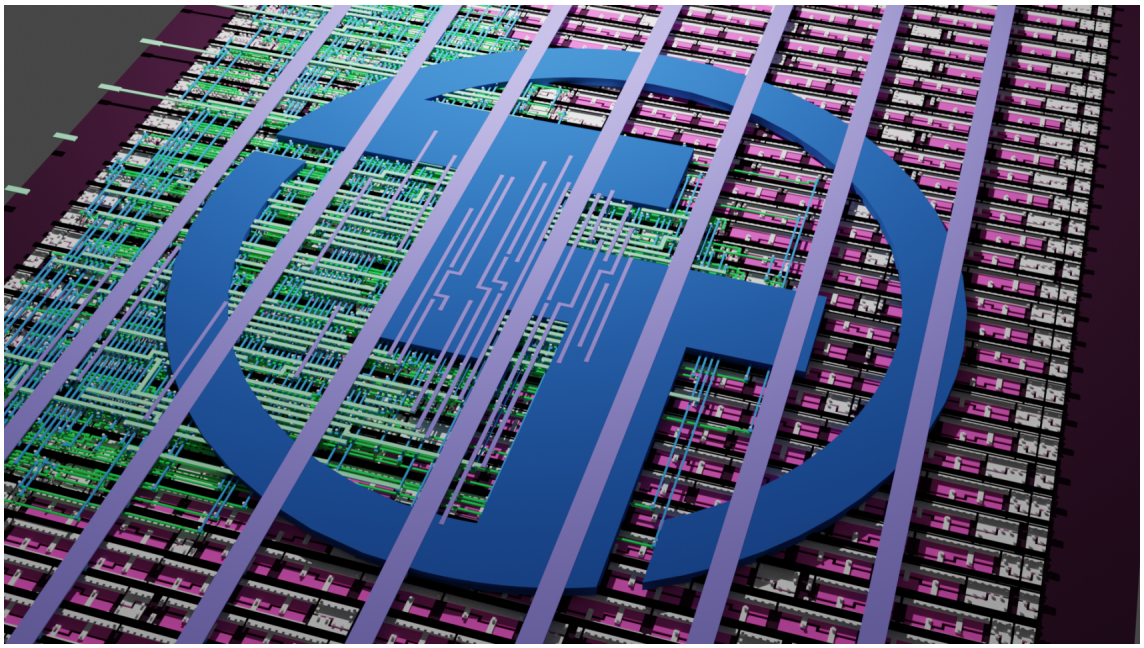

Tiny Tapeout 03 Datasheet

Project Repository

https://github.com/TinyTapeout/tinytapeout-03

June 2, 2023

Contents

Render of whole chip

Figure 1: Full GDS

Projects

0 : Test Inverter Project

- Author: Matt Venn
- Description: Inverts every line. This project is also used to fill any empty design spaces.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Uses 8 inverters to invert every line.

How to test

Setting the input switch to on should turn the corresponding LED off.

IO

#	Input	Output
0	a	segment a
1	b	segment b
2	c	segment c
3	d	segment d
4	e	segment e
5	f	segment f
6	g	segment g
7	dot	dot

1 : a and c or b

- Author: Joao Pedro Pedrosa
- Description: confirmation of a and c or b
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The gates works together to light a LED Explain how your project works

How to test

Just turn it on and flip the switches Explain how to test your project

IO

#	Input	Output
0	a	LED
1	b	none
2	c	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

2 : A AND B OR C

- Author: Carter Clay
- Description: LED will light up
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

IO

#	Input	Output
0	A	led
1	B	none
2	C	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

3 : B and C or A

- Author: ImmanuelZL
- Description: led lights up
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

IO

#	Input	Output
0	A	led
1	B	none
2	C	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

4 : A+!B+C

- Author: Charlize Hill
- Description: A circut that illuminates a led light
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

IO

#	Input	Output
0	A	none
1	B	none
2	C	LED light
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

5 : A xor b and c

- Author: Dino Danovic
- Description: Simple verification of a xor b and c via one LED
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works If the circuit is correct it lights up the LED light

