Tiny Tapeout GF 0p3 Datasheet

github.com/TinyTapeout/tinytapeout-gf-0p3

July 3, 2026

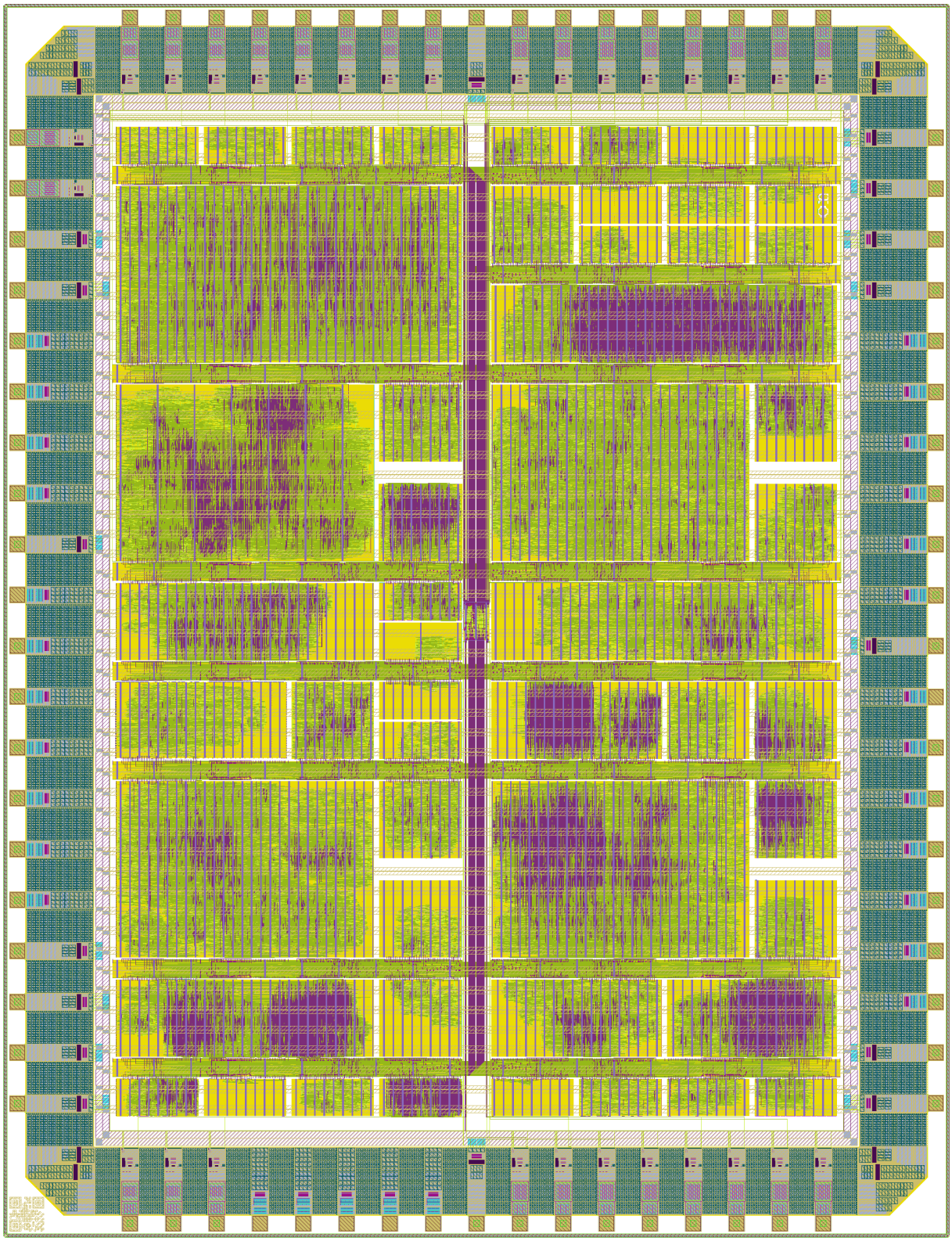
Table of Contents

Chip Renders

Online chip viewer

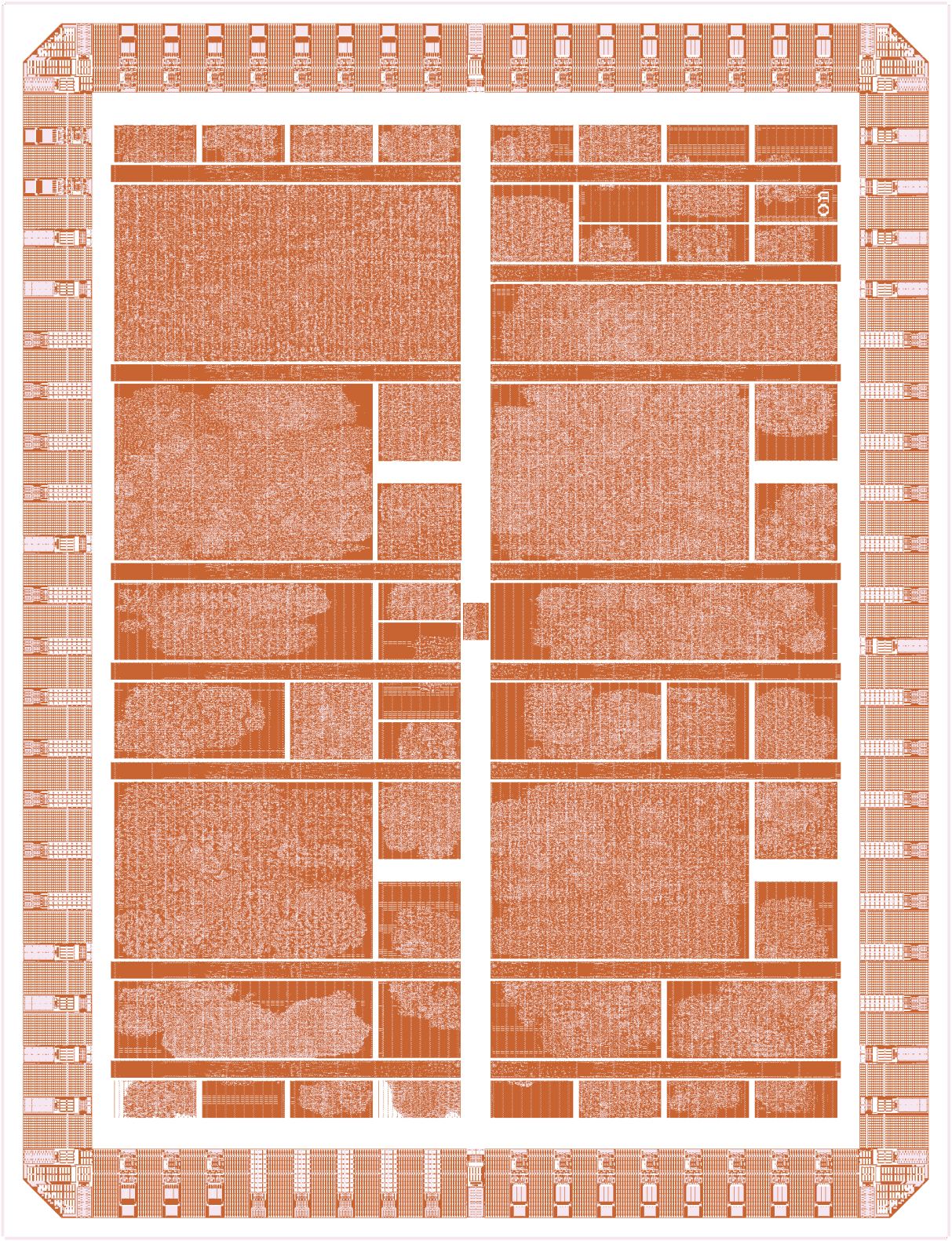
tinytapeout.com/
decap/ttgf0p3

GDS



Logic Density

Local Interconnect Layer



Projects

Chip ROM

by **Uri Shaked**

0000 HDL Project

`github.com/TinyTapeout/tt-chip-rom`

"ROM with information about the chip"

How it works

ROM memory that contains information about the Tiny Tapeout chip. The ROM is 8-bit wide and 256 bytes long.

The ROM layout

The ROM layout is as follows:

Address	Length	Encoding	Description
0	8	7-segment	Shuttle name (e.g. "tt07"), null-padded
8	8	7-segment	Git commit hash
32	96	ASCII	Chip descriptor (see below)
248	4	binary	Magic value: `"TT\xFA\xBB"`
252	4	binary	CRC32 of the ROM contents, little-endian

The chip descriptor

The chip descriptor is a simple null-terminated string that describes the chip. Each line is a key-value pair, separated by an equals sign. It contains the following keys:

Key	Description	Example value
shuttle	The identifier of the shuttle	tt07
repo	The name of the repository	TinyTapeout/tinytapeout-07
commit	The commit hash *	a1b2c3d4

* The commit hash is only included for Tiny Tapeout 5 and later.

Here is a complete example of a chip descriptor:

```
shuttle=tt07
repo=TinyTapeout/tinytapeout-07
commit=a1b2c3d4
```

How the ROM is generated

The ROM is automatically generated by tt-support-tools while building the final GDS file of the chip. Look at the `rom.py` file in the repository for more details.

Reading the ROM

There are two ways to address ROM, depending on the value of the `rst_n` pin:

1. When `rst_n` is high: Set the `ui_in` pins to the desired address.
2. When `rst_n` is low: Toggle the `clk` pin to read the ROM contents sequentially, starting from address 0.

In both cases, the ROM data for the selected address will be available on the `uo_out` pins, one byte at a time.

How to test

The first 16 bytes of the ROM are 7-segment encoded and contain the shuttle name and commit hash. You can dump them by holding `rst_n` low and toggling the `clk` pin, and observing the on-board 7-segment display.

Alternatively, you can keep `rst_n` high and set the `ui_in` pins to the desired address using the first four on-board DIP switches, while observing the on-board 7-segment display.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	addr[0]	data[0]	—
1	addr[1]	data[1]	—
2	addr[2]	data[2]	—
3	addr[3]	data[3]	—
4	addr[4]	data[4]	—
5	addr[5]	data[5]	—
6	addr[6]	data[6]	—
7	addr[7]	data[7]	—

Tiny Tapeout Factory Test

by **Tiny Tapeout**

0001 HDL Project

github.com/TinyTapeout/ttgf26a-factory-test

"*Factory test module*"

How it works

The factory test module is a simple module that can be used to test all the I/O pins of the ASIC.

It has three modes of operation:

1. Mirroring the input pins to the output pins (when `rst_n` is low).
2. Mirroring the bidirectional pins to the output pins (when `rst_n` is high `sel` is low).
3. Outputing a counter on the output pins and the bidirectional pins (when `rst_n` is high and `sel` is high).

The following table summarizes the modes:

rst_n	sel	Mode	uo_out value	uio pins
0	X	Input mirror	ui_in	High-Z
1	0	Bidirectional mirror	uio_in	High-Z
1	1	Counter	counter	counter

The counter is an 8-bit counter that increments on every clock cycle, and resets when `rst_n` is low.

How to test

1. Set `rst_n` low and observe that the input pins (`ui_in`) are output on the output pins (`uo_out`).
2. Set `rst_n` high and `sel` low and observe that the bidirectional pins (`uio_in`) are output on the output pins (`uo_out`).
3. Set `sel` high and observe that the counter is output on both the output pins (`uo_out`) and the bidirectional pins (`uio`).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sel / in_a[0]	output[0] / counter[0]	in_b[0] / counter[0]
1	in_a[1]	output[1] / counter[1]	in_b[1] / counter[1]
2	in_a[2]	output[2] / counter[2]	in_b[2] / counter[2]
3	in_a[3]	output[3] / counter[3]	in_b[3] / counter[3]
4	in_a[4]	output[4] / counter[4]	in_b[4] / counter[4]
5	in_a[5]	output[5] / counter[5]	in_b[5] / counter[5]
6	in_a[6]	output[6] / counter[6]	in_b[6] / counter[6]
7	in_a[7]	output[7] / counter[7]	in_b[7] / counter[7]

VGA Nyan Cat

by **Andy Sloane**

0003 | 25.175 MHz | HDL Project

`github.com/urish/ttgf0p2-a1k0n-nyan`

"Displays the classic nyan.cat animation"

VGA nyan cat

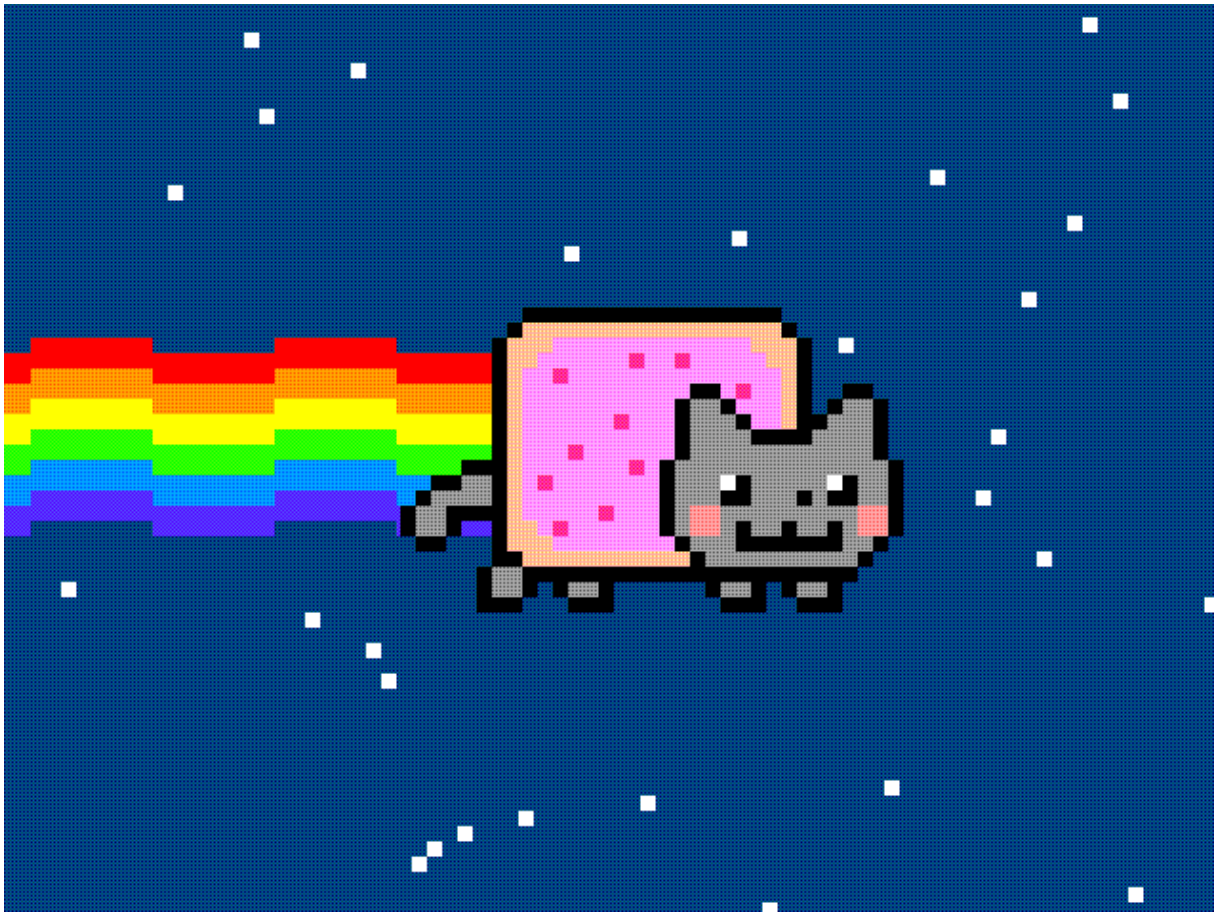


Figure 3.1: nyancat preview

How it works

Outputs nyancat on VGA with music!

Colors and animation are all from the original nyan.cat site, using a 2x2 Bayer dithering matrix which inverts on alternate frames for better color rendition on the Tiny VGA Pmod.

Sound is generated from a MIDI file, split into melody and bass parts. Melody and bass are each square waves mixed with a simple exponential decay envelope, which is then fed to a low-pass filter and then a sigma-delta DAC.

This was designed to fit into 1 tile, and it *almost* did – the cells take up about 93% of 1 tile, but detailed routing doesn't finish. With the deadline approaching I was forced to grow it to 1x2, so I threw in a little easter egg.

How to test

Set clock to 25.175MHz or thereabouts, give reset pulse, and enjoy

External hardware

TinyVGA Pmod for video on o[7:0].

1-bit sound on io[7], compatible with Tiny Tapeout Audio Pmod, or any basic 20kHz RC filter on io7 to an amplifier will work.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	R1	—
1	—	G1	—
2	—	B1	—
3	—	VSync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	—	HSync	AudioPWM

MPW-6 "Hack SoC" – Illustrated by Máximo Balestrini.

