PYTHON DRIVEN VERIFICATION

INTRODUCTION TUTORIAL

ILIA GREENBLAT@MAC.COM +972-54-4927322

WHAT YOU NEED:

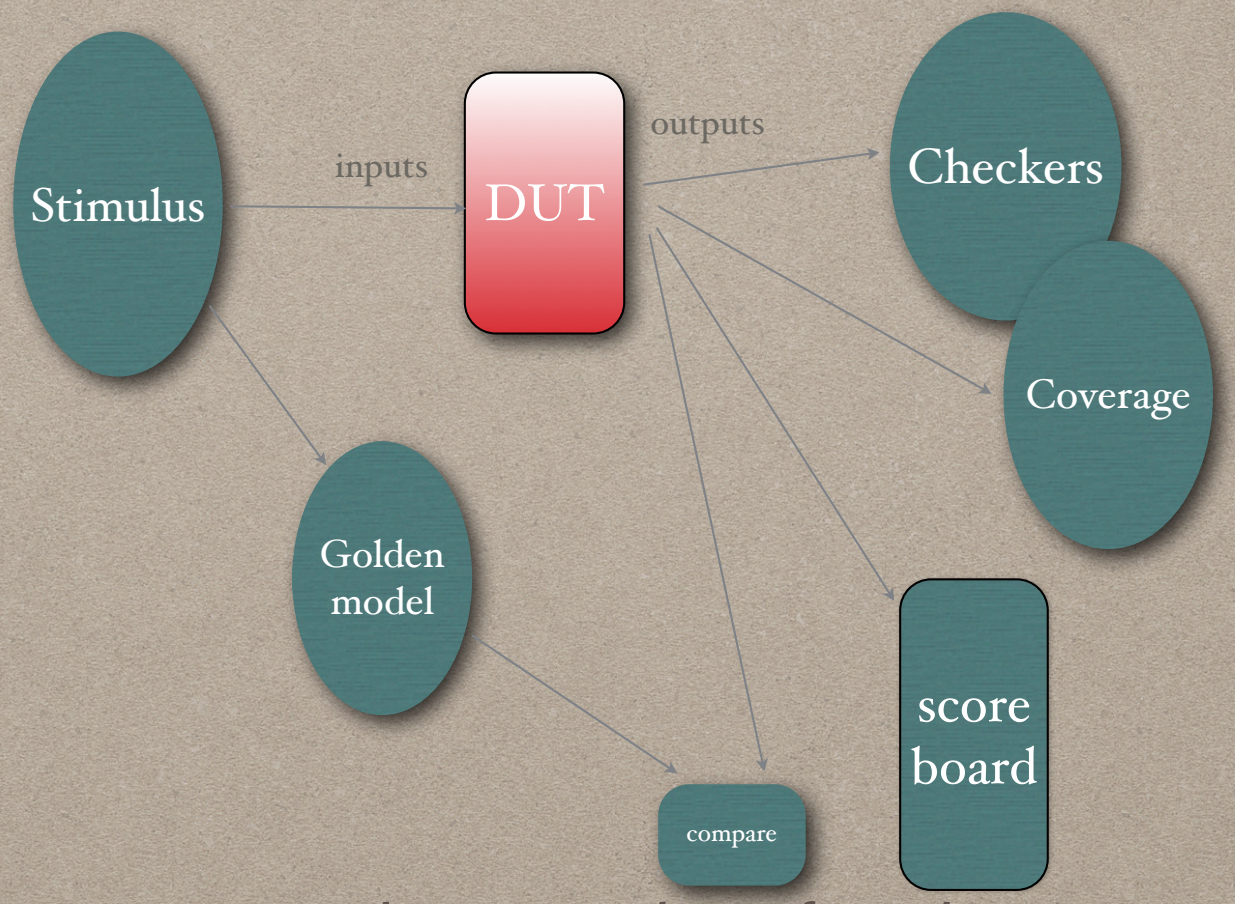
Icarus Verilog

1. verilog simulator. icarus is fine free simulator. but any simulator will do.
2. VCD waves viewer. GTKWAVE is great and free. but any commercial is fine.
3. clone files from GitHub:

   git clone https://github.com/greenblat /vlsistuff.git vlsistuff
4. from cloned vlsi stuff (vlsistuff/python-verilog/vpi) You need to compile the : for vcs and ncverilog: vpi7_nc_27.so or for icarus:vpi7.vpic
5. examples are in vlsistuff/python-verilog/examples
6. cat or dog for play while taking breaks.
7. coffee or tea and probably lots of it.
8. sunglasses: adhering to this methodology - you will see the light of day and your kids growing.

