Everything mentioned here is OPEN SOURCE and clonable from GitHub

Python driven verification

Why it is good for You

Ilia greenblat@mac.com +972-54-4927322

Preliminaries

My flow is not CocoTb.
It is developed and maintained by me. git-pullable from:

git clone https://github.com/greenblat/vlsistuff.git

** CocoTb is nice, but overly complicated to my taste.
Easy to get lost.

It is all free, so why do i bother to preach about it?

Python has it all.

It is simple to learn, powerful dynamic language.

No hidden magic under the hood.

Easy to integrate into simulation

Works the same with free and commercial sims

Unlike SV+UVM where new inventions of complexity rule the day.

What is left to show is that Python works for verification.

Layers of PyDrVer

SIMULATOR + DUT

SYNOPSYS®
cadence®
Mentor®
A Siemens Business
Icarus Verilog
VERILATOR

vpi7.so

compiled "C" code. You never see it unless You are a geek.

veri api

veri(y) few API functions

verilog.py

top level python.
the filename is set in stone.

"UVM" layer

the usual: monitors, checkers, drivers, scoreboards

+loadvpi=/home/ilia/github/vlsistuff/python-verilog/vpi/vpi7_ml.so:vpit_RegisterTfs

+vpi -load /home/iliag/software/vpi/vpi7_nc_27.so:vpit_RegisterTfs

RTL and TestBench

Are always different languages

RTL

Verilog

Vhdl

SV

system-C

HLS?

TB

E is not verilog

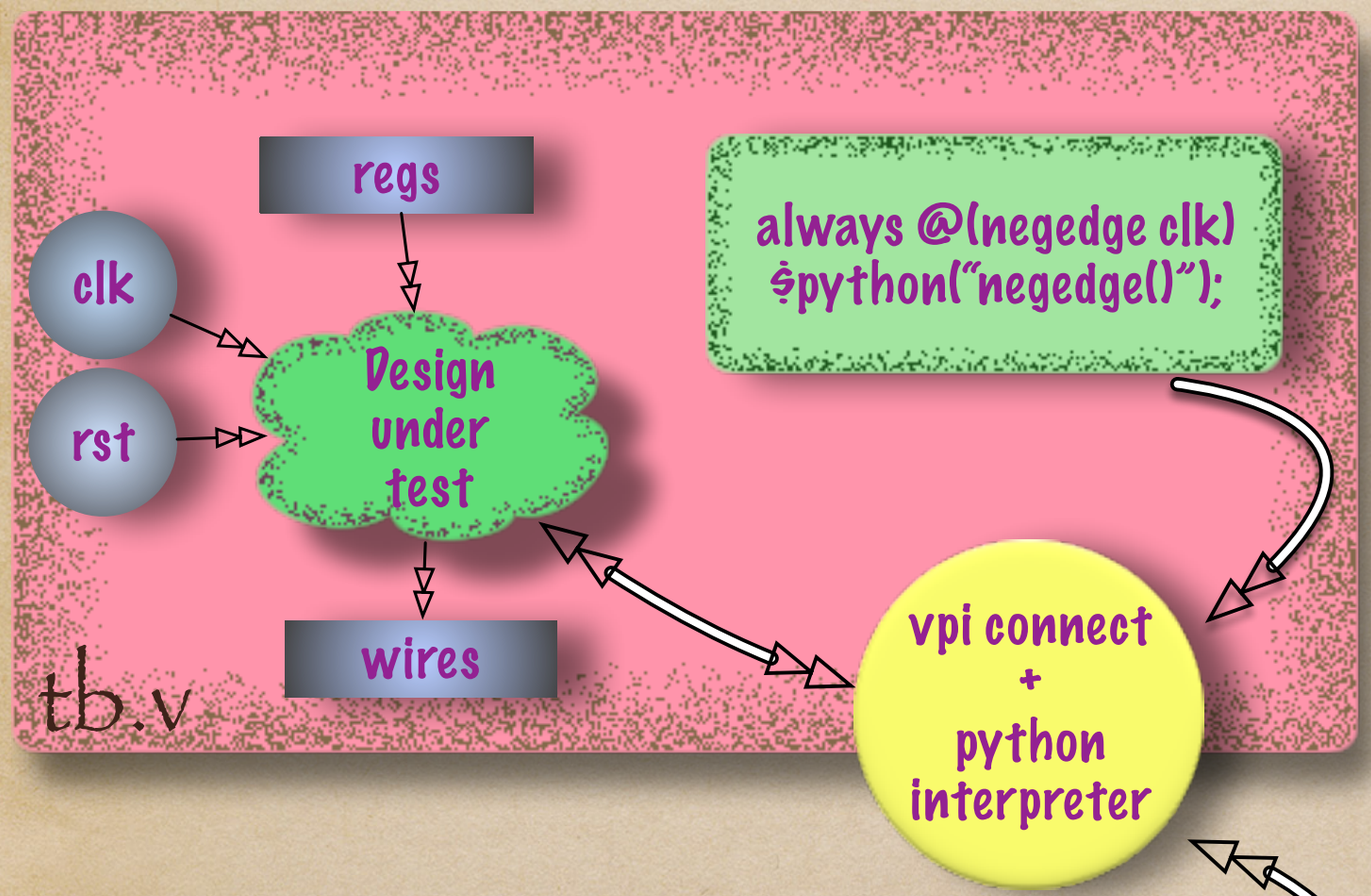
SV-TB is not synthesizable

Python

C C++

Matlab

Verification with python



Ingredients:

DUT

Shallow test harness (*)

vpi3.so

verilog.py

import .py modules

*** checkout companion software that auto generates test bench.

```
CADENCE:   vcs . . . +loadvpi=vpi/vpi7_ml.so:vpit_RegisterTfs
SYNOPSYS:  xrun . . . +vpi -load vpi7_nc_27.so:vpit_RegisterTfs
```

py VIPs

diving in... basics

On verilog side (tb.v)

```
reg fast_clk, slow_clk;
initial $python("initial()",12);
```

```
always @(negedge fast_clk)
    $python("pos_fast()");

always @(negedge slow_clk)
    $python("pos_slow()");
```

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 pos_fast():
  valid  = veri.peek('tb.dut.valid')

def pos_slow():
   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')
```

in 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 the 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' . "force" is misnomer, it is more like deposit. Binary string length is unlimited.

These two are almost all there is.

Passing run time params

In tb.v. (testbench):

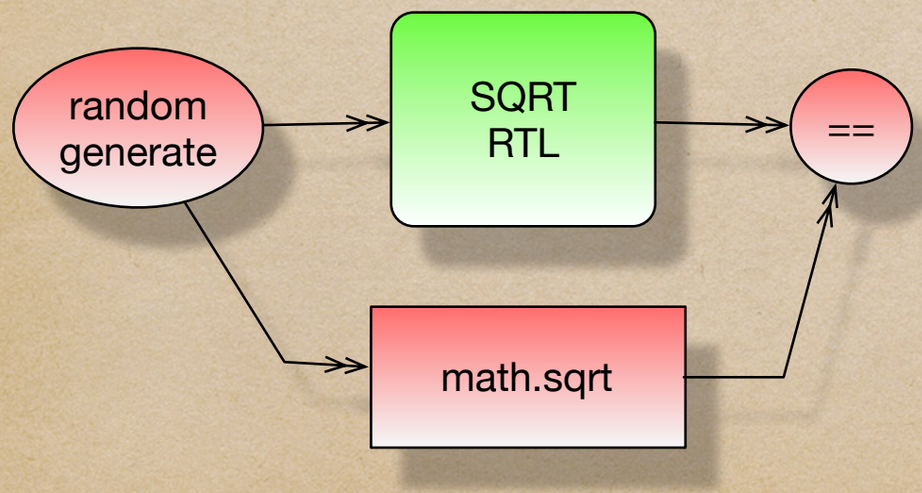
```
reg [1023:0] testname;
initial begin
    if ($value$plusargs("pymon=%s",testname)) begin
        $python("pymonname()",testname);
    end
    if ($value$plusargs("seed=%s",testname)) begin
        $python("random_seed()",testname);
    end
    if ($value$plusargs("SEQ=%s",testname)) begin
         $display(" Running SEQ= %s.",testname);
    end else begin
        testname = 0;
        $display(" default test");
    end
    #10;
    if (testname!=0) $python("sequence()",testname);
end
endmodule
```

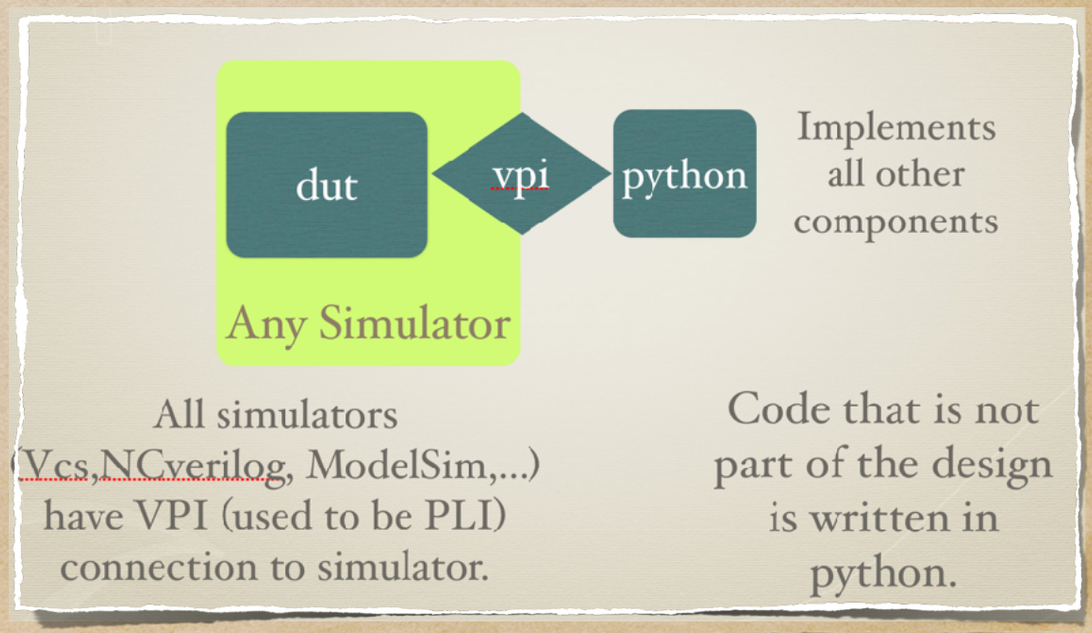
In verilog.py:

