Summer Fellowship Report

On

eSim - NGHDL

Submitted by

Rahul Paknikar

Neel Shah

Under the guidance of

Prof.Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

July 18, 2019

Acknowledgment

It is really a pleasure to acknowledge the help and support that has gone to in making this successful project. We express sincere gratitude to Prof. Kannan Moudgalya and FOSSEE Team at IIT-Bombay for providing us with this opportunity to work on this project and also having faith in our abilities. We would also like to express our gratitude towards Prof. Madhav Desai for showing us a clear path to simulate the process. We would like to thank our mentors Saurabh Bansode & Gloria Nandihal and fellow interns Mallikarjun Reddy & Bharghav Katakam for helping us out with all the problems that we faced while simulation and making our fellowship experience an enjoyable one. Lastly we would like to thank Usha Vishwanathan and Vineeta Gharvi mam for helping us to communicate and manage the meetings with the professors.

Contents

List of Figures

Chapter 1

Introduction

1.1 VHDL

VHDL is an acronym for Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (HDL), which is a programming language used to describe a logic circuit by function, data flow behavior, or structure.

1.2 GHDL

GHDL is a shorthand for G Hardware Design Language (currently, G has no meaning). It is a VHDL compiler that can execute (nearly) any VHDL program. GHDL is not a synthesis tool: you cannot create a netlist with GHDL (yet).Unlike some other simulators, GHDL is a compiler: it directly translates a VHDL file to machine code, without using an intermediary language such as C or C++.

1.3 NGSPICE

Ngspice is a mixed-level/mixed-signal circuit simulator based on three open source software packages: Spice3f5, Cider1b1 and Xspice:

- Xspice is an extension to Spice3 that provides code modeling support and simulation of digital components through an embedded event driven algorithm.

Since GHDL can't generate netlist files for ngspice simulations, the aim here is to generate code models using xspice and simulate digital as well as mixed-mode simulation through embedded event driven algorithm, in this case socket programming between GHDL and ngspice.

Chapter 2

Installing NGHDL

2.1 GHDL

As of July 2019, GHDL doesn't have a package in the Ubuntu sources, or a PPA to download GHDL from, so we need to build GHDL from source.
We need to use either `'LLVM'` backend or `'GCC'` backend, since we need to interface GHDL with C here, which needs `-Wl` flag, which requires either LLVM or GCC backend.
For building with LLVM backend has a few dependencies as follows-

1. make
2. gnat
3. llvm
4. clang
5. zlib1g-dev

After installing there, steps for GHDL with LLVM backend is as follows, *(after downloading from source)*-

```
./configure --with-llvm-config
make
make install
```

2.2 Ngspice

Unlike GHDL, ngspice has a package in the Ubuntu sources, so it can be downloaded directly, but here we need to build from source again as we need to do two things -

1. Add GHDL code models, so that GHDL code can be executed at runtime
2. Add a patch to handle orphan testbench processes.
   Orphan testbench processes are those simulation which have run until completion and plotted, but the testbench files are still executing in an infinite loop

For adding GHDL codemodels, so that they get executed at runtime, we need to-

1. Add GHDL to `GNUmakefile` in the codemodels list
2. Add GHDL code models to spinit file, to include them at runtime.*(Also create the appropriate directories for GHDL)*

For handling orphan testbench process, a log of testbench process id is kept in `/tmp` folder, under `NGHDL...` filename, it contains process id for the currently running testbenches, they are read and killed.
The patch can be found here

Before building Nspice the above 2 patches have to be applied, then it needs to be installed at the appropriate location

2.3 Others

Other dependency are installed when you install eSim. They include python and pyqt4. Also make sure to create symlink if they aren't created for Ngspice, GHDL or Nghdl.

Final modified installation script can be found here.

Chapter 3

Nghdl workflow

There are primarily two steps to simulating your vhdl code using nghdl

1. Executing nghdl generates codemodel and DUT files for ngspice
2. Executing ngspice to exchange messages between GHDL and ngspice
   This is where the socket programming part comes into picture

3.1 Generating Codemodel (ngspice) and DUT files

When ngspice executes your netlist files, it parses the models used and searches for the same in its libraries. If we want to use our own library, we can add those using codemodels which are supported by xspice.

