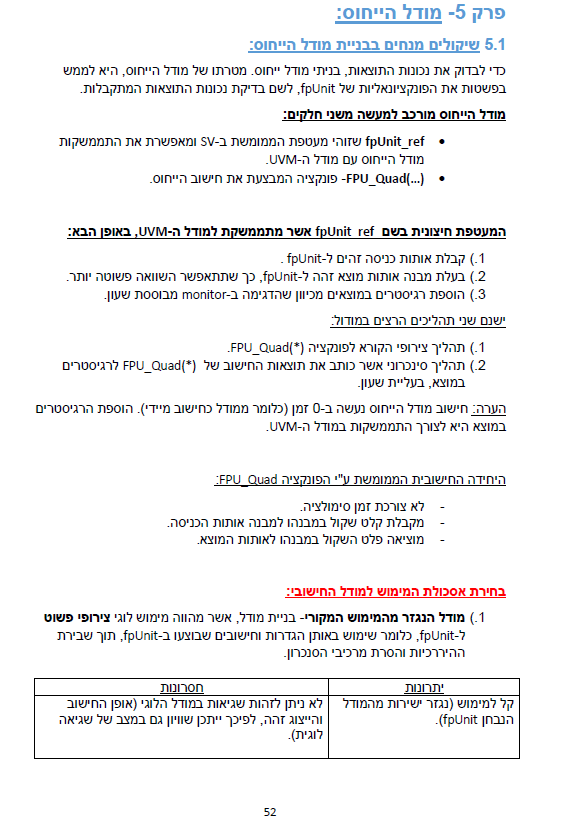
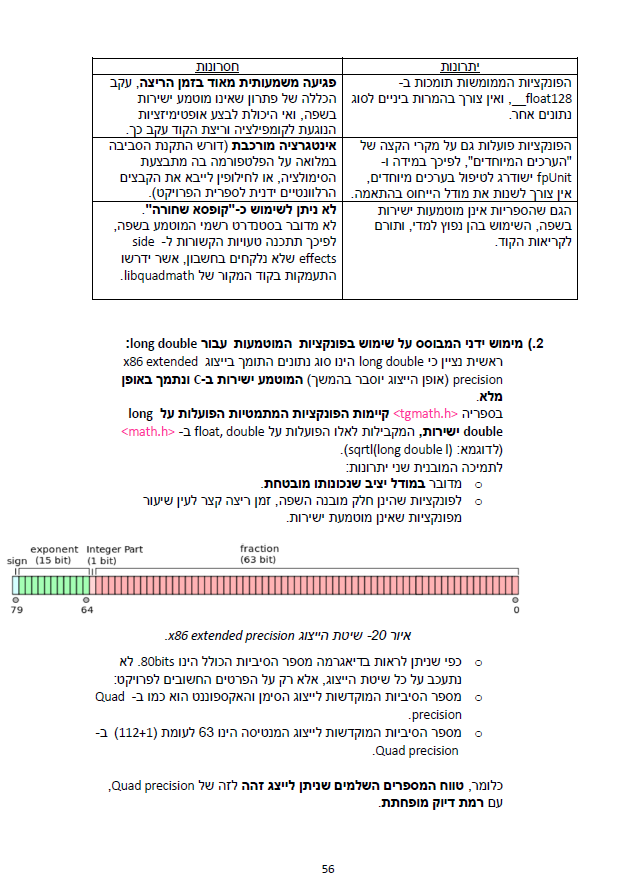
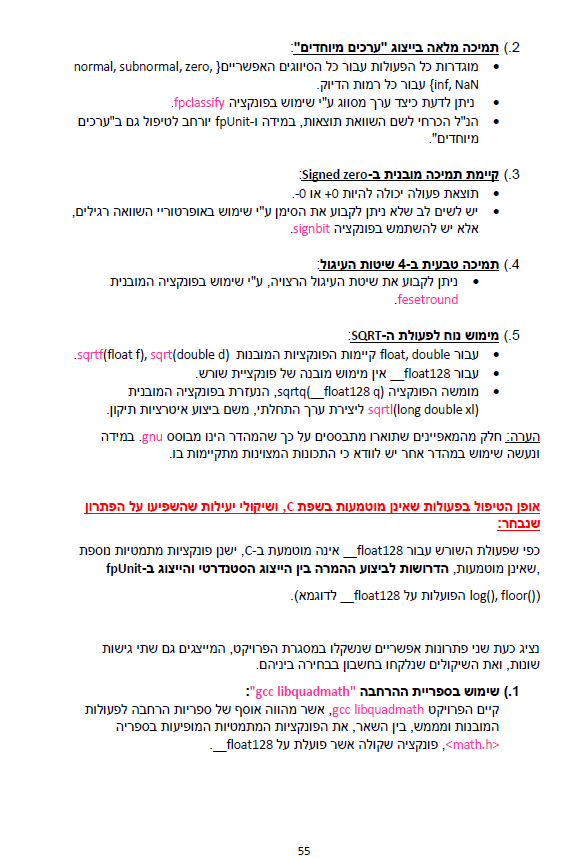