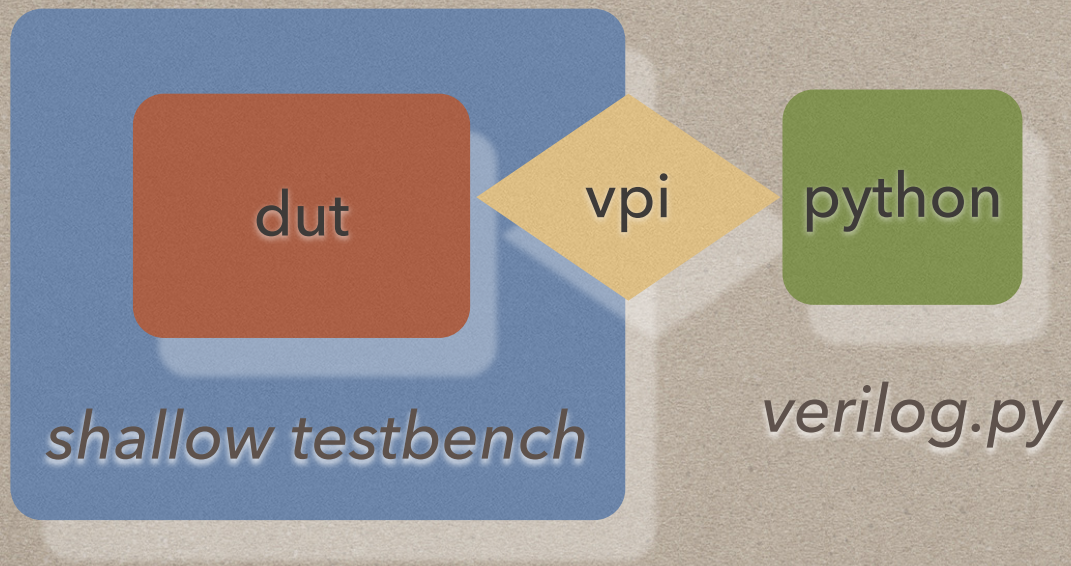
BASICS

establishing common language



It is our aim to provide examples of each component in upcoming test-cases.

verification setup the python way



Code that is not part of the design is written in python.

All simulators (Vcs,NCverilog, ModelSim,Icarus) have VPI connection to external C code. We compile once a C program that embeds python interpreter. The result is shared library.
Test-bench has all DUT inputs as regs, outputs as wires and toggles the clocks - thus "shallow".
VPI connection is "C" code written once and changes very-very infrequently.
Presented to simulators as "shared library".

add in ncverilog: ***+loadvpi="/nis/bin/vpi:vpit_RegisterTfs"***
add in vcs: ***+vpi_load="/nis/bin/vpi.so:vpit_RegisterTfs"***

PYTHON - WHY BOTHER?

* python is structured, object-O, organised scripting language.
* widely used for many different tasks.
* www.python.org is great starting point for learning it.
* it is open source, platform agnostic and everything is downloadable.
* has wealth of libraries written by many people.
* these libraries cover almost anything imaginable. for example MPEG encoders/decoders, GZIP algorithms, encryption algorithms and more.

PYTHON, WHY?