In this case, the codemodel files are generated by nghdl. Nghdl has a few steps as follows-

1. **Create model directory**
   Directory is created with name of the model to be generated at- `/ngspice-nghdl/src/xspice/icm/ghdl`. This directory shall contain all the files generated for the particular model
2. **Adding model in modpath**
   The name of the model that is created is added at modpath.lst, can be found at `~/ngspice-nghdl/src/xspice/icm/ghdl` this contains the list of all the ghdl models. When `make` is getting executed for ngspice, the modpath.lst is read and then respective models are generated to codemodel files.
3. **Generating model files**
   The vhdl file is parsed for it's input and output ports and accordingly models are generated, this is done by `/nghdl/src/model_generation.py`. The files that are generated are -
   - connection_info.txt

- cfunc.mod
- ifspec.ifs
- modelName_tb.vhdl
- start_server.sh
- sock_pkg_create.sh

4. **Generate codemodel files**
   Finally `make` and `make install` are called for ngspice, which goes through all the the codemodels *(ghdl)* as well, and then goes through their `modpath.lst` as well, and accordingly executes all the models mentioned, generates `cfunc.c` and links them under `ghdl.cm` in our case
   Also, here the `cfunc.mod` file is converted to `cfunc.c` file format by actually passing the data provided by ngspice, which is stored in `mif_private`, the strcutre for this data can be found at-
   `/ngspice-nghdl/src/include/ngspice/mifcmdat.h` file.

   `mif_private` consists of all the data ngspice sends, the live data as well as the limits, such as stop time for simulation, current time, current input loads, among others.

This is the first part of nghdl, which generates the codemodel files as well as the DUT files that contain testbench file for the model mentioned.

In case nghdl gets stuck while generating the codemodel files, make sure that you delete all the files corresponding to the model from `/ngspice-nghdl`, since when next time `make` and `make install` are executed they will go through the `modpath.lst` and all the models are generated, even the ones that didn't work.
So, remove the models that didn't work incase nghdl stops working, the models are places primarily at-

1. /ngspice-ghdl/src/xspice/icm/ghdl
2. /ngspice-ghdl/src/xspice/icm/ghdl/modpath.lst
3. /ngspice-ghdl/release/src/xspice/icm/ghdl

3.2 Ngspice - GHDL interaction

When the ngspice simulation is called for digital or mixed model circuits, the following steps taking place-

1. `cfunc.c` file located at the release folder- */ngspice-nghdl/release/src/xspice/icm/ghdl/modelName* is called, this is the client side

2. Parameters are passed from ngspice to this file, using `mif_private` structure, and accordingly parameters such as socketid, time-limit, load, among others are set

3. Once the parameters are set, the `start_server.sh` file is called from `/ngspice-nghdl/src/xspice/icm/ghdl/modelName/DUTghdl`, and the portid parameter is passed

4. The above step, analyzes the `.vhdl` files and finally, elaborates and interfaces `ghdlserver.o` and `modelName_tb` file, and lastly executes the `modelName_tb`

5. `modelName_tb` file contains the calls to the `ghdlserver` file, namely-

   - Vhpi_Initialize
     Initialises server, setup the socket, bind it, and start listening to it
   - Vhpi_Send
   - Vhpi_Listen
   - Vhpi_Get_Port_Value
   - Vhpi_Set_Port_Value

   Vhpi commands are used for interaction of vhdl code with C code

6. This testbench file, initialize, listens and sends data to and from `ghdlserver.c` file using the `Vhpi` commands

7. On the client side *(cfunc.c)* meanwhile, time is checked for each event, and then accordingly data is loaded from `mif_private` and sent to server, and when time is over, `END` signal is sent by the client, which closes the connection.

Chapter 4

Generating Kicad library files

Kicad components are just boxes with no inner logic and can be generating with just the port info and no other logic required, and will be same for same number of ports.

- `createKicadLibrary.py` file creates kicad library components by using the template defined in `Appconfig.py` under `kicad_lib_template`
- The generated kicad components are stored at `eSim_Kicad.lib` in the home directory.
- Also, `-e` flag is required to activate creating kicad libraries, if you are running nghdl from the terminal. ie-
  `sudo nghdl -e`, this will produce the library component as well, just
  `sudo nghdl` won't generate library component.

Chapter 5

Limitations with Earlier Version

- If there are more than one input/output in the declaration, then we need to declare using std logic, but can't declare signal.
  Example: "*in std_logic;*" but cannot declare "*in std_logic(3 downto 0);*".
- NGHDL works with only one output and cannot implement any device with two and more outputs (Socket Programming).
- The declaration of all the ports in VHDL should be std_logic_vector only.
- The declaration of every input and output port should be done in a new VHDL statement separately.
- The VHDL Testbench never terminates for all the instances of Ngspice except the first one (Infinite Loop Issue).
- The GHDL compilation fails for Arithmetic operations to a vector in VHDL.
- The names of input and output ports in VHDL code should not be same for different VHDL examples.
- It is not possible to use Structural Style of VHDL.