74HCT00 Quad 2-Input NAND (3.3 V)

by **Vipul Sharma**

0005 Analog Project

`github.com/engrvip123/tt_um_74hct00`

"Open-source silicon realization of 74HCT00 TTL-compatible quad 2-input NAND on gf180mcuD at 3.3 V"

How it works

This project is an open-source silicon implementation of **quad 2-input NAND** on the GlobalFoundries **gf180mcuD** process, operating from a **3.3 V** supply. Each NAND gate presents a TTL-compatible input window (VIL,max = 0.8 V, VIH,min = 2.0 V) and is sized for **±4 mA** drive at the output.

The HCT input behaviour is realised through a skewed-inverter / NOR2 / output-buffer chain rather than a plain CMOS NAND topology:

```
nand2\_hct  =  inv\_skewed (A) ┐
               inv\_skewed (B) ┴─►  nor2  ──►  inv\_out  ──►  Y
```

The four NAND gates share a common VDD/VSS rail pair. Each gate output is routed to an analog pin in `ua\[0..3]` rather than to the digital pins, allowing VOH and VOL to be measured at the rated ±4 mA drive without the digital output buffer of the TT pad frame in series.

Gate 1's A input (A1) is brought out exclusively on the analog pin `ua\[4]` rather than on a digital `ui\_in` pin. This bypasses the TT digital pad buffer entirely and is used for two purposes:

(a) functional drive at 0 V or 3.3 V for the truth-table test, and

(b) a slow DC ramp from 0 V to 3.3 V to characterise the HCT input window on silicon. A digital `ui\_in` pin would otherwise snap any intermediate voltage to a clean 0 V or 3.3 V at the TT pad buffer's own CMOS threshold before it ever reached the gate, making the HCT-window measurement impossible.

Schematics

The design was captured in **xschem** and simulated in **ngspice**. Per-cell LVS was performed in KLayout.

Chip-level testbench (`tb\_74hct00\_top.sch`):

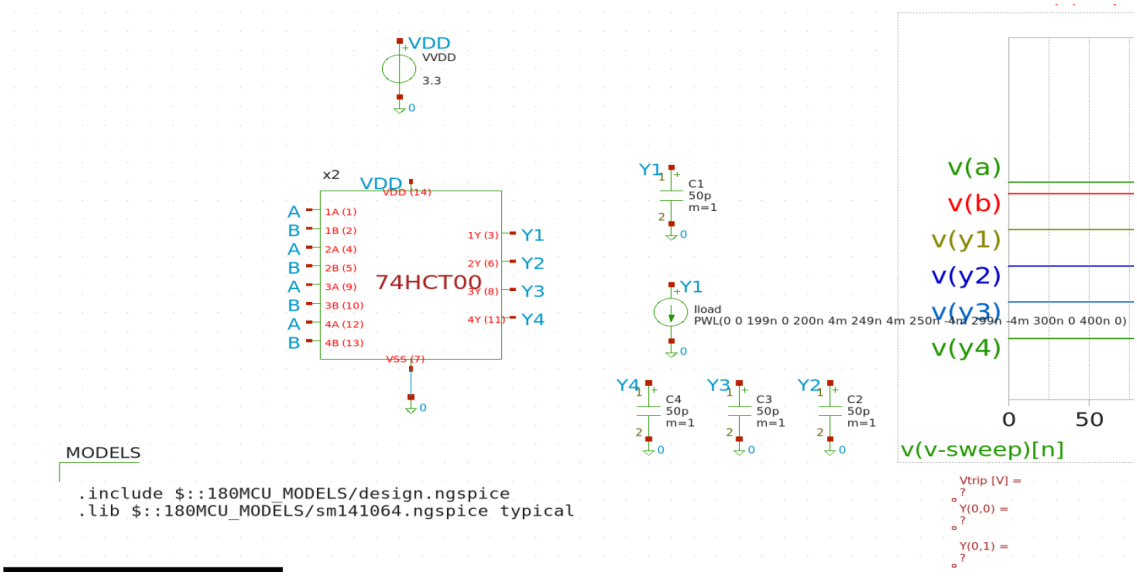


Figure 5.1: Chip-level testbench

Top-level schematic (`74hct00\_top.sch`):

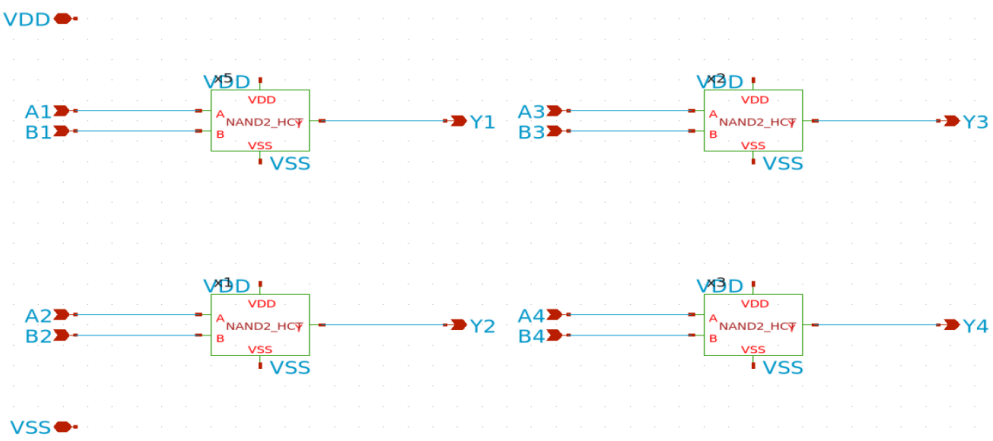


Figure 5.2: Chip top

NAND gate (`nand2\_hct.sch`):

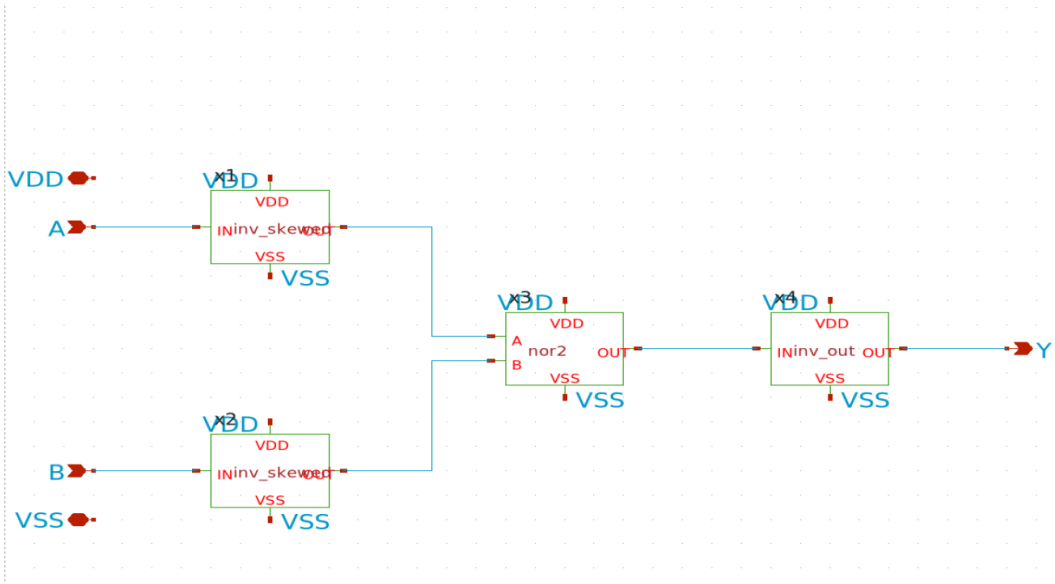


Figure 5.3: NAND gate

Interior NOR2 (`nor2.sch`):

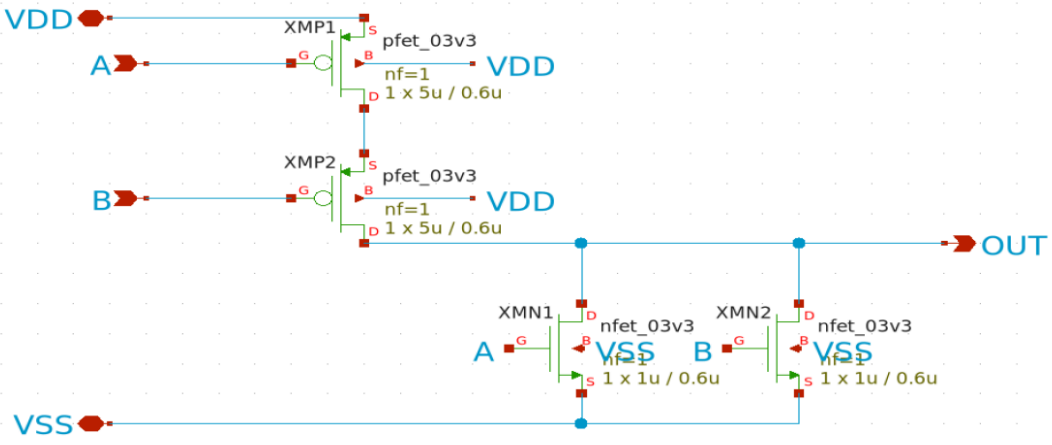


Figure 5.4: Interior NOR2

Pinout

Tile input pins mapped to NAND gate inputs:

Tile pin	Cell pin
`ua\[4]`	A1 (Gate 1 input A; truth-table drive + HCT-window DC sweep)
`ui\_in\[1]`	B1 (Gate 1 input B)
`ui\_in\[2]`	A2 (Gate 2 input A)

ui_in\[3]	B2 (Gate 2 input B)
ui_in\[4]	A3 (Gate 3 input A)
ui_in\[5]	B3 (Gate 3 input B)
ui_in\[6]	A4 (Gate 4 input A)
ui_in\[7]	B4 (Gate 4 input B)
ui_in\[0]	unused

Tile output pins mapped to NAND gate outputs:

Tile pin	Cell pin
ua\[0]	Y1 (Gate 1 output)
ua\[1]	Y2 (Gate 2 output)
ua\[2]	Y3 (Gate 3 output)
ua\[3]	Y4 (Gate 4 output)

Power pins: VDPWR is the 3.3 V supply, VGND is ground.

How to test

The procedure below is a basic functional verification of all four NAND gates.

1. Apply **3.3 V** between VDPWR and VGND.
2. For each gate *i*, drive the two input pins to either 0 V or 3.3 V and read the corresponding output Yi on ua\[i−1]:
   - Gate 1: A1 on ua\[4], B1 on ui_in\[1]
   - Gate 2: A2 on ui_in\[2], B2 on ui_in\[3]
   - Gate 3: A3 on ui_in\[4], B3 on ui_in\[5]
   - Gate 4: A4 on ui_in\[6], B4 on ui_in\[7]

   The output must follow the NAND truth table:

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

3. **HCT input-window check.** Apply a slow DC ramp on ua\[4] from 0 V to 3.3 V while holding B1 (ui_in\[1]) at 3.3 V. Observe Y1 on ua\[0]. The output must remain at logic high for Vin ≤ 0.8 V and must be at logic low for Vin ≥ 2.0 V.

External hardware

- 3.3 V DC power supply (≥ 50 mA capable).
- Digital multimeter for output-voltage measurement.
- (Optional) Variable low-voltage DC source for the HCT input-window check.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	B1 (Gate 1 input B)	—	—
2	A2 (Gate 2 input A)	—	—
3	B2 (Gate 2 input B)	—	—
4	A3 (Gate 3 input A)	—	—
5	B3 (Gate 3 input B)	—	—
6	A4 (Gate 4 input A)	—	—
7	B4 (Gate 4 input B)	—	—

Analog Pins

ua#	analog#	Description
0	21	Y1 (Gate 1 output)
1	20	Y2 (Gate 2 output)
2	17	Y3 (Gate 3 output)
3	18	Y4 (Gate 4 output)
4	19	A1 (Gate 1 input A; 0/3.3 V for truth table, slow DC ramp 0->3.3 V for HCT window)

TT GF180mcuD Analog Factory Test

by **Sylvain Munaut**

0007 Analog Project

github.com/smunaut/ttgf_analog_factory_test

"Test structures for GF180mcuD analog support"

How it works

It's a couple of big current mirror loads.

A reference current is input, it's multiplied internally by about 50x and draws from selected power rail.

A few sense wires are routed out allowing to measure both voltage drop of the power rails as well as the voltage raise of the VGND rail.

How to test

Provide a test current to the iref input, then toggle the appropriate enable pin to enable to current mirror load on the power rail to be tested.

External hardware

- Programmable current source
- Precision multimeter

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ena_vdpwr	—	—
1	ena_vapwr	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	20	iref

ua#	analog#	Description
1	16	vgnd_sense
2	21	vdpwr_sense
3	17	vapwr_sense
4	18	loopback[0]
5	19	loopback[1]

Simple Signal Generator

by **Matt Venn**

0033 | 50 MHz | HDL Project

github.com/mattvenn/simple-signal-generator

"Square wave generator with a phase-shifting second channel for metastability experiments"

How it works

Two square wave generators drive `uo[1:0]`.

Channel 0 (`uo[0]`): independent square wave. High-time (`on_count`) and low-time (`off_count`) are set independently via SPI, giving full control over frequency and duty cycle.

Channel 1 (`uo[1]`): phase-shifted replica of ch0. It uses the same `on_count` and `off_count` as ch0, but its rising edge is offset by a programmable delay. This is designed for exploring metastability on an SR latch: by placing ch1's edge close to ch0's edge, the two inputs can be brought arbitrarily close together.

Phase control

The total delay applied to ch1's rising edge is:

```
total_delay = spi_offset + enc_int + sigma_delta_carry
```

`spi_offset` (registers 4–5): signed 16-bit static offset in clock cycles, set via SPI. Positive = ch1 lags ch0; negative = ch1 leads ch0. Range: ±32767 cycles.

Encoder (ui[4]=A, ui[5]=B): a quadrature encoder adjusts the phase in real time. Each click changes the phase by `enc_step / 256` cycles (Q8 fixed-point). The integer part of the encoder accumulator is added directly to the delay; the fractional part is dithered via first-order sigma-delta modulation, so sub-cycle offsets are averaged accurately over multiple periods. Encoder range: ±127 cycles (use `spi_offset` for coarse positioning).

`enc_step` (register 10): step size per encoder click in 1/256-cycle units. Examples:

- `enc_step = 1` → 0.004 cycles/click (78 ps at 50 MHz), finest resolution
- `enc_step = 64` → 0.25 cycles/click
- `enc_step = 128` → 0.5 cycles/click
- `enc_step = 255` → 1 cycle/click

Encoder button (`ui[2]`, PmodEnc's `SWT` pin, active-high): while held, multiplies `enc_step` by 8x for fast scanning across the phase range. Releasing it reverts the very next click to the normal step size.

ch1's delay (`spi_offset + enc_int + sigma_delta_carry`) covers the full `[0, period)` range with no clamping — it wraps modulo the period, so ch1's pulse can land anywhere relative to ch0's, including spanning the period boundary. (For very small periods combined with large `spi_offset`/encoder values — larger in magnitude than the period — the wrap is computed mod one period only, so the resulting delay may not match the mathematical mod for `|total_delay| >= period`; this is a pre-existing edge case unrelated to normal use.)

Frequency formula

At 50 MHz, for a 50% duty-cycle square wave at frequency F:

```
on_count = off_count = 25_000_000 // F
```

Both counts are 16-bit → minimum frequency ≈ 382 Hz (`on_count = off_count = 65535`).

Setting `on_count = 0` silences both channels.

The SPI peripheral is reused from calonso88/tt07_alu_74181.

How to test

The SPI master (e.g. RP2350 running MicroPython) programs the design using `machine.SoftSPI`. Each SPI frame is 2 bytes:

- Byte 0: `0x80 | reg_addr` (write), or `reg_addr` (read)
- Byte 1: data

SPI mode: CPOL and CPHA are set via `ui[0]` and `ui[1]` respectively (both 0 for mode 0).

Register map

Reg	Field	Notes
0	ch0 `on_count[15:8]`	MSB
1	ch0 `on_count[7:0]`	LSB
2	ch0 `off_count[15:8]`	MSB
3	ch0 `off_count[7:0]`	LSB
4	`spi_offset[15:8]`	Signed 16-bit MSB; +ve=lag, −ve=lead
5	`spi_offset[7:0]`	Signed 16-bit LSB
6	`enc_step[7:0]`	Q8 step per encoder click (0–255)
7	—	reserved

MicroPython example — 10 kHz, 500-cycle lag, encoder fine-tune

```
from machine import Pin, SoftSPI
```

```
spi = SoftSPI(baudrate=100_000, polarity=0, phase=0, bits=8,
              firstbit=SoftSPI.MSB,
              sck=Pin(30), mosi=Pin(31), miso=Pin(28))
cs = Pin(29, Pin.OUT, value=1)

def write_reg(addr, val):
    cs(0); spi.write(bytes([0x80 | addr, val])); cs(1)

# ch0: 10 kHz, 50% duty (on=2500, off=2500 @ 50 MHz)
write_reg(0, 0x09); write_reg(1, 0xC4)  # on_count  = 2500
write_reg(2, 0x09); write_reg(3, 0xC4)  # off_count = 2500

# ch1: 500-cycle static lag (10 µs)
offset = 500  # cycles
write_reg(4, (offset >> 8) & 0xFF)
write_reg(5,  offset       & 0xFF)

# Encoder: 0.25 cycles/click for fine phase adjustment
write_reg(10, 64)
```

For RP2350 on the TT demo board, `scripts/demo.py` provides ready-made helpers (`set_ch0`, `set_ch0_counts`, `set_phase_offset`, `set_enc_step`, `silence`). Use `scripts/run_freq.py` to program from the command line:

```
python scripts/run_freq.py 1000 --offset 500 --enc-step 64
```

Encoder hardware

Connect a Digilent PModEnc to the **bottom row** of the input Pmod connector on the Tiny Tapeout demo board. This maps the encoder A/B outputs to `ui[4]` and `ui[5]`, and the encoder's pushbutton (`SWT`) to `ui[2]` (fast-scan, see "Phase control" above).