How to test

Explain how to test your project Put it in a simulation on wokwi

IO

#	Input	Output
0	clock	xor
1	reset	and
2	none	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	dot

6 : Tiny tapeout

- Author: Jermaine
- Description: when a and b or a and b to find what it equals
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

IO

#	Input	Output
0	a	none
1	b	led
2	none	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

7 : tiny tape out not a and not b

- Author: daniel joseph
- Description: turn s led light on
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

Explain how to test your project

IO

#	Input	Output
0	not a	led light
1	not b	none
2	none	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

8 : Tiny Tapeout 3 Template Jesse

- Author: Jesse Mugisha
- Description: A not B and C
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Explain how your project works

How to test

1 on, 2 off, Explain how to test your project

IO

#	Input	Output
0	and	and
1	not	not
2	and	and
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none11132

9 : Not A Or B

- Author: Laura Holmes
- Description: Simple design of Not A or B
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

Input 1 is A, Input 2 is B Explain how your project works

How to test

Explain how to test your project

IO

#	Input	Output
0	A	LED
1	B	segment b
2	none	segment c
3	none	segment d
4	none	segment e
5	none	segment f
6	none	segment g
7	none	dot

10 : Simple Elevator Circuit

- Author: Dan Fiumara
- Description: This circuit represents a four floor elevator via two LEDs.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The input switch determines whether the elevator will move up or down a floor. The input button will make the change defined by the switch.

How to test

The two LEDs are the current floor number in binary. If both LEDs are off, flip the switch to high and press the button a few times. If both LEDs are on, flip the switch to low and press the button a few times. The button will need to be debounced.

IO

#	Input	Output
0	Up/Down lever	Current Floor bit 1
1	Go button	Current Floor bit 2
2	none	none
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

11 : Traffic Light Circuit

- Author: Redemption Christian Academy
- Description: This circuit simulates a Traffic Light by cycling through Green, Yellow, and Red lights.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 10000 Hz
- External hardware:

How it works

A 10KHz clock signal gets stepped down by a series of toggle flip flops until it is roughly the speed necessary for a traffic light cycle. A state machine then uses the clock signal to cycle through three states: Green, Yellow, and Red.

How to test

So long as the circuit is powered and the green, yellow, and red leds are connected, you should see the traffic light cycle.

IO

#	Input	Output
0	clock	Green
1	none	Yellow
2	none	Red
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

12 : 4 Bit Lock Circuit

- Author: Redemption Christian Academy
- Description: The circuit represents a lock with a 4 bit combination.
- GitHub repository
- Wokwi project
- Extra docs
- Clock: 0 Hz
- External hardware:

How it works

The circuit is a state machine that takes 4 bits as input and determines whether or not the lock should open based on the input string.

How to test

Click the submit button until the Blue LED is illuminated. Switch the first input to the bit you would like to submit. Press the submit button. Repeat three more times. When you are finished, if the red LED is illuminated, the circuit is locked. If the green LED is illuminated, the circuit is unlocked.

IO

#	Input	Output
0	Input Bit	Red LED (Lock is locked)
1	Submit button	Green LED (lock is unlocked)
2	none	Blue LED (No bits have been input)
3	none	none
4	none	none
5	none	none
6	none	none
7	none	none

Technical info

Scan chain

All 250 designs are joined together in a long chain similiar to JTAG. We provide the inputs and outputs of that chain (see pinout below) externally, to the Caravel logic analyser, and to an internal scan chain driver.

The default is to use an external driver, this is in case anything goes wrong with the Caravel logic analyser or the internal driver.

The scan chain is identical for each little project, and you can read it here.