* the language was defined first, implemented later.
* very structured. Has easy powerful classes (objects).
* Easy to write and being able to understand later - both your code and team.
* Is powerful, high level and fast.
* **all test bench components, like scoreboard and golden model are easily implemented. This comes from our experience of 10 years.**
* VPI interface is written in C once and effectively hidden from the user

DIVING IN... BASICS

On verilog side (tb.v)

```
reg clk;
initial $python("initial()",12);
always begin
   clk=0;
   #10;
   $python("posedge()");
   clk=1;
   #10;
   $python("negedge()");
end
```

On python side (verilog.py file)

```
import veri  #all verilog conns are  in veri lib (from VPI)
import logs     # lib of useful stuff.

cycles=0    # just count cycles
def initial(What):    # put here startup code
   logs.log_info( 'put here initial code, open files ...')

def posedge():
   global cycles
   cycles=cycles+1

def negedge():
   if (cycles>1000):
         veri.finish()   #exit from simulator
     Req = veri.peek('top.dut.request')
    if (Req=='1'):
       logs.log_info('requestid %s'%(veri.peek('reqid'))
       veri.force('top.dut.ack','1')
    else:
       veri.force('top.dut.ack','0')
```

$PYTHON() SYSTEM TASK

*In verilog code, **$python()** system task is the only addition.*

$python("pyfunc()"); *– transfer control to python. There is no return value. The connection will search for "**def pyfunc()**:" in python code.*

$python("pyfunc()",bus[3:0],param); *– adding parameters to system task. The connection will search for "**def pyfunc(prm0,prm1)**:" in python code. All params are presented as **binary strings** to python.*

PYTHON BASICS:

import veri *– connection from python back to verilog simulator.*

prdata = ***veri.peek****('tb.dut.prdata')*
veri.peek returns binary values string '01zx'. Length of the string is width of the bus.

veri.force*('tb.dut.pwdata','0x34f5')*
veri.force allows to set value to register and inputs. The parameter is always a string. either decimal : "345", or hex: '0xffaa' or binary: '0b1101101'

These two are almost all there is.

PYTHON BASICS : MORE

*Time = **veri.stime()** – pretty self explainatory*
***veri.finish()** – pretty self explainatory*
***veri.peek_mem(Path,Index)** – value of item in array.*
***veri.force_mem(Path,Index,Value)** – set value of item in array.*

***veri.listing(Path,Depth,FileName)** – dump instant values of signals from Path deep (Depth levels) into a file named FileName. Tip: Useful to grep "= z" to catch all un-driven nets.*

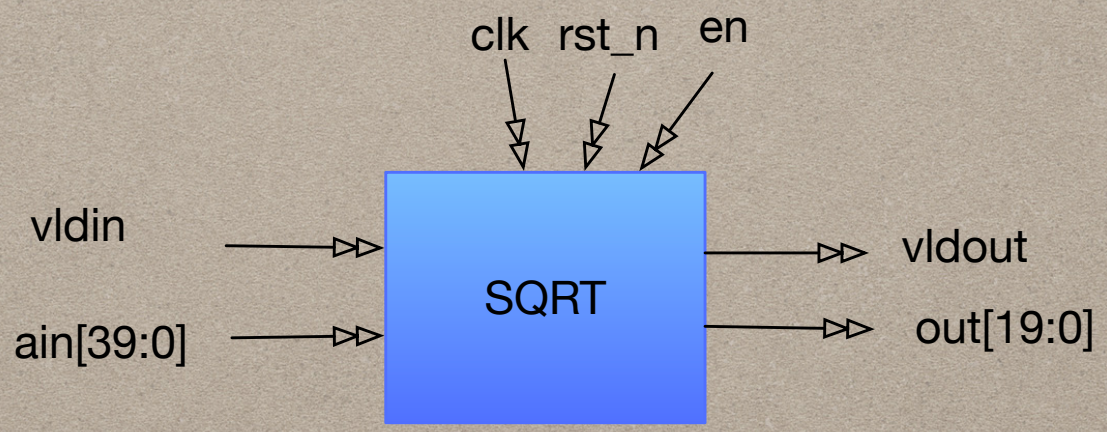
MY PYTHON GUIDELINES

1. no need to use inheritance. it will bite You later in the a…
2. variable names - never just one letter. at least two: "i" -> "ii"
"rewrite" vs "reuse". -
*3. please dont use: from <LIB> import * , only plain import.*

my recommendations

STEP BY STEP, A REAL ONE..