Testing

Simulation (cocotb)

```
source /home/matt/oss-cad-suite/environment
cd test && make
```

Test	What it checks
`test_spi_registers`	Round-trip read/write of all config registers
`test_frequency_generation`	ch0 outputs correct frequency (edge count)
`test_ch1_inphase`	offset=0 → ch1 fires simultaneously with ch0
`test_spi_phase_offset`	Positive SPI offset → ch1 lags ch0 by correct cycle count

`test_channel_silence`	on_count=0 holds both outputs LOW
`test_encoder_integer_phase`	Encoder clicks accumulate to integer cycle offset
`test_encoder_sigma_delta`	Fractional enc_step dithers delay via sigma-delta
`test_encoder_button_fast_scan`	Holding the encoder button multiplies enc_step by 8x
`test_encoder_fast_scan_clamp_no_wrap`	Holding the button at ENC_MAX/ ENC_MIN clamps correctly instead of wrapping (16->17-bit overflow fix)

Hardware-in-the-loop (HIL)

`test/test_hil.py` verifies frequency accuracy on real hardware using a Keysight oscilloscope and the RP2350. Probes on `uo[0]` (CH1) and `uo[1]` (CH2).

External hardware

- RP2350 (or any SPI master) connected to `uio[6:4]` (MOSI, CLK, CS_N) and `uio[3]` (MISO)
- Digilent PModEnc on the bottom row of the input Pmod connector (→ `ui[4]`=A, `ui[5]`=B, `ui[2]`=button/SWT for 8x fast-scan)
- Oscilloscope on `uo[1:0]`
- SR latch with S=`uo[0]` and R=`uo[1]` (or vice versa) for metastability experiments

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SPI CPOL	Channel 0 square wave output	—
1	SPI CPHA	Channel 1 square wave output	—
2	Encoder button (hold for 8x fast-scan)	—	—
3	—	—	SPI MISO
4	Encoder A	—	SPI CS_N (active low)
5	Encoder B	—	SPI CLK
6	—	—	SPI MOSI
7	—	—	—

MPW-2 poly layers – Designed by Matt Venn. Illustrated by Máximo Balestrini.

GF R2R DAC

by **Matt Venn**

0035 Analog Project

github.com/mattvenn/gf-r2r-dac

"8 bit R2R DAC"

How it works

A simple 8 bit R2R DAC. Driven externally or by an digitally generated sine waveform generator.

How to test

Drive externally

Set the `external data` input high to provide the DAC with external data.

Then drive the 8 inputs and observe the analog output.

Drive with internal sine wave generator

Set the `external data` input low to enable the sine generator. A sine wave should be seen on the analog output. Everytime the sine counter is at 0, digital output 0 should go high for one clock.

To change the frequency, set the inputs and then raise the 'load divider' input.

External hardware

An oscilloscope to measure the output voltage on analog pin 0.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	15	output

algofoogle analog stuff

by **algofoogle (Anton Maurovic)**

0037 Analog Project

`github.com/algofoogle/ttgf0p3-algofoogle-analog`

"Some analog and custom layout experiments in gf180mcuD for ttgf0p3 (the TT GF 1st analog experimental shuttle)"

How it works

This is a simple VCO experiment using a current-starved ring oscillator. It consists of 5 current-starved inverter stages in a ring, followed by a buffer pair. This is output via `ua[1]` and also fed to a digital test block where it is divided using a 5-bit synchronous counter and presented on `uio_out[5:1]`, beside a buffered (but non-divided) copy of the VCO output on `uio_out[0]` (aka `osc_out`).

For a given input voltage (`vin`, i.e. `ua[0]`) in the range 0.55V to 3.3V, the oscillator output (`vco_out`, i.e. `ua[1]`) is expected to be a square wave roughly in the range of 2MHz to 400MHz.

How to test

- Apply power with `vin` held at 0V and `rst_n` high. No TT `clk` is required. Expect to see no oscillation on `vco_out` or `uio_out[5:1]`.
- Raise `vin` to 0.55V, and you *might* see `vco_out` oscillating at about 2MHz, 3.3Vpp, about 50% duty cycle.
- Raise `vin` slowly and if `vco_out` wasn't already oscillating then you should see it start at least by the time `vin` reaches 0.65V (if not sooner), and as you raise `vin` further the frequency at `vco_out` should rapidly increase.
- Observe digital-buffered (and TT-digital-mux-shaped) output of the VCO on `uio_out[0]`, and then halving of the frequency up through each of `uio_out[1]` to `uio_out[5]`.

External hardware

- Precision variable voltage source for `vin`.
- Oscilloscope to monitor `vco_out`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	dac_d0	R1	avo[0]
1	dac_d1	G1	avo[1]
2	dac_d2	B1	avo[4]
3	dac_load	VSync	avo[5]
4	dac_shift	R0	dvo[0]
5	vco_disable	G0	dvo[1]
6	test_mode	B0	dvo[4]
7	vga_mode	HSync	dvo[5]

Analog Pins

ua#	analog#	Description
0	12	ana_vco_vin
1	13	ana_vco_out
2	14	dac_vco_in
3	15	dac_vco_out

Hardware Entropy Explorer: UART/SPI TRNG and PUF

by **gojimmypi**

0038 | 25 MHz | HDL Project

`github.com/gojimmypi/ttgf0p3-UART-FSM-TRNG-Lab`

"UART/SPI-controlled ASIC lab for exploring true-random number generation and PUF-style hardware entropy sources."

How it works

A ring oscillator is implemented at the core of this project as an entropy source for a TRNG (True Hardware Random Number Generator).

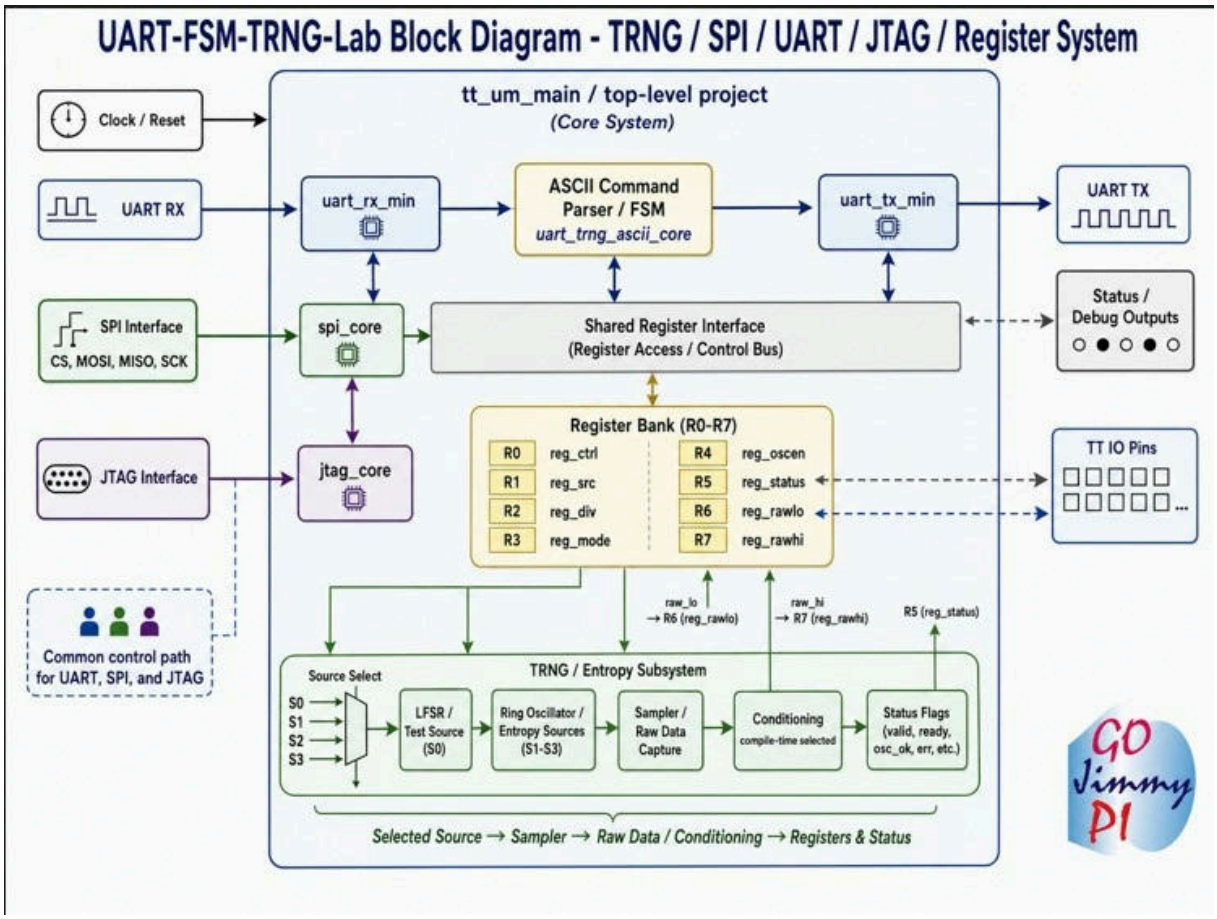


Figure 38.1: UART-FSM-TRNG-Lab-block-diagram.png

This project exposes a UART-controlled interface to a ring-oscillator-based entropy source. A host such as a PC, ESP32, or test script can send simple ASCII commands over UART to configure internal registers, control the oscillator network, and read back raw entropy samples.

At a high level:

- A bank of ring oscillators generates jitter-based entropy
- A sampling clock (controlled by a divider) captures this behavior
- Control and configuration are managed through memory-mapped registers
- Data and status are read back over the same UART interface

Why? The National Institute of Standards and Technology (NIST) notes that random numbers are essential for cryptographic and security applications, and that cryptography makes extensive use of random numbers and random bits, particularly for generating cryptographic keying material.

See presentations:

- NIST Standards on Random Bit Generation slides.
- Why Random Numbers for Cryptography?

SPI vs JTAG Special Note

JTAG is experimental only.

Build / board	Physical setting	`ui_in[4]`	`debug_is_jtag`	Active interface
TT Demoboard	INPUT `SW4` / `IN4` up/off	`0`	`0`	SPI
TT Demoboard	INPUT `SW4` / `IN4` down/on	`1`	`1`	JTAG
ULX3S	`gp4` high / unconnected pull-up	`1`	`0`	SPI
ULX3S	`gp4` pulled low	`0`	`1`	JTAG

For additional related information:

https://gojimmypi.github.io/trng/

https://gojimmypi.github.io/tinytapeout/

External Hardware

It can be helpful to have a TTY-UART USB adapter on hand to interact with the FSM and TRNG on the FPGA or ASIC. This can be used to send commands and read responses from the FSM and TRNG.

Most of the scripts to test assume the external UART. Testing and interactive commands could still be entered via the TT prompt.

FPGA Tests

This project can be tested on an FPGA such as these examples:

- Tiny Tapeout FPGA Development Kit Demoboard in the `ice40` directory.
- ULX3S ECP5 + ESP32 FPGA Development Board in the `ulx3s` directory, and ESP32 SPI Example.

Note that the ring oscillators will not be implemented on the FPGA builds, rather a deterministic Linear-Feedback Shift Register (LFSR) is used in trng_lab_core.v to simulate the TRNG bitstream.

See the `FPGA_NIST_PRNG_SOURCE` and `FPGA_BASIC_LFSR_RO_TAPS` options in `project_config.v` that are disabled for the TT build.

Commander App Tests

Use the commander.tinytapeout.com to connect to the tt-commander-app

---

How to test

The TT projects usually start in a reset mode = `True`. Connect to TT Breakout (or Demoboard) USB.

Once connected, there should be a Python REPL command prompt.

Don't confuse the TT board serial connection with the external UART.

Ensure all the dip input switches are in the `up` default (off) position.

Select the project, set the clock to 25 MHz, and reset. (see project_reset.py):

```
# select project and reset ttgf
tt.shuttle.tt_um_gojimmypi_ttgfa_UART_FSM_TRNG_Lab.enable()

tt.clock_project_PWM(25000000)
tt.reset_project(True)
tt.reset_project(False)
```

Connect a UART terminal (e.g. PuTTY) to the TT Breakout (or Demoboard) I/O pins with the following connections:

- UART/TTY USB Tx to `IN3/Rx`
- UART/TTY USB Rx to `OUT4/Tx`
- GND to `GND`

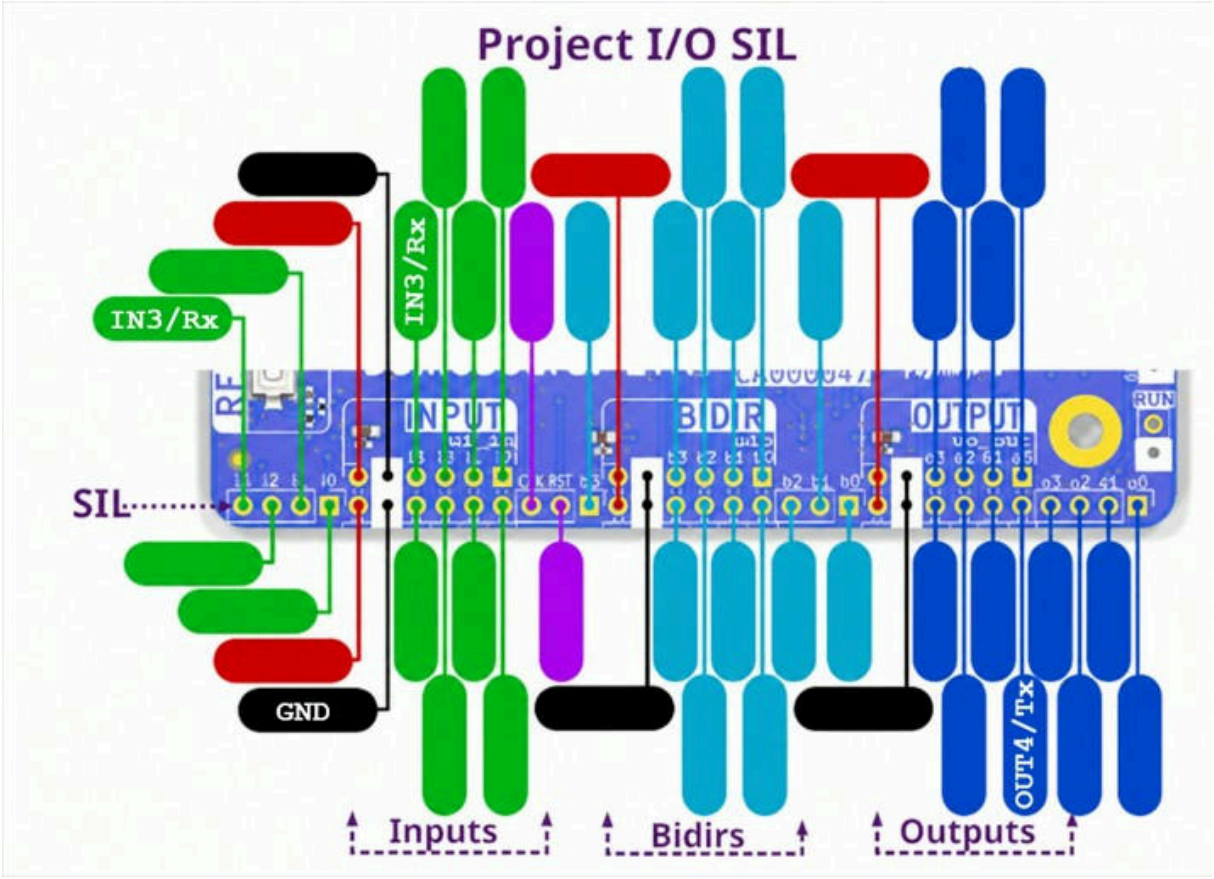


Figure 38.2: PMOD-connector-test1.png

⚠ ** **CAUTION:** ** Pins are 3v3 and NOT expected to be 5v tolerant.

⚠ ** **CAUTION:** ** TT IO pins such as `Tx` and `Rx` are likely ** **NOT** ** tolerant to reversal. See TT Discord.