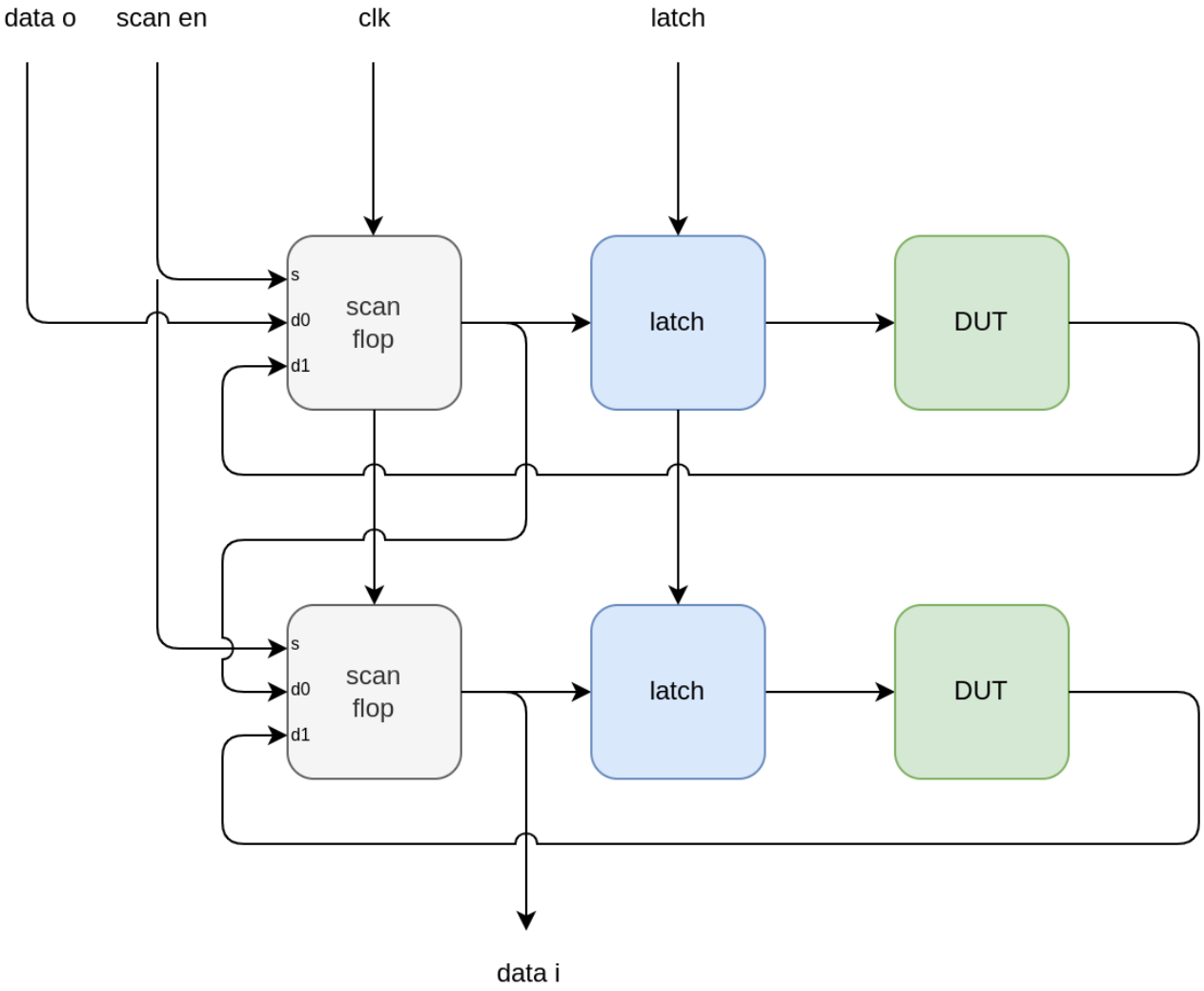

Figure 2: block diagram

Scan chain changes since TT02

After doing more simulation, we found that the 500 buffers in the scanchain were distorting the clock pulse to the point where we needed to further slow the clock. For

more information watch this video.

Sylvain 'tnt' Munaut suggested a change to the scanchain that prevents the distortion and reduces the number of buffers:

The main change to the `scanchain` block is related to the clocking and the data output. Previously, you'd have something like this :

```
clk_in ---|>----|>---- clk_out
              |
              '------- Internal FFs
```

That is, you had an input buffer receving clock from the previous block and sending it to all the scan chain FF and to a strong output buffer that would send it to the next block.

