VSDBabySoC Modelling

Day - 12

20-09-2023

Today’s agenda

- Recap on SoC
- What does modelling mean?
- Modelling of the VSDBaby SoC
  - PLL
  - DAC
  - RVMYTH
- Basic introduction to iverilog & GTKwave
- References

Recap on SoC

- SoC is a single-die chip that has some different IP cores on it. These IPs could vary from microprocessors (completely digital) to 5G broadband modems (completely analog).

- SoC with equivalent functionality will have increased performance and reduced power consumption as well as a smaller semiconductor die area.

What does modelling mean?(electronics terminology)

Modeling and simulation is the use of a physical or logical representation of a given system to generate data and help determine decisions or make predictions about the system.
Models are representations that can aid in defining, analyzing, and communicating a set of concepts.

M&S is widely used in the VLSI domain.

Purpose of modelling :

1. System models are specifically developed to
    a. support analysis, specification,
    b. design,
    c. verification,
    d. and validation of a system,
    e. as well as to communicate certain information.

What are we modelling?

- Let us look into VSDBabySoC modelling.
  - Here we are going to model and simulate the VSDBabySoC

- Some initial input signals will be fed into vsdbabysoc module,
- That will get the pll start generating the proper CLK for the circuit.
- The clock signal will make the rvmyth to execute instructions and some values are generated, these values are used by DAC core to provide the final output signal named OUT.
- So we have 3 main elements (IP cores) and a wrapper as an SoC and of-course there would be also a testbench module out there.

This weeks task is to model the 3 main IP cores

1. RVMYTH modelling
2. PLL modelling
3. DAC modelling

Before that lets understand how each component works.

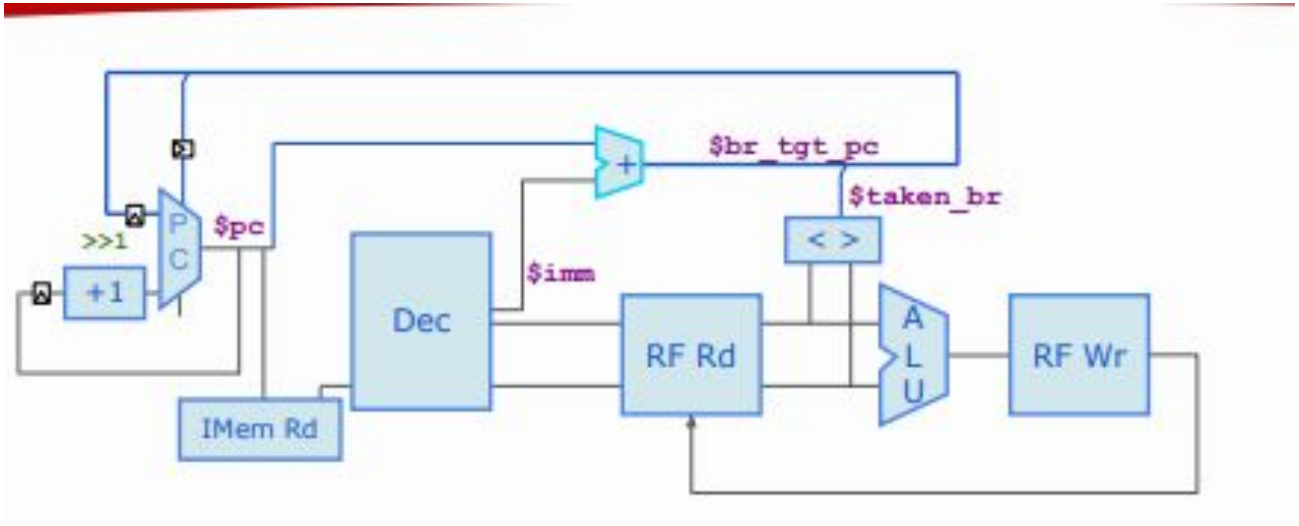
RVMYTH - Risc-V based MYTH (Microprocessor for You in Thirty Hours)