```
def pymonname(TestName):
    Fname = logs.bin2string(TestName)
    logs.pymonname(Fname)
def random_seed(TestName):
    Seed = logs.bin2string(TestName)
    random.seed(Seed)

def sequence(TestName):
    Seq = logs.bin2string(TestName)
    seq.readfile(Seq)
    logs.setVar('sequence',Seq)
    Dir = os.path.dirname(Seq)
    logs.setVar('testsdir',Dir)
    logs.log_info('SEQUENCE
%d'%len(seq.Sequence))
```

use example

- debug of sqrt, sinus, cosines and such. just call import math, and compare against math.sqrt()
- debug of encryption (triple des, arc4) and signature (hash256) modules, using freely available libraries. lately XTEA.
- Shallow test bench: instance of the DUT, regs on inputs, wires on outputs

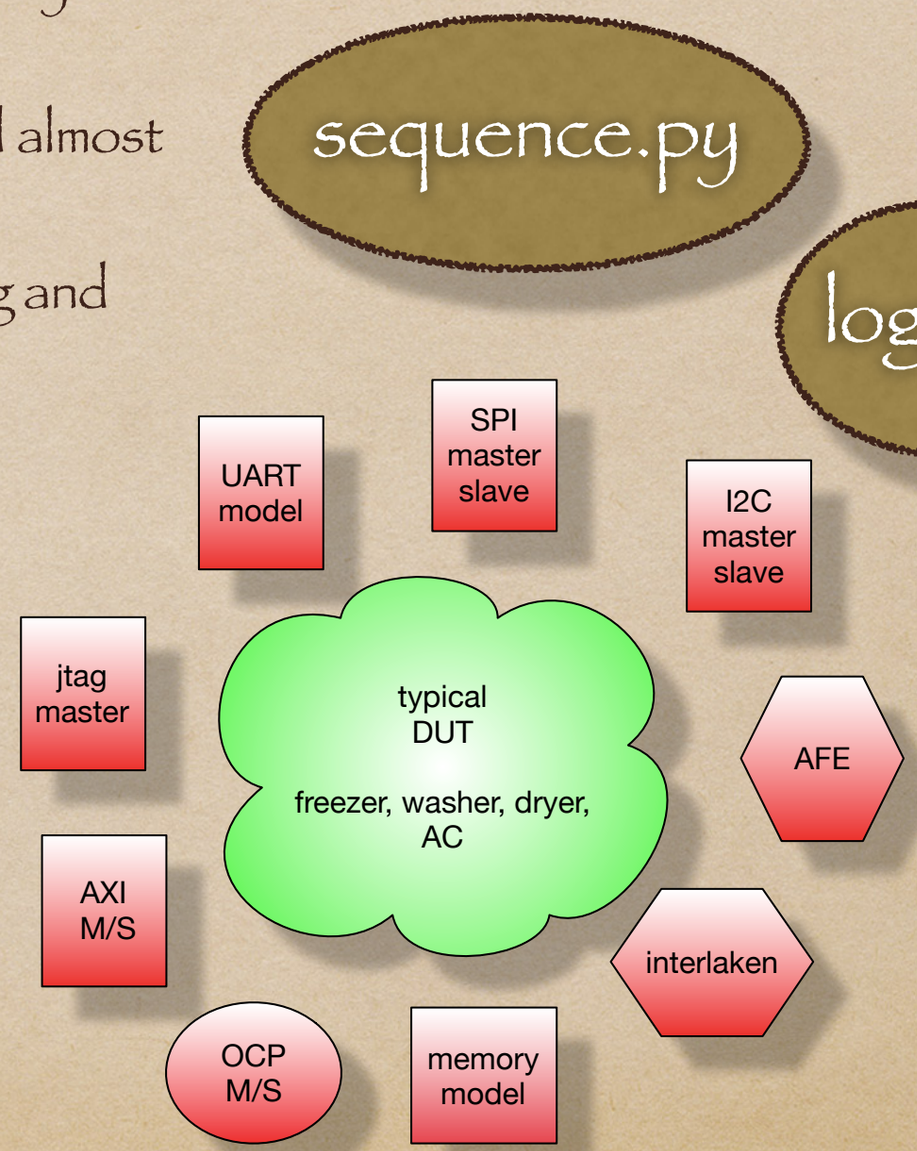




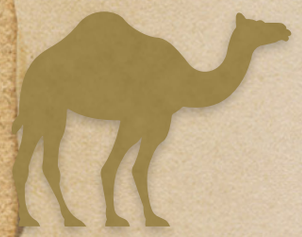
Typical module level

Typical module level verification. It is backed by many PyVIPs. But more importantly, the effort to build any interface VIP is small.
There is no deep hierarchy of python classes and almost no inheritance (because it is not needed).
logs.py is common file that takes care of reporting and converters.