That's the same as shorting them. They're definitely not designed for it, but they won't die immediately either.

Note: `IN3` and `OUT4` are Tiny Tapeout logical signal names, not PMOD physical pin numbers. On the shown PMOD adapter:

- `in3` is PMOD `IO4` /physical pin 4.
- `out4` is PMOD `IO5` / physical pin 7.

Project config:

- `clock_hz: 25000000` in `info.yaml`
- `define PROJECT_CLOCK_HZ 32'd25_000_000` in `src/project_config.v`
- `define PROJECT_UART_BAUD 32'd115_200` in `src/project_config.v`

At a 25 MHz project clock with `PROJECT_UART_BAUD = 115_200`:

- `CLKS_PER_BIT = 25_000_000 / 115_200` = 217
- Terminal baud rate: 115,200

At a 50 MHz project clock, if the design is rebuilt with `PROJECT_CLOCK_HZ = 50_000_000`:

- `CLKS_PER_BIT = 50_000_000 / 115_200` = 434
- Terminal baud rate: 115,200

If the bitstream was built for 25 MHz but the board is actually clocked at 50 MHz, the effective UART baud rate doubles to approximately 230,400 baud.

Terminal session at 25 MHz clock is

- 115,200 baud
- 8 data bits
- No parity
- 1 Stop
- No flow control (Although the default XON/XOFF should also work, but ignored)

Or:

```
stty -F "$PORT" "$BAUD" cs8 -cstopb -parenb -ixon -ixoff -crtscts
raw -echo min 0 time 5
```

Type `V` and press `Enter` to query the version string (if enabled in the build, on by default for TT). Then you can send commands to configure the TRNG and read back entropy samples.

⚠️ The TT Build is Case Sensitive. Although there are case-insensitive settings available for local FPGA builds, they have been disabled for TT ASIC due to observed increased slew and setup violations.

Type `RD` and press enter to view the Build Target ID. The expected value for GF180 ASIC is 42.

Send the appropriate commands to configure and read from the TRNG core. See Register Overview, below.

NIST Validation

NIST has a Resource for Random Bit Generation testing:

Overview

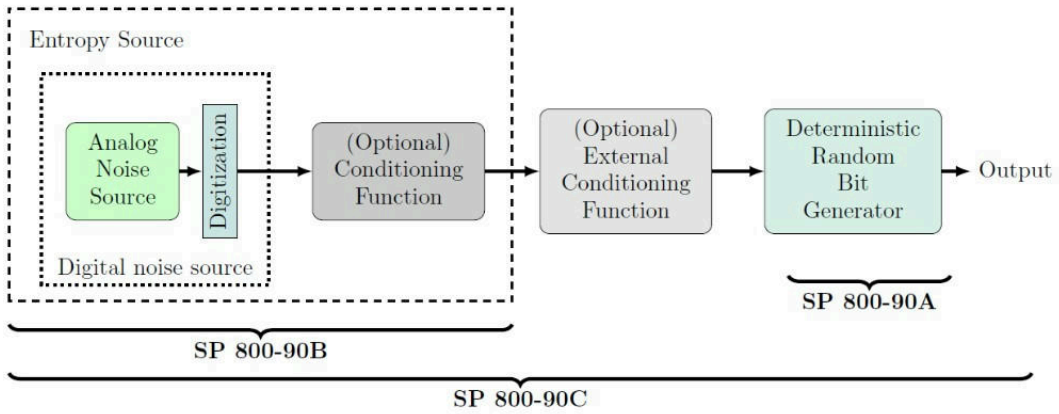
The National Institute of Standards and Technology (NIST) Random Bit Generation (RBG) project focuses on the development and validation of generating random numbers that are essential for cryptographic and security applications.

SP 800-90 Series

The project provides guidelines through the SP 800-90 series, which includes recommendations on deterministic random bit generator (DRBG) mechanisms, entropy sources, and construction principles for RBGs, and has three parts:

- SP 800-90A, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, specifies mechanisms for generating random bits using deterministic methods. NIST is revising SP 800 90A to be consistent with SP 800-90C.
- SP 800-90B, *Recommendation for the Entropy Sources Used for Random Bit Generation*, specifies the design principles and requirements for the entropy sources used by RBGs and the tests for the validation of entropy sources.
- SP 800-90C, *Recommendation for Random Bit Generator (RBG) Constructions*, specifies constructions for the implementation of RBGs.

The following figure explains the relationship of the three parts of the series.



NIST IR 8427, *Discussion on the Full Entropy Assumption of the SP 800 90 Series*, provides technical discussions to support the full entropy definition used in the SP 800 90 series.

Image credit: screen snip from csrc.nist.gov/Projects/random-bit-generation

See the `capture_trng_raw_uart.py` script to capture a binary file of random data from this project, large enough for 100 runs of 1,000,000-bit NIST-style tests:

```
# WSL /dev/ttyS[n] == COM[n] on Windows, other Linux: /dev/
ttyUSB[n], /dev/ttyACM[n], etc

./capture_trng_raw_uart.py  --port /dev/ttyS12  --bytes 16777216
--out trng_raw.bin
```

This script requires a build with `TRNG_BINARY_STREAM` enabled.

The raw output is intended for experimentation and characterization. It is not a certified cryptographic random number generator.

When the optional `define TRNG_CONDITIONED_STREAM` is used in `project_config.v`, the conditioned output can be generated with the `--conditioned` option:

```
./capture_trng_raw_uart.py \
    --port /dev/ttyS12 \
    --bytes 16777216 \
    --out trng_conditioned.bin \
    --fast-baud \
    --conditioned
```

See also:

```
# The official STS package from NIST CSRC:
# https://csrc.nist.gov/CSRC/media/Projects/Random-Bit-Generation/
documents/sts-2_1_2.zip

unzip sts-2_1_2.zip
cd sts-2.1.2
make

#
# or this UNOFFICIAL mirror:
# https://github.com/terrillmoore/NIST-Statistical-Test-Suite.git

cd NIST-Statistical-Test-Suite
./setup.sh
cd sts
make
```

For further testing information see NIST Random Bit Generation RBG - Guide to the Statistical Tests.

Quickstart Simulation

```
cd /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/test

./my_test.sh

./jtag_test.sh
```

Quickstart Testing on TT Demoboard

If all the toolchains are installed:

```
cd /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ice40

source ./env_ice40.sh
./build_and_flash.sh
./project_reset.sh
./run_tests.sh
```

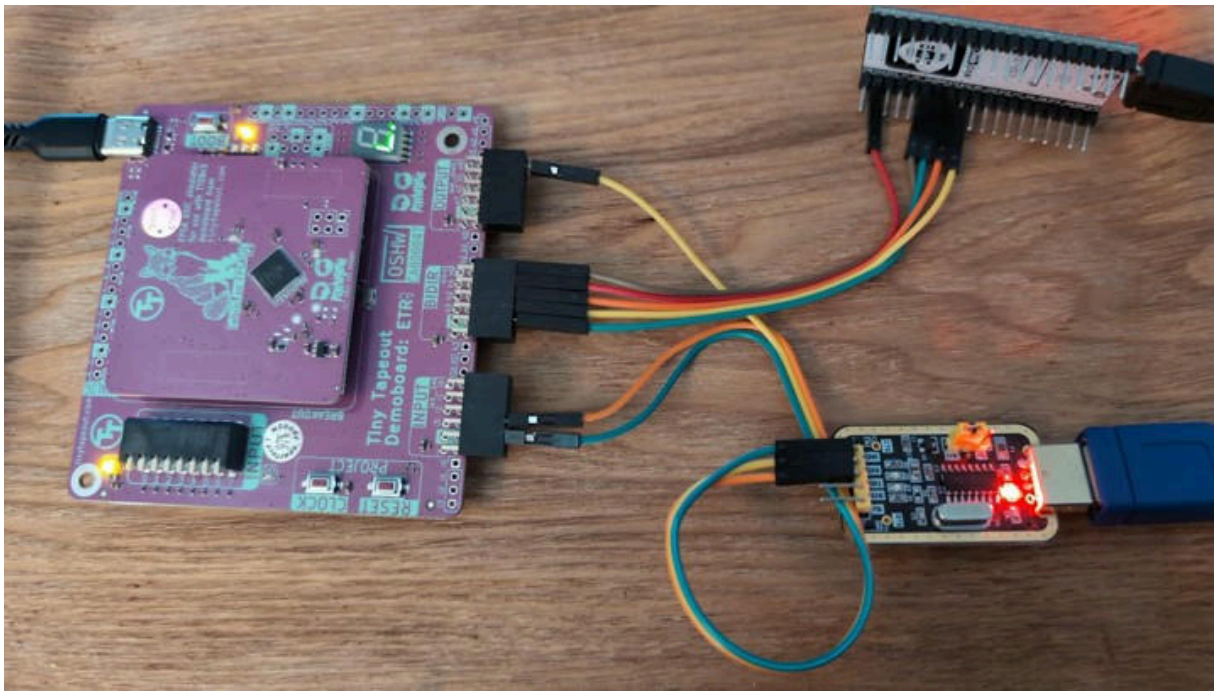


Figure 38.3: TT-Demoboard-SOFT_UART-SOFT-SPI-Wiring.jpg

Sample Soft SPI connected to ESP32 and Soft UART connected to external USB/TTY UART.

Despite the “F” that may be repeatedly displayed on the 7-segment display during testing, that does not indicate failure:

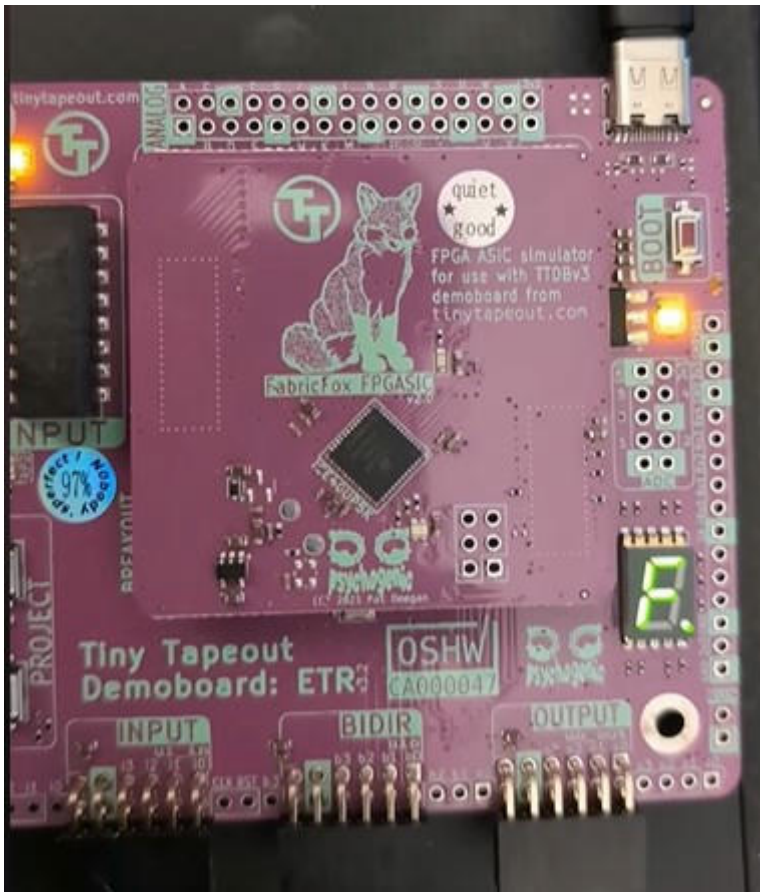


Figure 38.4: Demoboard_F_is_for_Fun_Success.jpg

From youtube.com/shorts/zFnfsl1DQHE

Quickstart Testing on ULX3S

See the `[project]/ulx3s` and `[project]/test-hw` directories.

ULX3S Connections

All pins are 3v3 and assumed to NOT be 5v tolerant.

Soft External UART

⚠ Do not connect to 5V TTY

- `GND` on `J1` pin 4; Ground connection. Beware of adjacent `3v3` on `J1` pins 1 and 2.
- `GP0` for `Rx` on `J1` pin 6 (connect to external USB/TTY UART `Tx`)
- `GP1` for `Tx` on `J1` pin 8 (connect to external USB/TTY UART `Rx`)

Soft SPI

Select SPI by leaving TT `IN4` up/off, or leaving ULX3S `gp4` high/unconnected/pull-up.

- For TT boards, `INPUT` Dip Switch `IN4` up/off gives `ui_in[4] = 0`, selecting SPI.

- For ULX3S, `gp4` high/unconnected/pull-up gives `shared_spi_jtag_select = 1`, selecting SPI.

Pins are already connected to the on-board ESP32 - but for debugging reference:

- `GND` on `J1` pin 5; Ground connection. Beware of adjacent `3v3` on `J1` pins 1 and 2.
- `GN2` -> (TT `uio[0]`) TMS
- `GP2` -> (TT `uio[1]`) TDI
- `GN3` <- (TT `uio[2]`) TDO
- `GP3` -> (TT `uio[3]`) TCK

See `/ulx3s/ESP32/main/ulx3s_spi_lib.c`

⚠ Do not accidentally wire ESP32 `GPIO2` to PMOD `GP2`. `GPIO2` goes to `GN3`, because it is `MISO/TDO`. Also be careful around `J1`: use pin 5 `GND`, not the adjacent `3v3` pins 1/2.

ULX3S ESP32 SPI Pins

```
#define PIN_NUM_MISO        2
#define PIN_NUM_MOSI        15
#define PIN_NUM_CLK         14
#define PIN_NUM_CS          13
#define SPI_CLOCK_HZ        1000000
```

ESP32 signal	ESP32 GPIO	TT/ PMOD pin	TT signal	JTAG-style name	Direction	Wire
PIN_NUM_CS	GPIO13	GN2	uio[0]	TMS	ESP32 -> TT	Brown
PIN_NUM_MOSI	GPIO15	GP2	uio[1]	TDI	ESP32 -> TT	Red
PIN_NUM_MISO	GPIO2	GN3	uio[2]	TDO	TT -> ESP32	Orange
PIN_NUM_CLK	GPIO14	GP3	uio[3]	TCK	ESP32 -> TT	Yellow
GND	ESP32 GND	J1 pin 5	GND	-	common ground	Green

Stand-alone ESP32 SPI Pins

⚠ Do not use these pins on the ULX3S ESP32.

Disable `IS_ULX3S_ESP32` macro in `ulx3s_spi_lib.c` to use external stand-alone ESP32:

```
#define PIN_NUM_MISO        19
#define PIN_NUM_MOSI        23
#define PIN_NUM_CLK         18
#define PIN_NUM_CS          21
#define SPI_CLOCK_HZ        1000000
```

ESP32 signal	ESP32 GPIO	TT/PMOD pin	TT signal	Direction	Wire
`PIN_NUM_CS`	GPIO21	`GN2`	`uio[0]` / CS / TMS	ESP32 -> TT	Brown
`PIN_NUM_MOSI`	GPIO23	`GP2`	`uio[1]` / MOSI / TDI	ESP32 -> TT	Red
`PIN_NUM_MISO`	GPIO19	`GN3`	`uio[2]` / MISO / TDO	TT -> ESP32	Orange
`PIN_NUM_CLK`	GPIO18	`GP3`	`uio[3]` / SCK / TCK	ESP32 -> TT	Yellow
`GND`	ESP32 GND	`J1` pin 5	`GND`	common ground	Green

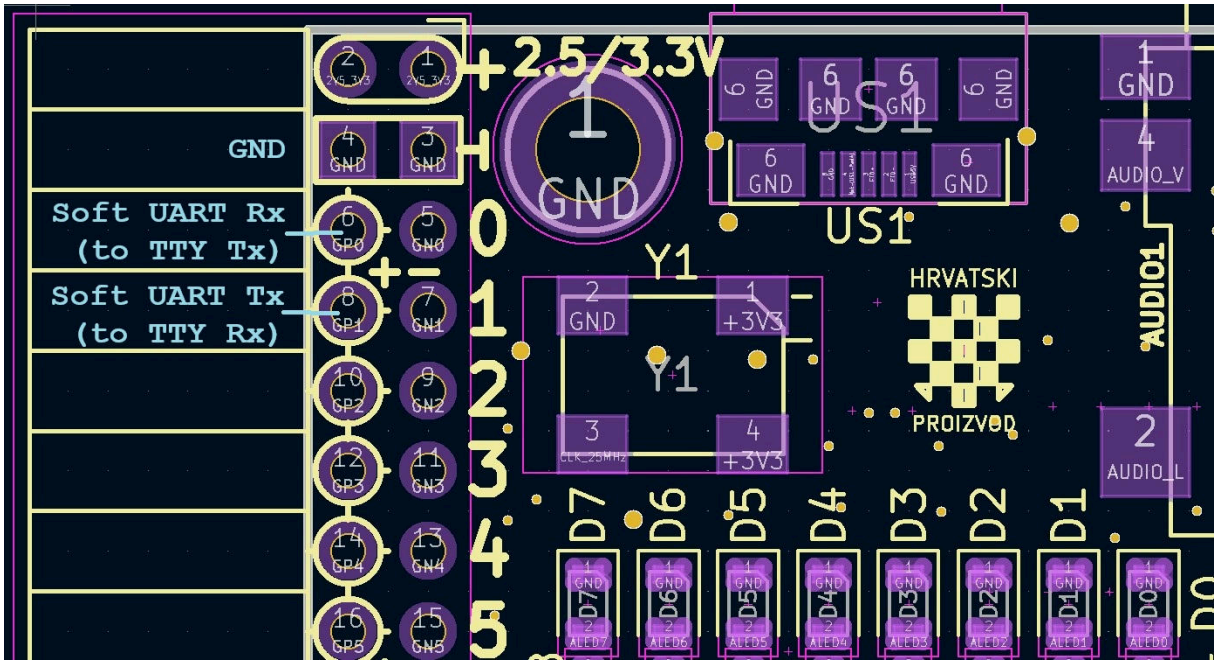


Figure 38.5: ULX3S-Pin-Connections.jpg

Build and run tests from the `./test-hw` directory.

```
cd /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/test-hw

# may need to remove generated file
rm  ../src/_tt_fpga_top.v
```

```
# Edit board version as needed, tested on older v3.0.7:
./run_tests.sh  --with-build  --ulx3s-board-version v307  --ignore-
combinational-warning  --no-warning-pause  --port /dev/ttyS12 --
pause-for-test
```

Quickstart on ULX3S ESP32

The onboard ESP32 is pre-configured to work with this TT project. No external wiring is needed.

```
# [project]/ulx3s/ESP32
cd /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ESP32

PORT=/dev/ttyS3

idf.py -p $PORT -b 115200 flash
idf.py -p $PORT -b 115200 monitor
```

See also Comprehensive Testing below and the TT MicroPython SDK v3.