RISC stands for Reduced instruction set computer

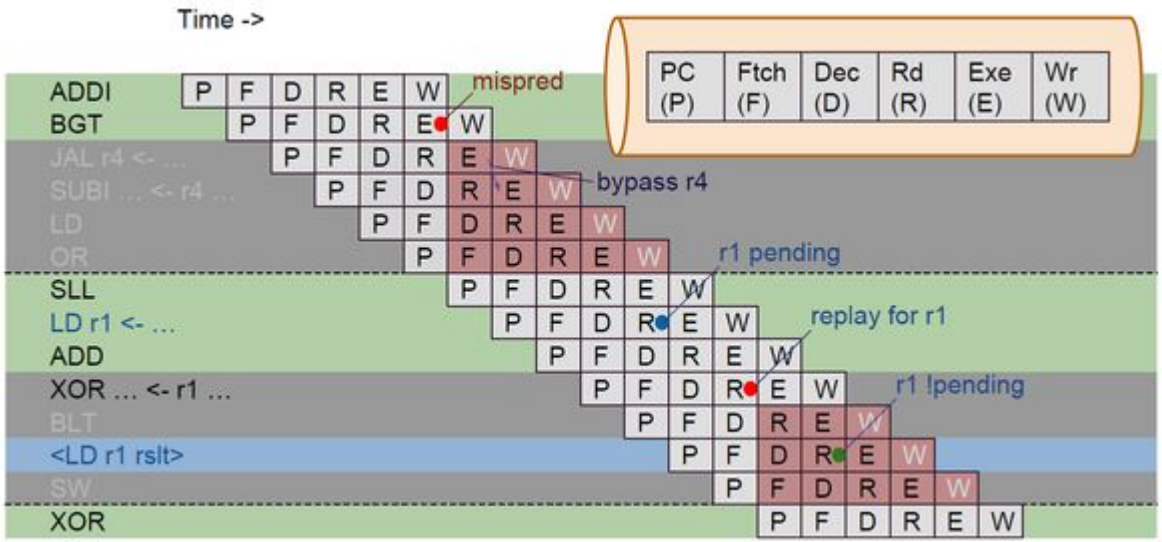
RISC-V(pronounced “risk-five”) ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA.

Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers. There are two primary base integer variants, RV32I and RV64I.

A simple one cycle CPU for Risc-V



Waterfall flow diagram for a pipelined Risc-v processor instructions



Phase Locked Loop

A phase-locked loop (PLL) is an electronic circuit with a voltage or voltage-driven oscillator that constantly adjusts to match the frequency of an input signal.

PLLs are used to generate, stabilize, modulate, demodulate etc

Now, question is why do we need a PLL for our SoC?

Before that how is a clock generated?

Quartz crystal oscillator

For 100Mhz and below off chip oscillator will do, but for 100Mhz and above it won’t be good enough.-how?

Why off-chip clocks can’t be used all the time?

- The clock will be a supply for a lot of blocks on the chip, it will have delays due to long wires(if used only one clock source) - also reasons like clock jitter
- Some blocks might need 200Mhzs and some might need 100Mhz - point is different frequencies just on one small chip
- A concept of ppm(clock accuracy) comes in, when ever quartz is acquired, it comes with a x ppm error

What is this ppm error? {ppm - parts per million}

For ex: 20ppm quartz used in watches

this translates as 20/1e6 (2e-5)

which gives an error over a day of 86400 * 2e-5 = 1.73 seconds per day

so in a month it loses 30 x1.72 = 51 seconds or 1 minute a month

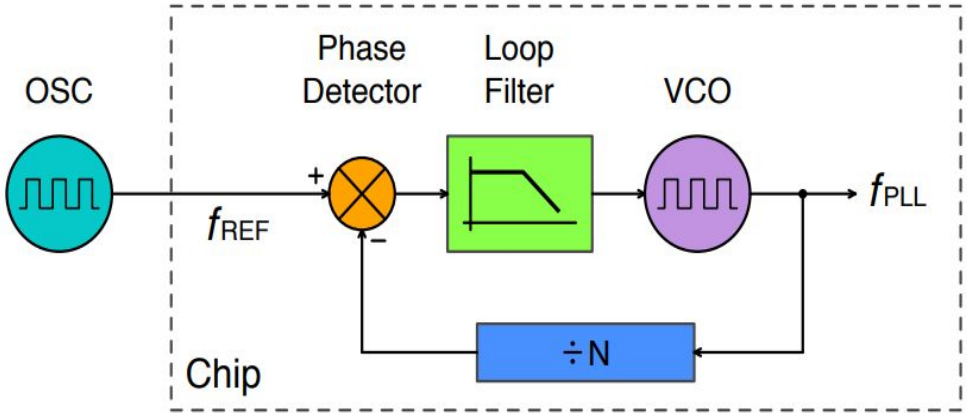
Now, in terms of a chip, just imagine the mishap it will cause just due to very small error for ,microseconds, when the processor works at nanoseconds -------- it can be a huge blow.

