functions.

The interface between the PLI routine and the simulator is provided in a table which indicates, among other details, the following:

- the nature of the routine - whether it is a function (returning a value) or a task (returning void or nothing).
- names of the functions which form the routine.
- name of the system call corresponding to the PLI routine.

In this section, we are going to discuss the details of the interfacing mechanisms that Cadence Verilog-XL and Synopsys VCS simulators use. Although there are other vendors who support PLI, it is believed they will not be significantly different from these two.

5.1 Interface in Verilog-XL

Verilog-XL has a pre-defined data type `s_tfcell` which is defined as an array of the following items in order:

Table 2. Entries in the veriusertfs table

Entry no.	Must be	Data type	Used for indicating
1	Either `usertask` or `userfunction`	int	Whether the PLI routine is a usertask (returning void or nothing) or a userfunction (returning non-zero number of bits) to the environment
2	Valid data or 0, if unused	int	Data parameter
3	Checktf function name or 0, if unused	Function pointer	checktf function
4	Sizetf function name or 0, if unused	Function pointer	sizetf function
5	Calltf function name (Unlike other function name, this is not optional)	Function pointer	calltf function
6	Misctf function name or 0, if unused	Function pointer	misctf function
7	A $-prefixed string within double quotes	char *	Name of the system call

For detailed discussion on checktf, sizetf, calltf and misctf functions, refer to Section 3 earlier this chapter.

There may be few other optional entries after this. They can be ignored for most of the cases.

Each PLI routine has its own set of entry of `s_tfcell` data type to indicate the interface. All the PLI routines together makes an array of `s_tfcell` called `veriusertfs[]`. This array has all the entries of the PLI routines and a mandatory last entry of {0}. A typical `verisertfs[]` table is shown below.

```
s_tfcell veriusertfs[] =
{
/*** Template for an entry :
{usertask|userfunction,data,checktf(),
 sizetf(),calltf(), misctf(), "$tfname"}
***/
```

```
{usertask, 0, my_checktf1, 0, my_calltf1, 0, "$rtn1"},
{userfunction, 0, 0, my_sizetf2, my_calltf2, my_misctf2,
 "$rtn2"},
{usertask, 'h', my_checktf3, 0, my_calltf3, 0, "$fn_h"},
{usertask, 'b', my_checktf3, 0, my_calltf3, 0, "$fn_b"},
{0} /* Final entry must be zero */
}
```

Each system call must have an entry enclosed within '{' and '}' in the `veriusertfs` structure.

In the above example, the first entry says that the system call `$rtn1` returns void after its execution (indicated by `usertask`) and it uses `my_checktf1` and `my_calltf1` as its checktf and calltf routines. Also, it does not have any misctf routine, as indicated by a 0 in the corresponding place. The second entry is for a user-defined function `$rtn2`, which does not have a checktf function, but uses `my_sizetf2`, `my_calltf2` and `my_misctf2` as its sizetf, calltf and misctf function. The third and fourth system calls use the same functions for their checktf and calltf, but, as explained in Section 4.1, the data parameters are different for them ('h' and 'b'). For all user task, the corresponding field for a sizetf function must be 0.

5.2 Interface in VCS

In the VCS environment, unlike the `veriusertfs[]` array of Verilog-XL, there is no pre-defined data type to be initialized to define the interface. Instead, the user needs to enter the names of the constituent functions etc. in an ASCII text file, normally referred to as the PLI table file or *pli.tab*. Additionally, as we will see in Chapter 4, this interface file may also contain other information related to the scope of the library functions, which are not part of the interface.

Each line in a PLI table file corresponds to one PLI routine (similar to each `s_tfcell` entry to `veriusertfs[]` in Verilog-XL environment). Each entry starts with the name of the system call followed by the names of different functions associated with the call. Remember the entire entry must be in a single line.

`$`*name*`[call=<`*function*`>] [check=<`*function*`>] [data=<`*num*`>] [misc =<`*function*`>] [size=<`*num*`>]`

It should be noted that in VCS environment

- The relative position of the function names in the line is irrelevant, as long as it is preceded by one of the keywords (call, check, data etc.) and a "=".
- All constituent functions, such as calltf or misctf, are optional and can be omitted.
- An entry in the PLI table file with just the system call name can be used to disable the system call.
- Unlike Verilog-XL, there is no sizetf function. The return size (in bits) is indicated as an integer.

A PLI table file, equivalent to the `veriusertfs[]` shown earlier, is given below.

```
$rtn1 check=my_checktf1 call=my_calltf1
// The following line assumes my_sizetf2 returns 16 bits.
// Also note that comments like this are allowed inside
// PLI table file.
$rtn2 size=16 call=my_calltf2 misc=my_misctf2
// The following two use same functions, but
// different data
$fn_b data='b' check=my_checktf3 call=my_calltf3
$fn_h data='h' check=my_checktf3 call=my_calltf3
```

6. LINKING THE PLI ROUTINE WITH THE SIMULATOR

Linking the PLI routine with the main simulator is very similar to linking a library in a C program. Once the PLI routine is written, it is compiled to an object format and linked with the current simulation environment. But, since the simulators in VCS and Verilog-XL work in different ways, the linking procedures are also different.

6.1 Linking with Verilog-XL

Figure 2 shows how to link a PLI routine with the main simulator in Verilog-XL environment. At the end of this process, a new simulator binary is generated. This new simulator would be able to parse *any* Verilog program that uses the PLI routine (along with all other normal language constructs).

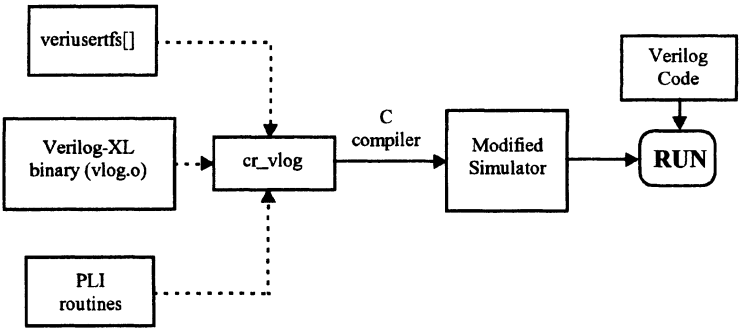

Figure 2. Linking PLI routines with Verilog-XL

The process of recompilation starts with running a Cadence supplied program called `vconfig` to generate a shell script. The compiled binary is generated by running this script. `vconfig` prompts the user for a number of options. These include:

- preferred names for the generated script and the compiled binary.
- whether to add a set of facilities to the standard language construct (for example, delay annotation or interface to certain library vendors etc.). A default option is provided in each case. Most often the default answers are sufficient.
- the list of file(s) which contain the PLI routine(s). This list should include the `veriuser.c` (or its equivalent by any other name) and any other file(s) where PLI functions are defined. At the end, one should put a "." as the file name in the list, which tells `vconfig` that the list is complete.

The default name of the shell script that `vconfig` generates from user input, is `cr_vlog`. This shell script usually contains three sections:

- setting up the path for the C compiler and other environment variables.
- compilation command for `veriuser.c` and other input C file, if any, to object code
- command for binding all the object codes generated in the last operation with the basic simulator binary (`vlog.o`) to a compiled binary.

If you are familiar with the shell programming, a quick look at `cr_vlog` will reveal that it is a text file containg a simple shell script. This means it can be edited manually if required i.e. new files can be added to the compile list without re-running `vconfig`. However, for more significant changes, such as, integrating a new third party vendor library, it is better to run `vconfig` all over again.

6.2 Linking with VCS

VCS translates the input Verilog program to an equivalent C program. The simulation behavior is produced when this C program is compiled and executed. Since PLI routines are also written in C, linking a PLI routine in this case is similar to binding two or more C programs. This is done from the command prompt as shown below. The following command generates an incrementally compilable (through `-M` flag) binary file `my_vcs` (through `-o` flag, the default name is `simv`) which incorporates all the system calls given in the PLI table file `my_pli_table.file` with their corresponding functions given in the files `pli1.c`, `pli2.c` and `pli3.c`.

```
$ vcs -M -o my_vcs -P my_pli_table.file input.v pli1.c
pli2.c pli3.c
```

`vcs` internally invokes the C compiler to compile the given files. Note that, unlike the re-compilation in Verilog-XL, here the binary executable `my_vcs` is only meant for the Verilog program `input.v`. To simulate another Verilog program which uses the same PLI routine, one needs to recompile again replacing `input.v` with the corresponding file for the new Verilog program. Also, if there is any change in any of the input files (PLI or Verilog) you need to recompile again. The process is shown in Figure 3.

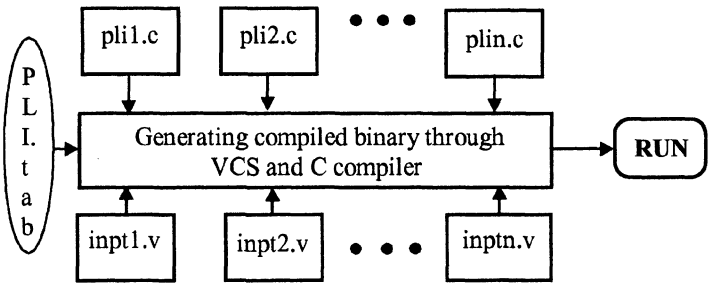

Figure 3. Linking PLI routines in VCS

7. DIFFERENT TYPES OF LIBRARY FUNCTIONS

As mentioned in Section 2, the internal database generated by the simulator can be accessed by the user only through a pre-defined set of library functions. These functions are essentially the backbone of the whole

PLI mechanism. They act as a transparent interface hiding the complex software structures of the design database from the PLI user, who are primarily hardware engineers. On the other hand, they protect the integrity of the database by restricting user access only to certain data structures.

The library functions in PLI1.0 can be divided into two classes. The first type of functions are primarily used for reading and modifying the arguments to the system calls and for checking error conditions. These are known as the *utility routines*.

The second class of functions are called the *access routines*. They are used to read from and write to generic Verilog HDL objects by accessing internal data structures. Access routines are more diverse and powerful compared to the utility routines. Also, on many occasions utility routines can be replaced by access routines. On the other hand, utility routines are easier to use and, if need arises, to debug.

According to IEEE 1364 standard, all utility routines in PLI1.0 start with `tf_`, whereas access routines have a prefix `acc_`.

8. SUMMARY

In this chapter, we have seen how the interfacing mechanism between a PLI routine and the Verilog simulator works. The program structure for a PLI routine has been discussed giving details of the functionalities of each constituent function and the set of pre-defined parameters passed to them. Lastly, we have seen how to integrate a PLI routine into the simulator and how this process differs in different commercial simulators.

Chapter 3

UTILITY ROUTINES IN PLI1.0

In Chapter 2, we have seen how to integrate a user-defined system call with the main simulator. Also, we found that in order to access the internal database of the simulator, a PLI routine must use a set of library functions. In this chapter, we are going to learn about the first group of these library functions, known as *Utility routines*. We will start with a general discussion of utility routines and certain conventions that they follow. To take a closer look at these functions, we will divide them into different categories according to their functionalities and discuss each of these categories separately. We will wind up the chapter with a discussion on comparative merits and demerits of using utility routines.

1. WHAT ARE UTILITY ROUTINES ?

Utility routines are library functions defined in PLI1.0 to perform a wide variety of operations on the parameters passed to the system call. They can also be used to do many other things, such as simulation synchronization or to implement conditional program breakpoints.

Utility routines are the first generation of PLI routines. All of them (except `io_printf()`, `io_mcdprintf` and `mc_scan_plusarg()`), start with the prefix `tf_`. For this reason, they are also called tf routines.

S. Mittra, *Principles of Verilog PLI*
© Kluwer Academic Publishers 1999

2. USING UTILITY ROUTINES

Utility routines are mainly used for reading and manipulating the parameters passed to a user defined system call. Most of the utility routines take one of these parameters itself as their arguments. However, instead of passing it by its actual symbol, a parameter is passed to a utility routine by its *parameter index number* (PIN) – its position in the parameter list of the corresponding system call with the leftmost parameter numbered as 1.

As an example, consider the following instantiation of a user-defined system call `$my_pli_routine`.

```
$my_pli_routine(some_signal, another_signal, a_register);
```

Let us assume, inside the system call, we need to read the value of `some_signal`. As we have seen in Chapter 1, in order to read the value of a parameter with PIN=`n`, we should use the utility routine `tf_getp(n)`. In this case, since `some_signal` is the first parameter in the argument list (or its PIN is 1), `tf_getp()` should be called as shown below.

```
int val_param1;
val_param1 = tf_getp(1);
```

Similarly, in case of the parameter `a_register`, which is 3rd in the list, an equivalent call would look like

```
int val_param3;
val_param3 = tf_getp(3);
```

3. MODIFYING THE VALUE OF A DESIGN OBJECT

One of the main uses of utility routines is to read or modify the value of design objects, which are passed to the system call as parameters. However, as we have seen in Chapter 2, we can not modify the value of any arbitrary design object, as that may destroy the data coherency in the design database. Keeping this in mind, all the design objects accessible to the user via utility routines can be divided into two sets.

An object is *writable*, when it can be used on the left hand side of a procedural assignment statement of Verilog source description. Otherwise, it

is *read-only*. For example, objects of data type reg, integer, real or time are writable, whereas nets and names of objects (module name, named block, UDP instance name etc.) are read-only.

Only the value of a writable object can be modified using utility routines; read-only objects can still be accessed to read their values, but any modification is prohibited. Any attempt to modify an object, which is read-only, will usually result in no action or a return value of 0 as an indication of error. As is obvious, we can always read the value of a writable object.

4. HANDLING DATA TYPES IN UTILITY ROUTINES

All utility routines assume a 32-bit implementation of the C `int` data type. It further assumes any `long` is 64-bit, but can be expressed as a concatenation of two 32-bit integers.

When reading or modifying a scalar parameter using utility routines, unless the parameter is of type `time`, its numeric value is usually obtained or specified as a 32-bit quantity. This means, for example, while reading a 1-bit `reg` or `wire` using the function `tf_getp()` mentioned above, an integer will be returned. The user should take special care (for example, by using another function `tf_strgetp()`, rather than `tf_getp()`) if the register/wire is more than 32-bit wide.

Verilog data type `time` is assumed to be 64-bit and is returned in two parts, each of them a 32-bit `int`. Consequently, values of all delays or simulation times are read as two integers containing lower and higher order 32 bits. Since a C function can return only one variable, an integer pointer is passed as a place holder to the utility routine. After returning from the function, the returned integer and the value of the place holder together give the full 64-bit data.

Verilog data type `real` is treated as `double` in C.

5. CLASSIFICATION OF UTILITY ROUTINES

The utility routines play an important role in the overall PLI methodology by providing a handshaking mechanism between the user C program and the Verilog design database. However, to provide interface to a wide range of objects, the total number of such functions is considerably high (IEEE Std. 1364-1995 recommends 65 basic utility routines and most

of them have a second version to use with an instance pointer) and it is no longer easy for a beginner to remember all of them. In this section, we are going to categorize the utility functions into different groups, emphasizing the behavior of each group rather than their constituent functions.

The following table shows which utility routine to use for a specific purpose.

Table 3. List of utility routines in PLI1.0

Purpose	Function
Reading or modifying parameter value	tf_getp()
	tf_getlongp()
	tf_getrealp()
	tf_putp()
	tf_putlongp()
	tf_putrealp()
	tf_strgetp()
	tf_getcstringp()
	tf_strdelputp()
	tf_strlongdelputp()
	tf_strrealdelputp()
	tf_exprinfo()
	tf_evaluatep()
	tf_nodeinfo()
	tf_propagetep()
To detect parameter value change	tf_asynchon()
	tf_asynchoff()
	tf_copypvc_flag()
	tf_movepvc_flag()
	tf_testpvc_flag()
	tf_getpchange()
Getting simulation time and related things	tf_gettime()
	tf_getlongtime()
	tf_getrealtime()
	tf_strgettime()
	tf_gettimeunit()
	tf_gettimeprecision()
Synchronizing simulation	tf_synchronize()
	tf_rosynchronize()
	tf_getnextlongtime()
	tf_setdelay()
	tf_setlongdelay()
	tf_setrealdelay()
	tf_clearalldelays()
Displaying message	io_printf()
	io_mcdprintf()
	tf_warning()

Purpose	Function
	tf_error()
	tf_text()
	tf_message()
Getting instance pointer, module and scope instance name	tf_getinstance()
	tf_mipname()
	tf_spname()
Saving information between calls	tf_getworkarea()
	tf_setworkarea()
Getting parameter related information	tf_nump()
	tf_typep()
	tf_sizep()
Stopping or finishing simulation	tf_dostop()
	tf_dofinish()
Getting plus argument from the environment	mc_scan_plusargs()
Arithmetic and conversion	tf_compare_long()
	tf_add_long()
	tf_subtract_long()
	tf_multiply_long()
	tf_divide_long()
	tf_long_to_real()
	tf_longtime_tostr()
	tf_real_to_long()
Save or restore	tf_write_save()
	tf_read_restart()
Scaling/Unscaling delay	tf_scale_longdelay()
	tf_scale_realdelay()
	tf_unscale_longdelay()
	tf_unscale_realdelay()

We will spend the rest of this section, by dividing the utility functions into different groups and discussing each of these groups separately.

5.1 Reading and modifying parameter values

Depending on how the value of an object is treated, the functions for reading and modifying parameter values can be divided into the following categories.

5.1.1 Accessing 2-state values

5.1.1.1 Functions in this category

```
int tf_getp(pin)
int pin;
```

```
int tf_getlongp(higher_32, pin)
int higher_32, pin;

double tf_getrealp(pin)
int pin;

void tf_putp(pin, value)
int pin, value;

void tf_putlongp(pin, lower_32, higher_32)
int pin, lower_32, higher_32;

void tf_putrealp(pin, value)
int pin;
double value;
```

5.1.1.2 Purpose

These functions are used to read (the 'get' functions) or modify (the 'put' functions) the 2-states value of an object. In 2-states value, each bit can hold only 1 or 0, but not X or Z.

Each of the 'get' functions takes `pin`, the PIN of the parameter it wants to read as its argument. For instance, to read upto 32-bit long numeric value of an object, `tf_getp()` is used. For reading 64-bit or floating point numeric values in Verilog, `tf_getlongp()` and `tf_getrealp()` are used respectively. If the value contains logic X or Z, a 0 is returned in that bit position.

Similarly, for 'put' functions, the new value of the object is passed as argument(s) along with its PIN. Depending on the type of a writable object, different functions are used to modify its value. For example, for scalar or vectored `register` or `integer` variables, `tf_putp()` is used, whereas for `real` and `time` variables `tf_putrealp()` and `tf_putlongp()` are used respectively.

Additionally, when used from calltf, `tf_putp()`, `tf_putrealp()` and `tf_putlongp()` can be used to set the return value of the system call by using a PIN 0. If there is no legal value specified for such a call (for example, `tf_putp(0,)`), the default return value is set to 0.

5.1.1.3 Example

A basic example of how to use `tf_getp()` has been mentioned in Section 2 of this Chapter. The following example shows how to read a 64-

bit long value (such as `time` data type) using the function `tf_getlongp()`. The least significant 32 bits are returned as an integer and the most significant 32 bits are stored in an integer variable passed to the function. For example, if the first parameter of a user-defined system call is `time` data type, the following code segment fetches the value into two integers `lower_32` and `upper_32`.

```
int lower_32, upper_32 ;
lower_32 = tf_getlongp(upper_32, 1);
```

5.1.2 Accessing 4-state values

5.1.2.1 Functions in this category

```
char *tf_strgetp(pin, format)
int pin, format;

char *tf_getcstringp(pin)
int pin;

int tf_strdelputp(pin, bitlength, format,
                  val_str, delay, delay_type)
int pin, bitlength, format, delay_type;
int delay;
char *val_str;

int tf_strlongdelputp(pin, bitlength, format,
                  val_str, delay_lo32,
                  delay_hi32, delay_type)
int pin, bitlength, format, delay_type;
int delay_lo32, delay_hi32;
char *val_str;

int tf_strrealdelputp(pin, bitlength, format,
                  val_str, delay, delay_type)
int pin, bitlength, format, delay_type;
double delay;
char *val_str;
```

5.1.2.2 Purpose

This set of functions are used to read or modify a 4-state object value. Unlike the 2-state functions shown before, these functions take care of all possible 4 states of a bit – 0, 1, X and Z. As X or Z can not be expressed in an integer form, the value is represented as a string.

The function `tf_strgetp()` returns the value of the passed parameter as a string in one of the binary, octal, decimal or hexadecimal formats, as specified by its second argument `format`. `format` can have one of the following pre-defined values.

Table 4. Different values of `format`

Value of `format`	Format returned
`'b'` or `'B'`	Binary
`'o'` or `'O'`	Octal
`'d'` or `'D'`	Decimal
`'h'` or `'H'`	Hexadecimal

If the parameter happens to be a string itself, `tf_strgetp()` returns the literal string ignoring the value of `format`.

`tf_getcstringp()` also returns a string, but all leading zeroes are dropped and each 8 bits starting from the least significant bit are converted to corresponding ASCII characters. If the passed parameter itself is a string, `tf_getcstringp()` returns the literal string.

All 'put' functions modify the value of the parameter specified by `pin`. The new value and its bit length are passed to the functions as `val_str` and `bit_length` respectively. The functions also provide a facility for specifying a finite delay `delay`, so that the actual change in the parameter value will take place after a delay. Depending on whether the intended delay is 32-bit integer, 64-bit integer or 64-bit double, `tf_strdelputp()`, `tf_strlongdelputp()` or `tf_strrealdelputp()` is used. Apart from the length of the delay, one can also specify the nature of the delay as shown in the following table.

Table 5. Types of delay

Value of `delay_type`	Nature of delay
0	Inertial delay
1	Modified transport delay
2	Pure transport delay

Let us see what are the implications of using each type of delay.

An *inertial delay* causes all scheduled events on the output parameter in the Verilog model to be removed before scheduling a new event. This ensures that the final value of the modified object at the end of this simulation time slot will be controlled by what specified in this function.

A *modified transport delay* causes all events on the output parameter, that are scheduled for times later than the new event, to be removed before scheduling a new event. Using this delay type overrides the effect of any previously scheduled event on the modified object.

In case of a *pure transport delay*, no scheduled event on the output parameter is removed before scheduling a new event. Care should be taken in using the pure transport delay, as the last event to be scheduled may not be the last one to occur.

5.1.2.3 Example

The following call gets the value of the second parameter and assigns it to the character pointer `par2_val` in binary format.

```
par2_val = tf_strgetp(2, 'b');
```

To obtain the value in octal, decimal or hexadecimal format, the last argument should be replaced by `'o'` (or `'O'`), `'d'` (or `'D'`) or `'h'` (or `'H'`) respectively.

The next example shows how to use the function `tf_strdelputp()` to change the value of an object. Suppose we want to write an arbitrary value of "10x0z1" to the object with PIN=3. The actual value change will take place after a transport delay of 4 time units from the current simulation time. Here is how to do it.

```
tf_strdelputp(3,
              /* 3rd parameter */
              strlen("10x0z1"),
              /* Number of bits in the value */
              'b',
              /* Binary format */
              "10x0z1",
              /* The value as a string */
              4,
              /* Amount of delay */
              1
              /* Mod. transport delay */
              );
```

5.1.3 Reading generic object value

5.1.3.1 Functions in this category

```
struct t_tfexprinfo *tf_exprinfo(pin,val_struct)
int pin;
struct t_tfexprinfo *val_struct;

void tf_evaluatep(pin)
int pin;
```

5.1.3.2 Purpose

These functions are used for reading the value of any read-only or writable object and are among the most powerful utility routines. They encompass the functionalities of all routines described so far for reading the value of an object.

The main function in this category is `tf_exprinfo()`. It takes the PIN of the parameter whose value is to be read. Also, as its second argument, it takes a pointer `val_struct` to structure type `t_tfexprinfo`. This structure is updated with the value of the parameter when the function returns. In addition, the pointer `val_struct` is also returned by the function. The structure type `t_tfexprinfo` is defined in `veriuser.h` (or a variation of it) and is given below.

```
typedef struct t_tfexprinfo
{
   short   expr_type;
   double  real_value;
   char    *expr_string;
   struct  t_vecval *expr_value_p;
   int       expr_ngroups;
   int       expr_vec_size;
   int       expr_sign;
   /* Other unrelated entries not mentioned */
} s_tfexprinfo, *p_tfexprinfo;
```

After return, `expr_type` can have one of the following pre-defined values.

Table 6. Possible values of `expr_type`

When the value is	expr_type is	Detailed description of the parameter

When the value is	expr_type is	Detailed description of the parameter
Non-existent	tf_nullparam	Non-existent (no parameter was passed to the system call or PIN beyond the range)
String	tf_string	String
Real	tf_readonlyreal	Read-only real
	tf_readwritereal	Writable real
Integer	tf_readonly	Read-only integer
	tf_readwrite	Writable integer
Vector	tf_rwbitselect	Bit-select of a vector
	tf_rwpartselect	Part-select of a vector
Memory	tf_rwmemselect	Part of a memory

If the parameter is of string or real type, its value is stored in the `expr_string` or `real_value` field of the structure respectively.

On the other hand, in case of a parameter having integer value (for example, a scalar/vector net or reg), the value is stored in an array `expr_value_p` of structure type `t_vecval`. This structure, once again defined in `veriuser.h`, has the following fields.

```
typedef struct t_vecval
{
   int  avalbits;
   int  bvalbits;
} s_vecval, *p_vecval;
```

Each element `t_vecval` of the array `expr_value_p` contains upto 32 bits of the 4-state value of the parameter. In case of a scalar object, the array will have only one element. For a vectored object, the first 32 bits are stored in the first element, next 32 are stored in the second element, and so on. The 4-state value of a particular bit is given by the combination of the corresponding bits in `avalbits` and `bvalbits` together as shown below.

Table 7. Encoding 4-states using `t_vecval`

Bit value in avalbits	Bit value in bvalbits	Logic value
0	0	0
1	0	1
0	1	z
1	1	x

Notice that, it is the bvalbits part, which, by having a value 0 or 1, determines whether the logic value is numeric (0 or 1) or not (z or x). For

this reason, the bvalbits part is known as the *control bit* whereas the avalbits part is called *normal bit*.

The bit length of the vectored object is stored in `expr_vec_size` and the number of elements in the array `expr_value_p` is given by `expr_ngroups`. Or, in other words, assuming 32-bit implementation of integer, the relation between `expr_vec_size` and `expr_ngroups` is

```
expr_ngroups
= (int)ceil((double)expr_vec_size/32.0)
```

For illustrating the properties of `tf_exprinfo()`, let us consider a user-defined system call `$my_PLI()`, which takes a vectored register as its parameter. A small Verilog program which uses `$my_PLI()` is shown below.

```
module try;
reg [35:0] reg_array; // 36-bit long reg

initial begin
   reg_array = 36'b11101...zx10;
      // all 36 bits are not shown
   $my_PLI(reg_array);
end
endmodule
```

Let us assume that inside the PLI routine, we call `tf_exprinfo()` to gather information about `reg_array`. Since it is 36-bit long, once again assuming a 32-bit implementation of `int` data type, it will take 2 elements of `expr_value_p` to express the value of `reg_array`. The following figure shows the contents of different fields in the `expr_value_p` array for an arbitrary value of `reg_array = 36'b11101...zx10`. In this case, values of `expr_ngroups` and `expr_vec_size` are 2 and 36 respectively. Note that, avalbits and bvalbits for bit positions 36 to 63 in the second integer are zero-filled.

The function `tf_evaluatep()` updates the current value of the specified parameter in a `t_tfexprinfo` structure returned by a previous `tf_exprinfo()` call, saving the user from calling `tf_exprinfo()` repeatedly.

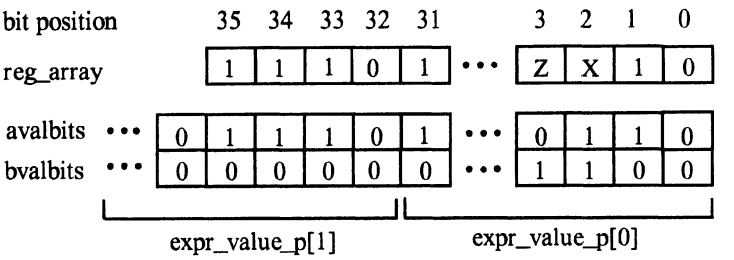

Figure 4. Contents of different fields in `expr_value_p`

5.1.3.3 Example

The following function `read_value()` prints the value of a bit-select, part-select or mem-select parameter, starting from the lower-most bit. Continuing with the example given earlier in the last section, let us assume that the PLI routine corresponding to `$my_PLI()` employs `read_value()`. If the Verilog module `try` calls

```
$my_PLI(reg_array[32:3]);
```

the values of each bit starting from bit position 3 and upto 32 will be printed.

```
void read_value() {
s_tfexprinfo detailed_info;
int i, j;
int ngroups, vecsize, bits_left,
    aval, abit, bval, bbit;

   tf_exprinfo(1, &detailed_info);
   /* reading the info about the parameter */
   if ((detailed_info.expr_type==tf_readonly) ||
       (detailed_info.expr_type==tf_readwrite) ||
       (detailed_info.expr_type==tf_rwbitselect) ||
       (detailed_info.expr_type==tf_rwpartselect)||
       (detailed_info.expr_type==tf_rwmemselect)) {
      /* check the data type of the parameter */

      ngroups = detailed_info.expr_ngroups;
         /* No. of integer used */
      vecsize = detailed_info.expr_vec_size;
         /* No. of parameters in the group */
      for (i=0;i<ngroups;i++) {
```

```
            aval = detailed_info.expr_value_p[i].avalbits;
            bval = detailed_info.expr_value_p[i].bvalbits;

            if (vecsize >= 32) {
               bits_left=32;
               vecsize -= 32;
            } else
               bits_left=vecsize;

            for (j=1; j<=bits_left; j++) {
               abit=aval & 0x1;
               bbit=bval & 0x1;
               /* Masking only the LSB */
               aval >>= 1;
               bval >>= 1;
               /* Shifting right to get the next bit */
               io_printf("Bit %d = ",i*32+j);
               if ((abit==0) && (bbit==0))
                  io_printf("0\n");
               else if ((abit==1) && (bbit==0))
                  io_printf("1\n");
               else if ((abit==0) && (bbit==1))
                  io_printf("z\n");
               else if ((abit==1) && (bbit==1))
                  io_printf("x\n");
            }
         }
      } else
         tf_error("Invalid parameter type...\n");
   }
```

5.1.4 Getting node information

5.1.4.1 Functions in this category

```
struct t_tfnodeinfo *tf_nodeinfo(pin, node_struct)
int pin;
struct t_tfnodeinfo *node_struct;

void tf_propagatep(pin)
int pin;
```

5.1.4.2 Purpose

The function `tf_nodeinfo()` performs a job very similar to that of `tf_exprinfo()`. But when `tf_exprinfo()` is appropriate for obtaining general information about parameters, `tf_nodeinfo()` provides information regarding the *nodes* of a design. A node in a design can be any of the following: register, integer, real, time, vectored and scalar nets and memory. Furthermore, as will be explained shortly, `tf_nodeinfo()` returns a data structure which is more concise compared to `tf_exprinfo()`.

`tf_nodeinfo()` takes the PIN corresponding to the node as the first argument, and stores the information in a structure of type `t_tfnodeinfo` as specified by its second argument. The function also returns the same structure.

The definition of the structure type `t_tfnodeinfo` is shown below.

```
typedef  struct  t_tfnodeinfo {
   short node_type;
   union {
      struct t_vecval *vecval_p;
      struct t_strengthval *strengthval_p;
      char *memoryval_p;
      double *real_val_p;
   } node_value;
   int node_ngroups;
   int node_vec_size;
   int node_mem_size;
   int node_sign;
   char *node_symbol;
/* Other unrelated entries not mentioned */
} s_tfnodeinfo, *p_tfnodeinfo;
```

After the function returns, `node_type` may have one of the pre-defined values, as shown in the first two columns of the following table.

Table 7A. Pre-defined values of `node_type` and corresponding field of `node_value`

When the parameter is	The value of node_type is	The used field of node_value is
Not a node	tf_null_node	None
Register	tf_reg_node	vecval_p
Integer	tf_integer_node	vecval_p
Real	tf_real_node	real_value_p
Time	tf_time_node	vecval_p
Vectored net	tf_netvector_node	vecval_p

When the parameter is	The value of node_type is	The used field of node_value is
Scalar net	tf_netscalar_node	strengthval_p
Memory	tf_memory_node	memoryval_p

Unlike `t_tfexprinfo`, where each data type (string, real etc.) has their own field, `t_tfnodeinfo` has a union `node_value` for all the value types. Since for any node only one type will be useful, this saves the total amount of allocated memory for the structure.

As shown in the third column of the Table 7A above, if the `node_type` is `tf_reg_node`, `tf_integer_node`, `tf_time_node` or `tf_netvector_node`, the `node_value` is an array of structures of type `t_vecval`. (The fields of this structure and bit-encoding are similar to what we discussed in Section 5.1.3.2 except that all occurrences of `expr_ngroups` and `expr_vec_size` fields should be replaced by their corresponding counterparts `node_ngroups` and `node_vec_size` respectively for the present case). The value of such a node can be extracted in a manner similar to what has been shown for `tf_exprinfo()`.

Similarly, if the `node_type` is `tf_real_node`, the pointer `real_val_p` can be used to obtain the value.

The function can also be used to obtain charge strength information of a scalar net. If the `node_type` is `tf_netscalar_node`, the `strengthval_p` field of the `node_value` points to an array of structure of type `t_strengthval` as defined below.

```
typedef  struct t_strengthval
{
   int strength0;
   int strength1;
}
```

The fields `strength0` and `strength1` get the 0-strength and 1-strength bit patterns of the value respectively.

If the parameter is of type `tf_memory_node`, the value is represented as an array of characters. As in the case of registers, here also the value consists of separate avalbit and bvalbit for each bit (See Table 7). However, for memory parameters their arrangement is different compared to that of a register.

For memory parameters, the avalbits of an entire row of the memory comes first, followed by the bvalbits. The field `node_ngroups` indicates how many characters are required for each of avalbit and bvalbit. Also, the width of the memory (i.e. the number of bits in each of its row) is given by

`node_vec_size`. Or in other words, assuming an 8-bit implementation of `char` data type, the relation between `node_ngroups` and `node_vec_size` is given by

```
node_ngroups = (int)ceil((double)node_vec_size/8.0);
```

As a result, the avalbits and bvalbits of the address `addr` of the memory start at location (`node_ngroups*2*addr`) and (`node_ngroups*2*addr + node_ngroups`) respectively.

When the value is unsigned, `node_sign` is 0. `node_symbol` indicates the identifier name of the parameter.

The function `tf_propagatep()` propagates the value change of an object accessed earlier by `tf_nodeinfo()` (for memory objects) or `tf_exprinfo()` (for all other objects) to any continuous assignment.

5.1.4.3 Example

The following example shows how to read a particular row of a memory variable. Let us assume that the memory has been declared in the Verilog program as shown below.

```
reg [18:0] my_mem [1023:0];
```

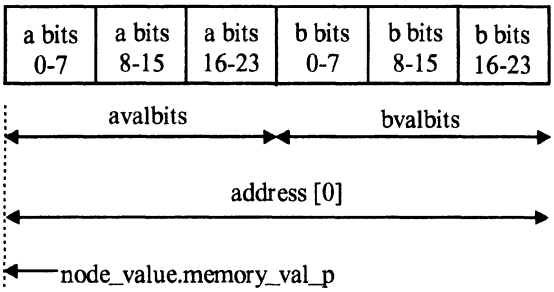

Figure 5. The arrangement of a and b parts for address 0 of mem1. Other addresses will follow successively.

Since, there are 19 bits in each row of `my_mem`, assuming `char` data type is implemented as 8-bit, it will require 3 characters to store each of avalbits and bvalbits of any row. In this case, the bit positions 19 to 23 of each row of `my_mem`, which are represented by the 3rd pair of avalbits and bvalbits will not contain any meaningful data. The arrangement of the characters for

`mem1[0]` is shown below. The data for other indices will be on successive locations.

The following example shows a function `get_value()` which returns the avalbits and bvalbits of the `n`th row of `my_mem`.

```
void get_value(pin, nrow, a_part, b_part)
int pin, nrow;
char *a_part, *b_part;
{
   s_tfnodeinfo my_mem_info;
   char *my_mem_start;
   int ngroups;

   tf_nodeinfo(pin, my_mem_info);
   my_mem_start=my_mem_info.node_value.memoryval_p;
   ngroups=my_mem_info.node_ngroups;

   a_part=ngroups*2*nrow;
   b_part=ngroups*2*nrow+ngroups;
}
```

The information about the memory passed as a parameter to the system call is obtained by calling the `tf_nodeinfo()` with proper PIN. The starting address of the value is given by `my_mem_start`.

The reader is encouraged to write an accompanying function which will take these 2 parts and print the value of the row (refer to the example of Section 5.1.3.3 for hint).

Also, note that an entire memory data type can not be passed as a single parameter in a system call. For this reason only the first (or any row) of the memory variable should be passed. For example, if we assume that the above function `get_value()` is used by a system call `$get_value()`, from the Verilog program this system call will be invoked as follows:

```
$get_value(my_mem[0], nrow);
```

5.2 Value change detection

5.2.1 Functions in this category

```
void tf_asynchon()
```

```
void tf_asynchoff()

int tf_copypvc_flag(pin)
int pin;

int tf_movepvc_flag(pin)
int pin;

int tf_testpvc_flag(pin)
int pin;

int tf_getpchange(pin)
int pin;
```

5.2.2 Purpose

This set of functions is used to detect the occurrence of a parameter value change.

Once `tf_asynchon()` is called, whenever any parameter or its driver changes value, misctf function is called with a reason of `reason_paramvc` or `reason_paramdrc` respectively (See Chapter 2 Section 4.2.5 for details). This continues until `tf_asynchoff()` is called to disable the callback.

Parameter Value Change or pvc flags are indicators associated with each parameter passed to the system call showing whether the parameter has changed its value. The flags are active only after `tf_asynchon()` is called. There are two such flags associated with each parameter - the first one is the *current pvc flag*, which the Verilog simulator updates during the simulation. The other flag is the *saved pvc flag*, which is controlled by the user. The value of the current pvc flag is 0, if the corresponding parameter value is unchanged; otherwise it is non-zero.

The following functions are used for handling the pvc flags. `tf_copypvc_flag()` copies the current pvc flag of a parameter to its saved pvc flag. `tf_movepvc_flag()` moves its current pvc flag to the saved pvc flag and clears the current pvc flag for the current instance of the system call. These two functions return the final value of the saved pvc flag. `tf_testpvc_flag()` returns the current value of the saved pvc flag. All of them takes a parameter index number as their argument. However, if the passed parameter is -1, all parameter pvc flags are copied/moved/tested, and the logical OR of all saved flags are returned to check if any of the PVC flag is set.

`tf_getpchange()` takes a parameter index number and searches for any other changed parameter with a parameter index number greater than this. If such a parameter exists, its parameter index number is returned; otherwise if an error occurs, it returns 0. This function should be called after calling `tf_movepvc_flag(-1)` as it tests the saved pvc flag.

5.2.3 Example

The following function takes different actions depending on which parameter has changed its value. If the parameter 1 changes, it simply prints the new value. If the parameter 2 changes its value, the bitwise ANDed value of the 1st and the 2nd parameters is printed.

```
#include <stdio.h>
#include "veriuser.h"

int myCalltf(), myMisctf();

/* calltf routine */
int myCalltf(){
    tf_asynchon();
}

/* misctf */
int myMisctf(data, reason);
int data, reason;
{
 int i, j;

 if (reason == reason_paramvc) {
    tf_movepvc_flag(-1);
    for (i=0; i<tf_nump();) {
       j=tf_getpchange(i);
       switch(j) {
          case 0 : /* Error or no change-quit*/
             tf_error("No parameter has changed value");
             exit;
          case 1 : /* 1st parameter has a new value */
             io_printf(" 1st param:%d\n", tf_getp(1));
             break;
```

```
            case 2 : /* 2nd parameter has a new value */
               io_printf(" 1st param AND 2nd param:%d\n",
                  tf_getp(1) & tf_getp(2));
               break;
         } /* switch-j */
      } /* for-i */
   } /* if reason == reason_paramvc*/

   } /* myMisctf */
```

5.3 Simulation time

5.3.1 Getting simulation time

5.3.1.1 Functions in this category

```
int tf_gettime()

int tf_getlongtime(time_hi32)
int time_hi32;

double tf_getrealtime()

char *tf_strgettime()
```

5.3.1.2 Purpose

These functions, as their names suggest, get the current simulation time in different formats.

With the exception of `tf_getlongtime()`, which takes one integer argument – a place-holder for the higher 32 bits of time, all other functions are called without parameter. `tf_getrealtime()` and `tf_strgettime()` return the time as a double precision number and character pointer respectively. `tf_gettime()` and `tf_getlongtime()` both return 32-bit integers. Since `time` is a 64-bit data, `tf_gettime()` returns only the lower most 32 bits and should be used only when simulation time is known to fit within 32 bits.

For getting the entire 64 bits, the function `tf_getlongtime()` should be used. The function stores the most significant 32 bits of the time in the parameter that it takes and returns the lower 32 bits as another integer.

5.3.1.3 Example

The following code segment gets the upper and lower 32 bits of the current simulation real time.

```
int upper_32b_time, lower_32b_time;

lower_32b_time = tf_getlongtime(upper_32b_time);
```

5.3.2 Timescale and precision of a module

5.3.2.1 Functions in this category

```
int tf_gettimeunit()

int tf_gettimeprecision()
```

5.3.2.2 Purpose

These two functions return the simulation time unit and the timescale precision respectively. Unless defined by the `` `timescale `` compiler directive in the Verilog program, the simulator imposed default values are returned. Both of them return an integer, ranging from 2 to -15. The actual value of the time unit or the timescale precision is given by $10^{(return_value)}$ seconds. For example, if the return value of `tf_gettimeunit()` is 2, the simulation time unit is 100 seconds.

5.3.2.3 Example

The following example prints the current timescale precision.

```
#include <math.h>

io_printf("Current precision: %f seconds\n",
        pow(10.0,(double)tf_gettimeprecision()));
```

5.3.3 Conversion between scaled delay and internal simulation time

These functions will be discussed in Section 6.

5.3.4 Simulation synchronization

5.3.4.1 Functions in this category

```
void tf_synchronize()

void tf_rosynchronize()

int tf_getnextlongtime(time_lo32, time_hi32)
int *time_lo32, *time_hi32;

int tf_setdelay(delay)
int delay;

int tf_setlongdelay(delay_lo32, delay_hi32)
int delay_lo32, delay_hi32;

int tf_setrealdelay(delay)
double delay;

void tf_clearalldelays()
```

5.3.4.2 Purpose

This category of functions provide the user two different means to control the event queue in a simulation. The first set provides user control within the same simulation time slot, whereas the next gives the flexibilty to schedule an event in the future time slot. Both of them work through callback mechanism.

Both `tf_synchronize()` and `tf_rosynchronize()` initiate a callback mechanism, such that the misctf function of the same PLI routine is executed after all the scheduled events in that simulation time slot complete their execution. The function `tf_rosynchronize()` further guarantees that no event from the called misctf function can be scheduled at a future simulation time, making calls like `tf_strdelputp()` illegal. The reasons associated with the callbacks for these functions are `reason_synch` and `reason_rosynch` respectively (Refer to Chapter 2, Section 4.2.4). None of these functions takes argument.

`tf_getnextlongtime()` is a function which can be used in association with `tf_rosynchronize()`. It takes two 32-bit integers as its arguments, which are used as placeholders for the 64-bit simulation time of the next scheduled event according to the following rule. If it is called from a misctf

function with a reason `reason_rosynch`, it returns 0 and gets the simulation time of the next scheduled event. If there is no future event scheduled, the function returns 1 and the placeholder arguments contain 0s. If `tf_getnextlongtime()` is called from a misctf function, but with a reason other than `reason_rosynch`, it returns 2 and present simulation time is assigned to its two arguments.

The functions `tf_setdelay()`, `tf_setlongdelay()` and `tf_setrealdelay()` can schedule a callback at a future point of time, as determined by their integer/long (i.e. two integers)/double argument(s). The reason associated with such a callback is `reason_reactivate` (See Chapter 2 Section 4.2.1 for details). All these functions return 0, if an error occurs and 1 otherwise. Note that, misctf function will be called multiple times at a simulation time, if there are multiple callbacks scheduled at this time by these functions.

The function `tf_clearalldelays()`, on the other hand, deletes all the events scheduled by `tf_setdelay()` in the event queue for future simulation time . This function is particularly suitable for use with `reason_disable`.

5.3.4.3 Example

The following example assigns a value of 1 to the first parameter of the system call. It schedules this event 10 time units from the current time. It also ensures the event happens only at the end of the time slot.

```
/* calltf routine */
int myCalltf() {
   tf_setdelay(10);
   /* Scheduling at (present time + 10) */
}

/* misctf routine */
int myMisctf(data, reason)
int data;
int reason;
{
 int i, j;

 switch(reason) {
    /* tf_setdelay() has the following reason*/
    case reason_reactivate: {
       tf_synchronize();
```

```
            break;
         }
         /* tf_synchronize() has the
            following reason */
         case reason_synch: {
            /* Put the event description here*/
            tf_putp(1,1);
            break;
         }
      }
   }
```

5.4 Displaying messages

5.4.1 Functions in this category

```
void io_printf(format, arg1, ..., arg12)

void io_mcdprintf(mcd,format, arg1, ..., arg12)

void tf_warning(format, arg1, ..., arg5)

void tf_error(format, arg1, ..., arg5)

void tf_text(message, arg1, ..., arg5)

void tf_message(level, facility, code, message,
                arg1, ..., arg5)
```

5.4.2 Purpose

These functions are similar to `printf()` and other output functions of C as defined in `stdio.h`. All of them, except `io_mcdprintf()` and `tf_text()`, write the passed arguments to both the standard output and the Verilog log file. `io_mcdprintf` writes to the opened files (multi-channel descriptor `mcd`) whereas `tf_text()` keeps the arguments into a buffer `message` to be retrieved by `tf_message()` later. Also, maximum number of arguments is restricted to 12 in case of `io_printf()` and `io_mcdprintf()` and 5 for the rest.

Among these functions, `tf_message()` is the most generalized one and can be used for various levels and purposes of messaging. The parameters which are unique to this function, are shown in the following table.

Table 7B. Parameter definition in `tf_message()`

Parameter name	Purpose	Possible values
level	Severity of the message	One of five pre-defined levels: ERR_ERROR ERR_SYSTEM ERR_INTERNAL ERR_MESSAGE ERR_WARNING
facility	The software package	Any user-defined string of less than 10 characters
code	Part of the code concerning the message	Any user-defined string of less than 10 characters

5.4.3 Example

The following example shows how to use `tf_message()` to generate a warning (essentially with the same effect as that of `tf_warning()`) with a facility of `my_verilog` and `TFARGMISS` software product code (since facility and code are user-defined strings to help in debugging, their proper and meaningful use solely lies with the user).

```
tf_message(ERR_WARNING, "my_verilog", "TFARGMISS",
"Null argument passed for the %dth argument",
arg_number);
```

5.5 Instance pointer, module and scope instance name

5.5.1 Functions in this category

```
char *tf_getinstance()

char *tf_mipname()

char *tf_spname()
```

5.5.2 Purpose

In the absence of any handle mechanism, the functions in this category provide a convenient way to find out a reference to the current system call. When `tf_getinstance()` returns an instance pointer (as a `char *`) to the current system call, `tf_mipname()` and `tf_spname()` returns a string containing the path name of the parent module and generic scope of the current instance of the system call respectively. The function `tf_getinstance()` is normally used in association with the utility routines using pointers. We will learn more about this in the next section.

5.5.3 Example

The `io_printf()` statement below prints the module and scope path names of the current instance.

```
io_printf("Module and scope pathnames are %s\n",
            tf_mipname(), tf_spname());
```

5.6 Saving information between calls

5.6.1 Functions in this category

```
char *tf_getworkarea()

void tf_setworkarea(workarea)
char *workarea;
```

5.6.2 Purpose

Workarea is a scratch-pad space associated with any user-defined system call. It can be used for temporary storage of useful data. The function `tf_setworkarea()` is used to save a pointer to user data in the workarea. This pointer can be retrieved either upon a subsequent invocation of the PLI routine or within the same call but in a different function (such as calltf and misctf) using `tf_getworkarea()`. Since workarea pointer is defined as `char *`, the pointer to the user data also should be cast to the same.

5.6.3 Example

The following example stores a file pointer `myFile` in the workarea in checktf and retrieves it as `lastFile` in calltf.

```
int myChecktf() {
FILE *myFile;
:
tf_setworkarea((char *)myFile);
   /* store file pointer */
:
}

int myCalltf() {
FILE *lastFile;
:
lastFile = (FILE *)tf_getworkarea();
   /* get file pointer opened in myChecktf */
}
```

5.7 Miscellaneous parameter related functions

5.7.1 Functions in this category

```
int tf_nump()

int tf_typep(pin)
int pin;

int tf_sizep(pin)
int pin;
```

5.7.2 Purpose

These functions provide general information about the parameters passed to the user-defined system call.

While `tf_nump()` gets the total number of parameters, `tf_typep()` returns the type (one of `tf_nullparam`, `tf_string`, `tf_readonly`, `tf_readwrite`, `tf_readonlyreal` and `tf_readwritereal`) of a specified parameter and `tf_sizep()` returns the bit length of a parameter.

It should be noted that there is a difference between a system call with no parameter and another which takes a null parameter as the following example illustrates.

```
$my_call();  // takes a null parameter
$your_call;  // takes no parameter
$yet_another_call(,); // takes two null parameters
```

It is interesting to note that for the user-defined system call `$my_call()`, `tf_nump()` returns 1 and `tf_typep(1)` returns `tf_nullparam`, indicating that the call takes a null parameter, whereas for `$your_call`, `tf_nump()` returns 0.

5.7.3 Example

Perhaps the most frequent use of these functions occurs in the checktf function of a PLI routine. For example, if a system call `$myPLI` expects an 8-bit wide net as its parameter, the following code fragment checks the validity of the passed parameter.

```
if ((tf_nump() != 1) ||
    (tf_typep(1) != tf_readonly) ||
    (tf_sizep(1) != 8))
   tf_error("$myPLI : Invalid argument\n");
```

5.8 Stopping and finishing the simulation

5.8.1 Functions in this category

```
void tf_dostop()

void tf_dofinish()
```

5.8.2 Purpose

These two functions are used for halting the simulation.

`tf_dostop()` temporarily stops the simulation and takes the simulator to the interactive mode. This is equivalent to the `$stop` built-in system call in Verilog.

`tf_dofinish()` aborts the simulation. Its behavior is same as that of `$finish` built-in system call in Verilog.

None of these functions require any parameters.

5.8.3 Example

The following example shows part of a PLI routine, which goes to the interactive mode, if total number of parameters passed to the system call is different from what is expected. However, it aborts the simulation, if any of the parameters is found to be null.

```
if (tf_nump() != EXPECTED_NUMP)
   tf_dostop();
   /* Goes to the interactive mode */
else
   for (i=1; i<= tf_nump(); i++)
      if (tf_typep(i) == tf_nullparam)
         tf_dofinish();
         /* Aborts the simulation */
```

5.9 Getting plus arguments from the environment

5.9.1 Function in this category

```
char *mc_scan_plusargs(start_arg)
char *start_arg;
```

5.9.2 Purpose

This function returns the remaining part of a command line *plus argument*, if it exists, which starts with the string passed as the argument `start_arg` to the function. If there is no such plusarg, it returns a `NULL` string. If there are multiple plusargs matching the search, only the first plusarg is returned.

5.9.3 Example

If the compiled binary Verilog executable `myVerilog` is invoked as shown below:

```
myVerilog  test.v  +useCycle
```

inside the PLI code, the call

```
mc_scan_plusargs("use")
```

will return "Cycle".

5.10 Arithmetic and conversion functions

5.10.1 Functions in this category

```
int tf_compare_long(op1_lo32, op1_hi32,
                    op2_lo32, op2_hi32)
int op1_lo32, op1_hi32, op2_lo32, op2_hi32;

void tf_add_long(final_lo32, final_hi32,
                 op1_lo32, op1_hi32)
int *final_lo32, *final_hi32;
int op1_lo32, op1_hi32;

void tf_subtract_long(final_lo32, final_hi32,
                      op1_lo32, op1_hi32)
int *final_lo32, *final_hi32;
int op1_lo32, op1_hi32;

void tf_multiply_long(final_lo32, final_hi32,
                      op1_lo32, op1_hi32)
int *final_lo32, *final_hi32;
int op1_lo32, op1_hi32;

void tf_divide_long(final_lo32, final_hi32,
                    op1_lo32, op1_hi32)
int *final_lo32, *final_hi32;
int op1_lo32, op1_hi32;
```

5.10.2 Purpose

These functions are used to perform arithmetic operations on long (64-bit) integers.

`tf_compare_long()` takes four 32-bit integers (lower and upper 32 bits of the two operands) as arguments. It returns -1, 0 and 1 depending on whether the first operand is less than, equal to or greater than the second operand respectively.

For the rest of the functions, the lower and upper 32 bits of the first operand are passed as integer pointer and the other operand as two integers. Operands are expected to be in 2's complement form and the result replaces the first operand (i.e. first two arguments).