Chapter 6

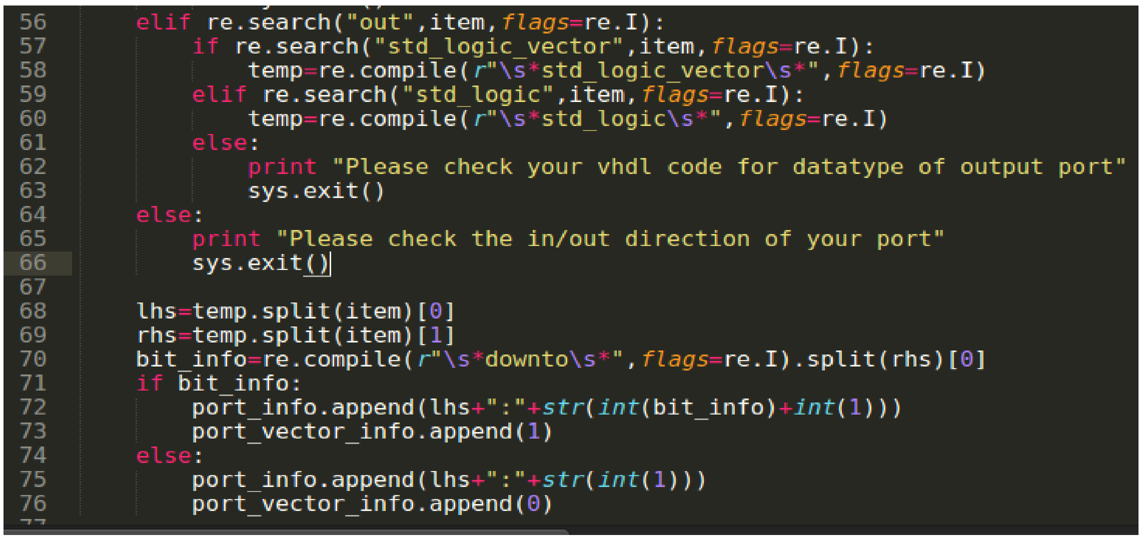
Declaration of Ports

6.1 Issue

The declaration of ports in VHDL were restricted to only '*std_logic_vector*'. It is possible to use '*std_logic*' for port declarations in GHDL without any elaboration error. However, when '*std_logic*' ports are simulated through Ngspice, the simulation fails giving errors.

6.2 Methodology

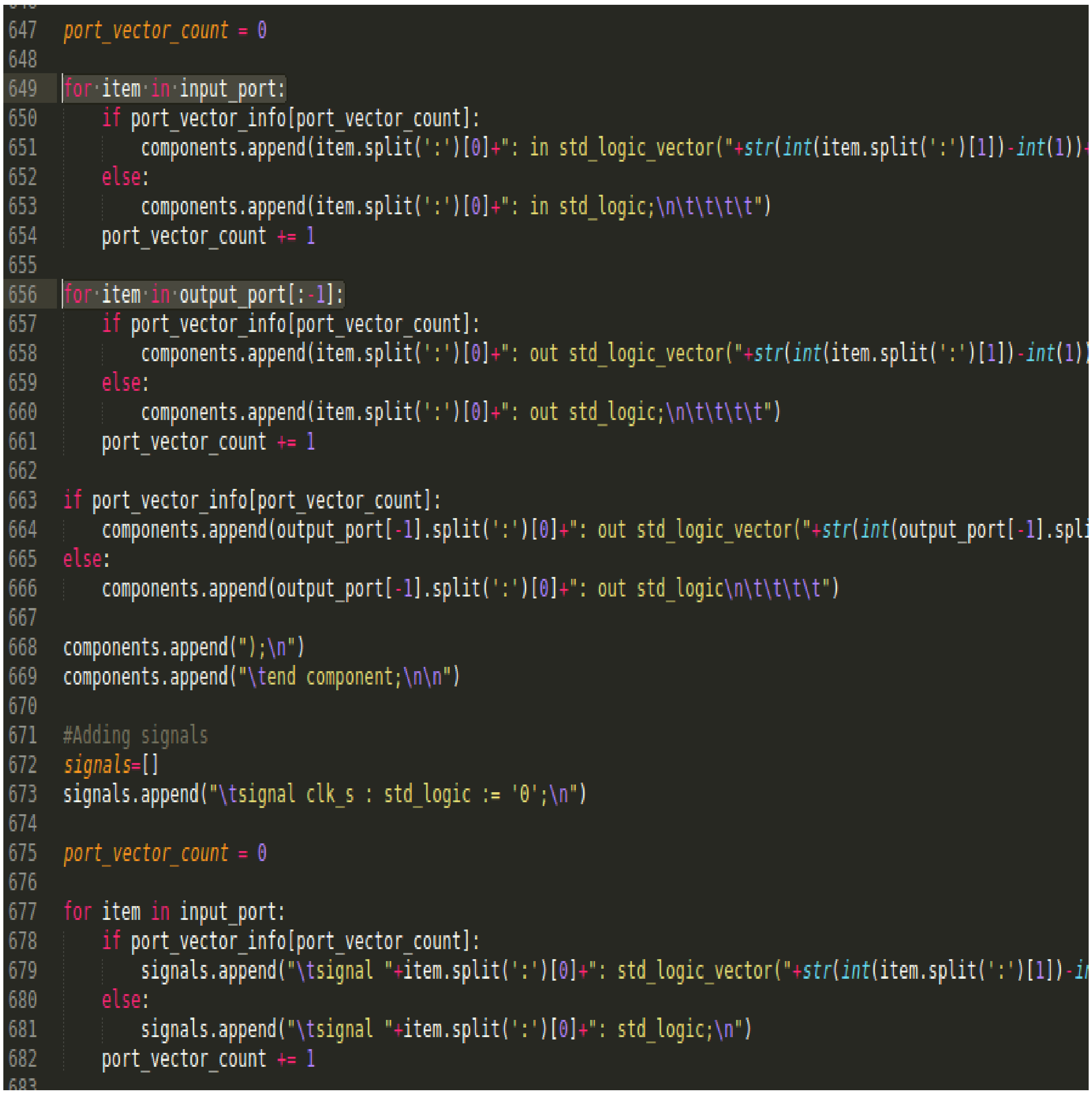
This issue is traced down to the Testbench file that is auto-generated for the corresponding VHDL code. The file '*model_generation.py*' in the '*src*' folder of NGHDL is responsible for generating this Testbench. This file, however, has been coded to work with only '*std_logic_vector*'.



```
    elif re.search("out",item,flags=re.I):
        if re.search("std_logic_vector",item,flags=re.I):
            temp=re.compile(r"\s*std_logic_vector\s*",flags=re.I)
        elif re.search("std_logic",item,flags=re.I):
            temp=re.compile(r"\s*std_logic\s*",flags=re.I)
        else:
            print "Please check your vhdl code for datatype of output port"
            sys.exit()
    else:
        print "Please check the in/out direction of your port"
        sys.exit()

    lhs=temp.split(item)[0]
    rhs=temp.split(item)[1]
    bit_info=re.compile(r"\s*downto\s*",flags=re.I).split(rhs)[0]
    if bit_info:
        port_info.append(lhs+":"+str(int(bit_info)+int(1)))
        port_vector_info.append(1)
    else:
        port_info.append(lhs+":"+str(int(1)))
        port_vector_info.append(0)
```