```
module tb;
reg [31:0] DELAY1 = 1000;
always begin
   #(DELAY1);
   $python("work1()")
end
```



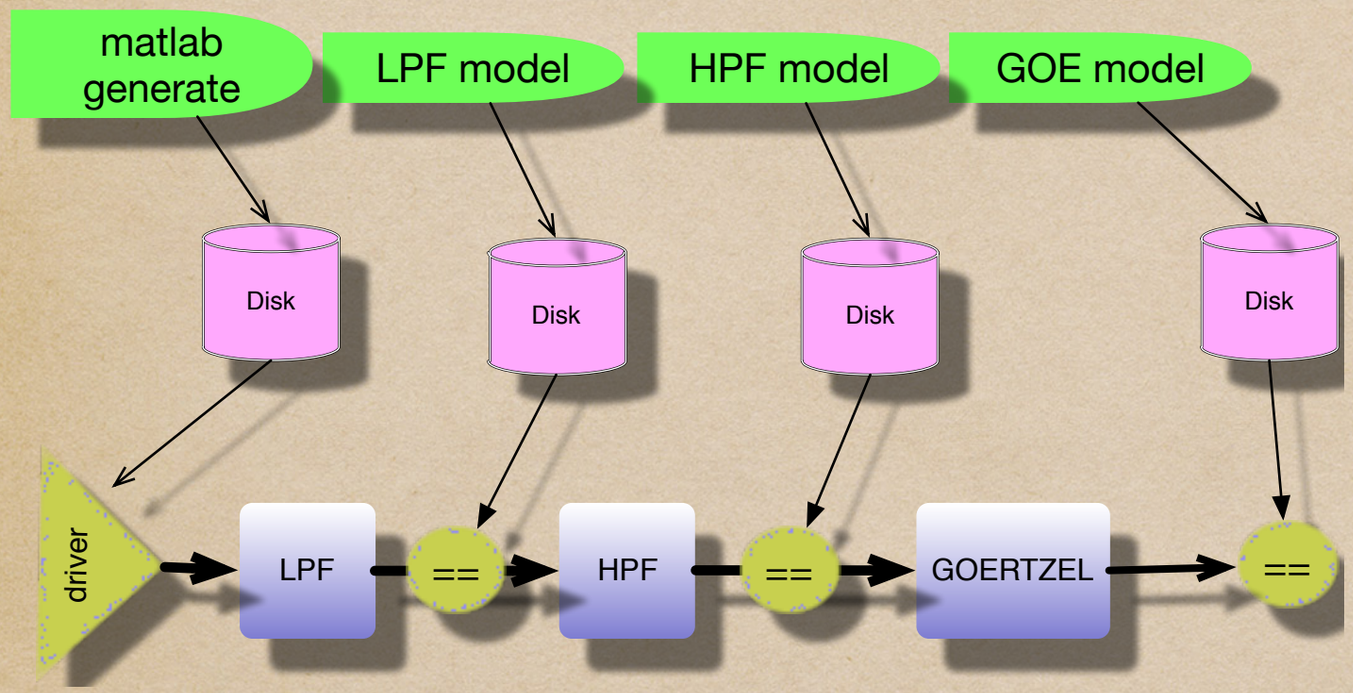
Python - why bother?



* python is structured, object-O, organised and fast.
* It was built for humans and keeps making life easier.
* it is open source, platform agnostic and everything is downloadable. has wealth of libraries written by many people.
* these libraries cover almost anything imaginable: MATH, MPEG, encryption, video, AI, ML, compression and all other buzzwords. Writing Your own is simple.
* All objects are first class variables and there is eval()
* RTL is compiled once. You can improve python code endlessly without need to recompile RTL/GLV.
* Legit question: What python gives, that E/SV/ Verilog doesn't give?

DSP verification

Classic: Onion Style



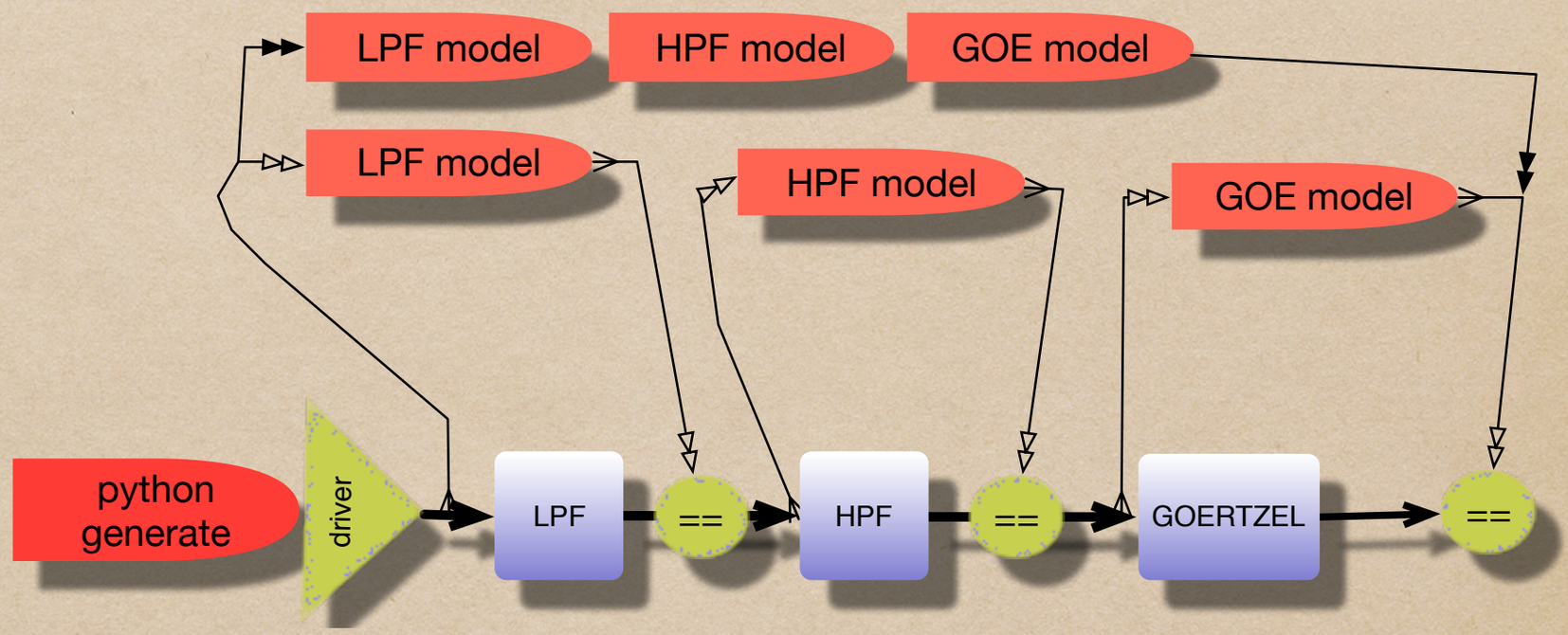
Matlab golden model produces files with driver data and expected results

Cons: first module in line not to be bit exact, foils it for every module downstream.

Cool-startup.com offices , 23:00, still peeling onions.

DSP verification

Onion unraveled



"Aluka" monitors all module inputs, passes the values to golden model, and lastly compares results.

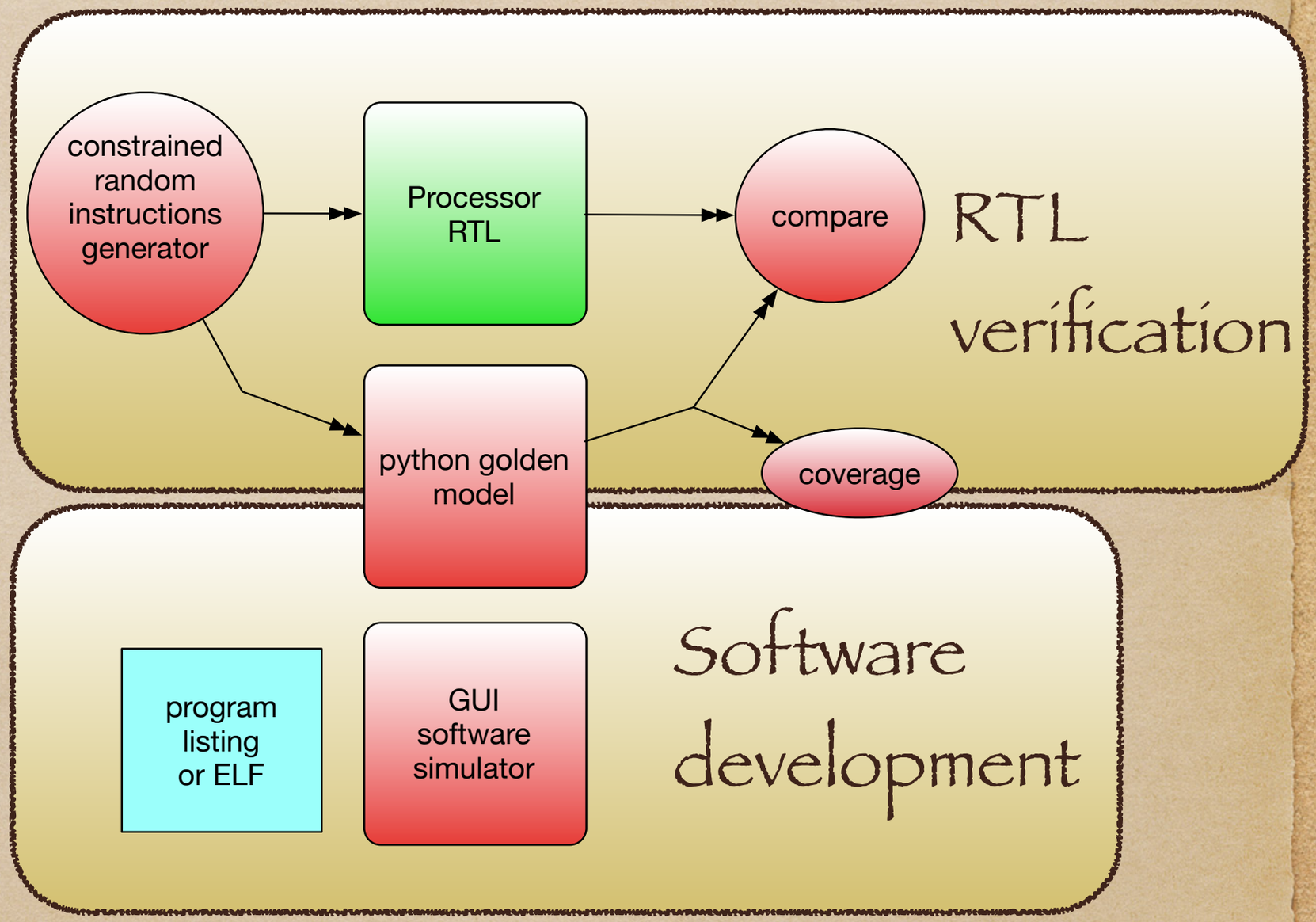
End-to-end values checked also.

Processor verification

Based on a true story

compares regFile values, program addresses. Coverage tallies opcodes and opcode sequences.

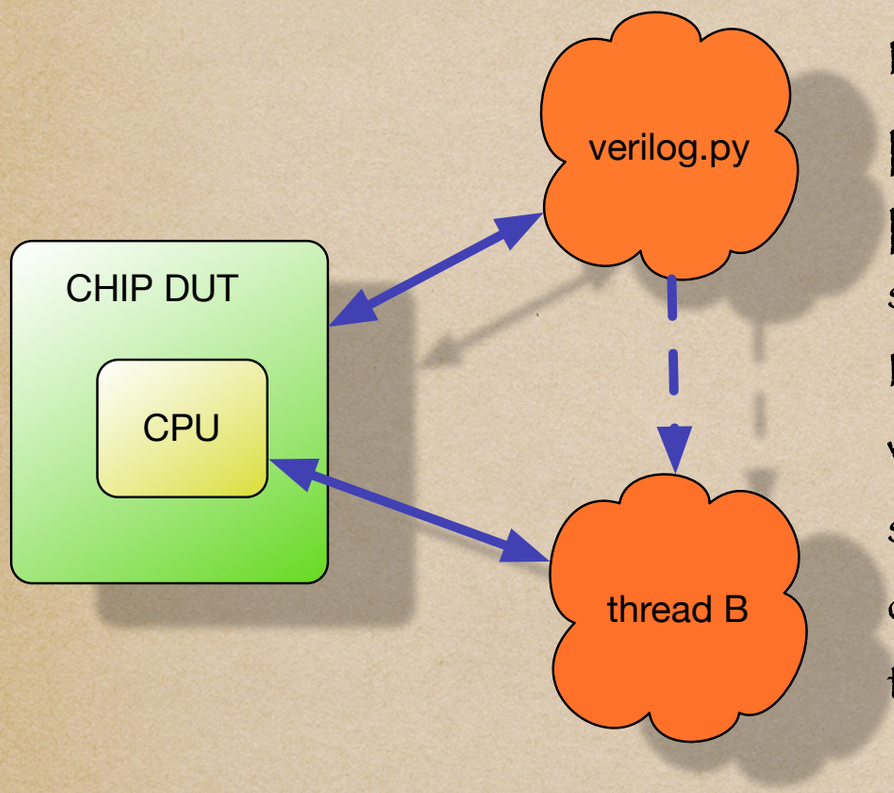
Same CPU model used for software development



Few Notes

- Constrained random has no need for elaborate solver. random is enough in 99%
- Strings driven verification. Interaction with simulator trough strings: signal path, values.
- Any python module has global visibility, if so needed.
- Python language has simple consistent syntax. You can understand anybody else's code (including Your own).
- Benevolent dictator notion. Keeps whole ecosystem intact.

interactive verification



Here is the new thing: I import the "thread" package to python code and open new thread. This thread uses "cmd" package to emulate shell terminal command loop. Now the simulation and command loop run in parallel threads.
It is pretty trivial to teach the commands loop to understand writes and any other command helpful to debug, like reset system, peek or force internal nets and so on. The commands that affect AXI (like reads and writes) are sent to AXI wiggle object.

The test is run in a bit different fashion. All the time the simulation is progressing. There is no finish directive. Nothing predefined. Typing on the terminal, I can modify different registers and observe in "real time" waves how the system is affected. The test is built interactively just like with real chip and system. I can give command to activate some external agent to start generate the traffic or shut down. Looking at live scrolling waves, it is obvious that only when new command is issued, the waves show some activity. Once the target behavior is achieved and everything looks OK, i can use the trace file to edit the final test.

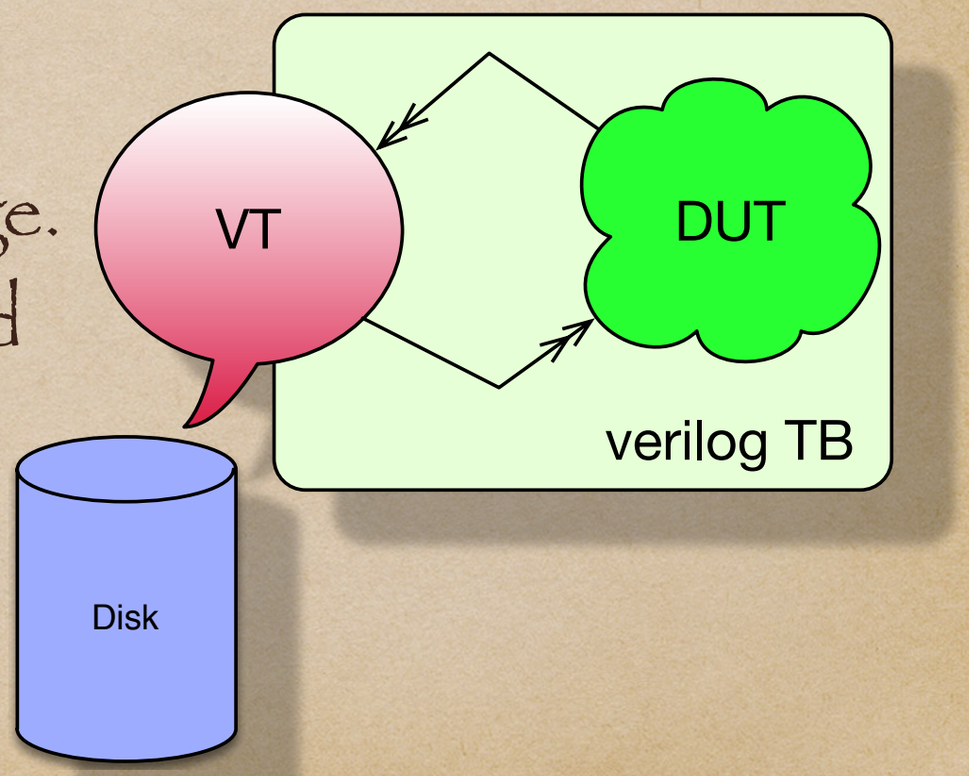
Few Notes

- "python mymodule.py" : simple way to check Your syntax.
- Python error messages are helpful and informative.
- While there are IDE's - i never had a need to use one.
- 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.

Virtual tester

Files sent to production tester, come in several flavors, most common today is STIL.
Python is used to read STIL file, imitate IC tester, drive and sample DUT pins.
This reduces test house reruns.

Python is a general purpose language. As such it has all the facilities to read and write all kinds of files. It makes it easy to implement such apps.

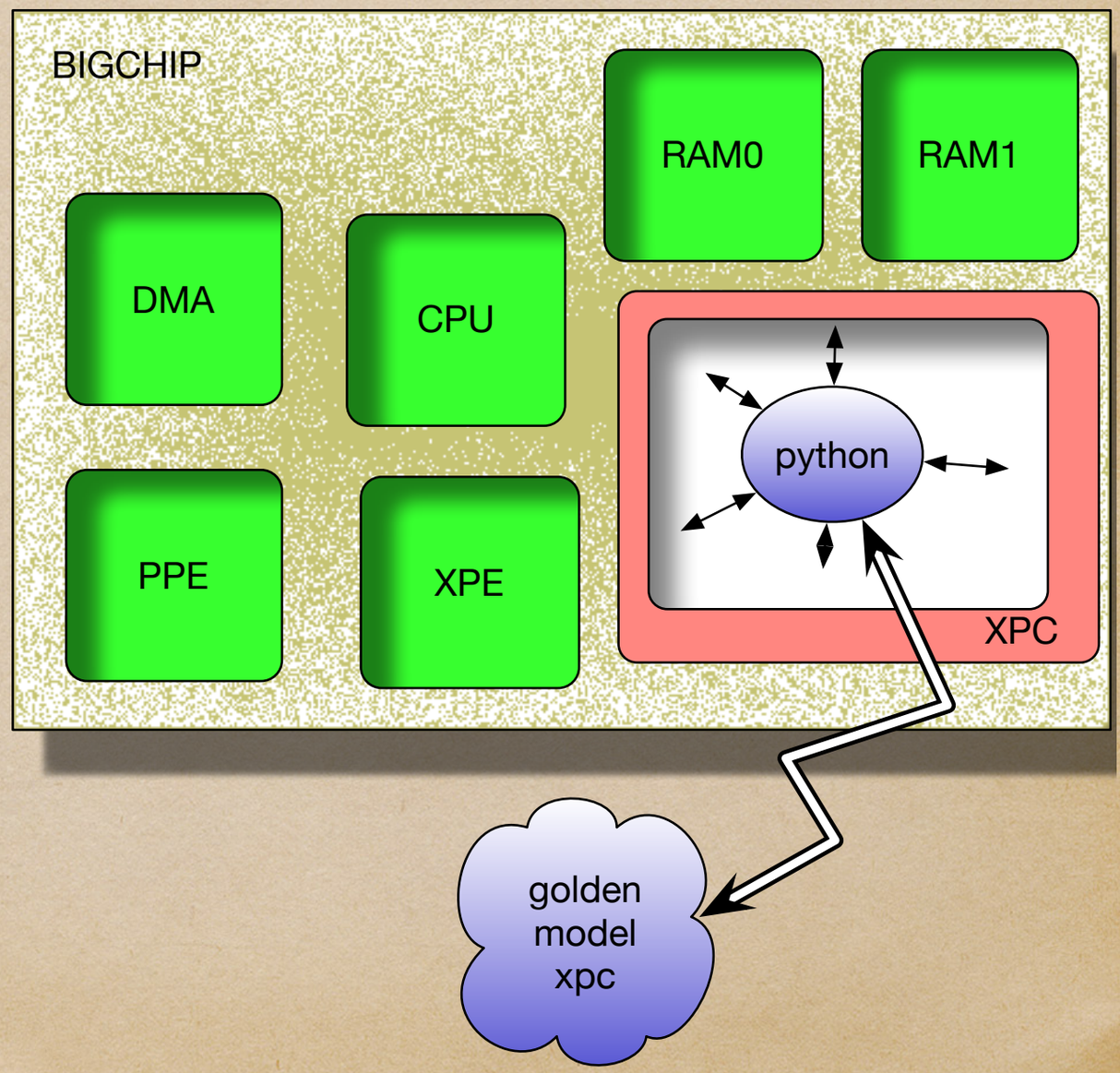


Early module filler

InsideOut Aluka

Long before some RTL module is ready or even written, Python golden model can be used as early implementation placeholder.

Create “pretzel” or “beigale” verilog module of the pinout. This module has just inputs and all outputs are regs (can be helped by pyver.py). Wrap around the golden model an adaptor to the actual module pins. This wrapper can model the processing delays. Voilà, system integration can proceed.

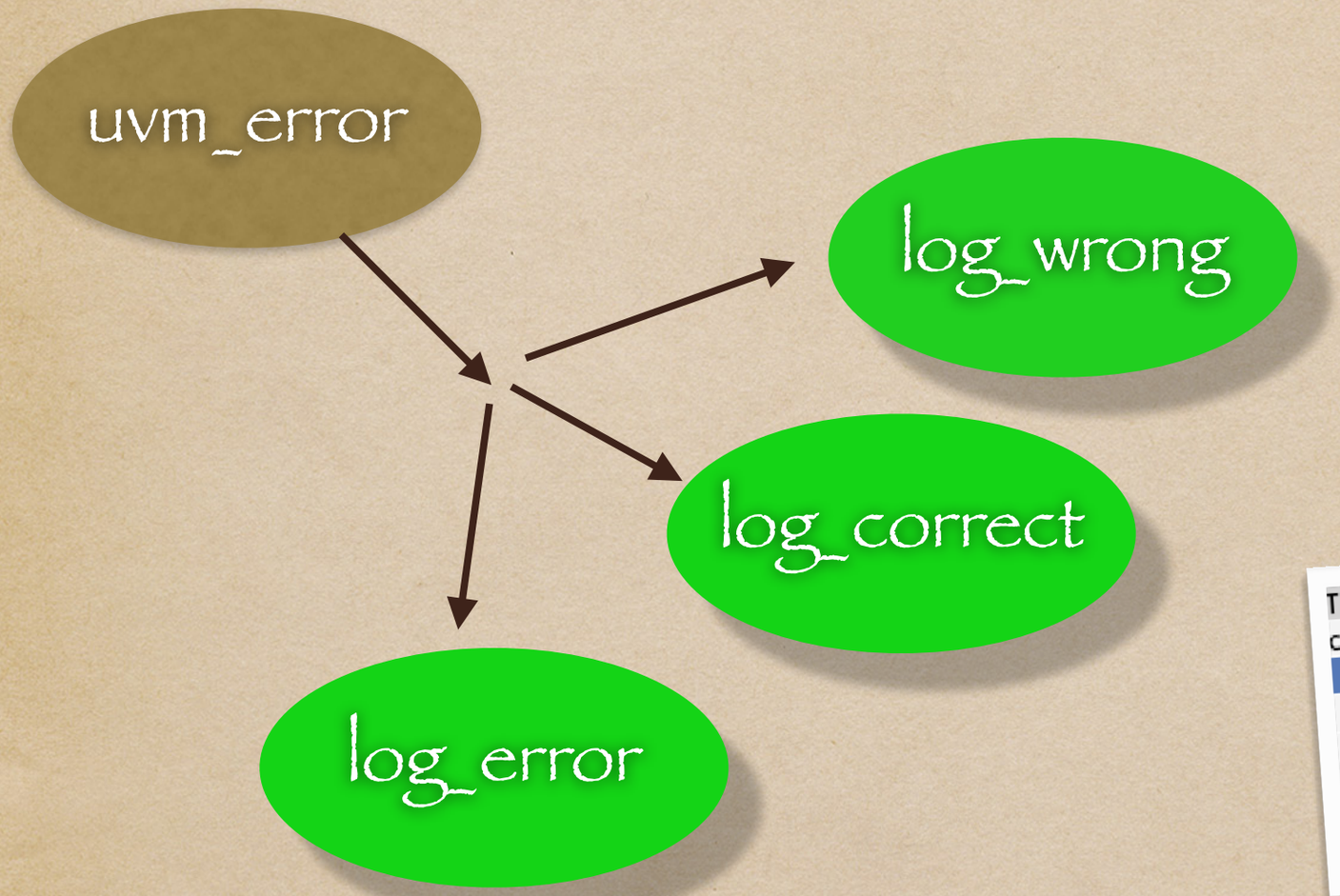


What is above veri?

In verilog.py