Register Overview

Register	Description
`reg_ctrl`	Global control bits (enable, feature flags)
`reg_src`	Selects entropy source or oscillator group
`reg_div`	Clock divider controlling sampling rate
`reg_mode`	Operating mode configuration
`reg_oscen`	Bitmask enabling individual oscillators
`reg_status`	Status flags (data ready, internal state)
`reg_rawlo`	Low byte of raw sampled entropy
`reg_rawhi`	High byte of raw sampled entropy

Key Concepts

- **Enable (`E`)**
  Must typically be cleared (`E0`) before changing configuration, then set (`E1`) to run.
- **Oscillator Control (`O`)**
  Enables one or more ring oscillators. More oscillators can improve entropy but may affect stability.
- **Sampling (`D`)**
  The divider controls how frequently entropy is sampled. This impacts randomness quality and bias.

- **Source Selection (`S`)**
  Allows switching between different entropy paths or test modes (implementation-specific).
- **Raw Data (`R6`, `R7`)**
  Returns unprocessed entropy bytes. These are not whitened and may require post-processing.

Typical Flow

1. Disable the core (`E0`)
2. Configure source, divider, mode, and oscillators
3. Enable the core (`E1`)
4. Read entropy and status via `R6`, `R7`, `R5`

This simple interface allows interactive exploration of TRNG behavior directly from a terminal.

UART TRNG Command Interface

All commands are ASCII and terminated with `\r`.
Responses are ASCII for normal register/configuration commands, typically:

`R<n>=<value>`

The optional `Bxx` raw stream command returns binary bytes and does not append `OK<CR>`.

Write Commands

Cmd	Description
`E<n>`	Write enable bit (0=disable, 1=enable)
`S<n>`	Write source select
`V<n>`	Write control bit 1
`W<n>`	Write control bit 2
`D<hex>`	Write divider
`M<hex>`	Write mode
`O<hex>`	Write oscillator enable mask

Special:

- `V\r` -> returns version string (if enabled in build)

Read Commands

Cmd	Description
R0	Read reg_ctrl
R1	Read reg_src
R2	Read reg_div
R3	Read reg_mode
R4	Read reg_oscen
R5	Read reg_status
R6	Read reg_rawlo
R7	Read reg_rawhi

Examples

Enable and configure:

```
E0\r
V0\r
W0\r
S0\r
D10\r
M00\r
O01\r
E1\r
```

Read back registers:

```
R0\r -> R0=01
R2\r -> R2=10
R6\r -> R6=7B
R7\r -> R7=3C
```

Version query:

```
V\r -> Version x.x.x <date>
```

Binary raw stream, when enabled:

```
B10<CR> -> 16 raw binary bytes
B64<CR> -> 100 raw binary bytes
BFF<CR> -> 255 raw binary bytes
B00<CR> -> ?<CR>
```

The `xx` byte count is hexadecimal, not decimal.

Do not use a normal terminal to view `Bxx` output. The response may contain arbitrary byte values, including control characters. Use `capture_trng_raw_uart.py` or another binary-safe capture tool.

Notes

- Commands are stateful; configure with `E0` before changes
- `R6`/`R7` provide raw entropy bytes
- `O` controls active oscillators (entropy source)
- `D` affects sampling rate and bias

UART

Connect with your favorite terminal program such as putty.

For the ULX3S FPGA, the UART is connected to pins `gp0` and `gp1` The default baud rate is 115200.

See the default reference ULX3S `ulx3s_v20.lpf` restraint file.

The `B11` (aka `gp[0]` or `gp0`) is Rx, to UART Tx. The `A10` (aka `gp[1]` or `gp1`) is Tx, to UART Rx.

```
# UART pins for testing

LOCATE COMP "uart_rx_pin" SITE "B11"; # formerly "gp[0]"; # J1_5+
GP0 PCLK
IOBUF PORT "uart_rx_pin" IO_TYPE=LVCMOS33;

LOCATE COMP "uart_tx_pin" SITE "A10"; # formerly "gp[1]"; # J1_7+
GP1 PCLK
IOBUF PORT "uart_tx_pin" IO_TYPE=LVCMOS33;
```

Comprehensive Testing

There are TT simulation tests and local ULX3S FPGA tests.

Set the `TT_PROJECT_ROOT` environment variable to the root of the project directory before running the tests or other scripts.

```
export TT_PROJECT_NAME="ttgf0p3-UART-FSM-TRNG-Lab"
export TT_PROJECT_ROOT="/mnt/c/workspace/$TT_PROJECT_NAME"
```

Testing on the Tiny Tapeout FPGA Development Kit

See the overview video for the FPGA Development Kit.

Testing ULX3S / TT

First run this script in one bash terminal, note test pause “Press Enter to continue…” (see concurrent `Testing ESP32`, below)

```
cd "$TT_PROJECT_ROOT/test-hw"

./run_tests.sh --with-build --ulx3s-board-version v307 --ignore-
combinational-warning --no-warning-pause --port /dev/ttyS12 --
pause-for-test
```

Testing SPI with ESP32

The ULX3S has a built-in ESP32, but a standalone ESP32 can also be used to test the SPI interface.

Current testing scripts:

```
# change directory to your ESP-IDF directory:
cd /mnt/c/SysGCC/esp32-master/esp-idf/v5.5

source ./export.sh

cd "$TT_PROJECT_ROOT/ulx3s/ESP32"
idf.py build

idf.py -p /dev/ttyS3 -b 115200 flash

idf.py -p /dev/ttyS3 -b 115200 monitor
```

There should be output from the ESP32 showing the SPI transactions and register values. This can be used to verify that the SPI interface is working correctly and that the TRNG lab core is responding to commands. (See example output):

```
gojimmypi:/mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ESP32
$ idf.py -p /dev/ttyS3 -b 115200 monitor
Executing action: monitor
Running idf_monitor in directory /mnt/c/workspace/ttgf0p3-UART-FSM-
TRNG-Lab/ulx3s/ESP32
Executing "/home/gojimmypi/.espressif/python_env/idf5.5_py3.10_env/
bin/python /mnt/c/SysGCC/esp32-master/esp-idf/v5.5/tools/
idf_monitor.py -p /dev/ttyS3 -b 115200 --toolchain-prefix xtensa-
esp32-elf- --target esp32 --revision 0 /mnt/c/workspace/ttgf0p3-
UART-FSM-TRNG-Lab/ulx3s/ESP32/build/ulx3s_esp32.elf /mnt/c/
workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ESP32/build/bootloader/
bootloader.elf -m '/home/gojimmypi/.espressif/python_env/
idf5.5_py3.10_env/bin/python' '/mnt/c/SysGCC/esp32-master/esp-idf/
v5.5/tools/idf.py' '-p' '/dev/ttyS3' '-b' '115200'"...
--- esp-idf-monitor 1.6.2 on /dev/ttyS3 115200
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
I (13) boot: ESP-IDF v5.5 2nd stage bootloader

  [... snip. etc ... ]
I (350) main: SPI write mode: boot config once
I (350) ulx3s_spi: SPI regs: R0=00 R1=00 R2=10 R3=00 R4=01 R5=04
R6=3E R7=84 raw=0x843E status=0x04 src=0 div=0x10 mode=0x00
oscen=0x01
I (460) main: TRNG deterministic LFSR test
I (460) main: lfsr test sample 00: raw=0x7F2E status=0x00
I (460) main: lfsr test sample 01: raw=0x9F33 status=0x00
```

```
  [... snip. etc ... ]

I (740) ulx3s_spi: SPI regs: R0=00 R1=00 R2=10 R3=00 R4=01 R5=04
R6=3E R7=84 raw=0x843E status=0x04 src=0 div=0x10 mode=0x00
oscen=0x01
I (740) ulx3s_spi: SPI regs: R0=00 R1=00 R2=10 R3=00 R4=01 R5=04
R6=3E R7=84 raw=0x843E status=0x04 src=0 div=0x10 mode=0x00
oscen=0x01
```

If the `./run_tests.sh` was left at the `Press Enter to continue...` prompt, press `Enter` to continue with the next set of tests. The output should look something like this example:

```
Build PASSED
Flash...
Flashing file:
-rw-r--r-- 1 gojimmypi gojimmypi 294455 Jun  4 08:22 /mnt/c/
workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ulx3s.bit
ULX2S / ULX3S JTAG programmer v4.8 (git 96ebb45 built Oct  7 2020
22:42:00)
Copyright (C) Marko Zec, EMARD, gojimmypi, kost and contributors
Using USB cable: ULX3S FPGA 12K v3.0.3
Programming: 100%
Completed in 12.78 seconds.
/mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/test-hw
Press Enter to continue...

Skipping register reset. Use --reset-registers to start from
configured defaults.

Running: version_if_present
Version probe response: b'Version 0.1.5d 6/3/2026\r'
PASS: Version command

  [... snip. etc ... ]
```

The continued output from the ESP32 in the separate TTY window should look something like this:

```
I (186760) ulx3s_spi: SPI regs: R0=06 R1=00 R2=10 R3=00 R4=01 R5=00
R6=00 R7=00 raw=0x0000 status=0x00 src=0 div=0x10 mode=0x00
oscen=0x01
I (187760) ulx3s_spi: SPI regs: R0=06 R1=00 R2=2A R3=5C R4=0F R5=00
R6=00 R7=00 raw=0x0000 status=0x00 src=0 div=0x2A mode=0x5C
oscen=0x0F
I (188760) ulx3s_spi: SPI regs: R0=00 R1=00 R2=10 R3=00 R4=01 R5=04
R6=00 R7=00 raw=0x0000 status=0x04 src=0 div=0x10 mode=0x00
oscen=0x01
```

TT Simulation tests

Commit changes. See results in actions.

In particular, note the output of the gds workflow:

- Linter output
- Routing Stats
- Cell usage by Category
- Tiny Tapeout Precheck Results
- Viewer summary

Test on ULX3S FPGA

Build and flash the bitstream to the FPGA, then run the test script. The test script will print the output of the FSM and TRNG.

Test locally with ULX3S ECP5 FPGA in /ulx3s/ directory.

- verilator_lint.sh
- ulx3s_build.sh
- ulx3s_flash.sh

Example:

```
cd ulx3s

./ulx3s_build.sh
./ulx3s_flash.sh
```

Connect to the FPGA using a serial terminal (e.g., `putty` or `minicom`) to view the output of the FSM and TRNG.

Local Loopback Test

There are two loopback tests: a basic loopback test and a deep loopback test. The basic loopback test verifies that the UART is functioning correctly by sending data from the FPGA to the host and back. The deep loopback test verifies that the FSM and TRNG are functioning correctly by sending commands to the FPGA and reading the responses.

Basic Loopback Test

The basic loopback assigns `Tx` to `Rx` in `top_ulx3s.v`.

```
    assign uart_tx_pin = uart_rx_sync;
```

All characters should be echoed back in the terminal when you type. This verifies that the UART is working correctly.

Sample loopback build defines `FORCE_LOOPBACK=1` macro in `ulx3s_build.sh`:

```
./ulx3s_build.sh --loopback --ignore-combinational-warning --no-
warning-pause
./ulx3s_flash.sh
```

Deep Loopback Test

```
./ulx3s_build.sh --deep-loopback --ignore-combinational-warning --
no-warning-pause
./ulx3s_flash.sh
```

Extensive loopback tests

Additional loopback tests:

```
    # The safest test to start (default write_with_delay when --
bulk not specified)
    python ./loopback_test.py --port $PORT -b 115200
|| exit 1

    echo "Test non-bulk mode, delay = 0.005"
    python ./loopback_test.py --port $PORT -b 115200 --tx-delay
0.005 || exit 1

    echo "Test non-bulk mode, delay = 0.001"
    python ./loopback_test.py --port $PORT -b 115200 --tx-delay
0.001 || exit 1

    echo "Test non-bulk mode, delay = 0.000"
    python ./loopback_test.py --port $PORT -b 115200 --tx-delay
0.000 || exit 1

    echo "Test bulk mode most challenging"
    python ./loopback_test.py --port $PORT -b 115200 --bulk
|| exit 1
```

The `run_tests.sh` can be used to run the loopback tests with the appropriate flags:

```
cd "$TT_PROJECT_ROOT/test-hw

./run_tests.sh --with-build --ignore-combinational-warning --no-
warning-pause --loopback
./run_tests.sh --with-build --ignore-combinational-warning --no-
warning-pause --deep-loopback
./run_tests.sh --with-build --ignore-combinational-warning --no-
warning-pause
```

Local Automated Hardware Operation Tests

Generic local hardware operation tests in /test-hw/.

- tt_uart_test.py - Python script to test the UART functionality of the FSM and TRNG on the ULX3S FPGA. It sends commands to the FPGA and reads the responses to verify correct operation.
- run_tests.sh - Shell script to run the hardware tests. It can be configured to build the FPGA bitstream, flash it to the FPGA, and run the Python test script.

```
cd test-hw

 ./run_tests.sh --with-build --ignore-combinational-warning --no-
warning-pause
```

UART FSM TRNG Lab Datasheet

Document revision: 1.0.5 RTL revision string: `Version 1.0.5 6/27/2026`
Project family: Tiny Tapeout UART/SPI configurable TRNG experiment
Primary top modules: `tt_um_gojimmypi_ttgfa_UART_FSM_TRNG_Lab` (conditional based on build) License: Apache-2.0, as declared in the source files

1. Overview

The UART FSM TRNG Lab is a Tiny Tapeout compatible experimental random-number and entropy-source project. It exposes a small register bank through an ASCII UART command interface and, when enabled, an SPI mode 0 register-access interface. The design is intended for laboratory bring-up, education, FPGA validation, and ASIC ring-oscillator entropy experiments.

The core supports multiple sample sources:

- Deterministic LFSR source for repeatable tests
- Single ring-oscillator sample source
- XOR of multiple ring-oscillator sources
- Mixed source combining ring-oscillator and LFSR-derived state

The design is not a certified cryptographic random number generator. Raw output should be characterized, health-tested, and conditioned before use in security-sensitive systems.

2. Key Features

- Tiny Tapeout standard digital pin interface
- UART RX/TX control path using ASCII commands
- Optional SPI register-access slave
- SPI mode 0, MSB first
- Shared UART and SPI register bank
- 8-byte logical register map
- Configurable sample divider
- Selectable entropy/sample source
- Ring oscillator enable mask
- Deterministic single-step mode for test reproducibility
- Reset control through a configuration bit
- Default 25 MHz project clock
- Default 115200 baud UART
- ASIC real ring-oscillator path for selected PDK builds
- FPGA/simulation-safe LFSR tap substitute when real ring oscillators are disabled

3. Design Status and Intended Use

This block is intended as an experimental TRNG lab core. It is suitable for:

- Tiny Tapeout project demonstration
- FPGA bring-up on ULX3S or similar wrappers
- UART command parser testing
- SPI register interface testing
- Ring oscillator experimentation in supported ASIC flows
- Deterministic regression testing using the LFSR source

It is not, by itself, suitable as a drop-in cryptographic RNG. The raw output is unconditioned and no formal entropy claim is made in this datasheet.