5.10.3 Example

The following example shows how to use `tf_add_long()`. Before scheduling a callback event at `nticks` (assumed to be an integer) time units from the current simulation time, it prints a warning showing if the expected time of the scheduled event has crossed a certain time limit.

```
int curr_time_low, curr_time_high;

curr_time_low = tf_getlongtime(&curr_time_high);
tf_add_long(&curr_time_low, &current_time_high,
            nticks, 0);
if (tf_compare_long(curr_time_low, curr_time_high,
                    time_limit_low, time_limit_high)>0)
   tf_warning("New event was not scheduled");
else
   tf_setdelay(nticks);
```

5.11 Conversion between data types

5.11.1 Functions in this category

```
void tf_long_to_real(val_lo32, val_hi32, val_real)
int val_lo32, val_hi32;
double *val_real;

char *tf_longtime_tostr(val_lo32, val_hi32)
int val_lo32, val_hi32;

void tf_real_to_long(val_real,
```

```
                        val_lo32, val_hi32)
double val_real;
int * val_lo32, *val_hi32;
```

5.11.2 Purpose

These functions converts the value of the argument from one format to another.

5.11.3 Example

The following code segment prints the current simulation time in string format.

```
int current_time_low, current_time_high;

current_time_low =
          tf_getlongtime(&current_time_high);
io_printf("Current time is %s\n",
          tf_longtime_tostr(current_time_low,
                            current_time_high));
```

5.12 Save and restore operation

5.12.1 Functions in this category

```
int tf_write_save(block_ptr, block_len)
char *block_ptr;
int block_len;

int tf_read_restart(block_ptr, block_len)
char *block_ptr;
int block_len;
```

5.12.2 Purpose

The functions `tf_write_save()` and `tf_read_restart()` are used in conjunction with the reasons (see Section 4.2 in Chapter 2) `reason_save` and `reason_restart` respectively in misctf.

`tf_write_save()` appends a block of user-defined data to a *save file*. The start address of the data block (as a character pointer) and the block length are passed as parameters.

`tf_read_restart()` retrieves back the user data, written earlier by `tf_write_save()`, from the save file. It takes the same two parameters that `tf_write_save()` takes. Care should be taken to ensure that only as many bytes are being read as was saved earlier by `tf_write_save()`. Also the data must be casted to character pointer type before saving.

5.12.3 Example

These two routines are mainly used for initializing a variable for further processing after a `$restart` call. Common examples of such situations are: keeping track of toggle count of nets in a circuit, how many times a particular register has been updated since the beginning of the simulation etc. The example below shows a typical case where the ratio between two clocks and the frequency of one of them is stored in a save file. It can be retrieved by a `$restart` call later and used subsequently. A switch instruction inside the misctf function handles the reasons.

```
int myMisctf(data, reason)
int data, reason;
{
int frequency;
float ratio;
:
switch (reason) {
   case reason_save:
      /* frequency and ratio are computed first:
      frequency = ...;
      ratio = ... ;
      */
      tf_write_save((char *)&frequency,
                    sizeof(int));
      tf_write_save((char *)&ratio,
                    sizeof(float));
      break;
   case reason_restart:
      tf_read_restart((char *)&frequency,
                      sizeof(int));
```

```
        tf_read_restart((char *)&ratio,
                        sizeof(float));
        break;
    }
}
```

6. USING INSTANCE POINTERS IN UTILITY ROUTINES

6.1 Using instance pointer

Once a system call is written, it can be used in more than one place in the user Verilog program. Also there may be multiple system calls in the same program. Each instance of a system call is identified by an *instance pointer*. As described in Section 5.5, the instance pointer corresponding to the current instance of the system call can be obtained as a character pointer by using `tf_getinstance()` function. Note that, an instance pointer is only a reference to the current instance of the system call. It is *not* a path name or module instance name. Although it is a character pointer, it should *never* be treated as a string, e.g. using with functions like `strcmp()` etc.

Many of the utility routines have two forms. The first one is meant for the current instance of the system call. All the functions described in Section 5 are of this nature. In the other form, the main functionality of the routine remains the same. However, this time the function takes an additional parameter – an instance pointer – at the end of normal parameter list. In this case, the function affects the particular instance of the system call given by the instance pointer, rather than the current instance. All such functions have a prefix `tf_i` in their names.

As an example, in the following case, using the instance pointer of one instance of a system call, we can change the value of the second parameter of that instance without leaving the current instance.

```
tf_iputp(2, inst_ptr);
   /* instance pointer to another system call */
```

The following call is equivalent of `tf_getp(3)`.

```
tf_igetp(3,tf_getinstance());
```

6.2 Functions for handling instance pointer

The following table shows the functions which use the instance pointer. As mentioned, their functionalities are same as their normal counterpart, but they work on the instance pointed by the instance pointer, rather than the current instance.

Table 7C. Functions which use instance pointer

tf_iasynchoff()	tf_iasynchon()	tf_iclearalldelays()	tf_icopypvc_flag()
tf_ievaluatep()	tf_iexprinfo()	tf_igetcstringp()	tf_igetlongp()
tf_igetlongtime()	tf_igetp()	tf_igetpchange()	tf_igetrealp()
tf_igetrealtime()	tf_igettime()	tf_igettimeprecision()	tf_igettimeunit()
tf_igetworkarea()	tf_imipname()	tf_imovepvc_flag()	tf_inodeinfo()
tf_inump()	tf_ipropagatep()	tf_iputlongp()	tf_iputp()
tf_iputrealp()	tf_irosynchronize()	tf_isetdelay()	tf_isetlongdelay()
tf_isetrealdelay()	tf_isetworkarea()	tf_isizep()	tf_ispname()
tf_istrdelputp()	tf_istrgetp()	tf_istrlongdelputp()	tf_istrrealdelputp()
tf_isynchronize()	tf_itestpvc_flag()	tf_itypep()	

6.3 Functions which only work with an instance pointer

The following functions work only with an instance pointer. They do not have a counterpart to work on the current instance. Furthermore, these functions do not start with a prefix `tf_i`.

6.3.1 Functions in this category

```
void tf_scale_longdelay(inst_ptr,
                        delay_lo32, delay_hi32,
                        result_lo32, result_hi32)
char *inst_ptr;
int delay_lo32, delay_hi32;
int *result_lo32, *result_hi32;

void tf_scale_realdelay(inst_ptr, real_delay,
                        result)
char *inst_ptr;
double real_delay;
double *result;

void tf_unscale_longdelay(inst_ptr,
```

```
                        delay_lo32, delay_hi32,
                        result_lo32, result_hi32)
char *inst_ptr;
int delay_lo32, delay_hi32;
int *result_lo32, *result_hi32;

void tf_unscale_realdelay(inst_ptr, real_delay,
                        result)
char *inst_ptr;
double real_delay;
double *result;
```

6.3.2 Purpose

The 'scale' functions of this category convert a long integer/real delay into the timescale of the module, containing the system call pointed to by `inst_ptr`. The 'unscale' functions do just the opposite – they convert a delay expressed in internal simulation time units to the timescale of a particular module, as given by the instance pointer. The result is stored in the place holder(s) passed as pointer. As is apparent, a form of these functions which work on the current module will not be necessary.

7. LIMITATIONS OF UTILITY ROUTINES

The most obvious advantage of using utility routines is their simplicity. If a program uses utility routines, the code remains concise and easy to read. This also increases the overall maintainability of the code. Most often the operation to be performed can be achieved using only one call of a specific function, resulting in smaller program sizes.

The other big plus point of the utility routines is the way it handles the callback mechanism. This provides precise control over the schedule of events for almost all desired situations.

On the flip side, utility routines work only on the objects which are directly passed to the system call as parameters. As only a few limited type of objects can be passed as parameters to a system call (for example, a particular instance of a sequential UDP can not be passed as parameter), this puts a serious limitation on the capability of the utility routines.

There is no mechanism for indirect addressing of the objects (such as, handle provided by the access/vpi routines). So it is not possible to pass

objects as a parameter in a function call. Furthermore, this limitation prevents us from doing such things as finding out any relationship between two design objects (e.g., parent and child or driver and load etc.) or traversing all the nets within a particular module in a design hierarchy. For this reasons, more complex programs, which often require such features, can not depend on utility routines alone.

8. SUMMARY

In this chapter, we have discussed the utility routines, the first generation of the PLI library functions. The main classifications of the routines are described with their main purpose and suitable examples. Also, we found out the limitations of such routines.

In the next chapter, we are going to see how these limitations are overcome in access routines, the second generation of PLI library functions.

Chapter 4

ACCESS ROUTINES IN PLI1.0

As mentioned earlier, PLI1.0 provides two different sets of library functions. We have discussed the first of them, the utility routines, in Chapter 3. In this chapter, we will take up the access routines. While utility routines limit themselves in reading/modifying the parameters of the user-defined task or function, access routines expand these capabilities to a wide range of design objects. Access routines also introduce a new powerful data type called handle. We will start with the basics of access routines. Later, different classifications of these routines will be discussed.

1. WHAT ARE ACCESS ROUTINES ?

Access routines, as the name suggests, are used to access the properties of a wide range of simulation objects. The following table shows a list of read-only and read-write objects (see Section 3 of Chapter 3 for definitions of read-only and read-write objects) for access routines as recommended by the IEEE Std. 1364-1995.

Table 8. Read-only and read-write objects for access routines

Read-only objects	Read-write objects
Module instances	Intermodule path delays
Module ports	Module path delays
Module paths	Module input port delays
Intermodule paths	Primitive instance delays
Top-level modules	Timing check limits
Primitive instances	Register logic values
Primitive terminals	Sequential UDP logic values

S. Mittra, *Principles of Verilog PLI*
© Kluwer Academic Publishers 1999

Read-only objects	Read-write objects
Nets	
Registers	
Parameters	
Specparams	
Timing checks	
Named events	
Integer, real and time variables	

Access and utility routines can be used in the same user function in a PLI routine and the compilation and other methodologies remain the same.

All access routines start with the prefix `acc_` and, for this reason, they are also known as ACC or acc routines.

2. USING ACCESS ROUTINES IN A USER FUNCTION

In order to use access routines in a PLI application, certain objects should be setup before and after the main body of the program. The following code fragment shows a user function `my_func()`, which calls access routines.

```
#include "acc_user.h"
void my_func(){
    acc_initialize();
    /* The main body of the function goes here */
    acc_close();
}
```

As the above example shows:

- The header file `acc_user.h` must be included at the beginning of the program. All data-structures related to access routines are declared and defined in this file.
- The function `acc_initialize()` initializes certain variables and sets up the environment to be used later by other acc routines.
- The main body of the code contains the user-defined application which uses one or more access routines.
- The function `acc_close()` is called in the end. Its task is to undo the actions taken by the function `acc_initialize()`, such as freeing up memory allocated to the variables etc.

3. HANDLE

Handle is a predefined data type whose functionality is similar to that of a pointer in C. Handles can be used to point to an object in the design database. However, unlike pointers in C, where each data type has its corresponding pointer type, a handle can be used to refer to *any* kind of object in the design database.

A handle is declared as shown below.

```
handle my_handle;
```

Once declared, a handle can be passed between functions in the user routine just like any other kind of variable.

Handles are the backbone of the access routine methodology and probably the most important new concept introduced in this part of PLI1.0. Handles ensure that almost all design objects can be treated as ordinary variables, keeping the entire data-structure transparent to the user.

4. CLASSIFICATIONS OF ACCESS ROUTINES

Quite often a typical user application, which uses one or more access routine, looks as follows.

start *:*

1. get a handle to a given design object;

2. fetch a property of the object;

3. modify the property, if required;

4. also, if required, monitor the object for any value change and take action depending on that;

5. do some housekeeping jobs, such as printing messages etc.;

6. move to the next object;

goto ***start*** *until all required objects are over;*

Depending on the six different tasks listed above, all access routines can be divided into six categories.

Handle routines return a handle to a specified object. Normally, certain properties of the object (e.g. its full name etc.) are used as input arguments to these routines to get the handle to the object. All handle routines start with

the prefix *acc_handle_*. The following table shows a set of functions and their uses.

Table 9. How to use handle routines

Purpose	Functions
Obtaining handle to a scope or object	acc_handle_scope()
	acc_handle_by_name()
	acc_handle_object()
	acc_handle_interactive_scope()
Obtaining handle for the calling module of the system call or parent module of an object	acc_handle_calling_mod_m()
	acc_handle_parent()
Obtaining handle to port and related objects	acc_handle_port()
	acc_handle_hiconn()
	acc_handle_loconn()
	acc_handle_path()
Obtaining handle to module path, data path and related objects	acc_handle_modpath()
	acc_handle_datapath()
	acc_handle_pathin()
	acc_handle_pathout()
	acc_handle_condition()
Obtaining handle to timing check and related objects	acc_handle_tchk()
	acc_handle_notifier()
	acc_handle_tchkarg1()
	acc_handle_tchkarg2()
Obtaining handle to simulated net, terminal and related objects	acc_handle_simulated_net()
	acc_handle_terminal()
	acc_handle_conn()
Obtaining handle to a system call and its parameter	acc_handle_tfinst()
	acc_handle_tfarg()
	acc_handle_itfarg()

Fetch routines do just the opposite of what handle routines achieve – using a handle to an object, they retrieve properties associated with that object. All fetch routines start with the common prefix *acc_fetch_*. The following table lists the functions against their common functionality.

Table 10. How to use fetch functions

To get this	Use one of these functions
Command line arguments	acc_fetch_argc()
	acc_fetch_argv()
Attribute parameter and specparam	acc_fetch_attribute()
	acc_fetch_attribute_int()
	acc_fetch_attribute_str()
	acc_fetch_paramtype()
	acc_fetch_paramval()
Object name, type and location	acc_fetch_defname()
	acc_fetch_fullname()

To get this	Use one of these functions
	acc_fetch_name()
Delay	acc_fetch_delay_mode()
	acc_fetch_delay()
Vector size and range	acc_fetch_size()
	acc_fetch_range()
System call argument	acc_fetch_tfarg()
	acc_fetch_itfarg()
	acc_fetch_tfarg_int()
	acc_fetch_itfarg_int()
	acc_fetch_tfarg_str()
	acc_fetch_itfarg_str()
Timescale and precision	acc_fetch_precision
	acc_fetch_timescale_info()
Port direction and index	acc_fetch_direction()
	acc_fetch_index()
Module path and related information	acc_fetch_edge()
	acc_fetch_polarity()
	acc_fetch_pulsere()

After retrieving the property, *modify* routines are used to change its value, if it needs to be modified (and if the property is writable). The next table shows which function to use for modifying a certain object.

Table 11. How to use modify routines

To modify this	Use one of these functions
Delay value	acc_append_delays()
	acc_replace_delays()
Logic and strength value	acc_set_value()
Pulse handling values	acc_append_pulsere()
	acc_replace_pulsere()
	acc_set_pulsere()

Value change link (VCL) routines are used for monitoring a value change in an object and a subsequent callback. There are two VCL routines – `acc_add_vcl()` and `acc_delete_vcl()` – used for adding and deleting a value change link respectively.

As part of each iteration, as shown in the beginning of this section, the application may need to do some housekeeping jobs. This is done using *miscellaneous* routines. Such housekeeping jobs include memory allocation/de-allocation, setting configuration parameters etc. The following table lists the miscellaneous routines against their functionality.

Table 12. How to use miscellaneous functions

To do this	Use one of these functions
Managing the access routine environment	acc_initialize()

To do this	Use one of these functions
	acc_close()
	acc_configure()
	acc_reset_buffer()
	acc_product_type()
	acc_product_version()
	acc_version()
	acc_set_scope()
	acc_set_interactive_scope()
Accessing all target objects simulataneously	acc_count()
	acc_collect()
	acc_free()
Comparing and matching objects	acc_compare_handles()
	acc_object_in_typelist()
	acc_object_of_type()

If the current object is part of an array, the handle to the next object (for example, in case of a port bit, the next port bit) can be obtained using *next* routine.All next functions start with a common prefix *acc_next_*.

Table 13. How to use next routines

To get a handle to the next object, when it is of type	Use one of these functions
Objects inside a module	acc_next_cell()
	acc_next_child()
	acc_next_modpath()
	acc_next_net()
	acc_next_parameter()
	acc_next_port()
	acc_next_portout()
	acc_next_primitive()
	acc_next_specparam()
	acc_next_tchk()
	acc_next_terminal()
Objects related to the design hierarchy	acc_next_scope()
	acc_next()
	acc_next_topmod()
Objects related to driving or loading of a net	acc_next_cell_load()
	acc_next_load()
	acc_next_driver()
	acc_next_hiconn()
	acc_next_loconn()
Objects related to a vector	acc_next_bit()

To get a handle to the next object, when it is of type	Use one of these functions
Objects related to a module path	acc_next_input() acc_next_output() acc_release_object() [This is not really a next routine, but is closely associated with the above two functions].

Note that some of the acc functions (such as `acc_fetch_name()`) return a character pointer. The string corresponding to this character pointer is copied to an internal buffer. This internal buffer is overwritten whenever another acc function from the user application returns a character pointer. In order to save the previous string, you must copy it to a user-defined string.

5. HANDLE ROUTINES

As mentioned in the last section, the object oriented nature of the access routines almost entirely relies on the use of the newly introduced data type `handle`. Using handle, the user can treat a design object in the same way as any other variable type, such as, integer or string.

A handle to a given object is obtained by using one of the *Handle* routines. The given object is specified using one or more of its properties which are passed to the handle routine as input arguments. A number of different handle routines are provided to work with different input properties.

5.1 Getting a handle to a scope or object

5.1.1 Functions in this category

```
handle acc_handle_scope(objHndl)
handle objHndl;

handle acc_handle_by_name(objName, scopeHndl)
char   *obj_name;
handle scopeHndl;

handle acc_handle_object(objName)
char   *objName;
```

```
handle acc_handle_interactive_scope()
```

5.1.2 Purpose

`acc_handle_scope()` is used to obtain a handle to the scope of an object, whose handle `objHndl` is passed as the argument.

On the other hand, if the name of the object and its scope are available, then a handle to the object can be obtained using `acc_handle_by_name()`.

If the object is under the current scope or its full pathname is available, we can get a handle to it by passing only its name to the function `acc_handle_object()`.

Both `acc_handle_by_name()` and `acc_handle_object()` can return handle to module, primitive, parameter, specparam, net, named block and event, register, Verilog task and function and variable type. `acc_handle_object()` can additionally return handle to module path, module port and data path. In both cases, the name of the object is passed as a string. If a full hierarchical path (starting with the name of the top level module) is given, the function does not perform any search and a handle to the specified object is returned. However, if there is no path name or a relative path name is specified, a search for the object is performed according to the search rules of Verilog language. No such search is performed for `acc_handle_by_name()` and the object is assumed to be in the scope mentioned. If the scope is `NULL`, the object name is taken as the full hierarchical pathname.

The function `acc_handle_interactive_scope()` returns a handle to the current interactive scope of the simulator.

5.1.3 Example

Let us first look at the following Verilog program which we will use as our reference in next few sections. In this example program, there are two highest-level modules in the design hierarchy called `top` and `mod_handle`. The module `top` instantiates two child modules `m0` and `m1`, whereas `mod_handle` invokes a user-defined system call `$get_handle()` twice. First, it calls the task with an argument of `"top.m1"` followed by another call with the argument `"top.m0.begin_reset"`. Let us assume that the purpose of the system call `$get_handle()` is to retrieve a handle to an object whose full pathname has been passed to it as an argument. (We will see how to use these handles in next few sections).

```
module top;
wire rst;

   reset_block m0 (rst);  // rst is output
   main_block  m1 (rst);  // rst is input

endmodule

module reset_block (reset);
output reset;
reg r_reset;
wire reset;

   assign reset = r_reset;
   initial  begin: begin_reset
      r_reset = 1'b0;
      #100;
      r_reset = 1'b1;
   end
endmodule

module main_block(reset);
input reset;

   reg [1:0] clk;

   initial
      clk = 2'b0;

   always
      if (reset === 1'b1)
      #5 clk[0] = ~clk[0];
      else
      #5 clk[1] = ~clk[1];

endmodule

module mod_handle;
   initial begin
      $display($time,"-Before Calling the PLI routine-");
```

```
        $get_handle("top.m1");
    #1
        $get_handle("top.m0.begin_reset");
        $display($time," -After Calling the PLI routine-");
    #100 $finish;
    end
endmodule
```

The following code listing shows how to implement the task `$get_handle`. Note that, in this example, since the Verilog program provides the full pathname of the object, the second argument to the function `acc_handle_by_name()` is `NULL`.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{
   handle reqdHndl;

   acc_initialize();
   reqdHndl = acc_handle_by_name(tf_strgetp(1), NULL);
   /* Get a handle to the object in argument 1.
      If argument 1 is "top.m0.begin_reset", reqHndl
      will be a handle to the begin block */
   if (!reqdHndl)
      tf_error(" Object %s not found\n", tf_strgetp(1));
   acc_close();
}
```

5.2 Getting a handle for the calling module and parent of an object

5.2.1 Functions in this category

```
handle acc_handle_calling_mod_m()

handle acc_handle_parent(objHndl)
```

```
handle objHndl;
```

5.2.2 Purpose

The function `acc_handle_calling_mod_m()` in an application returns a handle to the module that invoked the current instance of the system call associated with the application.

`acc_handle_parent()` returns the handle to the parent module of an object specified by the handle `objHndl` with two exceptions. For a primitive terminal, it returns a handle to the primitive containing the terminal and for a port bit, a handle to the port containing the bit is returned.

5.2.3 Example

We will use the same Verilog program of Section 5.1.3 here, but this time we will assume `$get_handle()` retrieves a handle to the parent of the object passed to it as argument. The following listing shows an implementation of `$get_handle()`. If the passed argument is `"top.m0.begin_reset"`, `parentHndl` in the following code will be a handle to the module `top.m0`.

```
#include <stdio.h>
#include "vcsuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{

   handle reqdHndl, parentHndl;

   acc_initialize();

   reqdHndl = acc_handle_by_name(tf_strgetp(1), NULL);
   /* Get a handle to the object in argument 1 */
   if (!reqdHndl)
      tf_error(" Object %s not found\n", tf_strgetp(1));

   parentHndl = acc_handle_parent(reqdHndl);
   if (!parentHndl)
```

```
        tf_error(" Parent of object %s not found\n",
                 tf_strgetp(1));

    acc_close();
}
```

5.3 Getting a handle to port and related objects

5.3.1 Functions in this category

```
handle acc_handle_port(modHndl, portIndex)
handle modHndl;
int    portIndex;

handle acc_handle_hiconn(portHndl)
handle portHndl;

handle acc_handle_loconn(portHndl)
handle portHndl;

handle acc_handle_path(opPortHndl, ipPortHndl)
handle opPortHndl, ipPortHndl;
```

5.3.2 Purpose

The function `acc_handle_port()` returns a handle to a module port. A handle to the module and the index of the port are passed as arguments to the function. The index of the port is determined by its position in the port list starting with 0 for the leftmost port, 1 for the next one on its right and so on.

The *hiconn* of a port is defined as the net connected to the port at the scope where the parent module of the port is instantiated. Similarly, *loconn* of a port is defined as the net connected to the port within the module. For example, in the Verilog program given in Section 5.1.3, `top.rst` and `top.m0.reset` are the hiconn and loconn nets for the port `reset` of `m0`. The functions `acc_handle_hiconn()` and `acc_handle_loconn()` are used to get handles to the hiconn and loconn nets connected to a port. The argument to these functions can be a handle to either the entire port or its bit-select.

`acc_handle_path()` is used to get a handle to an intermodule path. The arguments of this function are handles to an output/inout port and an input/inout port respectively.

5.3.3 Example

Continuing with our Verilog program of Section 5.1.3, in this section we will further enhance the system call `$get_handle()`. In Section 5.1.3, we started with a version of `$get_handle()` which retrieved the handle to the object passed to it as an argument. We enhanced it in Section 5.2.3 to retrieve the handle to the parent module of the passed object. Here we are going to extend it further to retrieve handles to the hiconn and loconn nets connected to the first port of the of the parent module of the object. For example, if the passed argument is `"top.m0.begin_reset"`, in the following code, `port0Hndl` will be a handle to the port `reset` of `m0` and `netHiHndl` and `netLoHndl` will be handles to the nets `top.rst` and `top.m0.reset` respectively.

```
#include <stdio.h>
#include "vcsuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{
   handle reqdHndl, parentHndl, port0Hndl;
   handle netHiHndl, netLoHndl;

   acc_initialize();

   reqdHndl = acc_handle_by_name(tf_strgetp(1), NULL);
   /* Get a handle to the object in argument 1 */
   if (!reqdHndl) {
      tf_error(" Object %s not found\n", tf_strgetp(1));
      tf_dostop();
   }

   parentHndl = acc_handle_parent(reqdHndl);
   if (!parentHndl) {
      tf_error(" Parent of the object %s not found\n",
               tf_strgetp(1));
      tf_dostop();
   }
```

```
    port0Hndl = acc_handle_port(parentHndl, 0);
    /* The second arg == 0 means it is the first port */
    if (!port0Hndl) {
       tf_error(" Port0 not found\n");
       tf_dostop();
    }

    netHiHndl = acc_handle_hiconn(port0Hndl);
    netLoHndl = acc_handle_loconn(port0Hndl);
    if ((!netHiHndl) && (!netLoHndl)) {
       tf_error(" Nets not found\n");
       tf_dostop();
    }

    acc_close();
}
```

5.4 Getting a handle to module path, datapath and related objects

5.4.1 Functions in this category

```
handle acc_handle_modpath(modHndl,
       srcnetName, destnetName,
       srcnetHndl, destnetHndl)
handle modHndl;
char   *srcnetName, *destnetName;
handle srcnetHndl, destnetHndl;

handle acc_handle_datapath(modpathHndl)
handle modpathHndl;

handle acc_handle_pathin(pathHndl)
handle pathHndl;

handle acc_handle_pathout(pathHndl)
handle pathHndl;

handle acc_handle_condition(pathHndl)
handle pathHndl;
```

5.4.2 Purpose

The function `acc_handle_modpath()` returns a handle to a module path (a module path is defined as a specify block path for delays in a Verilog source description). The last two arguments of the functions are optional and are used only when `accEnableArgs` is set to `acc_handle_modpath` and either `srcnetName` or `destnetName` is NULL.

To get a handle associated with the datapath (defined as an edge sensitive part of a module path), the function `acc_handle_datapath()` is used.

The functions `acc_handle_pathin()` and `acc_handle_pathout()` return the first (unless it is the only) input and output of the module path respectively. Notice that, a module path (and so a datapath) can have more than one input source or output destination.

The function `acc_handle_condition()` returns a handle to a conditional expression for the specified module path, datapath or timing check terminal, if one exists; otherwise it returns a NULL.

5.4.3 Example

Consider the following Verilog source description which has a specify block as shown below.

```
module top;

wire reset, a, clock, clock2, qq, dd;

specify
   if (!reset & a)
      (posedge clock, clock2 => (qq +: dd)) = (10, 8);
endspecify

initial
#10 $modpath("top", "clock2", "qq");

endmodule
```

This means that the effect of any change in `clock2` or a positive edge of `clock` on `qq` will be visible after a rise delay of 10 units and a fall delay of 8 units. The specify block also tells that the input `dd` is propagated to the

output qq un-inverted. In this example, clock and clock2 are two sources and qq is the destination and a data path extends from dd to qq.

Now, let us design the system call $modpath shown in the above example. $modpath takes

1. the full pathname of the module where the specify block is instantiated (which, in the above example, is "top"),
2. one of the source points ("clock2" in our example) and
3. the destination point ("qq")

and prints

1. the full pathname of the module path,
2. the first source point for the module path,
3. the destination point and
4. also detects whether the module path is conditional (which it is in our example).

The following function shows how to implement $modpath. To save space, only the calltf function has been shown.

```
int myCalltf();
/* This is the calltf function */
int myCalltf()
{
handle modHndl, modPathHndl, dataPathHndl;
handle pathInHndl, pathOutHndl, pathCondHndl;

acc_initialize();

/* Step 1: Print the full pathname of the module path */
modHndl = acc_handle_by_name(tf_strgetp(1), NULL);
modPathHndl = acc_handle_modpath(modHndl,
                                 tf_strgetp(2),
                                 tf_strgetp(3));
if (!acc_error_flag)
   io_printf(" Module Path name: %s\n",
             acc_fetch_fullname(modPathHndl));

/* Step 2: Print the first source point
   of the module path */
pathInHndl = acc_handle_pathin(modPathHndl);
if (!acc_error_flag)
   io_printf(" Path in: %s\n",
```

```
                acc_fetch_name(pathInHndl));

/* Step 3: Print the destination point */
pathOutHndl = acc_handle_pathout(modPathHndl);
if (!acc_error_flag)
   io_printf(" Path Out: %s\n",
                acc_fetch_name(pathOutHndl));

/* Step 4: Find out whether the module path is
   conditional */
pathCondHndl = acc_handle_condition(modPathHndl);
if (pathCondHndl)
   io_printf(" The module path is conditional\n");
else
   io_printf(" The module path is unconditional\n");

acc_close();
}
```

5.5 Getting a handle to timing check and related objects

5.5.1 Functions in this category

```
handle acc_handle_tchk(modHndl, timingChkType,
       arg1NetName, arg1EdgeType,
       arg2NetName, arg2EdgeType,
       arg1ConnHndl, arg2ConnHndl)
handle modHndl;
int    timingCheckType;
char   *arg1NetName, *arg2NetName;
int    arg1EdgeType, arg2EdgeType;
handle arg1ConnHndl, arg2ConnHndl;

handle acc_handle_notifier(timingChkHndl)
handle timingChkHndl;

handle acc_handle_tchkarg1(timingChkHndl)
handle timingChkHndl;

handle acc_handle_tchkarg2(timingChkHndl)
```

```
handle timingChkHndl;
```

5.5.2 Purpose

The functions in this category are related to the following timing check system tasks: `$setup`, `$hold`, `$setuphold`, `$period`, `$skew`, `$recovery`, `$width` and `$nochange`.

The function `acc_handle_tchk()` returns a handle for the specified timing check of a module or cell. The module and the nature of the timing checks are specified using the first two arguments of the function. The allowed timing check types are `accHold`, `accNochange`, `accPeriod`, `accRecovery`, `accSetup`, `accSkew` and `accWidth`. The next two arguments determine the net name and its edge that activates the timing check. The edge may be specified as one of these – `accNegedge`, `accNoedge` and `accPosedge`. Alternately, edges can be more precisely defined as a list of '+' separated elements from either `accEdge01`, `accEdge0x`, `accEdgex1` or `accEdge10`, `accEdge1x`, `accEdgex0`. If the timing check needs a second net, the next two arguments specify its name and the edge type in the same manner as the first net. The last two arguments are optional and used to specify the handles to the nets only if `accEnableArgs` is set and both `arg1NetName` and `arg2NetName` are `NULL`.

On the other hand, if the handle to the timing check is available, handles for the timing check terminals connected to the first and the second net (if exists) can be obtained by using `acc_handle_tchkarg1()` and `acc_handle_tchkarg2()` respectively. The function `acc_handle_notifier()` is used to get a handle to the notifier register of a timing check.

5.5.3 Example

The following example shows how to use the above functions in a PLI routine. Consider the following Verilog source description which has a specify block as shown below.

```
`define accSetup   366
`define accPosedge 13
`define accNegedge 50
`define accNoedge  0
/* These values are defined in acc_user.h */
```

```
module top;
wire reset, a, clock, qq, dd;
reg notifier_reg;

specify
   specparam tSU = 10;
   (posedge clock => (qq +: dd)) = (10, 8);
   $setup (posedge clock, qq, tSU, notifier_reg);
endspecify
endmodule

module try;
   initial
   #10 $timing_chk("top", `accSetup,
                   "clock", `accPosedge,
                   "qq", `accNoedge);
endmodule
```

This is similar to the program we have used for accessing properties of module paths in the last section. Here, we have a setup timing check within the specify block to monitor any timing violation between the signals `clock` and `qq`. A change in `qq` should occur no later than `tSU` time unit after the posedge of clock, otherwise `notifier_reg` will be set. (Of course, in this trivial example, since both the rise time (10) and the fall time (8) are less than or equal to `tSU` (10), the setup violation never occurs)

The system call `$timing_chk`, whose implementation is shown below, takes the following arguments:

1. the full pathname of the module where the specify block is instantiated,
2. the type of the timing check for this module,
3. the name of the net associated with the first timing check argument,
4. its edge sensitivity,
5. the name of the net associated with the second timing check argument,
6. the edge sensitivity.

It then prints the full pathname of

1. the notifier register and
2. the full pathnames of the two timing check arguments.

The following function shows how to implement `$timing_chk`. Only the calltf function has been shown to save space.

```
#include <stdio.h>
```

```
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf() {

   handle modHndl, timingChkHndl, notifierHndl, netHndl;
   int timingChkType, edgeType1, edgeType2;

   acc_initialize();

   /* Get a handle to the module handle */
   modHndl = acc_handle_by_name(tf_strgetp(1), NULL);

   /* Get the values of timing check and edge types */
   timingChkType = tf_getp(2);
   edgeType1 = tf_getp(4);
   edgeType2 = tf_getp(6);

   timingChkHndl= acc_handle_tchk(modHndl, timingChkType,
                     tf_strgetp(3), edgeType1,
                     tf_strgetp(5), edgeType2);

   notifierHndl = acc_handle_notifier(timingChkHndl);
   io_printf("$timing_chk: notifier reg name: %s\n",
             acc_fetch_fullname(notifierHndl));

   netHndl = acc_handle_tchkarg1(timingChkHndl);
   io_printf("$timing_chk: Timing check arg1 name: %s\n",
             acc_fetch_fullname(netHndl));

   netHndl = acc_handle_tchkarg2(timingChkHndl);
   io_printf("$timing_chk: Timing check arg2 name: %s\n",
             acc_fetch_fullname(netHndl));

   acc_close();
}
```

5.6 Getting a handle to simulated net and terminal

5.6.1 Functions in this category

```
handle acc_handle_simulated_net(collapsedNetHndl)
handle collapsedNetHndl;

handle acc_handle_terminal(primitiveHndl,
       terminalIndex)
handle primitiveHndl;
int    terminalIndex;

handle acc_handle_conn(terminalHndl)
handle terminalHndl;
```

5.6.2 Purpose

A *simulated net* is the one that is used for simulation, when essentially a single physical net runs in different names in different hierarchies and is collapsed by the simulator. `acc_handle_simulated_net()` returns a handle to a simulated net associated with a collapsed net.

The function `acc_handle_terminal()` is used for getting handle to a primitive terminal by its position (starting as 0 for the leftmost terminal and incrementing for successive terminals to its right). Note that, this function works only for terminals of a primitive (gate, switch or UDP declaration).

The function `acc_handle_conn()` returns a handle to the net connected to a primitive (or path or timing check) terminal.

5.6.3 Example

A function `return_terminal_net()` is shown below, which returns a handle to a net connected to a primitive terminal. The handle to the primitive and the index of the terminal are passed as parameters to the function.

```
handle return_terminal_net(primHndl, termIndex)
handle primHndl;
int termIndex;
{
   handle termHndl;
   termHndl = acc_handle_terminal(primHndl,termIndex);
```

```
    return(acc_handle_conn(termHndl));
}
```

5.7 Getting a handle to the parameters of a system call

5.7.1 Functions in this category

```
handle acc_handle_tfinst()

handle acc_handle_tfarg(pin)
int    pin;

handle acc_handle_itfarg(pin, instHndl)
int    pin;
handle instHndl;
```

5.7.2 Purpose

The function `acc_handle_tfinst()` returns a handle to the user-defined system call.

Both the functions `acc_handle_tfarg()` and `acc_handle_itfarg()` are used to obtain handles to the parameters of the system call. A parameter is specified by its parameter index number (PIN), (defined as the position of that parameter in the argument list of the call with PIN of the leftmost parameter designated as 1). Note that, the system call argument can be any unquoted Verilog HDL identifier or quoted string name of any object.

The handle obtained from `acc_handle_tfinst()` can be used later with `acc_handle_itfarg()` for accessing parameters passed to that particular instance of the system call.

Also note that many simulators do not support functions which deal with instance pointer.

5.7.3 Example

The following code fragment gets a handle to the first parameter of the system call.

```
handle regHndl;
   /* let us assume the first parameter is a reg */
```

```
regHndl = acc_handle_tfarg(1);
```

This handle can be used later in the program for obtaining more information regarding the parameter.

6. FETCH ROUTINES

The *fetch* routines do just the opposite of what handle routines are meant for – they retrieve the value of a property of an object whose handle is passed as an argument. A number of fetch routines have been designed to cover different properties of an object, such as, its value, its attribute, the number of objects its related to etc.

According to their functionalities, the fetch routines can be categorized as follows.

6.1 Command line arguments

6.1.1 Functions in this category

```
int acc_fetch_argc()

char **acc_fetch_argv()
```

6.1.2 Purpose

The function `acc_fetch_argc()` returns the number of command line arguments and `acc_fetch_argv()` returns an array of string containing these arguments.

6.1.3 Example

```
void print_arg()
{
   int i;
   int argc = acc_fetch_argc();
   char **argv = acc_fetch_argv();

   if (argc > 1)
      for (i=1; i<argc; i++)
```

```
            io_printf("Arg. #%d: %s\n",i,argv[i]);
    else
        io_printf("No command line argument\n");
}
```

6.2 Attribute, parameter, specparam

6.2.1 Functions in this category

```
double acc_fetch_attribute(objHndl, attr, defaultVal)
handle objHndl;
char *attr;
double defaultVal;

int acc_fetch_attribute_int(objHndl, attr, defaultVal)
handle objHndl;
char *attr;
double defaultVal;

char *acc_fetch_attribute_str(objHndl, attr, defaultVal)
handle objHndl;
char *attrStr;
double defaultVal;

int acc_fetch_paramtype(paramHndl)
handle paramHndl;

double acc_fetch_paramval(paramHndl)
handle paramHndl;
```

6.2.2 Purpose

The functions `acc_fetch_attribute()` and its variants return the value of a parameter or specparam that is declared as an attribute in the source description. While `acc_fetch_attribute()` returns the value as double, `acc_fetch_attribute_int()` and `acc_fetch_attribute_str()` returns it as integer and string respectively.

Among the arguments, `attr` is the left hand side of the specparam statement that defines the attribute. If an attribute specified by `attr` is not found for an object whose handle is `objHndl`, a default value is returned.

This default value is 0, if `accDefaultAttr0` is set to `"true"` using `acc_configure()` or `defaultVal` when `accDefaultAttr0` is `"false"`. `defaultVal` is optional and can be omitted when `accDefaultAttr0` is `"true"`.

`acc_fetch_paramtype()` and `acc_fetch_paramval()` are used to obtain the data type and value of a parameter or specparam.

`acc_fetch_paramtype()` returns one of the following four pre-defined integer constants to specify the data type of the parameter: `accIntParam`, `accIntegerParam`, `accRealParam` and `accStringParam`.

Note that `acc_fetch_paramval()` returns a double. To obtain the value of an integer or string parameter, you must cast the return value to `(int)` or `(char *)(int)`.

6.2.3 Example

In this example we are going to create a system call `$attr_chk`, which takes signal and attribute name pairs and prints their values. The number of such pairs is specified as the first parameter to the system call. It also prints the values of all parameters specified after the signal and attribute name pairs.

The following Verilog program invokes the system call `$attr_chk`.

```
module top;
wire clock, out;

specify
   specparam tSetUp_clock = 10;
endspecify

parameter my_parameter = "thirteen";
endmodule

module try;
   initial
   #10 $attr_chk(2, "tSetUp_", "clock", "tSetUp_", "out",
                 "my_parameter");
endmodule
```

The following PLI application shows the implementation of `$attr_chk`.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{
   handle netHndl, paramHndl;
   int i;

   acc_initialize();
   acc_configure(accDefaultAttr0, "false");

   for (i=1; i<=tf_getp(1); i++) {
      netHndl =
         acc_handle_by_name(tf_strgetp(2*i+1), NULL);
      io_printf(" The setup time for %s= %d\n",
                tf_strgetp(2*i+1),
                (int)acc_fetch_attribute(netHndl,
                                         tf_strgetp(2*i),
                                         8.0));
   }

   if (tf_nump() >= 2*tf_getp(1)+2)
      for (i=2*tf_getp(1)+2; i<=tf_nump(); i++) {
         paramHndl =
            acc_handle_by_name(tf_strgetp(i), NULL);
         io_printf(" Value of the parameter = ");
         switch(acc_fetch_paramtype(paramHndl)) {
         case accRealParam:
            io_printf("%lf\n",
                      acc_fetch_paramval(paramHndl));
         case accIntegerParam:
            io_printf("%d\n",
                     (int)acc_fetch_paramval(paramHndl));
         case accStringParam:
            io_printf("%s\n",
             (char *)(int)acc_fetch_paramval(paramHndl));
         }
      }
```

```
    acc_close();
}
```

The output of the simulation with the above Verilog program is shown below. Note that, since there is no setup time defined for the signal `out`, the program prints the default value 8.

```
The setup time for clock= 10
The setup time for out= 8
Value of the parameter = thirteen
```

6.3 Object name, type and location

6.3.1 Functions in this category

```
char *acc_fetch_defname(objHndl)
handle objHndl;

char *acc_fetch_fullname(objHndl)
handle objHndl;

char *acc_fetch_name(objHndl)
handle objHndl;

int acc_fetch_fulltype(objHndl)
handle objHndl;

int acc_fetch_type(objHndl)
handle objHndl;

char *acc_fetch_type_str(type)
int type;

void acc_fetch_location(locPtr, objHndl)
p_location locPtr;
handle objHndl;

char *acc_fetch_value(objHndl, format, value)
handle objHndl;
char *format;
```

```
s_acc_value *value;
```

6.3.2 Purpose

The function `acc_fetch_defname()` returns the definition name of a module or primitive instance as a string. Similarly, `acc_fetch_name()` returns the instance name of any named object or module path. In the following instantiation, `core_logic` is the definition name, `u1` and `u2` are the instance names.

```
core_logic u1 (net11, ..., net1m);
core_logic u2 (net12, ..., net2m);
```

In addition to module and primitive instances, `acc_fetch_name()` also supports the following type of named objects: module port, module path, data path, primitive, net, register, variable, named event, parameter, specparams, named blocks, tasks and functions. `acc_fetch_fullname()` does the same task as `acc_fetch_name()`, but returns the full hierarchical name.

The functions `acc_fetch_fulltype()` and `acc_fetch_type()` return the *fulltype* and *type* of an object respectively as a predefind integer. The type of an object specifies the generic class that it belongs to (for example, a primitive has a type `accPrimitive`), whereas the fulltype gives its detailed and specific classification (for example, `accAndGate` for a primitive AND gate). The returned integer from these functions can be passed to `acc_fetch_type_str()` to obtain the type as a string.

The function `acc_fetch_location()` returns a pointer `locPtr` to the predefined structure type `t_location`. This structure contains information regarding the file name and the line number of the Verilog source file where the object specified by the handle `objHndl` is located. The definition of the structure is shown below.

```
typedef  struct t_location {
   int line_no;       /* line number in the file */
   char *filename;    /* file name */
} s_location, *p_location;
```

The following table shows the object types supported by `acc_fetch_location()`.

Table 14. Object types supported by `acc_fetch_location()`

Object type	Location returned
Module	Module instantiation
Module port	Module definition
Module path, data path	Module path line
Primitive	Instantiation line
Explicit net, register, variable, named event, parameter, specparam, named block, task and function	Definition line
Implicit net	Line where first used

The function `acc_fetch_value()` gets the logic or strength value of a net, register or variable (but *not* of a memory element) as specified by the `objHndl`. The value can be returned either as a string or an `aval/bval` pair (refer to Section 5.1.3, Chapter 3 for the definition of `aval/bval`) depending on the `format` string. If the `format` string is specified as "%b", "%d", "%h" or "%o", the returned value is a string containing the logic value in binary, decimal, hexadecimal or octal format. If the `format` string is "%v", the strength value is returned instead. In all such cases, the third argument `value` to the function is optional and can be omitted.

However, if the `format` string is "%%", the value of the object is placed in a structure of type `s_acc_value` and pointed to by the `value`. The definition of `s_acc_value` is as shown below.

```
typedef  struct t_setval_value {
   int format;
   union {
      char *str;
      int scalar;
      int integer;
      double real;
      p_acc_vecval vector;
   } value;
} s_setval_value, *p_setval_value,
  s_acc_value, *p_acc_value;
```

The value is returned in one of the fields of the union `value`, depending on the format field specified by the user. The following table shows different values of the `format` field.

Table 15. Permitted values of the `format` field

Value of the `format` field	Used field of the union `value`
accBinStrVal, accOctStrVal, accDecStrVal,	str

Value of the `format` field	Used field of the union `value`
accHexStrVal and accStringVal	
accScalarVal	scalar
accIntVal	integer
accRealVal	real
accVectorVal	vector

In case of a vector, the `value` is returned in a structure `vector` of type `s_acc_vecval` pointed to by the `p_acc_vecval`. This structure is defined as shown below and works in the same way as of `s_vecval` structure for utility routines.

```
typedef struct t_acc_vecval {
   int aval;
   int bval;
} s_acc_vecval, *p_acc_vecval;
```

6.3.3 Example

Following is the calltf function for a system call `$type`, which takes the names of a number of nets and prints their values and locations.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{
   handle netHndl, modHndl;
   int i;
   s_location loc_ptr;

   acc_initialize();

   for (i=1; i<=tf_nump(); i++) {
      netHndl = acc_handle_by_name(tf_strgetp(i), NULL);
      io_printf("%s:\n", acc_fetch_fullname(netHndl));
      io_printf("\tValue: 0x%s\n",
                acc_fetch_value(netHndl, "%h", NULL));
      acc_fetch_location(&loc_ptr, netHndl);
      io_printf("\tFilename: %s and line number: %d\n",
```

```
                    loc_ptr.filename, loc_ptr.line_no);
    }

    acc_close();
}
```

A trivial example of a Verilog program invoking the system call `$type` is listed below with line numbers.

```
1 module top;
2 wire clock,
3      out;
4
5   initial
6   #10 $type("clock", "out");
7 endmodule
```

Assuming that the program is kept in the file `test.v`, it produces the following output.

```
top.clock:
        Value: 0xz
        Filename: test.v and line number: 2
top.out:
        Value: 0xz
        Filename: test.v and line number: 3
```

6.4 Delays

6.4.1 Functions in this category

There are two functions in this category. They are `acc_fetch_delay_mode()` and `acc_fetch_delays()`. The prototype declaration of `acc_fetch_delay_mode()` is shown below.

```
int acc_fetch_delay_mode(modHndl)
handle modHndl;
```

Depending on the configuration parameter `accMinTypMaxDelays` and the type of the input object, the function `acc_fetch_delays()` takes variable number of arguments.

When `accMinTypMaxDelays` is set to "false", the function `acc_fetch_delays()` has three different versions for fetching the delays associated with different object types.

For primitives, ports, port bits and intermodule paths:

```
int acc_fetch_delays(objHndl, rise_delay,
    fall_delay, z_delay)
handle objHndl;
double *rise_delay, *fall_delay, *z_delay;
```

For module paths:

```
int acc_fetch_delays(objHndl,
    d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12)
handle objHndl;
double *d1, *d2, *d3, *d4, *d5, *d6, *d7,
       *d8, *d9, *d10, *d11, *d12;
```

For timing checks:

```
int acc_fetch_delays(objHndl, limit)
handle objHndl;
double *limit;
```

When `accMinTypMaxDelays` is set to "true", the function prototype takes the following form.

```
int acc_fetch_delays(objHndl, delay_ptr)
handle objHndl;
double *delay_ptr;
```

6.4.2 Purpose

The delay mode of a module or cell instance determines the nature of the delays for primitives and paths within that instance. This can be specified by a set of plusargs in the command line. Depending on these plusargs, the function `acc_fetch_delay_mode()` returns the delaymode as one of the following predefined integers as shown in the table below.

Table 16. Implications of different delay modes

Used plusarg	Returned delay mode	Primitive delay	Path delay	MTM delay

Used plusarg	Returned delay mode	Primitive delay	Path delay	MTM delay
None	accDelayModeNone	*No delay mode has been specified for the module or cell instance*		
+delay_mode_zero	accDelayModeZero	0	ignored	not applicable
+delay_mode_unit	accDelayModeUnit	1	ignored	not applicable
+delay_mode_distrib	accDelayModeDistrib	used	ignored	not applicable
+delay_mode_path	accDelayModePath	ignored	used	not applicable
+delay_mode_MTM	accDelayModeMTM	not used	not used	See below

When the configuration parameter `accMinTypMaxDelays` is set to "false", the function `acc_fetch_delays()` is used for obtaining the single delay values for a number of different objects. (As opposed to minimum:typical:maximum triplet, single delay value simulation model applies a single delay value in all cases). As shown in the function prototype declarations above, in such a case, depending on the nature of the object, the function can have three different forms as described below.

When `objHndl` points to an object of type primitive, port, port bit or intermodule path, the second and third arguments of the function contain the rise and fall delays of the object respectively upon return. If the object is not a 2-state primitive, a third argument returns the "to-Z" delay.

When the object specified by `objHndl` is of type module path, the function may return upto 12 different delay values. What these delay values imply, depends on the value of the configuration parameter `accPathDelayCount` and is shown in the following table.

Table 17. Delay values for a module path

Argument	accPathDelayCount	The value indicates
d1	1	delay for all transitions for module paths
	2, 3	rise transition delay for module paths
	6, 12	0→1 transition delay for module paths
d2	2, 3	fall transition delay for module paths
	6, 12	1→0 transition delay for module paths
d3	3	"to-Z" transition delay for module paths
	6, 12	0→Z transition delay for module paths
d4	6, 12	Z→1 transition delay for module paths
d5	6, 12	1→Z transition delay for module paths
d6	6, 12	Z→0 transition delay for module paths
d7	12	0→X transition delay for module paths
d8	12	X→1 transition delay for module paths
d9	12	1→X transition delay for module paths
d10	12	X→0 transition delay for module paths
d11	12	X→Z transition delay for module paths
d12	12	Z→X transition delay for module paths

When `objHndl` points to an object of type timing check, the `limit` contains the limit of the timing check upon return.

When the configuration parameter `accMinTypMaxDelays` is set to "true", `acc_fetch_delays()` returns the minimim, typical and maximum values for each of the delays associated with an object. The delay values are returned in an array of `double` as specified by the pointer `delay_ptr`. Since there may be as many as 12 different delays for a module path, when `accMinTypMaxDelays` is set to "true", the number goes upto 36.

6.4.3 Example

The following example shows how to use `acc_fetch_delay_mode()`. It fetches the delay mode of the module named `top` in the design.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
char *delaymode();

int myCalltf()
{
handle modHndl;

   acc_initialize();

   modHndl = acc_handle_by_name("top", NULL);
   io_printf(" Delay mode for %s is:%s\n",
      acc_fetch_fullname(modHndl),
      delaymode(acc_fetch_delay_mode(modHndl)));

   acc_close();
}

char *delaymode(n)
int n;
{
   switch (n) {
   case accDelayModeNone : return("accDelayModeNone");
```

```
    case accDelayModePath : return("accDelayModePath");
    case accDelayModeZero : return("accDelayModeZero");
    case accDelayModeUnit : return("accDelayModeUnit");
    case accDelayModeDistrib :
       return("accDelayModeDistrib");
    case accDelayModeMTM : return("accDelayModeMTM");
    default:  return("Oops !");
    }
}
```

We will postpone creating an application using `acc_fetch_delay()` until we discuss `acc_replace_delays()` and `acc_append_delays()` in Section 7.1.

6.5 Vector size and range

6.5.1 Functions in this category

```
int acc_fetch_range(vecHndl, msb, lsb)
handle vecHndl;
int *msb, *lsb;

int acc_fetch_size(objHndl)
handle objHndl;
```

6.5.2 Purpose

The function `acc_fetch_range()` gets the most and the least significant bit positions of a vector specified by the handle `vecHndl`.

`acc_fetch_size()` returns the number of bits in a net, register or port.

6.5.3 Example

The function `my_acc_fetch_size()` in this example demonstrates one of the many redundancies found in PLI1.0 – it implements `acc_fetch_size()` using `acc_fetch_range()`. `my_acc_fetch_size()` takes the handle to a variable and prints the size, irrespective of whether it is a scalar or vector.

```
#include <stdio.h>
```

```
#include "veriuser.h"
#include "acc_user.h"

int my_acc_fetch_size();

int my_acc_fetch_size(netHndl)
handle netHndl;
{
   int msb, lsb;

   acc_fetch_range(netHndl, &msb, &lsb);
   return((msb > lsb) ? (msb-lsb-1) : (lsb-msb+1));
}
```

6.6 Task/function arguments

6.6.1 Functions in this category

```
double acc_fetch_tfarg(pin)
int pin;

double acc_fetch_itfarg(pin, tfinst)
int pin;
handle tfinst;

int acc_fetch_tfarg_int(pin)
int pin;

int acc_fetch_itfarg_int(pin, tfinst)
int pin;
handle tfinst;

char *acc_fetch_tfarg_str(pin)
int pin;

char *acc_fetch_itfarg_str(pin, tfinst)
int pin;
handle tfinst;
```

6.6.2 Purpose

All of the above functions fetch the value of a specified argument of a system call. The value is returned as a double, integer and string for `acc_fetch_tfarg()`, `acc_fetch_tfarg_int()` and `acc_fetch_tfarg_str()` respectively. In each case, the PIN is passed to the function.