```
import axiMaster, apbMaster, sequence, logs
Monitors = []
axi  = axiMaster.AxiMaster ('tb.dut',Monitors)
apb = apbMaster.apbMaster('tb.slv',Monitors)
Agents = [('axi',axi'),('apb',apb)]
Seq = sequence.sequenceClass('tb',Monitors,Agents)


def negedge():
    for agent in Monitors:
        agent.run()
```

logs.py : common lib




```
module tb;
  reg [31:0] corrects;
  reg [31:0] wrongs;
  reg [31:0] errors;
  reg [31:0] cycles;
```

This module - keep logs and also update the tb.v counters. Thus we see in waves, where each occurred.

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

Sequencer agent

```
label  CHECK DELAY STUFFS
  include usefullstuff.file
  define STSTOP 3
  define CPU 0x3e0
  define RA  CPU+0x0
  prog_orig PROG
  Axi write 0x10 0x12222
  Apb read RAM+0xc
  writeProg  CLR+2
  catchTimer (4+6+1)*11+rnd(10)
  writeProg  INCR+0
  catchTimer (0x20<<4)*(4+6)
  writeProg  0x000   # stop
  wait 50
  force dut.run 1
  write 0     1        #controller enable
  wait 7000             #cpu stopped after this
  check cpu.state    STSTOP
  write RA  0x123
  waitUntil  (counter>0x100)
  check cpu.R0 0x456
  check cpu.R1 0x89a
  check cpu.jmp_ctrl_select0 0xd
  check cpu.jmp_ctrl_select1 0x2b
  check cpu.control_data  0xefab_defa_cdef_bcde
  wait 50
  finish
```

Easy to write class that reads files like the example. This module instances and can operate various agents. It reads the file and executes lines one by one. eval() function helps keep it readable. rnd() function facilitates constrained random. This naive methodology is applicable to many use cases.

Super agent

Super agent is a module that advances simulation by code

Coming soon.

Agents