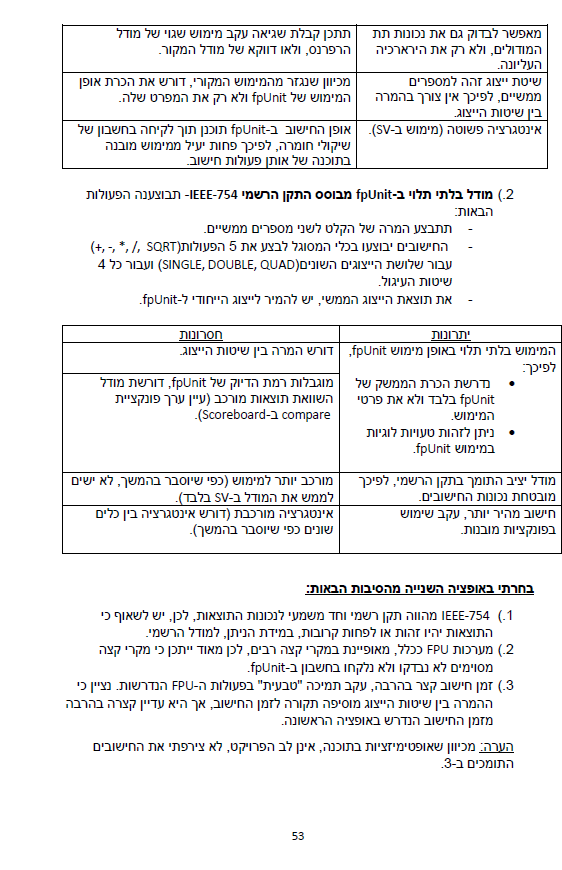
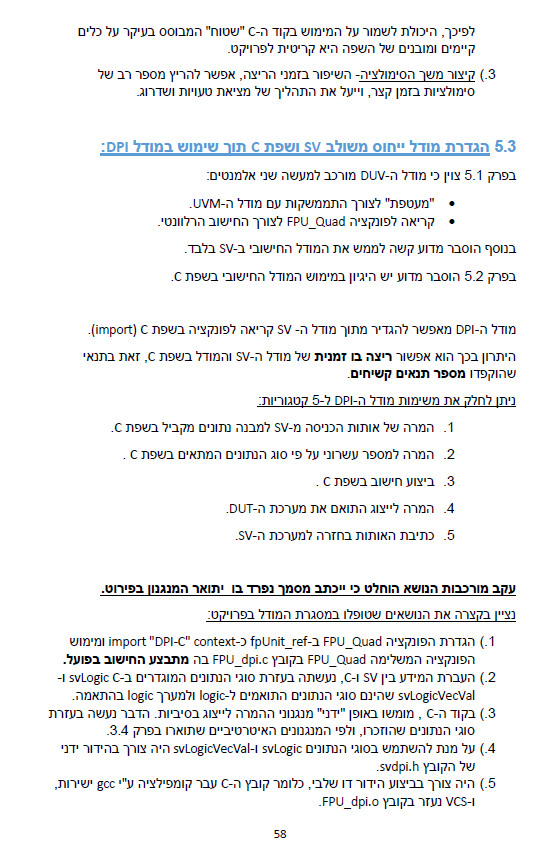
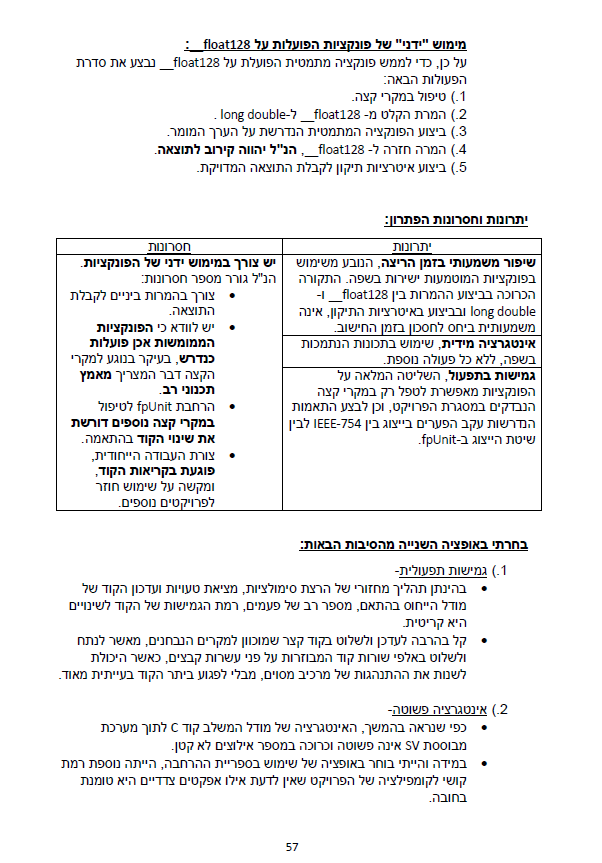
Integrating C Code :