4. Source Files

The main RTL files are:

File	Purpose
`project.v`	Top-level Tiny Tapeout project wrapper and project feature defines
`project_config.v`	Project clock and UART baud configuration
`target_pdk.v`	PDK target selection
`tt_um_main.v`	Tiny Tapeout pin mapping and UART/SPI/TRNG integration
`UART/uart_rx_min.v`	Minimal UART receiver
`UART/uart_tx_min.v`	Minimal UART transmitter
`UART/uart_trng_ascii_core.v`	UART and TRNG integration core
`TRNG/trng_cfg_ascii_core.v`	ASCII command parser and register bank
`TRNG/trng_lab_core.v`	Experimental TRNG/lab source logic
`TRNG/trng_stub.v`	Stub/test TRNG replacement when the lab core is not enabled
`SPI/spi_slave.v`	SPI mode 0 register-access slave
`JTAG/jtag_core.v`	Optional JTAG-related logic

5. Top-Level Parameters

Parameter	Default	Description
`CLOCK_HZ`	`25000000`	Project clock frequency in Hz
`UART_BAUD`	`115200`	UART baud rate

The source contains parameter checks that intentionally fail elaboration if either parameter is zero or if `CLOCK_HZ / UART_BAUD` would be zero.

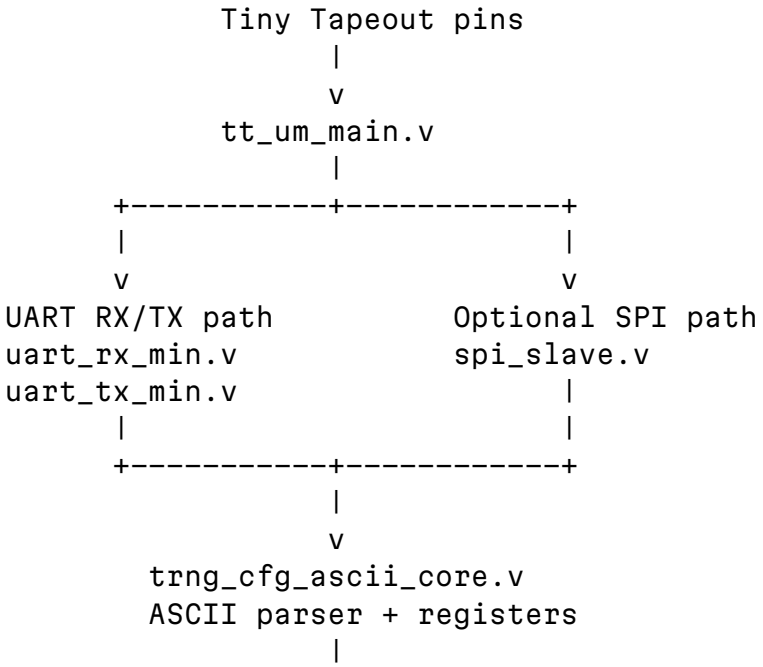
The ULX3S_USE_GN12_50MHZ configuration path can set PROJECT_CLOCK_HZ to 50 MHz for the optional ULX3S gn12 clock path. Normal builds use the 25 MHz project clock.

6. Build-Time Feature Defines

Define	Purpose
UART_ENABLED	Enables UART-related integration
SPI_ENABLED	Enables the SPI slave pin path
SPI_REG_ACCESS	Enables SPI access to the shared register bank
TRNG_ENABLED	Selects the TRNG lab core instead of the stub core
JTAG_ENABLED	Enables JTAG-related build integration
USE_LONG_STRINGS	Enables the long version string reply path
TRNG_USE_RO	Requests the real ring-oscillator TRNG path
TRNG_ALLOW_REAL_RO	Explicit guard required with TRNG_USE_RO
TRNG_BINARY_STREAM	Enables the UART Bxx raw binary byte-stream command
PDK_TARGET_SKY130	Selects SKY130 inverter cell instantiation
PDK_TARGET_GF180	Selects GF180 inverter cell instantiation
ULX3S	Selects ULX3S FPGA wrapper/build behavior

For ULX3S builds, the source intentionally rejects TRNG_USE_RO and TRNG_ALLOW_REAL_RO. FPGA and normal simulation paths use deterministic LFSR-derived substitutes for the ring oscillator signals.

7. Functional Block Diagram



```
              v
      trng_lab_core.v
LFSR / RO / mixed sampling
              |
              v
status, raw low byte, raw high byte
```

8. Clock and Reset

Signal	Active level	Description
`clk`	Rising edge	Main synchronous project clock
`rst_n`	Low	Global reset

On reset, the register bank is initialized as shown below:

Register	Reset value
`reg_ctrl`	`0x00`
`reg_src`	`0x00`
`reg_div`	`0x10`
`reg_mode`	`0x00`
`reg_oscen`	`0x01`

The TRNG lab core internally resets the LFSR to `0x1ACE`, clears `sample_shift`, clears the sample counter, clears status, and clears raw output registers.

9. Tiny Tapeout Pin Map

Dedicated inputs: `ui_in[7:0]`

Pin	Direction	Function
`ui_in[7:5]`	Input	Reserved / unused
`ui_in[4]`	Input	SPI/JTAG select, 0 = SPI, 1 = JTAG (`INPUT` Dip Switch `SW4` down; when JTAG_ENABLED is defined)
`ui_in[3]`	Input	UART RX
`ui_in[2:0]`	Input	Reserved / unused

The UART RX input is synchronized through a two-stage synchronizer before it enters the UART receive logic.

Dedicated outputs: `uo_out[7:0]`

Pin	Direction	Function
`uo_out[0]`	Output	Debug visibility: `trng_bit`
`uo_out[1]`	Output	Debug visibility: `reg_status[0]`

uo_out[2]	Output	Debug visibility: reg_status[1]
uo_out[3]	Output	Debug visibility: reg_status[2]
uo_out[4]	Output	UART TX
uo_out[5]	Output	reg_rawlo[0]
uo_out[6]	Output	reg_rawlo[1]
uo_out[7]	Output	reg_rawlo[2]

Bidirectional IO: `uio[7:0]` **when SPI is enabled**

Pin	Direction	Function
uio[0]	Input	SPI CS_N
uio[1]	Input	SPI MOSI
uio[2]	Output	SPI MISO
uio[3]	Input	SPI SCK
uio[7:4]	Output	reg_rawhi[7:4] debug visibility

When SPI is enabled, `uio_oe` is driven as `0xF4`, making `uio[2]` and `uio[7:4]` outputs while leaving `uio[0]`, `uio[1]`, and `uio[3]` as inputs.

Bidirectional IO when SPI is disabled

When SPI is not enabled, `uio_out[7:0]` drives the full `reg_rawhi` byte and `uio_oe` is driven as `0xFF`.

10. UART Interface

UART settings

Setting	Value
Baud rate	UART_BAUD, default 115200
Data bits	8
Parity	None
Stop bits	1
Byte order	ASCII command bytes
Command terminator	Carriage return, 0x0D

Line feed, `0x0A`, is ignored in command wait states, allowing common CRLF terminal behavior.

UART command summary

Command	Arguments	Effect	Reply

Bxx	2 hex nibbles, 01..FF	Stream xx raw binary bytes from reg_rawlo/ reg_rawhi alternately, when TRNG_BINARY_STREAM is enabled	Binary bytes, no OK<CR>
E0 / E1	1 hex nibble	Write reg_ctrl[0], TRNG enable	OK<CR>
Sx	1 hex nibble	Write reg_src[1:0]	OK<CR>
Vx	1 hex nibble	Write reg_ctrl[1], deterministic single-step request	OK<CR>
Wx	1 hex nibble	Write reg_ctrl[2], TRNG reset control	OK<CR>
Dxx	2 hex nibbles	Write reg_div[7:0]	OK<CR>
Mxx	2 hex nibbles	Write reg_mode[7:0]	OK<CR>
Oxx	2 hex nibbles	Write reg_oscen[7:0]	OK<CR>
Rn	n = 0..7	Read register n	Rn=HH<CR>
V	None	Version query	Version string + <CR>

Invalid syntax returns ?<CR>.

UART command examples

```
V<CR>       -> Version 1.0.5 6/27/2026<CR>
R2<CR>      -> R2=10<CR>
E1<CR>      -> OK<CR>
D10<CR>     -> OK<CR>
S0<CR>      -> OK<CR>
O01<CR>     -> OK<CR>
R6<CR>      -> R6=HH<CR>
R7<CR>      -> R7=HH<CR>
```

To reconstruct the current 16-bit raw sample from UART register reads:

```
raw16 = (R7 << 8) | R6
```

11. SPI Interface

The SPI interface is available when `SPI_ENABLED` and `SPI_REG_ACCESS` are enabled.

SPI electrical/protocol settings

Setting	Value
Mode	SPI mode 0
CPOL	0
CPHA	0
Bit order	MSB first
Chip select	Active low, `CS_N`
Register address width	3 .. 7 bits (see project_config.v)

SPI command byte

Bit field	Description
`bit[7]`	`1` = read, `0` = write
`bit[6:3]`	Ignored
`bit[2:0]`	Register address `0..7` or `0..15` or `0..127` (see project_config.v)

SPI read transaction

```
byte 0: 0x80 | addr
byte 1: dummy byte; returned MISO byte is the register value
```

For example, reading `reg_rawlo` at address 6:

```
TX: 86 00
RX: xx HH
```

The useful read value is the second received byte.

SPI write transaction

```
byte 0: addr
byte 1: data byte
```

Only addresses 0 through 4 are writable. Writes to addresses 5 through 7 are ignored by the register bank.

For example, writing divider register `R2 = 0x10`:

```
TX: 02 10
```

12. Register Map

Addr	UART read	Name	Access	Reset	Description
0	`R0`	`reg_ctrl`	R/W	`0x00`	Control bits

1	R1	reg_src	R/W	0x00	Source selection
2	R2	reg_div	R/W	0x10	Sample divider
3	R3	reg_mode	R/W	0x00	Mode/debug field
4	R4	reg_oscen	R/W	0x01	Ring oscillator enable mask
5	R5	reg_status	Read-only	0x00	Status mirror
6	R6	reg_rawlo	Read-only	0x00	Raw sample low byte
7	R7	reg_rawhi	Read-only	0x00	Raw sample high byte

13. Control Register: `reg_ctrl`, Address 0

Bit	Name	Description
0	enable	Enables periodic sampling when set
1	step	Deterministic single-step request. A rising edge creates one sample event
2	reset	Resets the TRNG lab core while asserted
7:3	Reserved	Currently unused

UART aliases:

Command	Field
E0 / E1	reg_ctrl[0]
V0 / V1	reg_ctrl[1]
W0 / W1	reg_ctrl[2]

Note: bare V<CR> is the version query. V0<CR> and V1<CR> are control writes.

14. Source Register: `reg_src`, Address 1

Only `reg_src[1:0]` is used.

Value	Source	Description
0	SRC_LFSR	LFSR bit source, deterministic and repeatable
1	SRC_RO0	Sampled ring oscillator 0 source
2	SRC_ROX	XOR of the ring oscillator raw bits
3	SRC_MIX	Mixed source using RO XOR, LFSR taps, and sample history

UART alias: Sx<CR> writes `reg_src[1:0]`.

15. Divider Register: `reg_div`**, Address 2**

`reg_div` controls the periodic sample interval when `reg_ctrl[0]` is enabled. The internal sample counter increments while enabled. A sample event is generated when:

```
sample_ctr >= reg_div
```

A single-step event through `reg_ctrl[1]` can also generate a sample event without waiting for the periodic divider.

UART alias: `Dxx<CR>` writes the full divider byte.

16. Mode Register: `reg_mode`**, Address 3**

`reg_mode` is a full 8-bit writable register. In the current TRNG lab core, `reg_mode[2:0]` is mirrored into `reg_status[7:5]`. Other bits are reserved for future use.

UART alias: `Mxx<CR>` writes the full mode byte.

17. Oscillator Enable Register: `reg_oscen`**, Address 4**

`reg_oscen` is an 8-bit enable mask for the ring oscillator instances in real RO builds.

There's a conditional `BASIC_RO_SET` (not defined) in `trng_lab_core.v` for RO states 3 .. 17.

The default build contains 7 .. 21 stages:

Bit	Real RO instance	Stage count
0	`u_ro0`	7
1	`u_ro1`	9
2	`u_ro2`	11
3	`u_ro3`	13
4	`u_ro4`	15
5	`u_ro5`	17
6	`u_ro6`	19
7	`u_ro7`	21

In FPGA and normal simulation builds, the RO raw bits are derived from LFSR taps instead of real ring oscillators.

UART alias: `Oxx<CR>` writes the full oscillator enable mask.

18. Status Register: `reg_status`**, Address 5**

Bit field	Description
`bit[0]`	Mirrors TRNG enable

bit[1]	Mirrors sample tick condition
bit[2]	Indicates at least one oscillator enable bit is set
bits[4:3]	Mirrors source selection
bits[7:5]	Mirrors reg_mode[2:0]

reg_status is read-only from the external UART/SPI register interfaces.

19. Raw Output Registers: reg_rawlo and reg_rawhi

The TRNG lab core maintains a 16-bit sample shift register. On each sample event, the selected source bit is shifted into the sample history and the raw output registers are updated.

Register	Description
reg_rawlo	Low byte of the latest raw sample history
reg_rawhi	High byte of the latest raw sample history

The current 16-bit raw sample value is reconstructed as:

```
raw16 = (reg_rawhi << 8) | reg_rawlo
```

20. Sampling Behavior

A sample event occurs when either condition is true:

```
do_sample = (enable && sample_tick) || step_pulse
```

Where:

```
sample_tick = sample_ctr >= reg_div
step_pulse  = reg_ctrl[1] && !previous_reg_ctrl_bit_1
```

On each sample event:

- sample_ctr is cleared to zero
- The 16-bit LFSR advances
- The selected source bit shifts into sample_shift
- reg_rawlo and reg_rawhi are updated

When enable is deasserted, the sample counter is held at zero. A single-step pulse can still advance one sample.

21. LFSR Details

The deterministic LFSR path is used for repeatable tests and for FPGA/simulation-safe substitute RO signals. On TRNG reset, the LFSR seed is:

```
0x1ACE
```

The next LFSR bit is computed as:

```
lfsr_next_bit = lfsr[15] ^ lfsr[13] ^ lfsr[12] ^ lfsr[10]
```

The LFSR then shifts as:

```
lfsr <= {lfsr[14:0], lfsr_next_bit}
```

This path is deterministic and should not be treated as an entropy source.

22. Ring Oscillator Path

In real ASIC RO builds, the default design instantiates eight odd-length ring oscillators with stage counts from 7 to 21. The oscillator outputs feed the source selection logic through synchronizers.

In FPGA and normal simulation builds, real ring oscillators are not instantiated. Instead, the RO raw signals are mapped to LFSR taps so the rest of the design can be tested safely without combinational oscillator loops.

Supported real RO PDK cell paths in the current source are:

PDK define	Inverter cell
`PDK_TARGET_SKY130`	`sky130_fd_sc_hd__inv_2`
`PDK_TARGET_GF180`	`gf180mcu_fd_sc_mcu7t5v0__inv_2`

23. Recommended Bring-Up Sequence

A conservative UART bring-up sequence is:

```
V<CR>       Read version string
R0<CR>      Confirm control reset value
R1<CR>      Confirm source reset value
R2<CR>      Confirm divider reset value, expected 10
R3<CR>      Confirm mode reset value
R4<CR>      Confirm oscillator enable reset value, expected 01
S0<CR>      Select deterministic LFSR source
D10<CR>     Set divider to 0x10
M00<CR>     Clear mode
O01<CR>     Enable oscillator mask bit 0
E1<CR>      Enable sampling
R6<CR>      Read raw low byte
R7<CR>      Read raw high byte
```

For deterministic regression testing, use source `S0`, assert and release reset through `W1` and `W0`, then issue single-step pulses through `V1` and `V0` as required by the test harness.

24. Known Deterministic Regression Sequence

With the deterministic LFSR path and the established single-step/reset test flow, the known-good 16-bit sample sequence used in current hardware regression is:

```
sample 01: 0x7F2E
sample 02: 0x9F33
sample 03: 0xFC1C
```

```
sample 04: 0x6F03
sample 05: 0x4B7D
sample 06: 0x52C8
sample 07: 0xD6B7
sample 08: 0xEF2A
```

This sequence is a reproducibility check for the deterministic path. It is not an entropy-quality claim.

25. UART and SPI Concurrency Notes

UART and SPI share the same logical register bank. SPI writes are applied when `spi_reg_wr_en` is asserted. UART commands also update the same configuration registers.

When using SPI as a passive monitor while UART is active, individual register reads are separate transactions. Reading `reg_rawlo` and `reg_rawhi` separately is not atomic, so the two bytes may occasionally come from adjacent sample updates. For exact sample capture, add an atomic snapshot/latch mechanism or temporarily stop sampling before reading both bytes.

26. Limitations

- No cryptographic certification is claimed.
- No built-in conditioner, extractor, or DRBG is provided by this RTL block.
- Raw RO entropy quality must be measured on the actual ASIC implementation.
- FPGA and normal simulation builds do not use real ring oscillators.
- SPI multi-byte reads of raw low/high registers are not atomic.
- UART command parsing is intentionally small and accepts only the documented command forms.
- Register addresses 5 through 7 are read-only through SPI writes and UART write aliases do not target them.