If the return value of `acc_fetch_tfarg()` is required to be assigned to an integer or character pointer type variable, you must cast this return value to `(int)` or `(char *)(int)` before the assignment takes place.

```
int i;
char *string;
...
i = (int)acc_fetch_tfarg(param_number);
string = (char *)(int)acc_fetch_tfarg(param_number);
```

The function `acc_fetch_itfarg()` and its other variants are used to obtain the value of an argument to some other specific instance of the system call. In this case, the handle to that instance (obtained using `acc_handle_tfinst()`prior to calling this function) is passed to the function as the second argument.

6.6.3 Example

The following function `print_arg_val()` prints the numeric or literal value of an integer, time, real or string variable passed as parameter to a system call which utilizes this function.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int print_arg_val();

int print_arg_val()
{
   handle objHndl;
   int i;
```

```
    acc_initialize();

    for (i=1; i<=tf_nump(); i++) {
       if (tf_typep(i) == tf_string)
          io_printf("String arg: %s\n",
      acc_fetch_tfarg_str(i));
       else {
          objHndl = acc_handle_tfarg(i);
          io_printf("%s:", acc_fetch_fullname(objHndl));
          switch(acc_fetch_type(objHndl)) {
             case accIntegerVar:
             case accTimeVar:
               io_printf(" %d\n", acc_fetch_tfarg_int(i));
               break;
             case accRealVar:
              io_printf(" %lf\n", acc_fetch_tfarg(i));
              break;
             default:
               io_printf(" %d\n", acc_fetch_tfarg_int(i));
          }
       }
    }
    io_printf("\n");
    acc_close();
}
```

6.7 Timescale, precision

6.7.1 Functions in this category

```
int acc_fetch_precision()

void acc_fetch_timescale_info(objHndl,
     timescale_p)
handle objHndl;
p_timescale_info timescale_p;
```

6.7.2 Purpose

The function `acc_fetch_precision()` returns the smallest of all time precisions as specified by `` `timescale `` compiler directives in the entire source description. The precision is returned as an exponent of second. For instance, a returned value of -6 means that the smallest precision is 1 μs. If there is no `` `timescale `` compiler directive in the source description, the returned value is 0, indicating the default precision as 1 sec.

The function `acc_fetch_timescale_info()` is used to obtain the timescale information (both unit and precision) for an object or for an active `$timeformat` system call. If the `objHndl` is a handle to a module instance or module definition, the function gets the timescale information of the corresponding module definition. If it is a handle to a user-defined system call, the timescale information of the parent module definition is obtained. When `objHndl` is `NULL`, the function gets the timescale for an active `$timeformat` system call. In all cases, the timescale information is returned to an `s_timescale_info` structure pointed to by the `timescale_p` argument. The definition of `s_timescale_info` is given below.

```
typedef  struct t_timescale_info {
  short unit;
  short precision;
} s_timescale_info, *p_timescale_info;
```

The values of both these fields are expressed as an exponent of second.

6.7.3 Example

The following function `print_timescale_info()` prints the simulation timeunit and precision information of a module. A handle to the module is passed to the function as an argument. The function prints both the smallest precision information using `acc_fetch_precision()` as well as the timescale information (simulation time unit and precision) using `acc_fetch_timescale_info()`.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int print_timescale_info();

int print_timescale_info(objHndl)
```

```
handle objHndl;
{
   s_timescale_info timescale_info;

   acc_fetch_timescale_info(objHndl, &timescale_info);
   io_printf(" The smallest precision = (10)^%d sec.\n",
      acc_fetch_precision());

   io_printf(" The time unit = (10)^%d sec.\n",
      timescale_info.unit);
   io_printf(" The precision = (10)^%d sec.\n",
      timescale_info.precision);
}
```

6.8 Direction/value of a port

6.8.1 Functions in this category

```
int acc_fetch_direction(objHndl)
handle objHndl;

int acc_fetch_index(objHndl)
handle objHndl;
```

6.8.2 Purpose

The function `acc_fetch_direction()` returns the direction of a port or terminal as one of the predefined constants: `accInput` (for input only), `accOutput` (for output only), `accInout` (for bidirectional only) and `accMixedIo` (when a vectored port or terminal is a concatenation of input and output port bits).

The function `acc_fetch_index()` returns the index number for a port or terminal as defined in its original module definition. Starting from the leftmost port or terminal in the list, which has an index of 0, the index is incremented by one for each successive port or terminal.

6.8.3 Example

The following example shows the calltf function of a system call `$port`, which takes a module instance followed by a series of port indices as its

parameter and prints the direction of each of these ports. So, calling `$port` inside a Verilog program as shown below

```
$port(u1, 0, 3, 2, 6);
```

will print the directions of the 0th, 3rd, 2nd and 6th port of the instance `u1`. Note that, the number of port indices passed to the system call is not fixed, nor are they in any particular order.

```
#include <stdio.h>
#include "vcsuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{
   handle modHndl, portHndl;
   int i;
   char *dirn;

   acc_initialize();

   dirn = (char *)malloc(10);
   modHndl = acc_handle_tfarg(1);
   for (i=2; i<=tf_nump(); i++) {
      portHndl = acc_handle_port(modHndl, tf_getp(i));
      switch (acc_fetch_direction(portHndl)) {
         case accInput:
            strcpy(dirn, "input"); break;
         case accOutput:
            strcpy(dirn, "output"); break;
         case accInout:
            strcpy(dirn, "inout"); break;
         case accMixedIo:
            strcpy(dirn, "mixed IO "); break;
         default:
            tf_error(" Unsupported port direction\n");
      }
      io_printf(" The %dth port is %s port:\n",
         acc_fetch_index(portHndl), dirn);
```

```
            /* acc_fetch_index(portHndl) should
               be same as tf_getp(i) */
    }
    acc_close();
}
```

6.9 Module path

6.9.1 Functions in this category

```
int acc_fetch_edge(pathioHndl)
handle pathioHndl;

int acc_fetch_polarity(pathHndl)
handle pathHndl;

int acc_fetch_pulsere(pathHndl, r1, e1, r2, e2,
    r3, e3, r4, e4, r5, e5, r6, e6, r7, e7,
    r8, e8, r9, e9, r10, e10, r11, e11, r12, e12)
handle pathHndle;
double *r1, *e1, *r2, *e2, *r3, *e3, *r4, *e4,
       *r5, *e5, *r6, *e6, *r7, *e7, *r8, *e8,
       *r9, *e9, *r10, *e10, *r11, *e11, *r12, *e12;
```

6.9.2 Purpose

The function `acc_fetch_edge()` returns the edge specifier of a module path input terminal as one of the predefined integer constant. These constants are `accNoedge`, `accEdge01`, `accEdgex1`, `accEdge10`, `accEdge1x`, `accEdge0x`, `accEdgex0`, `accPosedge` (defined as `accEdge01 | accEdgex1 | accEdge0x`) and `accNegedge` (defined as `accEdge10 | accEdgex0 | accEdge1x`).

The polarity of a path determines how a signal transition at its source propagates to its destination in the absence of logic simulation events. The function `acc_fetch_polarity()` returns the polarity of a path as predefined integer constants. These constants are `accPositive` (indicating a rise/fall at the source causes a rise/fall at the destination), `accNegative` (a rise/fall at the source causes a fall/rise at the destination) and `accUnknown` (a change in the source causes an unpredictable behavior at the destination).

The function `acc_fetch_pulsere()` obtains the *reject value* and *error value* of a pulse for a module path. A *pulse* is defined as two successive signal transitions on the same path occurring in a shorter period of time than the path delay. The reject and error values determine how a pulse propagates from the source to the destination. The following table summarizes the relationship. Note that, the error value is always greater than or equal to the reject value and none of them can be more than the path delay itself.

Table 18. Effect of reject and error values on pulse propagation

If the duration of the pulse is	The effect on the destination
less than the reject value	no effect; the pulse is suppressed.
greater than the reject value, but less than the error value	an X is propagated to the destination.
greater than the error value	the correct signal value is passed to the destination.

The number of actual reject and error value argument pairs to `acc_fetch_pulsere()` is same as the value of the configuration parameter `accPathDelayCount`. Reject and error values are set from a Verilog program using a special specparam `PATHPULSE$` as shown below. (VCS users must use `+pathpulse` plusarg during compilation while using `PATHPULSE$`). Also, note that most commercial simulators support only one pair of reject and error values to be set using `PATHPULSE$`. Accordingly, inside the PLI routine the configuration parameter `accPathDelayCount` must be set to "1" before calling `acc_fetch_pulsere()`.

6.9.3 Example

The following Verilog program shows how to set the reject and error values for a module path.

```
module top;
wire clock, in, out;

specify
   (posedge clock => (out +: in)) = (10, 8);
   specparam PATHPULSE$clock$out = ( 2, 3);
endspecify

initial begin
   #15 $modpath_param("top", "clock", "out");
   #15 $finish;
```

```
end

endmodule
```

In the above program, let us assume that the system call `$modpath_param` prints the reject and error values of the module path between clock and out in the module top. An implementation of `$modpath_param` is shown below.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{

   handle modHndl, modPathHndl;
   double reject_limit1, err_limit1;

   acc_initialize();
   acc_configure(accPathDelayCount, "1");

   modHndl = acc_handle_by_name(tf_strgetp(1), NULL);
   modPathHndl = acc_handle_modpath(modHndl,
                                    tf_strgetp(2),
                                    tf_strgetp(3));
   if (!acc_error_flag)
      io_printf(" Module Path name: %s",
                acc_fetch_fullname(modPathHndl));

   acc_fetch_pulsere(modPathHndl, &reject_limit1,
                     &err_limit1);
   io_printf("reject value=%lf, error value=%lf\n",
             reject_limit1, err_limit1);

   acc_close();
}
```

Remember that unless specified by the specparam `PATHPULSE$` (or from the command line plusargs *+pulse_r/n* and *+pulse_e/n* in VCS, where n is a number between 0 to 100 and indicates the fraction of the path delay), both the reject and error limits are set to the module path delay.

7. MODIFY ROUTINES

7.1 Delay values

7.1.1 Functions in this category

There are two functions in this category: `acc_append_delays()` and `acc_replace_delays()`. Depending on the value of the configuration parameter `accMinTypMaxDelays` and the nature of the input object, the functions take variable number of arguments.

When `accMinTypMaxDelays` is set to "false", the functions have three different versions for adding to or replacing the existing delays associated with different object types.

For primitives, ports, port bits and intermodule paths:

```
int acc_append_delays(objHndl, rise_delay,
    fall_delay, z_delay)
handle objHndl;
double *rise_delay, *fall_delay, *z_delay;

int acc_replace_delays(objHndl, rise_delay,
    fall_delay, z_delay)
handle objHndl;
double *rise_delay, *fall_delay, *z_delay;
```

For module paths:

```
int acc_append_delays(objHndl,
    d1, d2, d3, d4, d5, d6, d7, d8, d9, d10,
    d11, d12)
handle objHndl;
double *d1, *d2, *d3, *d4, *d5, *d6, *d7,
       *d8, *d9, *d10, *d11, *d12;

int acc_replace_delays(objHndl,
```

```
    d1, d2, d3, d4, d5, d6, d7, d8, d9, d10,
    d11, d12)
handle objHndl;
double *d1, *d2, *d3, *d4, *d5, *d6, *d7,
       *d8, *d9, *d10, *d11, *d12;
```

For timing checks:

```
int acc_append_delays(objHndl, limit)
handle objHndl;
double *limit;

int acc_replace_delays(objHndl, limit)
handle objHndl;
double *limit;
```

When `accMinTypMaxDelays` is set to "true", the function prototype takes the following form.

```
int acc_append_delays(objHndl, delay_ptr)
handle objHndl;
double *delay_ptr;

int acc_replace_delays(objHndl, delay_ptr)
handle objHndl;
double *delay_ptr;
```

7.1.2 Purpose

The above functions are used for adding to or replacing the existing delay of a design object. When the configuration parameter `accMinTypMaxDelays` is set to "false", the above functions modify the single delay values for a number of different objects. As shown in the prototype declarations above, in such a case, depending on the nature of the object, the functions can have three different forms.

When `objHndl` points to an object of type primitive, port, port bit or intermodule path, the second and third arguments of the functions contain the rise and fall delays of the object to be added/replaced respectively. If the object is not a 2-state primitive, a third argument changes the "to-Z" delay.

When the object specified by `objHndl` is of type module path, the functions require upto 12 different delay values. What these delay values

imply depends on the value of the attribute `accPathDelayCount` and is shown in Table 10.

When `objHndl` points to an object of type timing check, the `limit` contains the limit of timing check.

When the configuration parameter `accMinTypMaxDelays` is set to "true", both the functions modify the minimum, typical and maximum delay values for an object in an array of `double` as specified by the pointer `delay_ptr`. The actual number of elements in this array depends on the configuration parameter `accPathDelayCount` and has been explaind in Section 6.4.2 earlier.

Note that, `acc_append_delays()` and `acc_replace_delays()` do not affect reject/error percentages.

7.1.3 Example

We will take two examples in this section. Both of these show a very simple *delay back annotation* program. A delay back annotation program modifies the delay properties of an object in a design reading the delay values from an external database. The delay property being modified can be path delay, timing check, cell output net delay, interconnect net delay etc.

In our first example, the program has two functions `write_gate_delays()` and `read_gate_delays()` for modifying and reading the propagation delays associated with primitives. It also assumes that the configuration parameter `accMinTypMaxDelays` is set to "false" and there are 3 delay values (0 to 1, 1 to 0 and to-Z delays) associated with each of the primitives. The system call associated with `write_gate_delays()` assumes that the name of the file where all delay values are stored is passed as the argument to the call.

Here is the first version of our back annotation program.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int write_gate_delays();
int read_gate_delays();

int write_gate_delays()
{
```

```
    handle gateHndl;
    FILE   *infile;
    double rise, fall, toz;
    char   *full_path_name;

    acc_initialize();
    acc_configure(accPathDelayCount, "3");
       /* 3-delay model*/
    full_path_name = (char *)malloc(35);

    infile = fopen(tf_strgetp(1),"r");
       /* tf_strgetp(1) gives the name
          of the delay back annotation file*/
    while (fscanf(infile, "%s %lf %lf %lf",
           full_path_name, &rise, &fall, &toz) != EOF) {
       io_printf("Module: %s,rise=%lf,fall=%lf,toz=%lf\n",
          full_path_name, rise, fall, toz);
       gateHndl=acc_handle_by_name(full_path_name, NULL);
       acc_replace_delays(gateHndl, rise, fall, toz);
    }
    acc_close();
}

int read_gate_delays()
{
    handle modHndl, gateHndl;
    double rise, fall, toz;

    acc_initialize();
    acc_configure(accPathDelayCount, "3");
       /* 3 delay model */
    modHndl = acc_handle_tfarg(1);

    gateHndl = NULL;
       while (gateHndl=
              acc_next_primitive(modHndl,gateHndl)) {
       acc_fetch_delays(gateHndl, &rise, &fall, &toz);
       io_printf("gate:%s,rise=%lf,fall=%lf,toz=%lf\n",
                 acc_fetch_fullname(gateHndl),
                 rise, fall, toz);
```

```
    }
    acc_close();
}
```

The following example shows the typical contents of the delay back annotation file. Note that the timescale precision of the module/cell must be adjusted to reflect the timescale precision accurately.

```
top.u1 1.0 2.2 0.2
top.u2 1.0 2.2 0.3
top.u3 0.9 1.2 0.4
top.u4 0.1 0.2 0.2
```

The second example demonstrates a similar application as above but this time it assumes that the configuration parameter `accMinTypMaxDelays` is set to "true". Also, in this example, instead of cell propagation delay, we will modify the module input port (MIP) delay.

Just as in the previous one, in this example also there are two sets of functions – one each for modifying and reading the delays. A typical delay back annotation file in this case for a 3-delay model is shown below.

```
1.0 1.1 1.2 2.1 2.2 2.3 3.0 3.1 3.2
1.0 1.1 1.2 2.1 2.2 2.3 3.0 3.1 3.2
0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
```

The system call associated with `write_MTM_delay_checktf()` and `write_MTM_delay_calltf()` assumes that there are input parameters. The first parameter is the full path name of the module whose MIP delays will be modified and the second argument gives the name of the delay back annotation file.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

#define DELAY_MODEL 3

int write_MTM_delay_checktf(),
    write_MTM_delay_calltf();
```

```
int read_MTM_delay_checktf(),
    read_MTM_delay_calltf();

FILE *fp;

write_MTM_delay_checktf()
{
   if (tf_nump() != 2) {
      tf_error(" Arg#1 must be the module name");
      tf_error(" Arg#2 must be a file_name");
   }

   if ((fp = fopen(tf_strgetp(2), "r")) == NULL) {
      tf_error(" File %s can not be opened",
               tf_strgetp(2));
      tf_dostop();
   }
}

write_MTM_delay_calltf()
{
   handle portHndl, moduleHndl;
   char *delay_mode;
   double *delay_array;
   int i;

   acc_initialize();
   acc_configure(accMinTypMaxDelays, "true");
   acc_configure(accPathDelayCount, "3");

   moduleHndl = acc_handle_tfarg(1);

   delay_array =
      (double *)malloc(3*DELAY_MODEL*sizeof(double));
   portHndl = NULL;
   io_printf("\n");
   while (portHndl=acc_next_port(moduleHndl, portHndl)) {
      fscanf(fp, "%lf %lf %lf %lf %lf %lf %lf %lf %lf",
            delay_array, delay_array+1, delay_array+3,
            delay_array+4, delay_array+5, delay_array+6,
```

```
            delay_array+7, delay_array+8, delay_array+9);
      if (acc_fetch_direction(portHndl) == accInput) {
         acc_replace_delays(portHndl, delay_array);
         io_printf("$write_MTM_delay():\n");
         for (i=0; i<=8; i=i+3)
            io_printf("%lf\t%lf\t%lf\n",
                      delay_array[i], delay_array[i+1],
                      delay_array[i+2]);
      }
   }
   acc_close();
}

read_MTM_delay_checktf()
{
   if (tf_nump() != 1)
      tf_error(" Arg: the full path name of the module");
      tf_dostop();
}

read_MTM_delay_calltf()
{
   handle portHndl, moduleHndl;
   char *delay_mode;
   double *delay_array;
   int i;

   acc_initialize();
   acc_configure(accMinTypMaxDelays, "true");
   acc_configure(accPathDelayCount, "3");

   moduleHndl = acc_handle_tfarg(1);
   delay_array =
      (double *)malloc(3*DELAY_MODEL*sizeof(double));
   portHndl = NULL;

   io_printf("\n");
   while (portHndl = acc_next_port(moduleHndl, portHndl))
      if (acc_fetch_direction(portHndl) == accInput) {
         acc_fetch_delays(portHndl, delay_array);
```

```
            io_printf("$read_MTM_delay(): \n");
            for (i=0; i<=8; i=i+3)
               io_printf("%lf\t%lf\t%lf\n",
         delay_array[i], delay_array[i+1], delay_array[i+2]);
         }
      acc_close();
   }
```

7.2 Logic values

7.2.1 Functions in this category

```
int acc_set_value(objHndl, valuePtr, delayPtr)
handle objHndl;
p_setval_value valuePtr;
p_setval_delay delayPtr;
```

7.2.2 Purpose

`acc_set_value()` is used for setting a specific value to a variable. It can be used for

- setting and propagating a value of a register, variable or a sequential UDP.
- procedurally assign/deassign a register or a variable.
- forcing a register, variable or net.
- setting a return value of the system call by passing a handle to the system call as `objHndl`.

The value to be written is passed in a pointer `valuePtr` to structure type `s_setval_value`. For the definition and description of each field of this structure, refer to Section 6.3.2. The delay associated with the propagation of this value is passed to the function using the third argument `delayPtr` which is a pointer to structure type `s_setval_delay` as shown below.

```
typedef struct t_setval_delay {
   s_acc_time time;
   int model;
} s_setval_delay, *p_setval_delay;

typedef struct t_acc_time {
   int type;
```

```
    int low, high;
    double real;
} s_acc_time, *p_acc_time;
```

The `model` field of the structure `s_setval_delay` determines the nature of the delay/assignment and has the following pre-specified values.

Table 19. Values of `model`

Value	Delay/ assignment model	Applicable on	Effect on the object
accNoDelay	No delay	register, sequential UDP	The value gets assigned without any delay.
accInertialDelay	Inertial delay	register	The value gets assigned after the specified delay. All events scheduled on the register are cancelled.
accTransportDelay	Modified transport delay	register	The value gets assigned after the specified delay. All events scheduled on the register after this delay period are cancelled.
accPureTransportDelay	Pure transport delay	register	The value gets assigned after the specified delay. No event is cancelled.
accAssignFlag	Procedural assign	register	Same as Verilog procedural `assign` statement
accDeassignFlag	Procedural deassign	register	Same as Verilog procedural `deassign` statement
accForceFlag	Force assign	register, net	Same as Verilog `force` statement
accReleaseFlag	Release	register, net	Same as Verilog `release` statement

The `time` field of the structure `s_setval_delay` is another structure of type `s_acc_time` and indicates the value and the type of the delay. The `type` field of this structure determines the unit of time for the delay and can have the following values.

Table 20. Values of `type`

Value	Data type	Implication
accTime	64-bit integer	Time is scaled to the timescale of the parent module of the object. The

Value	Data type	Implication
		delay is stored in `high` and `low` fields of the structure.
accSimTime	64-bit integer	Time is scaled to the time units being used by the simulator. Once again, the delay is stored in `high` and `low` fields of the structure.
accRealTime	double	Time is scaled to the timescale of the parent module of the object. The delay is stored in `real` field of the structure.

7.2.3 Example

The following example shows the implementation of a system call `$set` which takes an object and a string value, both assumed to be 4bit wide. If the object is a register, it replaces the contents of the register with the value specified in the string after an inertial delay of 5 time units. On the other hand, if the object is a net, it immediately forces the net to have a value specified by the string.

The following code shows the calltf function of the `$set`.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{

   handle          objHndl;
   s_setval_value valueStruct;
   s_setval_delay delayStruct;
   s_acc_time      timeStruct;

   acc_initialize();

   /* Initializing valueStruct */
   valueStruct.format = accBinStrVal;
```

```
    valueStruct.value.str = (char *)malloc(4);
       /* Since the reg and the net are 4bit wide */
    strcpy(valueStruct.value.str, tf_strgetp(2));

    /* Initializing delayStruct */
    objHndl = acc_handle_tfarg(1);
    if (acc_fetch_type(objHndl) == accNet){
       delayStruct.model = accForceFlag;
    }
    else if (acc_fetch_type(objHndl) == accReg) {
       timeStruct.type = accTime;
       timeStruct.low = 5;
          /* 5 units of delay for reg assignment */
       timeStruct.high = 0;

       delayStruct.time = timeStruct;
       delayStruct.model = accInertialDelay;
    }
    acc_set_value(objHndl, &valueStruct, &delayStruct);

    acc_close();
}
```

An example of a Verilog program utilizing this call is shown below.

```
module top;

   reg [3:0] r1;
   wire [3:0]  w1;

   assign w1 = r1;

   initial begin
      r1 = 4'b1111;
      $monitor($time, " r1 = %b, w1 = %b", r1, w1);
      $set(r1, "1010");
      #10 $set(w1, "0101");
      #10 $finish;
   end
```

```
endmodule
```

It produces the following output:

```
                    0 r1 = 1111, w1 = 1111
                    5 r1 = 1010, w1 = 1010
                   10 r1 = 1010, w1 = 0101
$finish at simulation time                    20
```

7.3 Pulse handling values

7.3.1 Functions in this category

```
int acc_append_pulsere(pathHndl, r1, e1, r2, e2,
    r3, e3, r4, e4, r5, e5, r6, e6, r7, e7,
    r8, e8, r9, e9, r10, e10, r11, e11,
    r12, e12)
handle pathHndle;
double *r1, *e1, *r2, *e2, *r3, *e3, *r4, *e4,
       *r5, *e5, *r6, *e6, *r7, *e7, *r8, *e8,
       *r9, *e9, *r10, *e10, *r11, *e11,
       *r12, *e12;

int acc_replace_pulsere(pathHndl, r1, e1,
    r2, e2, r3, e3, r4, e4, r5, e5, r6, e6,
    r7, e7, r8, e8, r9, e9, r10, e10, r11, e11,
    r12, e12)
handle pathHndle;
double *r1, *e1, *r2, *e2, *r3, *e3, *r4, *e4,
       *r5, *e5, *r6, *e6, *r7, *e7, *r8, *e8,
       *r9, *e9, *r10, *e10, *r11, *e11,
       *r12, *e12;

void acc_set_pulsere(pathHndl,
   reject_percentage, error_percentage)
handle pathHndl;
double reject_percentage, error_percentage;
```

7.3.2 Purpose

All functions in this category are used for modifying the reject and error values of a pulse. (For definitions of pulse and these values, refer to Section 6.9.2). As mentioned earlier, the error value is always greater than or equal to the reject value and none of them can be more than the path delay itself.

For the functions `acc_append_pulsere()` and `acc_replace_pulsere()`, the number of actual reject value and error value argument pairs is same as the value of the configuration parameter `accPathDelayCount` and can go upto 12 depending on how many different transition delays are being modified.

`acc_set_pulsere()` sets the reject and error values as a percentage of the path delay.

7.3.3 Example

The following function reads the reject and error limits of a module path and if they are less than some arbitrary values, it replaces them by a new set of values. The function assumes that the module name and the terminals of the module path are passed as parameters to the system call. It also assumes that a 2-value delay is used in the original module.

```
int test_re()
{
   handle modHndl, modPathHndl, dataPathHndl;
   double rej_limit1, err_limit1;
   double rej_limit2, err_limit2;

   acc_initialize();
   acc_configure(accPathDelayCount, "2");

   modHndl = acc_handle_by_name(tf_strgetp(1), NULL);

   modPathHndl = acc_handle_modpath(modHndl,
                                    tf_strgetp(2),
                                    tf_strgetp(3));
   if (!acc_error_flag)
      io_printf(" Module Path name: %s\n",
                acc_fetch_fullname(modPathHndl));
   acc_fetch_pulsere(modPathHndl,
```

```
                    &rej_limit1, &err_limit1,
                    &rej_limit2, &err_limit2);
io_printf("Initial values:\n");
io_printf(" rej1=%lf, err1=%lf, rej2=%lf, err2=%lf\n",
          rej_limit1, err_limit1,
          rej_limit2, err_limit2);

if ((rej_limit1 >= 2.0) ||
    (err_limit1 >= 3.0) ||
    (rej_limit2 >= 4.0) ||
    (err_limit2 >= 5.0)) {
   acc_replace_pulsere(modPathHndl,
                       2.0, 3.0, 4.0, 5.0);
   io_printf("Setting:\n");
   io_printf(" rej1=2, err1=3, rej2=4, err2=5\n");
}

acc_fetch_pulsere(modPathHndl,
                  &rej_limit1, &err_limit1,
                  &rej_limit2, &err_limit2);
io_printf("Final values:\n");
io_printf(" rej1=%lf, err1=%lf, rej2=%lf, err2=%lf\n",
          rej_limit1, err_limit1,
          rej_limit2, err_limit2);

acc_close();
}
```

Remember that unless specified by the specparam PATHPULSE$ (or from the command line plusargs *+pulse_r/n* and *+pulse_e/n* in VCS where n is a number between 0 to 100 and indicates the fraction of the path delay), both the reject and error limits are set to the module path delay.

8. VALUE CHANGE LINK (VCL) ROUTINES

Value change link routines are used for calling a user-defined function, whenever a given object changes its value (logic or strength). Conceptually this mechanism is similar to the callback used by tf_ routines and, in fact, has its own set of *reasons* (Refer to Chapter 2 for the definition of reason).

Also, the user can pass his own *data* to the called function in a way similar to the implementation of callback mechanism using tf_ routines.

8.1 Functions in this category

```
void acc_vcl_add(objHndl, consumerFunc,
                 user_data, vcl_flag)
handle objHndl;
int (*consumerFunc)();
char *data;
int vcl_flag;

void acc_vcl_delete(objHndl, consumerFunc,
                    user_data, vcl_flag)
handle objHndl;
int (*consumerFunc)();
char *data;
int vcl_flag;
```

8.2 Purpose

The function `acc_vcl_add()` adds a value change monitor on the object specified by `objHndl` such that the user-defined function `consumerFunc` will be called, whenever there is a change in the logic value (if `vcl_flag` is `vcl_verilog_logic`) or strength value (if `vcl_flag` is `vcl_verilog_strength`). Different kinds of information regarding the value change are gathered in a structure pointer `vc_record` of type `s_vc_record`. The user function must pass this structure as its argument. The definition of this structure has been shown below.

```
typedef struct t_vc_record {
   int     vc_reason;
   int     vc_hightime;
   int     vc_lowtime;
   char    *user_data;
   union   {
      unsigned char logic_value;
      double        real_value;
      handle        vector_handle;
      s_strengths   strengths_s;
```

```
        } out_value;
} s_vc_record, *p_vc_record;
```

The structure `s_strengths` used above in `out_value` is defined as follows:

```
typedef struct t_strengths {
   unsigned char logic_value;
   unsigned char strength1;
   unsigned char strength2;
} s_strengths;
```

In `vc_record`, the {`vc_hightime`, `vc_lowtime`} denotes the 64-bit time when the change occured. The `user_data` field contains the data supplied by the user when `acc_vcl_add()` was called. The `vc_reason` field describes the type of change in terms of a set of predefined values. Depending on the value of vc_reason, only a single field of the union `out_value` is assigned to the current value of the object. The following table shows various possible values of `vc_reason`, their meanings, the corresponding field of `out_value` to be assigned and possible values of this field of `out_value`. Note that, if a value change link is added to a module port, the `vc_reason` is based on the loconn of the port.

Table 21. Predefined values of different fields of `vc_record`

When vc_reason is	The object type which has changed its value	The nature of the change	The valid field of out_value
logic_value_change	scalar net, bit-select	logic value	logic_value
strength_value_change	scalar net, bit-select	logic or strength value	strengths_s
vector_value_change	vector net or its part-select	logic value	vector_handle
sregister_value_change	scalar register	logic value	logic_value
vregister_value_change	vector register or its part select	logic value	logic_value
integer_value_change	integer	value	vector_handle
real_value_change	real	value	real_value
time_value_change	time	value	vector_handle
event_value_change	event	–	–

For `logic_value_change` and `sregister_value_change`, the logic_value field of `out_value` should be one of the following predefined integer constants: `vcl0`, `vcl1`, `vclx`, `vclX`, `vclz` and `vclZ`.

A value change link on a particular object can be associated with as many user-defined functions as required, by calling `acc_vcl_add()` multiple times.

The function `acc_vcl_delete()` removes the value change monitoring on an object specified by `objHndl`. The value change monitoring must be active on this object when calling `acc_vcl_delete()`. Since an object can have a number of different user-defined functions associated with its value change, `acc_vcl_delete()` requires the function whose callback will be disabled as well as the corresponding user data passed to `acc_vcl_add()` earlier as input arguments.

8.3 Example

Many a time in simulation, the circuit under the test (CUT) consists of a few newly designed modules along with several already verified logic modules. If you are a board designer, this may particularly sound familiar to you. In a board, most of the glue logic supplied by the vendor can be thought of as bugfree (well, in most of the cases!) leaving the primary focus of verification on the ASICs designed for this board.

The system call `$peripheri` in this example takes the full pathname of a sub-module. Once called, whenever there is a change in value of any of the ports (input, output or inout) of the given module, the system call lists the values of all the ports. This will provide the designer with important insights into the functionality of the module when integrated in a system.

A typical example of using `$peripheri` is shown below to monitor the ports of the module `environment.u1`.

```
module environment;
   top u1 (op1, ip1, ip2, ip3, ip4);
   testbench u2 (ip1, ip2, ip3, ip4);
   clkgen u3 (clk);

   always @(posedge clk)
      $peripheri("environment.u1");
endmodule

module top(op1, ip1, ip2, ip3, ip4);
output op1;
input  ip1, ip2, ip3, ip4;
```

```
    and u1 (ip5, ip1, ip2);
    and u2 (ip6, ip3, ip4);
    nor u3 (op1, ip5, ip6);
endmodule

module testbench (ip1, ip2, ip3, ip4);
output ip1, ip2, ip3, ip4;
reg    ip1, ip2, ip3, ip4;

    initial begin
    #13    {ip1, ip2, ip3, ip4} = 4'b0000;
    #10    {ip1, ip2, ip3, ip4} = 4'b0101;
    #10    {ip1, ip2, ip3, ip4} = 4'b1100;
    #10    {ip1, ip2, ip3, ip4} = 4'b1101;
    #10    {ip1, ip2, ip3, ip4} = 4'b0110;
    #7     {ip1, ip2, ip3, ip4} = 4'b1110;
    $finish;
    end
endmodule

module clkgen(clk);
output clk;
reg clk;

    initial
       clk = 1'b0;

    always
    #5 clk = ~clk;
endmodule
```

In this example, as is apparent, we want to link a C function which will print out the values of the nets connected to the ports of the module. Also, this single function should be linked to value changes of *all* such nets.

In order to accomplish this we first retrieve the handles to the ports of the given module. Next, handles are obtained to the net connected to each port. Once the handle to a net is obtained, the C function `print_peripheri()` is linked using `acc_vcl_add()`.

Note that, the function `print_peripheri()` is called whenever there is a value change on *any* net. This will create an interesting problem when two or

more nets change values at the same simulation time. For example, in the Verilog program given earlier, at time 13 both `ip1` and `ip4` change values simultaneously. However, `$peripheri()` should not print the net values twice for value changes on `ip1` and `ip4`. It should only list the final net values.

In order to print only the final values of the nets, the function `print_peripheri()` should not print the current net values immediately. Instead it should store them in a string. Each time `print_peripheri()` is called, if the simulation time is same as the previous call, this string is overwritten with the new net values. When the function `print_peripheri()` is called at a future simulation time, this string is printed, as it will contain the final net values at the previous simulation time.

The only exception to the above algorithm is to print the result when the simulation is over (as there will not be any future simulation time). This is done using a callback with the reason `reason_finish` and passing a `NULL` parameter to the function.

The following program shows the implementation of `$peripheri`. In stead of printing to the standard output, the routine prints the net values to a file called `"peripheri.log"`.

```
#define LOG_LENGTH 100
/* The length of the string to
   accommodate all net values */
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myChecktf(), myCalltf(), myMisctf();
int print_peripheri();
handle modHndl;

int myChecktf()
{
   /* Check the number and types of the parameter */
   if (tf_nump() != 1)
      tf_error(" The task should have 1 parameter only");
   if (tf_typep(1) != tf_string)
      tf_error(" The parameter is not a string");
}
```

```
int myCalltf()
{
   handle portHndl;
   handle netHndl;

   acc_initialize();

   modHndl = acc_handle_by_name(tf_strgetp(1), NULL);
   portHndl = NULL;
   while (portHndl = acc_next_port(modHndl, portHndl)) {
      /* Note: ports are assumed to be scalar */
      netHndl =  acc_handle_loconn(portHndl);
      acc_vcl_add(netHndl, print_peripheri, 0,
                  vcl_verilog_logic);
   }
   acc_close();
}

int myMisctf(data, reason)
int data, reason;
{
   if (reason == reason_finish)
      print_peripheri(NULL);
}

/* This function prints all the port values */
int print_peripheri(vc_record)
p_vc_record vc_record;
{
   handle portHndl;
   handle netHndl;
   static int time = 0, prev_time = 0;
   static int first_time = 0;
   static char *log_line;
   static FILE *fp;

   if (first_time == 0) {
      log_line = (char *)malloc(LOG_LENGTH);
      fp =fopen("peripheri.log","w");
      first_time = 1;
```

```
    }

    if (((time = tf_gettime()) != prev_time) ||
        ((vc_record == NULL) && (first_time == 1))){
       fprintf(fp, "%s\n", log_line);
       prev_time = time;
    }

    strcpy(log_line, tf_strgettime());
    strcat(log_line, ": ");

    portHndl = NULL;
    while (portHndl = acc_next_port(modHndl, portHndl)) {
       /* Note: ports are assumed to be scalar */
       netHndl =  acc_handle_loconn(portHndl);
       strcat(log_line, acc_fetch_value(netHndl, "%b"));
       strcat(log_line, "  ");
    }
}
```

9. NEXT ROUTINES

As mentioned in Section 4, often we need to find out all the objects of a given type which are related to a particular object in a design hierarchy. For example, a typical PLI application may find out the values of all the registers within a given module. Since the number of such related objects is unknown to the user (e.g., number of registers in this case), the PLI mechanism must provide a facility which will search out all such objects and provide them to the user. *Next* routines have been designed to do this task. Each next routine takes at least two arguments: a handle to the *reference object*, with respect to which the related objects are found out and an uninitialized handle which will point to the next related object, also called *target object*, upon return. To retrieve all target objects, usually the next routine is put inside a loop. For the first iteration of this loop, the handle to the target object is passed as NULL. On each successive iteration, the current target handle is replaced by the next target handle. The loop terminates when all target objects are retrieved and a NULL handle is returned as the next target. The following pseudo-code segment shows the mechanism using an imaginary next function acc_next_example().

```
void foo(ref_handle)
handle ref_handle;
/* A handle to the reference object should be
   obtained before using the next functions */
{
   handle target_handle = NULL;
   /* target_handle should be initialized
      to NULL before using it for the first time */
   while (target_handle = acc_next_example(ref_handle,
                                           target_handle)) {
   /* The main body of the program goes here*/
   }
}
```

The next routines can be divided into the following categories according to their functionalities.

9.1 Objects related to a module

9.1.1 Functions in this category

```
handle acc_next_cell(modHndl, cellHndl)
handle modHndl, cellHndl;

handle acc_next_child(modHndl, childHndl)
handle modHndl, childHndl;

handle acc_next_modpath(modHndl, modpathHndl)
handle modHndl, modpathHndl;

handle acc_next_net(modHndl, netHndl)
handle modHndl, netHndl;

handle acc_next_parameter(modHndl, paramHndl)
handle modHndl, paramHndl;

handle acc_next_port(mod_or_netHndl, portHndl)
handle mod_or_netHndl, portHndl;
```

```
handle acc_next_portout(modHndl, portHndl)
handle modHndl, portHndl;

handle acc_next_primitive(modHndl, primHndl)
handle modHndl, primHndl;

handle acc_next_specparam(modHndl, s_paramHndl)
handle modHndl, s_paramHndl;

handle acc_next_tchk(modHndl, tchkHndl)
handle modHndl, tchkHndl;

handle acc_next_terminal(primHndl, terminalHndl)
handle primHndl, terminalHndl;
```

9.1.2 Purpose

For all functions in this category, the reference object is a module and the target object is contained within the module.

`acc_next_cell()` returns handles to cell instances in hierarchy below the module specified by `modHndl`. (A *cell* in Verilog is defined as a module whose definition appears either between `` `celldefine `` and `` `endcelldefine `` compiler directives or in a file declared as a library during simulation.)

`acc_next_child()` returns handles to all children module within a reference module specified by `modHndl`. If `modHndl` is `NULL`, the function returns handles to all top level modules.

The function `acc_next_modpath()` gets handles to the module paths defined within a module specified by `modHndl`. (A *module path* is defined as a specify block path for delay as mentioned at the left hand side of the delay assignment.)

`acc_next_net()` returns handles to nets in a module specified by `modHndl`. Note that, handles are returned for each net, not each bit of a net, if it is a vector. (As we will see later, once a handle to a vector (a net or otherwise) is available, handle to each of its bits can be obtained using `acc_next_bit()`.)

`acc_next_parameter()` returns handles to parameters in a module specified by `modHndl`.

If `mod_or_netHndl` specifies a module, `acc_next_port()` returns handles to ports of that module. If it specifies a net, the function returns

handles to ports connected to the net within its scope. The handles to the ports are returned in the order they are specified in the port list of the module.

`acc_next_portout()` returns handles to output and inout ports of a module. (One wonders why this function is part of an IEEE standard, when it is merely a combination of `acc_next_port()` and `acc_fetch_direction()`!)

`acc_next_primitive()` returns handles to all primitives within a module specified by `modHndl`. (A *primitive* is defined as a gate, switch or UDP.)

The function `acc_next_specparam()` returns handles to the specparams in a module specified by `modHndl`.

`acc_next_tchk()` get handles to timing checks within a module specified by `modHndl`.

`acc_next_terminal()` returns the terminals of a primitive.

9.1.3 Example

In this example, we will use the functions `acc_next_child()` and `acc_next_portout()` to list the outport ports of all hierarchically lower modules below a given module. The function `myCalltf()` assumes that the given module is passed to the system call as the parameter. It invokes the function `find_child_outport()` with a handle to the given module as the argument. `find_child_outport()` calls itself recursively unless all modules are traversed. The function also lists the output ports of the current module.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
void find_child_outport();

int myCalltf()
{
   handle modHndl;

   acc_initialize();
```

```
    modHndl = acc_handle_tfarg(1);
    find_child_outport(modHndl);

    acc_close();
}

void find_child_outport(modHndl)
handle modHndl;
{
    handle childModHndl, portHndl;

    io_printf("\nModule name: %s\n",
              acc_fetch_fullname(modHndl));

    portHndl = NULL;
    while (portHndl = acc_next_portout(modHndl, portHndl))
        io_printf("\tPort:%s\n",
                  acc_fetch_fullname(portHndl));

    childModHndl = NULL;
    while (childModHndl =
           acc_next_child(modHndl, childModHndl))
        find_child_outport(childModHndl);
}
```

9.2 Objects related to the design hierarchy

9.2.1 Functions in this category

```
handle acc_next_scope(scopeHndl,
                      internal_scopeHndl)
handle scopeHndl, internal_scopeHndl;

handle acc_next(objTypeArray, scopeHndl,
                objHndl)
static int *objTypeArray;
handle scopeHndl, objHndl;

handle acc_next_topmod(top_modHndl)
handle top_modHndl;
```

9.2.2 Purpose

`acc_next_scope()` returns handles to the internal scope within a given scope specified by `scopeHndl`.

`acc_next()` is a generic function for returning handles to different object types as listed in `objTypeArray` within a given scope specified by `scopeHndl`. `objTypeArray` is an array of static integers containing types or fulltypes of objects to be scanned. The last element of this array must be a 0. Notice that unlike the rest of the next functions, `acc_next()` can cover more than one type of target object. The following table shows a list of types and fulltypes supported by `acc_next()` and their corresponding predefind integer constants.

Table 22. Types and fulltypes supported by `acc_next()`

Type	Integer constant	Fulltypes, if any	Integer constant
Integer variable	accIntegerVar		
Module	accModule	Top-level module	accTopModule
		Module instance	accModuleInstance
		Cell module instance	accCellInstance
Named event	accNamedEvent		
Net	accNet	Wire nets	accWire, accTri
		Wired-AND nets	accWand, accTriand
		Wired-OR nets	accWor, accTrior
		Pulldown nets	accTri0
		Pullup nets	accTri1
		Supply nets	accSupply0, accSupply1
		Storage nets	accTriReg
Primitive	accPrimitive	Logic gates	accAndGate, accNandGate, accOrGate, accNorGate, accXorGate, accXnorGate, accBufGate, accNotGate, accBufif0, accNotif0, accBufif1, accNotif1
		Transistor	accPmosGate, accNmosGate, accRpmosGate, accRnmosGate, accCmosGate, accRcmosGate
		Transmission gate	accRtranGate, accRtranif0Gate, accRtranif1Gate, accTranGate, accTranif0Gate, accTranif1Gate
		Pulldown, pullup gates	accPulldownGate, accPullUpGate
		UDP	accCombPrim, accSeqPrim

Type	Integer constant	Fulltypes, if any	Integer constant
Real variable	accRealVar		
Register	accRegister		
Time variable	accTimeVar		

The function `acc_next_topmod()` finds out all the top level modules in a design. An important point to note here is that this function does not work with `acc_collect()` (to be discussed in a short while) or `acc_count()`. The function `acc_next_child()` (to be discussed later in this chapter) with `NULL` reference object should be used for this purpose.

9.2.3 Example

The function `print_topmod_vars()` in this example prints the integer, time and real variables in all the top modules in a design description. It also prints the scopes within each of these top modules. Note that, for each of the top modules, we store a handle to one of the variables within it. This handle is used later to access the scopes within that top module.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int  print_topmod_vars();
int  print_topmod_vars()
{

   handle top_modHndl = NULL;
   static int rarelyUsedVars[] =
      {accIntegerVar, accTimeVar, accRealVar, 0};
   handle rarelyUsedHndl;
   handle topScopeHndl, childScopeHndl;
   handle validHndl;

   acc_initialize();
   while (top_modHndl = acc_next_topmod(top_modHndl)) {
      /* get a handle to one of the top modules */
      io_printf("Top-level module : %s\n",
         acc_fetch_name(top_modHndl));

      io_printf("List of int, time and real vars:\n");
      rarelyUsedHndl = NULL;
```

```
        while ( rarelyUsedHndl =
                    acc_next(rarelyUsedVars,
                             top_modHndl,
                             validHndl = rarelyUsedHndl)) {
           io_printf("\t%s: %s\n",
                     acc_fetch_name(rarelyUsedHndl),
                     acc_fetch_value(rarelyUsedHndl,
                                     "%h", NULL));
        }

        topScopeHndl = acc_handle_scope(validHndl);
        /* set the scope to that of the top module */
        io_printf("List of scopes within this topmod:\n");
        childScopeHndl = NULL;
        while ( childScopeHndl =
                acc_next_scope(topScopeHndl,
                               childScopeHndl))
           io_printf("\t%s\n",
                     acc_fetch_name(childScopeHndl));
     }

     acc_close();
  }
```

9.3 Objects related to driving or loading of a net

9.3.1 Functions in this category

```
handle acc_next_cell_load(netHndl, terminalHndl)
handle netHndl, terminalHndl;

handle acc_next_load(netHndl, terminalHndl)
handle netHndl, terminalHndl;

handle acc_next_driver(netHndl, terminalHndl)
handle netHndl, terminalHndl;

handle acc_next_hiconn(portHndl, netHndl)
handle portHndl, netHndl;
```

```
handle acc_next_loconn(portHndl, nethndl)
handle portHndl, netHndl;
```

9.3.2 Purpose

Both `acc_next_load()` and `acc_next_cell_load()` return input terminals of primitives which load a net specified by `netHndl`. While `acc_next_load()` returns handles to all input primitive terminals driven by the net, `acc_next_cell_load()` returns only one primitive terminal per cell or module instance which loads the net.

`acc_next_driver()` returns all the output and inout primitive terminals that drive a net.

The functions `acc_next_hiconn()` and `acc_next_loconn()` return the hierarchically higher and lower net connection to a port respectively.

9.3.3 Example

The following example lists the drivers of a net and the primitives loading it. It is assumed that the driver of the net is a structural construct (fanout of a gate, UDP or switch). The net is passed to the system call as the parameter.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int print_driver_load();
int print_driver_load()
{
   handle netHndl, terminalHndl = NULL, cellHndl;

   acc_initialize();

   netHndl = acc_handle_tfarg(1);
   io_printf(" The primitives loading the net %s:\n",
      acc_fetch_fullname(netHndl));

   while (terminalHndl =
         acc_next_load(netHndl, terminalHndl)) {
      cellHndl = acc_handle_parent(terminalHndl);
```

```
        io_printf("   %s\n", acc_fetch_fullname(cellHndl));
    }

    io_printf(" The drivers of this net:\n");

    terminalHndl = NULL;
    while (terminalHndl =
            acc_next_driver(netHndl, terminalHndl)) {
        cellHndl = acc_handle_parent(terminalHndl);
        io_printf("   %s\n", acc_fetch_fullname(cellHndl));
    }

    acc_close();
}
```

9.4 Objects related to a vector

9.4.1 Function in this category

```
handle acc_next_bit(refHndl, bitHndl)
handle refHndl, bitHndl;
```

9.4.2 Purpose

The function `acc_next_bit()` returns handles to bits of a vector, such as a vectored port, path terminals or net. The individual bit of a vectored net can only be accessed if the property `accExpandedVector` is true for that net. The bits are accessed starting from the most significant bit of the vector to the least significant bit.

9.4.3 Example

The following example prints the values of the loconn nets connected to the input and output ports of a module. The module is passed to the system call as the first parameter. The function uses the access routine `acc_object_of_type()` to determine whether a vector net is expandable. If it is, then the values of the individual bits are printed, otherwise, the vectored value is printed. We will discuss about `acc_object_of_type()` in a short while.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int myCalltf();
int myCalltf()
{

   handle modHndl, portHndl;
   handle loconnHndl, bitHndl;
   int msb, i;

   acc_initialize();

   modHndl = acc_handle_tfarg(1);

   portHndl = NULL;
   while (portHndl = acc_next_port(modHndl, portHndl)) {
      io_printf("Loconn net connected to the port %s:\n",
                acc_fetch_fullname(portHndl));
      loconnHndl = NULL;
      while (loconnHndl =
             acc_next_loconn(portHndl, loconnHndl)) {
         if (acc_fetch_size(loconnHndl) == 1) {
         /* if the net is a scalar */
            io_printf(" %s:%s\n",
                      acc_fetch_fullname(loconnHndl),
            acc_fetch_value(loconnHndl, "%b", NULL));
         } else { /* if the net is a vector */
            if (acc_object_of_type(loconnHndl,
                                   accExpandedVector)) {
               /* if the net is an expanded vector */
               bitHndl = NULL;
               acc_fetch_range(loconnHndl, &msb, &i);
               /* We do not need to know the lsb */
               i = msb;
               while (bitHndl =
                      acc_next_bit(loconnHndl, bitHndl))
                  io_printf(" %s[%d]:%s\n",
                            acc_fetch_fullname(loconnHndl),
```

```
                                i--,
                                acc_fetch_value(bitHndl,"%b",
                                                NULL));
                } else {
                /* if the net is a vector but
                   not expanded vector */
                io_printf(" %s:%s\n",
                          acc_fetch_fullname(loconnHndl),
                          acc_fetch_value(loconnHndl, "%b",
                                          NULL));
                }
            }
        }
    }

    acc_close();
}
```

9.5 Objects related to a module path

9.5.1 Functions in this category

```
handle acc_next_input(modpathHndl,terminalHndl)
handle modpathHndl, terminalHndl;

handle acc_next_output(modpathHndl,terminalHndl)
handle modpathHndl, terminalHndl;

void acc_release_object(objHndl)
handle objHndl;
```

9.5.2 Purpose

The functions `acc_next_input()` and `acc_next_output()` return handles to the input and output path terminals of a module path respectively. They can also be used to return handles to source or destination terminals of a data path. Additionally, `acc_next_input()` can return handles to the timing check terminals of a timing check.

`acc_release_object()` frees up memory allocated by the functions `acc_next_input()` and `acc_next_output()`. Note that, this function

should be called only when all required terminals associated with a module path have been retrieved.

9.5.3 Example

The following example prints input and output terminals of all the module path within a module. The function assumes that the module is passed to the system call as the first argument.

```
#include <stdio.h>
#include "veriuser.h"
#include "acc_user.h"

int print_modpath_terminal();
int print_modpath_terminal()
{

   handle modHndl, modPathHndl;
   handle inptTerminalHndl, outptTerminalHndl;

   acc_initialize();

   modHndl = acc_handle_tfarg(1);

   modPathHndl = NULL;
   while (modPathHndl =
          acc_next_modpath(modHndl, modPathHndl)) {

      io_printf(" The input to the module path %s:\n",
                acc_fetch_fullname(modPathHndl));
      inptTerminalHndl = NULL;
      while (inptTerminalHndl =
            acc_next_input(modPathHndl,inptTerminalHndl))
         io_printf(" %s\n",
                   acc_fetch_fullname(inptTerminalHndl));

      io_printf(" The output to the module path %s:\n",
                acc_fetch_fullname(modPathHndl));
      outptTerminalHndl = NULL;
      while (outptTerminalHndl =
```

```
            acc_next_output(modPathHndl, outptTerminalHndl))
            io_printf(" %s\n",
                    acc_fetch_fullname(outptTerminalHndl));
    }

    acc_release_object(modPathHndl);
    acc_close();
}
```

10. MISCELLANEOUS ROUTINES

Miscellaneous routines are designed to perform a number of different housekeeping jobs. These include initializing certain environmental variables at the beginning of the program and freeing the memory allocated to them at the end, comparing two objects for their equality, setting a scope etc. Miscellaneous routines are typically used to perform the 'remainder' tasks at the end of each iteration in a loop.

According to their functionalities, miscellaneous functions can be divided into following categories.

10.1 Managing the acc routine environment

10.1.1 Functions in this category

```
void acc_initialize()

void acc_close()

int acc_configure(configParam, configVal)
int configParam;
char *configVal;

void acc_reset_buffer()

int acc_product_type()

char *acc_product_version()

char *acc_version()
```

```
char *acc_set_scope(modHndl, modName)
handle modHndl;
char *modName;

int acc_set_interactive_scope(scopeHndl)
handle scopeHndl;
```

10.1.2 Purpose

`acc_initialize()` is used at the beginning of any program which uses other access routines later on. It initializes all configuration parameters to their default values and also allocates the memory for different internal variables used by the access routines.

`acc_close()` frees up these memories at the end of the program. It also resets the value of the configuration parameters to their factory defaults.

The function `acc_configure()` is used for setting the value of 10 predefined configuration parameters. The configuration parameter, whose value is being changed, is passed to the function as a pre-defined integer constants. The final value of the parameter is passed as a string. The following table summarizes the configuration parameters, their valid values (with the default value shown in bold), which access routines they affect and how. Note that, a single configuration parameter may affect a number of different functions.