The issues that were identified there are :

- 2 buffers per block means 500 buffers in the whole chain and the more stages, the more delay.
- The buffers don't have symmetrical rise/fall time. So each buffer would slightly alter the pulse width. Multiply that by 500 buffers and the shape at the end of the scan chain would be nowhere close to what was fed in.

The new block is like this :

```
clk_in ---|>o--------- clk_out
              |
              '-|>o--- Internal FFs
```

You have a single inverter that receives the clock and sends it inverted to the next block. That inverted clock is also fed to another small inverter whose output will be used to clock all the internal effects.

The result is that we have half as many stages in the chain of inverter. And also because the clock is inverted at each stage, any change in duty cycle that one stage causes on the clock pulses is "undone" by the next. It's not perfect of course since they are not perfectly match, but it's MUCH MUCH better.

Of course this required some other changes to deal with the fact the clock is inverted at the output.

The first change is in scanchain block itself. Previously we'd have a negative edge FF at the output so that the output clock rising edge would happen in the middle of the output data valid window. Now because the clock is inverted, this has the same effect by itself. We still need a FF though (else the last bit of each block would always be

reflected to the first bit of the next), but now it needs to be a rising edge FF.

The second change is in the scan controller since the drive waveform needs to be slightly different. Every two stages, a clock pulse is "lost" effectively and so an extra pulse must be injected to keep moving the data long.

Updating inputs and outputs of a specified design

A good way to see how this works is to read the FSM in the scan controller. You can also run one of the simple tests and check the waveforms. See how in the scan chain verification doc.

- Signal names are from the perspective of the scan chain driver.
- The desired project shall be called DUT (design under test)

Assuming you want to update DUT at position 2 (0 indexed) with inputs = 0x02 and then fetch the output. This design connects an inverter between each input and output.

- Set scan_select low so that the data is clocked into the scan flops (rather than from the design)
- For the next 8 clocks, set scan_data_out to 0, 0, 0, 0, 0, 0, 1, 0
- Toggle scan_clk_out 17 times to deliver the data to the DUT (new scanchain needs 1 extra clock for every odd numbered project in the chain)
- Toggle scan_latch_en to deliver the data from the scan chain to the DUT
- Set scan_select high to set the scan flop's input to be from the DUT
- Toggle the scan_clk_out to capture the DUT's data into the scan chain
- Toggle the scan_clk_out another 8.5 x number of remaining designs to receive the data at scan_data_in

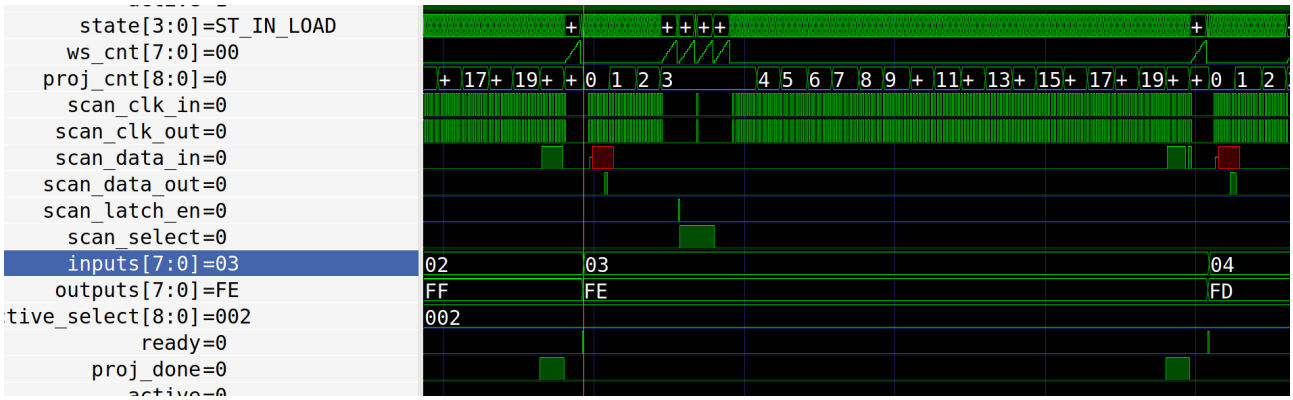

Figure 3: update cycle

Notes on understanding the trace

- There are large wait times between the latch and scan signals to ensure no hold violations across the whole chain. For the internal scan controller, these can be configured (see section on wait states below).
- The input looks wrong (0x03) because the input is incremented by the test bench as soon as the scan controller captures the data. The input is actually 0x02.
- The output in the trace looks wrong (0xFE) because it's updated after a full refresh, the output is 0xFD.
- Generate the trace yourself by running `make test_single` in `verilog/dv/scan_contr` See VERIFICATION.md for more info on running tests.