So a PLL used on SoC's

Main components:

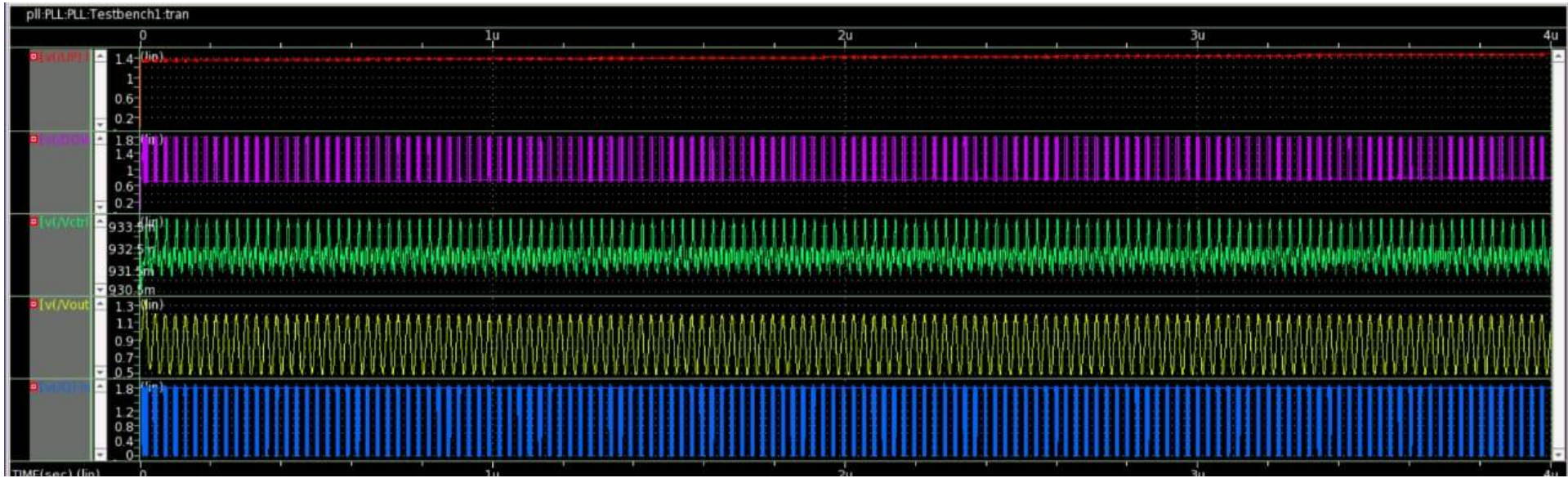
1. Phase detector
2. Loop filter
3. Voltage controlled oscillator
4. Frequency divider

Make an on-chip copy of the external clock:



Control loop locks **phase** of f_{PLL} to f_{REF}

Expected output



Digital-to-Analog Converter

A Digital to Analog Converter (DAC) converts a digital input signal into an analog output signal.

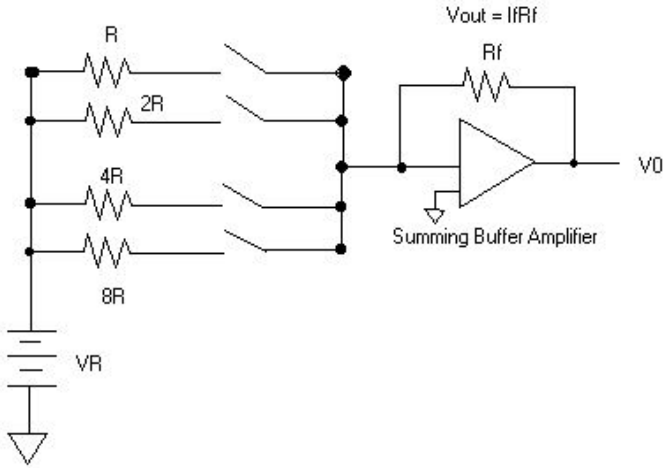
The digital signal is represented with a binary code, which is a combination of bits 0 and 1. A Digital to Analog Converter (DAC) consists of a number of binary inputs and a single output.