Integrating Matlab code

Reference :

<https://www.mathworks.com/products/hdl-verifier.html>

# 7. Summary and conclusions

## 7.1 Reference model implementation

* We had many options for implementing the reference model [[4]](file:///Z:\reports\win2020\GuyOhadUVMFIR_V2.docx#_References). These are the prominent ones we considered and chose not to use:
  + Writing the entire reference model in SystemVerilog – This is a decent solution from all aspects except that it takes a lot of time to implement this (and this code would have required testing of it’s own) , and we already had MATLAB code from the previous project that modeled the design.
  + Using “shell tracing” – using our testbench to invoke a MATLAB script that will use information from our testbench to stimulate a MATLAB reference model. This is a simple and reliable solution, but MATLAB scripts run (way) slower than SystemVerilog code- so this would have slowed the tests significantly.
  + DPI-C MATLAB(chosen solution) – SystemVerilog **D**irect **P**rogramming **I**nterface allows us to access an external, precompiled, C code from anywhere with function like calls. MATLAB has an extension called “HDL verifier” [[5]](file:///Z:\reports\win2020\GuyOhadUVMFIR_V2.docx#_References), which we used to create DPI-C for SystemVerilog. We than scripted the Makefile to compile these and were able to use them like functions from our code! This is a good solution because it allowed us to use the existing reference model with minor changes (mainly initializing the MATLAB variables). Furthermore – because the code for these functions is in C, it didn’t slow the simulation time at all, and as we will see later, this is much faster than using “shell tracing”.