RTL module that uses CORDIC to compute square root of 40 bits unsigned number.



few clocks after "vldin" is active, "vldout" will be active and "out" has the result.
"vldin" may be active on any clock. The design is pipelined. "EN" input makes the pipeline move.

* *all files can be found in cloned from gitHub :*
  vlsituff/verilog-python/examples/sqrt

TEST BENCH *usually generated by "pyver.py" system*

```
module tb;
reg [31:0] cycles;   initial cycles=0;
reg [31:0] errors;   initial errors=0;
reg [31:0] wrongs;   initial wrongs=0;
reg [31:0] corrects; initial corrects=0;
reg [31:0] marker;   initial marker=0;
reg [39:0] ain;
reg  clk,en,rst_n,vldin;
wire [19:0] out;
wire  vldout;

always begin
    clk=0;
    #10;
    clk=1;
    #3; $python("negedge()"); #7;
end
initial begin
    $dumpvars(0,tb);
    ain[39:0] = 0;
    en = 0;
    rst_n = 0;
    vldin = 0;
    #100;
    rst_n=1;
end
sqrt_u40_5 dut (
     .clk(clk) ,.rst_n(rst_n) ,.en(en)
    ,.ain(ain[39:0]) ,.vldin(vldin)
    ,.out(out[19:0]) ,.vldout(vldout)
);
endmodule
```

all DUT inputs are regs and outputs are wires. Python code toggles the inputs and monitors the outputs. "cycles" counts clocks and is visible on waves. All info messages refer to that number.
"marker" is for python use to mark special places.

"corrects"/"wrongs"/"errors" are counts set by "logs.py" module when errors or warnings are logged. The error number appears both in waves and in log file.

VERILOG.PY *PART 1:*

VPI connection, when activated by simulator looks for "verilog.py" file and looks there for any $python reference.

```
import os,sys,string,random,math
import veri
sys.path.append('/Users/ilia/verification_libs')
import logs

cycles=0
state='idle'
def negedge():
    global cycles,idle,Ain,state,Max
    cycles += 1
    veri.force('tb.cycles',str(cycles))
    if (cycles>1000):
        veri.listing('tb','1000','cucu.list')
        veri.finish()
        return
    veri.force('tb.en','1')

    rst_n = veri.peek('tb.rst_n')
    if (rst_n!='1'):
        ain = veri.peek('tb.ain')
        Len = len(ain)
        Max = 1<<Len
        return
```

logs.py is library full of logging/ conversion and more functions.
def negedge(): is the "lead" function called from test-bench.

"Max" variable keeps the max value allowable to be driven to the DUT. It does that by asking the width of the input bus.
"veri.listing()" prints state of all nets at the end of simulation. It helps to see whether some untied ends exist.

VERILOG.PY *continued.*

```
if state=='idle':
     Ain = random.randint(0,Max)
     veri.force(‘tb.ain',hex(Ain))
     veri.force('tb.vldin','1')
     state='work0'
 elif state=='work0':
     veri.force('tb.vldin','0')
     state='wait0'
 elif state=='wait0':
     vldout = logs.peek('tb.vldout')
     if vldout==1:
         Out = logs.peek('tb.out')
         Exp = int(math.sqrt(Ain))
         if Out==Exp:
             logs.log_correct('in=%d out=%d'%(Ain,Out))
         else:
             logs.log_wrong('ain=%d max=%d expected=%d act=%d’%(
                 Ain,Max,int(math.sqrt(Ain)),Out))
         state='idle'
```

The code here activates the DUT in single/wait to completion mode. It remembers the random input and checks the result when “validout” becomes active.