Clocking

Assuming:

- 100MHz input clock
- 8 ins & 8 outs
- 2 clock cycles to push one bit through the scan chain (scan clock is half input clock rate)
- 250 designs
- scan controller can do a read/write cycle in one refresh

So the max refresh rate is 100MHz / (8 * 2 * 250) = 25000Hz.

Clock divider

A rising edge on the set_clk_div input will capture what is set on the input pins and use this as a divider for an internal slow clock that can be provided to the first input bit.

The slow clock is only enabled if the set_clk_div is set, and the resulting clock is connected to input0 and also output on the slow_clk pin.

The slow clock is synced with the scan rate. A divider of 0 mean it toggles the input0 every scan. Divider of 1 toggles it every 2 cycles. So the resultant slow clock frequency is scan_rate / (2 * (N+1)).

See the test_clock_div test in the scan chain verification.

Wait states

This dictates how many wait cycle we insert in various state of the load process. We have a sane default, but also allow override externally.

To override, set the wait amount on the inputs, set the driver_sel inputs both high, and then reset the chip.

See the test_wait_state test in the scan chain verification.

Pinout

```
PIN     NAME                DESCRIPTION
20:12   active_select       9 bit input to set which design is active
28:21   inputs              8 inputs
36:29   outputs             8 outputs
37      ready               goes high for one cycle everytime the scancha
10      slow_clk            slow clock from internal clock divider
11      set_clk_div         enable clock divider
9:8     driver_sel          which scan chain driver: 00 = external, 01 =

21      ext_scan_clk_out    for external driver, clk input
22      ext_scan_data_out   data input
23      ext_scan_select     scan select
24      ext_scan_latch_en   latch
29      ext_scan_clk_in     clk output from end of chain
30      ext_scan_data_in    data output from end of chain
```

Instructions to build GDS

To run the tool locally or have a fork's GitHub action work, you need the GH_USERNAME and GH_TOKEN set in your environment.

GH_USERNAME should be set to your GitHub username.

To generate your GH_TOKEN go to https://github.com/settings/tokens/new . Set the checkboxes for repo and workflow.

To run locally, make a file like this:

```
export GH_USERNAME=<username>
export GH_TOKEN=<token>
```

And then source it before running the tool.

Fetch all the projects

This goes through all the projects in project_urls.py, and fetches the latest artifact zip from GitHub. It takes the verilog, the GL verilog, and the GDS and copies them to

the correct place.

```
./configure.py --clone-all --fetch-gds
```

Configure Caravel

Caravel needs the list of macros, how power is connected, instantiation of all the projects etc. This command builds these configs and also makes the README.md index.

```
./configure.py --update-caravel
```

Build the GDS

To build the GDS and run the simulations, you will need to install the Sky130 PDK and OpenLane tool. It takes about 5 minutes and needs about 3GB of disk space.

```
export PDK_ROOT=<some dir>/pdk
export OPENLANE_ROOT=<some dir>/openlane
cd <the root of this repo>
make setup
```

Then to create the GDS:

```
make user_project_wrapper
```

Changing macro block size

After working out what size you want:

- adjust configure.py in `CaravelConfig.create_macro_config()`.
- adjust the PDN spacing to match in openlane/user_project_wrapper/config.tcl:
  - `set ::env(FP_PDN_HPITCH)`
  - `set ::env(FP_PDN_HOFFSET)`

Verification

We are not trying to verify every single design. That is up to the person who makes it. What we want is to ensure that every design is accessible, even if some designs are broken.

We can split the verification effort into functional testing (simulation), static tests (formal verification), timing tests (STA) and physical tests (LVS & DRC).