Table 23. Configuration parameters and their effect

Configuration parameter	Valid values/ Default value	Affected access routine
accDefaultAttr0	"true", **"false"**	acc_fetch_attribute(). When "true", for a missing attribute, a 0 is returned as its value. When "false", the function returns the default value for a missing object.
accDevelopmentVersion	Any string, no default	No function is affected. The config parameter can be used to determine the version of access routine used in the development of the PLI routine.
accDisplayErrors	**"true"**, "false	When "true", access routines display error messages. Otherwise, they do not.
accDisplayWarnings	"true",	When "true", access routines

Configuration parameter	Valid values/ Default value	Affected access routine
	"false"	display warning messages. Otherwise, they do not.
accEnableArgs	"acc_handle_modpath", **"no_acc_handle_modpath"**	acc_handle_modpath(). When the config parameter is set to "acc_handle_modpath", the function recognizes its optional arguments; otherwise, they are ignored.
	"acc_handle_tchk", **"no_acc_handle_tchk"**	acc_handle_tchk(). When the config. parameter is set to "acc_handle_tchk", the function recognizes its optional arguments; otherwise, they are ignored.
	"acc_set_scope", **"no_acc_set_scope"**	acc_set_scope(). When the config. parameter is set to "acc_set_scope", the function recognizes its optional arguments; otherwise, they are ignored.
accMapToMipd	**"max"**, "min", "latest"	acc_replace_delays(). Depending on the value of the config. parameter, the longest/shortest/last intermodule path delay will be mapped to the MIPD.
accMinTypMaxDelays	"true", **"false"**	acc_append_delays(), acc_fetch_delays(), acc_replace_delays(), acc_append_pulsere(), acc_fetch_pulsere() and acc_replace_pulsere(). When the config. parameter is "true", these functions use a min:typ:max delay sets. Otherwise, a single delay value is used.
accPathDelayCount	"1", "2", "3", "4", "5", **"6"**	acc_append_delays(), acc_fetch_delays(), acc_replace_delays(), acc_append_pulsere(), acc_fetch_pulsere() and acc_replace_pulsere(). Depending on the value of the config. parameter, the functions

Configuration parameter	Valid values/ Default value	Affected access routine
		use 1, 2, 3, 4, 5 or 6 delay value(s) or value set(s).
accPathDelimStr	Any string, numbers, **$**, _	acc_fetch_name(), acc_fetch_fullname() and acc_fetch_attribute(). The config. parameter is used as the delimeter for the module path names.
accToHiZDelay	"avarage", "max", "min", **"from_user"**	acc_append_delays() and acc_replace_delays(). Depending on the value of the config. parameter, the "to-Z" delay is derived by these functions from the avarage/larger/smaller of the rise and fall delays or from the user-supplied argument(s).

`acc_configure()` returns 1 if successful and 0 if an error occurred.

`acc_reset_buffer()` frees up memories allocated internally to string buffers by different access routines.

`acc_product_type()` returns the type of product that is using the PLI routine. Although primarily used for simulation, PLI routines can also be used by other types of product. The product type is returned as one of the predefined integers: `accSimulator`, `accTimingAnalyzer`, `accFaultSimulator` and `accOther`.

The function `acc_product_version()` returns a string with a description of the vendor of the product and its version .

Similarly, `acc_version()` returns a string showing the version of the access routines.

`acc_set_scope()` is used in association with the function `acc_handle_object()` to set a scope to define the search space in the design hierarchy. It sets the scope to the level of the module specified by `modHndl`. Optionally, it also takes a module name as a string depending on how the configuration parameter `accEnableArgs` has been set using `acc_configure()`. Note that, the default value of `accEnableArgs` is "`no_acc_set_scope`". The following table shows how different arguments are used by the function.

Table 24. How the arguments of `acc_set_scope()` work

If accEnableArgs is set to	And modHndl is	acc_set_scope()
"no_acc_set_scope"	valid	Uses modHndl and ignores modName

If accEnableArgs is set to	And modHndl is	acc_set_scope()
"no_acc_set_scope"	NULL	Sets the scope to the level of the first top level module appeared in the design hierarchy
"acc_set_scope"	valid	Uses modHndl and ignores modName
"acc_set_scope"	NULL	Sets scope to the level of the module modName, if it is not NULL. Otherwise, the scope is set to the level of the first top level module appeared in the design hierarchy

The return value of the function is a string containing the full hierarchical name of the scope set.

The function `acc_set_interactive_scope()` sets the interactive scope of the simulator to the scope specified by `scopeHndl`.

10.1.3 Example

The following code snippet shows the use of the functions in this category. Apart from `acc_initialize()` and `acc_close()` to be used at the beginning and at the end of the program (or part of the program which uses access routines) respectively, there is no other restriction on using these functions as long as they are used judiciously.

```
:
acc_initialize();
acc_configure(accPathDelayCount, "3");
acc_configure(accEnableArgs, "acc_set_scope");
:
switch(acc_product_type()) {
case accSimulator:
   io_printf("This simulator uses"); break;
case accTimingAnalyzer:
   io_printf("This timing analyzer uses"); break;
case accFaultSimulator:
   io_printf("This fault simulator uses"); break;
case accOther:
   io_printf("This unknown product type uses"); break;
default:
   tf_error(" Product type not found\n");
}
io_printf("Acc routine ver.:%s in its current ver: %s\n",
   acc_version(), acc_product_version());
```

```
acc_set_scope(NULL, "top.u1");
:
acc_reset_buffer();
:
acc_close();
```

10.2 Accessing all target objects simultaneously

10.2.1 Functions in this category

```
int acc_count(accNextRoutine, objHndl)
handle (*accNextRoutine)();
handle objHndl;

handle *acc_collect(accNextRoutine,
                    objHndl, objNum)
handle (*accNextRoutine)();
handle objHndl;
int    *objNum;

void acc_free(handlePtr)
handle *handlePtr;
```

10.2.2 Purpose

The functions in this category are used to manage target objects related to a particular reference object (For definitions of reference and target objects, see Next routines).

The function `acc_count()` returns the number of target objects related to a particular reference object as specified by `objHndl`. The relation between the target objects and the reference object is determined by the acc_next function `accNextRoutine` used for finding out the targets. (The only exception of acc_next function which does not work with `acc_count()` is `acc_next_topmod()`. In this case, `acc_next_child()` can be used with a `NULL` reference object handle.)

`acc_collect()` returns an a array of (more precisely, a pointer to) handle data type. This array contains handles to all target objects related to a reference object specified by `objHndl` and the nature of the relationship is described by the function `accNextRoutine`. The number of target objects is specified by `objNum`, which can be obtained by using `acc_count()` prior to

using this function. Note that, `acc_collect()` should only be used when all target handles are required to be processed together. The total number of target objects can be pretty large in a real-life design and consequently the memory requirement to store the returned array may adversely affect the performance of the simulation.

`acc_free()` frees up the memory allocated and used by `acc_collect()` to store the array of handles.

10.2.3 Example

The following example shows a function `collect_net()` which takes a module handle as the parameter and collects handles to all nets within that module. It then prints the net names.

```
void collect_net(moduleHndl)
handle moduleHndl;
{
   int no_of_nets;
   handle *netarr;

   no_of_nets = acc_count(acc_next_net, moduleHndl);
      /* Get the total no of nets in the ref module */
   netarr = acc_collect(acc_next_net,
                        moduleHndl,
                        &no_of_nets);

   for (i=0; i<no_of_nets; i++)
      io_printf(" Net %d: %s\n",
                i, acc_fetch_name(netarr[i]));

   acc_free(netarr);
}
```

10.3 Comparing and matching objects

10.3.1 Functions in this category

```
int acc_compare_handles(handle1, handle2)
handle handle1, handle2;
```

```
int acc_object_in_typelist(objHndl,
                           objTypeArray)
handle objHndl;
static int *objTypeArray;

int acc_object_of_type(objHndl, objType)
handle objHndl;
int objType;
```

10.3.2 Purpose

The function `acc_compare_handles()` compares two handles `handle1` and `handle2` and returns `TRUE` if they specify the same object. Otherwise, it returns `FALSE`.

`acc_object_in_typelist()` finds out whether an object specified by its handle `objHndl` belongs to one of the types or fulltypes mentioned in the array `objTypeArray`. If a match is found, the function returns `TRUE`. Otherwise a `FALSE` is returned. The array `objTypeArray` can contain any number of predefined type and fulltype integer constants but the last element of the array must be a 0. As shown in the following table, in addition to types and fulltypes integer constants, `acc_object_in_typelist()` also supports a set of other special properties as element of the array `objTypeArray`.

The function `acc_object_of_type()` determines whether an object specified by the handle `objHndl` belongs to a particular type, fulltype or special property. Listed in the following table, these special properties are same as those supported by `acc_object_in_typelist()`.

Table 25. Special properties supported by `acc_object_in_typelist()` and `acc_object_of_type()`

Property	Predefined integer constant	Property	Predefined integer constant
Scalar	accScalar	Vector	accVector
Collapsed net	accCollapsedNet	Expanded net	accExpandedNet
Unexpapnded net	accUnexpandedNet	Hierarchy scope	accScope
Module path with *ifnone* condition	accModPathHasIfNone.		

10.3.3 Example

In this example, the function print_vectored_wor() prints all wired-or vectored nets in a module. The module is specified by its handle which is passed to the function as a paramemter.

```
void print_vectored_wor(modHndl)
handle modHndl;
{
handle netHndl;
static int vector = {accVector, 0},
           wor_net = {accWor, 0};

netHndl = NULL;
while(netHndl=acc_next_net(modHndl, netHndl))
   if ((acc_object_in_typelist(netHndl,vector)) &&
       (acc_object_in_typelist(netHndl, wor_net)))
      io_printf("Net name: %s\n",
                acc_fetch_name(netHndl));
}
```

11. USING ACCESS ROUTINES WITH VCS

As discussed in Chapter 2, VCS translates the Verilog source files to files in C before compiling them into object files. The compiled code is optimized trimming off any unnecessary information so that the executable binary runs faster. However, a PLI routine can have access to any object in the design database if it uses access routines, which means the compilation process must not remove any information regarding design objects.

To satisfy these contradictory objectives, VCS divides the `acc_` capabilities into several categories. You need to specify the categories used in your PLI routine in the pli.tab file. This information is preserved during the compilation.

The syntax for specifying the `acc_` capabilities is as follows.

$name [pli_spec] [acc_spec]

where $name is the name of the system call and pli_spec specifies the names of the functions as described in Section 5.2 of Chapter 2.

For discussing acc_spec, let us see an example and then we will describe each of its component. Here is an example of the acc_spec.

acc+=rw:top_module+

As shown in the example above, acc_spec always start with the keyword *acc*. The operator += means "add these capabilities to the specified scope". The following table lists all valid operators.

Table 26. Operators for acc_ capabilities

Operator	Used for
= +=	Add these capabilities to the specified scope.
-=	Remove these capabilities from the specified scope.
:=	Set exactly these capabilities to the specified scope.

The term *rw* means that the read_write capabilities of the access routines have been enabled. The next table summarizes the list of acc_ capabilities.

Table 27. List of acc_ capabilities

Capability	Abbreviation	Enabled operation
read	r	Reading of nets, registers and variables
read_write	rw	Reading as above and writing of registers and other writable variables
callback	cbk	Callbacks on named object value changes
callback_all	cbka	Callbacks on unnamed objects such as primitive terminal
force	frc	Forcing of nets and registers
timing_check_backannotation	tchk	Backannotation of timing checks
gate_backannotation	gate	Backannotation of gate delays
module_path_backannotation	mp	Backannotation of module path
module_input_port_backannotation	mip	Backannotation of module input ports
module_input_port_bit_annotation	mipb	Backannotation of individual bits of module input ports

The : between *rw* and *top_module* is the delimiter separating the capabilities and the scope.

top_module is the scope where the capabilities are getting added. The + sign indicates that the capabilities would be applied to all its children as well. The scope can also be one of the following wildcards.

Table 28. Wildcard scopes

Wildcard	Applied scope
%CELL	All cell definitions
%TASK	The module(s) where the PLI routine is invoked.
*	All modules used in the design.

12. SUMMARY

In this chapter we have learnt different classifications of access routines in PLI1.0 and how to use them. To keep the functions and their uses simple, a dedicated function has been created for each possible reason. But this has increased the total number of functions to a point where it is next to impossible to remember the syntax of all of them. As we will find later, this issue will be addressed in PLI2.0 or VPI and total number of functions will be drastically reduced.

As you might have noticed, in many cases the operation of the tf and acc routines overlap. In such cases, we have the option of choosing either of them. However, the final choice, apart from the user's familiarity with the functions, also depends on several other factors. For example, many compilers need special arrangement for using access routines and generated code may be less efficient. As a matter of fact, many simulators specify the comparative efficiency of the two libraries in their datasheet.

So far, we have used each of the functions atomically, i.e., microscopic applications were written only to demonstrate the use of each function. In Chapters 5 and 6, we are going to use the library functions that we have learnt in the last two chapters to develop some real life applications.

Chapter 5

PUTTING PLI TO WORK

The tf and access routines discussed in the last two chapters are the basic building blocks for writing useful user applications in PLI1.0. In this chapter, we are going to create a number of such applications. Our primary goal will be to examine how to approach a solution, rather than covering all possible corner cases of the problems. These applications are chosen from different aspects of hardware designs and a few from software domains. Also, the applications have been chosen such that, with a few modifications, each of them can be transformed into many other similar applications. Some of them are left as exercise to the reader in the *Self check* sections.

As discussed in Chapter 2, for registering the component functions of a PLI routine, Verilog-XL uses the structure `veriusertfs[]`, while VCS uses a separate file to list these functions. For sake of brevity, in all examples in this chapter, we will skip the registration part of a PLI routine concentrating on the component functions only. For identifying these functions, all examples follow a common naming convention – they start with a prefix `my` followed by their generic name; for example, the calltf function is called `myCalltf` etc.

Also, to avoid the confusion regarding the names of the header files from different vendors, we will assume that `veriuser.h` is the basic PLI header file and `acc_user.h` contains information related to the access routines.

S. Mittra, *Principles of Verilog PLI*
© Kluwer Academic Publishers 1999

1. PROBLEM 1: GETTING INFORMATION FROM THE ENVIRONMENT

In this example, we will create a system call `$env`, which will perform the following tasks:

- It will list the current directory where the simulation is being run.
- Additionally, the system call will also take a variable number of string arguments. If there is an environment variable defined in the operating system shell which matches one of these strings, it will print the value of the corresponding environment variable.

A typical example of using this call from a Verilog program is shown below.

```
module test;
   initial begin
      $env("DISPLAY","USER","MY_BIN");
   #100;
      $finish;
   end
endmodule
```

1.1 Approaching a solution

Far from the traditional applications in hardware domains, this example shows how diverse the uses of PLI can be.

Furthermore, it is a good example to show that a single problem can be solved in many different ways. For example, there are at least two possible ways of solving the current problem and as we will see soon, we are going to use a combination of both of them.

The first part of the program lists the current directory. This is equivalent of running the UNIX command `ls`. One way of achieving this will be to use the function `system()` defined in `stdlib.h`, which takes *any* UNIX (or the active OS) command as a string argument and executes it under a new shell. As we will see, in some cases, this is the only possible approach to interact with the system. A set of similar problems is given in the *Self check* part later.

However, in certain cases (though not all), it is possible to use standard C library functions to access information about the system. This usually is a much better way to handle the system resources. For example, in order to print the values of the passed arguments, when there is a matched

environment variable, we will use the C function `getenv()` defined in the header file `stdlib.h`. The function `getenv()` takes a string as an argument and prints its value if there is a matched environment variable. (An alternate, but more wasteful way of doing this would be to call `system()` with an `echo` command. On the other hand, the previous example of directory listing could have been done by the C library function `readdir()`, defined in `dirent.h` in SVR4 and in `sys/dir.h` in BSD UNIX, which returns the entries in a directory).

1.2 Program listing

The following listing shows the implementation of `$env`.

```
#include <stdio.h>
#include <stdlib.h>
#include "veriuser.h"

int myCalltf(), myChecktf();

int myChecktf() {
int pin;

   if (tf_nump() >= 1)
      for (pin=1; pin <= tf_nump(); pin++)
         if (tf_typep(pin) != tf_string)
            tf_error("$env: Argument type mismatch\n");
}

int myCalltf() {
int pin;
   /* Listing Directories */
   io_printf("Your directory listing :\n");
   system("ls -l");

   /* Listing some environment variable */
   if (tf_nump() > 1)
      for (pin=1; pin <= tf_nump(); pin++)
         if (getenv(tf_strgetp(pin)))
            io_printf(" The value of %s: %s\n",
                      tf_strgetp(pin),
```

```
                    getenv(tf_strgetp(pin)));
          else
            io_printf(" The env variable %s not found\n",
                      tf_strgetp(pin));
}
```

1.3 Results

After compilation, a typical output of `$env` with the above test case will look like as shown below.

```
Your directory listing :
total 3914
-rwxr-xr-x    1 mittra   1617    Jul  9 10:56 cr_vlog
drwxrwxr-x    2 mittra   512     Jul 16 01:54 csrc
-rw-rw-r--    1 mittra   35      Jul 16 01:51 pli.tab
-rw-r--r--    1 mittra   1421    Jul  9 10:56 sample.c
-rw-r--r--    1 mittra   2052    Jul  9 10:56 sample.o
-rw-r--r--    1 mittra   1087    Jul 16 01:54 simple.c
-rwxrwxr-x    1 mittra   1992076 Jul 16 01:55 simv
-rw-r--r--    1 mittra   101     Jul 16 01:50 test.v
 The value of DISPLAY: calcutta:0
 The value of USER: mittra
 The env variable MY_BIN not found
$finish at simulation time                  100
```

1.4 Self check

– Create a PLI routine for a system call `$convert2html` which will convert the current Verilog log file to a HTML page. All error messages in the log file should be displayed in a separate color than the normal text.

2. PROBLEM 2: CREATING A TEST BENCH

This example shows how to create a test bench using PLI1.0 for a logic module. The logic module to be tested (also known as *circuit under test* or CUT) is assumed to be synchronous changing state only at the positive edge of the input clock.

We start with a Verilog program which illustrates how to use a system call to implement a test bench. The top level module `environment` contains the CUT along with other components in the system, such as the clock generator etc. The system call `$testbench` is called from `environment` at the positive edge of the clock signal `clk` using an `always` block.

```
module environment;
reg ip1, ip2, ip3;

   clock_gen u2 (clk);

   mymodule u1 (op1, op2, op3, ip1, ip2, ip3, clk);

   // mymodule is supposed to be posedge operated
   always @(posedge clk)
      $testbench(op1, op2, op3, ip1, ip2, ip3,
                 "test.bench" );

   initial
   #100 $finish;

endmodule

module clock_gen(clk);
output clk;
reg clk;

   initial
      clk = 1'b0;

   always
   #5 clk = ~clk;
endmodule

module mymodule (op1, op2, op3, ip1, ip2, ip3, clk);
output op1, op2, op3;
input ip1, ip2, ip3;
input clk;

   /* user-defined logic */
```

```
    assign #1 op1 = ip1 | ip2 ;
    assign #1 op2 = ip1 & ip2 ;
    assign #1 op3 = ip3 ^ ip1 ;
endmodule
```

`$testbench` opens a given file `test.bench` containing the test vectors. Each test vector consists of a set of input values to be applied to the CUT and the expected outputs at a particular time instance. Each time `$testbench` is called, it reads a test vector from the file, applies the input values to the CUT, reads its outputs and compares them with the expected values read from the file. If the actual output does not match the expected value, the system call generates an error message.

Here is an example of the file `test.bench` containing the test vectors.

```
3 3
x x x 0 0 0
0 0 0 0 0 1
0 0 1 0 1 0
1 0 0 0 1 1
1 0 1 1 0 0
1 0 1 1 0 1
1 0 0 1 1 0
1 1 1 1 1 1
z 1 0 0 0 0
```

The first line of the file indicates that the CUT has 3 outputs and 3 inputs. From the next line onward, the expected outputs and inputs are listed with a space as the delimiter.

2.1 Approaching a solution

When called for the first time, `$testbench` should open the file containing the test vectors. The name of this file as well as all outputs and inputs of the CUT are passed as parameters to this system call. Remember, since the number of arguments in a system call can be variable, this allows `$testbench` to work with any number of inputs or outputs. Using a set of hard coded full path names for inputs/outputs of a specific CUT inside the PLI routine would have prohibited us from having such flexibility.

Once the file containing the test vectors is opened, at each clock the inputs and the expected outputs are read from the file.

2.2 Program listing

The following program shows the implementation of `$testbench`. We will assume that there is no inout port for the CUT. This will simplify the problem of deciding the direction of an inout port at a given clock edge.

After reading the input and expected output values from the file, they are stored in separate arrays. These arrays are defined as pointers to arrays of characters with maximum number of entry permitted for each of them as `MAXPORT`. The routine continues to read one test vector per clock until it reaches the end of the file. The inputs are applied to the CUT using `tf_strdelputp()` and the outputs are read using `tf_strgetp()`. The expected output is compared to the actual value using `strcmp()` function.

```
#include <stdio.h>
#include "veriuser.h"

int myChecktf(), myCalltf();
FILE *fp;
int num_out, num_in;

int myChecktf()
{
   static int first_time = 0;

   /* Open the file only for the first time */
   if (first_time == 0) {
      /* The last param is the test vector filename */
      if ((fp=fopen(tf_strgetp(tf_nump()), "r")) == NULL)
         io_printf("%s can not be opened\n",
                   tf_strgetp(tf_nump()));

      /* read the number of input and output ports */
      fscanf(fp, "%d %d", &num_out, &num_in);

      first_time = 1;
   }

   /* Check the number and types of the parameters
      It should be no of ports plus 1 for the filename */
   if (tf_nump() != num_out+num_in+1)
```

```
        tf_error(" The task should have %d parameters",
                 num_out+num_in+1);

    /* A check for type of the parameters
       may be included here */
}

int myCalltf()
{
#define MAXPORT 1000

    char *out[MAXPORT], *inp[MAXPORT];
    int i;
    int file_status;

    /* if file_status == EOF, the program stops */
    file_status = 0;

    /* Allocate memory for all the variables
       and read the expected values and the
       input vectors from the file */

    for (i=1; (i<=num_out) && (file_status != EOF); i++) {
       out[i-1] = (char *)malloc(tf_sizep(i)+1);
       file_status = fscanf(fp, "%s", out[i-1]);
    }

    for (i=1; (i<=num_in) && (file_status != EOF); i++) {
       inp[i-1] = (char *)malloc(tf_sizep(i+num_out)+1);
       file_status = fscanf(fp, "%s", inp[i-1]);
    }

    if (file_status != EOF) {
       /* Apply the inputs, assumed to
          be in binary format*/
       for (i=1; i<=num_in; i++)
           tf_strdelputp(i+num_out,
                         tf_sizep(i+num_out),
                         'b',
                         inp[i-1],
```

```
                                0,
                                0);

        /* read the outputs and compare
           with the expected values*/
        for (i=1; i<=num_out; i++)
           if (strcmp(out[i-1], tf_strgetp(i, 'b')) != 0) {
              tf_error("time=%s: Error!", tf_strgettime());
              io_printf(" output[%d]:expected:%s, got:%s",
                        i, out[i-1], tf_strgetp(i, 'b'));
           }
        free(out);
        free(inp);
        }
     else
        fclose(fp);
  }
```

2.3 Results

The result of running the Verilog program of Section 2.1 with `test.bench` is as follows.

```
Error: time=85: Error!
       From $testbench at time 85 in file test.v line 10:
 output[1]:expected :z, got:1
$finish at simulation time                    100
```

The error message is generated because the expected value z of `op1` (the first entry in the last line of `test.bench`) does not match its actual value 1 (the bitwise-OR of `ip1` and `ip2`, the 4th and 5th entries of the previous line).

2.4 Self check

– Extend the scope of the present program to handle inout ports. Assume that the number of inout ports is specified in the first line of `test.bench` following the number of inputs. The direction of an inout port is determined by a control signal specified in the same line of the file. An inout port acts as an output only if the value of this control

signal at the time of applying the test vector is logic-1. For example, if the first line of `test.bench` looks as follows:

```
1 2 1 "environment.u1.ctrl"
```

it indicates that the CUT has 1 output, 2 inputs and 1 inout ports. The inout port acts as an output if the signal `environment.u1.ctrl` is logic-1. The program should flag an error if there are fewer control signals than the number of inout ports.

3. PROBLEM 3: A SIMPLE VALUE CHANGE DUMP TOOL

Although most of the complex simulation environments are self-checking in nature, visualization is still an important aspect of debugging. Usually once a test fails, the simulation is run again and the behaviors of the related signals are recorded for a time window surrounding the point of failure. This process of recording signal values is usually referred to as *dumping* and the file, which stores the signal value changes is known as *Value Change Dump (VCD) file* or simply, *dump file*. Later on, these signals are displayed with the help of a waveform viewer for the actual debugging.

In this example, we are going to create two system calls. The first of them is `$dumpon_nets(my_module)`, which starts dumping all nets inside a module `my_module`. The dumping will continue, until the other system call `$dumpoff_nets(my_module)` is called. For example, the following Verilog program starts dumping the nets inside the module `top.u1` at time=100 and continues upto time=200. (If you are familiar with the Verilog's built-in system calls for dumping, our system calls are primitive versions of `$dumpon` and `$dumpoff`).

```
module dump;

initial begin
   #100 dumpon_nets(top.u1);
   #100 dumpoff_nets(top.u1);
end
endmodule
```

3.1 Approaching a solution

Verilog HDL follows a particular format for dumping the signal values into the VCD file. This format is quite simple in nature and moreover, we will use only a subset of it for our purpose.

First, let us see how a VCD file looks like. It is a text (ASCII) file which has two main parts. The first part lists the scopes of the variables to be dumped along with their properties (name, width etc.). Each variable is assigned a 'shorthand' notation to be used later in the file. This part also contains the listing of initial values of the variables to be dumped.

An example of the first part of a VCD file is shown below.

```
$timescale  1 ns $end
$scope module top $end
$scope module u1 $end
$var tri 2 !! clkarray $end
$var tri 1 @! clock $end
$upscope
$upscope
$enddefinitions $end
#100
$dumpvars
b01 !!
1@!
$end
```

The `$timescale` statement determines the timescale of the simulation. It is followed by two consecutive `$scope` statements that set the current scope of dump to `top.u1` (note that this `$scope` has no relation to the built-in system call `$scope`). There are two variables inside the scope `top.u1`: `clkarray` and `clock`. The properties of each of these nets are described inside a `$var` statement. For instance, the `$var` statement for `clkarray` says that

- the net is of type *tri*
- the width of the net is *2*
- the shorthand notation for this net to be used in the rest of the file is *!!* and
- the name of the net is *clkarray*

The `$upscope` statement brings the current scope of the variables one level up.

The #100 after this specifies the time to start the dump. This is followed by the `$dumpvars-$end` statement, which encloses the initial values of the variables. Each value is printed in a separate line. However, the listing of the values is done differently for a scalar and a vector. For a scalar (such as `clock` in our example), the value of the variable is immediately followed by its shorthand notation with no space in between. For a vector (such as `clkarray`), the line starts with the format of the data ('b' for binary, 'h' for hex etc.). It is immediately followed by the value and then a space is put between the value and the shorthand notation of the vector.

The second part of the VCD file lists the time and changed value of a variable. Here is an excerpt of the second part.

```
#100
0@!
b10 !!
#105
1@!
b01 !!
#110
0@!
b10 !!
```

The above segment shows that at time #100, the scalar variable whose notation is `@!` has changed its value to 0 and the vector with notation `!!` has a new value of binary 10. The interpretation is done similarly for time #105 and #110 etc. Remember that the time is specified in absolute units. A timing diagram for *clock* and *clkarray* is shown below.

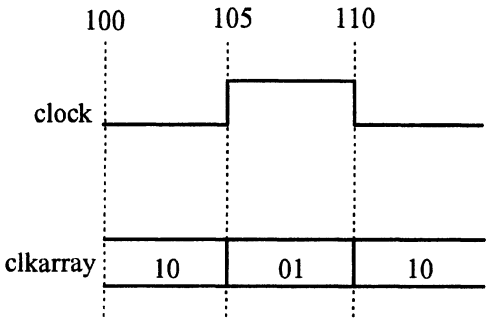

Figure 6. Timing diagram for *clock* and *clkarray*

In order to print the net values when they change, we first need to have handles to the nets. This is done by getting a handle to the module passed to the system call as a parameter and then using it to obtain handles to all nets inside the module. Each net is assigned a shorthand name and they are stored in an array `id_code` as shown below. For `$dumpon_net()`, all nets are added to the value change link list using the function `acc_vcl_add()`, so that if any net changes value, the function `dump_net()` is invoked. The initial values of the net at the time of invoking `$dumpon_net()` is first written into a temporary file `verilog.dump.aux` and then copied into the main VCD file `verilog.dump`.

The system call `$dumpoff_net` works in a similar way. The function `acc_vcl_delete()` is used to disable the value change link. Note that, in order to disable the value change link for a particular net, apart from the handle to the net, we must pass the exact same function and user data to `acc_vcl_delete()` which were used in `acc_vcl_add()` earlier.

The program assumes that there are no more than 100 nets within the module.

3.2 Program listing

```
#include <stdio.h>

#include "veriuser.h"
#include "acc_user.h"

int myChecktf();
int dumpon_Calltf();
int dumpoff_Calltf();
int dump_net();
int ten_to_the_power();

char *id_code[100];
   /* id_code is made global so that both dumpon_Calltf()
      and dumpoff_Calltf() have access to it */
FILE *fp;
   /* fp is made global so that both dumpon_Calltf()
      and dump_net() have access to it */

int myChecktf()
{
```

```
    /* Check the number of parameters */
    if (tf_nump() > 1)
       tf_error(" USAGE:my_dump(module);");
}

int dumpon_Calltf()
{
    handle modHndl, netHndl;

    char temp_id_code[3], *fullname, *inst, unit_s[3];
    char char_arr[] =
    {'!', '@', '#', '$', '%', '^', '&', '*', '(', ')'};

    int  i=0, net_width, level=0, c, unit_d;

    FILE *fp_aux;

    s_acc_value value;
    s_timescale_info timescale_info;

    /* Initialization part */
    acc_initialize();

    fp = fopen("verilog.dump", "w");
    fp_aux = fopen("verilog.dump.aux", "w");
    value.format = accBinStrVal;
    temp_id_code[2] = '\0';
    fullname = (char *)malloc(100);
    inst = (char *)malloc(20);

    modHndl = acc_handle_tfarg(1);

    /* print the current time in verilog.dump.aux */
    fprintf(fp_aux, "#%d\n$dumpvars\n", tf_gettime());

    /* print the timescale in the dump file */
    acc_fetch_timescale_info(modHndl, &timescale_info);
    if (timescale_info.unit >= 0) {
       strcpy(unit_s, "s");
       unit_d = ten_to_the_power(timescale_info.unit);
```

```
      printf("unit_d = %d\n", unit_d);
   } else if (timescale_info.unit >= -3) {
      strcpy(unit_s, "ms");
      unit_d = ten_to_the_power(timescale_info.unit+3);
   } else if (timescale_info.unit >= -6) {
      strcpy(unit_s, "us");
      unit_d = ten_to_the_power(timescale_info.unit+6);
   } else if (timescale_info.unit >= -9) {
      strcpy(unit_s, "ns");
      unit_d = ten_to_the_power(timescale_info.unit+9);
   } else if (timescale_info.unit >= -12) {
      strcpy(unit_s, "ps");
      unit_d = ten_to_the_power(timescale_info.unit+12);
   } else {
      strcpy(unit_s, "fs");
      unit_d = ten_to_the_power(timescale_info.unit+15);
   }

   fprintf(fp, "$timescale\n\t%d %s\n$end\n",
           unit_d, unit_s);

   /* print the scope in the dump file */
   strcpy(fullname, acc_fetch_fullname(modHndl));
   do {
      inst = strtok(fullname, ".");
      if (inst) {
         fprintf(fp, "$scope module %s $end\n",  inst);
         level++;
      }
      fullname = (char *)NULL;
   } while (inst);

   netHndl = NULL;
   while (netHndl = acc_next_net(modHndl, netHndl)) {
      /* first write the net description
         to the  logfile */
      fprintf(fp,
              "$var tri %d %c%c %s $end\n",
              acc_fetch_size(netHndl),
              char_arr[i/10],
```

```
            char_arr[i],
            acc_fetch_name(netHndl));

    id_code[i] = (char *)malloc(3);
    temp_id_code[1] = char_arr[i];
    temp_id_code[0] = char_arr[i/10];
    strcpy(id_code[i], temp_id_code);

    acc_vcl_add(netHndl, dump_net,
                id_code[i], vcl_verilog_logic);

    /* Write the initial values to verilog.dump.aux */
    /* First determine whether the net is a scalar
       or a vector */

    if (acc_fetch_size(netHndl) == 1)
       /* if the net is scalar */
       fprintf(fp_aux, "%s%s\n",
               acc_fetch_value(netHndl, "%b", &value),
               id_code[i]);
    else /* if the net is a vector */
       fprintf(fp_aux, "b%s %s\n",
               acc_fetch_value(netHndl, "%b", &value),
               id_code[i]);
    i++;
}

/* go level-1 up in the scope */
for (i=1; i<=level;i++)
   fprintf(fp, "$upscope\n");
fprintf(fp, "$enddefinitions $end\n");

fprintf(fp_aux, "$end\n");
fclose(fp_aux);

/* Now print the initial values of the nets */
/* from fp_aux to the actual dumpfile       */
fp_aux = fopen("verilog.dump.aux", "r");
while ((c = fgetc(fp_aux)) != EOF) fputc(c, fp);
```

```
    /* Close down */
    unlink("verilog.dump.aux");
    acc_close();
}

/* This function prints all the nets value */
/* to the dump file                        */

int dump_net(vc_record)
p_vc_record vc_record;
{
    static int time = 0, prev_time = 0;
    s_acc_value value;

    if ((time = vc_record->vc_lowtime) != prev_time){
        fprintf(fp, "#%d\n", vc_record->vc_lowtime);
        prev_time = time;
    }

    if (vc_record->vc_reason == vector_value_change) {
        /* If the net is a vector */
        fprintf(fp, "b%s %s\n",
        acc_fetch_value(vc_record-out_value.vector_handle,
                        "%b", &value),
        vc_record->user_data);
    } else if (vc_record->vc_reason==logic_value_change) {
        /* If the net is a scalar */
        fprintf(fp, "%d%s\n",
                vc_record->out_value.logic_value,
                vc_record->user_data);
    }
}

int ten_to_the_power(expo)
int expo;
{
     int prod = 1;

     if (expo) {
        while (expo--) prod = prod*10;
```

```
        return prod;
    }
    else
        return(1);
}

int dumpoff_Calltf()
{
    handle modHndl, netHndl;
    int i=0;

    modHndl = acc_handle_tfarg(1);
    netHndl = NULL;
    while (netHndl = acc_next_net(modHndl, netHndl))
        acc_vcl_delete(netHndl, dump_net,
                       id_code[i++], vcl_verilog);
}
```

3.3 Self Check

- The above program only dumps the net values within the given module. Modify the above program to dump *all* variables within a given module.
- Modify the program so that it takes an additional parameter `n` and dumps the net values for the given module as well as all submodules which are within `n` level below it in the hierarchy.

4. PROBLEM 4: A COVERAGE TOOL FOR GATE LEVEL NETLIST

This example shows how a set of seemingly unrelated problems can be solved using a common program structure. The system call `$toggle` in this example takes the full pathname of a module as a string and computes the following:

- The total power consumed by the module for the duration when `$toggle` is enabled.
- The percentage of all nets which toggled during this period as a result of the test vectors applied at the inputs of the module.

Furthermore, the same program, with minor modifications, can be used to implement a design rule checker to find out any violation in the number of fan-outs for each gate within the given module.

We will make the following assumptions to focus our main attention to the core problem.

- The module passed to the system call is a flat structural netlist containing only the primitives (gates and switches) interconnected by scalar nets.
- The given module is synchronous to the positive edge of the input clock.
- The propagation delay for all primitive gates is less than the clock period (which has to be true for any valid design). This will allow us to invoke the system call at every positive edge of the clock and check for any toggled net within the same clock period.

A typical example of using `$toggle` from a Verilog program is shown below. Here `environment` is the top-level module which contains a clock generator `clkgen`, a testbench `testbench` and the main circuit under the test `comparator`. The toggle coverage is checked at the positive edge of the clock `clk` generated from the module `clkgen`.

```
module environment;

   comparator u1 (op1, ip1, ip2, ip3, ip4);
   testbench u2 (ip1, ip2, ip3, ip4);
   clkgen u3 (clk);

   always @(posedge clk)
      $toggle("environment.u1");

endmodule

module comparator(op1, ip1, ip2, ip3, ip4);
/* A simple 2-bit comparator
   op1 = 1'b1, if {ip1,ip2} < {ip3,ip4}
   op1 = 1'b0, otherwise                 */
output op1;
input  ip1, ip2, ip3, ip4;
wire w5, w6, w7, w8;

   not u1 (ip1_, ip1);
   not u2 (ip2_, ip2);
```

```
   xor u3 (w5, ip3, ip4);
   and u4 (w6, ip2_, ip1_, w5);
   and u5 (w7, ip1_, ip2, ip3);
   and u6 (w8, ip1, ip2_, ip3, ip4);
   or  u7 (op1, w6, w7, w8);
endmodule

module testbench (ip1, ip2, ip3, ip4);
output ip1, ip2, ip3, ip4;
reg    ip1, ip2, ip3, ip4;

   initial begin
   #13 {ip1, ip2, ip3, ip4} = 4'b0000;
   #10 {ip1, ip2, ip3, ip4} = 4'b0101;
   #10 {ip1, ip2, ip3, ip4} = 4'b1100;
   #10 {ip1, ip2, ip3, ip4} = 4'b1101;
   #10 {ip1, ip2, ip3, ip4} = 4'b0110;
   #7  $finish;
   end
endmodule

module clkgen(clk);
output clk;
reg clk;

   initial
      clk = 1'b0;

   always
   #5 clk = ~clk;
endmodule
```

4.1 Approaching a solution

In order to find out the toggle coverage of the module, you must keep track of whether each net in the module has toggled during the entire length of the simulation. This is done by comparing the current value of the net to its previous value at all positive edges of the clock. If the two values are different, a flag is set corresponding to that net.

On the other hand, for measuring the power, assuming a CMOS-type circuitry, we need to know whenever a particular net toggles, how much power is consumed by its fan-outs for changing the states. When `$toggle` is invoked for the first time, it calculates the power consumption of each net by adding up the power consumed by all the gates driven by the net. Once again, for simplicity, we will assume power consumption for all transitions (0 to 1, 0 to z or 1 to 0 etc.) are same for a particular type of gate. The power consumed by each type of gate is read from a file `power.h`. Most of the modern synthesis tools provide a similar library for popular foundries. A typical `power.h` will look like as follows.

```
#define POWER_AND  2
#define POWER_OR   2
#define POWER_XOR  2
#define POWER_NAND 1
#define POWER_NOR  1
#define POWER_XNOR 1
```

In the main program, a structure `net_info` stores the information regarding a net. The definition of the structure `net_info` is shown below.

```
struct net_info {
   handle netHndl;        /* handle to the net */
   char *prev_value;      /* value of the net at the
                             previous clock edge */
   int  consumed_power;   /* power consumed when this
                             net toggles */
   int toggled;           /* has this net toggled ? */
}
```

A handle to a net inside the given module can be obtained by getting a handle to the module first and then using `acc_next_net()` function in a loop. In this example, since we need to access all the nets at every clock edge, to make the program run faster, the functions `acc_count()` and `acc_collect()` are used to collect the handles to all the nets at the beginning and then storing them as part of `net_info` structure for each net.

From the second clock edge onwards, the current values of the nets are compared to their values at the previous clock edge. If both of these values are valid (either 1 or 0) and different from each other, the `toggled` field of `net_info` is set for the corresponding net.

If a net is toggled, power consumed by its fan-outs is added to the total power consumed by the entire module so far.

At the end of the simulation (or when the current instance of `$toggle` is disabled, the program prints out the total power consumed by all the gates and also finds out the percentage of nets toggled.

4.2 Program listing

The program listing for the system call `$toggle` is shown below.

```
#define MAX_NET 100

#define "power.h"
#include <stdio.h>
#include "vcsuser.h"
#include "acc_user.h"

int myChecktf(), myCalltf(), myMisctf();
int is_toggled();

int myChecktf()
{
   acc_initialize();
   /* Check the number and types of the parameter */
   if (tf_nump() != 1)
      tf_error(" Expected #parameter: 1, found=%d\n",
               tf_nump());
   if (tf_typep(1) != tf_string)
      tf_error(" The parameter is not a string");
}

int total_power_consumed = 0;
int number_of_nets;

struct net_info {
   handle netHndl;
   char *prev_value;
   int  consumed_power;
   int toggled;
} netInfo[MAX_NET];
```

```
/* For better memory allocation, use a linked list */

int myCalltf()
{
   handle modHndl;
   handle *netHndl;
   handle terminalHndl;
   handle primitiveHndl;

   int i;
   static int first_time = 0;
   /* first_time == 0 indicates the first
      time calltf is being executed */

   /* Get a handle to the module specified */
   modHndl = acc_handle_by_name(tf_strgetp(1), NULL);

   if (first_time == 0) {

      /* Create a database for nets
         iterating on each net */
      number_of_nets = acc_count(acc_next_net, modHndl);
      netHndl =
         (handle *)malloc(number_of_nets*sizeof(handle));
      netHndl = acc_collect(acc_next_net, modHndl,
                            &number_of_nets);

      for (i=0; i<number_of_nets; i++) {
         netInfo[i].netHndl = netHndl[i];

         netInfo[i].prev_value = (char *)malloc(2L);
         strcpy(netInfo[i].prev_value,
                acc_fetch_value(netHndl[i], "%b"));
         /* Warning! All nets are assumed to be SCALAR! */

         netInfo[i].consumed_power = 0;
         if (acc_count(acc_next_load, netHndl[i]) > 0)  {
            terminalHndl = NULL;
            while (terminalHndl =
               acc_next_load(netHndl[i], terminalHndl)) {
```

```
            primitiveHndl =
               acc_handle_parent(terminalHndl);
            switch(acc_fetch_fulltype(primitiveHndl)) {
            /* Power consumption of AND gate */
            case accAndGate:
               netInfo[i].consumed_power += POWER_AND;
               break;
            /* Power consumption of XOR gate */
            case accXorGate:
               netInfo[i].consumed_power += POWER_XOR;
               break;
            /* Power consumption of OR gate */
            case accOrGate:
               netInfo[i].consumed_power += POWER_OR;
               break;
            /* Power consumption of NAND gate */
            case accNandGate:
               netInfo[i].consumed_power +=
                  POWER_NAND;
               break;
            /* Power consumption of XNOR gate */
            case accXnorGate:
               netInfo[i].consumed_power +=
                  POWER_XNOR;
               break;
            /* Power consumption of NOR gate */
            case accNorGate:
               netInfo[i].consumed_power += POWER_NOR;
               break;
            /* Add other gates here */
            /* Power consumption for the rest */
            default:
               netInfo[i].consumed_power +=
                  POWER_DFLT;
            }
         }
      }

      netInfo[i].toggled = 0;
   }
```

```
            free(netHndl);
            first_time = 1;
          } /* if first time*/
          /* If is not the first time */
          else
             for (i=0; i<number_of_nets; i++) {
                 if (strcmp(netInfo[i].prev_value,
                       acc_fetch_value(netInfo[i].netHndl,
                                        "%h")) != 0) {
                    if (is_toggled(netInfo[i]==1) {
                        netInfo[i].toggled = 1;
                        total_power_consumed +=
                           netInfo[i].consumed_power;
                    }
                    strcpy(netInfo[i].prev_value,
                         acc_fetch_value(netInfo[i].netHndl,
                                          "%h"));
                 }
             }
}

/* This function returns 1 if net is toggled else it
   returns 0. This may be debated, but we will consider
   a net toggled, only when the prev_value and
   current value are valid logic values */
int is_toggled(net)
struct net_Info net;
{
if (((strcmp(net.prev_value,"0")==0) ||
     (strcmp(net.prev_value,"1")==0)) &&
   ((strcmp(acc_fetch_value(net.netHndl,"%h"),"0")==0) ||
     (strcmp(acc_fetch_value(net.netHndl,"%h"),"1")==0)))
   return(1);
else
   return(0);
}

int myMisctf(data, reason)
int data, reason;
{
```

```
    int i, untoggled;

    /* Time to check the consumed power
       and coverage of the nets */
    if (reason == reason_finish) ||
       (reason == reason_disable) {
       io_printf("Total power consumed: %d\n",
                 total_power_consumed);

       untoggled = 0;
       for (i=0; i<number_of_nets; i++) {
          if (netInfo[i].toggled == 0) {
             if (untoggled == 0)
                io_printf(" List of untoggled nets:\n");
             untoggled++;
             io_printf(" -> %s\n",
                acc_fetch_fullname(netInfo[i].netHndl));
          }
       }
       io_printf(" Coverage = %f%%\n",
(double)(number_of_nets-untoggled)*100.0/number_of_nets);
       acc_close();
    }
}
```

4.3 Self check

- In the above program, all nets are assumed to be scalar. Modify the program to handle buses in the current module.
- Modify the above program to check for any fan-out violation. For simplicity, assume that the maximum number of inputs the output of any gate can drive is constant (say, 3).
- Repeat the same problem when input capacitance depends on the type of gate.

5. PROBLEM 5: COMMUNICATING BETWEEN VERILOG PROCESSES

So far all the applications discussed in this chapter were developed around a single system call meant for one specific task. In our next example, two different system calls from two Verilog simulations, possibly running on two different machines, interact with each other. Regarded as one of the classic problems solved using PLI, this example also shows how simulation performance can be improved using networking and multiprocessing.

The following figure shows a typical scenario where the system calls will be used. As shown in the figure, P1 and P2 represent two Verilog simulation processes. Together, their corresponding Verilog programs M1 and M2 represent a single model. Some of the possible reasons for breaking up the model into two parts and running them as separate processes are:

- The model is too big to run as a single process.
- The two parts of the model are written for different platforms (for example, one uses a library developed on UNIX OS and the other uses the same library developed on Windows NT) or resources (such as one for simulation and the other for hardware emulation).
- As a single process, the runtime of the simulation will be too high.

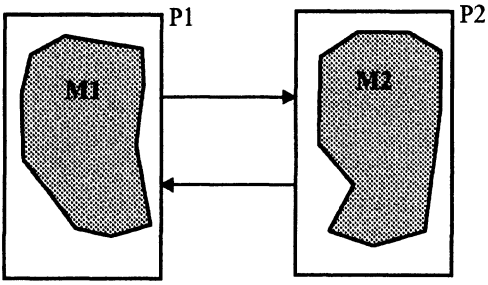

Figure 7. Processes communicating with each other

The entire model is assumed to be synchronous. Since the model will be divided into M1 and M2, the outputs of each part will become the inputs to the other and vice-versa. At certain (positive/negative/both) edges of the clock, each module passes the values of its output signals to the other. The second module use these values as its inputs. Subsequently, these input values are used to produce the outputs of the second module at the next clock.

The following examples show (much) smaller versions of M1 and M2 called `client` and `server`. Remember that these two modules must be simulated as separate processes.

```
module client;

reg [3:0] counter;
reg clk;
reg right_rcv;
integer i;

initial  begin
   counter = 4'b1001; // arbitrary value
   clk = 1'b0;
   $monitor($time, "  clk = %b,reg = %b",clk, counter);
   #100 $finish;
end

always
#5 clk = ~clk;

always @(posedge clk) begin
   $data_xchange_cl(counter[3], right_rcv);
   right_rcv = counter[0];
   for (i=1; i<=3; i=i+1)
      counter[i-1] = counter[i];
end
endmodule

module server;
reg [3:0] counter;
reg clk;
reg right_rcv;
integer i;

initial  begin
   counter = 4'b0101; // some arbitrary value
   clk = 1'b0;
   $monitor($time, "  clk =%b, reg = %b",clk, counter);
```

```
    #100 $finish;
  end

  always
  #5 clk = ~clk;

  always @(posedge clk) begin
    $data_xchange_sr(counter[3], right_rcv);
    right_rcv = counter[0];
    for (i=1; i<=3; i=i+1)
      counter[i-1] = counter[i];
  end

  endmodule
```

The modules `server` and `client` each represent a 4-bit shift register. At each positive edge of `clk`, each register is right-shifted by one bit. Also, using one of the system calls, the leftmost bit of each register is loaded with the shifted out value of the rightmost bit of the other register as shown in Figure 8. In effect, `server` and `client` together form a 8-bit ring counter.

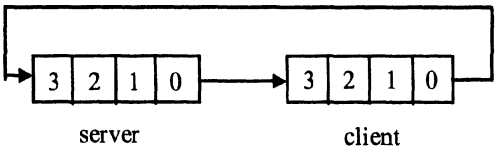

Figure 8. An 8-bit ring counter

There are a few things to note in the above Verilog description of the two modules `client` and `server`.

- Although from the hardware functionality point of view the modules are identical, for reasons to be discussed later, they invoke separate system calls. When `client` invokes `data_xchange_cl()`, `server` calls `data_xchange_sr()`.
- The clock signal `clk` is initialized separately in two modules. Since the time periods of these two clocks are same and data is exchanged only at their positive edges, the two clocks behave identically, as if the entire circuit is synchronized with any of them. In a more general case, the data transfer may take place asynchronously as long as a coherent protocol is followed.

5.1 Approaching a solution

It is clear that in order to exchange data between them, the processes must communicate with each other. There are various protocols available for interprocesses communication. Among them, TCP-IP is perhaps the most widely used for its simplicity and high portability. A thorough discussion of TCP-IP is beyond the limit of this book. However, here are the basic concepts that will be used to solve our problem.

1. The communication among two or more processes is done by using *sockets*. A socket is an endpoint for communication between processes. It is a software entity that provides the basic building block for interprocess communication.

2. A *communication domain* is a type of network with particular conventions such as how sockets are named and used. Sockets normally exchange data only with sockets in the same domain.

3. The two most widely used communication domains are *UNIX* and *internet*. Remember, they are not related to UNIX operating system or the Internet as such. For example, UNIX communication domain may also be implemented on non-UNIX operating systems.

Going by the popularity, we will implement this example on internet domain.

4. For internet addresses, a socket is usually identified by the pair <*node.port*> where *node* is a four byte internet address and *port* is two bytes identifying the network interface.

An interprocess communication in internet domain consists of the following three main parts

- Opening and initializing a socket.
- Transferring data using this socket.
- Closing the socket.

Each of this steps is described below.

5. A socket is opened by calling the function `socket()` as shown in the code fragment below.

```
int s;
s = socket(AF_INET, SOCK_STREAM, 0);
```

The first argument `AF_INET` indicates that we are using the internet domain. The argument `SOCK_STREAM` implies that the type of the socket `s` is *stream*, one of the most widely used sockets for normal data communication.

The 0 as the last argument indicates that the inherent protocol governing this socket will be the same as the default protocol for the system.

6. When two sockets are connected, one of them acts as the *client* and the other as the *server*. The client is the one which initiates a connection to the server. For this reason, the server socket is usually created before the client (just to make sure that the server exists when the request from the client arrives).

First the server is assigned a *name* using the function `bind()`. This name is actually a set of properties that define the server socket. These properties are defined using a structure type `sockaddr_in`. The essential fields of this structure are shown below.

```
struct sockaddr_in {
short   sin_family;
   /* The domain name - in our case AF_INET */
unsigned short sin_port;
   /* The port number of the server socket */
struct  in_addr sin_addr;
   /* We will use the system default value of sin_addr
      by specifying  INADDR_ANY for this field */
};
```

The following example assigns a name `addr_server` to a server socket `s_server`.

```
bind(s_server,
     (struct sockaddr *) &addr_servr,
     sizeof(addr_servr))
```

7. Once the server socket is assigned a name, it waits for the client's request to arrive. This is done by using the function `listen()` as shown below.

```
listen(s_server,MAX_CLIENT);
/* MAX_CLIENT denotes the system default maximum number
   of clients that this server is supposed to handle.
   In our case, there will be only one client */
```

8. Now that the server is in listening mode, the client can request for a connection to the server using the function `connect()`. In the following

example, `connect()` establishes the connection between the client socket `s_client` and the server, whose properties are defined in a structure `addr_server` of type `sockaddr_in` defined earlier.

```
connect(s_client,
        (struct sockaddr *)&addr_server,
        sizeof (addr_server))
```

9. With the socket marked as listening, the server can now accept a connection:

```
fromlen = sizeof (from);
snew = accept(s, (struct sockaddr *) &from, &fromlen);
```

If the server wishes to find out who its client is, it may supply a buffer for the client socket's address `from`. However, in our case, since there is only one client, the second and the third arguments will be set to 0.

Once the connection is established, the communication can go on in both directions and both client and server behave in exactly the same way.

10. Transferring data via a socket is done using the library functions `read()` and `write()`. The function `read()` is used to read data from a socket `s` as shown below.

```
read(s, buf, sizeof(buf));
```

where the data is read into a character buffer `buf`.

Similarly, data kept in a buffer `buf` is written to a socket `s`, using

```
write(s, buf, sizeof(buf));
```

11. Once the data transfer is completed, the sockets on both ends of communication are closed by their respective parent processes using

```
close(s);
```

where `s` is the socket to be closed.

5.2 Program listing

In order to hide the details of how an interprocess communication is implemented inside our PLI routine, we will create several functions which will build up the TCP stack. Inside the system call, instead of calling the basic TCP/IP functions, we will use these custom-made functions making the overall code more concise. Furthermore, we will club these functions together in a header file `client_server_inet.h` and include it in the main PLI routine.