LOGS.PY : PY MOD HILITES.

logs.log_info(Txt,Log=0)	print info to screen and log file (pymon.log0)
logs.log_correct(Txt)	comparison of expected and actual value agrees. each message has serial number that appears in waves and in print.
logs.log_wrong(Txt)	comparison of expected and actual value failed.
logs.log_error(Txt)	either error condition in design or test-bench.
logs.intx(Bin)	convert binary string from peek into integer. 'x' or 'z' in string returns -1
logs.peek(Sig)	return logs.intx(veri.peek(Sig))
logs.peeksigned(Sig)	like logs.peek, but based on MSB can return negative.

there are many more helpful functions in logs.py, but in Python as in Python, it is easy enough to write anything yourself.

ALUKA MODULES

Sometimes RTL of a module is not finished, but interface is defined and we want to see how it fits and works in it's environment. Or the real RTL is big for simulation and we want to run a model. For example uart. The "aluka" module is empty, with all outputs declared as "reg". Python function identifies which "uart" is calling it and calls a corresponding object. Also python code that plays this module, can be used later to monitor inputs/outputs of a real RTL and be used as checker.

```
module py_uartx2 (
     input [15:0] baudrate
    ,input    clk
    ,input [7:0] control
    ,input    nreset
    ,input    read_rx
    ,input    rxd
    ,input [7:0] txdata
    ,input    write_tx
    ,output reg    rx_valid
    ,output reg [7:0] rxdata
    ,output reg [7:0] status
    ,output reg    tx_empty
    ,output reg    txd
);
initial begin
    rx_valid = 0;
    rxdata = 0;
    status = 0;
    tx_empty = 0;
    txd = 0;
end
```

in tb.v we have "iid" integer that is used to give unique ids.

```
integer myid;
reg [255:0] mypath;
initial begin
    #1;
   myid = tb.iid;
   $swrite(mypath,"%m");
   tb.iid = tb.iid + 1;
   $display("mypath %s %d",mypath,myid);
end

always @(negedge clk) begin
    $python("uart_operator()",myid,mypath);
end
endmodule
```

MONITORS

```
Monitors=[]
import smarttimer
smrt = smarttimer.smarttimerClass('tb.dut.smartimer',Monitors)

import apbDriver
apb = apbDriver.apbDriver('tb',Monitors)
apb.load_renames_file(‘apb_renames.defines’)

import axiMaster
axi = axiMaster.axiMasterClass(‘tb',Monitors)
import axiSlave

slave = axiSlave.axiSlaveClass('tb',Monitors)

def negedge():
   if (cycles>30):
         for Mon in Monitors: Mon.run()
```

interface pythons

jtag
uart
axi (m,s)
apb (m)
ahb
axiLight
i2c (m,s)
spi (m,s)

Classes that implement interfaces or monitors can be "hung" on "Monitors" list. During simulation all classes appearing in Monitors are called through .run() . Their behaviour may be modified through direct commands to the objects. themselves.

as example, in axiMaster.py, you will see:

```
def __init__(self,Path,Monitors):
   Monitors.append(self)
```

SEQUENCE.PY : A NICE CLASS

class that receives a scenario text file and runs it, executing commands from it. The file name can come from plus argument: simv +SEQ=seq0
This class recognises many commands, and also can activate other classes. Each command is line with several words. Agents can be called from commands, if they are connected as last argument in instantiation - (apb,'apb'). First is object and second is the nickname that will activate it.

in verilog.py

```
import sequenceClass
seq =
sequenceClass.sequenceClass('tb',Monitors,'',
[(apb,'apb')])

def sequence(SeqFname):
    Name = logs.bin2string(SeqFname)
    logs.log_info('sequence %s'%Name)
    seq.Sequence = open(Name).readlines()
    seq.workIncludes()
```

in test-bench verilog

```
reg [1023:0] testname;
initial begin
    if ($value$plusargs("SEQ=%s",testname)) begin
         #10;
         $python("sequence()",testname);
    end
end
```

SEQUENCE COMMANDS

In works is more powerful version with flow control, like loops, conditions and more. but as usual in python, DYI works just fine.

```
exec random.seed(500.0)
define uart 0x32+0x45
include rgf.defines
marker 3
apb write map1_0 0x2356
apb write enable 1
wait 500
print killed  dut.killed
finish
```

finish	exits simulation
wait 0x34+aaa	evals the parameter and idles for that number of clocks.
define Var Val	assigns eval(Val) value to Var. can be used in evals later
exec <python stmtnt>	execs interactively python statement.
print <words>	logs words, every word that is real signal is peeked also.
apb write enable 1	"apb" was connected to object apb. rest of words passed to .action() function of that object.
include <filename>	read commands from that file.