```
class ahbliteMaster(logs.driverClass):
    def __init__(self,Path,Monitors,Translations={},Name='noName'):
        logs.driverClass.__init__(self,Path,Monitors)
        self.queue=[]
        self.seq=[]
        self.Name=Name
        self.waiting=0
```

def action(self,Txt):

    # Parses the Txt and executes the command.

    apb write RAM+0x40 0x55

def run(self):

    # Activated on every clock.

def busy(self):

    # Returns True if the agent still something to do.

Def onFinish(self):

    # Write to log internal status and statistics.

Panic driven verification

addition/replacement of asserts.

in rtl modules, we assign wires called "panic*" with expressions catching illegal/unexpected conditions/ results.

overflow, underflow, bad state, div by zero, bad value, etc.

During simulation, veri.listing() function can be used to scan and extract all signals called "panic*" and then monitor them throughout the simulation. At any clock they are active, an error is reported.

Same idea is to catch "X" (unknown) of important signals, like CPU opcode or fetch address. This is class of "coverage/interesting" equations.

Few Notes

Python connection can be added to existing testbench, it doesn't preclude of using SV/UVM/

Line Coverage: same as You do today.

Functional Coverage:

either as today or through python

Directed/ConstRandom tests: sequence.py

VIPs : write your own, or share with others.

what is bad about PyDrVer?

- You can work when others run out of licenses
- You can still work when servers go kaput.
- VPN frozen? No problem! Nx dies? So what!
- Develop all on Your laptop. Even WiFi is a luxury.
- No excuses for idling.

because using Free tools…

Level 1 Objections

here is list of the usual ones:

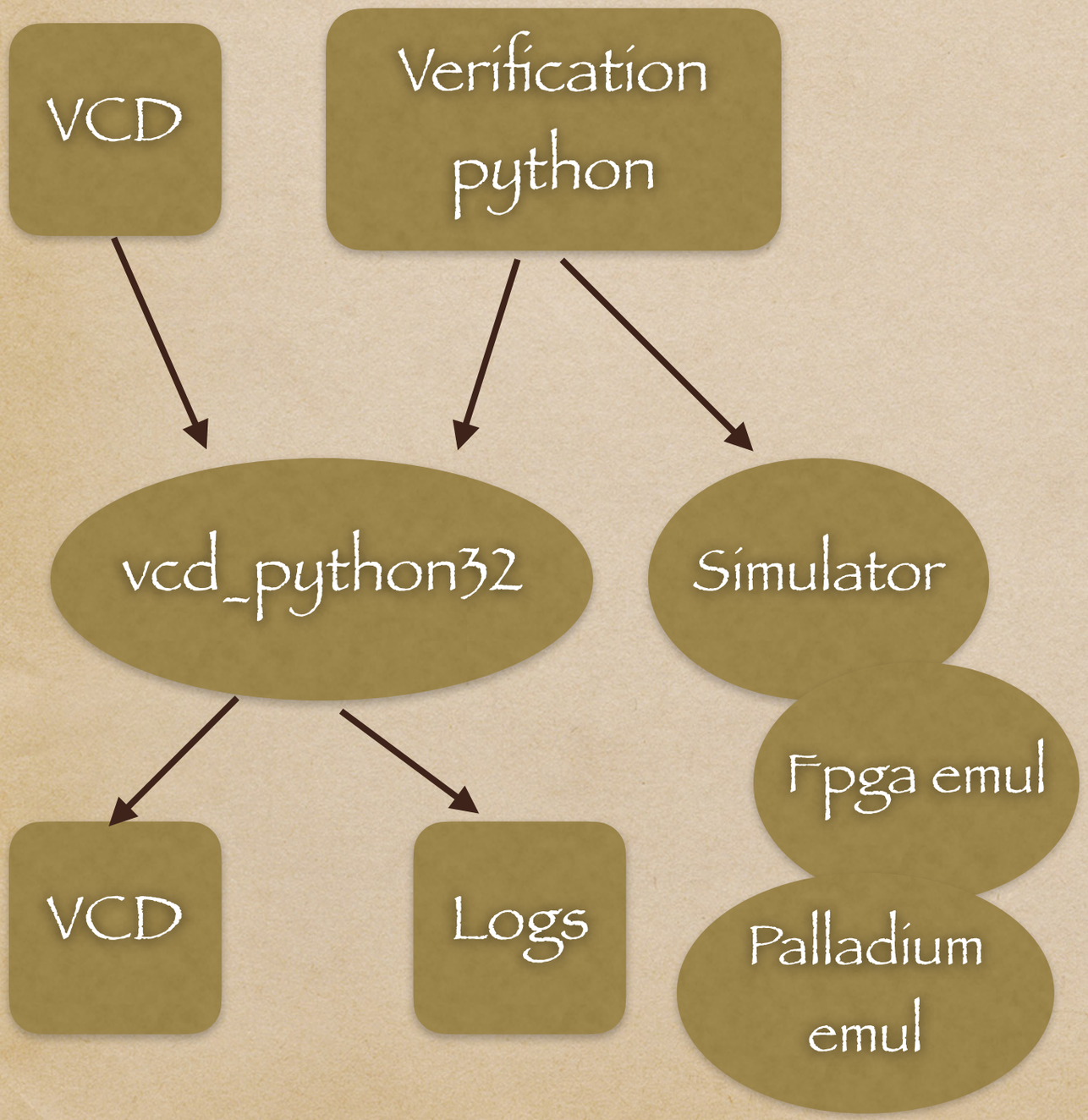
1. Constraints Solver (megoldó)
2. Threads: Fork /Join
3. Coverage (lefedettség)
4. TB debug
5. industry standard
6. throw away and redo
7. No “father” for free tools

Why Not CoCoTB?

In one word: Simplicity

CocoTb is also python based verification. It uses iterators / corotines. It looks sophisticated. But it complicates the debug. My way is string based all the way. No parallelism of coroutines. Everything is visible and simple.

vcd_python32



* This app applies python queries on VCD file. Python function can be “hung” on any signal or clock to be invoked when condition is met.
* Usually it is same Python VIP used in verification inall different platforms.
* New feature is ability to create a modified VCD file, Where python can change predefined variables in special “tracer” hierarchy. Flagging interesting events, errors, processing stages and more. I also debug VIP, where VIP class sees all input changes and drives out to “tracer” and to log file. It works well when there is no feedback from the design into VIP.

Wave formats

Verdi.	Synopsys.	Cadence
FSDB	VPD	SHM
Fsdb2vcd	vpd2vcd	simvisdbutil

VCD Value Change Dump

VCD is pure text file, editable with VIM

Some final observations

- all of the above goodies are possible, not because it is python, but because it is painless to implement in python.
- python verification (my way, not CoCoTb) doesn't have neither timing, nor fork/join constructs. It may look like a weakness, but i found that these things only complicate test bench debug and confuse things.
- power of python reveals itself in lack of need for deep inheritance. Inheritance is one of these things that look good on paper, but should have been tried on dogs first.
- VIP is scary buzzword for verification managers. Encrypted VIPs is the big rain maker for their providers.. My experience is that most interfaces (including DDR) is pretty easy to create. Sure it takes reading the standards, but after that - faster than negotiating with the supports. And no less important, when debugging the interfaces, You always need to be an expert in them. Anyway.
- Having python as the verification language enables us to use plethora of free tools - free simulators: icarus, cdc64 and others ; Wavers and even synthesis (Yosys).

more Ideas?

Be My Guest!

The End

I am on this ridiculous pet crusade to dethrone UVM/SV as primary hardware verification weapon and crown Python instead.
Annoyning retort, usually by design managers, is: "this is a language and that is just a language, what is the big deal?".
Yet none of them drives horse buggies, but rather hybrids. This reply reminds me of the timeless motto: "Anything looks easy, if you don't have to do it yourself."
But really, what is the biggest differentiator between SV/C++/UVM and using verification language like Python or Perl?
IMHO: it is "eval()" function. (and it's cousin "exec()"). Beat that C++!!
This function is a gateway to Shangri-La, no less. It single-handedly allows to create and execute code dynamically, reacting to ever changing environment.
It turns strings into actions.
But... not a single "I am hiring" post in LinkiDinky looks for verification experts with python skills.
I come across invites to SV/UVM classes and i pity the victims that go there, because "it is an industry standard".
Well, it is a poorly contrived standard, IMHO again. with SV/UVM manpower utilized under 50%, It is funny/sad to see companies crying that it is impossible to find verificators.

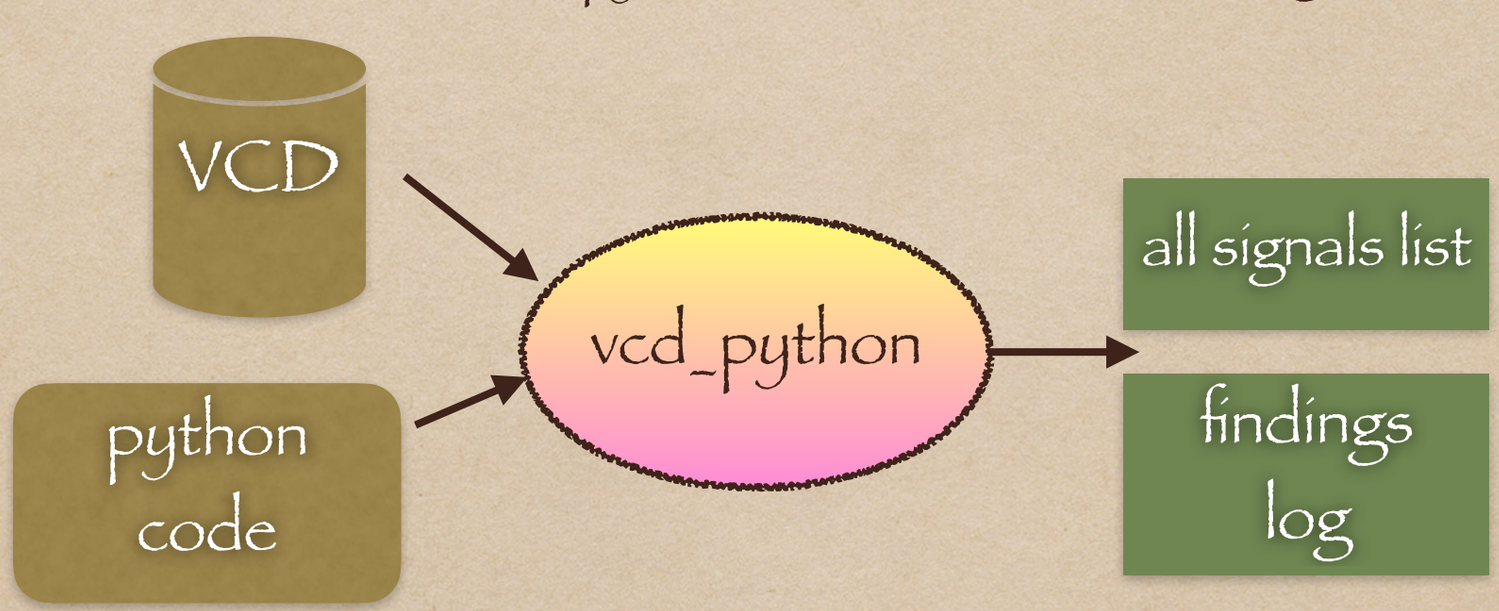
Other free stuff

more shareable ideas

not part of the presentation

vcd_python

python driven verification is a great tool to replace SV/UVM or E. But when my design is verified by somebody else, all i get back is cryptic UVM log and waves. To speed up the debug process, vcd_python is useful. It takes my python verification code and applies it to waves - mimicking real simulation. Of course, it can only monitor. But think about AXI sequences or SERDES lanes - Same python code extracts useful insights.