The following table describes our custom functions and what they do.

Table 29. Custom functions and their descriptions

Function name	Description of the function
init_client	Opens the socket for the client
client_connect	Connects the client to the server
client_read	Reads a data buffer for the client initialized earlier by the server
client_write	Writes to a data buffer for the client to be read by the server later
close_client	Closes the client socket
init_server	Opens a socket for the server and puts it into the listen mode
server_read	Reads a data buffer for the server initialized by the client earlier
server_write	Writes to a data buffer for the server to be read by the client later
close_server	Closes the server socket

The header file `client_server_inet.h.` is shown below. Note that the header file will be included in both client and server programs, but each will use its own set of functions.

Also, the program is tested using IRIX6.2 OS; however, the implementation will not be significantly different in other versions of UNIX.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORTNUM 6666
   /* Use another port number, if this one is not
      available.  */

void init_client();
```

```
void client_connect();
void client_read();
void client_write();
void close_client();

void init_server();
void server_read();
void server_write();
void close_server();

int s_client;
int s_server;
int snew_server;

struct sockaddr_in addr_client;
struct sockaddr_in addr_servr;

char buf[80];

/*****************************************
 *
 * All client side routines are here
 *
 ****************************************/

void init_client() {
   s_client = socket(AF_INET, SOCK_STREAM, 0);
   if (s_client < 0) {
      perror(" Error in opening the client...Exiting\n");
      exit(2);
   }
}

void client_connect(hostname)
char *hostname;
{
   /* Initializing the "name" of the server
      with mutually agreed upon values */
   bzero(&addr_servr, sizeof(addr_servr));
   addr_servr.sin_family = AF_INET;
```

```
    addr_servr.sin_port=htons(PORTNUM);
    addr_servr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (connect(s_client,
                (struct sockaddr *) &addr_servr,
                sizeof (addr_servr)) < 0) {
       perror("Connection failed... Exiting\n");
       exit(2);
    }
}

void client_read(str_buf)
char *str_buf;
{
    char *buf;
    int rval;

    buf = (char *)malloc(BUFSIZE);
    bzero(buf, BUFSIZE);
    if ((rval = read(s_client, buf, BUFSIZE)) < 0) {
       perror("Error reading stream message...Exiting\n");
       exit(2);
    }
    else
       strcat(str_buf, buf);
}

void client_write(buf)
char *buf;
{
    int wval;

    if ((wval = write(s_client, buf, BUFSIZE)) < 0) {
       perror("Error writing stream message...Exiting\n");
       exit(2);
    }
}

void close_client() {
    close(s_client);
```

```
}

/****************************************
 *
 * All server side routines are here
 *
 ****************************************/
void init_server() {
   /* Opening the server socket */
   if ((s_server = socket(AF_INET, SOCK_STREAM, 0))<0) {
      perror("Error opening server...Exiting\n");
      exit(2);
   }

   /* Initializing the "name" of the server */
   addr_servr.sin_family = AF_INET;
   addr_servr.sin_port = htons(PORTNUM);
   addr_servr.sin_addr.s_addr = htonl(INADDR_ANY);

   /* Binding the server in the name space */
   if (bind(s_server,
            (struct sockaddr *) &addr_servr,
            sizeof(addr_servr)) < 0)  {
      perror ("Error binding the server...Exiting\n");
      exit(2);
   }

   /* Putting the server in the listen mode */
   if (listen(s_server,MAX_CLIENT) < 0) {
      perror("Error in listening ...Exiting\n");
      exit(2);
   }

   /* Putting the server in accept mode */
   if ((snew_server = accept(s_server, 0, 0)) < 0) {
      perror(" accept error in server...Exiting\n");
      exit(2);
   }
}
```

```
void server_read(str_buf)
char *str_buf;
{
   char *buf;
   int rval;

   buf = (char *)malloc(BUFSIZE);
   bzero(buf, BUFSIZE);
   if ((rval = read(snew_server, buf, BUFSIZE)) < 0) {
      perror("Error reading stream message...Exiting\n");
      exit(2);
   }
   else
      strcat(str_buf, buf);
}

void server_write(buf)
char *buf;
{
   int wval;

   if ((wval = write(snew_server, buf, BUFSIZE)) < 0) {
      perror("Error writing stream message...Exiting\n");
      exit(2);
   }
}

void close_server() {
   close(snew_server);
   close(s_server);
}
```

Now that we have the basic building blocks for the interprocess communication, we are all set for writing the actual PLI routine. Remember, for each of the server and the client, we have a separate system call. However, we can create a single PLI routine, but use it differently for the client and the server depending on the values of *data* parameter.

The PLI routine calls `init_client()` or `init_server()` at the beginning using the *reason* `reason_endofcompile`.

The data is exchanged across the boundary at the end of the simulation time slot when all other activities scheduled for that time slot are finished. This is to ensure that only the final values of the interfacing signals are exchanged. This is achieved using the reason `reason_synch`.

Lastly, using the reason `reason_finish`, the functions `close_server()` or `close_client()` close the opened sockets at the end of the simulation.

Here is the listing for the PLI routine for implementing the system calls `$data_xchange_cl` and `$data_xchange_sr`.

```
#include "veriuser.h"
#include <string.h>

/* We have only one client, but just in case ;-) */
#define MAX_CLIENT 2
/* The buffer size for data exchange */
#define BUFSIZE 100
#include "client_server_inet.h"

#define CLIENT 1
#define SERVER 0

int myCalltf();
int myMisctf();

static char *my_buf;

int myCalltf() {
   tf_synchronize();
}

int myMisctf(data, reason )
int data;
int reason;
{

  switch(reason) {
  case reason_endofcompile:
     /* This part is executed at the beginning of the
        simulation */
```

```
        my_buf = (char *)malloc(BUFSIZE);
        switch(data) {
           case CLIENT:   /* client */
              init_client();
              /* My_Machine is the hostname for server*/
              client_connect("My_Machine");
              break;
           case SERVER:   /* server */
              init_server();
              break;
        }
        break;

     case reason_synch:
        /* This part is executed at the end of the
           simulation time slot when the system call
           is invoked */
        switch(data) {
           case CLIENT:   /* client */
              my_buf = tf_strgetp(2, 'b');
              my_buf[1] = '\0';
              /* The client writes first and then reads,
                 just the opposite of the server */
              client_write(my_buf);
              client_read(my_buf);
              tf_strdelputp(1, 1, 'b', my_buf, 0, 0);
              break;
           case SERVER:   /* server */
              /* The server reads first and then writes */
              server_read(my_buf);
              tf_strdelputp(1, 1, 'b', my_buf, 0, 0);

              my_buf =  tf_strgetp(2, 'b');
              my_buf[1] = '\0';
              server_write(my_buf);
              break;
        }
        break;
     case reason_finish:
        /* This part is executed at the end of the
```

```
            simulation */
        switch(data) {
            case CLIENT:    /* client */
                close_client();
                break;
            case SERVER:    /* server */
                close_server();
                break;
        }
        break;
    }
}
```

5.3 Self check

- Let us consider a simulation job which has been partitioned into two parts. The first part runs as usual simulation as a single process. But, the second part runs on a hardware emulator connected to the (serial or parallel) port of the same machine. Assuming the I/O is memory mapped and the data is exchanged through files, create a system call for the software simulation to handshake with the hardware emulation.
- Modify the above program to break up a simulation job into three processes. Generalize this concept further to break down a job into *n* number of processes.

6. PROBLEM 6: VIRTUAL MEMORY MANAGEMENT

Quite often to verify the functionality of a complex system (such as one involving a processor, micro-controller or DSP processor) you need to have a large chunk of memory instantiated along with other components of the system. For example, many modern computer systems have a main memory of 100MB to 1GB. It is also usual to have the address range of the memory partitioned into different zones, each dedicated for a specific purpose. Unless some restrictions on memory access is applied, this requires all the memory modules (or banks) to be instantiated in the simulation model of the system. However, in most commercial simulators (notable exception: VERA) memory variables are statically allocated. This means a simulation can not proceed unless that much memory is available on the simulating machine.

The system call `$mem_read` and `$mem_write` described below provide a convenient way to integrate theoretically infinite amount of memory (constrained by the size of your hard disk) in a simulation. The actual contents of the memory are stored in a file. Then a small amount of memory is instantiated in the Verilog code. The system call `$mem_read` reads data from any location of the memory and `$mem_write` modifies a given location.

As an example, the following program instantiates a memory `mymemory` of depth 4 (3 to 0). The depth of the actual memory in this example is 16. The file `file.dat` stores the initial values of the memory locations. In the program, all 16 memory locations are read, modified and then read back again. The register `my_reg` acts as a buffer variable. `$mem_read` reads the contents of the memory location `i` and puts them in `my_reg` and `$mem_write` modifies the memory location `i` with the contents of `my_reg`.

```
module mymodule;
reg [15:0] mymemory [3:0];
reg [15:0] my_reg;
integer i;

initial begin
    $readmemb("file.dat", mymemory);
       /* Initialize the memory at the beginning */
    for (i=0; i<16; i=i+1) begin
       #100;
       $mem_read (mymemory[0],  // memory
                  "file.dat",   // data file
                  i,            // index
                  my_reg);      // register
       #1;
       $display(" index = %d, my_reg = %b ", i, my_reg);
       my_reg = i;
       my_reg = {my_reg[3:0], my_reg[3:0],
                 my_reg[3:0], my_reg[3:0]};
       // modifying the contents of my_reg
       #100;
       $mem_write (mymemory[0],  // memory
                   "file.dat",   // data file
                   i,            // index
                   my_reg);      // register
       #100;
```

```
        $mem_read (mymemory[0],
                   "file.dat",
                   i,
                   my_reg);
        #1;
        $display(" index = %d, my_reg = %b ", i, my_reg);
    end
    #100 $finish;
end
endmodule
```

There are two important points to note in the above Verilog program.

- Note that, unlike a scalar or vectored object, a memory can not be passed to a system call as an argument. How a memory should be passed to a system call is simulator dependent. In most of the simulators, the first memory element should be passed as the argument (which in our case is `mymemory[0]`).
- You might wonder what is the use of defining a smaller Verilog memory when all accesses are done to or from the file. The small memory acts as a local cache and reduces the number of fetching from the hard-disk (or making a cache flush) making the simulation faster. The choice of a proper memory size is a trade-off between simulation performance and system resources.

The following list shows the typical contents of the data file `file.dat`. Remember, if any location is uninitialized, it should contain Xs in the corresponding location in the file.

```
0000000000000000
0000000100000001
0000001000000010
0000001100000011
0000010000000100
0000010100000101
0000011000000110
0000011100000111
0000100000001000
0000100100001001
0000101000001010
0000101100001011
0000110000001100
```

```
0000110100001101
0000111000001110
0000111100001111
```

6.1 Approaching a solution

Essentially, here we are going to implement a write back cache coherency protocol. The small amount of memory (`mymemory` in our example) instantiated in the Verilog program will act as the cache and the data file (`file.dat`) as the main memory. For the rest of our discussion, we will call the memory variable and the data file by the names cache and main memory respectively.

Throughout the program, the two system calls keep track of two variables. They are:

1) The current starting index of the cache: This is the index of the main memory which is currently loaded into the `mymemory[0]`. Since the depth of the cache is known, this will determine whether an access to a particular index is available in the cache (i.e. a *cache-hit*) or not (a *cache-miss*).

2) The *dirty* state of the cache: A variable to determine whether the contents of `mymemory` are different from what was fetched from `file.dat`. If the cache is dirty, its contents have to be written back to the main memory, each time there is a cache miss during a read or write.

The algorithm for a memory read is shown below.

```
if (the index is a cache miss) {
  if (cache is dirty)
    write back the contents of the memory;
  load the memory from the file;
  starting index = index;
}
load the contents of the buffer register from the memory;
```

The algorithm for a memory write is almost identical.

```
if (the index is a cache miss) {
  if (cache is dirty)
    write back the contents of the memory;
  load the memory from the file;
  starting index = index;
}
```

modify the memory location;
mark the cache as dirty;

Since, the two algorithms share a common flow, it makes more sense for their corresponding PLI routines to share a set of functions between them.

6.2 Program listing

It is easy to see that both the applications can share the same checktf function.

Also, a careful look at the two algorithms will also reveal that the calltf functions `mem_read_call()` and `mem_write_call()` also share a few other functions as shown below. Remember, this program assumes that the contents of the memory given in the data file are in binary format.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "veriuser.h"

#define MEMORY      1
#define DATAFILE    2
#define INDEX       3
#define REG         4

int mem_check();
int mem_read_call();
int mem_write_call();

char *read_it_from_memory();
void read_in_values_from_the_file();
void write_back_the_memory_into_file();
void put_str_into_memory();
char *ch2a();

s_tfnodeinfo info;
static int strt_index=0, changed=0;

/****************************************
   This is the checktf function for both
```

```
    $mem_read and $mem_write.
 ****************************************/
int mem_check()
{
  FILE *fp;

   if (tf_nump() != 4) {
      io_printf("mem_read:\n");
      io_printf(" USAGE: $mem_read(mem_variable, \n");
      io_printf("          name_of_the_file,\n");
      io_printf("          index_to_be_read,\n");
      io_printf("          reg_to_be_assigned);\n");
      exit(2);
   }

   if (tf_typep(MEMORY) != tf_readwrite) {
      io_printf(" Arg 1 must be a memory variable;\n");
      exit(2);
   } else {
      tf_nodeinfo(MEMORY, &info);
      if ( info.node_type != tf_memory_node ) {
         io_printf(" Arg 1 must be a memory variable;\n");
         exit(2);
      }
   }

   if (tf_typep(DATAFILE) != tf_string) {
      io_printf(" Arg 2 must be a string (filename);\n");
      exit(2);
   }
   else if ((fp = fopen(tf_strgetp(DATAFILE),"r+"))
            == (FILE *)NULL) {
      io_printf(" Data file %s could not be opened;\n",
                tf_strgetp(DATAFILE));
      exit(2);
   }

   if ((tf_typep(INDEX) != tf_readonly) &&
       (tf_typep(INDEX) != tf_readwrite)) {
      io_printf(" Arg 3 must be an integer or const;\n");
```

```
        exit(2);
    }

    if (tf_typep(REG) != tf_readwrite) {
        io_printf(" Arg 4 should be a reg;\n");
        exit(2);
    }
    fclose(fp);
  }

  /****************************************
   This is the calltf function for $mem_read
   ****************************************/
  int mem_read_call()
  {
    int length, index;

    length = info.node_mem_size;
    index = tf_getp(INDEX);

    if (( strt_index > index) ||
        ( strt_index+length <= index))  {
       /* If it is a cache miss */
       if (changed)   /* if the cache is modified */
          write_back_the_memory_into_file(strt_index);
       read_in_values_from_the_file(index);
       strt_index = index;
    }

    tf_strdelputp(REG, tf_sizep(REG), 'b',
                  read_it_from_memory(index-strt_index),0,0);
  }

  /****************************************
   This is the calltf function for $mem_write
   ****************************************/

  int mem_write_call()
  {
    int length, index;
```

```
  char *val_str;

  length = info.node_mem_size;
  index = tf_getp(INDEX);
  val_str = (char *)malloc(tf_sizep(REG));

  if (( strt_index >= index) ||
      ( strt_index+length < index)) {
     /* if it is a cache miss */
     if (changed)   /* if the cache is modified */
        write_back_the_memory_into_file(strt_index);
     read_in_values_from_the_file(index);
     strt_index = index;
  }

  strcpy(val_str, tf_strgetp(REG, 'b'));
     /* copy the value of the reg to
        the val_str as a string */
  put_str_into_memory(val_str, index-strt_index);
  changed = 1;
}

/*********************************************
   This function reads a word from the memory
   and returns its value as a string.
 *********************************************/
char *read_it_from_memory(index)
int index;
{
   int i, j;
   char c;
   char *astring, *bstring, *cstring;
   int ngroups;

   ngroups = info.node_ngroups;

   astring=(char *)malloc(8*ngroups*sizeof(char)+1);
   strcpy(astring,"");
   bstring = (char *)malloc(8*ngroups*sizeof(char)+1);
   strcpy(bstring,"");
```

```
    cstring = (char *)malloc(8*ngroups*sizeof(char)+1);

    for (i=2*ngroups*index;i<2*ngroups*index+ngroups;i++)
    {
       c = *(info.node_value.memoryval_p+i);
       strcat(astring, ch2a(c));
    }

    for (j=i; j<2*ngroups*index+2*ngroups; j++)
    {
       c = *(info.node_value.memoryval_p+j);
       strcat(bstring, ch2a(c));
    }

    for (i=0; i<strlen(astring); i++)
    {
       if ((astring[i]=='0') && (bstring[i]=='0'))
          cstring[i] = '0';
       else if ((astring[i]=='1') && (bstring[i]=='0'))
          cstring[i] = '1';
       else if ((astring[i]=='1') && (bstring[i]=='1'))
          cstring[i] = 'X';
       else if ((astring[i]=='0') && (bstring[i]=='1'))
          cstring[i] = 'Z';
    }
    cstring[i] = '\0';
    return(cstring);
}

/**************************************************
   This function converts each bit of a character
   variable to a character and collects them as a
   string. It returns the string after some
   rearrangement. Remember 8 bits make a byte ;-)
 **************************************************/

char *ch2a(c)
unsigned c;
{
   char *s, *t, ch;
```

```
    int i;

    s = (char *)malloc(sizeof(char)+1);
    t = (char *)malloc(sizeof(char)+1);

    for(i=0; i<sizeof(char)*8; i++) s[i] = '0';
    s[i] = '\0';
    i = 0;
    do {
       ch = s[i++] = c%2 + '0';
    } while ((c/= 2) > 0);

    for (i = 0; i < sizeof(char)*8; i++)
       t[i] = s[sizeof(char)*8-i-1];
    t[i] = '\0';
    return(t);
}

/***********************************************
   This function initializes the index ii of
   memory after reading the corresponding
   value from the data file.
 ***********************************************/
void read_in_values_from_the_file(ii)
int ii;
{
   int i, j, k, m, beyond;
   char *s;
   int ngroups, depth;
   FILE *fp;

   /* Initialize all local variables */
   ngroups = info.node_ngroups;
   depth = info.node_mem_size;
   s = (char *)malloc(8*ngroups*sizeof(char)+8);
   fp = fopen(tf_strgetp(DATAFILE),"r+");

   /* Take the file pointer right
      before the indexed element */
   fseek(fp, 0, 0);
```

```
   beyond = 0;
   for (i=0; i<ii; i++)
      if (fscanf(fp, "%s", s) == EOF) {
         i = ii;
         beyond = 1;
      }

   /* Now read as many elements as the depth of the
      memory from the file and put it into memory.  */
   for (j=ii; j<(ii+depth); j++) {
      /* read an element */
      if ((beyond == 1) || (fscanf(fp, "%s", s) == EOF)) {
         for (i=0; i<8*ngroups*sizeof(char); i++)
            s[i] = 'x';
         s[i] = '\0';
      }
      put_str_into_memory(s,j-ii);
   }
   fclose(fp);
}

/*************************************************
   This function writes back the contents of the
   memory to the data file. The current starting
   index of the memory is passed to the function
   as its argument.
 *************************************************/
void write_back_the_memory_into_file(strt)
int strt;
{
   int i;
   int ngroups, depth;
   char *s;
   FILE *fp;

   s = (char *)malloc(8*sizeof(char)*ngroups);
   ngroups = info.node_ngroups;
   depth = info.node_mem_size;
   fp = fopen(tf_strgetp(DATAFILE),"r+");
```

```
   fseek(fp, strt*(8*sizeof(char)*ngroups+1), 0);
   for (i=0; i<depth; i++)
      fprintf(fp, "%s\n", read_it_from_memory(i));
   changed = 0;
   fclose(fp);
}

/*******************************************************
   This function initializes the index ind of the memory
   with a value given by the string s.
 *******************************************************/
void put_str_into_memory(s,ind)
char *s;
int ind;
{
   char *astring, *bstring, *each_char;
   int ngroups;
   int i, k, m;

   ngroups = info.node_ngroups;

   astring = (char *)malloc(8*ngroups*sizeof(char)+1);
   bstring = (char *)malloc(8*ngroups*sizeof(char)+1);
   each_char = (char *)malloc(8*sizeof(char));

   for (i=0; i<8*ngroups*sizeof(char); i++)
      astring[i] = bstring[i] = '0';
   astring[i] = bstring[i] = '\0';

   /* convert s into astring and bstring */
   for (k=0; k<strlen(s); k++)
      if (s[k] == '0') {
         astring[k] = '0';
         bstring[k] = '0';
      } else if (s[k] == '1') {
         astring[k] = '1';
         bstring[k] = '0';
      } else if ((s[k] == 'x') || (s[k] == 'X')) {
         astring[k] = '1';
         bstring[k] = '1';
```

```
        } else if ((s[k] == 'z') || (s[k] == 'Z')) {
           astring[k] = '0';
           bstring[k] = '1';
        }

     /* convert the strings astring and bstring
        into ngroups number of characters. */
     for (k=0; k<ngroups; k++) {
        for (m=0; m<8*sizeof(char); m++)
           each_char[m] = astring[k*8*sizeof(char)+m];
        each_char[m] = '\0';
        *(info.node_value.memoryval_p+2*ind*ngroups+k)
           = (char)strtol(each_char, (char **)NULL, 2);
     }

     for (k=0; k<ngroups; k++) {
        for (m=0; m<8*sizeof(char); m++)
           each_char[m] = bstring[k*8*sizeof(char)+m];
        each_char[m] = '\0';
              *(info.node_value.memoryval_p+
                2*ind*ngroups+ ngroups+ k)
              = (char)strtol(each_char,(char **)NULL, 2);
     }
}
```

If this is run in VCS environment, the `pli.tab` file will look like:

```
$mem_read check=mem_check call=mem_read_call
$mem_write check=mem_check call=mem_write_call
```

6.3 Self check

- Note that the above program works only for *one* instance of memory. This is because the variables `strt_index` and `changed` are declared as `static int`. In real hardware, a dirty bit is associated with each memory location (or block) and the starting location is limited to certain indices, known as *page boundaries* to tackle this situation. Modify the above program to handle multiple instances of the memory.
- Design a dual port memory working on the same principle. This means two different events can access the memory at the same time. You will

have to be careful to avoid any race condition on the static variables. Solve this problem using a spin locked semaphore.

7. PROBLEM 7: ASSIGNING A REGISTER VALUE FROM A FILE

One of the most frequently asked questions on PLI in *comp.lang.verilog* newsgroup is how to read a value from a file and assign it to a register. Although the built-in system calls `$readmemb`/`$readmemh` do this for memory elements, this often-needed system call is not part of IEEE Std.1364-1995 Verilog.

In this example, we are going to develop a PLI routine for a system call `$fscanf`. The overall functionality of `$fscanf` is similar to its counterpart `fscanf()` in standard C library `stdio.h`. It takes 3 or more arguments as shown below.

- The first argument must be a string giving the name of the input file.
- The second argument, also called the *format string*, is another string specifying the format of the input values. This string consists of one or more *format specifiers* and white space. Each of the format specifiers starts with a '%'. We will discuss the syntax of a format specifier in a short while.
- Starting from the third argument, each argument specifies the register to be assigned. If there are more registers than the number of format specifiers, the excess registers are ignored.

As mentioned above, the format string (2nd argument) may contain white space character (space, tab, newline, form feed, vertical tab, return). Each of the white space characters in the format string matches zero or more white space characters in the input file.

The format specifiers in the format string, as mentioned earlier, must start with a '%'.

Next is an optional '*', which means, even though the value will be read from the input file, the assignment to the register will not be done.

After this there is an optional decimal number which specifies the number of characters to be read from the file for this value.

The next character in the format specifier is a non-optional *format character* and must be one of the following.

Table 30. Different format characters and their implications

format character	indicates	and can read the following characters from the input string
b	Binary values	0, 1, X, x, Z, z
o	Octal values	0-7, X, x, Z, z
d	Decimal values	0-9, **no** X, x, Z or z
h	Hexadecimal values	0-9, a-f, X, x, Z, z

Any character in the format string, which is not located between a '%' and a format character must match the same character from the input file.

Let us consider the Verilog program shown below.

```
module mymodule;
reg [7:0] r1, r2;
   initial begin
      r1 = 8'h0;
      r2 = 8'h0;
      #100 $fscanf("myfile.txt", " %2o  %*d %h", r1, r2);
      #100 $display($time, " r1 = %b, r2 = %b", r1, r2);
      #100 $finish;
   end
endmodule
```

Also, assume that the file `myfile.txt` looks as shown below.

```
74      12      a1
```

Here, the format string is `" %2o  %*d %h"`. There are 3 format specifiers - %2o, %*d and %h. `$fscanf` opens the file `myfile.txt` and read the value 74 as an octal number and assigns it to the register r1. Next it reads the value 12 as a decimal but does not assign it to r2 because of the * following the % character. At the end it reads the value a1 and assigns it to r2.

7.1 Approaching a solution

The task of this system call can be divided into the following parts:

Parse the format string into different format specifiers.

Read the value from the input file corresponding to each of the format specifier.

Assign the value to the appropriate register.

It is easy to see that the most difficult part of writing this system call would be the parsing of the format string. Notice that the format specifiers can not be retrieved directly as tokens using `strtok()` function, as the delimiters separating two format specifiers can be either a white space character or a set of characters matching the input stream. As a result, each character of the format string must be read serially and interpreted.

The second part of our job will be easier as we can use the C function `fscanf()` which has almost a one-to-one correspondence with the present system call. Here, note that we need to consider two special cases. First, `fscanf()` can not read binary numbers. Also, we must make proper arrangement to deal with x and zs.

Assigning the value to the register will be done using standard `tf_strdelputp()` function.

7.2 Program listing

The following program shows the implementation of `$fscanf()`. The entire program is divided into two parts and stored in separate files for convenience. The first file `calltf.c` contains the main PLI functions. The second file `lex_me.c` contains the function `lex_me()` which does the lexical analysis of the format string and is called from the calltf function `myCalltf()` of the first file.

7.2.1 calltf.c

```
#include <stdio.h>
#include "veriuser.h"

FILE *fp;
int myCalltf(),
    myChecktf();

extern int lex_me();

int myChecktf() {
   int i;
   static int first_time = 1;
```

```
   if (tf_nump() < 3)
      tf_error ("$fscanf: Minimum no. of args = 3\n");

   if (tf_typep(1) != tf_string)
      tf_error("$fscanf: Arg#1 must be a string\n");
   if (tf_typep(2) != tf_string)
      tf_error("$fscanf: Arg#2 must be a string\n");

   for (i=3; i<=tf_nump(); i++)
      if (tf_typep(i) != tf_readwrite)
         tf_error("$fscanf: Arg#%d must be a reg\n", i);

   if (first_time == 1)
      if ((fp = fopen(tf_strgetp(1), "r"))
          == (FILE *)NULL)
         tf_error("$scanf: File %s couldn't be opened\n",
                  tf_strgetp(1));
      else
         first_time = 0;
}

int myCalltf() {
   /* Algorithm for fscanf :
   store  the second argument (the control string);
   call the lexical analyzer function to parse the
   control string;
   */

   static char *ctrl_str;

   ctrl_str = (char *)malloc(strlen(tf_strgetp(2))+1);
   strcpy(ctrl_str, tf_strgetp(2));

   lex_me(ctrl_str, fp);
   free(ctrl_str);
   fclose(fp);
}
```

7.2.2 lex_me.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "veriuser.h"

int lex_me(ctrl_str, fp)
char *ctrl_str;
FILE *fp;
{
   int i=0;
   int cur_position = 0;
   int limit;
   int inside_format = 0;
   int skip = 0;
   int c;
   int width;
   int nparam = 3;
   char *val_str;
   char format;
   char *format_str;

   format_str = (char *)malloc(32);
   val_str = (char *)malloc(1024);
   limit = strlen(ctrl_str);

   while ((cur_position < limit) && !feof(fp)) {
      switch(ctrl_str[cur_position]) {
         case '%' :
         /* if this is a % start parsing the format*/
         inside_format = 1;
         skip = 0;
         width = 0;
         break;

         case '*':
         /* skip this field */
         if (inside_format)
            skip = 1;
```

```
else {
   c = fgetc(fp);
   if (c != '*')
     tf_error("Couldn't match '*'\n");
}
break;

case '0': case '1': case '2':
case '3': case '4': case '5':
case '6': case '7': case '8':
case '9':
   if (inside_format) {
      sscanf(ctrl_str+cur_position,"%d",&width);
      while (isdigit(ctrl_str[cur_position]))
         cur_position++;
      cur_position--;

      if (skip)
      /* if there is a width specifier
         (e.g. 2 in %*2d), then the
         format char (d) does not matter */
         for (i=1; i<=width; i++)
            c = fgetc(fp); /*swallow */
   } else { /* if it is outside format */
       c = fgetc(fp);
       if (c != ctrl_str[cur_position])
          tf_error("Couldn't  match %c\n",
                   ctrl_str[cur_position]);
   }
   break;

   case 'b': case 'o': case 'd': case 'h':
      if (inside_format) {
         format = ctrl_str[cur_position];
         switch(format) {
            case 'b':
            /* if it is in binary format */
            /* prepare a format string   */
            if (width)
               sprintf(format_str,
```

```
            "%%%d[01xzXZ]", width);
else
   strcpy(format_str, "%[01xzXZ]");

/* Read the value from the file */
fscanf(fp, format_str, val_str);

/* Now the assignment part */
if (!skip)
   tf_strdelputp(nparam++,
                 strlen(val_str),
                 'B',
                 val_str,
                 0,
                 0);
break;

case 'o':
/* if it is in octal format */
/* prepare a format string */
if (width)
   sprintf(format_str,
           "%%%d[01234567xzXZ]",
           width);
else
   strcpy(format_str,
          "%[01234567xzXZ]");

/* Read the value from the file */
fscanf(fp, format_str, val_str);

/* Now the assignment part */
if (!skip)
   tf_strdelputp(nparam++,
                 strlen(val_str)*3,
                 'O',
                 val_str,
                 0,
                 0);
break;
```

```
                    case 'd':
                    /* if it is in decimal format */
                    /* prepare a format string */
                    if (width)
                       sprintf(format_str,
                               "%%%d[0123456789]",
                               width);
                    else
                       strcpy(format_str,
                              "%[0123456789]");

                    /* Read the value from the file */
                    fscanf(fp, format_str, val_str);

                    /* Now the assignment part */
                    if (!skip)
                       tf_strdelputp(nparam++,
(int)(ceil(log((double)atoi(val_str)+1.0)/log(2.0))),
                                     'D',
                                     val_str,
                                     0,
                                     0);
                    break;

                    case 'h':
                    /* if it is in hexadecimal format */
                    /* prepare a format string */
                       if (width)
                          sprintf(format_str,
                          "%%%d[0123456789abcdefxzXZ]",
                                  width);
                       else
                          strcpy(format_str,
                             "%[0123456789abcdefxzXZ]");

                      /* Read the value from the file */
                       fscanf(fp, format_str, val_str);

                       /* Now the assignment part */
```

```
                if (!skip)
                   tf_strdelputp(nparam++,
                                 strlen(val_str)*4,
                                 'H',
                                 val_str,
                                 0,
                                 0);
                break;
             }
          inside_format = 0;
          width = 0;
          skip = 0;
       } else {
          c = fgetc(fp);
          if (c != ctrl_str[cur_position])
             tf_error("Couldn't match %c\n",
                      ctrl_str[cur_position]);
       }
    break;

    default:
       if (isspace(ctrl_str[cur_position])) {
          /* if this is a white space,
             jump to the next non-white space
             char. Also skip the white
             spaces in the file.
          */
          while
             (isspace(ctrl_str[++cur_position]));
          cur_position--;

          c = fgetc(fp);
          while (isspace(c))
             c = fgetc(fp);
          ungetc(c, fp);
       } else  {
          c = fgetc(fp);
          if (c != ctrl_str[cur_position])
             tf_error("Couldn't match %c\n",
                         ctrl_str[cur_position]);
```

```
            }
        }
        cur_position++;
    }
    free(format_str);
    free(val_str);
    return 0;
}
```

7.3 Self check

Our system call implements only part of the functionality of `fscanf()`. Here are some additional features that our system call is lacking.

- Extend the program to read string (%s), character (%c), float (%f) and double (%lf) formats. Also it should understand the meaning of special characters. Special characters are \" for the " character, \\ for the \ character, \oNNN is a single character whose ASCII value is specified by the octal number NNN, and %% for the character %.
- Extend the program so that it understands a '_' (as done in Verilog) in a value and skips it accordingly.
- Modify the program so that it can read the values from a given string, rather than a file.

8. SUMMARY

In this chapter, we have implemented a wide variety of system calls. We have also discussed how to apply more advanced techniques of PLI, such as, sharing data among applications, synchronizing simulation using callbacks, interfacing between the processes etc. You are strongly encouraged to explore how a particular example can be modified to use it somewhere else.

Also, you might have noticed that although all our examples in this chapter were written to help the process of simulation, they themselves did not affect the result of the simulation. Or in other words, these applications themselves were not part of a simulation model. In the next chapter, we are going to discuss a specific type of application of PLI used to build up a simulation model for a design object. As we will find out, this particular type of models, known as Bus Functional Models, are very useful in a number of places.

Chapter 6

WRITING BUS FUNCTIONAL MODEL

In today's world, it is often required by the members of a design team to share their design information with people outside the organization. This process of sharing valuable design data with the outside world – third party vendors, OEM manufacturers, contractors etc. – is unaviodable in most cases. *Functional Models*, also known as *Bus Functional Models* (BFM), play an important role in protecting intellectual property in such circumstances. PLI can be used to take advantage of the easy-to-program aspect of C together with rich simulation features of Verilog to produce functional models. In this chapter, we will find out the reasons behind making PLI as the choice for writing such models. Our first example of a simple crossbar switch will illustrate how to write a functional model. Once this gives the basic idea, we will move to designing a functional model of a rudimentary processor. From these examples, it will become apparent why PLI is the only choice for writing such a model.

1. WHAT IS A BUS FUNCTIONAL MODEL ?

The primary objective of any hardware description language is to describe a design concept that will eventually transform itself into a piece of silicon. However, at different stages of the design, depending on what level this concept is being conveyed to next, the description itself may take different shapes. For example, the description during the initial phase of the design when the designer needs to check the validity of the concept itself, is different from the description at the final phase when it is inputted into a

S. Mittra, *Principles of Verilog PLI*
© Kluwer Academic Publishers 1999

synthesis tool to obtain an equivalent gate structure. To summarize, as the design cycle matures, the description or *the model* also changes.

Depending on which phase of the design cycle they apply to, there are three approaches to write a model. They are:

- Behavioral/Bus Functional model
- RTL model
- Structural model.

We will discuss each of them separately below.

1.1 Behavioral / Functional Model

A behavioral or functional model allows you to use many high level constructs, such as nested loops, conditional statements, etc., which are common to any high-level language, such as C. Functional models are not synthesizable. The only objective of writing such a model is to achieve a port behavior which is exactly the same as what is desired from the actual silicon. How this behavior is achieved inside the model is not important. Or in other words, functional models, like a black box, hide the internal details of the implementation and derive only the results that will appear at the pins of the module. For this reason, functional models are often termed as Bus Functional Models (BFM). These types of models are typically used at the early phase of the design to prove the algorithm. They are also used in places where a gate level model is not required (such as designing a stub etc.).

1.2 Register Transfer Level (RTL) Model

Modeling at the register transfer level (RTL) also allows one to use high level constructs to some extent. But, at this stage, the main intention of the designer is to get a synthesized version after translating the design through a proper synthesis tool. Almost all synthesis tools impose certain rules/restrictions on coding methodology which must be adhered to in order to use that tool. From a designer's perspective, RTL is an intermediate stage of the design which eventually will lead to a gate level design.

1.3 Structural Model

A structural model is made up of primitive cells and interconnections between them. In this case the main intention is to get an application specific integrated circuit (ASIC) targeted to the technology of a specific foundry and as a result, all primitive cells used in the model must be defined in the

cell library of that foundry. Most often this type of model is obtained as the output of a synthesizer at the final phase of a design cycle.

2. WHY USE A BFM ?

The usefulness of a BFM goes beyond just proving the algorithm at the initial phase of the design. It can be used in many other situations.

For example, it is quite natural for a system designer to use an ASIC, designed by an outside vendor, for his printed circuit board. Furthermore, he may want to simulate the functionality of the entire board so that any design integration problems can be detected before sending the design for manufacturing. In order to achieve this, a simulation model of the ASIC is required. However, the ASIC vendor may not wish to reveal the RTL level design code to the rest of the world, in order to protect his intellectual property (IP).

Also, in today's fast-paced world, it is possible that the design of the ASIC may not be ready yet for a full fledged simulation and the system has to be designed in parallel with the ASIC to reduce the time-to-market.

To take care of such situations, one option is to write an RTL model in Verilog and then encrypt it before making it available to the external world. One way of achieving this will be to use encryption using *`protect* and *`endprotect* compiler directives of Verilog. Unfortunately, this process is not as reliable as one wishes and there were past instances when hackers deciphered them easily. Also, as stated earlier, a full fledged design of the ASIC may not be available when the model is needed.

Fortunately, this is one situation where Programming Language Interface (PLI) can be used to address all the problems. Furthermore, it improves the overall quality of the solution. The following are the advantages of writing a BFM using PLI.

- Writing a model in C eliminates some of Verilog's inabilities to do certain file I/O operations, such as file read.
- Writing a model in C and then providing the corresponding machine code ensures IP protection. Any checking for license, in case of a proprietary BFM written in C, can also be done much easily.
- Also, many designers find it easier to translate an algorithm from the concept to C, compared to Verilog. This keeps the time required to write a model to the minimum and also ensures the availability of the model to a system designer at the earliest.

– Usually a C-code simulating only the external port behavior of a design takes much less time to run, compared to its corresponding RTL code. This makes the simulation run much faster.

3. GENERAL METHODOLOGY FOR WRITING A BFM

The whole process of writing a BFM in C and compiling it to generate machine code thereafter may seem a little complex at first glance. But all BFMs essentially follow a common structure. This means once the first model is written, writing the next model becomes much easier as the core structure of the code essentially remains the same. A commonly followed methodology for writing a BFM using PLI is described below.

The BFM is defined as a Verilog module which can be instantiated in other parts of the Verilog program. This Verilog module, like any other module, has a port list. However, the module description usually consists of only one `initial` block and sometimes a set of continuous assignment statements to handle the input and output variables. This `initial` block invokes the system call associated with the PLI routine of the BFM passing all the necessary signals (net or reg) as parameters. Inside the PLI routine, output values are calculated as functions of the input values. Later on, these output values are assigned to the output of the Verilog module. The following is a typical example of such a module.

```
module BFM_example(my_inout, my_out, my_in);
inout my_inout;
output my_out;
input my_in;

reg r_my_inout;
reg my_out;

assign my_inout = (my_in==1'b0)?
                  r_my_inout :
                  1'bz;
initial
   $BFM_pli(my_inout, my_out, my_in);
endmodule
```

Note that, apart from the input/output information, other information, such as any local register of the module, can also be passed to the system call.

A more complete example of a Verilog model and its corresponding PLI instance is described in the next section.

4. WRITING A BUS FUNCTIONAL MODEL – A CROSSBAR SWITCH

We will design a small BFM for a basic crossbar switch (such as 74334). Although, probably one does not need to write a BFM to implement this simple module, this will help us to understand the process of writing a BFM.

4.1 Problem description

A crossbar switch has 2 sets of ports- `P0, P1, ... P7` as inputs and `Q0, Q1, ..., Q7` as outputs. Also, there are two input signals `dir` and `dirn`, which control the data flow through the crossbar switch as follows.

Table 31. Truth table for the crossbar switch

dir	dirn	Q
1	Don't care	P
Don't care	0	P
0	1	z

To summarize the contents of Table 1, unless `dir` is 0 and `dirn` is 1, `Q` is same as `P`. Otherwise `Q` is `8'bz`.

4.2 Algorithm for the crossbar

The algorithm for our crossbar switch, as derived from Table 1, is shown below using pseudo-code.

```
whenever ((dirn == 1'b1) and (dir == 1'b0))
    Q = 8'bz;
otherwise
    Q = P;
```

If you recall, this algorithm is very similar to the example given in Chapter 1 with two important differences. First, instead of writing a user-provided value to a register, this time we need to read the value of the port `P` and write it to the port `Q`. More importantly, since the BFM must remain active throughout the duration of the simulation, this routine has to be invoked in *every simulation cycle*, rather than just once.

4.3 Implementation

Following our general methodology stated earlier, we define a wrapper module `xbar_switch` for the model as shown below.

```
module xbar_switch(P, Q, dir, dirn);
input [7:0] P;
wire [7:0] P;

output [7:0] Q;
reg [7:0] Q;

input dir, dirn;
wire dir, dirn;

initial
  $xbar(P, Q, dir, dirn);
endmodule
```

There are many different ways of implementing `$xbar` so that the algorithm of Section 4.2 is executed for each simulation cycle. All of these methods essentially depend on a callback mechanism (See Chapter 2 for discussion on callbacks). We start with the simplest implementation. Other better and faster methods will be discussed in the *Self Check* section.

As we know, the PLI library function `tf_setdelay(delay)` can be used to call misctf function after `delay` number of simulation cycles. The main body of the PLI routine implementing the above algorithm is placed in the misctf routine. At the beginning (i.e. when `$xbar` is called), the calltf function does a callback by using `tf_setdelay(0)`, invoking the misctf function at the same simulation time slot.

```
xbar_calltf() {
   tf_setdelay(0);
```

```
}
```

Once the misctf function is called, it calls itself after unit time delay by using `tf_setdelay(1)` at its end. Also note that a callback due to the function `tf_setdelay()` is associated with a reason `reason_reactivate`.

```
xbar_misctf(data, reason)
int data, reason;
{

switch (reason) {
   case reason_reactivate :
   /* The main algorithm will go here */
   tf_setdelay(1);
   break;
   }
}
```

Here is the PLI routine for the system call `$xbar` implementing the functional model of the crossbar switch. The following program shows the implementation in Verilog-XL environment.

```
#include "veriuser.h"
#include "vxl_veriuser.h"

int xbar_misctf(),
    xbar_calltf(),
    xbar_checktf();

char *veriuser_version_str = "\
***********************************\n\
VERILOG PLI PROGRAM FOR XBAR SWITCH\n\
***********************************\n\
";

int (*endofcompile_routines[])() = {
    /*** no_eoc_routine, ***/
    0 /*** final entry must be 0 ***/
};
```

```
bool err_intercept(level,facility,code)
int level; char *facility; char *code;
{ return(true); }

s_tfcell veriusertfs[] = {
  {usertask, 0, xbar_checktf, 0, xbar_calltf,
   xbar_misctf, "$xbar"},
  {0}
};

int xbar_checktf() {
if (tf_nump() != 4) {
    tf_error("$xbar:Parameter missing");

if ((tf_typep(1) != tf_readonly) ||
    (tf_typep(2) != tf_readwrite) ||
    (tf_typep(3) != tf_readonly) ||
    (tf_typep(4) != tf_readonly))
    tf_error("$xbar: Argument type mismatch");
}

int xbar_calltf() {
   tf_setdelay(0);
}

int xbar_misctf(data, reason)
int data, reason;
{
int direction1, direction2;

switch(reason) {
   case reason_reactivate : {
      direction1 = tf_getp(3);
      direction2 = tf_getp(4);
      if ((direction1==1)||(direction2==0)) {
         if (tf_strdelputp(2,
                           tf_sizep(1),
                           'b',
                           tf_strgetp(1,'b'),
                           0,
```

```
                                0) == 0)
                tf_error("$xbar: ERROR in data transfer");
        } else if ((direction1==0)&&(direction2==1)) {
           if (tf_strdelputp(2,
                                tf_sizep(1),
                                'b',
                                "zzzzzzzz",
                                0,
                                0) == 0)
                tf_error("$xbar: ERROR in data transfer");
        }
        tf_setdelay(1); /*Invoke again in the next clock*/
        break;
    } /* end case */
  } /* end switch */
  }
```

4.4 Results

The following is an example of using the crossbar switch we have just designed. It invokes the wrapper `xbar_module` and passes the signals to it. Notice that it uses a single signal and its negation for the `dir` and `dirn` control signal pair.

```
module my_crossbar;
reg [7:0] b1;
wire [7:0] b2;
reg dir;

xbar_module x1 (b1, b2, dir, ~dir);

initial begin
   $monitor( $time,
             ": b1 = %h, b2 = %h, dir = %b, dirn = %b\n",
             b1, b2, dir, ~dir);
   b1 = 8'haa;
   // some arbitrary value to pass to b2
    dir = 1'b0;
#25 dir = 1'b1;
#25 dir = 1'bz;
```

```
#25 dir = 1'bx;
#25 $finish;
end
endmodule
```

Assuming the above Verilog program is stored in a file `test.v`, a sample run produces the following output.

```
***********************************
VERILOG PLI PROGRAM FOR XBAR SWITCH
***********************************
Compiling source file "test.v"
Highest level modules:
my_crossbar
 0: b1 = aa, b2 = xx, dir = 0, dirn = 1
 1: b1 = aa, b2 = zz, dir = 0, dirn = 1
25: b1 = aa, b2 = zz, dir = 1, dirn = 0
26: b1 = aa, b2 = aa, dir = 1, dirn = 0
50: b1 = aa, b2 = aa, dir = z, dirn = x
75: b1 = aa, b2 = aa, dir = x, dirn = x
L16 "test.v": $finish at simulation time 100
```

4.5 Self check

- As can be seen from the above result, the PLI routine that we have just written does not work with values of z and x for `dir` and `dirn`. Modify the routine to take care of all possible values of these signals.
- Even if none of the signals `b1` or `dir` changes (as in our example above), the PLI routine is active in every cycle slowing down the overall simulation. Instead, a better scheme will be to change the signal `b2` only when there is a change in any of the controlling signals. One way of doing this is to use the routine `tf_asynchon()` together with the `paramvc` argument to check for any change in `b1` or `dir` (see Section 4.2 of Chapter 2 for detailed discussion on reason). The basic program structure for this scheme is shown below.

```
int xbar_calltf() {
   tf_asynchon();
}
```

```
int xbar_misctf(data, reason, paramvc)
int data, reason, paramvc;
{
   switch(reason) {
      case reason_paramvc :
      if (changed_signal(b1) ||
          changed_signal(dir))
         transport_b1_to_b2();
      break;
      /* Other reasons will go here */
   }
}
```

In the above example, `changed_signal(signal)` returns a Boolean value of whether the callback has happened due to a change in `b1` or `dir`. The function `transport_b1_to_b2()` assigns the value of b1 to b2. Design the functions `changed_signal()` and `transport_b1_to_b2()`. Now complete the program by handling other possible values of reasons, such as, `reason_disable`. (We will see this approach in greater detail in the next example.)

- Implement the above scheme using the value change link functions `acc_vcl_add()` (refer to Chapter 4).
- As a variation, try designing a bus functional model of a 8-bit bi-directional transceiver (such as 74245) whose functionality is shown in the following table.

Table 32. Truth table for bi-directional crossbar switch

`dir` (Input)	`P` (Inout)	`Q` (Inout)
0	Don't care	P
1	Q	Don't care
z,x	x	x

5. WRITING A BUS FUNCTIONAL MODEL – A RUDIMENTARY PROCESSOR

Our next example discusses how to write the BFM of a processor using PLI.

5.1 Problem description

Since our main objective is to explore more on how to write a BFM using PLI, rather than going into details of the implementation of a processor architecture, we will simplify the architectural issues by making some assumptions.

For example, we will assume that our processor has only 4 internal general purpose registers – g0, g1, g2 and g3. The following table shows the port list of the processor. In order to keep the design simple, we will not consider signals such as byte enable, Memory/~IO or CACHE at this point.

Table 33. Port list of the processor

Port name	Purpose	Type of Port
`CLK`	Clock	Input
`READY`	Ready signal from the target	Input
`ADS`	Address strobe	Output
`ADDR[15:0]`	Address bus	Output
`DATA[31:0]`	Data bus	Inout
`W_R`	Write or read indicator	Output

Furthermore, we will assume that all our processor can perform are read and write operations, that too both in chunks of 4-bytes only. We are interested in these two operations because, as opposed to the commands bound to CPU core (like add/subtract etc.), these are the ones which interact with the external world. Since, while writing BFM, we are mainly concerned about the processor's interface with the external world (although CPU bound commands can also be taken care of by a BFM), this simplification makes sense.

We specify the operations the processor performs (i.e. its 'assembly' program) in a file. This file follows a simple structure. It starts with a keyword `main:` followed by the operations performed by the processor. Also, the program terminates with the keyword `end` . A simple write of the hex number `deadbeef` at the address `1234` followed by a read from the same location into register `g2` will be as shown below.

```
main:
  write 1234  deadbeef
  read  1234  g2
end
```

5.2 Implementation overview

Once again, following the general methodology for writing a BFM (as outlined in Section 3), we first create a Verilog wrapper module. This module invokes the system call which implements the BFM. This module, like any other Verilog module, can be instantiated in other parts of the design as the top level processor model. The Verilog wrapper is shown below.

```
module proc (ready, ads, addr, w_r, data, clk);
input ready, clk;
output ads, w_r;
output [15:0] addr;
inout [31:0] data;

reg ads, w_r;
reg [15:0] addr;
reg [31:0] r_data;

reg [31:0] g0, g1, g2, g3;
// the general purpose registers

assign data = ((w_r==1'b1)&&(ready==1'b1))?r_data:32'bz;

initial
begin
    $proc(ready, ads, addr, w_r, data, r_data, clk,
          "op1.input");
end
endmodule
```

Here `$proc` is a system call which is invoked at the simulation time = 0 to implement the functionality of the BFM. It reads the input file `op1.input` (containing the assembly code), parses it (i.e. analyzes the meaning of what is written there) and then executes the code sequentially as given in the file.

Note that even though the model needs to access the local registers, only the signals attached to the ports of the module are passed to the system call. Although it is absolutely possible to pass g0, ..., g3 to the system call, conventionally only the handshaking signals (I/O signals of the module) are passed as parameters. All internal variables, such as local registers, are accessed through handles.

The algorithm of the processor model is given below using pseudo-code.