Figure 6.1: Port Declaration - port_ vector_info

Thus, we need to modify the code to adjust the '*std_logic*' ports. While scanning for ports in the VHDL file, a list of flags called '*port_vector_info*' is maintained as



```
port_vector_count = 0

for item in input_port:
    if port_vector_info[port_vector_count]:
        components.append(item.split(':')[0]+": in std_logic_vector("+str(int(item.split(':')[1])-int(1))+
    else:
        components.append(item.split(':')[0]+": in std_logic;\n\t\t\t\t")
    port_vector_count += 1

for item in output_port[:-1]:
    if port_vector_info[port_vector_count]:
        components.append(item.split(':')[0]+": out std_logic_vector("+str(int(item.split(':')[1])-int(1))
    else:
        components.append(item.split(':')[0]+": out std_logic;\n\t\t\t\t")
    port_vector_count += 1

if port_vector_info[port_vector_count]:
    components.append(output_port[-1].split(':')[0]+": out std_logic_vector("+str(int(output_port[-1].spli
else:
    components.append(output_port[-1].split(':')[0]+": out std_logic\n\t\t\t\t")

components.append(");\n")
components.append("\tend component;\n\n")

#Adding signals
signals=[]
signals.append("\tsignal clk_s : std_logic := '0';\n")

port_vector_count = 0

for item in input_port:
    if port_vector_info[port_vector_count]:
        signals.append("\tsignal "+item.split(':')[0]+": std_logic_vector("+str(int(item.split(':')[1])-i
    else:
        signals.append("\tsignal "+item.split(':')[0]+": std_logic;\n")
    port_vector_count += 1
```

Figure 6.2: Port Declaration - Testbench Generation

shown in Figure 5.1; wherein a value of **1** indicates that the port is '*std_logic_vector*' and a value of **0** indicates '*std_logic*'. An integer variable '*port_vector_count*' is used to keep track of the input and output ports while generating the Testbench file as shown in Figure 5.2. As a result, the input and output ports can now be declared as:

- Only Non-Vectors ('*std_logic*')
- Only Vectors ('*std_logic_vector*')
- A combination of Vectors and Non Vectors ('*std_logic_vector*' and '*std_logic*')