```
veri.sensitive(CLK,’0','work()')
def work():
    if valid('validin') and valid('takenin'):
        data = peek('datain')
        logs.log_info('data valid = %x'%data)
```

verilog reader /writer

- verilog parser (based on yacc/lex) - It reads verilog files, builds data structure and passes control to an application which can manipulate them. At the end it can dump corrected verilog back or just report.

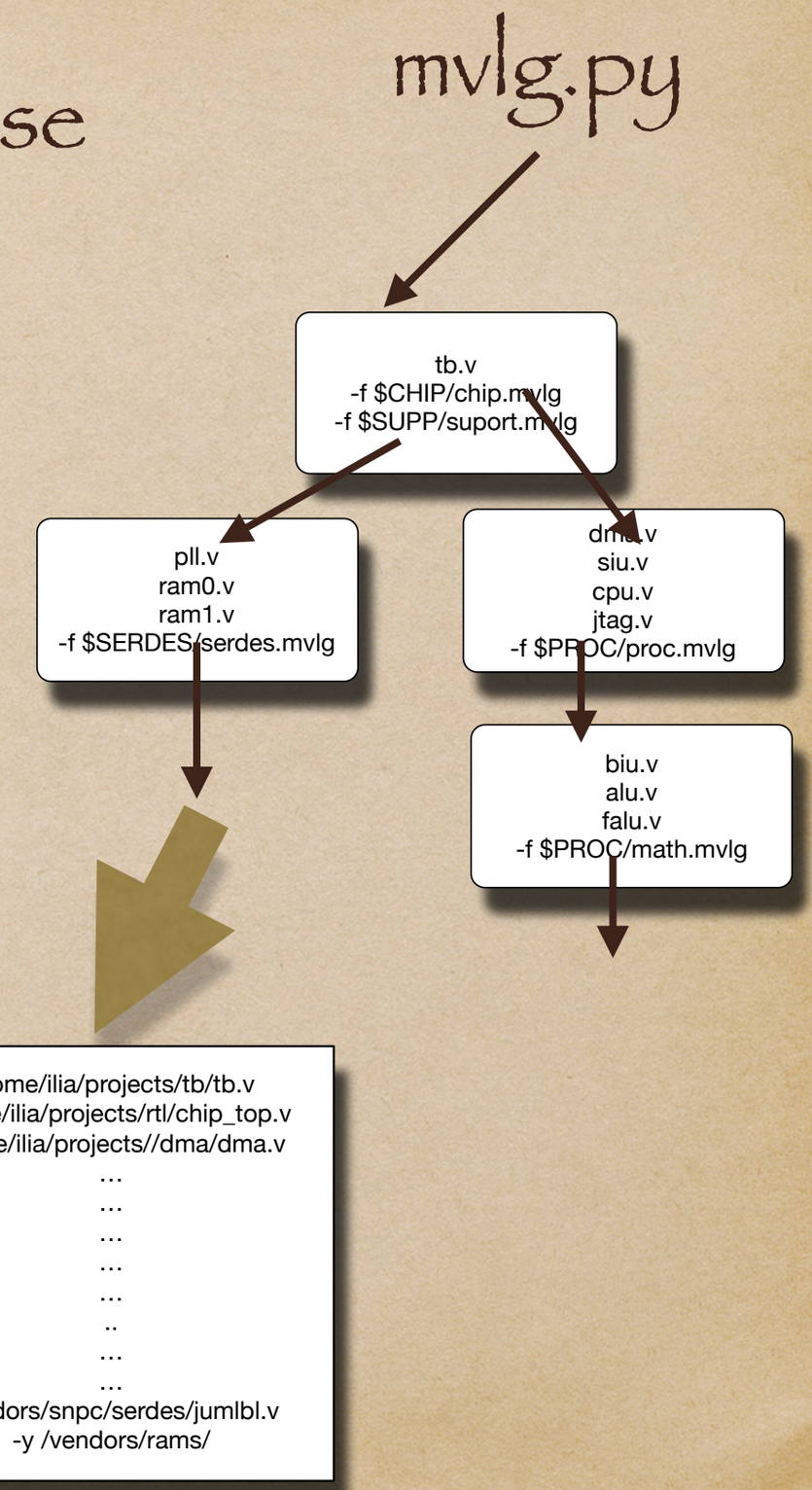
- Why? build hierarchies, re-arrange hierarchies for backend or simulations, change ASIC to FPGA code, answer queries about the database, Insert pieces of debug code. Insert and remove BISTs. Look for loops and basic check integrity stuff.

- It gives You sense of control over the source code.

MVLG

How sim/syn knows what files to use

- original RTL is organized in several directories. Each directory holds many RTL files.
- For simulation, there are also support files, like TB, RAMS, ROMS, Flash,Pll and such.
- We place .mvlg file in each directory. This file lists all relevant local files for simulation/synthesis. It also has -f lines to indicate dependencies. -f is just like in simulator :: pointer to another MVLG file. It may also have -y lines and use $ENVIR variables.
- The script MVLG receives top MVLG file name and recursively builds list of all relevant files. The result lists all files with absolute pathnames. It removes double definitions.
- There is an option to concat all files into one big file, or create directory with copies or links or split one multi module file into directory of modules.
- Either the list or big file is then given to simulation or synthesis or logic equivalence.
- original MVLG idea is copied/duplicated/shamelessly taken from Virata RIP.



register file gen.

XLS file

or

text file

RTL

Py script

XML

XLS

.h

.vh

The whole flow is

“GIT”, “SVN”, “CLEARCASE”

compatible.

External IP

- Ceva, Mips, Synopsys IPs come in multitude of directories.
- After configuration, to use the IP, You still face numerous RTLs and many "define" files. RTL files are laced with ifdefs and includes.

- The bottom line it is hard to find Your way around it and also hard to pass to simulation or synthesis.

- Our solution? Script to concat all files into one big happy file and resolve all ifdef and includes and replace all defines with their computed value.
- Easier on synthesis just to give one file. Easy in debug - because everything is spelled out.
- Don"t buy IP if You can and want to do it yourself.

TSMC cell libraries

- We employ a parser of liberty files. Why?
- it reads liberty files (.lib) and allows to create many different views of the cells.
- Views: simple for simulation and debug, for gate level debug, for atpg, for power counting, for fault simulation.
- for choosing specific subset or reducing size of libraries. for mixing libraries.

Layout reader/writer

- What is it? Reads GDSII and dumps readable format. Reads this readable format and dumps back GDSII.
- Why? modify final GDS (add logo?). Verify the health of GDS. Find hidden instances. Find hidden texts. Extract coordinates of nets.

Genver: replacement of generate.

- I hate generate statement in verilog (and instance[0:3] for that matter too).
- It produces strange names and ugly code, You cannot always deduce what is produced and it is limited to whole syntax structures.
- Solution? humble app called genver.py. It is macro preprocessor, where the macro language is python.
- Looks like verilog code spiced with python commands. Then expanded into full verilog file. This file is readable and debuggable.
- The translation is fast and usually done online with simulation or synthesis.

```
#WID = 30
##  this will become python comment for yourself
module example (   input clk,input rst
#for II in range(WID):
   ,output [II:0] outII
#    JJ = WID-II
   ,input [JJ:0] inII
#
);

#for II in range(WID):
son sonII #(II,WID)  (.in(inII),.out(outII),.clk(clk),.rst(rst));
#

#for II in range(10):
wire [WID:0]  wwII = 0
#    for JJ in range(5):
   + cucu[JJ]*cooef[II]
#<
   ;
#
endmodule
```

Fixed and Float math lib generators

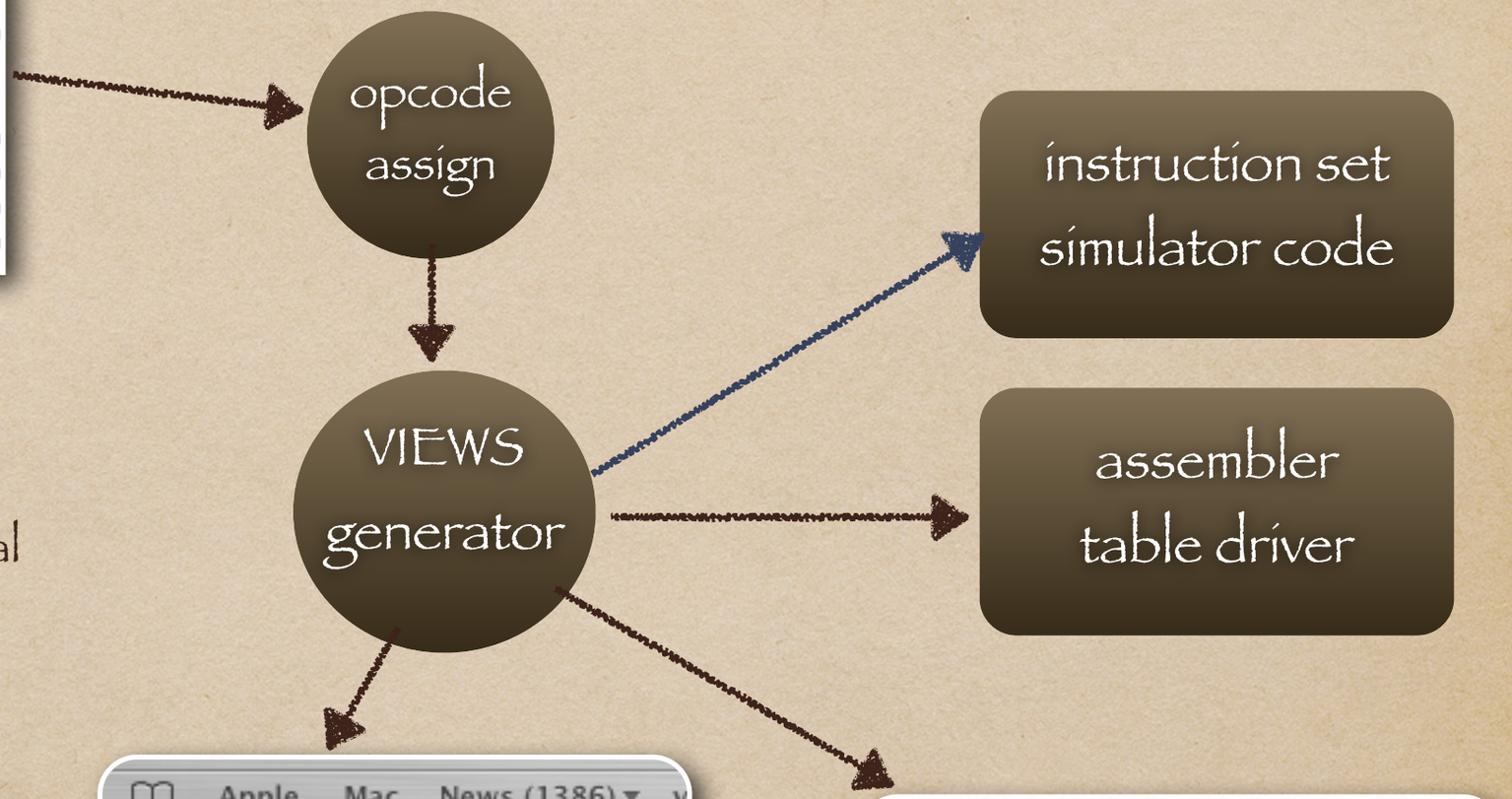
- Script accepts either list or single math module name and creates it.
- It can create mul, limited mul, divider, limited divider, square root, distance module for fixed point. Configurable are width of operands and number of stages.
- Cordic and table based sin/cos/atan/asin/acos were added lately.
- The system generates to exact spec, no defines nor parameters. WYSIWIG!!
- and no surprises in logic equivalence. and if better algorithm exists - it can be incorporated in matter of minutes.

Instruction set gen