```
open the input file;
while (there is one more command in the file) {
  get the command from the file;
  if (command == "write")
    execute the write sequence;
  else if (command == "read")
    execute the read sequence;
}
close the input file;
```

In the above algorithm, a read or write *sequence* means asserting/de-asserting certain signals according to the protocol of that operation for next several cycles until the operation is finished.

5.3 Forming the State Machine

The timing diagrams of the read and write protocols are shown in Figures 10 and 11 respectively. Although we will discuss each of these protocols separately later, it is clear from the timing diagram that once any of these operations starts, the next event (i.e. change of value of any of the associated signals) is strictly determined by the current state of the operation and/or any change in the inputs.

Or in other words, it is possible to implement these protocols as *state machines*. Also, as shown in the figures, we assume that all state changes occur only at the positive edge of the clock signal CLK. This makes the state machine a *synchronous* one. (In a *synchronous state machine*, all internal state values are updated at the edge of a single clock. *Asynchronous state machines*, on the other hand, do not have such dependency and can change the internal states solely based on the input to that state.)

Based on the algorithm given above, the *state diagram* of the scheme is shown in Figure 9. From Figure 10 and 11 one can see that all output changes happen within the first two positive edges of the clock in both the operations. These are marked as states CYCLE0 and CYCLE1 respectively in Figure 9. After fetching the instruction, in each of these states the processor executes part of the instruction as shown in the timing diagrams.

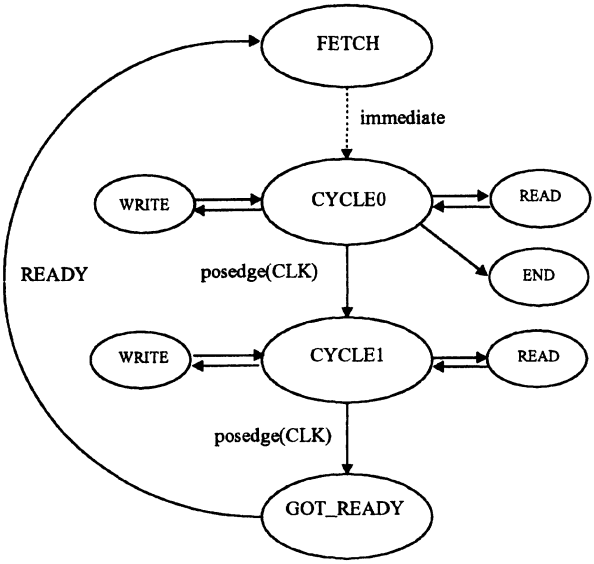

Figure 9. The state diagram for the processor

Also note that:

- The fetch cycle is also the beginning of the execution cycle.
- The processor keeps on looping on the fetch-execute-fetch path of the diagram, unless there is an 'end' in the input file to bring the state machine to the END state.
- The transitions from CYCLE0 to CYCLE1 and from CYCLE1 to GOT_READY occur in two successive clock cycles. After that, in both read or write operations, the state machine waits until the external input READY is asserted.

5.4 Algorithm for the state machine

We now refine our existing algorithm to go into the details of the processor operations as given in the state diagram. This time we will focus on how the BFM works on cycle-by-cycle basis.

open the input_file for reading;

/ First search for the beginning of the program. */*
/ The program starts with the token "main:" */*

```
got_MAIN = false;
while ((got_MAIN == false)
      AND (end_of_file(input_file) == false)) {
  read(the next line from the input file);
  if (error)
    quit;
  else {
    break the line into tokens;
    if (first_token == "main:")
      got_MAIN = TRUE;
  }
}

/* Now start executing the instructions */
cycle = CYCLE0;
in_the_loop = true;

/* The main loop of the state machine*/
while (in_the_loop == true) {
  if (cycle == CYCLE0) {
    /* The instruction from the input file */
    /* is read only at the CYCLE0 and used */
    /* in other states                     */
    read(the next line from the input file);
    if (error) quit;
    break the line into tokens;
    if (the first_token == "end")
      in_the_loop = FALSE;
    else
      execute(CYCLE0, first_token);
    cycle = CYCLE1;
  } else if (cycle == CYCLE1) {
    execute (CYCLE1, first_token);
    cycle = GOT_READY;
  } else {
    while (READY is not asserted)
      wait till the next clock;
    execute(GOT_READY, first_token);
    cycle = CYCLE0;
  }
```

```
    wait(till the next posedge of CLK);
  }

  /* Clean up and close down */
  close(input_file);
```

The salient part of the algorithm is the main while loop. The loop is executed once in each positive edge of the clock CLK. The functionality of the loop is determined by the current state.

Note that, we have used a pseudo-function `execute()` in the above algorithm to assert/deassert the necessary signals depending on the current state and operation. In actual implementation, the functionality of `execute()` will be governed by the read and write protocols for the processor. We will discuss these protocols in the next sections.

5.5 The algorithm for processor write

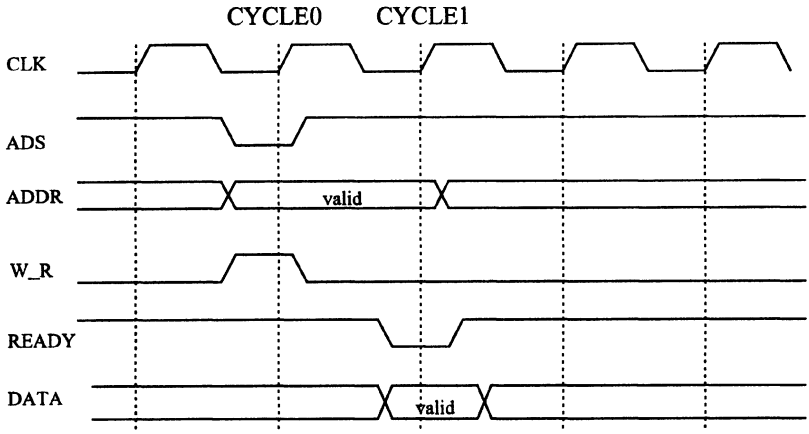

Figure 10. Timing diagram for processor write

Let us now see how a write cycle proceeds in the processor. The timing diagram of a 4-byte simple write is shown in the Figure 10. For simplicity, we will treat rise times of all output signals to be zero. This means that the input signals affecting an output signal are sampled at the positive edge of the clock and the computed value of the output signal is assigned to it immediately. Of course, in practice, the rise times and other timings will not

only be different in each case, but will also change every aspect of the design.

Based on the timing diagram of Figure 10, the algorithm for each cycle is given in Table 4 below.

Table 34. Cycle by cycle description of write operation

Cycle no.	Operation
CYCLE0	Assert ADS = low,
	ADDR = address as obtained from the input file,
	W_R = high,
	DATA = data as obtained from the input file
CYCLE1	Deassert ADS
	Assert READY
Otherwise	Do nothing

5.6 Algorithm for read

Here is the algorithm for the read operation on a cycle by cycle basis.

Table 35. Cycle by cycle description of the read operation

Cycle no.	Operation
CYCLE0	Assert ADS = low,
	ADDR = address as obtained from the input file
	W_R = low
CYCLE1	Deassert ADS
GOT_READY	Wait till READY gets asserted
	Copy the value of DATA bus into the register given in the input file

The timing diagram for the read operation is shown below in Figure 11.

Note that the write operation virtually ended from the processor side as soon as the data was sent. However, in the case of a read, we do not have any control over when we get back the data from the target address. That entirely depends on when the target will send it. The processor waits until the READY is asserted by the target indicating that the DATA bus now carries the read value from the target address. Subsequently, the processor reads this value into the register specified in the input file.

Let us look at the example of the read instruction again.

```
read   1234   g1
```

Since we are not passing the register name as a parameter (it is being read from the file) we can not use `tf_putp` or `tf_strdelputp` to perform

the data assignment to the register. However, since we know the name of the register and its scope (same as that of the system call `$proc` itself) we can get a handle to it. Using this handle, we can modify the value of the register.

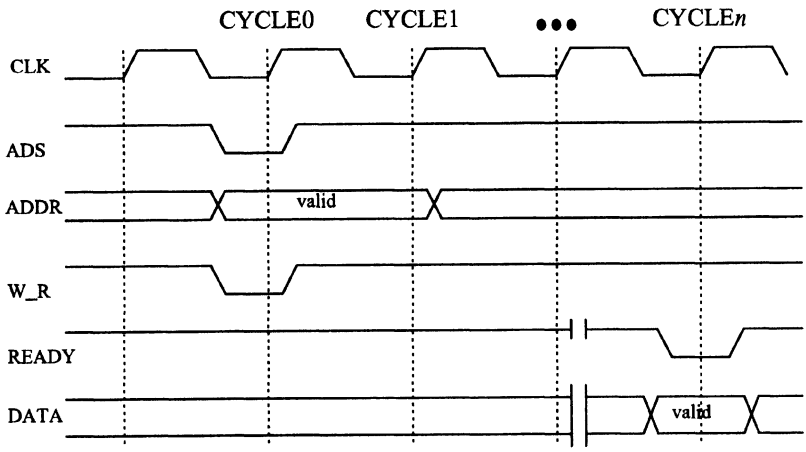

Figure 11. Timing diagram for processor read

5.7 Implementation of the algorithm

It is easy to see that the implementation of the algorithm is primarily based on detecting the positive edge of the clock `CLK` and then switching to the appropriate state.

The most convenient way to do this is to use a callback mechanism, so that whenever there is a positive edge of `CLK`, the necessary functions will be invoked automatically. As discussed in Section 4.5, this can be done by using the function `tf_asynchon()` and then implementing the main loop inside the misctf function. Once `tf_asynchon()` is activated, whenever there is any change in value of any of the parameters to the system call `$proc`, the misctf function is invoked with a reason `reason_paramvc`.

The following program shows the implementation of the state machine in the Verilog-XL environment.

```
#include <stdio.h>
#include <string.h>

#include "veriuser.h"
#include "vxl_veriuser.h"
```

```
#include "acc_user.h"

int   calltf_proc();
int   checktf_proc();
int   misctf_proc();
int   is_posedge();

char  *veriuser_version_str = "\
***********************************\n\
VERILOG PLI PROGRAM FOR PROCESSOR\n\
***********************************\n\
";

int (*endofcompile_routines[]) () ={
   /*** no_eoc_routine, ***/
   0  /*** final entry must be 0 ***/
};

bool err_intercept(level, facility, code)
int  level;
char *facility;
char *code;
{
   return (true);
}

s_tfcell veriusertfs[] = {
  {usertask,
   0,
   checktf_proc,
   0,
   calltf_proc,
   misctf_proc,
   "$proc"},
   {0}
};

/* The parameters are :
  1:   READY
  2:   ADS
```

```
 3:   ADDR
 4:   W_R
 5:   DATA
 6:  R_DATA
 7:   CLK
 8:  FILENAME */

#define READY          1
#define ADS            2
#define ADDR           3
#define W_R            4
#define DATA           5
#define R_DATA         6
#define CLK            7
#define FILENAME       8

/* Check the number and types of the parameter */
int checktf_proc()
{
   acc_initialize();
   if (tf_nump() != 8) {
      tf_error("$proc:Parameter missing\n");
      tf_dostop();
   }
   if ((tf_typep(READY)     != tf_readonly) ||
       (tf_typep(ADS)       != tf_readwrite) ||
       (tf_typep(ADDR)      != tf_readwrite) ||
       (tf_typep(W_R)       != tf_readwrite) ||
       (tf_typep(DATA)      != tf_readonly) ||
       (tf_typep(R_DATA)    != tf_readwrite) ||
       (tf_typep(CLK)       != tf_readonly) ||
       (tf_typep(FILENAME) != tf_string)) {
      tf_error("$proc:Argument type mismatch");
      tf_dostop();
   }
}

#define TRUE          1
#define FALSE         0
#define MAX_LENGTH   80
```

```
/* Calltf routine. Its only job is to initiate the */
/* callback mechanism.                              */
int calltf_proc()
{
   tf_asynchon();
}

#define READ_COMMAND      0
#define WRITE_COMMAND     1
#define INVALID_COMMAND   2

#define CYCLE_ZERO        0
#define CYCLE_ONE         1
#define CYCLE_TWO         2
#define CYCLE_THREE       3

int misctf_proc(data, reason, paramvc)
int data, reason, paramvc;
{
   /* The following variable are used
      for reading the commands */
   static FILE    *fp;
   static char    *line;
   static int     searching_main = TRUE;
   static int     searching_command = TRUE;
   static int     at_least_one_command = FALSE;
   static char    *current_token;
   long           p1, p2;
   /* The following variables are used
      for the execution of the commands */
   static int current_command = INVALID_COMMAND;
   static int cycle = CYCLE_ZERO;
   static int first_time = TRUE;
   static handle reg_handle,
                 parent_handle,
                 tf_handle;
   static s_setval_delay delay_s =
          {{accTime}, accNoDelay};
   static s_setval_value value_s;
```

```
switch (reason) {
case reason_paramvc:{
   if (is_posedge(CLK, paramvc))
   /* Is this a posedge of CLK ? */
      switch (cycle) {
      case CYCLE_ZERO:{
         if (first_time == TRUE) {
            /* The input file has to be opened */
            first_time = FALSE;
            fp = fopen(tf_strgetp(FILENAME), "r");
            line = (char *) malloc(MAX_LENGTH);

            /* find out the word main: */
            while (!feof(fp) &&
                    (searching_main == TRUE)) {
               fgets(line, MAX_LENGTH, fp);
               if (strcmp(strtok(line, "\t "),
                           "main:\n") == 0)
                  searching_main = FALSE;
            }
            if (feof(fp) && searching_main)
               tf_error("$proc: No  'main'\n");
         } /* if (first_time) */

         /* Search for commands starts here */
         fgets(line, MAX_LENGTH, fp);
         current_token = strtok(line, "\t ");
         if (strcmp(current_token, "write") == 0) {
            /* if the next command is a write */
            at_least_one_command = TRUE;
            current_command = WRITE_COMMAND;
            p1 = strtol(strtok(NULL, "\t "),
                        (char **) NULL, 16);
            p2 = strtol(strtok(NULL, "\t\n "),
                        (char **) NULL, 16);
            tf_putp(ADS, 0);
            /* ADS goes low, write cycle starts */
            tf_putp(ADDR, p1);
            /* address is sent out */
            tf_putp(W_R, 1);
```

```
        /* w_r high means it is a write */
        tf_putp(R_DATA, p2);
        /* write data goes out */

        cycle = CYCLE_ONE;
    } else if (strcmp(current_token, "read")==0) {
        at_least_one_command = TRUE;
        current_command = READ_COMMAND;
        p1 = strtol(strtok(NULL, "\t "),
                    (char **) NULL, 16);
        current_token = strtok(NULL, "\t\n ");
        tf_putp(ADS, 0);
        /* ADS goes low, read cycle starts */
        tf_putp(ADDR, p1);
        /* address is sent out */
        tf_putp(W_R, 0);
        /* w_r low means it is a read */

        cycle = CYCLE_ONE;
    } else if (strcmp(current_token,"end\n")==0){
         searching_command = FALSE;
    } else
        tf_error("Invalid command:%s\n",
                 current_token);
break;
} /* CYCLE0 */

case CYCLE_ONE:{
    if (current_command == READ_COMMAND)
        tf_putp(ADS, 1);
        /* ADS goes high-be it read or write */
    else if (current_command == WRITE_COMMAND)
        tf_putp(ADS, 1);
        /* ADS goes high-be it read or write */
    cycle = CYCLE_TWO;
break;
} /* CYCLE1 */

default:
    if (tf_getp(READY)==0) {
```

```
                if (current_command==READ_COMMAND) {
                   value_s.format = accIntVal;
                value_s.value.integer =
                   (int)acc_fetch_tfarg(DATA);
                   /* Get the data from the bus */
                delay_s.model = accNoDelay;
                tf_handle = acc_handle_tfinst();
                /* Get a handle to the system call */
                parent_handle =
                   acc_handle_parent(tf_handle);
                /* Get a handle to the parent module */
                acc_set_scope(parent_handle);
                reg_handle =
                   acc_handle_object(current_token);
                /* Get a handle to the reg */
                acc_set_value(reg_handle,
                              &value_s, &delay_s);
                /* Set the value of the reg */
             }
             current_command = INVALID_COMMAND;
             cycle = CYCLE_ZERO;
          }
       } /* switch on cycle ends here */
       if ((cycle > CYCLE_ZERO) && (feof(fp))) {
          if (searching_command)
             tf_error("$proc: No 'end'\n");
          else if (!at_least_one_command)
             tf_error("$proc: No command\n");
       }
       break;
    } /* reason_paramvc */

    case reason_disable:{
       tf_error("$proc: Disabled calling block\n");
       break;
    } /reason_disable */
    } /* switch */
 }

 /* The function is_posedge() determines the posedge */
```

```
/* of the signal whose position is passed as an     */
/* argument (watched_signal). The other argument is */
/* the signal which has changed. Note that the      */
/* previous value of only one signal is preserved.  */
/* So in the entire program, this function can be   */
/* used only for one signal. For multiple signals,  */
/* modify the prev_value variable to an array to    */
/* store previous values of all signals.            */

int is_posedge(watched_signal, changed_signal)
int watched_signal, changed_signal;
{
static int prev_value;
int present_value;

present_value = tf_getp(watched_signal);
if (watched_signal == changed_signal)
   if ((prev_value==0)&&(present_value==1)) {
       prev_value = present_value;
       return (TRUE);
   } else {
       prev_value = present_value;
       return (FALSE);
   }
else
   return (FALSE);
}
```

5.8 Discussion of the code

5.8.1 Calling misctf routine

In this example calltf routine has very little to do, except starting the callback mechanism for the misctf routine using `tf_asynchon()`.

5.8.2 Declaring variables in misctf

Many local variables inside misctf are declared as `static`. Declaring them as `static` ensures that these variables will retain their current values during a future invocation of the misctf. (Another less elegant way of

achieving this will be to declare all of them as global variables. But considering the fact that all of the constituting functions in our program may actually be in different files, use of static declaration is highly recommended).

5.8.3 Reasons

As noted earlier (Chapter 2 Section 4.2), the parameter *reason* tells us the reason for current callback. In our case, the code corresponding to the state machine is executed only if the reason is `reason_paramvc`, the value associated with a parameter value change callback.

The function `is_posedge(CLK, paramvc)` determines whether the current callback is due to a positive edge of the `CLK` signal. The state machine makes a state transition only if the current callback is due to a positive edge of `CLK`.

Although the main loop is executed only if the reason is `reason_paramvc`, we have used another reason (`reason_disable`) as a precaution. This will be activated when the system task is disabled.

5.8.4 Reading the input file

As shown earlier, the instructions in the input file are enclosed within the keyword pair `main:` and `end`. When the input file is opened for the first time, the program starts parsing the input file to find out the keyword `main:`. After this, it reads each instruction between `main:` and `end` and executes them sequentially. Note that an instruction is read only at the `CYCLE0` and used for all other states of the state machine until the execution is complete.

The program also takes care of sanity checking for a program which violates the grammar rule of our small language (e.g. no `main:`, no `end`, no instruction etc.)

The function `strtok(main_string, delim_string)`, defined in standard library `string.h`, breaks down the `main_string` into tokens using the characters of `delim_string` as delimiters. When called for the first time, it returns the first token. Each subsequent token of `main_string` are obtained by invoking `strtok(NULL, delim_string)`. For example, once the token `write` is found, the address and data are retrieved by two subsequent calls to `strtok()`.

For complex models, often the input language must have more high level features, such as if-then-else or for statements. Also, often these languages are defined as free-form (meaning white space characters have no

significance; for example, C is a free form language). This, however, makes interpretation on a line-by-line basis very difficult. A more efficient way of handling this sort of situation is to define a formal grammar for the language and developing functions capable of lexical analysis and parsing. Usually these functions are generated using standard tools, such as, lex and YACC, but details of that methodology is beyond the scope of this book.

5.8.5 Assigning a value to a named register

In order to get a handle to the named register, first we get a handle to the current instance of the system call `$proc` using `acc_handle_tfinst()`. A handle to the Verilog wrapper module is obtained using `acc_handle_parent()` and passing the handle to the current system call instance as the argument. Since we know that the local registers are instantiated within the Verilog wrapper module, we set the scope same as that of the system call (and the registers) using `acc_set_scope()` with the handle to the parent module as the argument. Next a handle to the named register is obtained by calling `acc_handle_object(reg_name)`, where `reg_name` is the string containing the register name. Once we get the handle, its value is changed by the routine `acc_set_value()`.

5.8.6 Future expansion of the instruction set

Our current instruction set contains only two instructions: read and write. However, the current program can be easily expanded for other commands. For example in CYCLE_ZERO, additional branching can be done using more `else if` statements. Aa an example, to include a command "add" to the instruction set we need to add the following `else if` branch

```
else if (strcmp(current_token,
                "add") == 0) {
...
}
```

5.9 Self check

- Write a test bench for testing this model. You need to have a top level module (let us call it "*system*") which will instantiate two modules - our own `proc` BFM wrapper shown earlier this chapter and a `slave` to service the requests that the processor generates. The slave will be

another state machine, just like the processor itself. Make sure the slave services the processor requests for read or write only after a delay and READY should be asserted a couple of clocks later. Also, note that the DATA bus should not experience any contention at the top most system level with both the `proc` and the `slave` driving it simultaneously. You may want to construct the basic structure of the slave first in Verilog before moving onto PLI.

- A next step to increase the functionality of our processor will be to enable its interrupt handling capability. For this we need to add an extra pin INTR to the `proc` module. Also, we should specify a separate routine, like `main:`, in the assembly code, where the program control will jump to once INTR is asserted. A typical interrupt subroutine may look like as shown below, storing the present context in pre-defined locations.

```
interrupt:
  write 2000 L0
  write 3000 L1
  write 4000 L2
  write 5000 L3
end
```

Add an interrupt mechanism to the BFM of our processor. Remember, interrupt is an asynchronous event having higher priority than any read or write. So, it should be always serviced first, even if INTR gets asserted during a read or write operation.

- This exercise deals with another aspect of designing a BFM with PLI. Apart from the ASICs, modeling a bus protocol is another place where PLI is used extensively. Quite often, a bus protocol is standardized much before the design of the logic circuit which implements it (PCI or FutureBus being prime examples). A bus master/slave model for read/write gives the designer of the system early information about the performance of the proposed system. Write a BFM of any bus, such as, PCI Master read/write. Do not forget to take special care about any turn-around cycle. You can then pick up any other bus (SCSI, IDE, etc.) and write a BFM for a bridge between this bus and the previous one.

6. SUMMARY

In this chapter, we have applied our knowledge of PLI that we gathered over last few chapters into building bus functional models. We took two examples – a crossbar switch and a basic processor. We saw the basic steps of building a functional model from the example of the crossbar switch. The processor model showed us how to write a BFM when its functionality was entirely controlled by an external program. We also found, in spite of being so much different in functionalities, their essential constructs were more or less the same. This is true for any kind of bus functional model.

From next chapter we will go into the details of how to write PLI routines using newly evolved PLI 2.0. This will be exciting as we will discover how it simplifies different programming aspects.

Chapter 7

VERILOG PROCEDURAL INTERFACE (PLI2.0)

This chapter introduces the Verilog Procedural Interface (VPI) machanism of Verilog PLI. VPI is the first major revision of the existing PLI methodology and is now a part of IEEE Std. 1364-1995. The primary objective of VPI is to introduce a consistent interface throughout the domain of the language. This chapter starts with investigating why a new version was required. To illustrate how to use this new interface, we will write a small example program using VPI. Although the individual components of a program written using this new interface are conceptually similar to that of PLI1.0, they differ in syntax. Perhaps the most visible difference between the two methodologies is the way they handle the data access. VPI follows an object oriented approach for the data access associating a set of properties and relationships with each design object. Our discussion in this chapter concludes with a list of such properties and relationships for each defined object.

1. WHY ANOTHER INTERFACE ?

Verilog Procedural Interface (VPI), also commonly called PLI2.0, introduces the third generation of PLI library functions (after `tf_` and `acc_` functions). The main reason for its introduction is to remove the inconsistencies and incompleteness that one encounters in both tf and acc routines in PLI1.0. For example, although you can read the value of a net or modify the value of a register using tf or acc routines, you can not use these routines to read the condition that makes a *while* loop to terminate. VPI

S. Mittra, *Principles of Verilog PLI*
© Kluwer Academic Publishers 1999

removes these constraints and makes a uniform and consistent interface to almost all objects within the design database.

However, unlike utility and access routines, VPI is not just a set of library functions. It has been designed to be independent of the tf or acc interface. It not only covers the functionalities provided by these two sets, but also views the interface mechanism in a different way than both the other sets do as part of PLI1.0. For example, while the interface routines and system calls are defined through a structure `veriusertfs` at the beginning of the program in PLI1.0, the corresponding routines and calls are registered using a separate function in PLI2.0, making dynamic linking of new system calls possible.

Also, the library functions supplied with PLI2.0 are more generic in nature. Each of these functions is capable of handling a wide range of argument types and as a result of this consistency, the total number of functions is greatly reduced (27, compared to 65 basic utility routines alone in PLI1.0).

Functions in PLI1.0 and PLI2.0 are independent of each other and can co-exist in the same user application.

2. ANATOMY OF A PLI ROUTINE USING VPI

There are three main components of a user-defined routine using PLI2.0. They are (recommended, but not necessarily in order)

1. User C functions, also known as *applications*.
2. A C-function to register the system call(s), conventionally called `registration` and
3. A start-up array `vpi_register_systf[]`.

2.1 User C functions

The user-defined C functions constitute the main body of a PLI routine. These are the functions that you write in order to implement the functionality of the corresponding system call. If a user-defined function uses any of the library functions defined in VPI, it must include a header file called `vpi_user.h` at the beginning of the program. This is required in order to access the necessary data structures.

Similar to PLI1.0, VPI requires that each user-defined system call must consist of a set of pre-specified but user-defined functions, such as, calltf, misctf etc., where each of them has a specific task assigned to them.

However, as we will se below the tasks of these functions in some cases are little different in VPI compared to PLI1.0. Remember that these are the generic names of the functions; the actual name is completely user-defined.

2.1.1 sizetf

If the user-defined system call returns a value (i.e. if it is a function rather than a task), sizetf function returns an integer which is same as the number of bits in the returned value of the system call.

2.1.2 calltf

This is the main function which implements the functionality of the system call. Its purpose is same as that of calltf function of PLI1.0. Note that, there is no checktf routine in VPI and the validity of the parameter numbers and types, if needed, must be checked here.

2.1.3 compiletf

This function is executed when the compilation of the Verilog program is completed and/or at the beginning of the simulation. A possible use of this function is to initialize a set of variables which can be used later during the entire simulation. This function does the equivalent of a callback in PLI1.0 with reason value `reason_endofcompile`.

2.2 The registration function

A user-defined function `registration` (or any other name) is used for registering the associated functions and other properties of a system call. This is done in two parts. First, the names of the associated functions and other properties of the system call are assigned to different fields of a structure of type `s_vpi_systf_data`. Next, the existence of the system call is made known to the simulator, using the library function `vpi_register_systf()`.

The different fields of the structure `s_vpi_systf_data`, defined in `vpi_user.h`, are shown below.

```
typedef struct t_vpi_systf_data {
   int type;
      /*
```

```
        vpiSysTask, if returning void
        vpiSysFunction, Otherwise
       */
    int sysfunctype;
       /*
        If (type==vpiSysFunction) then
         vpiSysFunctionInt, if returning integer
         vpiSysFunctionReal, if returning real
         vpiSysFunctionTime, if returning time
         vpiSysFunctionSized, if returning bits
       */
    char *tfname;
       /*
        The "$"-prefixed name of the system call
       */
    int (*calltf)();
       /* The calltf function name */
    int (*compiletf)();
       /* The compiletf function name */
    int (*sizetf)();
       /*
        The sizetf function name. Valid only
        if type == vpiSysFunction
       */
    char *user_data;
       /*
        User data supplied to the constituent
        function for each callback.
       */
}s_vpi_systf_data, *p_vpi_systf_data;
```

The user-defined function `registration` should allocate the memory required for the structure. It should also assign proper values to its fields. In case a particular function is not required (say for example, compiletf), a NULL pointer is placed in the corresponding place. For more than one system call, there will be as many such structures as the number of system calls. Alternatively, there may be more than one registration functions.

Also, since the registration is being done at the run-time, it may be done anytime during the simulation, not only at the beginning. It is even possible to have conditional registration depending on a particular scenario.

At the end, the function registers the system call using the function `vpi_register_systf()`, which takes the corresponding structure as its argument.

A registration function returns void type.

2.3 The start-up array `vlog_startup_routines[]`

The start-up array `vlog_startup_routines[]` lists all the registration functions and presents it to the simulator. More precisely, it is an array of pointers to functions returning void types. Remember that *no* other name may be used for this array, as the simulator will not understand it. The last element in this array should be zero. Its functionality is equivalent to that of `veriusertfs` array in PLI1.0.

2.4 A simple example – `$add_int()`

The following example illustrates a simple PLI routine `$add_int()`, which takes two integers and prints their sum.

```
#include <stdio.h>
#include "vpi_user.h"

/****************************************
 * The user C function add_int_calltf()
 ****************************************/
int add_int_calltf();
int add_int_calltf() {
int n1, n2;
   printf("Enter the first integer:");
   scanf("%d", &n1);
   printf("Enter the second integer:");
   scanf("%d", &n2);
   printf("The sum of two integers: %d\n", n1+n2);
}

/****************************************
 * The registration function
 ****************************************/
void registration();
void registration()
```

```
/* Place all the function registration here */
{
s_vpi_systf_data task_data_s ;
/* Declare the structure */

task_data_s.type = vpiSysTask;
/* No return value for our system call */
task_data_s.tfname = "$add_int";
/* HDL task name */
task_data_s.calltf
   = add_int_calltf;
/* The calltf routine */
task_data_s.compiletf = NULL;
/* No function, enter NULL */

vpi_register_systf(&task_data_s);
/* Registering the task */
}

/****************************************
 * The start-up array, the last element
 * must be a zero.
 ****************************************/
void (*vlog_startup_routines[])() =
   {registration, 0};
```

2.5 Integrating user application with Verilog simulator

The methodology for integrating user-defined applications using VPI with the Verilog simulator is the same as what has been described in Chapter 2, Section 6.1. In Verilog-XL, for example, you must run the program *vconfig* to generate a shell script. This script, in turn, generates the compiled binary.

3. DATA ACCESS METHODOLOGY

The basic methodology of a data access in VPI is significantly different from that in PLI1.0. Although accessing the simple properties of an object,

such as its numeric or string value, remains straightforward, VPI introduces a new methodology to traverse the relationship within the design database.

We will start with the basic data structure for all data access – a *handle*. Before a handle is used, it has to be declared and then properly initialized so that it points to the object of interest. First, we will see how to declare a handle. After that, we will learn the mechanism of traversing one-to-one and one-to-many relationships in the Verilog design database.

3.1 Handle and its declaration

In VPI, in order to access the properties of an object, you must set up a handle to that object. A handle can be thought of as an equivalent of a C language pointer, pointing to the data structure of the object. It also provides a way to use indirect addressing. All references to that object, such as getting its value, traversing a relationship etc., must be done using its handle. Whenever an object is required to be passed to a function, its handle is passed as an argument.

The datatype of a handle is defined in `vpi_user.h` as `vpiHandle`. Note that, the datatype remains the same irrespective of the nature of the object (module, net or while loop) the handle is pointing to. The following is an example of handle declaration.

```
vpiHandle netHandle,
          modHandle,
          some_Handle,
          yet_another_Handle;
```

Remember, `vpiHandle` behaves like any other datatype and all variations of a variable declaration in C can be used here too. For example, you might declare an array of handles, initialize the variable at the place of declaration etc.

3.2 Associating a handle to an object

Once a handle is declared, the next task is to associate the handle with the object of our interest i.e., the handle should point to the required object. This can be done in two ways.

First, if we know the name of the object and the handle to its scope, it is possible to obtain a handle to the object using the library function `vpi_handle_by_name()`. For example, in the following code snippet,

`my_handle` will point to the register `my_reg`, if `scope_handle` is a handle to its scope.

```
handle my_handle;
my_handle = vpi_handle_by_name("my_reg", scope_handle);
```

However, more often we need to find out a handle to an object which is in some way related to a given object. This given object could be, for example, the top level module, or an argument to the system call, etc. The data access methodology in VPI has been designed keeping this particular aspect in mind. It has well-defined procedures for traversing the relationship between different objects inside the design database. If a handle is available to an object, it is possible to get a handle to any other object related to it. For example, if a handle is available to a net or a register, it is possible to retrieve a handle pointing to its parent module instance.

At first look, this may seem to be a non-working proposition, as one always needs a properly assigned handle to start with. However, it is also possible to get a handle to some of the 'basic' objects, such as, the top level module or the parameters passed to the system call, without any handle to start with. These handles, in turn, may be used to get handles pointing to other objects related to them.

Before going into details of these, first let us see how to traverse a relationship among objects.

3.2.1 Relationship between two objects

The relationship between any two objects in the design database can be one of the following four types.

- not related (although, it is highly unlikely that any two objects within a single partition of the same design will be unrelated to each other.)
- one-to-one.
- one-to-many.
- many-to-one.

VPI handles the one-to-one and the one-to-many relationships in different ways. We discuss each of them separately below. The many-to-one relation is important for only a specific case and will be discussed along with the library function `vpi_handle_multi()`.

3.2.1.1 One-to-one relationship

An example of two objects having one-to-one relationship is a register and its parent module instance. The relationship is a one-to-one, because, there can be only one module containing that particular register.

When two objects have a one-to-one relationship between them, a handle to the second object can be obtained using the library function `vpi_handle()` and the handle to the first object. The function `vpi_handle()` takes the type of relationship (as a pre-defined integer constant) and the handle to the first object as its arguments and returns the handle to the second object.

The function `parent_module_of_reg()` shown below illustrates the use of `vpi_handle()`. Using our previous example, it obtains the handle `moduleHandle` to the module instance using a handle `regHandle` to the register. Here the type of this one-to-one relationship between the objects specified by `regHandle` and `moduleHandle` is `vpiModule`. (Details of the function `vpi_handle()` will be discussed in Chapter 8).

```
parent_module_of_a_reg(regHandle)
vpiHandle regHandle;
{
vpiHandle moduleHandle;

moduleHandle = vpi_handle(vpiModule, regHandle);
/* moduleHandle can be used here for other purposes */
}
```

In general, while traversing a one-to-one relationship, given the handle to the starting object `handleStart`, the handle `handleDest` to the destination object, whose symbol is `Obj`, can be obtained by

```
handleDest = vpi_handle(vpiObj, handleStart);
```

In other words, the first parameter to the function is the symbol for the destination object with a prefix `vpi`. The second parameter is a handle to the starting object.

3.2.1.2 One-to-many relationship

Although a register can have only one parent module instance, the same module instance, however, can have more than one register within it. Accordingly, the relationship between a module and the registers (or nets)

within it, is a one-to-many relationship. Other examples of one-to-many relationship are a port and its bits, a module and primitive instances within it, etc.

Two library functions `vpi_iterate()` and `vpi_scan()` are used together to traverse a one-to-many relationship. The function `vpi_iterate()` takes the type of one-to-many relationship to traverse and a handle to a reference object as its arguments. It returns an *iterator* handle. This iterator handle is used by `vpi_scan()` as its argument in a loop to return a handle to the next object related to the reference object. When all such objects are traversed, `vpi_scan()` returns a NULL. (The details of functions `vpi_iterate()` and `vpi_scan()` will be discussed Chapter 8)

The following example shows how to use `vpi_iterate()` and `vpi_scan()` to traverse one-to-many relationship among a module and all the registers that it contains. The function `regs_in_a_module()` takes `moduleHandle`, the handle to the module, as its argument and uses it to find out all the registers within the module in a while loop. The loop terminates when all registers within the module are covered. If there is no register, the function returns -1; otherwise it returns 0;

```
int regs_in_a_module(moduleHandle)
vpiHandle moduleHandle;
{
   vpiHandle regItr, regHandle;

   regItr = vpi_iterate(vpiReg, moduleHandle);
   /* regItr is the iterator handle to all
      registers inside this module */
   if (!regItr) {
      printf("No reg inside module\n");
      return -1;
   } else {
      /*In each iteration the following while loop
        returns a handle to the next register
        instantiated within the module */
      while (regHandle = vpi_scan(regItr)) {
         /* Use regHandle here to process
            anything related to the register */
      }
   }
   return 0;
```

```
}
```

In general, while traversing a one-to-many relationship, given the handle to the starting object `handleStart`, the iterator handle `itrDest` to the destination objects with `Obj`, is obtained by

```
itrDest = vpi_iterate(vpiObj, handleStart);
```

In other words, the first parameter to the function is the symbol for the destination object with a prefix `vpi`. The second parameter is a handle to the starting object.

Later on, the handle to each individual object can be obtained by using `vpi_scan()`.

```
while (handleDest=vpi_scan(itrDest))
{ /* process handleDest */ }
```

3.2.2 Class and method

Sometimes objects of similar nature are grouped together to form a *class*. For example, object types gate, switch and UDPs are clubbed together to form a class named *primitive*.

A class is accessed using a *method* from a given object or another class. To see how to get a handle to a member of a class starting from an object, let us take the example of the class *regs*. The class *regs* comprises of register and individual bit of a register array. Clearly, the relationship between a register bit and its parent array of registers (i.e. the array which contains this register bit) is one-to-one. In order to obtain a handle to the register array from the handle to the register bit, we use

```
reg_array_handle = vpi_handle(vpiParent, reg_bit_handle);
```

Here, the method `vpiParent` is used to get the handle `reg_array_handle` to the parent object within the class regs. Note that, unlike a type (such as `vpiModule` used in a similar example earlier), a method is applicable to a general class of objects and does not depend on the type of the returned handle.

We take the inverse example of the above to demonstrate the use of a class and method for obtaining handle in a one-to-many relationship. As obvious, the relationship between a register array and a register bit within

that array is one-to-many. The following example shows how to get an iterator handle to the bits using the handle to the register array.

```
reg_bit_itr = vpi_iterate(vpiBit, reg_array_handle);
```

Also, the returned handle always points to an object; *not* to the class.

In general, the handle to the destination object in the class is obtained by using the method name as the first argument to `vpi_handle()` or `vpi_iterate()` and the handle to the starting object as the second argument. A method name always starts with `vpi`.

3.2.3 Tagged relationship

In order to specify a special meaning of a relationship, sometimes a predefined keyword – different from the object type or method – is used as the first argument for the `vpi_handle()` or `vpi_iterate()`. Commonly termed as *tag*, such a keyword allows accessing properties of the destination object (or class) of a specific type. As an example, an object of type memory has two objects of type expression associated with it – its left and right ranges. These expressions can be accessed by using the tags `vpiLeftRange` and `vpiRightRange` respectively.

3.3 Getting handles to 'basic' objects

So far, we have discussed how to retrieve a handle to an object when the handle to a related object is given. But, in order to do this, we need a handle to start with. The following cases show how a handle can be obtained to some of the most commonly referenced objects (we will call them the 'basic' objects) without taking the help of any other handle. These objects (or, rather their corresponding handles) provide a starting point for traversing a relationship.

3.3.1 Getting handles to the top level modules

An iterator handle to the top level modules of the design can be obtained by using `vpi_iterate()` using `vpiModule` as type and `NULL` as reference arguments as shown below. Once the handle to a top module is obtained, it can be used further for getting handles to other object or properties related to it. The following function `get_top_mods()` shows how to get the name of the top module(s) using `vpi_get_str()` library function.

```
int get_top_mods() {
   vpiHandle top_mod_iterator,
             top_mod_handle;

   top_mod_iterator = vpi_iterate(vpiModule,NULL);
   if (!top_mod_iterator) {
      printf("No top-level module found\n");
      return -1;
   } else
      while (top_mod_handle = vpi_scan(top_mod_iterator))
         printf("Top module: %s\n",
                vpi_get_str(vpiName, top_mod_handle));
   return 0;
}
```

3.3.2 Getting handle to the parameters passed to the current system call

In almost all cases, a PLI application requires to check the values and other properties of the parameters passed to the current system call. In order to get the properties associated with a parameter, we must first get a handle to it. This is done in two steps. First, a handle is assigned to point to the current instance of the system call itself. Then an iterator to the parameter list is obtained using the function `vpi_iterate()` passing a type `vpiArgument` and the handle to the system call as its arguments. The function `vpi_scan()` uses this iterator and returns the handle to the first parameter in the argument list. Handles to successive parameters can be obtained by calling `vpi_scan()` iteratively in a loop until all the parameters are covered. Once the handle to a parameter is obtained, it can be used further to obtain handle to any object or any other property related to the parameter. As an example, the following `list_params()` shows how to get the instance names of the parameters along with its PIN (position of the parameter in the argument list).

```
int list_param()
{
   vpiHandle tf_param_iterator,
             tf_param_handle;
   vpiHandle syscall_handle;
   int pin = 1;
```

```
    syscall_handle = vpi_handle(vpiSysTfCall, NULL);
    if (!syscall_handle) {
       printf("No handle to the system call ...\n");
       return -1;
    } else {
       tf_param_iterator =
          vpi_iterate(vpiArgument, syscall_handle);
       if (!tf_param_iterator) {
          printf("No arg. to the system call\n");
          return -1;
       } else
          while (tf_param_handle =
                 vpi_scan(tf_param_iterator))
             printf("Parameter %d: %s\n", pin++,
                   vpi_get_str(vpiName, tf_param_handle));
    return 0;
}
```

4. DESIGN OBJECTS – PROPERTIES AND RELATIONSHIPS

There are primarily two reasons why one may want to access an object in Verilog source description from a PLI routine. First, the system call may want to read/modify a certain property associated with the object. And second, we may need to access another object which is related to the current object. Accordingly, for each type of object or class in Verilog, *the data access interface can be defined in terms of the properties associated with that object or class and the relations it has with other objects/classes.*

Properties: The properties of an object or a class can be primarily divided into three categories depending on their types.

1. Boolean and integer (both have integer values).
2. String.
3. Complex, having different data types depending on the property and the object.

The *boolean* and *integer* properties can be obtained using `vpi_get()` function. The string properties are retrieved by using the function `vpi_str_get()`. In each case, the property name and the handle to the current object are passed as arguments.

In the following tables, unless otherwise mentioned, all integer and boolean types can be obtained using the function `vpi_get()` and the string types can be retrieved by the function `vpi_str_get()`. For example, to obtain the default net type inside the current module, one should use

```
int currNetType;
currNetType = vpi_get(vpiDefNetType, modHandle);
```

where `modHandle` is the handle to the current module. Since, `vpiDefNetType` is an integer property, the function `vpi_get()` has been used.

In case of *complex* properties, separate functions have been created to cater to the need of the individual case. In such cases, the name of the property has been kept blank in the table and the name of the required function has been mentioned under the column *Function to use*. As an example, the module instance port delay (MIPD), a complex property for a port, can be obtained using the function `vpi_get_delays()`. These functions will be discussed separately in Chapter 8.

Relationships: As we found earlier, to access an object related to a given object, we need to use `vpi_handle()` or `vpi_iterate()`, depending on whether the relationship is one-to-one or one-to-many. We also need to know the type of the destination object (or, a tag or, in case of a class, the appropriate method) to determine the first parameter of the function. Except for a few exceptions, the second parameter is always a handle to the starting object.

For the rest of this chapter, we are going to take each type of object and class defined in Verilog HDL and describe the properties and relations associated with it. We will discuss the properties and the relations associated with an object or a class using a tabular format[3]. In case of a class, we will also see what are its constituent objects and any relationship among themselves.

In the following sections, the type/method for each object/class is mentioned in parentheses along with its name in the header. For example, object Module is associated with a type `vpiModule`. Unless mentioned otherwise (i.e., when it is not a tagged relationship), this type or method should be used as the first argument of the `vpi_handle()` or

[3] Sometimes, this is done using, what is known as, a *data model diagram*. Unfortunately, many users find data model diagrams difficult to understand. Also, in many cases, they do not fully depict the scenario.

`vpi_iterate()` when this object/class is the destination for traversing a relationship.

Similarly, for a class, the types/methods for its constituent objects/classes are also mentioned. In case of certain classes, the method for the class is not available (as it is not required), and constituting objects/classes can be accessed directly.

Also note that relationships are not repeated in the following tables. For example, since the relationships between module and port is mentioned while discussing module, they are not repeated under port.

4.1 Module (`vpiModule`)

4.1.1 Category

Object.

4.1.2 Properties

Table 36. Properties of an object of type *module*

Explanation	Name of the property	Data type of the property	Function to use
Is the module a cell?	vpiCellInstance	boolean	
Decay time for trireg net	vpiDefDecayTime	integer	
Default net type; should be one of the following: vpiWire, vpiWand, vpiWor, vpiTri, vpiTri0, vpiTri1, vpiTriReg, vpiTriAnd, vpiTriOr, vpiSupply1, vpiSupply0	vpiDefNetType	integer	
Definition location by line number	vpiDefLineNo	integer	
Definition location by file name	vpiDefFile	string	
Definition name	vpiDefName	string	
Delay mode; should be one of these:	vpiDefDelayMode	integer	

Explanation	Name of the property	Data type of the property	Function to use
vpiDelayModeNone, vpiDelayModePath, vpiDelayModeDistrib, vpiDelayModeUnit, vpiDelayModeZero, vpiDelayModeMTM			
Instance location by line number	vpiLineNo	integer	
Instance location by file name	vpiFile	string	
Instance name	vpiName	string	
Full pathname for the instance	vpiFullName	string	
Is this module source protected ?	vpiProtected	boolean	
Time precision inside the module	vpiTimePrecision	integer	
Time unit inside the module	vpiTimeUnit	integer	
Is this a top level module ?	vpiTopModule	boolean	
Unconnected port drive strength; should be one of these: vpiHighZ, vpiPull1, vpiPull0	vpiUnconnDrive	integer	

To get the smallest time precision of all instantiated modules, pass a `NULL` handle with types `vpiTimePrecision` or `vpiTimeUnit` to `vpi_get()`.

4.1.3 Relationships

Table 37. Objects related to *module*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Module	Scope	One-to-many (vpiInternalScope)
Scope	Module	One-to-one
Module	Port	One-to-many
Port	Module	One-to-one
Module	Net	One-to-many
Net	Module	One-to-one
Module	Register	One-to-many
Register	Module	One-to-one

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Module	Variable	One-to-many
Variable	Module	One-to-one
Module	Memory	One-to-many
Memory	Module	One-to-one
Module	Named event	One-to-many
Named event	Module	One-to-one
Module	Process	One-to-many
Process	Module	One-to-one
Module	Cont assign	One-to-many
Cont assign	Module	One-to-one
Module	(Sub-)Module	One-to-many
(Sub-)Module	Module	One-to-one
Module	Primitive	One-to-many
Primitive	Module	One-to-one
Module	Mod Path	One-to-many
Mod Path	Module	One-to-one
Module	Timing check	One-to-many
Timing check	Module	One-to-one
Module	Parameter	One-to-many
Parameter	Module	One-to-one
Module	SpecParam	One-to-many
SpecParam	Module	One-to-one
Module	DefParam	One-to-many
DefParam	Module	One-to-one
Module	Param assign	One-to-many
Param assign	Module	One-to-one
Module	IO declaration	One-to-many
IO declaration	Module	One-to-one

For accessing top level modules, use `vpi_iterate()` with second argument as `NULL`.

4.2 Scope (`vpiScope`)

4.2.1 Category

Class.

4.2.2 Constituent objects/classes

- Module (`vpiModule`).
- Task and function.
- Named begin block (`vpiNamedBegin`).
- Named fork block (`vpiNamedFork`).

4.2.3 Relationship among the constituent objects

None.

4.2.4 Properties

Table 38. Properties of the class *Scope*

Explanation	Name of the property	Data type of the property	Function to use
Name of the task or the function	vpiName	string	
Full pathname of the task or the function	vpiFullName	string	

4.2.5 Relationships

Table 39. Objects related to *scope*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Scope	Statement	One-to-one
Statement	Task and Function	One-to-one
Named Begin, Named Fork	Statement	One-to-many
Scope	Named event	One-to-many
Named event	Scope	One-to-one
Scope	Variables	One-to-many
Variables	Scope	One-to-one
Scope	Reg	One-to-many
Reg	Scope	One-to-one
Scope	DefParam	One-to-many
DefParam	Scope	One-to-one
Scope	Memory	One-to-many
Memory	Scope	One-to-one
Scope	Scope	One-to-many (vpiInternalScope)
Scope	(upper level) Scope	One-to-one

4.3 Task and Function

4.3.1 Category

Class.

4.3.2 Constituent objects

- Task (`vpiTask`).
- Function (`vpiFunction`).

4.3.3 Relationships between the constituent objects

None.

4.3.4 Properties

Apart from the properties shown below, the class *Task and Function* also inherits the properties of the class *Scope* as its member.

Table 40. Properties of the class *Task* and *Function*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.3.5 Relationships

Table 41. Objects related to *task and function*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Task and function	Parameter	One-to-many
Parameter	Task and function	One-to-one
Task and function	UDP definition	One-to-one
UDP definition	Task and function	One-to-one
Task and function	Module	One-to-one
Module	Task and function	One-to-one

4.4 Task and Function call

4.4.1 Category

Class.

4.4.2 Constituent objects

- Task call (`vpiTaskCall`).
- Function call (`vpiFuncCall`).
- System task call (`vpiSysTaskCall`).
- System function call (`vpiSysFuncCall`).

4.4.3 Relationships among the constituent objects

None.

4.4.4 Properties

Table 42. Properties of the class *task and function call.*

Explanation	Name of the property	Data type of the property	Function to use
Task/Function name	vpiName	string	
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
The following property is only valid for System task/function call			
Is this system call user defined	vpiUserdefn	boolean	
The following properties are only valid for System function call			
Reading the current value of the system function	-	-	vpi_get_value()
Modifying the current value of the system function	-	-	vpi_put_value()
Type of the system function	vpiSysFuncType	integer	

4.4.5 Relationships

Table 43. Objects related to *task and function call*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Task call	Task	One-to-one
Function call	Function	One-to-one
Task and Function call	Expression	One-to-many

4.5 User-defined system call (`vpiUserSystf`)

4.5.1 Category

Object.

4.5.2 Property

Table 44. Property of an object of type *user-defined system call*

Explanation	Name of the property	Data type of the property	Function to use
System call information	-	-	vpi_get_systf_info()

4.5.3 Relationships

Table 45. Objects related to *user-defined system call*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
System task call, system function call	User-defined system call	One-to-one

Handles to all user-defined system calls can be retrieved using `vpiUserSystf` as the first argument and `NULL` as the second parameter in `vpi_iterate()`.

4.6 IO declaration (`vpiIODecl`)

4.6.1 Category

Object.

4.6.2 Properties

Table 46. Properties of an object of type *io declaration*

Explanation	Name of the property	Data type of the property	Function to use
Direction of the io declaration; should be one of these: vpiInput, vpiOutput, vpiInout, vpiMixedIO,	vpiDirection	integer	

Explanation	Name of the property	Data type of the property	Function to use
vpiNoDirection			
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Is it a scalar ?	vpiScalar	boolean	
Is it a vector ?	vpiVector	boolean	
Bit size	vpiSize	integer	

4.6.3 Relationships

Table 47. Objects related to *IO declaration*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
IO declaration	Module	One-to-one
Module	IO declaration	One-to-many
IO declaration	UDP definition	One-to-one
UDP definition	IO declaration	One-to-many
IO declaration	Net, reg, variables	One-to-one (vpiExpr)
IO declaration	Expression	One-to-one (vpiLeftRange, vpiRightRange)
IO declaration	Task and function	One-to-one
Task and function	IO declaration	One-to-many

4.7 Ports (`vpiPort`)

4.7.1 Category

Class.

4.7.2 Constituent objects

- Port (`vpiPort`).
- Port bit (`vpiPortBit`).

4.7.3 Relationships between the constituent objects

Table 48. Relationship between *port* and *port bit*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Port	Port bit	One-to-many (vpiBit)
Port bit	Port	One-to-one (vpiParent)

4.7.4 Properties

Table 49. Properties of an object of type port

Explanation	Name of the property	Data type of the property	Function to use
Is the port connected by name ?	vpiConnByName	boolean	
Reading MIPD delay	-	-	vpi_get_delays()
Modifying MIPD delay	-	-	vpi_put_delays()
The direction of the port; should be one of these: vpiInput, vpiOutput, vpiInout, vpiMixedIO, vpiNoDirection	vpiDirection	integer	
Is the port explicitly named ?	vpiExplicitName	boolean	
The port index	vpiPortIndex	integer	
Is this a scalar port ?	vpiScalar	boolean	
Is this a vector port ?	vpiVector	boolean	
The size of the port	vpiSize	integer	
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the port	vpiName	string	
Full pathname of the port	vpiFullName	string	

`vpiPortIndex` is used to determine the port order in the IO list of the module. As obvious, `vpiPortIndex` as well as `vpiName` and `vpiFullName` do not apply to port bit. Also, a vector port does not necessarily mean that a vectored net (i.e. a bus) is connected to it.

4.7.5 Relationships

Table 50. Objects related to *ports*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Ports	Expression	One-to-one (vpiHighConn, vpiLowConn)

`vpiHighConn` and `vpiLowConn` indicates hierarchically higher (closer to the top module) and lower (further from the top module) connection respectively.

4.8 Nets (`vpiNet`)

4.8.1 Category

Class.

4.8.2 Constituent objects

- Net (`vpiNet`).
- Net bit (`vpiNetBit`).

4.8.3 Relationship between the constituent objects

Table 51. Relationship between *net* and *net bit*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Net	Net bit	One-to-many (vpiBit)
Net bit	Net	One-to-one (vpiParent)

4.8.4 Properties

Table 52. Properties of the class *nets*.

Explanation	Name of the property	Data type of the property	Function to use
Delay	-	-	vpi_get_delays()
Is it an expanded net?	vpiExpanded	boolean	
Is it implicitly declared	vpiImplicitDecl	boolean	

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the net	vpiName	string	
Full pathname of the net	vpiFullName	string	
Net declaration assignment	vpiNetDeclAssign	boolean	
Net type; should be one of the following: vpiWire, vpiWand, vpiWor, vpiTri, vpiTri0, vpiTri1, vpiTriReg, vpiTriAnd, vpiTriOr, vpiSupply1, vpiSupply0	vpiNetType	integer	
Is the net scalar ?	vpiScalar	boolean	
Is the net vector ?	vpiVector	boolean	
Is it declared as explicitly scalared ?	vpiExplicitScalared	boolean	
Is it declared as explicitly vectored ?	vpiExplicitVectored	boolean	
The size of the net	vpiSize	integer	
0-Drive strength of the net	vpiStrength0	integer	
1-Drive strength of the net	vpiStrength1	integer	
Charge strength of the net	vpiChargeStrength	integer	
Reading the value of the net	-	-	vpi_get_value()
Modifying the value of the net	-	-	vpi_put_value()

For an implicit net, `vpiLineNo` and `vpiFile` will return 0 and the file where the net is first referenced.

4.8.5 Relationships

Table 53. Objects related to *nets*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Nets	Ports	One-to-many (vpiPortInst)
Nets	Ports	One-to-many
Ports	Nets	One-to-one (vpiHighConn, vpiLowConn)
Net	Expression	One-to-one (vpiDelay)
Nets	Expression	One-to-one (vpiLeftRange, vpiRightRange)
Nets	Assignment statement, force, primitive term, continuous assignment, ports	One-to-many (vpiLoad, vpiDriver)
Nets	Path term	One-to-many
Nets	Timing check term	One-to-many
Net bit	Expression	One-to-one (vpiIndex)

While using `vpiPort` or `vpiPortInst`, a handle to a port bit or a vectored port is returned, depending on whether the starting object is a net bit or the entire vectored net.

In case of `vpiLoad`, only active forces and assignment statements are returned, whereas `vpiDriver` returns only active forces.

4.9 Regs (`vpiReg`)

4.9.1 Category

Class.

4.9.2 Constituent objects

- Reg (`vpiReg`).
- Reg bit (`vpiRegBit`).

4.9.3 Relationships between the constituent objects

Table 54. Relationship between *reg* and *reg bit.*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Reg	Reg bit	One-to-many (vpiBit)
Reg bit	Reg	One-to-one (vpiParent)

4.9.4 Properties

Table 55. Properties of the class *regs.*

Explanation	Name of the property	Data type of the property	Function to use
Reading the value of the reg	-	-	vpi_get_value()
Modifying the value of the reg	-	-	vpi_put_value()
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the reg	vpiName	string	
Full pathname of the reg	vpiFullName	string	
Is the reg scalar ?	vpiScalar	boolean	
Is the reg vector ?	vpiVector	boolean	
The size of the reg	vpiSize	integer	

4.9.5 Relationships

Table 56. Objects related to *regs*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Regs	Ports	One-to-many (vpiPortInst)
Ports	Regs	One-to-one (vpiLowConn, vpiHighConn)
Reg	Expression	One-to-one (vpiDelay)
Regs	Expression	One-to-one (vpiLeftRange, vpiRightRange)
Regs	Assignment statement, force,	One-to-many

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
	primitive term, continuous assignment	(vpiLoad, vpiDriver)
Regs	Path term	One-to-many
Regs	Timing check term	One-to-many
Reg bit	Expression	One-to-one (vpiIndex)

While using `vpiDriver` or `vpiLoad`, only active forces and procedural continuous assignment statements are returned.

4.10 Variables (`vpiVariables`)

4.10.1 Category

Class.

4.10.2 Constituent objects

- Integer variable (`vpiIntegerVar`).
- Time variable (`vpiTimeVar`).
- Real variable (`vpiRealVar`).

4.10.3 Relationships among the constituent objects

None.

4.10.4 Properties

Table 57. Properties of the class *variables*.

Explanation	Name of the property	Data type of the property	Function to use
Is it an array	vpiArray	boolean	
Reading the value of the variable	-	-	vpi_get_value()
Modifying the value of the variable	-	-	vpi_put_value()
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the variable	vpiName	string	
Full pathname of the	vpiFullName	string	

Explanation	Name of the property	Data type of the property	Function to use
variable			
The size of the variable	vpiSize	integer	

4.10.5 Relationships

Table 58. Objects related to *variables*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Variables	Ports	One-to-many (vpiPortInst)
Ports	Variables	One-to-one (vpiHighConn, vpiLowConn)
Variables	Expression	One-to-one (vpiLeftRange, vpiRightRange)

`vpiLeftRange` and `vpiRightRange`, which are used for arrays of variables, are not applicable for real variables, as Verilog prohibits use of arrays of type real.

4.11 Variable-select (`vpiVarSelect`)

An object of type variable-select indicates that it is a part-select or bit-select of a variable.

4.11.1 Category

Object.

4.11.2 Properties

Table 59. Properties of an object of type *variable-select*

Explanation	Name of the property	Data type of the property	Function to use
Reading the value of the variable-select	-	-	vpi_get_value()
Modifying the value of the variable-select	-	-	vpi_put_value()
Location by line number	vpiLineNo	integer	
Location by file	vpiFile	string	

Explanation	Name of the property	Data type of the property	Function to use
name			

4.11.3 Relationships

Table 60. Objects related to *variable-select*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Variable select	Integer variable	One-to-one (vpiParent)
Integer variable	Variable select	One-to-many
Variable select	Time variable	One-to-one (vpiParent)
Time variable	Variable select	One-to-many
Variable select	Expression	One-to-one (vpiIndex)

4.12 Named event (`vpiNamedEvent`)

4.12.1 Category

Object.

4.12.2 Properties

Table 61. Properties of an object of type *named-event*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the named event	vpiName	string	
Full pathname of the named event	vpiFullName	string	

4.12.3 Relationships

Same as what was discussed before in *Scope*.

4.13 Memory (`vpiMemory`)

4.13.1 Category

Object.

4.13.2 Properties

Table 62. Properties of an object of type *memory*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the memory	vpiName	string	
Full pathname of the memory	vpiFullName	string	
Number of words in the memory	vpiSize	integer	

4.13.3 Relationships

Table 63. Objects related to *memory*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Memory	Expression	One-to-one (vpiLeftRange, vpiRightRange)

4.14 Memory word (`vpiMemoryWord`)

4.14.1 Category

Object.

4.14.2 Properties

Table 64. Properties of an object of type *memory word*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	

Explanation	Name of the property	Data type of the property	Function to use
Location by file name	vpiFile	string	
Name of the memory	vpiName	string	
Full pathname of the memory	vpiFullName	string	
Number of bits in the memory word	vpiSize	integer	
Reading the value of the memory word	-	-	vpi_get_value()
Modifying the value of the memory word	-	-	vpi_put_value()

4.14.3 Relationships

Table 65. Objects related to *memory word*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Memory	Memory word	One-to-many
Memory word	Memory	One-to-one (vpiParent)
Memory word	Expression	One-to-one (vpiIndex, vpiLeftRange, vpiRightRange)

4.15 Parameter (`vpiParameter`)

4.15.1 Category

Object.

4.15.2 Properties

Table 66. Properties of an object of type *parameter*.

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the parameter	vpiName	string	
Full pathname of the parameter	vpiFullName	string	

Explanation	Name of the property	Data type of the property	Function to use
Reading the value of the parameter	-	-	vpi_get_value()
Parameter constant type; should be one of the followings: vpiDecConst, vpiRealConst, vpiBinaryConst, vpiOctConst, vpiHexConst, vpiStringConst	vpiConstType	integer	

Since a parameter can be defined in more than one places in the source (using defparam), reading the value of a parameter returns its final value.

4.15.3 Relationships

Table 67. Objects related to *parameter*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Parameter	Expression	One-to-one
Scope	Parameter	One-to-many
Parameter	Scope	One-to-one

4.16 Specparam (`vpiSpecParam`)

4.16.1 Category

Object.

4.16.2 Properties

Table 68. Properties of an object of type specparam

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the specparam	vpiName	string	
Full pathname of the specparam	vpiFullName	string	

Explanation	Name of the property	Data type of the property	Function to use
Reading the value of the specparam	-	-	vpi_get_value()
Specparam constant type; should be one of the followings: vpiDecConst, vpiRealConst, vpiBinaryConst, vpiOctConst, vpiHexConst, vpiStringConst	vpiConstType	integer	

4.16.3 Relationships

Table 69. Objects related to *specparam*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Specparam	Expression	One-to-one

4.17 Defparam (`vpiDefParam`)

4.17.1 Category

Object.

4.17.2 Properties

Table 70. Properties of an object of type *defparam*.

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.17.3 Relationships

Table 71. Objects related to *defparam*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Defparam	Parameter	One-to-one (vpiLhs)

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Defparam	Expression	One-to-one (vpiRhs)

4.18 Parameter assignment (`vpiParamAssign`)

4.18.1 Category

Object.

4.18.2 Properties

Table 72. Properties of an object of type *parameter assignment.*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.18.3 Relationships

Table 73. Objects related to *parameter assignment*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Parameter assignment	Parameter	One-to-one (vpiLhs)
Parameter assignment	Expression	One-to-one (vpiRhs)

4.19 Primitive (`vpiPrimitive`)

4.19.1 Category

Class.

4.19.2 Constituent objects

- Gate (`vpiGate`).
- Switch (`vpiSwitch`).
- UDP (`vpiUdp`).

4.19.3 Relationships among the constituent objects

None.

4.19.4 Properties

Table 74. Properties of the class primitive

Explanation	Name of the property	Data type of the property	Function to use
Definition name	vpiDefName	string	
Reading the delay	-	-	vpi_get_delays()
Modifying the delay	-	-	vpi_put_delays()
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Name of the primitive	vpiName	string	
Full pathname of the primitive	vpiFullName	string	
Primitive type; should be one of the following: vpiAndPrim, vpiNandPrim, vpiNorPrim, vpiOrPrim, vpiXorPrim, vpiXnorPrim, vpiBufPrim, vpiNotPrim, vpiBufif0Prim, vpiBufif1Prim, vpiNotIf0Prim, vpiNotIf1Prim, vpiNmosPrim, vpiPmosPrim, vpiCmosPrim, vpiRnmosPrim, vpiRpmosPrim, vpiRcmosPrim, vpiRtranPrim, vpiRtranif0Prim, vpiRtranif1Prim, vpiTranPrim, vpiTranif0Prim,	vpiPrimType	integer	

Explanation	Name of the property	Data type of the property	Function to use
vpiTranif1Prim, vpiPullupPrim, vpiPulldownPrim, vpiSeqPrim, vpiCombPrim			
Number of inputs	vpiSize	integer	
0-Drive strength	vpiStrength0	integer	
1-Drive strength	vpiStrength1	integer	
Reading the output value of the primitive	-	-	vpi_get_value()
Modifying the value of a *sequential UDP only*	-	-	vpi_put_value()

4.19.5 Relationships

Table 75. Objects related to *primitive*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Primitive	Expression	One-to-one (vpiDelay)
UDP	UDP definition	One-to-one

4.20 Primitive terms (`vpiPrimTerm`)

The terms (input/output parameters etc.) associated with a primitive instantiation are known as primitive terms.

4.20.1 Category

Object.

4.20.2 Properties

Table 76. Properties of an object of type primitive term

Explanation	Name of the property	Data type of the property	Function to use
Direction of the term; should be one of the following:	vpiDirection	integer	

Explanation	Name of the property	Data type of the property	Function to use
vpiInput, vpiOutput, vpiInout, vpiMixedIO, vpiNoDirection			
Index of the term	vpiTermIndex	Integer	
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Reading the value of the term	-	-	vpi_get_value()

4.20.3 Relationships

Table 77. Objects related to *primitive term*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Primitive	Primitive term	One-to-many
Primitive term	Primitive	Many-to-one
Primitive term	Expression	One-to-one

4.21 UDP definition (`vpiUdpDefn`)

4.21.1 Category

Object.

4.21.2 Properties

Table 78. Properties of an object of type *UDP definition.*

Explanation	Name of the property	Data type of the property	Function to use
Definition name	vpiDefName	string	
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Number of inputs	vpiSize	integer	
Is the code protected ?	vpiProtected	boolean	
The type of the primitive; should be	vpiPrimType	integer	

Explanation	Name of the property	Data type of the property	Function to use
one of the following: vpiSeqPrim, vpiCombPrim			

4.21.3 Relationships

Table 79. Objects related to *UDP definition*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
UDP definition	IO declaration	One-to-many
IO declaration	UDP definition	One-to-one
UDP definition	Initial	One-to-one
Initial	UDP definition	One-to-one

4.22 UDP table entry (`vpiTableEntry`)

4.22.1 Category

Object.

4.22.2 Properties

Table 80. Properties of an object of type *UDP table entry*.

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Number of symbol entries	vpiSize	integer	
Value of an entry as string (decompilation) and vector (ASCII values)	-	-	vpi_get_value()

4.22.3 Relationships

Table 81. Objects related to *UDP table entry*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
UDP definition	UDP table entry	One-to-many

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
UDP table entry	UDP definition	One-to-one

4.23 Module path (`vpiModPath`)

4.23.1 Category

Object.

4.23.2 Properties

Table 82. Properties of an object of type *module path*.

Explanation	Name of the property	Data type of the property	Function to use
Reading the associated delay	-	-	vpi_get_delays()
Modifying the associated delay	-	-	vpi_put_delays()
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Path type; should be one of the following: vpiPathFull, vpiPathParallel	vpiPathType	integer	
Polarity of the module path; should be one of the following: vpiPositive, vpiNegative, vpiUnknown.	vpiPolarity	integer	
Polarity of the data path; can have any of the values shown for the module path.	vpiDataPolarity	integer	
Has the state-dependent module path *ifnone* condition specified ?	vpiModPathHasIfnone	boolean	

4.23.3 Relationships

Table 83. Objects related to *module path*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Module path	Expression	One-to-one (vpiCondition, vpiDelay)

4.24 Path term (`vpiPathTerm`)

4.24.1 Category

Object.

4.24.2 Properties

Table 84. Properties of the object of type *path term.*

Explanation	Name of the property	Data type of the property	Function to use
Direction; should be one of the following: vpiInput, vpiOutput, vpiInout, vpiMixedIO, vpiNoDirection	vpiDirection	integer	
Edge; should be one of the following: vpiNoEdge, vpiEdge01, vpiEdge10, vpiEdge0x, vpiEdgex0, vpiEdgex1, vpiEdge1x, vpiPosedge, vpiNegedge, vpiAnyedge	vpiEdge	integer	
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.24.3 Relationships

Table 85. Objects related to *path term*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Path term	Module path	One-to-one
Module path	Path term	One-to-many (vpiModPathIn, vpiModPathOut, vpiModDataPathIn)
Path term	Expression	One-to-one

4.25 Intermodule path (`vpiInterModPath`)

A handle to an intermodule path between `portA` and `portB` is obtained using the following.