TIP #1: TRIGGERS ON EVENTS

Question: i want to catch some condition event and run code when that happens. How?

—- Easiest way is to write class that it's run function checks for this event on every call. Then "hang" this class (or rather object) on "monitors" and every call to self.run() function to check whether conditions are ripe. The class/object can use "cycles" variable (counts clock cycles) to activate itself in the future. While looking like waste of resources - never saw significant overhead.

—— There is also option to activate "callback" - specify change of some net and related python function. I never use it though.

—- very simple option, in these cases to add

"always @(CONDITION) $python("relatedFunction()");

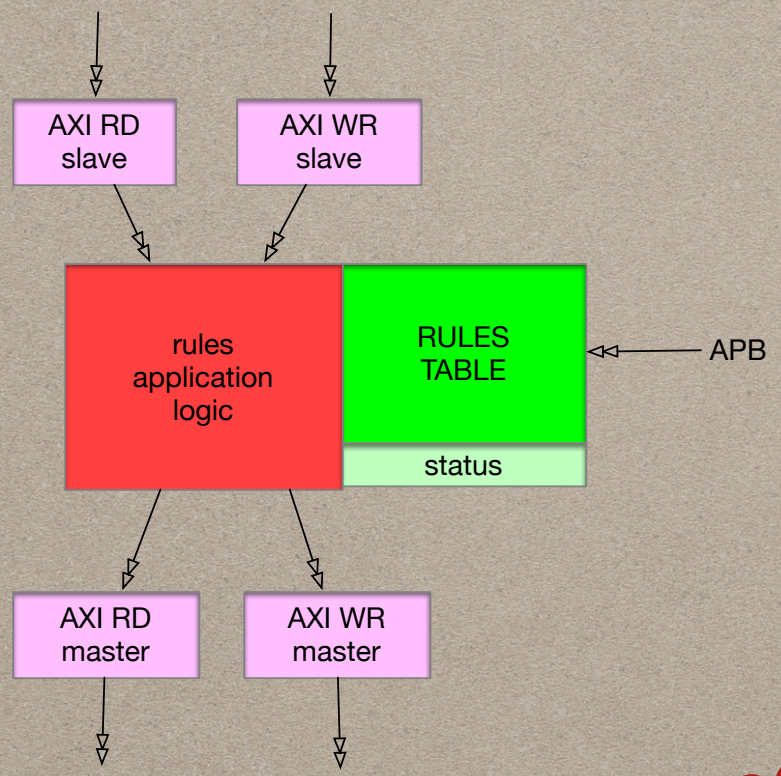
in tb.v

keep in mind that python usually has "GLOBAL" namespace, meaning functions/classes can talk among themselves to coordinate things like scoreboard, expected results and such.

TIP#2
TIME DEFINED BEHAVIOUR

Protocol like AXI,requires clock by clock sequence.
My AXI masters and slaves are implemented using queues.
It means that upon receiving a command (like read or write)
Several atomic operations are appended to the end of the queue. Each clock, top of the queue is popped and executed (sometimes conditionally, as in when arready==0).
Same applies to UART,APB,JTAG,SPI and all other serials master and slaves.