See the sections below for details on each type of verification.

Setup

You will need the GitHub tokens setup as described in INFO.

The default of 250 projects takes a very long time to simulate, so I advise overriding the configuration:

```
# fetch the test projects
./configure.py --test --clone-all
# rebuild config with only 20 projects
./configure.py --test --update-caravel --limit 20
```

You will also need iVerilog & cocotb. The easist way to install these are to download and install the oss-cad-suite.

Simulations

- Simulation of some test projects at RTL and GL level.
- Simulation of the whole chip with scan controller, external controller, logic analyser.
- Check wait state setting.
- Check clock divider setting.

Scan controller

This test only instantiates user_project_wrapper (which contains all the small projects). It doesn't simulate the rest of the ASIC.

```
cd verilog/dv/scan_controller
make test_scan_controller
```

The Gate Level simulation requires scan_controller and user_project_wrapper to be re-hardened to get the correct gate level netlists:

- Edit openlane/scan_controller/config.tcl and change NUM_DESIGNS=250 to NUM_DESIGNS=20.
- Then from the top level directory:

  make scan_controller make user_project_wrapper
- Then run the GL test

  cd verilog/dv/scan_controller make test_scan_controller_gl

single

Just check one inverter module. Mainly for easy understanding of the traces.

```
make test_single
```

custom wait state

Just check one inverter module. Set a custom wait state value.

```
make test_wait_state
```

clock divider

Test one inverter module with an automatically generated clock on input 0. Sets the clock rate to 1/2 of the scan refresh rate.

```
make test_clock_div
```

Top level tests setup

For all the top level tests, you will also need a RISCV compiler to build the firmware.

You will also need to install the 'management core' for the Caravel ASIC submission wrapper. This is done automatically by following the PDK install instructions.

Top level test: internal control

Uses the scan controller, instantiated inside the whole chip.

```
cd verilog/dv/scan_controller_int
make coco_test
```

Top level test: external control

Uses external signals to control the scan chain. Simulates the whole chip.

```
cd verilog/dv/scan_controller_ext
make coco_test
```

Top level test: logic analyser control

Uses the RISCV co-processor to drive the scanchain with firmware. Simulates the whole chip.

```
cd verilog/dv/scan_controller_la
make coco_test
```

Formal Verification

- Formal verification that each small project's scan chain is correct.
- Formal verification that the correct signals are passed through for the 3 different scan chain control modes.

Scan chain

Each GL netlist for each small project is proven to be equivalent to the reference scan chain implementation. The verification is done on the GL netlist, so an RTL version of the cells used needed to be created. See here for more info.

Scan controller MUX

In case the internal scan controller doesn't work, we also have ability to control the chain from external pins or the Caravel Logic Analyser. We implement a simple MUX to achieve this and formally prove it is correct.

Timing constraints

Due to limitations in OpenLane - a top level timing analyis is not possible. This would allow us to detect setup and hold violations in the scan chain.

Instead, we design the chain and the timing constraints for each project and the scan controller with this in mind.

- Each small project has a negedge flop flop at the end of the shift register to reclock the data. This gives more hold margin.

- Each small project has SDC timing constraints
- Scan controller uses a shift register clocked with the end of the chain to ensure correct data is captured.
- Scan controller has its own SDC timing constraints
- Scan controller can be configured to wait for a programmable time at latching data into the design and capturing it from the design.
- External pins (by default) control the scan chain.

Physical tests

- LVS
- DRC
- CVC

LVS

Each project is built with OpenLane, which will check LVS for each small project. Then when we combine all the projects together we run a top level LVS & DRC for routing, power supply and macro placement.

The extracted netlist from the GDS is what is used in the formal scan chain proof.

DRC

DRC is checked by OpenLane for each small project, and then again at the top level when we combine all the projects.

CVC

Mitch Bailey' CVC checker is a device level static verification system for quickly and easily detecting common circuit errors in CDL (Circuit Definition Language) netlists. We ran the test on the final design and found no errors.

- See the paper here.
- Github repo for the tool: https://github.com/d-m-bailey/cvc

Toplevel STA and Spice analysis for TinyTapeout-02 and 03.