Chapter 7

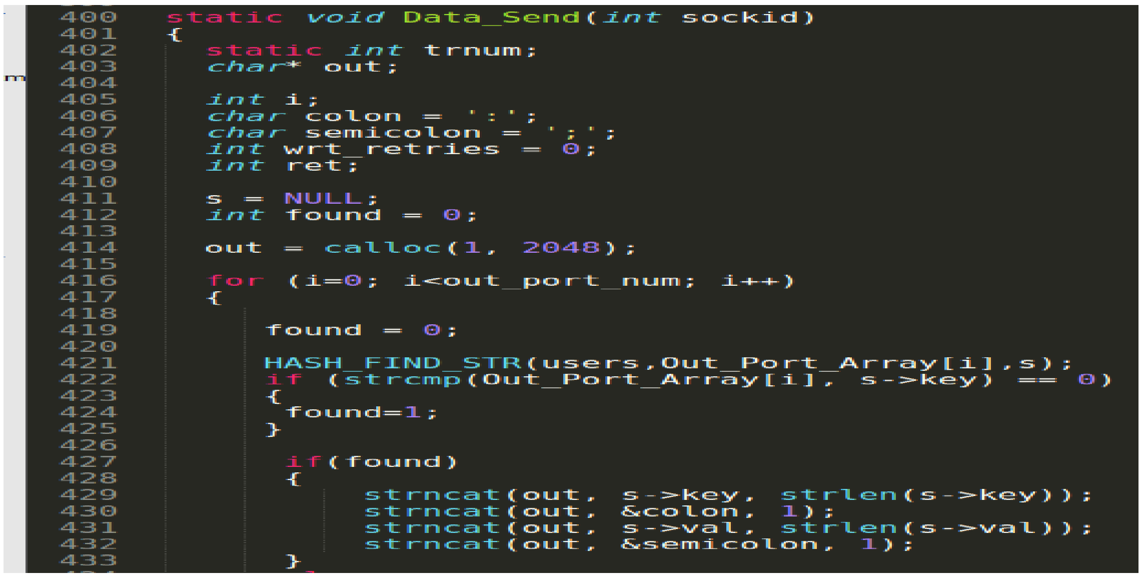
Multiple Outputs

7.1 Issue

NGHDL works with only one output and cannot implement any device with two and more outputs (Socket Programming).

7.2 Methodology

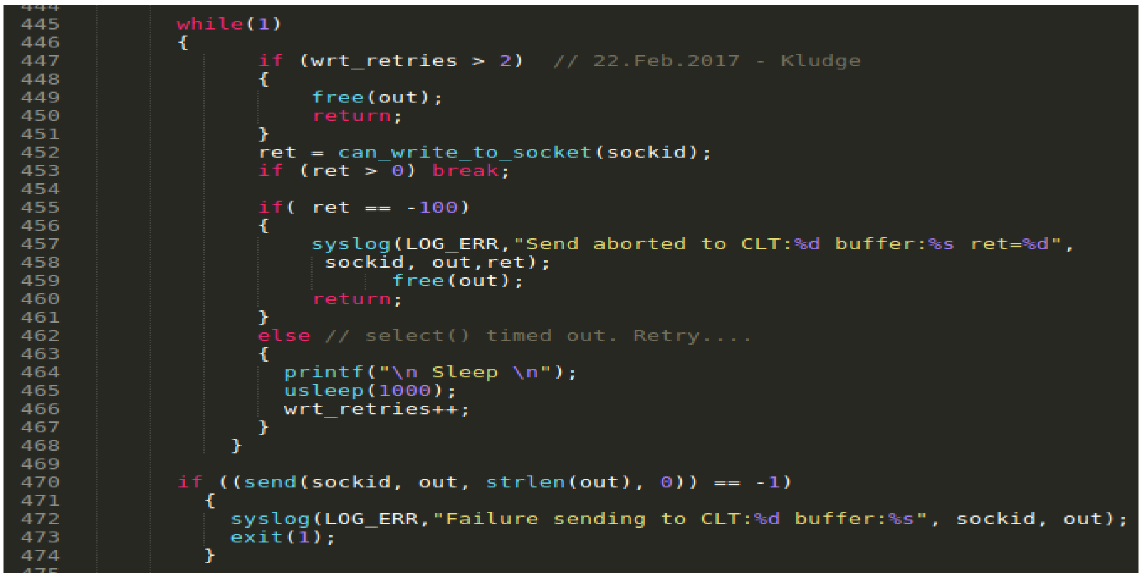
After running several simulations, it was observed that the client (Ngspice) always waited for the first output from the server (GHDL-Vhpi). However, after receiving this first output, the communication between client and server became asynchronous in nature. Thus, the client would receive any further outputs after Ngspice execution or the server would get terminated even before sending any further outputs.



```
static void Data_Send(int sockid)
{
    static int trnum;
    char* out;

    int i;
    char colon = ':';
    char semicolon = ';';
    int wrt_retries = 0;
    int ret;

    s = NULL;
    int found = 0;

    out = calloc(1, 2048);

    for (i=0; i<out_port_num; i++)
    {

        found = 0;

        HASH_FIND_STR(users,Out_Port_Array[i],s);
        if (strcmp(Out_Port_Array[i], s->key) == 0)
        {
            found=1;
        }

         if(found)
         {
                strncat(out, s->key, strlen(s->key));
                strncat(out, &colon, 1);
                strncat(out, s->val, strlen(s->val));
                strncat(out, &semicolon, 1);

         }
```