BIG EXAMPLE *AXI FIREWALL*

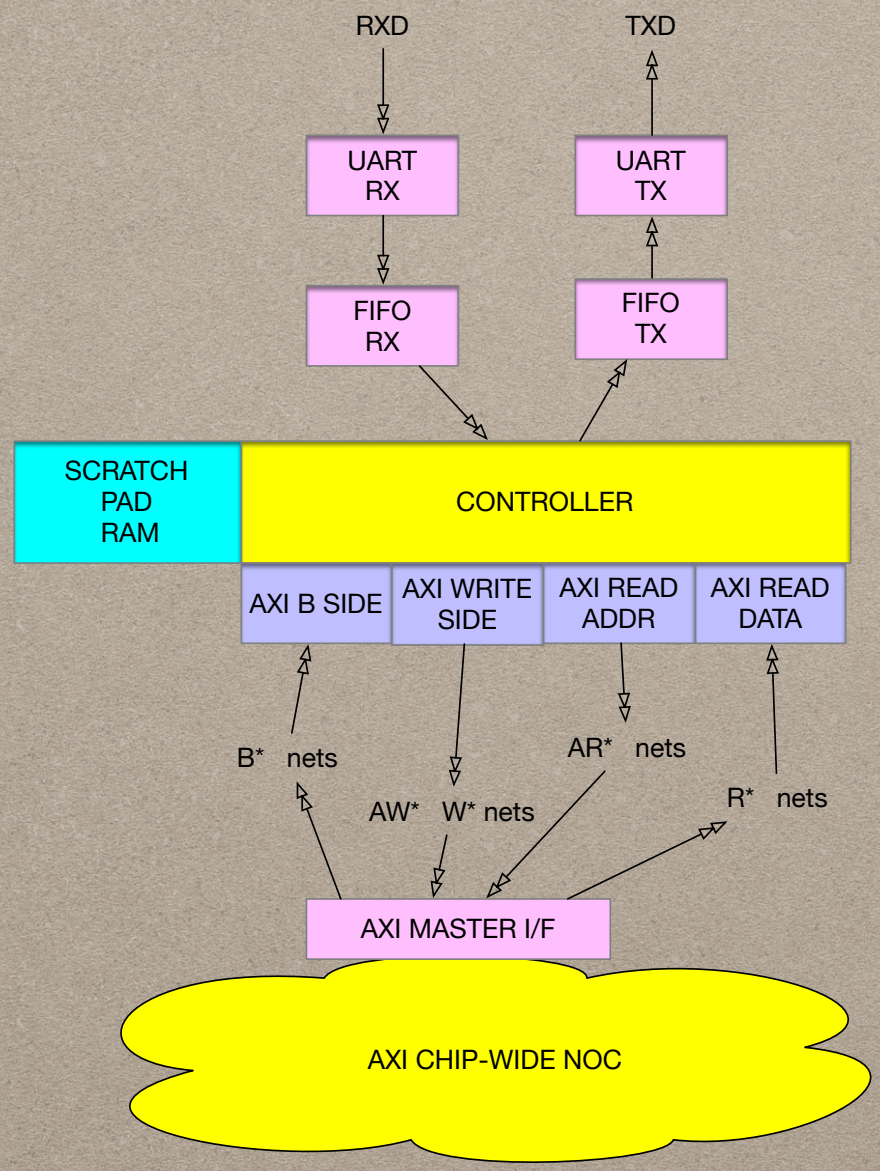


Too complex, dropped.

FILTER AXI TRANSACTIONS BY FIELDS.

Each coming transaction is checked against preprogrammed rules.First rule to fire determines its fate.
Failed write transactions are acknowledged (bvalid) but not passed on. Write data goes to /dev/null.
Failed read transactions return all data zeroes.
Normal completion is on purpose - not to alert the attacker.
APB is assumed to be trustworthy. Status keeps count of failed transactions and fields of first caught transaction.
Interrupt option to alert the monitors.
The simulation has VIP APB master, AXI master and AXI slave. All VIP are python.
First rule tables are filled through APB, then randomised transactions driven by VIP-AXI master onto DUT-AXI slaves.
In real firewall it is useful to add "source" identification field (outside AXI). The destination is assumed to be known by chip topology and address. Here we use transaction ID as replacement.

BIG EXAMPLE *debug UART to AXI.*



it is an AXI master operated by ascii commands to UART. The usage in real silicon is for bringup of the silicon through access to AXI network on a chip. Minimal external pins requirement (just two) with full internal access power.
In real silicon care must be taken to ensure cyber-hacking-safe operation of this module.

Coming soon