In general, the number of binary inputs of a DAC will be a power of two.

- There are two types of DACs :
  - Weighted Resistor DAC
  - R-2R Ladder DAC

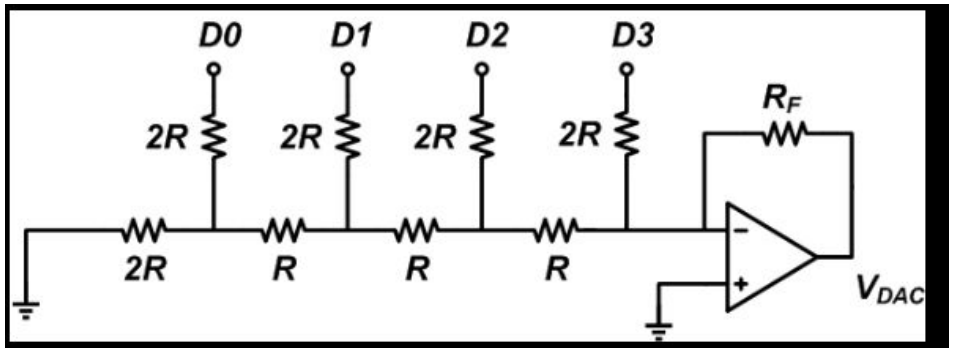
Weighted Resistor DAC

A weighted resistor DAC produces an analog output, which is almost equal to the digital (binary) input by using binary weighted resistors in the inverting adder circuit. In short, a binary weighted resistor DAC is called as weighted resistor DAC.