Figure 7.1: GHDL Server - Data Preparation (Data_Send function)

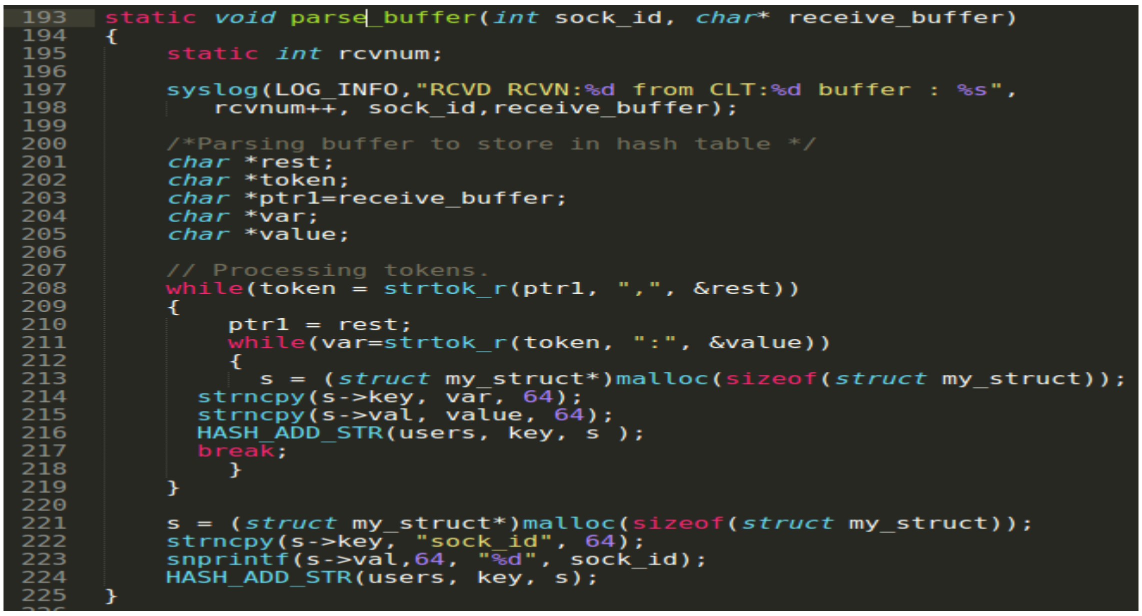
Hence, instead of sending data one by one from server to client, all the computed output ports are packed in a buffer '*out*' as shown in Figure 6.1. All of these port values are then send to the client as shown in Figure in 6.2. This output, being the very first one, is received by the client synchronously and all the ports are received in this single output.



```c
	while(1)
	{
		if (wrt_retries > 2)   // 22.Feb.2017 - Kludge
		{
			free(out);
			return;
		}
		ret = can_write_to_socket(sockid);
		if (ret > 0) break;

		if( ret == -100)
		{
			syslog(LOG_ERR,"Send aborted to CLT:%d buffer:%s ret=%d",
			 sockid, out,ret);
			      free(out);
			return;
		}
		else // select() timed out. Retry....
		{
		  printf("\n Sleep \n");
		  usleep(1000);
		  wrt_retries++;
		}
	    }

	if ((send(sockid, out, strlen(out), 0)) == -1)
	  {
	    syslog(LOG_ERR,"Failure sending to CLT:%d buffer:%s", sockid, out);
	    exit(1);
	  }
```

Figure 7.2: GHDL Server - Sending Data (Data_Send function)