```
1
2  opcode_width=32;
3
4  instruction=add16      coding=B32 one
5  instruction=return coding=B32 one
6
7  instruction=addi8      coding=B6,r
8  instruction=subi8      coding=B6,r
9  instruction=andi8      coding=B6,r
10 instruction=ori8       coding=B6,r
```

configuration text file

Start by creating opcodes definition file. The source file describes width and fields of opcodes, the script assigns actual static bits to the opcodes and produces verilogRtl and documents.
Useful if You want to implement custom processor or knock-off existing processor.



Apple .Mac News (1386)

opcode							
bits	31	30	29	28	27	26	2
load	0	0	1	1	sz[1:0]		
andi16	0	0	1	0	1	1	
imm	0	0	0	1	sign	clear	
addi8	0	0	0	0	0	0	
cntxsw	0	0	0	0	0	0	

html document

```
39 ,output [3:0] ra_field
40 ,output [3:0] rb_field
41 ,output sign_field
42 ,output reg_field
43 ,output clear_field
44 ,output t_field
45 );
46 assign load_code = (opcode & 32'hf0000
47 assign andi16_code = (opcode & 32'hfc0
48 assign subi8_code = (opcode & 32'hfc00
49 assign addi8_code = (opcode & 32'hfc00
39 ,output [3:0] ra_field
40 ,output [3:0] rb_field
```

verilog decoder code

Network on a chip

only messages run around

system built on no-instant gratification (everything takes time)

totality: only rings allowed. there are no wires coming out of modules, except rings.

comprehensive: rings carry control and data: interrupts, status, data, clocking, power save modes, debug and test.

topology doesn't influence functionality, only performance.

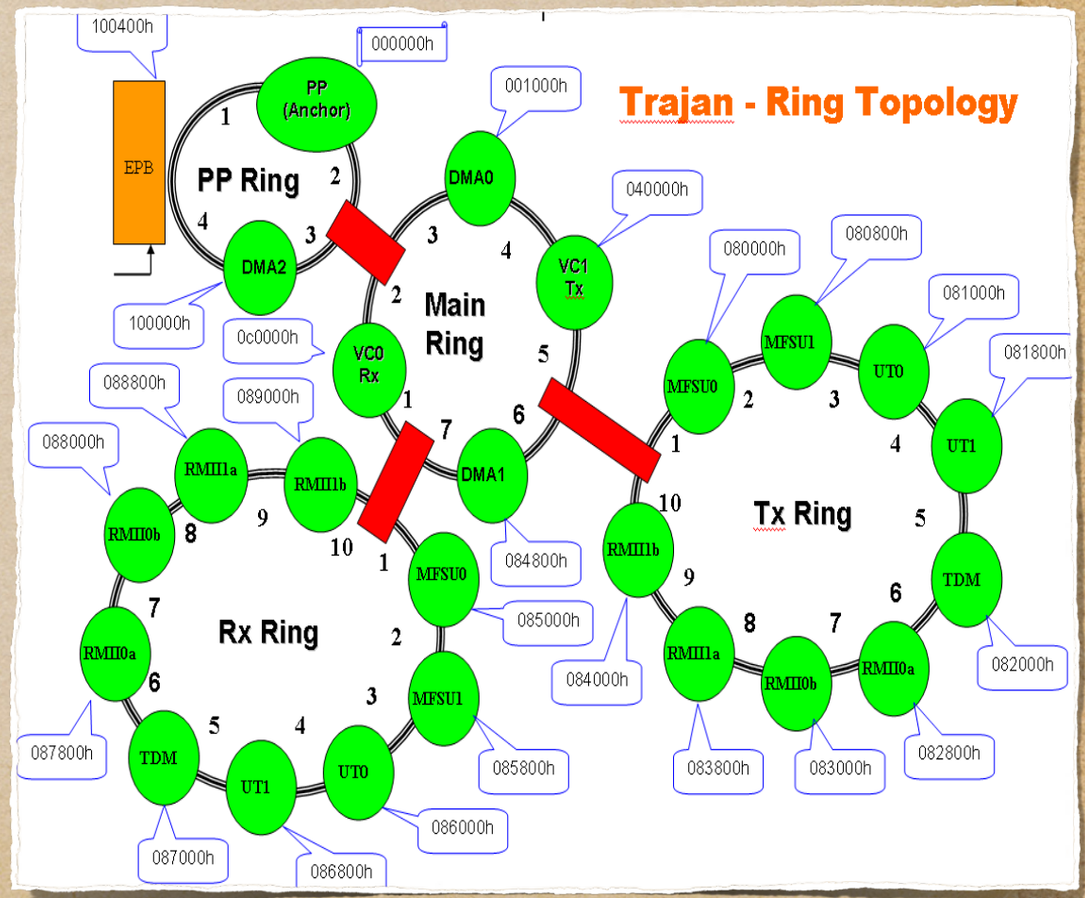
no masters / no slaves - all members are producers and consumers.

keep it simple: documentation should be shorter then the code.

ditch signals and busses - define the chip by transactions and flows. This applies to status registers, Interrupts and other requests and acknowledges.

e.g. read msg

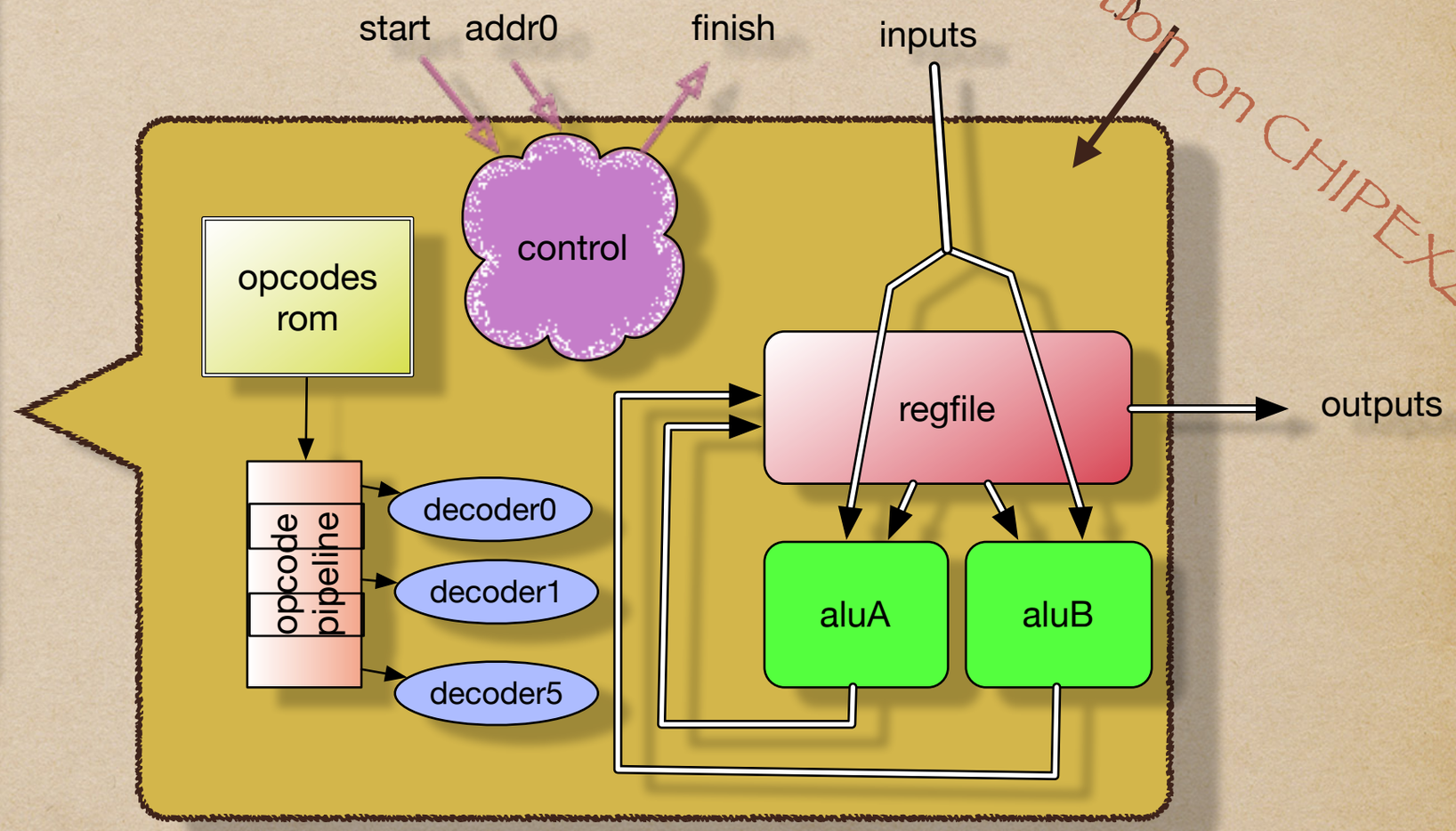
10	ctrl bits	addr		byte count	return addr
2bits	6bits	24bits	32bits	8bits	24bits



FloatProc

Look up presentation on CHIPEX2017

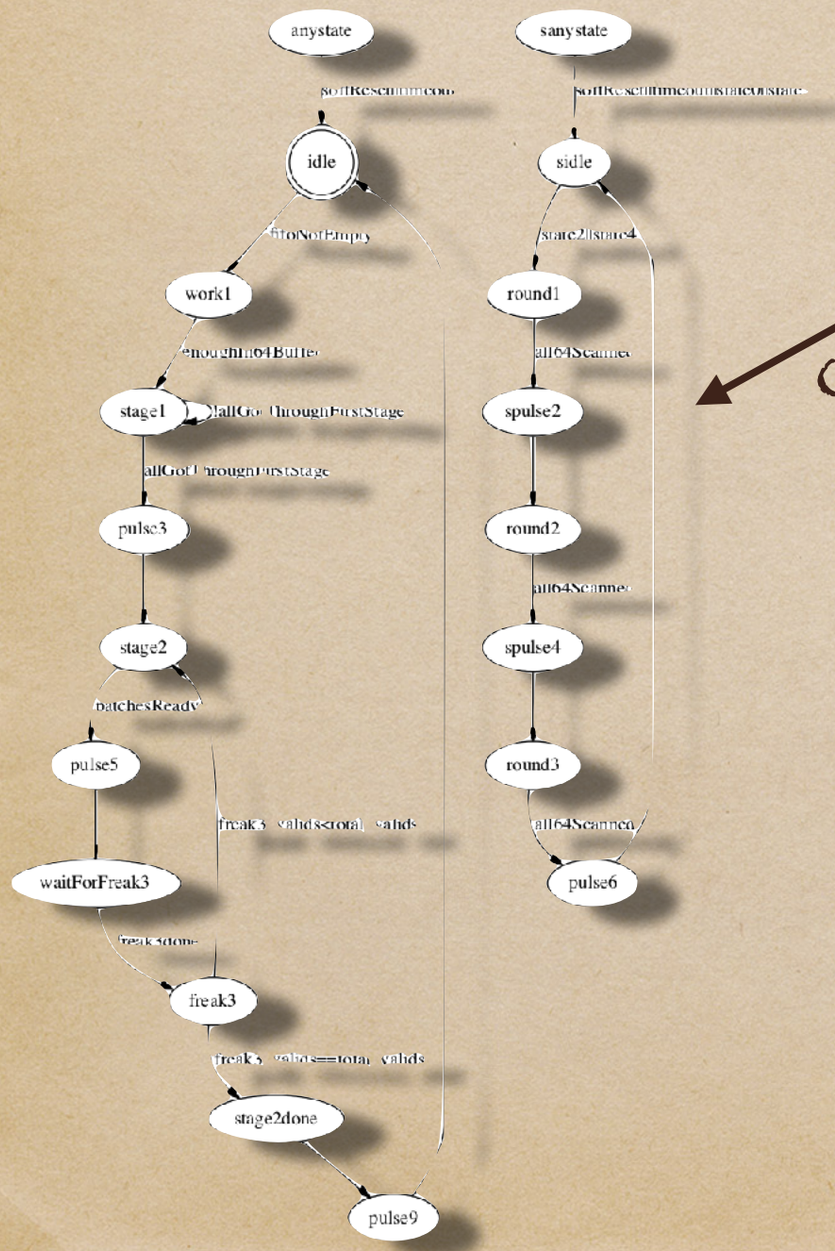
add
sub
mul
div
sin
cos
acos
asin
atan
abs
sqrt
select
compare



- dual issue
-black box setup
- up to 32 input values
-up to 32 registers
- dflow compiler programming

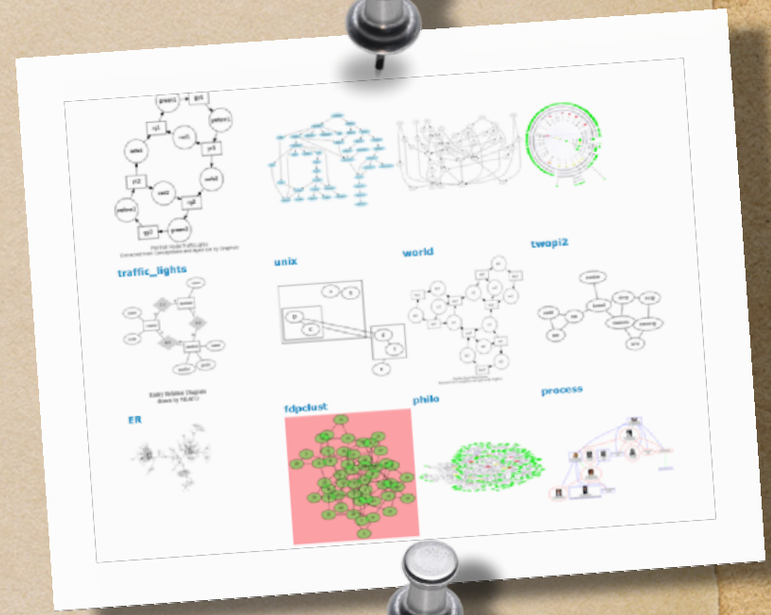
Dot : diagrams the human way

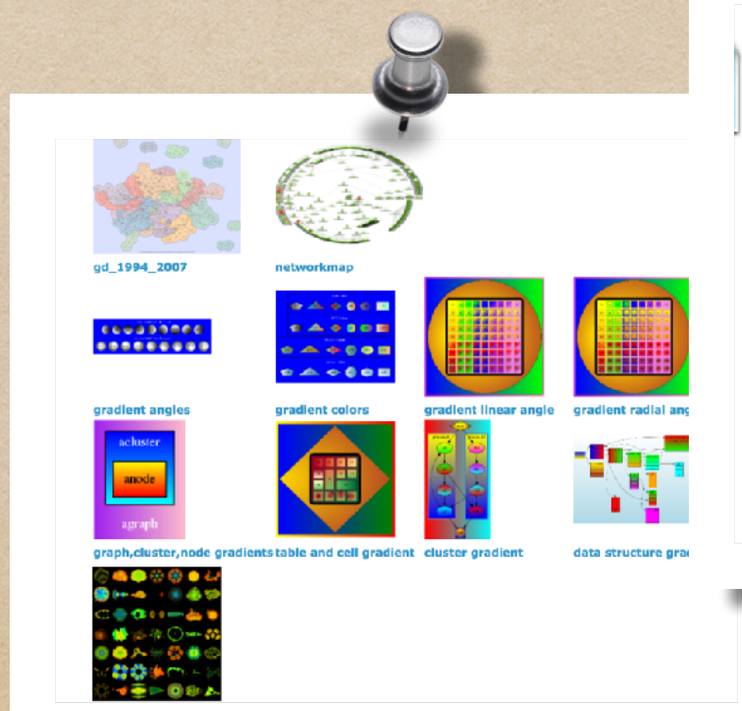
http://www.graphviz.org

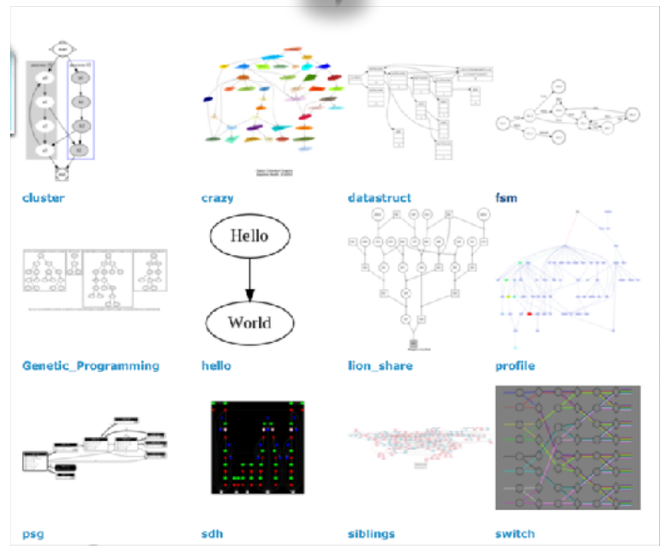


dot