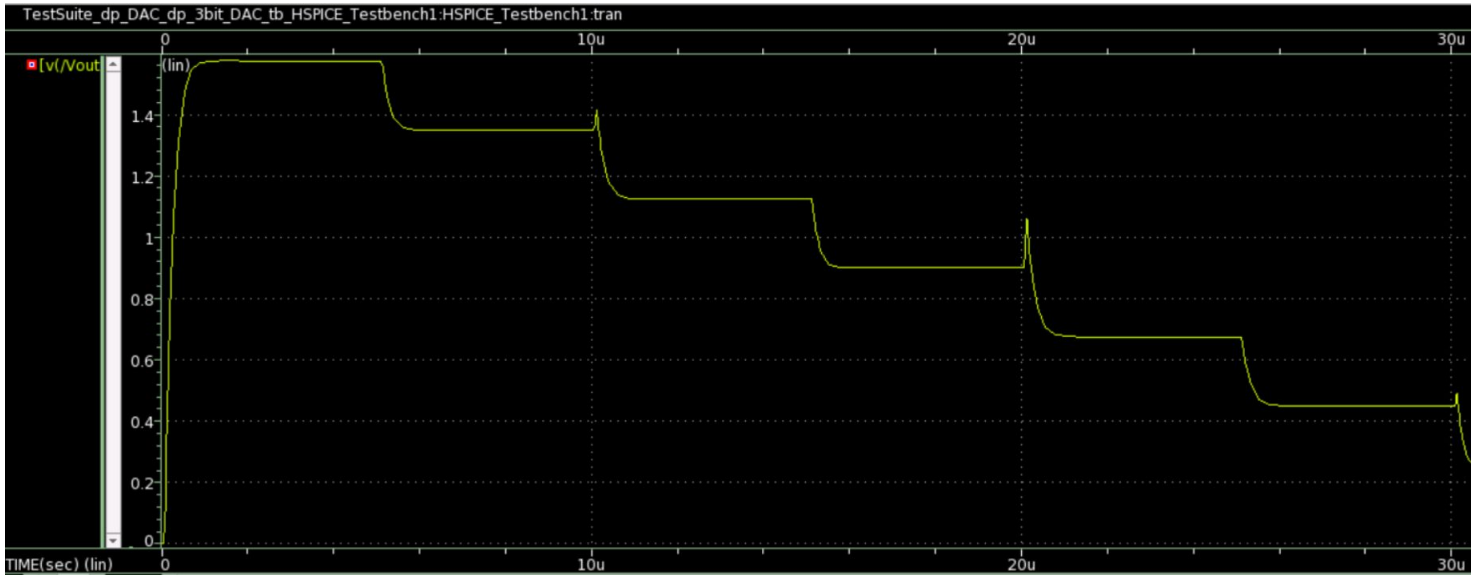


R-2R Ladder

The R-2R Ladder DAC **overcomes** the **disadvantages of a binary weighted resistor DAC**. As the name suggests, R-2R Ladder DAC produces an analog output, which is almost equal to the digital (binary) input by using a R-2R ladder network in the inverting adder circuit.



Expected output for 3-bit dac



For VSDBabySoC, it consists of a 10-Bit DAC

Let's start modelling

RVMYTH is a digital block, so yes we can use a HDL for designing and check its functionality using a testbench.

But! DAC and PLL are analog what to do? 😢

Because verilog can’t synthesis analog design

We are going to simulate it using verilog - we will be using data-types such real.

Our goal is to be able to simulate “functionality” - **to verify its logical correctness**

So we will be using **verilog** to model - and use **iverilog** to compile and simulate
Use **GTKWave** to see the waveforms & debug.

How do we model and simulate them?

We will be using **iverilog** - **Icarus Verilog** is an implementation of the Verilog hardware description language compiler that generates netlists in the desired format. It supports the 1995, 2001 and 2005 versions of the standard, portions of SystemVerilog, and some extensions.

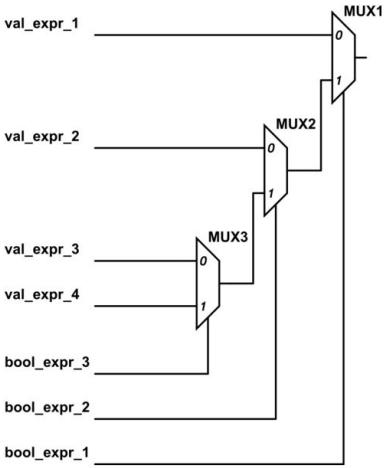
Modelling and simulating on iverilog involves 2 main steps, namely:

1. Compilation - iverilog builds the instance hierarchy and generates a binary executable a.out. This binary executable is later used for simulation.
2. Simulation - During compilation, iverilog generates a binary executable.

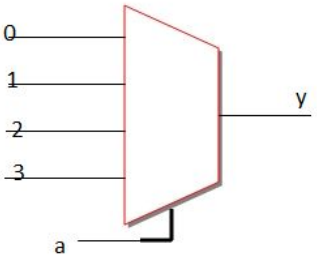
Few tips on modelling your design

1. Avoid race Conditions
2. Use a optimized Testbench for debugging your design
3. Creating models that simulate faster…
4. Case statement behaviour.

```
if(a==0) // assume a is a 3 bit signal
        Statement 4;
Else if(a==1)
        Statement 3;
Else if(a==2)
        statement 2 ;
…….
```




```
case(a)
0:
1 :
2 :
3:
Default :
```



Basic commands

1. iverilog design.v testbench.v (make sure to give relative path)
2. If too many files? - use +libdir+<directory of where the files are available>
3. If too many modules are there then use the -s flag.

For more commands refer the Manual.

Using GTKwave

GTKwave provides you with a graphical user interface to debug your design throw waveforms.