Due to this logic, the buffer size needs to be increased and thus '*out*' buffer is set to 2048 bytes and maximum limit of 64 output ports have been allowed for NGHDL as shown in Figure 6.3.



```c
static void parse_buffer(int sock_id, char* receive_buffer)
{
    static int rcvnum;

    syslog(LOG_INFO,"RCVD RCVN:%d from CLT:%d buffer : %s",
       rcvnum++, sock_id,receive_buffer);

    /*Parsing buffer to store in hash table */
    char *rest;
    char *token;
    char *ptr1=receive_buffer;
    char *var;
    char *value;

    // Processing tokens.
    while(token = strtok_r(ptr1, ",", &rest))
    {
        ptr1 = rest;
        while(var=strtok_r(token, ":", &value))
        {
          s = (struct my_struct*)malloc(sizeof(struct my_struct));
      strncpy(s->key, var, 64);
      strncpy(s->val, value, 64);
      HASH_ADD_STR(users, key, s );
      break;
        }
    }

    s = (struct my_struct*)malloc(sizeof(struct my_struct));
    strncpy(s->key, "sock_id", 64);
    snprintf(s->val,64, "%d", sock_id);
    HASH_ADD_STR(users, key, s);
}
```

Figure 7.3: GHDL Server - Parse Buffer

Chapter 8

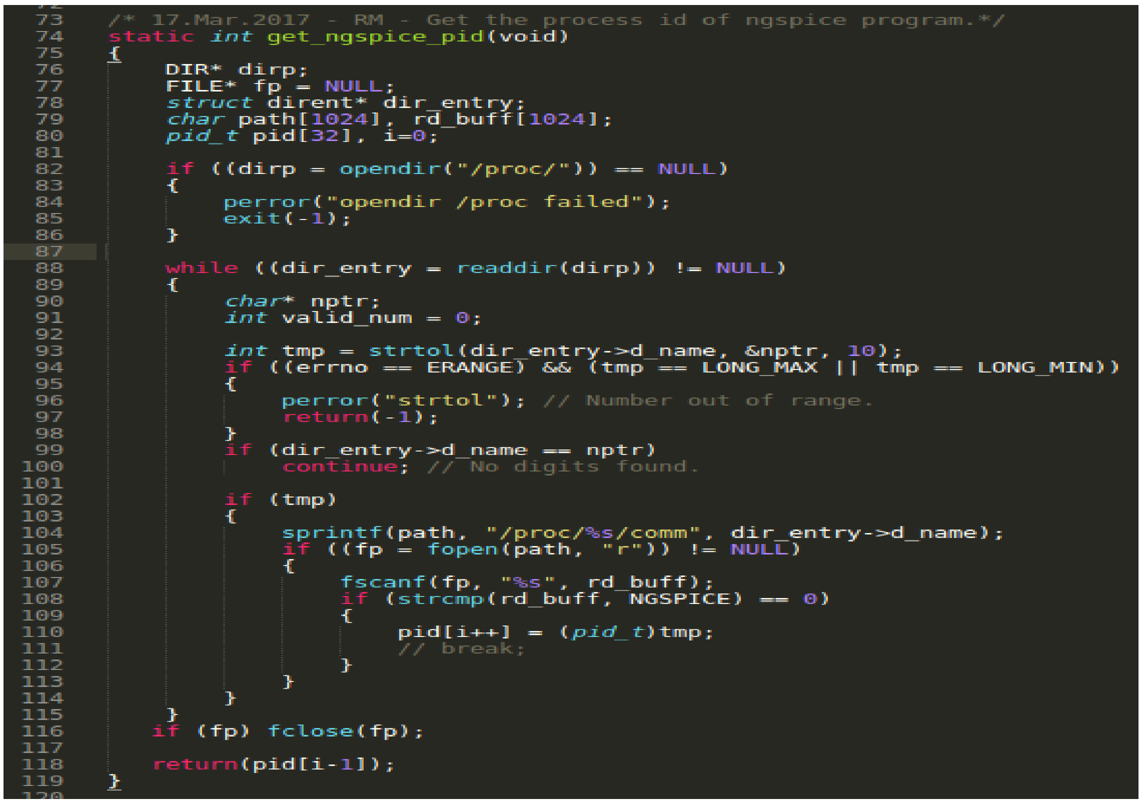
Termination of VHDL Testbench

8.1 Issue

If an instance of Ngspice is already running/simulating, then the VHDL Testbench in any further created instances of Ngspice will never terminate (Infinite Loop Issue).

8.2 Methodology

Every time a Ngspice instance is created, an associated Testbench and GHDL Server is started. To terminate them, Ngspice sends an ***END** Signal* to the Server.



```c
/* 17.Mar.2017 - RM - Get the process id of ngspice program.*/
static int get_ngspice_pid(void)
{
    DIR* dirp;
    FILE* fp = NULL;
    struct dirent* dir_entry;
    char path[1024], rd_buff[1024];
    pid_t pid[32], i=0;

    if ((dirp = opendir("/proc/")) == NULL)
    {
        perror("opendir /proc failed");
        exit(-1);
    }

    while ((dir_entry = readdir(dirp)) != NULL)
    {
        char* nptr;
        int valid_num = 0;

        int tmp = strtol(dir_entry->d_name, &nptr, 10);
        if ((errno == ERANGE) && (tmp == LONG_MAX || tmp == LONG_MIN))
        {
            perror("strtol"); // Number out of range.
            return(-1);
        }
        if (dir_entry->d_name == nptr)
            continue; // No digits found.

        if (tmp)
        {
            sprintf(path, "/proc/%s/comm", dir_entry->d_name);
            if ((fp = fopen(path, "r")) != NULL)
            {
                fscanf(fp, "%s", rd_buff);
                if (strcmp(rd_buff, NGSPICE) == 0)
                {
                    pid[i++] = (pid_t)tmp;
                    // break;
                }
            }
        }
    }
   if (fp) fclose(fp);

   return(pid[i-1]);
}
```