```
aa -> bb;
aa -> cc;
cc -> idle;
```



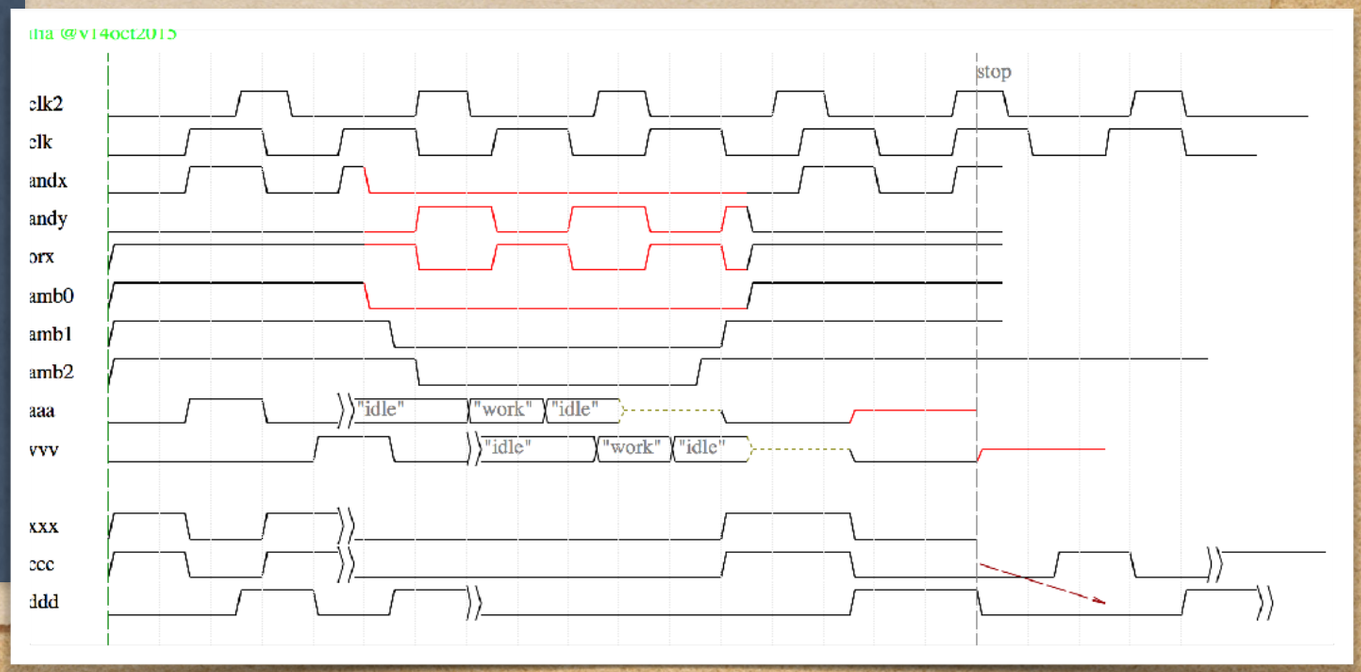




waveformer

```
sig clk (repeat 20 (3 0) (3 1))
sig clk2 (repeat 20 (5 0) (2 1))

sig amb0 (10 1) (color red) (15 0) (color black) (10 1)
sig amb1 (11 1) (13 0) (11 1)
sig amb2 (12 1) (11 0) (20 1)

sig aaa   (3 0) (3 1) (3 0) break (5 "idle") (3 "work") (3 "idle") (4 z) (5 0) (5 x)
vgrid 0 2
sig vvv (delay 5 aaa)
sig xxx (invert aaa)

sig ccc xxx (mark x0) (invert xxx)
sig ddd (delay 5 xxx)  (mark x1) (invert xxx)

vmark stop (mark x0)

sig zzzzzzzzzzz (3 0) (2 up) (4 1) (2 down) (5 0)
sig andx (and amb0 clk)
sig orx (or amb0 clk)
sig andy (and (not amb0) (not clk))

range 0 (len amb2)
arrow (mark x0) (mark x1)

color vmark 0.5 0.5 0.5
color sig 0 0 0
color arrow 0.6 0 0
display clk2 clk andx andy orx amb0 amb1 amb2 aaa vvv * xxx ccc ddd *
```

waveformer



Yosys : open synthesis

Yosys runs on linux. It accepts synopsys libraries after some filtering. It reads most of RTL (at least the ones i tried).

Why use it, when there is Synopsys/Cadence?

1. in seconds, You get indication whether RTL is valid.
2. Combi Loops easy to find.
3. Early area / Timing indications.
4. It is fun.

more Free tools that we use.

- gtkwave and dinotrace for VCD waves viewing.
- icarus, verilator and cvc64 verilog simulators.
- zDraw schematic editor.
- mWave analog waves viewer (from .csv files)

homemade tools examples

- rename module/instance names in the whole verilog hierarchy.
- expand (make explicit) defines/ifdefs/genvers/parameters in verilog code.
- Fubarize code.
- XLS/CSV to RegFile generator.
- lef / def parser and translator.
- VCD analyzers.
- ATPG formats translators.
- Simulink diagram convertor to rtl, processor or assembler.

Summary till now

- Each item above is not earth shattering by itself.
- But their collectivity and change of attitude do carry a big impact.
- Try to move the bar between self-made and purchased stuff higher.
- ריכוך ארטלרי
  All examples before this one were part of "artillery barrage" in the sense to make You less respect the well-trodden paths, open up Your mind to new ideas, especially python verification.