Contributed by J. Birch 22 March 2023

Introduction

TinyTapeout is built using the OpenLane flow and consists of three major components:

1) up to 250 user designs with a common interface, each of which has been composed using the OpenLane flow,
2) a scanchain block that is instanced for each user design to provide input data and to sink output data,
3) a scan chain controller.

The OpenLane flow allows for timing analysis of each of these individual blocks but does not provide for analysis of the assembly of them, which leaves potential holes in timing that could cause failures.

This work allows a single design to be assembled out of the sub-blocks and this can then be submitted to STA for analysis.

In addition, an example critical path (for the clock that is passed along the scanchain) is created using a Python script so that it can be run through SPICE.

GitHub action

The STA github action automatically runs the STA when the repository is updated.

Prerequisites

To run STA, the following files need to be available:

1) a gate level verilog netlist of the top level design (verilog/rtl/user_project_wrapper.v)
2) a gate level verilog netlist for each sub-block (verilog/rtl/)
3) a SPEF format parasitics file for the top level wiring (spef/)
4) a SPEF format parasitics file for each sub-block (spef/)
5) the relevant SkyWater libraries for STA analysis - installed with the PDK
6) OpenSTA 2.4.0 (or later) needs to be on the PATH. rundocker.sh shows how to use STA included in the OpenLane docker
7) python3.9 or later
8) verilog parser (pip3 install verilog-parser)

9) a constraints file (sta_top/top.sdc)

Assumptions

- all of the Verilog files are in a single directory
- all of the SPEF files are in a single directory

Issues

The Verilog parser has two bugs: it does not recognise 'inout' and it does not cope with escaped names (starting with a \). Locally this has been fixed but to use the off-the-shelf parser we preprocess the file to change inout to input and remove the escapes and substitute the [nnn] with *nnn* in the names

Invoking STA analysis

```
sta_top/toplevel_sta.py <path to verilog> <path to spef> <sdc>
```

eg:

```
sta_top/toplevel_sta.py ./verilog/gl/user_project_wrapper.v
./spef/user_project_wrapper.spef ./sta_top/top.sdc > sta.log
```

Alternatively you can enter interactive mode after analysis:

```
sta_top/toplevel_sta.py ./verilog/gl/user_project_wrapper.v
./spef/user_project_wrapper.spef ./sta_top/top.sdc -i
```

How it works

- preprocesses the main Verilog
- parses the main Verilog to find which modules are used
- creates a new merged Verilog containing all the module netlists and the main netlist
- creates a tcl script in the spef directory to load the relevant spefs for each instanced module in the main verilog
- creates all_sta.tcl in the verilog/gl directory that does the main STA steps (loading design, loading constraints, running analyses) - this mimics what OpenLane does
- creates a script sta.sh in the verilog/gl directory which sets up the environment and invokes STA to run all_sta.tcl
- runs sta.sh

Invoking Spice simulation:

Important notes

- This uses at least ngspice-34, tested on ngspice-39, which has features to increase speed and reduce memory image when running with the SkyWater spice models. In addition the following needs to be set in ~/spice.rc and/or ~/.spiceinit:

  set ngbehavior=hs

- Note the spelling of ngbehavior (no u). If this is not set then the simulation will run out of memory]
- If the critical path changes then the spice_path.py script will need to be adapted to follow suit

Instructions

- go to where the included spice files will be found

  cd $PDK_ROOT/sky130A/libs.tech/ngspice

- run the script to get the critical path spice circuit - 250 stages of the clock

  /sta_top/spice_path.py path.spice

- invoke spice (takes about 2 minutes)

  ngspice path.spice

- run the sim (takes about 2 minutes)

  tran 1n 70n

- show the results

  plot i0 i250

This chart shows the input clock and how it changes after 250 blocks. It was simulated for 200n to capture a clean pair of clock pulses.