## 7.2 Further use of DPI-C

* After seeing how well the DPI-C worked, we used it to implement other things in our code other than the reference model. This proved very efficient and easy because, for example, MATLAB has built in functions for generating gaussian pulses.
* Other than the reference model itself we used it in these modules:
  + fir\_seq\_item – for generating the pulse itself.
  + fir\_block\_scoreboard – for analyzing the results.

## 7.3 “DPI-C” vs “shell tracing” performance

* If we take the gaussian pulse generation as our case study:
  + Running the MATLAB code for this on our PC’s took approx. 5-10 minutes (for one gaussian pulse).
  + Running an entire simulation, generating a big number of **random** gaussian pulses( )takes less than a minute on VLSI lab hardware.
  + Even though this was tested on different hardware it’s probably safe to assume that DPI-C is way faster than simply sending traces to MATLAB.

## 7.4 Analyzing test results

* What should be considered a “good” output for this specific design?
* There is more than one possible answer for this because our reference because the DUT distorts the signal somewhat.
* At first, we used this criterion:



Figure 26 - preliminary comparison criteria: sum of diff for each sample normalized by the mean of ref model results

But this was problematic because there is some lag between the results –

As we can see here:

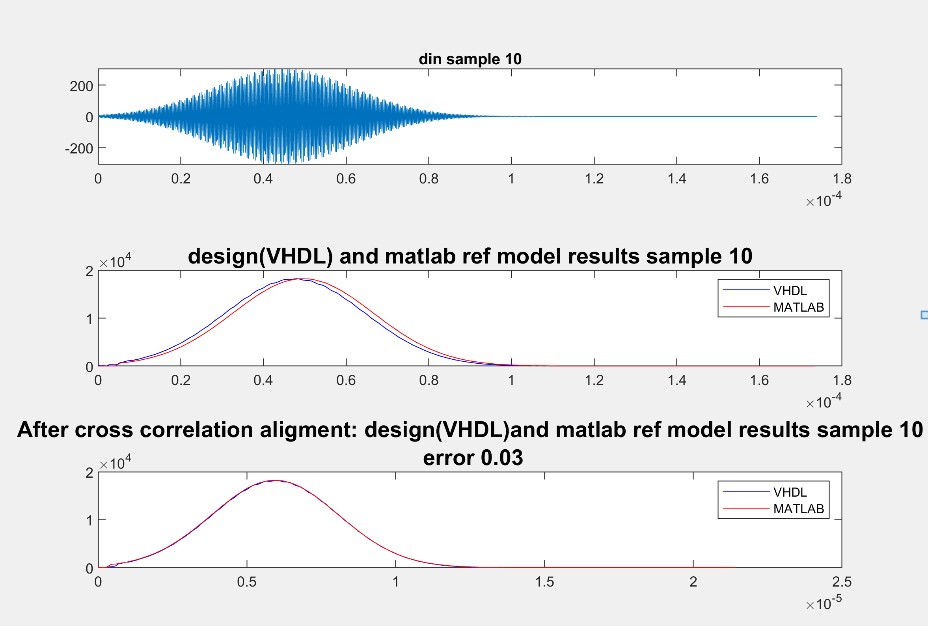


Figure 27 - MATLAB and VHDL result lag in results

* This lag was not the same for each run, so using a fixed offset was not an option.
* Instead we used this algorithm (implemented in MATLAB in calc\_error\_xcor\_align.m):
  + Find the **Cross-Correlation** between reference model output and DUT output.
  + Shift with the max cross correlation lag difference.
  + Afterwards, check the max delta between the DUT and the reference model outputs normalized by max(delta).

## 7.5 Overall conclusions:

* MATLAB DPI-C(via HDL verifier package) is a great tool for creating fast and bug-free test benches as it is easy to create reference models and transactions using MATLAB!
* This works very well in UVM because it allows us to easily synchronize the DUT, reference model and scoreboard and generate random stimuli.
* UVM dictates a more modular approach to building your testbench - this may seem like a big overhead for small DUT’s and writing in UVM has a steep learning curve - however it is well worth it!