27. Characterization Recommendations

Before using ASIC RO output as a source of entropy, characterize at minimum:

- Raw bit bias for each source selection
- Bit transition rate
- Autocorrelation
- Per-oscillator behavior across voltage and temperature
- Behavior across process corners and multiple chips
- Startup behavior after reset
- Sensitivity to `reg_div` and `reg_oscen`
- Health test behavior under stuck oscillator conditions

For security use, add a conditioning function and a health-test strategy appropriate for the target application.

28. Revision History

Datasheet rev	Date	Notes
0.1	2026-05-23	Initial datasheet generated from current TRNG source package

Example Outputs

- Example ESP32 Output
- Example Text Results

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	reserved_in0	trng_bit	spi_cs_n_jtag_tms
1	reserved_in1	status0	spi_mosi_jtag_tdi
2	reserved_in2	status1	spi_miso_jtag_tdo
3	uart_rx	status2	spi_sck_jtag_tck
4	spi_jtag_sel	uart_tx	rawhi4
5	reserved_in5	rawlo0	rawhi5
6	reserved_in6	rawlo1	rawhi6
7	reserved_in7	rawlo2	rawhi7

Detail - Fibonacci design (MPW-2) – Designed by Konrad Rzeszutek Wilk. Illustrated by Máximo Balestrini.

MoM capacitor

by **htfab**

0039 Analog Project

`github.com/htfab/ttgf0p3-momcap`

"Simple metal-oxide-metal capacitor for gf180mcuD that fills an analog 1x2 tile"

How it works

Implements a simple interdigitated metal-oxide-metal capacitor:

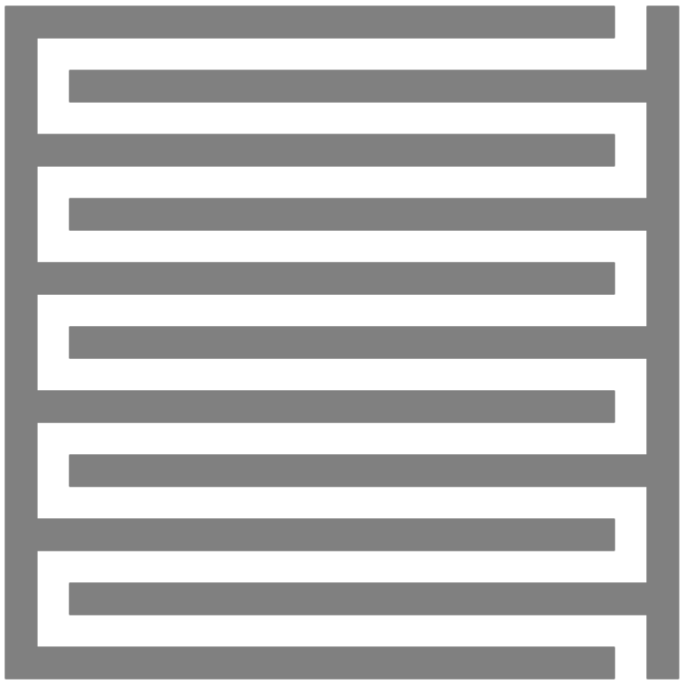


Figure 39.1: momcap layout illustration

The repo contains both a script to generate a custom size capacitor and a fixed instance filling a Tiny Tapeout analog slot (1x2 tiles).

How to test

Measure the capacitance between terminals A and B.

You can use the dummy terminals to control for parasitic capacitance on the path between your probes and the project pins.

For comparison, magic's circuit extraction estimates the capacitance at around 85 pF.

External hardware

Test equipment (function generator + oscilloscope, or VNA).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	12	capacitor terminal A
1	13	capacitor terminal B
2	14	dummy terminal
3	15	dummy terminal

Until heat death do us part

by **Julia Desmazes**

0099 50 MHz HDL Project

`github.com/Essenceia/Until_Heat_Death_Do_Us_Part`

"Counting until the heat death of the universe, broadcasting counter value every second over ethernet."

ASIC with a large enogth counter to theoretically counts until the heat death of the the universe. Counter value is incremented ever 20ns and every second it will broadcasts an ethernet frame over 100Mbps ethernet with the current counters value, not that anyone is going to be listening for long anyways.

Assuming the universe dies in approximatively 10^{100} years. If you are patient enogth to wait until then there is a hidden easter egg when we eventually overflow.

How it works

This project is re-using the Teapot Ethernet accelerator wrapper for communicating over 100Mbps Ethernet using a CAT-3/5 cable in full-duplex mode. Here is to hopeing someone still remeber how to talk Ethernet by then.

Ethernet packets

This wrapper workes with layer 2 ethernet packets, opperating at the level of the ethernet frame. It support two types of packets:

- `application packets, ethtype = 0x88B5` sent by the accelerator
- `configuration packets, ethtype = 0x88B6` used to set the ASICs MAC address, Vlan IDentifier, and the TX phase selection

All packets whos destination MAC do not match the ASIC's current MAC address will be filtered out. Unless otherwise specified the ASIC's MAC address is `00:90:CF:00:BE:EF` (read as Nortel:BEEF).

Application packets

The end of the universe counter does not listen to any incomming packets (we are all going to be dead soon anyways) and broadcasts the current counter value every second.

Response

```
[ dst mac (6 Bytes) FF:FF:FF:FF:FF:FF ][ src mac (6 Bytes) ]
[ ethtype = 0x88B5 (2 Bytes) ][ magic number = 0xCAFE (2 Bytes) ]
[ counter (48 Bytes) ][ FCS (4 Bytes) ]
0
```

Counter values are sent in little endian with a granule size of 1 byte (standard).

Configuration packets
Configuration packets are used to set the ASIC's current:

- MAC address
- Vlan ID
- TX data to clock phase offset

These packets are not forwarded to the accelerator, do not provide any acknoledgement and due to our area limitation are not store and forwarded. Any corrupted packets will result in a corrupted configuration.

Packet:

```
[ dst mac (6 Bytes) ][ src mac (6 Bytes) ][ ethtype = 0x88B6 (2
Bytes) ][ New MAC (6 Bytes) ][ padding (3 bits)][ VID (12 bits)]
[ padding (38 Bytes) ][ FCS (4 Bytes) ]
0
```

Unless otherwise configured application packets use ethertype `0x88B6`, the second IEEE 802.3 specified "Local Experimental Ethertype", whos existence linux networking libraries are unaware of. So trust me: it's real.

Configuration parameters

MAC address
ASIC's current MAC address, all packets not addressed (`dst mac`) to this address will be filtered out, and all responses will use this as the source address.

Default MAC: `00:90:CF:00:BE:EF` (read as Nortel:BEEF)

Configuration pins: TX data to reference clock phase offset
To comphensate for the output data to reference clock offset induced by the delay on the path from the clock input pin, to the tiny tapeout design's data out flip-flop and back to the output pin, the reference clock for the data out flip flop is selectable, allowing us to use a 180 degree dephased reference clock.

This dephasing configuration is captured during reset depending on the state of the `tx_phase` pin.

Values:

- `0` no phase shift
- 1 180 degree phase shift

How to test

Connect the ethernet 100Mbps capable connector to the asic, if the connector doesn't expose a `rx_err` signal clamp it to gnd. Cable the ethernet

connector to your local network, it doesn't have to be directly to your computer so long your are in the same local network (layer2 packets can be routed within it).

The ASIC will autonomously send an application packet (ethtype:`0x88b5`) every 1s.

Network sniffing tool

Build the packet sender/receiver app in `tools`:

```
cd tools
make
```

To run pass the name of your ethernet interface currently connected to the same LAN as the ASIC. Eg: I am connected though my wifi interface:

```
sudo ./packet_receiver wlp3s0
```

Example of the expected output:

```
[pitchu](master) ~/asic/heat_death/tools >sudo ./packet_receiver
wlp3s0
ethernet interface: wlp3s0
wlp3s0 mac address 60:e9:aa:92:dc:7d
asic mac address 00:90:cf:00:be:ef
dst mac ff:ff:ff:ff:ff:ff
src mac 00:90:cf:00:be:ef
ethtype 88b5
counter: 70282575969
counter debug: 0x000000005d2b0061
raw pkt:
ffffffffffff0090cf00beef88b5feca61002b5d1000000000000000000000000000000000

dst mac ff:ff:ff:ff:ff:ff
src mac 00:90:cf:00:be:ef
ethtype 88b5
counter: 70332579937
counter debug: 0x0000000060260061
raw pkt:
ffffffffffff0090cf00beef88b5feca6100266010000000000000000000000000000000000

dst mac ff:ff:ff:ff:ff:ff
src mac 00:90:cf:00:be:ef
ethtype 88b5
counter: 70382583905
counter debug: 0x0000000063210061
raw pkt:
ffffffffffff0090cf00beef88b5feca6100216310000000000000000000000000000000000

dst mac ff:ff:ff:ff:ff:ff
src mac 00:90:cf:00:be:ef
ethtype 88b5
```

```
counter: 70432522337
counter debug: 0x00000000661b0061
raw pkt:
ffffffffffff0090cf00beef88b5feca61001b661000000000000000000000000000000000000
```

In case of issues

You can also observe the packets being sent back by sniffing your live traffic via tcpdump:

```
sudo tcpdump -xx -e -v 'ether proto 0x88b5'
```

In case issues perciste

In case you still see no traffic.

Directly connect the ethernet cable to your computer and check the ethernet interface exposed low level counters to see if you are seeing traffic on your line.

eg:

```
sudo ethtool -S enp2s0
```

External hardware

Ethernet 100BASE-T Pmod connector, featuring:

- LAN8720A PHY
- 50MHz oscillator

Amazone version of the external pmod connector.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	phy_rx_data[0]	phy_tx_data[0]	—
1	phy_rx_data[1]	phy_tx_data[1]	—
2	phy_rx_v	phy_tx_v	—
3	phy_rx_err	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	tx_phase	—	—

TT GF180mcuD Testbuffer

by **Christoph Weiser**

0101 Analog Project

github.com/christoph-weiser/ttgf0p3-submission

"Analog voltage buffer"

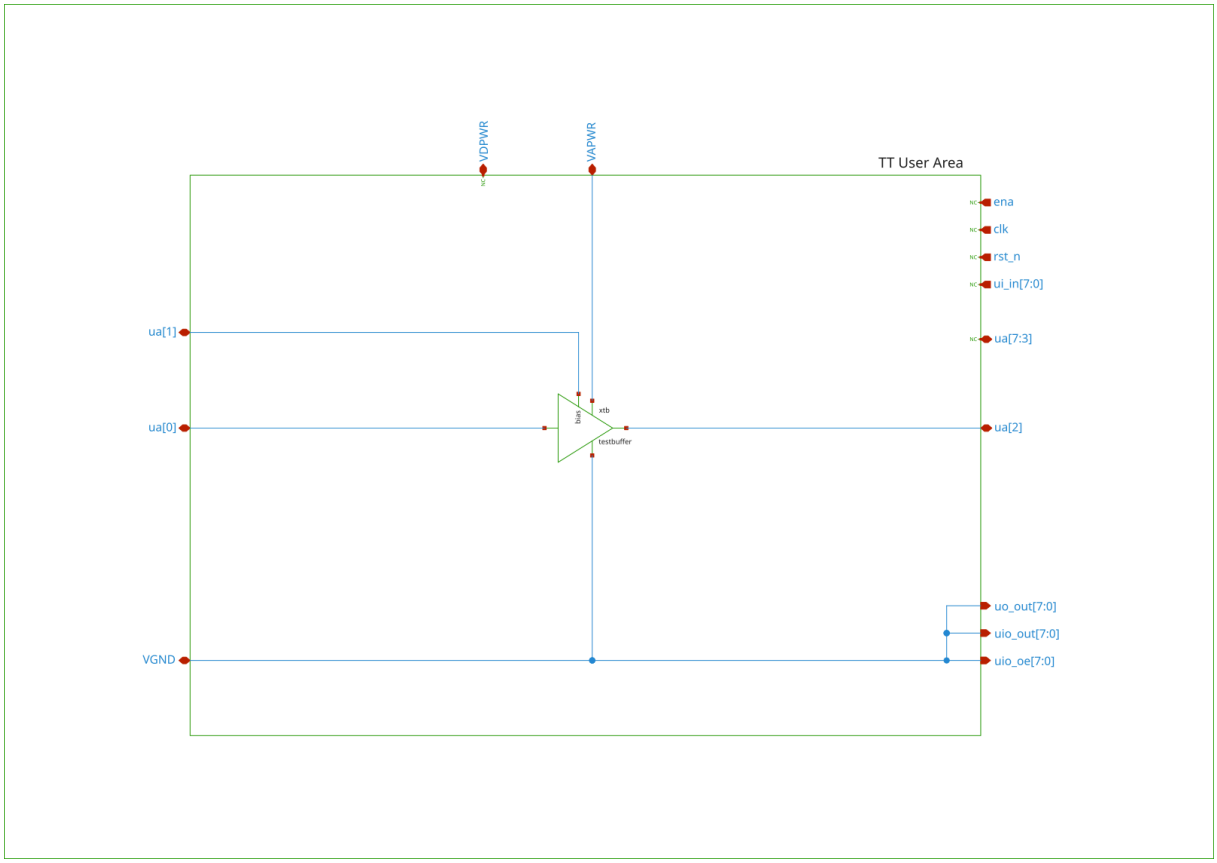


Figure 101.1: top-level schematic

How it works

A testbuffer that will buffer a voltage signal and drive a off-chip probe.

How to test

- Bias the amplifier with a 5uA current from pin ua[1].
- Apply test signal to input pin ua[0]
- Measure buffered signal at the output pin ua[2].

External hardware

- Oscilloscope and function generator or voltage source and multimeter

Specifications

Parameter	Symbol.	Min.	Typ.	Max.	Unit.	Condition
Supply Voltage	Vdd	3.0	3.3	5.0	V	—
Temperature Range	T	0	—	75	°C	—
Area	A	—	—	90x80	μm^2	—
Load Capacitance	Cl	—	10	50	pF	—
Load Resistance	Rl	—	1	—	MΩ	—
Bias Current	Ib	—	5	—	µA	—
Supply Current	Idd	47.9	50.6	57.5	µA	Ib = 5µA, All PVT, Rl = 1MΩ
Power consumption	Pd	143	167	292	µW	Ib = 5µA, All PVT, Rl = 1MΩ
—	—	—	—	—	—	—
Input offset voltage	Vos	—	10.2	—	mV	+-3σ, Vdd = 3.3V, T = 27°C
Integrated noise	eni	30.9	36.3	42.3	µVrms	f = 0.1Hz to 1MHz
Input referred noise	en	9.39	—	11.45	µV/sqrt(Hz)	f = 0.1Hz, All PVT
—	—	0.926	—	1.129	µV/sqrt(Hz)	f = 10Hz, All PVT
—	—	95.6	—	116.3	nV/sqrt(Hz)	f = 1kHz, All PVT
Open Loop Gain	Avol	83.5	85.9	—	dB	All PVT, Rl=1MΩ
Unity Gain Bandwidth	UGBW	1.78	2.6	3.4	MHz	All PVT, Cl=10pF
Phase Margin	φm	64.5	80	—	°	All PVT, Cl=10pF
Slew Rate	SR	1.2	1.72	—	MV/s	All PVT, Cl=10pF
Power Supply	PSRR+	103	123	—	dB	f = 1Hz, All PVT

Rejection Ratio Pos						
—	—	85	88	—	dB	f = 100Hz, All PVT
—	—	65	68	—	dB	f = 1kHz, All PVT
Power Supply Rejection Ratio Neg	PSRR-	101	115	—	dB	f = 1Hz, All PVT
—	—	88	91	—	dB	f = 10kHz, All PVT