```
vpi_handle_multi(vpiInterModPath, portA, portB);
```

4.25.1 Category

Object.

4.25.2 Properties

Table 86. Properties of an object of type *intermodule path.*

Explanation	Name of the property	Data type of the property	Function to use
Reading the associated delay	-	-	vpi_get_delays()
Modifying the associated delay	-	-	vpi_put_delays()

4.25.3 Relationships

Table 87. Objects related to *intermodule path*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Intermodule path	Ports	One-to-many

4.26 Timing check (vpiTchk)

4.26.1 Category

Object.

4.26.2 Properties

Table 88. Properties of an object of type timing check

Explanation	Name of the property	Data type of the property	Function to use
Reading the associated delay	-	-	vpi_get_delays()
Modifying the associated delay	-	-	vpi_put_delays()
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Timing check type; should be one of the following: vpiSetup ($Setup), vpiHold ($hold), vpiPeriod ($period), vpiWidth ($width), vpiSkew ($skew), vpiRecovery ($recovery), vpiNoChange ($nochange), vpiSetupHold ($setuphold)	vpiTchkType	integer	

4.26.3 Relationships

Table 89. Objects related to *timing check*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Timing check	Expression	One-to-one (vpiDelay)
Timing check	Regs	One-to-one (vpiTchkNotifier)

4.27 Timing check term (`vpiTchkTerm`)

4.27.1 Category

Object.

4.27.2 Properties

Table 90. Properties for an object of type *timing check term*

Explanation	Name of the property	Data type of the property	Function to use
Edge; should be one of the following: vpiNoEdge, vpiEdge01, vpiEdge10, vpiEdge0x, vpiEdgex0, vpiEdgex1, vpiEdge1x, vpiPosedge, vpiNegedge, vpiAnyedge	vpiEdge	integer	
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.27.3 Relationships

Table 91. Objects related to *timing check term*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Timing check term	Timing check	One-to-one
Timing check	Timing check term	One-to-one (vpiTchkRefTerm, vpiTchkDataTerm)
Timing check term	Expression	One-to-one
Timing check term	Expression	One-to-one (vpiCondition)

4.28 Continuous assignment (`vpiContAssign`)

4.28.1 Category

Object.

4.28.2 Properties

Table 92. Properties of an object of type *continuous assignment*

Explanation	Name of the property	Data type of the property	Function to use
Associated delay	-	-	vpi_get_delays()
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Is this assignment made at the time of net declaration ?	vpiNetDeclAssign	boolean	
0-Drive strength of the net	vpiStrength0	integer	
1-Drive strength of the net	vpiStrength1	integer	

4.28.3 Relationships

Table 93. Objects related to *continuous assignment*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Continuous assignment	Expression	One-to-one (vpiRhs, vpiLhs, vpiDelay)

4.29 Simple expression

4.29.1 Category

Class.

4.29.2 Constituent objects/classes

- Net (`vpiNet`).
- Reg (`vpiReg`).
- Variables (`vpiVariables`).

- Parameter (`vpiParameter`).
- Memory (`vpiMemory`).
- Variable select (`vpiVarSelect`)
- Memory Word (`vpiMemoryWord`)

4.29.3 Relationships among the constituent parts

None.

4.29.4 Properties

Table 94. Properties of the class *simple expression*

Explanation	Name of the property	Data type of the property	Function to use
Name of the expression	vpiName	string	
Full pathname of the expression	vpiFullName	string	

4.29.5 Relationships

Table 95. Objects related to *simple expression*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Simple expression	Primitive term, path term, timing check term, port, statement, continuous assignment	One-to-many (vpiUse)

For a vector, `vpiUse` returns a handle to any use of the vector or a part or bit-select of it. However, for a bit-select it returns any specific use of the bit or its parent or its part-select that uses this bit.

4.30 Part select (`vpiPartSelect`)

4.30.1 Category

Object.

4.30.2 Properties

Table 96. Properties of the object of type *part select*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.30.3 Relationships

Table 97. Objects related to *part select*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Part select	Expression	One-to-one (vpiLeftRange, vpiRightRange)

4.31 Operation

4.31.1 Category

Object.

4.31.2 Properties

Table 98. Properties of an object of type *operation*

Explanation	Name of the property	Data type of the property	Function to use
Type of the operation	vpiOpType	integer	
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.31.3 Relationships

Table 99. Objects related to *operation*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Operation	Expression	One-to-one (vpiOperation)

4.32 Constant (`vpiConstant`)

4.32.1 Category

Object.

4.32.2 Properties

Table 100. Properties of an object of type *constant*

Explanation	Name of the property	Data type of the property	Function to use
Type of the constant; should be one of the following: vpiDecConst, vpiRealConst, vpiBinaryConst, vpiOctConst, vpiHexConst, vpiStringConst	vpiConstType	integer	
Location by line number	vpiLineNo	integer	

4.32.3 Relationships

None.

4.33 Expression (`vpiExpr`)

4.33.1 Category

Class.

4.33.2 Constituent objects/classes

- Simple expression
- Part select (`vpiPartSelect`)
- Operation (`vpiOperation`)
- Constant (`vpiConstant`)
- Function call (`vpiFuncCall`)
- System function call (`vpiSysFuncCall`)

4.33.3 Relationships among the constituent objects

Table 101. Relationship between *simple expression* and *part select*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Part select	Simple expression	vpiParent

4.33.4 Properties

Table 102. Properties of the class expression

Explanation	Name of the property	Data type of the property	Function to use
Bit size of the expression	vpiSize	integer	
Reading the value of the expression	-	-	vpi_get_value()

4.33.5 Relationships

None, other than the relationships associated with its constituent objects, e.g. *part select* or *operation.*

4.34 Event statement (`vpiEventStmt`)

4.34.1 Category

Object.

4.34.2 Properties

Table 103. Properties of an object of type *event statement*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.34.3 Relationship

Table 104. Objects related to *event statement*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Event statement	Named event	One-to-one

4.35 Assignment (`vpiAssignment`)

4.35.1 Category

Object.

4.35.2 Properties

Table 105. Properties of an object of type *assignment*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Is it a blocking assignment ?	vpiBlocking	boolean	

4.35.3 Relationships

Table 106. Objects related to *assignment*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Assignment	Expression	One-to-one (vpiLhs, vpiRhs)
Assignment	Delay control	One-to-one
Assignment	Event control	One-to-one
Assignment	Repeat control	One-to-one

4.36 Delay control (`vpiDelayControl`)

4.36.1 Category

Object.

4.36.2 Properties

Table 107. Properties of an object of type *delay control*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	
Associated delay	-	-	vpi_get_delay()

4.36.3 Relationships

Table 108. Objects related to *delay control*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Delay control	Statement (for assignment, the handle will be NULL)	One-to-one
Delay control	Expression	One-to-one (vpiDelay)

4.37 Event control (`vpiEventControl`)

4.37.1 Category

Object.

4.37.2 Properties

Table 109. Properties of an object of type *event control*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.37.3 Relationships

Table 110. Objects related to *event control*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Event control	Expression, named event	One-to-one

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
		(vpiCondition)
Event control	Statement (for assignment, the handle will be NULL)	One-to-one

4.38 Repeat control (`vpiRepeatControl`)

4.38.1 Category

Object.

4.38.2 Properties

Table 111. Properties of an object of type *repeat control*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.38.3 Relationships

Table 112. Objects related to *repeat control*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Repeat control	Expression	One-to-one
Repeat control	Event control	One-to-one

4.39 While, Repeat and Wait statements

4.39.1 Category

Class.

4.39.2 Constituent objects

- While Statement (`vpiWhile`).
- Repeat Statement (`vpiRepeat`).
- Wait Statement (`vpiWait`).

4.39.3 Relationship among the objects

None.

4.39.4 Properties

Table 113. Properties of the class *while, repeat and wait statements*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.39.5 Relationship

Table 114. Objects related to *while, repeat and wait statements*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
While, repeat and wait statements	Expression	One-to-one (vpiCondition)
While, repeat and wait statements	Statement	One-to-one

4.40 For statement (`vpiFor`)

4.40.1 Category

Object.

4.40.2 Properties

Table 115. Properties of an object of type *for statement*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.40.3 Relationships

Table 116. Objects related to *for statement*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
For statement	Statement (the initialization statement)	One-to-one (vpiForInitStmt)
For statement	Expression (the condition for termination)	One-to-one (vpiCondition)
For statement	Statement (the statement executed at the end of each iteration)	One-to-one (vpiForIncStmt)
For statement	Statement (statement in loop)	One-to-one

4.41 Forever statement (`vpiForever`)

4.41.1 Category

Object.

4.41.2 Properties

Table 117. Properties of an object of type *forever statement*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.41.3 Relationship

Table 118. Objects related to *forever statement*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Forever statement	Statement	One-to-one

4.42 If and If-Else statements

4.42.1 Category

Class.

4.42.2 Constituent objects

- If (`vpiIf`).
- If-Else (`vpiIfElse`)

4.42.3 Relationship among the constituent objects

None.

4.42.4 Properties

Table 119. Properties of the class *if and if-else statement*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.42.5 Relationship

Table 120. Objects related to *if and if-else statement*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
If and if-else statement	Expression	One-to-one (vpiCondition)
If and if-else statement	Statement	One-to-one
If-else statement	Statement	One-to-one (vpiElseStmt)

4.43 Case statement (`vpiCase`)

4.43.1 Category

Object.

4.43.2 Properties

Table 121. Properties of an object of type *case statement*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	

Explanation	Name of the property	Data type of the property	Function to use
Location by file name	vpiFile	string	
Type of case; should be one of vpiCaseExact, vpiCaseX, vpiCaseZ	vpiCaseType	integer	

4.43.3 Relationships

Table 122. Objects related to *case statement*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Case statement	Expression	One-to-one (vpiCondition)

4.44 Case item (`vpiCaseItem`)

4.44.1 Category

Object.

4.44.2 Properties

Table 123. Properties of an object of type *case item*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.44.3 Relationships

Table 124. Objects related to *case item*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Case item	Expression	One-to-many
Case item	Statement	One-to-one (vpiStmt)

4.45 Force and procedural continuous assignment

4.45.1 Category

Class.

4.45.2 Constituent objects

- Force (`vpiForce`).
- Assign i.e. continuous assignment inside a procedural block (`vpiAssignStmt`)

4.45.3 Relationship between the constituent objects

None.

4.45.4 Properties

Table 125. Properties of the class *force and procedural continuous assignment*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.45.5 Relationships

Table 126. Objects related to *force and procedural continuous assignment*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Force and procedural continuous statement	Expression	One-to-one (vpiRhs, vpiLhs)

4.46 Release and Deassign statement

4.46.1 Category

Class.

4.46.2 Constituent objects

- Deassign (`vpiDeassign`).
- Release (`vpiRelease`).

4.46.3 Relationship between the constituent objects

None.

4.46.4 Properties

Table 127. Properties of the class *release and deassign statement*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.46.5 Relationships

Table 128. Objects related to *release and deassign statement*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Release and deassign statements	Expression	One-to-one (vpiLhs)

4.47 Disable statement (`vpiDisable`)

4.47.1 Category

Object.

4.47.2 Properties

Table 129. Properties of an object of type *disable statement*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.47.3 Relationship

Table 130. Objects related to disable statement

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Disable statement	Function, task, named fork or named begin	One-to-one (vpiExpr)

4.48 Atomic statement

4.48.1 Category

Class.

4.48.2 Constituent objects/classes

- If (`vpiIf`).
- If Else (`vpiIfElse`).
- While (`vpiWhile`).
- Repeat (`vpiRepeat`).
- Wait (`vpiWait`).
- Case (`vpiCase`).
- For (`vpiFor`).
- Delay Control (`vpiDelayControl`).
- Event Control (`vpiEventControl`).
- Event Statement (`vpiEventStmt`).
- Assignment (`vpiAssignment`).
- Assignment Statement (`vpiAssignStmt`).
- Deassign (`vpiDeassign`).
- Disable (`vpiDisable`).
- Task And Function Call
- Forever (`vpiForever`).
- Force (`vpiForce`).
- Release (`vpiRelease`).
- Null Statement (`vpiNullStmt`).

4.48.3 Relationships among the constituent objects/classes

None.

4.48.4 Properties

None.

4.48.5 Relationships

None.

4.49 Block statement

4.49.1 Category

Class.

4.49.2 Constituent objects

- Begin (`vpiBegin`)
- Named begin (`vpiNamedBegin`).
- Fork (`vpiFork`).
- Named fork (`vpiNamedFork`).

4.49.3 Relationships among the constituent objects

None.

4.49.4 Properties

Table 131. Properties of the class *block statement*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.49.5 Relationships

Table 132. Objects related to *block statement*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Block statement	Statement	One-to-many

4.50 Statement (`vpiStmt`)

4.50.1 Category

Class.

4.50.2 Constituent objects/classes

- Block statement
- Atomic statement

4.50.3 Relationship among the constituent objects/classes

None.

4.50.4 Properties

None.

4.50.5 Relationships

None, except those mentioned under *process* and *scope*.

4.51 Process (`vpiProcess`)

4.51.1 Category

Class.

4.51.2 Constituent objects

- Initial (`vpiInitial`).
- Always (`vpiAlways`).

4.51.3 Relationships among the object

None.

4.51.4 Properties

Table 133. Properties of an object of type *process*

Explanation	Name of the property	Data type of the property	Function to use
Location by line number	vpiLineNo	integer	
Location by file name	vpiFile	string	

4.51.5 Relationships

Table 134. Objects related to *process*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Process	Statement	One-to-one
Statement	Process	One-to-one

4.52 Callback

4.52.1 Category

Object.

4.52.2 Property

Table 135. Property of an object of type *callback*

Explanation	Name of the property	Data type of the property	Function to use
Callback information	-	-	vpi_get_cb_info()

4.52.3 Relationships

Table 136. Objects related to *callback*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Callback	Expression	One-to-one
Expression	Callback	One-to-many
Callback	Primitive term	One-to-one
Primitive term	Callback	One-to-many
Callback	Statement	One-to-one
Statement	Callback	One-to-many

4.53 Time queue (`vpiTimeQueue`)

4.53.1 Category

Object.

4.53.2 Property

Table 137. Property of an object of type *time queue*

Explanation	Name of the property	Data type of the property	Function to use
Time	-	-	vpi_get_time()

4.53.3 Relationship

Table 138. Objects related to *time queue*

Starting object/class	Destination object/class	Nature of relationship (Tag, if required)
Callback	Time queue	One-to-one (vpiParent)

4.54 Summary

This chapter has introduced the Verilog Procedural Interface (VPI). We have discussed basic methodologies of program construct, data-access and traversing relationships have been discussed. These methodologies form the basis of the VPI. Concept-wise, they are substantially different from the PLI1.0 methodology.

The other powerful feature of VPI is its set of library functions. We have seen the power of some of them in this chapter, when the properties of a large set of objects are obtained using only a handful of functions. We will discuss these library functions in detail in the next chapter.

Chapter 8

LIBRARY FUNCTIONS IN VPI

In this chapter we are going to discuss the library functions in Verilog Procedural Interface (VPI), also known as PLI2.0. We will divide the functions into broad categories according to their use. The examples accompanying each category of functions will demonstrate how to use VPI to solve some frequently encountered problems. The chapter ends with a discussion on the shortcomings of present PLI methodology and its future direction.

1. DIFFERENT CLASSES OF VPI FUNCTIONS

VPI functions have been designed in such a manner that the interface remains consistent for a variety of objects and purposes. As a result, the total number of library functions has been greatly reduced from the earlier versions of PLI. There are only 27 library functions defined to take care of all needs.

The following table summarizes which function to use for a specific purpose.

Table 139. Summary of functions and their properties

Purpose	VPI function
Accessing an object from its properties	vpi_handle_by_name()
	vpi_handle_by_index()
Accessing properties of an object	vpi_get()
	vpi_get_str()
	vpi_get_value()
	vpi_put_value()
	vpi_get_delays()
	vpi_put_delays()

S. Mittra, *Principles of Verilog PLI*
© Kluwer Academic Publishers 1999

Purpose	VPI function
Getting simulation time	vpi_get_time()
Simulation related callbacks	vpi_register_cb()
	vpi_remove_cb()
	vpi_get_cb_info()
User-defined system call routines	vpi_register_systf()
	vpi_get_systf_info()
Traversing relationships	vpi_handle()
	vpi_iterate()
	vpi_scan()
	vpi_handle_multi()
File related activities	vpi_mcd_open()
	vpi_mcd_close()
	vpi_mcd_print()
	vpi_mcd_name()
Displaying messages	vpi_printf()
Getting product invocation information	vpi_get_vlog_info()
Comparing objects	vpi_compare_objects()
Getting error information	vpi_chk_error()
Freeing allocated memory	vpi_free_object()

Note that some of the VPI functions (such as `vpi_mcd_name()`) return a character pointer. The string corresponding to this character pointer is copied to an internal buffer. This internal buffer is overwritten whenever another VPI function from the user application returns a character pointer. In order to save the previous string, you must copy it to a user-defined string.

2. FUNCTIONS IN VPI

2.1 Getting a handle to an object from its properties

2.1.1 Functions in this category

```
vpiHandle vpi_handle_by_name(name, scope)
char *name;
vpiHandle scope;

vpiHandle vpi_handle_by_index(object, index)
vpiHandle object;
int index;
```

2.1.2 Purpose

The above functions are used to obtain a handle to an object whose certain properties are known *a priori*. These properties are the path name of the object or the index of the object in the parent object.

A handle to an object can be obtained by specifying the handle to the current scope and the relative path name with respect to the current scope using `vpi_handle_by_name()`. If a `NULL` is specified as the handle to the current scope, the path name is assumed to be the full path name from the top level module.

A handle to an object can be obtained by using the function `vpi_handle_by_index()` when the handle to its parent as well as its index are known. For example, this function can be used to obtain handle to a net-bit when the handle to the entire net and the index of the bit is known.

2.1.3 Example

The following example gets a handle (`bitHandle`) to the 4th bit of a register, whose full path name is `top.submod2.myreg`.

```
vpiHandle regHandle, bitHandle;

regHandle = vpi_handle_by_name("top.submod2.myreg", NULL);
bitHandle = vpi_handle_by_index(regHandle, 4);
```

2.2 Accessing properties of an object

2.2.1 Reading the properties of an object

2.2.1.1 Functions in this category

```
int vpi_get(property, objHandle)
int property;
vpiHandle objHandle;

char *vpi_get_str(property, objHandle)
int property;
vpiHandle objHandle;
```

2.2.1.2 Purpose

These functions are used to obtain properties of an object. The type of the property determines which function to use.

For accessing integer and boolean (defined as 1 for true and 0 for false) properties, the function `vpi_get()` is used. (see Section 4 of Chapter 7 for a list of properties of an object). As a special case, for property `vpiTimeUnit`, if `objHandle` is `NULL`, then the simulation time unit is returned.

For retrieving properties of type `char *`, the function `vpi_get_str()` is used. The returned string is stored in a temporary buffer and must be copied to a user-allocated space before calling the function again.

2.2.1.3 Example

Given a handle `modHndl` to a module, the following function `print_location()` prints the file name and the line number where it is declared.

```
void print_location(modHndl)
vpiHandle modHndl;
{
   vpi_printf("%s:: File: %s, Line no = %d\n",
              vpi_get_str(vpiFullName, modHndl),
              vpi_get_str(vpiFile, modHndl),
              vpi_get(vpiLineNo, modHndl));
}
```

2.2.2 Reading/modifying object delays

2.2.2.1 Functions in this category

```
void vpi_get_delays(objHandle, delay_p)
vpiHandle objHandle;
p_vpi_delay delay_p;

void vpi_put_delays(objHandle, delay_p)
vpiHandle objHandle;
p_vpi_delay delay_p;
```

2.2.2.2 Purpose

The function `vpi_get_delays()` retrieves the delays or pulse limits associated with an object, whereas `vpi_put_delays()` modifies the existing delay values of an object. Both of them take the corresponding object handle as the first argument. The second argument is a pointer to `s_vpi_delay` structure. This structure and one of its fields `s_vpi_time` are defined in `vpi_user.h` and given below. You should allocate the required memory for these structures before passing them to the corresponding function.

```
typedef struct t_vpi_delay {
   struct t_vpi_time *da;
   /* user-allocated array of delay values */
   int no_of_delays;
   /* number of delays */
   int time_type;
   /* either vpiScaledRealTime or vpiSimTime */
   bool mtm_flag;
   /* true, if the delay is in min:typ:max format */
   bool append_flag;
   /* true, if appending to the existing delay,
      false, if replacing with a new value */
   bool pulsere_flag;
   /* true for pulse reject and error limits */
} s_vpi_delay, *p_vpi_delay;

typedef struct t_vpi_time {
   int type;
   /* either vpiScaledRealTime or vpiSimTime */
   unsigned int high, low;
   /* upper and lower 32 bits of time when
      type == vpiSimTime */
   double real;
   /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

The `no_of_delays` field indicates the number of delays you want to access. For a specific object, there can be 1, 2, 3 or 6 such delays for different types of transitions as shown in the following table.

Table 140. no_of_delays and delay types

Number of delays	Delay types (in order)
1	0-to-1
2	0-to-1, 1-to-0
3	0-to-1, 1-to-0, 0-to-z
6	0-to-1, 1-to-0, 0-to-z, z-to-1, 1-to-z, z-to-0

You must allocate the memory for the array `da`, before it is used for reading or modifying any delay value. If both `mtm_flag` and `pulsere_flag` are false, there are as many elements in the array `da` as the `no_of_delays`.

If `mtm_flag` is set, but `pulsere_flag` is still false, there are 3 (for minimum, typical and maximum) values of delays corresponding to each type of transition and the number of elements in `da` is 3 times that of the

no_of_delays. In such case, for example, da[0] indicates minimum delay for 0-to-1 transition, da[4] indicates typical delay for 1-to-0 transition etc.

If mtm_flag is false and pulsere_flag is true, there are again 3 values (corresponding to delay value, reject and error limit, in order) of delays for each transition type and number of elements in da is thrice the no_of_delays. In such a situation, for example, da[0] will indicate the delay value for 0-to-1 transition, da[5] will have error limit for 1-to-0 transition etc.

If both mtm_flag and pulsere_flag are set, there will be 9 values (delay value, reject region and error limit, and their minimum, typical and maximum values) of delays corresponding to each of the transition. So, for example, da[3] indicates the minimum reject region and da[7] will give the typical error limit value for 0-to-1 transition. (For details on the interpretation of these delays, refer to any Verilog HDL book)

2.2.2.3 Example

The function check_delay() in the following example reads the minimum, typical and maximum delays in a module, whose handle is modHndl. It replaces the typical delay with a pre-defined value TYPICAL_DELAY_VALUE if it is less than that value.

```
void check_delay(modHndl)
vpiHandle modHndl;
{
   s_vpi_delay delay_s;
   p_vpi_delay delay_p = &delay_s;
   s_vpi_time  da_mtm[3];

   delay_s.da = da_mtm;
   delay_s.no_of_delays = 1;
   delay_s.time_type = vpiSimType;
   delay_s.mtm_flag = true;
   delay_s.append_flag = false;
   delay_s.pulsere_flag = false;

   vpi_get_delays(modHndl, delay_p);
   if (da_mtm[1] < TYPICAL_DELAY_VAL) {
      da_mtm[1] = TYPICAL_DELAY_VAL;
      vpi_put_delays(modHndl, delay_p);
   }
}
```

2.2.3 Reading/modifying simulation value of object

2.2.3.1 Functions in this category

```
void vpi_get_value(objHandle, value_p)
vpiHandle objHandle;
p_vpi_value value_p;

vpiHandle vpi_put_value(objHandle, value_p,
                        time_p, delay_mode)
vpiHandle objHandle;
p_vpi_value value_p;
p_vpi_time time_p;
int delay_mode;
```

2.2.3.2 Purpose

The function `vpi_get_value()` retrieves the value of an object in user-specified format. As arguments, it takes the object handle and a pointer to a `s_vpi_value` structure. You must allocate memory for this structure (and any of its fields) before passing its address to the function. This structure and its components are defined in `vpi_user.h` and given below.

```
typedef struct t_vpi_value {
   int format;
   union {
      char *str;
      int scalar;
      int integer;
      double real;
      struct t_vpi_time *time;
      struct t_vpi_vecval *vector;
      struct t_vpi_strengthval *strength;
      char *misc;
   } value;
} s_vpi_value, *p_vpi_value;
```

The following table shows the permitted values of the field `format` of `s_vpi_value`.

Table 141. Permitted values of format

Format	The returned value	The member of value used
vpiBinStrVal	A string consisting of characters 0, 1, x and z	str

Format	The returned value	The member of value used
vpiOctStrVal	A string consisting of characters 0 to 7, x, X, z and Z	str
vpiDecStrVal	A string consisting of characters 0 to 9	str
vpiHexStrVal	A string consisting of characters 0 to f, x, X, z and Z	str
vpiScalarVal	One of the following integers `vpi0`, `vpi1`, `vpiX`, `vpiZ`, `vpiH` and `vpiL`	scalar
vpiIntVal	An integer	integer
vpiRealVal	A double	real
vpiStringVal	An ASCII string	str
vpiTimeVal	Two integers	time
vpiVectorVal	*aval/bval* representation of the value	vector
vpiStrengthVal	Value and strength	scalar, strength
vpiObjectVal	(discussed below)	(discussed below)

The structure `t_vpi_time` is given in Section 2.2.2. The structure `t_vpi_vecval` and `t_vpi_strengthval` are defined as shown below.

```
typedef struct t_vpi_vecval {
   int aval, bval;
   /* bit encoding of ab:
      00=0, 10=1, 11=x and 01=z */
} s_vpi_vecval, *p_vpi_vecval;

typedef struct t_vpi_strengthval {
   int logic;  /* vpi[0,1,X,Z] */
   int s0, s1; /* strength coding */
} s_vpi_strengthval, *p_vpi_strengthval;
```

Depending on the value of the `format` as specified by the user, the function assigns the value to a particular member of the union `value` upon return. If the `format` field is `vpiObjTypeVal`, the function automatically picks up a correct union member and also modifies the format field according to the following rule.

Table 142. Using *vpiObjTypeVal*

If the object is of type	The changed value of *format* will be	The member of *value* used
Integer	vpiIntVal	integer
Real	vpiRealVal	real
Scalar	vpiScalarVal/vpiStrengthVal	scalar/strength
Time	vpiTimeVal	time
Vector	vpiVectorVal	vector

For the `vpi_get_value()`, the required memory for the union members `str`, `time`, `vector`, `strength` and `misc` are allocated by the function itself and remains valid until the next call to the function. However, when using `vpi_put_value()`, the user must allocate the memory for the union member.

The structure and the bit encoding of the `s_vpi_vecval` is similar to the structure `t_vecval` in PLI1.0 (Chapter 3, Section 5.1.3.1).

The function `vpi_put_value()` schedules a value change of an object after a specified time (could be zero or non-zero). This delay is passed as a `p_vpi_time` pointer, as defined earlier. For UDPs, this delay must be zero.

The function can also control other events which change the value of the same object between the current time and the time when the scheduled value change takes place. This is done by taking a delay mode as an input parameter. The following delay modes are valid.

Table 143. Different delay modes

Delay mode	Effect of the delay mode on the object
vpiInertialDelay	All events between the current time and the scheduled time are canceled.
vpiTransportDelay	All events scheduled at a time later than the value change are removed. This is also known as Modified Transport Delay.
vpiPureTransportDelay	No change in other scheduled events.
vpiNoDelay	The value change takes place immediately. There is no effect of the *time_p* structure and can be set to NULL.
vpiForceFlag	This is equivalent to a *force* statement in Verilog HDL. The value change takes place immediately and the *time_p* structure is ignored.
vpiReleaseFlag	This is equivalent to a *release* statement in Verilog HDL. The previously forced value is released immediately and passed value is assigned to the object.
vpiCancelEvent	A previously scheduled event is canceled.

When the value change is not immediate, the `vpi_put_value()` returns a handle to the scheduled event, if the `delay_mode` flag is bit masked (bitwise ORed) with `vpiReturnEvent` and there is no error in scheduling the event. Otherwise it returns a `NULL`.

A previously scheduled event can be canceled by passing the handle to the event (object type is `vpiScheduled`) and setting the `delay_mode` as `vpiCancelEvent`.

2.2.3.3 Example

The following function `inv_reg()` inverts the present value of a scalar register. Assuming the handle `regHndl` to the register is known, the function finds out its current value and if it is not x or z, inverts the value. Then it immediately assigns back this inverted value to the register.

```
int inv_reg(regHndl)
vpiHandle regHndl;
{
s_vpi_value value_s;
p_vpi_value value_p = &value_s;

value_s.format = vpiScalarVal;
vpi_get_value(regHndl, value_p);
if (value_s.value.scalar == vpi1)
   value_s.value.scalar = vpi0;
else if (value_s.value.scalar == vpi0)
   value_s.value.scalar = vpi1;

if ((value_s.value.scalar == vpi1)  ||
    (value_s.value.scalar == vpi0))
   vpi_put_value(regHndl, value_p, NULL, vpiNoDelay);
return;
}
```

2.3 Getting the current simulation time

2.3.1 Function in this category

```
void vpi_get_time(object, timePtr)
vpiHandle object;
p_vpi_time timePtr;
```

2.3.2 Purpose

`vpi_get_time()` retrieves the current simulation time. The format of the retrieved time could be either scaled real or the integer simulation time depending on the value of the `timePtr->type` field set before the call. (For the structure of the `p_vpi_time` data type, refer to Section 2.2.2) The time is expressed using the time scale of the object whose handle is specified, or in simulation time unit if the handle is `NULL`.

2.3.3 Example

The function in the following example prints the current simulation time (this is equivalent of the built-in system call `$time`).

```
print_time() {
```

```
    p_vpi_time timePtr;
    int highTime, lowTime;

    timePtr = (p_vpi_time)malloc(sizeof(s_vpi_time));
    timePtr->type = vpiSimTime;

    vpi_get_time(NULL, timePtr);

    highTime = timePtr->high; /* upper 32 bits of time */
    lowTime = timePtr->low;   /* lower 32 bits of time */

    vpi_printf("%d%d", highTime, lowTime);
}
```

2.4 Simulation related callbacks

2.4.1 Functions in this category

```
vpiHandle vpi_register_cb(cb_data_ptr)
p_cb_data cb_data_ptr;

int vpi_remove_cb(cb_obj)
vpiHandle cb_obj;

void vpi_get_cb_info(obj, cb_data_ptr)
vpiHandle obj;
p_cb_data cb_data_ptr;
```

2.4.2 Purpose

Before we discuss the functions in this category, let us first see what a callback is and how it is initiated from a user program.

Callback for an API application is defined as a mechanism to call a specified user routine whenever a certain event occurs. Depending on what kind of event invokes the specified routine, callbacks may be divided into the following categories.

- Simulation event related (e.g. whenever an expression or a net changes value, or a behavioral statement is executed etc.)
- Simulation time related (e.g. scheduling an event at the end of the current event queue or at a future simulation time).
- Simulation action or feature related (e.g. when a certain simulator activity occurs, such as, end of compilation or end of save or restart).

Each of these categories of callback has a set of *reasons* to specify the exact nature of the event or situation when the callback occurs (for detailed discussion on reason in PLI1.0, see Chapter 2, Section 4.2. Although the value of reasons are different in PLI2.0, the concept remains the same). Whenever an event or situation corresponding to the reason specified by the user occurs, a user-defined function is invoked.

There are two main operations that are required from a user program to control the callback machanism.

a) To register a callback: This is to let the simulator know about the parameters of the callback and is done by using the function `vpi_register_cb()`.
b) To remove a callback: A callback puts extra burden on event scheduling by the simulator and should be removed whenever the need for the callback is over. This is done by the function `vpi_remove_cb()`.

To register a callback using `vpi_register_cb()`, a pointer to a structure of type `s_cb_data` is passed to the function indicating the reason, the associated user function and some other related fields. The definition of this structure is shown below.

```
typedef  struct t_cb_data {
   int reason;    /* Determines when the callback occurs */
   int (*cb_rtn)(); /* The user defined function */
   vpihandle obj;
   p_vpi_time time;
   p_vpi_value value;
   int index;
   char *user_data;
} s_cb_data, *p_cb_data;
```

As discussed below, the above structure is configured differently depending on the nature of the callback being registered. However, in all cases `cb_rtn()` refers to a user supplied function which is called by the simulator whenever the callback occurs. The simulator also passes a pointer `p_cb_data` (this pointer is different from the one passed earlier to `vpi_register_cb()` by the user) to `cb_rtn()`.

To register a simulation event related callback, the `reason` field should be set by the user according to the following table.

Table 144. Different types of reasons for simulation event related callbacks

To make the callback	The value of the reason should be
Before a behavioral statement is executed	cbStmt
After the value change of an expression	cbValueChange
After a force or release has occurred on a	cbForce/cbRelease

To make the callback	The value of the reason should be
simple expression	

The `obj` field of the structure should be a handle to the expression, terminal or the behavioral statement for which the callback would occur.

This structure can also be configured such that when the user function `cb_rtn()` is called during the callback, information regarding the simulation time and the changed value of the `obj` are passed to it using the pointer of type `p_cb_data` as discussed above.

In order to obtain the simulation time in real time or simulation time format, the field `cb_data_p->time->type` must be set to `vpiScaledRealTime` or `vpiSimTime` respectively (the structure of `p_vpi_time` has been shown in Section 2.2.2.2). This field should be set to `vpiSupressTime`, if the simulation time information is not required.

The field `value` is only applicable if the reason is one of `cbForce`, `cbRelease` or `cbValueChange` (the structure of `p_vpi_value` has been shown in 2.2.3.2). For the resaons `cbForce/cbRelease`, the field `cb_data_p->value->value` of the pointer `cb_data_p` passed to `cb_rtn()` contains the current value of forced or released `obj`. For `cbValueChange`, the field contains the changed value of `obj`. Additionally, if `obj` is a memory word or a variable array, the `index` field indicates the index of the memory word or variable select that changed value. `cb_data_p->value->value` should be set to `vpiSupressVal`, if the value information is not required

To register a simulation time related callback, the reason field should be set according to the following table.

Table 145. Different types of reasons for simulation time related callbacks

To make the callback	The value of the reason should be
Before execution of events, if any, in a simulation time slot	cbAtStartOfSimTime
After execution of events, if any, in a simulation time slot	cbReadWriteSynch
Same as above, but the callback routine can not schedule an event before the next scheduled event	cbReadOnlySynch
Before execution of events in the next event queue	cbNextSimTime
After a specified time delay, before execution of other events in that time slot	cbAfterDelay

In a simulation time related callback, the field `cb_data_p->time->type` is set to either `vpiScaledRealTime` or `vpiSimTime`. If the field is `vpiScaledRealTime` and the reason is `cbAfterDelay`, the time scaling is determined by the `obj`. In all other cases, `obj` is ignored. Also, the user must specify the requested time of the callback or the delay before the callback by setting `cb_data_p->time->high` and `cb_data_p->time_low`, when the type

is `vpiSimTime`, or `cb_data_p->time->real`, for the type `vpiScaledRealTime`. The field `cb_data_p->time` is ignored if the reason is `vpiNextSimTime`.

For simulation action and feature related callbacks, the reason should be set according to the following table.

Table 146. Different types of reasons for simulation action and feature related callbacks

To make the callback	The value of reason should be
After the compilation	cbEndOfCompile
At the beginning of the simulation	cbStartOfSimulation
At the end of the simulation	cbEndOfSimulation
When a run-time error occurs	cbError
When a timing check violation occurs	cbTchkViolation
Before a $save statement	cbStartOfSave
After a $save statement	cbEndOfSave
Before a $restart statement	cbStartOfRestart
After a $restart statement	cbEndOfRestart
After a $stop statement	cbEnterInteractive
After issuing a command in the interactive mode	cbExitInteractive
After changing the interactive scope	cbInteractiveScopeChange
When an unknown user-defined system call is encountered	cbUnresolvedSystf

For simulation action and feature related callbacks, `obj` field is a handle to the timing check and the new scope for the reasons `cbTchkViolation` and `cbInteractiveScopeChange` respectively.

The function `vpi_remove_cb()` removes a previously registered callback. It takes a handle to the callback object as its argument. Note that, after `vpi_register_cb()` is called, this handle is no longer valid.

The function `vpi_get_cb_info()` takes a handle to a callback object as its first argument and puts the information related to the callback in a `s_cb_data` structure. The memory for this structure must be allocated by the user.

2.4.3 Example

The following example assumes a handle `regHndl` to a reg is defined globally. Whenever there is value change on the reg, it calls back the user function `put_value_later()`. This function registers another callback after time `MY_DELAY`, when it calls the function `inv_reg_val()`. This function, using `inv_reg()` defined earlier in Section 2.2.3.3, inverts the contents of the register.

```
main_register()
```

```
{
static s_vpi_time time_s = {vpiSimTime};
static s_vpi_value value_s = {vpiScalarVal};
static s_cb_data cb_data_s =
   {cbValueChange, put_value_later, regHndl,
    &time_s, &value_s};

vpi_register_cb(&cb_data_s);
}

put_value_later(cb_data_ptr1)
p_cb_data cb_data_ptr1;
{
static s_vpi_time time_s = {vpiSimTime, 0, MY_DELAY};
static s_cb_data cb_data_s =
   {cbAfterDelay, inv_reg_val, NULL, &time_s, NULL};
}

inv_reg_val(cb_data_ptr2)
p_cb_data cb_data_ptr2;
{
   inv_reg(regHndl);
   /*This function is defined in Section 2.2.3.3. The
     handle regHndl is assumed to be defined globally*/
}
```

2.5 Registering and getting information about a user-defined system call

2.5.1 Functions in this category

```
vpiHandle vpi_register_systf(systf_data_p)
p_vpi_systf_data systf_data_p;

void vpi_get_systf_info(objHandle, systf_data_p)
vpiHandle objHandle;
p_vpi_systf_data systf_data_p;
```

2.5.2 Purpose

These functions are used to register and, later on, to retrieve information about the user-defined system task/function-related callbacks respectively.

Both of them uses the structure pointer `p_vpi_systf_data`, which is defined in `vpi_user.h` and given in Section 2.2 in Chapter 7.

While registering a system call, you must initialize the fields of this structure before passing it to `vpi_register_systf()`. A handle to the callback is returned. Note that, callbacks are registered dynamically during run-time. This gives us freedom to manipulate the registration process, such as, making it conditional or dynamically forming the name of the functions etc.

In case of `vpi_get_systf_info()`, the information regarding the system call whose handle is passed as the first argument, is retrieved and put into the structure.

2.5.3 Example

Refer to Section 2.4 in Chapter 7 for an illustration of how to use `vpi_register_systf()`.

2.6 Traversing relationships

2.6.1 Functions in this category

```
vpiHandle vpi_handle(type, ref_obj)
int type;
vpiHandle ref_obj;

vpiHandle vpi_iterate(type, ref_obj)
int type;
vpiHandle ref_obj;

vpiHandle vpi_scan(iterator)
vpiHandle iterator;

vpiHandle vpi_handle_multi(type, ref_obj1, ref_obj2, ...)
int type;
vpiHandle ref_obj1, ref_obj2, ...;
```

2.6.2 Purpose

The functions in this category are used to obtain a handle to a destination object for one-to-one, one-to-many and many-to-one relationships. The methodology of traversing relationships has been discussed in details in Chapter 7.

`vpi_handle()` returns a handle to a destination object in a one-to-one relationship when the type of relationship and the handle to the starting object are given.

For a one-to-many relationship, first an iterator handle is obtained using the function `vpi_iterator()` and later this iterator handle is used repeatedly by `vpi_scan()` to get handles to successive destinations.

Getting a handle to the inter modular path connecting ports of same width from different modules is a special case of many-to-one relationship and is traversed by `vpi_handle_multi()`.

2.6.3 Example

For examples related to traversing different types of relationships, refer to Chapter 7.

2.7 File related activities

2.7.1 Functions in this category

```
unsigned int vpi_mcd_open(filename);
char *filename;

char *vpi_mcd_name(channel_desc);
unsigned int channel_desc;

unsigned int vpi_mcd_close(mcd);
unsigned int mcd;

int vpi_mcd_printf(mcd, format1, format2, ...);
unsigned int mcd;
char *format1, *format2,...;
```

2.7.2 Purpose

The functions shown above are used for file-related activities and are specifically designed to send the output of a simulator (or other tool) to multiple destinations.

Every time `vpi_mcd_open()` is used for opening a file (whose name is passed to it as a string argument), it returns an unsigned integer as the *channel descriptor* (CD) for the file. The channel descriptor numbers 1, 2 and 3 have pre-defined meanings: 1 stands for the standard output, 2 for the standard error and 3 for the log file. A multi-channel descriptor (MCD) is an

unsigned integer whose bit 0 represents the file with CD as 1, bit 1 represents the file with CD as 2 and so on. Since the implementation of the PLI assumes integer to be 32bit, there can be a maximum of 32 open files at a given time. `vpi_mcd_open()` returns 0 on error.

`vpi_mcd_name()` returns the name of an opened file corresponding to a CD, which it takes as the argument. On error it returns a NULL.

`vpi_mcd_printf()` is similar to `fprintf()` function in C, except that it takes an MCD as its first argument. All the files represented by the MCD will be written simultaneously. It returns the number of bytes written.

`vpi_mcd_close()` closes the files mentioned in the MCD. It returns the MCD representing the files which could not be closed.