Now, first run the design, with a valid testbench using the command

- `iverilog design_file.v design_testbench.v`
- `./a.out`
  - This should run without any errors and also preferably without warnings.
- So one KEEP IN MIND point : if you remember in previous sessions, there was a introduction to *.vcd* file - means Value Change Dump.(cross check if its created or no).
- So in the testbench, always have this block
  - ```
    Initial
          Begin
          $dumpfile("design_tb.vcd");
          $dumpvars;
          end
    ```

NOTE : make sure to run these commands in your project directory/ design folder.
    ```

RVMYTH modelling

Following these steps for modelling RVMYTH

1. We have a RISC-V CPU core written in Verilog and an already written testbench code for the same.
2. The entire C program will be converted into a hex format and and will be loaded into memory.
3. The CPU will then read the contents of the memory, process it and finally display the output result of sum of numbers from 1 to n.

Follow these steps to model the IP cores separately

For modelling RVMYTH(RISC-V)

1. `git clone https://github.com/kunalg123/rvmyth/`
2. `cd rvmyth`
3. `csh`
4. `iverilog mythcore_test.v tb_mythcore_test.v`
5. `./a.out`
6. `gtkwave <file_name>.vcd &`
7. Add the required waveforms.

For modelling DAC

1. `git clone https://github.com/kunalg123/rvmyth/` `/// ignore if already done once!`
2. `cd rvmyth`
3. `csh`
4. `iverilog avsddac.v avsddac_tb_test.v`
5. `./a.out`
6. `gtkwave <file_name>.vcd &`
7. Add the required waveforms.

Follow the same for pll using design file as - `avsd_pll_1v8.v` and testbench- `pll_tb.v`

For simulating follow these steps

1. `git clone https://github.com/kunalg123/rvmyth/  // ignore if already done once`
2. `cd rvmyth`
3. `iverilog mythcore_test.v tb_mythcore_test.v`
4. `./a.out`
5. `gtkwave <file_name>.vcd`
6. `Add the required signals to the waveform.`

Task: understand the code and explore iverilog usage (either PLL/DAC/RISC-V -> as per your time).

Now, we have verified each block separately, lets interface blocks together

Lets simulate risc_v and pll

1. `cd rvmyth`
2. `iverilog rvmyth_pll.v rvmyth_pll_tb.v`
3. `./a.out`
4. `gtkwave <file_name>.vcd &.`
5. `Add the required signals to the waveform.`

Follow the same for DAC and RVMYTH using design file as - `rvmyth_avsddac.v` and testbench- `rvmyth_avsddac_TB.v`

Finally lets simulate and model all three IP’s together

Simulate https://github.com/manili/VSDBabySoC/blob/main/src/module/vsdbabysoc.v

Using iverilog with the testbench in https://github.com/manili/VSDBabySoC/blob/main/src/module/

There will be some errors you will be facing.
It's time to wear your debugging caps🎓 on!

Hint : mostly the error will be in ‘include files’ - that are available in https://github.com/manili/VSDBabySoC/tree/main/src/include &

https://github.com/manili/VSDBabySoC/blob/main/src/module/ it self, just have to go through the **simulation log** thoroughly.

For cross verification of your simulations (OR) how do we know which outputs to observe for

1. PLL/ RVMYTH - https://github.com/vsdip/rvmyth_avsdpll_interface
2. DAC/RVMYTH - https://github.com/vsdip/rvmyth_avsddac_interface
3. PLL interfacing RVMYTH - https://github.com/vsdip/rvmyth_avsdpll_interface
4. DAC interfacing RVMYTH - https://github.com/vsdip/rvmyth_avsddac_interface
5. vsdbabySoC - https://github.com/manili/VSDBabySoC/blob/main/images/pre_synth_sim.png

Tasks and to be added on to your repo.

1. To get more used to iverilog & GTKwave, each one take a example circuit for example
    a. 4- bit adder
    b. Counter
    c. 4x1 mux
    d. Decoder
    e. Encoder
    f. Any circuit of your choice -- PREFERRED

And simultate, attach snippets of code, screenshots of simulation, block diagrams etc on to the github repo.

2. After getting acquainted with iverilog & GTKwave, then get into modelling and simulation of VSDBabySoC.

Food for thought

References

NOTE : make sure to go through synopsys documentation for **VCS**.

1. Risc-v : https://myth3.makerchip.com/sandbox/#
2. Risc-v waterfall flow diagram : https://www.vlsisystemdesign.com/risc-v-waterfall-diagram-and-hazards/
3. Why PLL - https://www.mpi-inf.mpg.de/fileadmin/inf/d1/teaching/winter20/how_to_clock/lecture_13_PLLs_2020_12_14.pdf
4. https://github.com/manili/VSDBabySoC
5. https://github.com/Devipriya1921/VSDBabySoC_ICC2
6. https://github.com/steveicarus/iverilog#unsupported-constructs