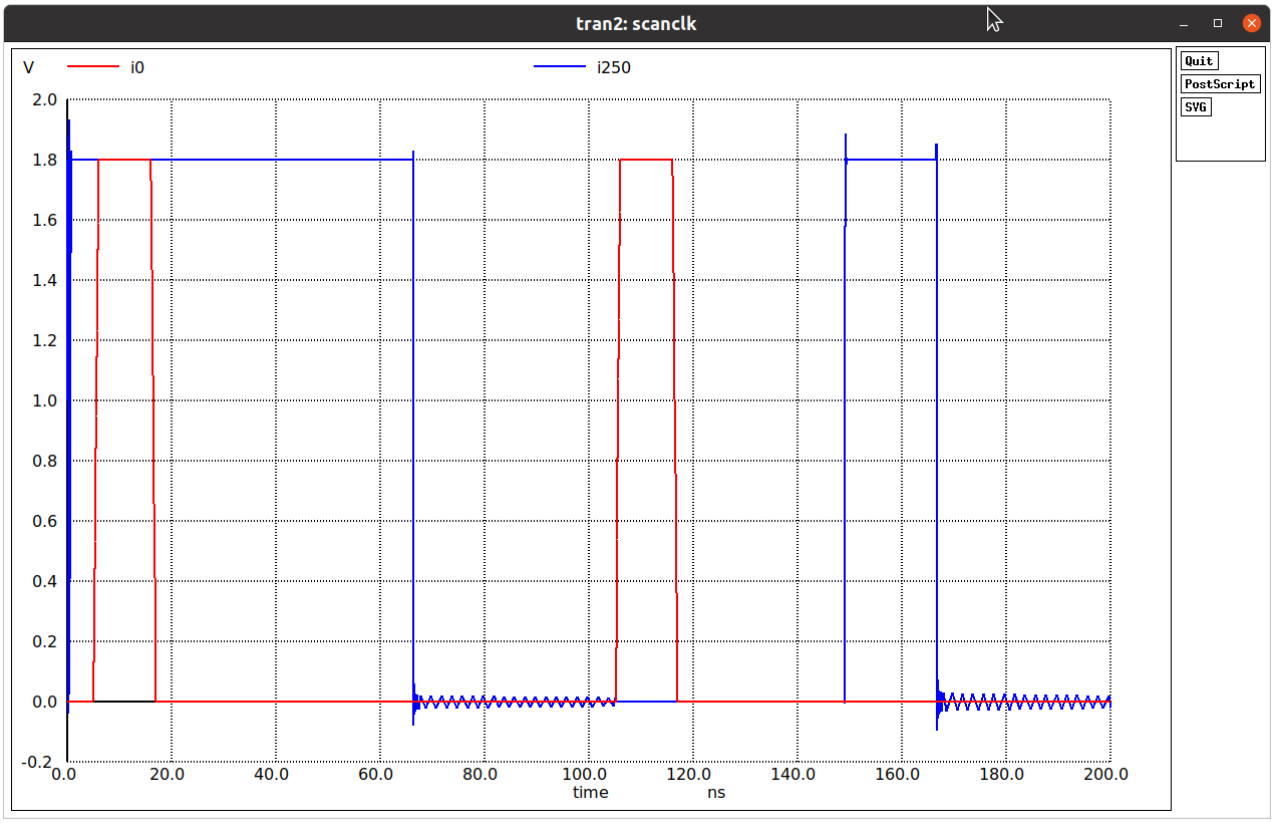

Figure 4: clock_spice.png

Sponsored by

Team

Tiny Tapeout would not be possible without a lot of people helping. We would especially like to thank:

- Uri Shaked for wokwi development and lots more
- Sylvain Munaut for help with scan chain improvements
- Mike Thompson for verification expertise
- Jix for formal verification support
- Proppy for help with GitHub actions
- Maximo Balestrini for all the amazing renders and the interactive GDS viewer
- James Rosenthal for coming up with digital design examples
- All the people who took part in TinyTapeout 01 and volunteered time to improve docs and test the flow
- The team at YosysHQ and all the other open source EDA tool makers
- Efabless for running the shuttles and providing OpenLane and sponsorship
- Tim Ansell and Google for supporting the open source silicon movement
- Zero to ASIC course community for all your support
- Jeremy Birch for help with STA
- Aisler for sponsoring PCB development