# Appendix A: Makefile

In this project we used a makefile to compile and run the simulation.

Usage:

make <run modes> <arg1>=<value> <arg2>=<value> …

* Run modes:
  + comp : compiles the project in the results directory
  + run : runs a simulation in the results directory
  + all : comp + run
  + run\_ww : runs a simulation and dumps waves in results dir (inter.vpd)
  + run\_gui : runs a simulation in gui mode
  + clean : deletes all files from results directory
  + **default :** all

Examples:

1. make

Compiles and run a simulation

2. make comp run\_gui

Compiles and run a simulation with gui

* Arguments (Optional):
  + RESULTS – changes the results directory. Default : results
  + UVM\_VERBOSITY – changes the verbosity of the simulation. Default : UVM\_MEDIUM
  + SEED – fix the seed to a specific number, Default : random
  + TEST – pick a different test to run. Default : fir\_simple\_test
  + NUM\_OF\_RUNS – pick the number of sequence items to run in a test. Default : 1
  + PRINT\_OUT – set 1 to create a directory of outputs(both matlab and design) in results directory. Default : 0 (off)

# Appendix B: Analyzing Waves

1. Analyzing simulation waveform:

To analyze waves in simulation we used DVE (Discovery Visual Environment) tool. For more elaborated details on how use the tool please address the user guide [[6]](file:///Z:\reports\win2020\GuyOhadUVMFIR_V2.docx#_References).

**Usage:**

To open a wave file after running a simulation with run\_ww flag, invoke the following command:

dve -vpd <inter file>

example:

dve -vpd results/inter.vpd

# Appendix C: Analyzing Coverage

To open a coverage report, use one of the two following methods:

**DVE**

To open the coverage report, invoke the following command

dve -cov -dir <simv.vdb dir>

example:

dve -cov -dir results/simv.vdb

**Urg (unified report Generato) Report**

To generate a html report, do the following steps:

1. urg -dir <simv.vdb dir> -report <target dir> -elfile <excludes file>

2. chrome <target dir>/dashboard.html &

Notes:

* exclude file is located in **bin/exclude\_file.el**
* For convenience, we create a script to do both in one command:

open\_cov <target\_dir>

# Appendix D: Creating DPI-C from MATLAB

To create an API of the matlab function to be used in the SV/UVM environment do the following steps: (Requires HDL Verifier package in MATLAB)

1. run build\_dpi.m

2. copy the codegen directory to matlab directory in unix

3. create the following makefile each in function directory that were created, while replacing <function\_name> to the matlab function name:

SRC=$(wildcard \*.c)

OBJ=$(SRC:.c=.o)

SHARE\_LIB\_NAME=<function\_name>\_dpi\_lib.so

all: $(SRC) $(SHARE\_LIB\_NAME)

@echo "### Successfully generated all binary outputs."

$(SHARE\_LIB\_NAME): $(OBJ)

gcc -shared -lm $(OBJ) -o $@

.c.o:

gcc -c -fPIC -Wall -ansi -pedantic -Wno-long-long -fwrapv -O0 $< -o $@

4. invoke: make -f <makefile\_name>

5. add the create .so file to **matlab/sv\_liblist**

**Example:**

#!SV\_LIBRARIES

codegen/dll/FIR\_RealTime/FIR\_RealTime\_dpi\_lib

codegen/dll/TopLevel/TopLevel\_dpi\_lib

codegen/dll/gaussian\_pulse\_gen/gaussian\_pulse\_gen\_dpi\_lib

codegen/dll/calc\_error\_xcor\_align/calc\_error\_xcor\_align\_dpi\_lib

6. import the created uvm package in **verif/tb/env/fir\_env\_pkg.sv**

**Example:**

import TopLevel\_dpi\_pkg::\*;