Figure 8.1: GHDL Server - Get Ngspice PID

Inorder to do that, Ngspice needs the **Process ID (PID)** of that GHDL Server wherein this PID is known from a file made by the GHDL Server itself. The server first iterates through the */proc/* directory of the root location in Linux based OS. It then scans all the process directories having the name *Ngspice* and breaks the iteration immediately on match.

However, since the server breaks immediately on first match of process *Ngspice*, it creates that file with the PID of the first instance of Ngspice whose content includes that server's own PID. Thus, when the current instance of Ngspice tries to open that corresponding file with its own PID, it doesn't find it as the name of that file is based on the first instance of Ngspice and therefore, goes into Infinite Loop.

Figure 7.1 shows a workaround to solve this issue. An array of PIDs is created which will store all the PIDs of Ngspice seen so far. Now, instead of breaking the logic on first match of *Ngspice*, let the search continue to find all other instances of the *Ngspice*. If it matches, then add its PID to that array. When the search is finished, then return the last PID inserted in that array. The significance of last PID inserted is that ***Process IDs*** are alloted to the processes in ***increasing order*** on Linux based Operating Systems. So the process with higher PIDs will come later in the search results and the highest PID of Ngspice instance will be inserted at last indicating that it is the most recent one.

Chapter 9

VHDL - Structural Style

9.1 Issue

The use of structural style in VHDL without any entity declaration and architecture of the components used by the main entity would give elaboration error in GHDL indicating that the component instances are not bounded.

9.2 Methodology

The structure of any VHDL code using structural style should be similar to that of the following example on Full Adder:

```
library ieee;
use ieee.std_logic_1164.all;

entity full_adder_structural is
port(a: in std_logic;
        b: in std_logic;
     cin: in std_logic;
     sum: out std_logic;
     carry: out std_logic);
end full_adder_structural;

library ieee;
use ieee.std_logic_1164.all;

entity andgate is
port(a: in std_logic;
     b: in std_logic;
     z: out std_logic);
end andgate;

architecture e1 of andgate is
begin
```

```
z <= a and b;
end e1;

library ieee;
use ieee.std_logic_1164.all;

entity xorgate is
port(a: in std_logic;
     b: in std_logic;
     z: out std_logic);
end xorgate;

architecture e2 of xorgate is
begin
z <= a xor b;
end e2;

library ieee;
use ieee.std_logic_1164.all;

entity orgate is
port(a: in std_logic;
     b: in std_logic;
     z: out std_logic);
end orgate;

architecture e3 of orgate is
begin
z <= a or b;
end e3;


architecture structural of full_adder_structural is

component andgate
port(a: in std_logic;
     b: in std_logic;
     z: out std_logic);
end component;

component xorgate
port(a: in std_logic;
         b: in std_logic;
     z: out std_logic);
end component;
```

```
component orgate
port(a: in std_logic;
        b: in std_logic;
    z: out std_logic);
end component;

signal c1,c2,c3: std_logic;

begin

u1 : xorgate port map(a,b,c1);
u2 : xorgate port map(c1,cin,sum);
u3 : andgate port map(c1,cin,c2);
u4 : andgate port map(a,b,c3);
u5 : orgate port map(c2,c3,carry);

end structural;
```

In above example, the main entity i.e. ***full_adder_structural*** should be declared at the top of the file because ***model_generation.py*** will generate a model only for the first entity in the VHDL file. Each subsequent entity must again use the ***ieee*** library and have their respective architectures defined. The architecture of the main entity, that will be using other entities as its components, should be defined at last so that it can find those entities which have been defined earlier. Note that the component names and the corresponding entity names should be same.

Reference

- Commits by Neel :
  https://github.com/nilshah98/nghdl/commits/2019Fellows
- Commits by Rahul :
  https://github.com/rahulp13/nghdl/commits/2019Fellows
- Creating Symlink in Linux :
  https://www.shellhacks.com/symlink-create-symbolic-link-linux/
- GHDL Source :
  https://github.com/ghdl/ghdl