2.7.3 Example

The following function opens 3 files, writes `"Hello VPI!\n"` to the first and the third, `"Good-bye\n"` to the second and then closes all of them.

```
void print_hello_vpi() {
   unsigned int mcd1 = 0, mcd2 = 0;
   unsigned int cd1, cd2, cd3;
   unsigned int one = 1;

   cd1 = vpi_mcd_open("file1.txt");
   cd2 = vpi_mcd_open("file2.txt");
   cd3 = vpi_mcd_open("file3.txt");

   /* Setting the bits corresponding
      to cd1 and cd3 in mcd1 */
   mcd1 = (one << (cd1-1)) |
             /* setting bit for cd1 */
          (one << (cd3-1));
             /* and then OR-ed with the bit for cd3 */
   vpi_mcd_printf(mcd1, "Hello VPI!\n");

   /* Setting the bit corresponding to cd2 in mcd2 */
   mcd2 = (one << (cd2-1));
   vpi_mcd_printf(mcd2, "Good-Bye\n");

   /* Now combine mcd1 and mcd2 into mcd1 and close them*/
   mcd1 |= mcd2;
   vpi_mcd_close(mcd1);
}
```

2.8 Miscellaneous functions

2.8.1 Displaying messages

2.8.1.1 Function in this category

```
int vpi_printf(format1, format2, ...)
char *format1, *format2, ...;
```

2.8.1.2 Purpose

As we have already seen in earlier examples, `vpi_printf()` does the same job as `printf()` of C, except that apart from printing the arguments to standard output, it also redirects the output to the log file. It returns the number of bytes written or EOF (usually -1 in most systems), if there is an error. Note that, unlike the function `io_printf()` in PLI1.0, there is no limit on the number of arguments.

2.8.1.3 Example

The following example prints the value of an integer variable `i`.

```
int i = 5; /* Some value */
vpi_printf("The value of i is = %d\n", i);
```

2.8.2 Comparing objects

2.8.2.1 Function in this category

```
int vpi_compare_objects(h1, h2)
vpiHandle h1, h2;
```

2.8.2.2 Purpose

This function compares two handles to check whether they are referring to the same object. If the corresponding objects are identical, the function returns a pre-defined integer `true`. Otherswise it returns `false`. It should be noted here that handle is a predefined data structure, and comparison of two handles can not be done using a comparison operator ("==") of C.

2.8.2.3 Example

The conditional statement in the following function takes different actions depending on whether two handles point to the same object.

```
int compare_handles(h1, h2)
vpiHandle h1, h2;
{
```

```
    if (vpi_compare_objects(h1, h2) == true){
       vpi_printf("Handles point to the same object\n");
       return 0;
    } else {
       vpi_printf("Handles point to different objects\n");
       return 1;
    }
}
```

2.8.3 Getting error information

2.8.3.1 Function in this category

```
int vpi_chk_error(ErrInfoPtr)
p_vpi_error_info ErrInfoPtr;
```

2.8.3.2 Purpose

If called *immediately after* a VPI function call, this function returns `true` if there was an error in the function call. It also retrieves the associated error information if optionally a `p_vpi_error_info` type variable is passed (otherwise, a NULL argument can be passed). The structure of the `p_vpi_error_info`, defined in `vpi_user.h`, is given below.

```
typedef  struct t_vpi_error_info {
   int state;
   /* State of the simulation when the error occurred: */
   /* 1. vpiCompile 2. vpiPLI 3. vpiRun                */

   int level;
   /* Severity of the error:                           */
   /* 1. vpiNotice 2. vpiWarning 3. vpiError           */
   /* 4. vpiSystem 5. vpiInternal                      */

   char *message; /* Error message */
   char *product; /* The tool encountering the error */
   char *code;    /* The error code */
   char *file;    /* The file containing
                     the erroneous source */
   int line;      /* The location of the error  */
} s_vpi_error_info, *p_vpi_error_info;
```

Note that, the error information is stored only for the last VPI function call.

2.8.3.3 Example

The following code segment shows how to use `vpi_chk_error()` in case an error occurs in reading the value of an object specified by `objHndl` using the VPI function `vpi_get_value()`.

```
s_vpi_value got_value;

vpi_get_value(objHndl, &got_value);
if (vpi_chk_error(NULL) == true)
   vpi_printf("Error in using vpi_get_value()\n");
```

2.8.4 Product invocation information

2.8.4.1 Function in this category

```
int vpi_get_vlog_info(InfoPtr)
p_vpi_vlog_info InfoPtr;
```

2.8.4.2 Purpose

This function is used to get the information regarding the command line options used during the execution and also the product name and the version. The functions returns `true` on success, `false` on failure.

The data type `p_vpi_vlog_info` is defined in `vpi_user.h` as follows.

```
typedef  struct t_vpi_vlog_info {
   int argc;      /* The number of command line options */
   char **argv;   /* The values of the command line options
                     expressed as string */
   char *product;/* The product name */
   char *version;/* The version of the product */
} s_vpi_vlog_info, *p_vpi_vlog_info;
```

2.8.4.3 Example

The following user-defined function `get_arg(n)` uses the function `vpi_get_vlog_info()` to return the `n`th command line argument as a string.

```
char *get_arg(n)
int n;
{
   s_vpi_vlog_info VlogInfo;

   if (vpi_get_vlog_info(&VlogInfo) == true) {
      if (n > (VlogInfo.argc-1))
```

```
            vpi_printf("Not that many arguments ...\n");
        else
            return(VlogInfo.argv[n]);
    }
}
```

2.8.5 Freeing allocated memory

2.8.5.1 Function in this category

```
int vpi_free_object(handle)
vpiHandle handle;
```

2.8.5.2 Purpose

Although `vpi_free_object()` can be used to free memory allocated for any object, it is mainly used for freeing memory associated with an iterator. The allocated memory to an iterator object is automatically freed (and marked for re-allocation to some other object) when the associated `vpi_scan()` returns NULL after the iteration is complete or some error occurs. However, in a situation where `vpi_scan()` has not returned a NULL, this function can be used to free the allocated memory of the iterator object. The routine returns `true` on success, `false` on failure.

2.8.5.3 Example

The following function `match_module()` searches for a specific top-level module whose handle `moduleHndl` is passed to the function.

```
int match_module (moduleHndl)
vpiHandle moduleHndl;
{
    vpiHandle mItr, mHndl;
    int exists;

    exists = false;
    mItr = vpi_iterate(vpiModule, NULL);

    if (!mItr) {
      vpi_printf("No top level module !\n");
      return;
    }

    while (mHndl = vpi_scan(mItr)) {
      if (vpi_compare_objects(mHndl, moduleHndl)==true) {
```

```
            vpi_printf("Yahoo !!! Got a match...\n");
            vpi_free_object(mItr);
            exists = true;
            break;
        }
    }
    return(exists);
}
```

3. USING VPI: AN X-DETECTOR

Now that we are familiar with the library functions in VPI, let us see how to use them to build applications. We start with an application which helps in detecting unknown states (X) generated by a module.

3.1 Problem description

While simulating a system or ASIC, quite often setting up the simulation environment poses as big a challenge as the simulation itself. A common problem is that an overlooked floating input to one of the sub modules leads to a propagation of Xs through-out the system and subsequently the simulation breaks down.

One way of finding out the floating input is to start from a signal whose value is X and then to backtrack its drivers recursively. However, this procedure is not only pain-staking and time-consuming, it also requires the knowledge about the internal structures of the ASIC, allowing only a few engineers in the design team to do that.

To avoid such a situation, it is better to check the inputs of the module first and then proceed further. Normally the only way to do that is to make a list of all the inputs and then use either a `$display` or `$showvars` at that point. When an ASIC has hundreds of inputs, this is a tedious process.

The following PLI system call `$peripheri` does just this. When invoked, it lists all the ports of the module passed as an argument, describes their types (input, output etc.) and then prints the value of the signal associated with each port. It also prints the current simulation time.

3.2 Program listing

An example of a Verilog code using this system call is shown below.

```
module mymux21test ;

  reg [1:0] a;
  reg s;

  mymux21 m1 (y, a[1], a[0], s);

  initial begin
         a  = 2'b10 ;
         s = 1'b1 ;
  #100   s = 1'b0 ;
  #100   $finish;
  end

  initial  begin
  #10    $peripheri(mymux21test.m1);
         // Use either the full path
  #100   $peripheri(m1);
         // Or the relative path
  end
endmodule

module mymux21(outp, inp1, inp0, select);
output outp;
input inp1, inp0, select;

  assign outp = (select == 1'b1) ? inp1 : inp0;

endmodule
```

After running the simulation, we get the following results.

```
Highest level modules:
mymux21test

 Module m1 :                Time 10
        Output  y :  1
        Input   a :  10
        Input   s :  1

 Module m1 :                Time 110
        Output  y :  0
```

```
Input   a :  10
Input   s :  0
```

The C code for implementing `$peripheri` is given below.

```
#include <stdio.h>
#include "vpi_user.h"

int gnt_task ()
/* This is the calltf routine */
{
vpiHandle netHndl;
vpiHandle modHndl2;
vpiHandle portItr, portHndl;
vpiHandle tfH, argI;

int arg_count;

s_vpi_value got_value;
s_vpi_time time;

char dirn[8];

   arg_count = 1; /* indicates the current PIN */
   /* get a handle to the system call itself */
   tfH = vpi_handle(vpiSysTfCall, NULL ) ;
   if (!tfH) {
          vpi_printf("USAGE :\n");
          vpi_printf("$peripheri(inst_name1, ...);\n");
          return;
   }

   /* Next, get handles to the arguments  */
   /* of the system call                  */
   argI = vpi_iterate(vpiArgument, tfH);
   if (argI) {
      while (modHndl2 = vpi_scan(argI)) {
         /* check that the argument is a module */
         if (vpi_get(vpiType, modHndl2) != vpiModule) {
            vpi_printf("ERROR! %d is not a module\n",
               arg_count);
            return;
```

```
}

/* get handle to the ports of the module */
portItr = vpi_iterate (vpiPort, modHndl2 );
if (!portItr) {
   vpi_printf("%s is top level module\n",
              vpi_get_str(vpiDefName, modHndl2));
   return;
}

/* retrieve the simulation time */
time.type = vpiSimTime;
vpi_get_time(NULL, &time);
if (time.high >0)
   vpi_printf("\n Module %s :Time %d%d\n",
              vpi_get_str(vpiName, modHndl2),
              time.high, time.low);
else
   vpi_printf("\n Module %s :Time %d\n",
              vpi_get_str(vpiName, modHndl2),
              time.low);

got_value.format = vpiBinStrVal;
/* get handle to the net connected to each port*/
while (portHndl = vpi_scan (portItr)) {
   netHndl = vpi_handle(vpiHighConn, portHndl);
   vpi_get_value (netHndl, &got_value);
   /* find out the port direction */
   switch(vpi_get(vpiDirection, portHndl)) {
   case vpiInput:
      strcpy(dirn, "Input");
      break;
   case vpiOutput:
      strcpy(dirn, "Output");
      break;
   case vpiInout:
      strcpy(dirn, "Inout");
      break;
   case vpiMixedIO:
      strcpy(dirn, "Mixed IO");
      break;
   case vpiNoDirection :
```

```
                strcpy(dirn, "No Dirn");
                break;
            }  / * switch */
            vpi_printf("\t%s\t%s :  %s\n",
                dirn, vpi_get_str(vpiName, netHndl),
                got_value.value.str);
        }  /* while portHndl */
    arg_count++;
    } /* while modHndl2 */
  } /* if argI */
}

void registration()
/* Place all the function registration here */
{
s_vpi_systf_data task_data_s;
p_vpi_systf_data task_data_p = & task_data_s;

task_data_p->type = vpiSysTask;
    /* No return value for a task */
task_data_p->tfname = "$peripheri";
    /* HDL task name */
task_data_p->calltf = (int(*)()) gnt_task;
    /* The routine associated with the task */
task_data_p->compiletf = NULL ;
    /*No function, enter Null */
vpi_register_systf(task_data_p) ;
}

/* Enter registration() in the startup array */
void (*vlog_startup_routines[])() = {registration, 0};
```

3.3 Self Check

– Create a testbench similar to the one in the example in Chapter 5 Section 2 but this time using VPI. The testbench should read the stimuli from a text file and apply it to the circuit under test (CUT). Further, it should compare the outputs of the CUT with expected results and flag an error in case of a miscompare.

4. USING VPI: DISPLAYING DRIVERS

4.1 Problem description

The following example shows how to traverse a relationship and extract any property of a related object.

In this example, a system call `$show_drivers()` displays the list of objects which drive a register. This is similar to the system task `$showvars()` in Verilog-XL, but, for simplicity, confines itself only to register type variables. It can be easily extended to nets and other objects.

For a register, there can be two types of drivers - (1) a procedural assignment, and (2) a force statement. Once a handle to the register is available, an iterator to its driver can be obtained by using the property `vpiDriver`. Note that, only active drivers are shown using this process, not the one whose effect was already seen at an earlier point of time.

4.2 Program listing

An example of a Verilog program using this system call is given below. The file containing the program is assumed to be `test.v`. The program is listed along with the line numbers.

```
 1 module test;
 2 reg p;
 3
 4 initial  begin
 5        p = 1'b1;
 6        $show_drivers(test.p);
 7 #100   $show_drivers(test.p);
 8 #100   $show_drivers(test.p);
 9 #100   $show_drivers(test.p);
10 #100   $finish;
11 end
12 endmodule
13
14 module mod;
15 reg q;
16
17 initial begin
18      q = 1'b0;
```

```
19  #25  q = 1'b1;
20  #150 test.p = 1'bz;
21  #100 force test.p = 1'b0;
22 end
23 endmodule
```

Part of the result of running the simulation is as follows. As mentioned earlier, among the four `$show_drivers()`, only the last one shows the active driver (which is the force statement).

```
Compiling source file "test.v"
Highest level modules:
test
mod

Time :  300:
 Driver 1: Force statement  File name = test.v and Line
number = 21
```

Following is the code for the system call `$show_drivers()`.

```
#include <stdio.h>
#include "vpi_user.h"

/****************************************
 * The user C functions
 ****************************************/
int show_drivers_main();

int show_drivers_main()
{
vpiHandle regHndl;
vpiHandle tfH;
vpiHandle argI;
vpiHandle itr, driver;

s_vpi_time time_s;

int driver_cnt, arg_count;
int time_hi, time_lo;

int prop;
```

```
arg_count = 1; /* indicates the PIN */

/* get a handle to the system call itself */
tfH = vpi_handle(vpiSysTfCall, NULL);
if (!tfH){
   vpi_printf("ERROR! Failure to get a task handle\n");
   return;
} else {
   /* get handle to the arguments of the system call */
   if (argI = vpi_iterate(vpiArgument, tfH)) {
      while (regHndl = vpi_scan(argI)) {
         if ((prop = vpi_get(vpiType, regHndl))!=vpiReg) {
            vpi_printf("ERROR! Arg %d is not reg",
                       arg_count);
            tf_dostop();
         }

         /* find out the simulation time */
         time_s.type = vpiSimTime;
         vpi_get_time(NULL, &time_s);

         vpi_printf(" Time : ");
         if (time_s.high != 0)
            vpi_printf(" %d", time_s.high);
         vpi_printf(" %d: \n", time_s.low);

         /* get handle to the drivers */
         itr = vpi_iterate(vpiDriver, regHndl);
         if (!itr) return;

         driver_cnt = 1;
         while (driver = vpi_scan(itr)) {
            /* find out the type of driver */
            switch (vpi_get(vpiType, driver)) {
            case vpiAssignStmt :
               vpi_printf(" Driver %d: AssignStmt ",
                       driver_cnt++);
               break;
            case vpiAssignment :
               vpi_printf(" Driver %d: Assignment ",
                          driver_cnt++);
               break;
```

```
            case vpiForce :
               vpi_printf(" Driver %d: Force statement ",
                          driver_cnt++);
               break;
            default :
               vpi_printf(" Statement type = %d\n",
                          vpi_get(vpiType, driver));
            }
            vpi_printf(" File = %s and Line = %d\n\n",
                       vpi_get_str(vpiFile, driver),
                       vpi_get(vpiLineNo, driver));
            driver = vpi_scan(itr);
         } /* while driver */
      arg_count++;
      } /*while regHndl*/
   } else {
      vpi_printf(" Usage:\n");
      vpi_printf("$show_drivers(reg_fullpathname);\n");
   }
}
   return 0;
}

/****************************************
 * The registration function
 ****************************************/
void registration();
void registration()
/* Place all the function registration here */
{
s_vpi_systf_data task_data_s ;
/* Declare the structure */

task_data_s.type = vpiSysTask;
/* No return value for our system call */
task_data_s.tfname = "$show_drivers";
/* HDL task name */
task_data_s.calltf
    = show_drivers_main;
/* The calltf routine */
task_data_s.compiletf = NULL;
/* No function, enter NULL */
```

```
vpi_register_systf(&task_data_s);
}

/*****************************************
 * The start-up array, the last element
 * must be a zero.
 *****************************************/
void (*vlog_startup_routines[])() =
   {registration, 0};
```

4.3 Self Check

For each gate in a design description, find out the number of fan outs (how many other gates it is driving). If this number is more than a specified maximum for that gate, issue a warning. [Hint: Use the `vpiLoad` property of a net to find out how many gates it is driving]

5. FUTURE DIRECTION

VPI has come a long way in meeting one of the primary requirements of any application program interface – providing consistency of interface to all objects inside the design. By separating the methodology from the data itself, it has also simplified the issue of extendibility.

However, there are several areas where VPI needs improvement. The following are some of the deficiencies of VPI.

1. The first and most apparent of these is the method of traversing the design database. The concept of relationship based traversal is new and many users find it to be unnecessarily complex.
2. Since VPI is supposed to work in a compiled environment, there is no way to get a reference to compiler directives (such as macros) which are replaced during the preprocessing phase.
3. References are restricted to named objects only. In certain cases, while traversing a design, if the application encounters an object reference, it may loose the context.
4. VPI can not create objects dynamically.
5. There is no way to access events inside the simulator.
6. Many of the useful utility functions are missing, such as, equivalents of `$stop` and `$finish`. Also, the initial effort to introduce a mechanism to invoke an interrupt subroutine in PLI1.0 is still unfulfilled.

Hopefully, a future version of IEEE Std. 1364 will include these enhancements.

Apart from enhancing PLI, another effort is going on in parallel in a different, but related field. We have seen how to write functional models in C in Chapter 6. Extending this concept, the next step is to standardize an interface for hardware models written in different languages including VHDL, Verilog, C and others. In order to avoid a separate interface for each language, the Open Model Forum (OMF) was founded to solve the problem of logic model availability. OMF has tackled this problem by defining a programming interface, called Open Model Interface (OMI), between models and simulators. This will allow models developed in one language to be used with simulators that support other languages. Currently in the process of becoming an IEEE standard, OMI will aid the interoperability of models and address the issue of intellectual property protection to a great degree.

6. SUMMARY

Verilog Procedural Interface is the first major effort to separate the interface from the actual data transfer. However, this implementation of a generic and consistent interface has in some cases made programming less intuitive compared to the PLI1.0. This is the main reason why, in spite of being much more sophisticated, VPI could not outdate the earlier version. Hopefully, in future, with more number of Verilog vendors supporting this new interface, VPI will gain popularity to make it commercially successful.

Index

+

+pathpulse, 119
+*pulse_e/n*, 121, 134
+*pulse_r/n*, 121, 134

A

acc routines, 78
acc_append_delays()
  function prototype, 121, 122
    for module path, 121
    for primitive, port, port bit and intermodule path, 121
    for timing check, 122
  purpose of, 123
acc_append_pulsere()
  function prototype, 132
  purpose of, 133
acc_close(), 78
  function prototype, 154
  purpose of, 155
acc_collect(), 147
  example of use, 160, 187
  function prototype, 159
  purpose of, 159
acc_compare_handles()
  function prototype, 160
  purpose of, 161
acc_configure()
  example of use, 102, 120, 124, 158
  function prototype, 154
  purpose of, 155, 157
acc_count(), 147
  example of use, 160, 187
  function prototype, 159
  purpose of, 159
acc_fetch_argc()
  example of use, 99
  function prototype, 99
  purpose of, 99
acc_fetch_argv()
  example of use, 99
  function prototype, 99
  purpose of, 99
acc_fetch_attribute()
  example of use, 102
  function prototype, 100

purpose of, 100
acc_fetch_attribute_int()
function prototype, 100
purpose of, 100
acc_fetch_attribute_str()
function prototype, 100
purpose of, 100
acc_fetch_defname()
function prototype, 103
purpose of, 104
acc_fetch_delay_mode()
example of use, 110
function prototype, 107
purpose of, 108
acc_fetch_delays()
example of use, 124, 127
function prototype, 107, 108
for module path, 108
for primitive, port, port bit and intermodule path, 108
for timing check, 108
purpose of, 109
acc_fetch_direction()
example of use, 117, 127
function prototype, 116
purpose of, 116
acc_fetch_edge()
function prototype, 118
purpose of, 118
acc_fetch_fullname()
example of use, 114, 124, 150
function prototype, 103
acc_fetch_fulltype()
example of use, 188
function prototype, 103
purpose of, 104
acc_fetch_index()
example of use, 117
function prototype, 116
purpose of, 116
acc_fetch_itfarg()
function prototype, 112
purpose of, 113
acc_fetch_itfarg_int()
function prototype, 112
acc_fetch_itfarg_str()
function prototype, 112
acc_fetch_location()
example of use, 106
function prototype, 103
purpose of, 104
acc_fetch_name(), 148
example of use, 147, 148, 162
function prototype, 103
purpose of, 104
acc_fetch_paramtype()
example of use, 102
function prototype, 100
purpose of, 101
acc_fetch_paramval()
example of use, 102
function prototype, 100
purpose of, 101
acc_fetch_polarity()
function prototype, 118
purpose of, 118
acc_fetch_precision()
example of use, 116
function prototype, 114
purpose of, 115
acc_fetch_pulsere()
example of use, 120, 133, 134
function prototype, 118
purpose of, 119
acc_fetch_range()
example of use, 111, 151
function prototype, 111
purpose of, 111
acc_fetch_size()
example of use, 111, 179, 180
function prototype, 111
purpose of, 111
acc_fetch_tfarg()
function prototype, 112

purpose of, 113
acc_fetch_tfarg_int()
example of use, 114
function prototype, 112
purpose of, 113
acc_fetch_tfarg_str()
example of use, 114
function prototype, 112
purpose of, 113
acc_fetch_timescale_info()
example of use, 116, 178
function prototype, 114
purpose of, 115
acc_fetch_type()
example of use, 114
function prototype, 103
purpose of, 104
acc_fetch_type_str()
function prototype, 103
purpose of, 104
acc_fetch_value()
example of use, 141, 148, 151, 180
function prototype, 103
purpose of, 105
acc_free()
example of use, 160
function prototype, 159
purpose of, 160
acc_handle_by_name()
example of use, 86, 102, 187
function prototype, 83
purpose of, 84
acc_handle_calling_mod_m()
function prototype, 86
purpose of, 87
acc_handle_condition()
example of use, 93
function prototype, 90
purpose of, 91
acc_handle_conn()
example of use, 98
function prototype, 97
purpose of, 97
acc_handle_datapath()
function prototype, 90
purpose of, 91
acc_handle_hiconn()
example of use, 90
function prototype, 88
purpose of, 88
acc_handle_interactive_scope()
function prototype, 84
purpose of, 84
acc_handle_itfarg()
function prototype, 98
purpose of, 98
acc_handle_loconn()
example of use, 90
function prototype, 88
purpose of, 88
acc_handle_modpath()
example of use, 92, 120
function prototype, 90
purpose of, 91
acc_handle_notifier()
example of use, 96
function prototype, 93
purpose of, 94
acc_handle_object()
example of use, 251
function prototype, 83
purpose of, 84
acc_handle_parent(
example of use, 87
acc_handle_parent()
example of use, 149, 150, 188
function prototype, 86
purpose of, 87
acc_handle_path()
function prototype, 88
purpose of, 88
acc_handle_pathin()
function prototype, 90
purpose of, 91

acc_handle_pathout()
function prototype, 90
purpose of, 91
acc_handle_port()
example of use, 90, 117
function prototype, 88
purpose of, 88
acc_handle_scope()
example of use, 148
function prototype, 83
purpose of, 84
acc_handle_simulated_net()
function prototype, 97
purpose of, 97
acc_handle_tchk()
example of use, 96
function prototype, 93
purpose of, 94
acc_handle_tchkarg1()
example of use, 96
function prototype, 93
purpose of, 94
acc_handle_tchkarg2()
example of use, 96
function prototype, 93
purpose of, 94
acc_handle_terminal()
example of use, 97
function prototype, 97
purpose of, 97
acc_handle_tfarg()
example of use, 99, 114, 117, 153
function prototype, 98
purpose of, 98
acc_handle_tfinst()
example of use, 251
function prototype, 98
purpose of, 98, 113
acc_initialize(), 78
example of use, 158
function prototype, 154
purpose of, 155
acc_next()
example of, 148
function prototype, 145
purpose of, 146
acc_next_bit(), 143
example of use, 151
function prototype, 150
purpose of, 150
acc_next_cell()
function prototype, 142
purpose of, 143
acc_next_cell_load()
function prototype, 148
purpose of, 149
acc_next_child(), 147
example of use, 144
function prototype, 142
purpose of, 143
acc_next_driver()
example of use, 150
function prototype, 148
purpose of, 149
acc_next_hiconn()
function prototype, 148
purpose of, 149
acc_next_input()
example of use, 153
function prototype, 152
purpose of, 152
acc_next_load()
example of use, 149, 187
function prototype, 148
purpose of, 149
acc_next_loconn()
example of use, 151
function prototype, 149
purpose of, 149
acc_next_modpath()
example of use, 153
function prototype, 142
purpose of, 143
acc_next_net()

example of use, 162
function prototype, 142
purpose of, 143
acc_next_output()
example of use, 154
function prototype, 152
purpose of, 152
acc_next_parameter()
function prototype, 142
purpose of, 143
acc_next_port()
example of use, 141
function prototype, 142
purpose of, 143
acc_next_portout()
example of use, 144
function prototype, 143
purpose of, 144
acc_next_primitive()
function prototype, 143
purpose of, 144
acc_next_scope()
example of use, 148
function prototype, 145
purpose of, 146
acc_next_specparam()
function prototype, 143
purpose of, 144
acc_next_tchk()
function prototype, 143
purpose of, 144
acc_next_terminal()
function prototype, 143
purpose of, 144
acc_next_topmod()
example of use, 147
function prototype, 145
purpose of, 147
acc_object_in_typelist()
example of use, 162
function prototype, 161
purpose of, 161
acc_object_of_type()
example of use, 150
function prototype, 161
purpose of, 161
acc_product_type()
function prototype, 154
purpose of, 157
acc_product_version(), 157
example of use, 158
function prototype, 154
acc_release_object()
example of use, 154
function prototype, 152
purpose of, 152
acc_replace_delays()
example of use, 124, 127
function prototype, 121, 122
for module path, 121
for primitive, port, port bit and intermodule path, 121
for timing check, 122
purpose of, 123
acc_replace_pulsere()
example of use, 134
function prototype, 132
purpose of, 133
acc_reset_buffer()
example of use, 159
function prototype, 154
purpose of, 157
acc_set_interactive_scope()
function prototype, 155
purpose of, 158
acc_set_pulsere()
function prototype, 132
purpose of, 133
acc_set_scope()
example of use, 159, 251
function prototype, 155
purpose of, 157
acc_set_value()
example of use, 131

function prototype, 128
purpose of, 128
acc_user.h, 21, 78
acc_vcl_add()
example of use, 140, 180
function prototype, 135
purpose of, 135
acc_vcl_delete()
function prototype, 135
purpose of, 137
acc_version(), 157
example of use, 158
function prototype, 154
accAndGate, 146, 188
accAssignFlag, 129
accBinStrVal, 105
accBufGate, 146
accBufif0, 146
accBufif1, 146
accCellInstance, 146
accCmosGate, 146
accCollapsedNet, 161
accCombPrim, 146
accDeassignFlag, 129
accDecStrVal, 105
accDefaultAttr0, 101, 155
accDelayModeDistrib, 109
accDelayModeMTM, 109
accDelayModeNone, 109
accDelayModePath, 109
accDelayModeUnit, 109
accDelayModeZero, 109
accDevelopmentVersion, 155
accDisplayErrors, 155
accDisplayWarnings, 155
accEdge01, 94, 118
accEdge0x, 94, 118
accEdge10, 94, 118
accEdge1x, 94, 118
accEdgex0, 94, 118
accEdgex1, 94, 118
accEnableArgs, 94, 156
access routines, 37, 77
accExpandedNet, 161
accExpandedVector, 150
accFaultSimulator, 157
accForceFlag, 129
accHexStrVal, 105, 106
accHold, 94
accInertialDelay, 129
accIntegerParam, 101
accIntegerVar, 146
accIntParam, 101
accIntVal, 106
accMapToMipd, 156
accMinTypMaxDelays, 108, 121, 122, 156
accModPathHasIfNone, 161
accModule, 146
accModuleInstance, 146
accNamedEvent, 146
accNandGate, 146, 188
accNegedge, 94, 118
accNet, 146
accNmosGate, 146
accNochange, 94
accNoDelay, 129
accNoedge, 94, 118
accNorGate, 146, 188
accNotGate, 146
accNotif0, 146
accNotif1, 146
accOctStrVal, 105
accOrGate, 146, 188
accOther, 157
accPathDelayCount, 119, 156
accPathDelimStr, 157
accPeriod, 94
accPmosGate, 146
accPosedge, 94, 118
accPrimitive, 146
accPulldownGate, 146
accPullUpGate, 146
accPureTransportDelay, 129

accRcmosGate, 146
accRealParam, 101
accRealTime, 130
accRealVal, 106
accRealVar, 147
accRecovery, 94
accRegister, 147
accReleaseFlag, 129
accRnmosGate, 146
accRpmosGate, 146
accRtranGate, 146
accRtranif0Gate, 146
accRtranif1Gate, 146
accScalar, 161
accScalarVal, 106
accScope, 161
accSeqPrim, 146
accSetup, 94
accSimTime, 130
accSimulator, 157
accSkew, 94
accStringParam, 101
accStringVal, 105, 106
accSupply0, 146
accSupply1, 146
accTime, 129
accTimeVar, 147
accTimingAnalyzer, 157
accToHiZDelay, 157
accTopModule, 146
accTranGate, 146
accTranif0Gate, 146
accTranif1Gate, 146
accTransportDelay, 129
accTri, 146
accTri0, 146
accTri1, 146
accTriand, 146
accTrior, 146
accTriReg, 146
accUnexpandedNet, 161
accVector, 161
accVectorVal, 106
accWand, 146
accWidth, 94
accWire, 146
accWor, 146
accXnorGate, 146, 188
accXorGate, 146, 188
avalbits, 49

B

BFM, 228
bind(), 195
bvalbits, 49

C

Callback
  definition of, 331
callback mechanism, 27
  definition, 27
  simulation action or feature related, 27
  simulation event related, 27
  simulation time related, 27
calltf, 24, 259
calltf routine
  example of, 7
cbAfterDelay, 333
cbAtStartOfSimTime, 333
cbEndOfCompile, 334
cbEndOfRestart, 334
cbEndOfSave, 334
cbEndOfSimulation, 334
cbEnterInteractive, 334
cbError, 334
cbExitInteractive, 334
cbForce, 332, 333
cbInteractiveScopeChange, 334
cbNextSimTime, 333
cbReadOnlySynch, 333

cbReadWriteSynch, 333
cbRelease, 332, 333
cbStartOfRestart, 334
cbStartOfSave, 334
cbStartOfSimulation, 334
cbStmt, 332
cbTchkViolation, 334
cbUnresolvedSystf, 334
cbValueChange, 332, 333
cell
  definition of, 143
checktf, 22
checktf routine
  example of, 6
circuit under test, CUT, 168
class, 267
client, 195
communication domain, 194
  internet, 194
  UNIX, 194
compiletf, 259
Compiling and linking a PLI routine, 9
connect(), 196
cr_vlog, 9, 35

D

data, 25, 26, 201
delay back annotation, 123
digital logic simulator
  principle of operation, 19

E

endofcompile_routines, 4
err_intercept, 5
error value, 119
event queue, 19

F

fetch routines, 80, 99
fulltype, 104

G

getenv(), 167

H

handle, 263
  declaration, 263
  definition of, 79
handle routines, 79, 83
hiconn, 88

I

IEEE Std. 1364-1995, 18, 77
inertial delay, 46
instance pointer, 73
  functions for handling, 74
internal buffer, 322
io_mcdprintf(), 42
  function prototype, 63
  purpose of, 63
io_printf(), 7, 42
  function prototype, 63
  purpose of, 63
iterator, 266

L

listen(), 195

M

mc_scan_plusargs(), 43
  example of use, 69

function prototype, 68
purpose of, 68
method, 267
example of, 267
miscellaneous routines, 81, 154
misctf, 24
modify routines, 81
module path
definition of, 143

N

next routine
reference object, 141
target object, 141
next routines, 82, 141
nodes
definition of, 53

O

one-to-many relationship, 266
example of, 266
one-to-one relationship, 265
example of, 265
OVI, 4, 18

P

parameter index number, 40
paramvc, 25, 31
Passing a memory variable to system call, 206
PATHPULSE$, 119, 134
PIN. *See* parameter index number
PLI routine example
Getting the simulation time, 10
modifying the value of a register, 11
reading the value of a register, 2
PLI table file. *See* pli.tab
pli.tab, 6
example of, 15, 33
PLI1.0, 18
PLI2.0, 18
Polaris
basic PLI data structures, 4
primitive
definition of, 144
pulse
definition of, 119
pure transport delay, 47
pvc flags, 57

R

race condition, 29
read(), 196
read-only object, 41
reason, 25, 27
reason_disable, 251
reason_endofcompile, 201, 259
reason_paramvc, 249
reason_reactivate, 233, 234
reason_disable, 28, 62
example of use, 190
reason_endofcompile, 30
example of use, 202
reason_endofreset, 31
reason_finish, 29
example of use, 190
reason_force, 30
reason_interactive, 29
reason_paramdrc, 30
reason_paramvc, 58
tf_asynchon() and, 31
tf_asynchon(), tf_asynchoff() and, 30
reason_reactivate, 28, 62
tf_setdelay() and, 28
reason_release, 30
reason_reset, 31
reason_restart, 30
reason_rosynch, 29, 61
reason_save, 30

reason_scope, 30
reason_startofsave, 30
reason_synch, 29, 61
  example of use, 203
reasons, 24
  example of use, 62
registration function, 259
reject value, 119
relationship
  definition of, 264

S

s_cb_data
  definition of, 332
s_tfcell, 32
s_timescale_info
  definition of, 115
s_vpi_delay
  definition of, 325
s_vpi_strengthval
  definition of, 328
s_vpi_systf_data, 259
s_vpi_time
  definition of, 325
s_vpi_vecval
  definition of, 328
server, 195
sizetf, 23, 259
  the default return size of, 24
socket, 194
socket(), 194
stream, 194
strtol(), 249
system call, 1
system calls, 18
system(), 166

T

t_acc_time
  definition of,, 128
t_acc_vecval
  definition of, 106
t_setval_delay
  definition of,, 128
t_strengths
  definition of,, 136
t_strengthval
  definition of, 54
t_tfexprinfo
  definition of, 48
t_tfnodeinfo
  definition of, 53
t_vc_record
  definition of,, 135
t_vecval
  definition of, 49
t_vpi_error_info
  definition of, 340
t_vpi_vlog_info
  definition of, 341
tag
  definition of, 268
tf_add_long(), 43
  example of use, 70
  function prototype, 69
  purpose of, 70
tf_asynchoff(), 42
  function prototype, 57
  purpose of, 57
tf_asynchon(), 42
  example of use, 58, 248
  function prototype, 56
  purpose of, 57
tf_clearalldelays(), 42
  function prototype, 61
  purpose of, 62
tf_compare_long(), 43
  function prototype, 69
  purpose of, 70
tf_copypvc_flag(), 42
  function prototype, 57

purpose of, 57
tf_divide_long(), 43
function prototype, 69
purpose of, 70
tf_dofinish(), 43
example of use, 68
function prototype, 67
purpose of, 68
tf_dostop(), 43
example of use, 68, 247
function prototype, 67
purpose of, 67
tf_error(), 7, 43
example of use, 167, 247
function prototype, 63
tf_evaluatep(), 42
function prototype, 48
tf_exprinfo
example of use, 51
tf_exprinfo(), 42
function prototype, 48
purpose of, 48
tf_getcstringp(), 42
function prototype, 45
purpose of, 46
tf_getinstance(), 43
example of use, 73
function prototype, 64
purpose of, 65
tf_getlongp(), 42
function prototype, 44
purpose of, 44
tf_getlongp().
example of use, 45
tf_getlongtime(), 42
example of use, 60
function prototype, 59
purpose of, 59
tf_getnextlongtime(), 42
function prototype, 61
purpose of, 61
tf_getp(), 7, 42
Example of use, 40, 250
function prototype, 43
purpose of, 44
tf_getpchange(), 42
example of use, 58
function prototype, 57
purpose of, 58
tf_getrealp(), 42
function prototype, 44
purpose of, 44
tf_getrealtime(), 42
function prototype, 59
purpose of, 59
tf_gettime(), 10, 42
function prototype, 59
purpose of, 59
tf_gettimeprecision(), 42
example of use, 60
function prototype, 60
purpose of, 60
tf_gettimeunit(), 42
function prototype, 60
purpose of, 60
tf_getworkarea(), 43
example of use, 66
function prototype, 65
purpose of, 65
tf_integer_node, 53
tf_long_to_real(), 43
function prototype, 70
purpose of, 71
tf_longtime_tostr(), 43
example of use, 71
function prototype, 70
purpose of, 71
tf_memory_node, 54
tf_message(), 43
example of use, 64
function prototype, 63
purpose of, 64
tf_mipname(), 43
example of, 65

function prototype, 64
purpose of, 65
tf_movepvc_flag(), 42
example of use, 58
function prototype, 57
purpose of, 57
tf_multiply_long(), 43
function prototype, 69
purpose of, 70
tf_netscalar_node, 54
tf_netvector_node, 53
tf_nodeinfo(), 42
example of use, 56, 209
function prototype, 52
purpose of, 53
tf_null_node, 53
tf_nullparam, 49
tf_nump(), 7, 43
example of use, 67, 167, 186, 220
function prototype, 66
purpose of, 66
tf_propagatep()
function prototype, 52
tf_propagetep(), 42
tf_putlongp(), 42
function prototype, 44
purpose of, 44
tf_putp(), 42
example of use, 249
function prototype, 44
purpose of, 44
tf_putrealp(), 42
function prototype, 44
purpose of, 44
tf_read_restart(), 43
example of use, 72
function prototype, 71
purpose of, 71
tf_readonly, 49
tf_readonlyreal, 49
tf_readwrite, 49
tf_readwritereal, 49
tf_real_node, 53
tf_real_to_long(), 43
function prototype, 70
purpose of, 71
tf_reg_node, 53
tf_rosynchronize(), 30, 42
function prototype, 61
purpose of, 61
tf_rwbitselect, 49
tf_rwmemselect, 49
tf_rwpartselect, 49
tf_scale_longdelay(), 43
function prototype, 74
purpose of, 75
tf_scale_realdelay(), 43
function prototype, 74
purpose of, 75
tf_setdelay(), 42
example of use, 62, 232, 235
function prototype, 61
purpose of, 62
tf_setlongdelay(), 42
function prototype, 61
purpose of, 62
tf_setrealdelay(), 42
function prototype, 61
purpose of, 62
tf_setworkarea(), 43
example of use, 66
function prototype, 65
purpose of, 65
tf_sizep(), 43
example of use, 67
function prototype, 66
purpose of, 66
tf_spname(), 43
example of, 65
function prototype, 64
purpose of, 65
tf_strdelputp
function prototype, 45
tf_strdelputp(), 12, 42

example of use, 47, 172, 203, 210, 223, 234, 235
purpose of, 46
tf_strgetp(), 12, 42
example of use, 47, 167, 171, 203, 249
function protoytpe, 45
purpose of, 46
tf_strgettime(), 42
example of use, 141
function prototype, 59
purpose of, 59
tf_string, 49
tf_strlongdelputp
function prototype, 45
tf_strlongdelputp(), 42
purpose of, 46
tf_strrealdelputp
function prototype, 45
tf_strrealdelputp(), 42
purpose of, 46
tf_subtract_long(), 43
function prototype, 69
purpose of, 70
tf_synchronize(), 42
example of use, 62
function prototype, 61
purpose of, 61
tf_testpvc_flag(), 42
function prototype, 57
purpose of, 57
tf_text(), 43
function prototype, 63
purpose of, 63
tf_time_node, 53
tf_typep(), 7, 43
example of use, 67, 167, 186, 220, 247
function prototype, 66
purpose of, 66
tf_unscale_longdelay(), 43
function prototype, 74
purpose of, 75
tf_unscale_realdelay(), 43
function prototype, 75
purpose of, 75
tf_warning(), 42
function prototype, 63
tf_write_save(), 43
example of use, 72
function prototype, 71
purpose of, 71
type, 104

U

User defined system call, 18
user-defined function, 23
user-defined task, 23
user-defined task or function, 1
usertask, 6
utility routines, 37, 39

V

Value Change Dump file. See VCD file
vc_reason
event_value_change, 136
integer_value_change, 136
logic_value_change, 136
real_value_change, 136
sregister_value_change, 136
strength_value_change, 136
time_value_change, 136
vector_value_change, 136
vregister_value_change, 136
VCD file, 174
$dumpvars, 175
$end, 175
$enddefinitions, 175
$scope, 175
$timescale, 175
$upscope, 175

$var, 175
format of, 175
VCL routines, 81
vcl_verilog_logic, 135
vcl_verilog_strength, 135
vconfig, 2, 35, 262
VCS
basic PLI data structures, 4
header file, 3
Integrating a PLI routine in, 33
pli.tab. *See* pli.tab
using access routines with, 162
vcsuser.h, 3
Verilog Procedural Interface. *See* VPI
Verilog-XL
header files, 3
Integrating a PLI routine in, 32
veriuser.c, 2
veriuser.h, 3, 21
veriuser_version_str, 4
veriusertfs, 4, 21, 32
example of, 5
vpi_chk_error(), 322
example of use, 341
function prototype, 340
purpose of, 340
vpi_compare_objects(), 322
example of use, 340
function prototype, 339
purpose of, 339
vpi_free_object(), 322
function prototype, 342
purpose of, 342, 343
vpi_get(), 270, 321
example of use, 324
function prototype, 323
purpose of, 324
vpi_get_cb_info(), 319, 322
function prototype, 331
purpose of, 334
vpi_get_delay(), 308
vpi_get_delays(), 280, 281, 293, 297, 299, 300, 302, 321
example of use, 326
function prototype, 324
purpose of, 324
vpi_get_str(), 321
example of use, 324
function prototype, 323
purpose of, 324
vpi_get_systf_info(), 278, 322
function prototype, 335
purpose of, 336
vpi_get_time(), 320, 322
example of use, 331
function prototype, 330
purpose of, 330
vpi_get_value(), 282, 284, 285, 286, 289, 290, 291, 294, 295, 296, 306, 321
example of use, 330
function prototype, 327
purpose of, 327
vpi_get_vlog_info(), 322
example of use, 341
function prototype, 341
purpose of, 341
vpi_handle
function prototype, 336
vpi_handle(), 322
example of use, 265, 268
purpose of, 337
vpi_handle_by_index(, 322
vpi_handle_by_index(), 321
example of use, 323
function prototype, 322
purpose of, 323
vpi_handle_by_name(), 321
example of use, 323
function prototype, 322
purpose of, 323
vpi_handle_multi
function prototype, 336

vpi_handle_multi(), 322
purpose of, 337
vpi_iterate
function prototype, 336
vpi_iterate(), 266, 322
example of use, 268
vpi_iterator()
purpose of, 337
vpi_mcd_close(), 322
function prototype, 337
purpose of, 338
vpi_mcd_name(), 322
function prototype, 337
purpose of, 338
vpi_mcd_open(), 322
example of use, 338
function prototype, 337
purpose of, 337
vpi_mcd_print(), 322
vpi_mcd_printf()
example of use, 338
function prototype, 337
purpose of, 338
vpi_printf(, 339
vpi_printf(), 322, 339
example of use, 339
function prototype, 339
purpose of, 339
vpi_put_delays(), 280, 293, 297, 299, 300, 321
example of use, 326
function prototype, 324
purpose of, 324
vpi_put_value(), 282, 284, 285, 286, 289, 321
example of use, 330
function prototype, 327
purpose of, 329
vpi_register_cb(), 322, 332
example of use, 335
function prototype, 331
vpi_register_systf(), 259, 322
example of use, 262
function prototype, 335
purpose of, 336
vpi_remove_cb(), 322, 332
function prototype, 331
purpose of, 334
vpi_scan
function prototype, 336
vpi_scan(), 266, 322
purpose of, 337
vpi_str_get(), 270
vpi_user.h, 21, 258
vpiAlways, 318
vpiAndPrim, 293
vpiAnyedge, 298, 301
vpiArgument, 270
vpiArray, 285
vpiAssignment, 307, 316
vpiAssignStmt, 314
vpiBegin, 317
vpiBinaryConst, 290, 291, 305
vpiBinStrVal, 327
vpiBit, 268, 284
vpiBlocking, 307
vpiBufif0Prim, 293
vpiBufif1Prim, 293
vpiBufPrim, 293
vpiCancelEvent, 329
vpiCase, 312, 316
vpiCaseExact, 313
vpiCaseItem, 313
vpiCaseType, 313
vpiCaseX, 313
vpiCaseZ, 313
vpiCellInstance, 272
vpiChargeStrength, 282
vpiCmosPrim, 293
vpiCombPrim, 293, 294, 295, 296
vpiCondition, 298, 308, 309, 310, 311, 312, 313
vpiConnByName, 280
vpiConstant, 305

vpiConstType, 290, 291, 305
vpiContAssign, 302
vpiDataPolarity, 297
vpiDeassign, 315, 316
vpiDecConst, 290, 291, 305
vpiDecStrVal, 328
vpiDefDecayTime, 272
vpiDefDelayMode, 272, 273
vpiDefFile, 272
vpiDefLineNo, 272
vpiDefName, 272, 293, 295
vpiDefNetType, 271, 272
vpiDefParam, 291
vpiDelay, 283, 284, 294, 298, 300, 302, 308
vpiDelayControl, 307, 316
vpiDelayModeDistrib, 272, 273
vpiDelayModeMTM, 272, 273
vpiDelayModeNone, 272, 273
vpiDelayModePath, 272, 273
vpiDelayModeUnit, 272, 273
vpiDelayModeZero, 272, 273
vpiDirection, 278, 279, 280, 294, 295, 298
vpiDisable, 315, 316
vpiDriver, 283, 284, 285, 348
vpiEdge, 298, 301
vpiEdge01, 298, 301
vpiEdge0x, 298, 301
vpiEdge10, 298, 301
vpiEdge1x, 298, 301
vpiEdgex0, 298, 301
vpiEdgex1, 298, 301
vpiElseStmt, 312
vpiEventControl, 308, 316
vpiEventStmt, 306, 316
vpiExpanded, 281
vpiExplicitName, 280
vpiExplicitScalared, 282
vpiExplicitVectored, 282
vpiExpr, 305, 316
vpiFile, 273, 276, 277, 279, 280, 282, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 295, 296, 297, 298, 300, 301, 302, 304, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 317, 319
vpiFor, 310, 316
vpiForce, 314, 316
vpiForceFlag, 329
vpiForever, 311, 316
vpiForIncStmt, 311
vpiForInitStmt, 311
vpiFork, 317
vpiFullName, 273, 275, 280, 282, 284, 285, 286, 287, 288, 289, 290, 293, 303
vpiFuncCall, 277, 305
vpiFunction, 276
vpiGate, 292
vpiHexConst, 290, 291, 305
vpiHexStrVal, 328
vpiHighConn, 281, 283, 284, 286
vpiHighZ, 273
vpiHold, 300
vpiIf, 312, 316
vpiIfElse, 312, 316
vpiImplicitDecl, 281
vpiIndex, 283, 285, 287, 289
vpiInertialDelay, 329
vpiInitial, 318
vpiInout, 278, 280, 294, 295, 298
vpiInput, 278, 280, 294, 295, 298
vpiIntegerVar, 285
vpiInterModPath, 299
vpiIntVal, 328
vpiIODecl, 278
vpiLeftRange, 283, 284, 286, 289, 304
vpiLhs, 291, 292, 302, 307, 314, 315
vpiLineNo, 273, 276, 277, 279, 280, 282, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 295, 296, 297, 298, 300, 301, 302, 304, 305, 306, 307,

308, 309, 310, 311, 312, 313, 314, 315, 317, 319
vpiLoad, 283, 284, 285
vpiLowConn, 281, 283, 284, 286
vpiMemory, 288, 303
vpiMemoryWord, 288, 303
vpiMixedIO, 278, 280, 294, 295, 298
vpiModDataPathIn, 299
vpiModPath, 297
vpiModPathHasIfnone, 297
vpiModPathIn, 299
vpiModPathOut, 299
vpiModule, 268, 272
vpiName, 269, 273, 275, 277, 280, 282, 284, 285, 287, 288, 289, 290, 293, 303
vpiNamedBegin, 317
vpiNamedEvent, 287
vpiNamedFork, 317
vpiNandPrim, 293
vpiNegedge, 298, 301
vpiNet, 281, 302
vpiNetBit, 281
vpiNetDeclAssign, 282, 302
vpiNetType, 282
vpiNmosPrim, 293
vpiNoChange, 300
vpiNoDelay, 329
vpiNoDirection, 280, 294, 295, 298
vpiNoEdge, 298, 301
vpiNorPrim, 293
vpiNotIf0Prim, 293
vpiNotIf1Prim, 293
vpiNotPrim, 293
vpiNullStmt, 316
vpiObjTypeVal, 328
vpiOctConst, 290, 291, 305
vpiOctStrVal, 328
vpiOperation, 304, 305
vpiOpType, 304
vpiOrPrim, 293
vpiOutput, 278, 280, 294, 295, 298
vpiParamAssign, 292
vpiParameter, 289, 303
vpiParent, 267, 284, 287, 289, 306, 320
vpiPartSelect, 303, 305
vpiPathTerm, 298
vpiPathType, 297
vpiPeriod, 300
vpiPmosPrim, 293
vpiPolarity, 297
vpiPort, 279, 283
vpiPortBit, 279
vpiPortIndex, 280
vpiPortInst, 283, 284, 286
vpiPosedge, 298, 301
vpiPrimitive, 292
vpiPrimTerm, 294
vpiPrimType, 293, 294, 295, 296
vpiProcess, 318
vpiProtected, 273, 295
vpiPull0, 273
vpiPull1, 273
vpiPulldownPrim, 293, 294
vpiPullupPrim, 293, 294
vpiPureTransportDelay, 329
vpiRcmosPrim, 293
vpiRealConst, 290, 291, 305
vpiRealVal, 328
vpiRealVar, 285
vpiRecovery, 300
vpiReg, 283, 302
vpiRegBit, 283
vpiRelease, 315, 316
vpiReleaseFlag, 329
vpiRepeat, 309, 316
vpiRepeatControl, 309
vpiRhs, 292, 302, 307, 314
vpiRightRange, 283, 284, 286, 289, 304
vpiRnmosPrim, 293
vpiRpmosPrim, 293
vpiRtranif0Prim, 293
vpiRtranif1Prim, 293
vpiRtranPrim, 293

vpiScalar, 279, 280, 282, 284
vpiScalarVal, 328
vpiScaledRealTime, 325, 333
vpiSeqPrim, 293, 294, 295, 296
vpiSetup, 300
vpiSetupHold, 300
vpiSimTime, 325, 333
vpiSize, 279, 280, 282, 284, 286, 288, 289, 294, 295, 296, 306
vpiSkew, 300
vpiSpecParam, 290
vpiStmt, 313, 318
vpiStrength0, 282, 294, 302
vpiStrength1, 282, 294, 302
vpiStrengthVal, 328
vpiStringConst, 290, 291, 305
vpiStringVal, 328
vpiSupply0, 272, 282
vpiSupply1, 272, 282
vpiSupressTime, 333
vpiSwitch, 292
vpiSysFuncCall, 277, 305
vpiSysFunction, 260
vpiSysFunctionInt, 260
vpiSysFunctionReal, 260
vpiSysFunctionSized, 260
vpiSysFunctionTime, 260
vpiSysFuncType, 277
vpiSysTask, 260
vpiSysTaskCall, 277
vpiTableEntry, 296
vpiTask, 276
vpiTaskCall, 277
vpiTchk, 300
vpiTchkDataTerm, 301
vpiTchkNotifier, 300
vpiTchkRefTerm, 301
vpiTchkTerm, 301
vpiTchkType, 300
vpiTermIndex, 295
vpiTimePrecision, 273
vpiTimeQueue, 320
vpiTimeUnit, 273
vpiTimeVal, 328
vpiTimeVar, 285
vpiTopModule, 273
vpiTranif0Prim, 293
vpiTranif1Prim, 293, 294
vpiTranPrim, 293
vpiTransportDelay, 329
vpiTri, 272, 282
vpiTri0, 272, 282
vpiTri1, 272, 282
vpiTriAnd, 272, 282
vpiTriOr, 272, 282
vpiTriReg, 272, 282
vpiUdp, 292
vpiUdpDefn, 295
vpiUnconnDrive, 273
vpiUse, 303
vpiUserdefn, 277
vpiUserSystf, 278
vpiVariables, 285, 302
vpiVarSelect, 286, 303
vpiVector, 279, 280, 282, 284
vpiVectorVal, 328
vpiWait, 309, 316
vpiWand, 272, 282
vpiWhile, 309, 316
vpiWidth, 300
vpiWire, 272, 282
vpiWor, 272, 282
vpiXnorPrim, 293
vpiXorPrim, 293
vxl_veriuser.h, 3

W

Workarea, 65
writable object, 40
write(), 196