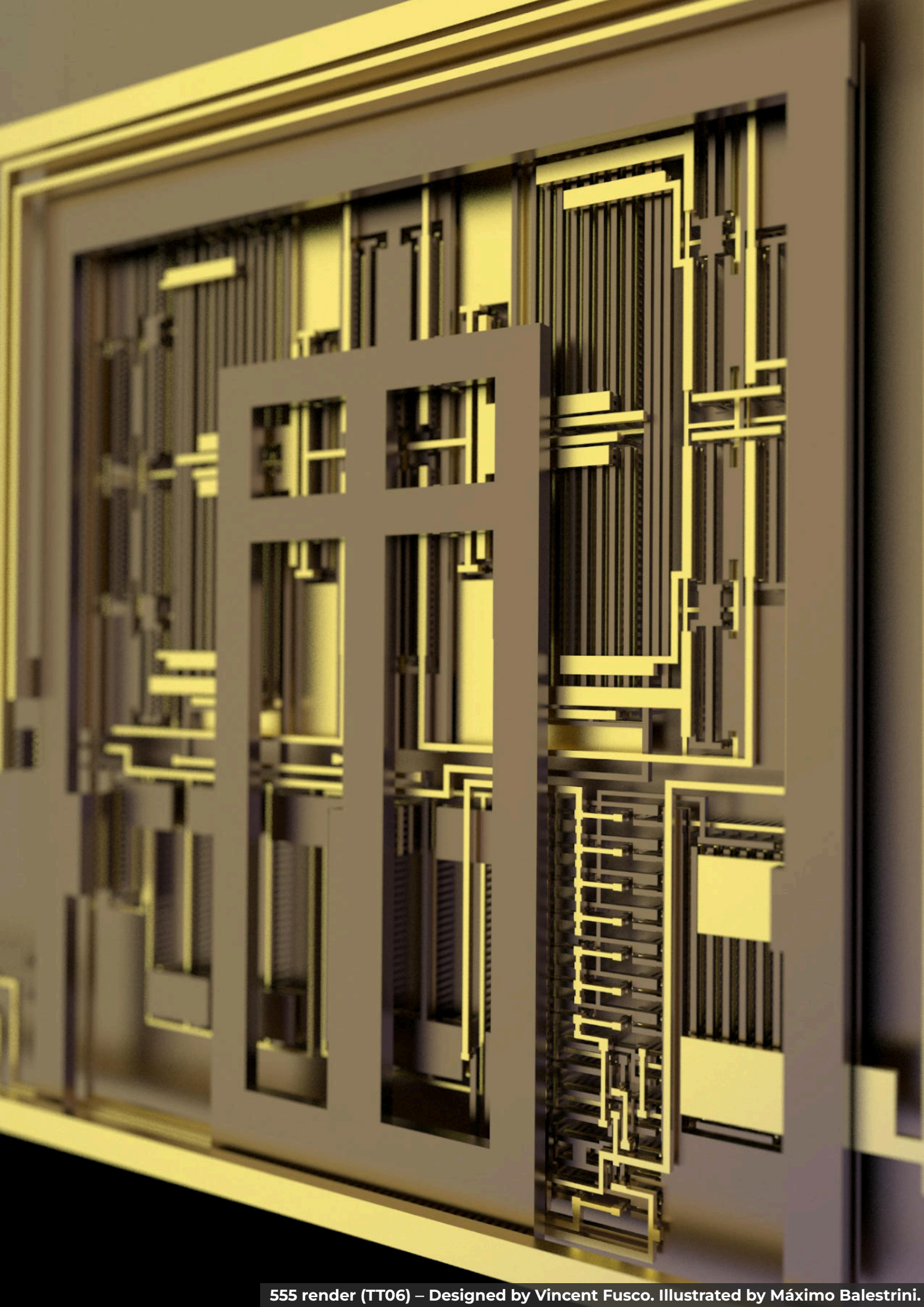
Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	10	in
1	8	bias
2	9	out



555 render (TT06) – Designed by Vincent Fusco. Illustrated by Máximo Balestrini.

Tiny FABulous FPGA

by **Leo Moser**

0102 HDL Project

github.com/mole99/tt-fabulous-gf-0p3

"A tiny FABulous FPGA fabric, ready for use with Yosys and nextpnr."

How it works

Tiny FABulous FPGA for GF0p3.

This design implements a tiny FPGA with 64 LUT4+FF. The FPGA fabric is 6x4 tiles in size, of which 4x2 are LUT4x8_ha tiles. The logic cells include a vertical carry-chain in upwards direction, allowing for fast additions up to 15-bits.

The I/Os resemble the Tiny Tapeout interface, allowing for clk, rst_n, uo, ui and uio signals. This enables to directly implement simple Tiny Tapeout designs on the FPGA.

The user design is synthesized using Yosys and implemented using nextpnr (currently forks are required to be used, but the changes will be upstreamed).

The bitstream is uploaded to the fabric using a bitbang interface (see how to test). The bitbang interface is active while reset is applied, this ensures that all I/Os are available for the active user design.

The exact available resources can be seen in this table:

Primitive	Available	Description
FABULOUS_LC	64	Logic cells with LUT4+FF and carry-chain.
IOBUF	26	Input/output buffers.
GBUF	4	Global buffers to supply clock, reset and enable to the flip-flops.
SYS_RESET	1	Can be used to reset the design after configuration.

Even though there are 26 IOBUF are available, only the uio signals are actually bidirectional. uo will always read zero when read from, and writing to clk, rst_n and ui has no effect.

The GBUFs are used for high-fanout signals. Their use is mandatory for the clock signal of flip-flops to ensure a balanced clock network. This means up to 4 clock domains are possible. The GBUFs can also be used for reset and enable of the FFs, although those can also be routed through "normal" fabric routing.

SYS_RESET applies a reset during fabric reconfiguration and can only be directly connected to a GBUF.

How to test

First, compile a bitstream for your user design. The bitstream is big-endian with 32-bit words.

1. Set rst_n to 1 to reset the configuration interface.
2. Set rst_n to 0 to enable the configuration interface.
3. Write the bitstream bits to ui[1] (MSB first) and the sample signal on ui[0].

The data is sampled on a rising edge of the sample signal. The interface is synchronous, so ensure that the clk signal is toggling faster than the sample signal. Is anything is unclear, have a look at the top-level cocotb tests.

Finally, set rst_n to 1 and enjoy your design on Tiny FABulous FPGA!

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ui[0] or sample_i	uo[0]	uio[0]
1	ui[1] or data_i	uo[1]	uio[1]
2	ui[2]	uo[2]	uio[2]
3	ui[3]	uo[3]	uio[3]
4	ui[4]	uo[4]	uio[4]
5	ui[5]	uo[5]	uio[5]
6	ui[6]	uo[6]	uio[6]
7	ui[7]	uo[7]	uio[7]

tnt's recreation of a 555 on GF180mcu

by **Sylvain Munaut**

0103 Analog Project

github.com/smunaut/tt_tnt_gf_555

"Just a simple classic timer IC re-imagined"

How it works

It's a recreation of a 555. The interal circuit doesn't match exactly what's in a (CMOS) 555 but function should be identical.

How to test

Refere to a 555 datasheet typical circuits and re-create some of them.

External hardware

- Resistors / Capacitors
- A scope to observe the output

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	On-Chip oscillator output	On-Chip oscillator discharge
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	6	threshold
1	7	trigger
2	11	reset_n
3	10	control
4	9	out

ua#	analog#	Description
5	8	discharge

TinyQV Risc-V SoC

by **Michael Bell**

`0199` `24 MHz` `Analog Project`

`git@github.com:MichaelBell/ttgf0p3-tinyQV.git`

"A Risc-V SoC for Tiny Tapeout"

How it works

TinyQV is a small Risc-V SoC, implementing the RV32EC instruction set plus the Zcb and Zicond extensions, with a couple of caveats:

- Addresses are 28-bits
- Program addresses are 24-bits
- gp is hardcoded to 0x1000400, tp is hardcoded to 0x8000000.

Instructions are read using QSPI from Flash, and a QSPI PSRAM is used for memory. The QSPI clock and data lines are shared between the flash and the RAM, so only one can be accessed simultaneously.

Code can only be executed from flash. Data can be read from flash and RAM, and written to RAM.

Many of the peripherals making up the SoC are contributed by the Tiny Tapeout community!

This version also includes Matt Venn's R2R DAC as an additional peripheral

Address map

Address range	Device
0x0000000 - 0x0FFFFFF	Flash
0x1000000 - 0x17FFFFF	RAM A
0x1800000 - 0x1FFFFFF	RAM B
0x8000000 - 0x8000033	DEBUG
0x8000034 - 0x8000037	DAC
0x8000040 - 0x800007F	GPIO
0x8000080 - 0x80000BF	UART
0x80000C0 - 0x80001FF	User peripherals 3-7
0x8000400 - 0x800043F	Simple user peripherals 0-3
0xFFFFF00 - 0xFFFFF07	TIME

DEBUG

Register	Address	Description
ID	0x8000008 (R)	Instance of TinyQV: 0x41 (ASCII A)
SEL	0x800000C (R/W)	Bits 6-7 enable peripheral output on the corresponding bit on out6-7, otherwise out6-7 is used for debug.
DEBUG_UART_DATA	0x8000018 (W)	Transmits the byte on the debug UART
STATUS	0x800001C (R)	Bit 0 indicates whether the debug UART TX is busy, bytes should not be written to the data register while this bit is set.

See also debug docs

DAC

Register	Address	Description
DAC	0x80000034 (RW)	Output level from the DAC, range 0-255.

TIME

Register	Address	Description
MTIME_DIVIDER	0x800002C	MTIME counts at clock / (MTIME_DIVIDER + 1). Bits 0 and 1 are fixed at 1, so multiples of 4MHz are supported.
MTIME	0xFFFFF00 (RW)	Get/set the 1MHz time count
MTIMECMP	0xFFFFF04 (RW)	Get/set the time to trigger the timer interrupt

This is a simple timer which follows the spirit of the Risc-V timer but using a 32-bit counter instead of 64 to save area. In this version the MTIME register is updated at 1/64th of the clock frequency (nominally 1MHz), and MTIMECMP is used to trigger an interrupt. If MTIME is after MTIMECMP (by less than 2^30 microseconds to deal with wrap), the timer interrupt is asserted.

GPIO

Register	Address	Description

OUT	0x8000040 (RW)	Control for out0-7 if the GPIO peripheral is selected
IN	0x8000044 (R)	Reads the current state of in0-7
AUDIO_FUNC_SEL	0x8000050 (RW)	Audio function select for uo7
FUNC_SEL	0x8000060 - 0x800007F (RW)	Function select for out0-7

Function Select	Peripheral
0	Disabled
1	GPIO
2	UART
3 - 15	User peripheral 3-15
16 - 31	User byte peripheral 0-15
32 - 39	User peripheral 16-23

Audio function select	Peripheral
0-3	PSRAM B enabled
4	5 PWL Synth out 7
5	4 Pulse Transmitter out 7
6	?
7	18 Matt PWM out 7

UART

Register	Address	Description
TX_DATA	0x8000080 (W)	Transmits the byte on the UART
RX_DATA	0x8000080 (R)	Reads any received byte
TX_BUSY	0x8000084 (R)	Bit 0 indicates whether the UART TX is busy, bytes should not be written to the data register while this bit is set. Bit 1 indicates whether a received byte is available to be read.
DIVIDER	0x8000088 (R/W)	13 bit clock divider to set the UART baud rate

RX_SELECT	0x800008C (R/W)	1 bit select UART RX pin: `ui_in[7]` when low (default), `ui_in[3]` when high

How to test

Load an image into flash and then select the design.

Reset the design as follows:

- Set rst_n high and then low to ensure the design sees a falling edge of rst_n. The bidirectional IOs are all set to inputs while rst_n is low.
- Program the flash and leave flash in continuous read mode, and the PSRAMs in QPI mode
- Drive all the QSPI CS high and set SD1:SD0 to the read latency of the QSPI flash and PSRAM in cycles. SD2 selects whether half a cycle is subtracted from the read latency by driving the SPI clock on the negative edge.
- Clock at least 8 times and stop with clock high
- Release all the QSPI lines
- Set rst_n high
- Set clock low
- Start clocking normally

At the target 24MHz clock a read latency of 1.5 is probably best (SD2:SD0 = 0b110). The maximum supported latency is 3.

The above should all be handled by some MicroPython scripts for the RP2 on the TT demo PCB.

Build programs using the customised toolchain and the tinyQV-sdk, some examples are here.

External hardware

The design is intended to be used with this QSPI PMOD on the bidirectional PMOD. This has a 16MB flash and 2 8MB RAMs.

The UART is on the correct pins to be used with the hardware UART on the RP2040 on the demo board.

It may be useful to have buttons to use on the GPIO inputs.

Project Pinout

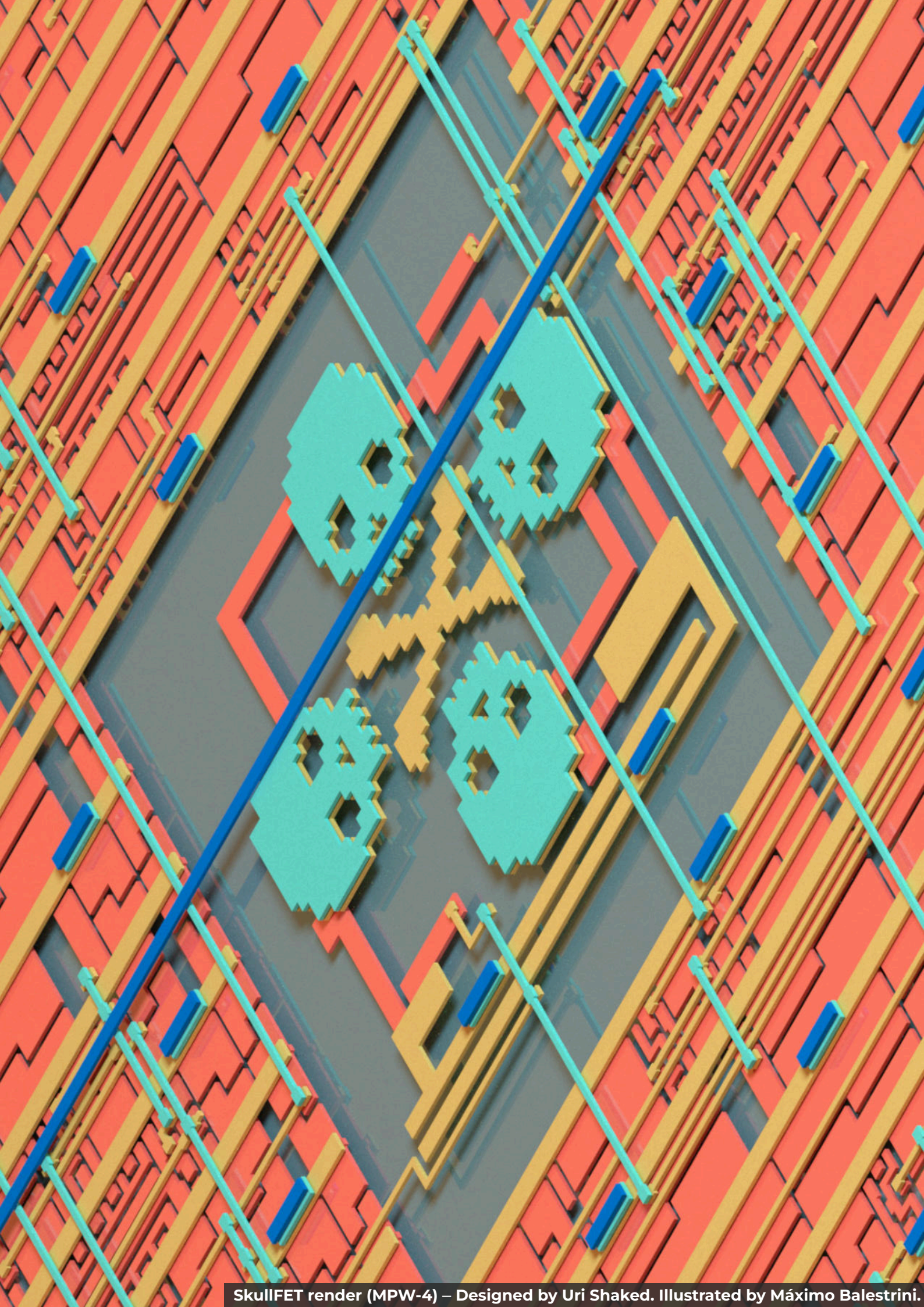
Digital Pins

#	Input	Output	Bidirectional
0	Interrupt 0	UART TX	Flash CS
1	Interrupt 1	UART RTS	SD0
2	SPI MISO	SPI DC	SD1

#	Input	Output	Bidirectional
3	1 MHz clock for time	SPI MOSI	SCK
4	Game controller latch	SPI CS	SD2
5	Game controller clock	SPI SCK	SD3
6	Game controller data	Debug UART TX	RAM A CS
7	UART RX	Debug signal / PWM	RAM B CS / PWM

Analog Pins

ua#	analog#	Description
0	0	DAC output
1	4	—
2	1	—
3	3	—
4	2	—



SkullFET render (MPW-4) – Designed by Uri Shaked. Illustrated by Máximo Balestrini.

256x8 SRAM

by **Julia Desmazes**

`256` `50 MHz` `HDL Project`

`github.com/Essenceia/tt_gf_ocd_sram_test`

"256x8 SRAM for testing"

Open Circuit Design 256x8 SRAM for gf180 test.

How it works

SRAM is entirely cleared on reset.

Although this is technically an 256x8 SRAM only the top 128 entries are addressable. There are 6 bits of address, 8 bits of input data and 8 bits of output data. De-assert `gwen` to write, else SRAM will be in read mode.

Beware, incomming data/write enable/address are flopped once before the SRAM and outgoing data is also flopped once before reaching the `uo_out` pins.

Pinout

From SRAM documentation:

Signal	Direction	Description
CLK	Input	Clock for the memory. Rising edge triggers operation. All inputs are latched at rising edge of the clock signal.
CEN	Input	Memory Enable, Active Low. When CEN is Low, the memory is enabled. When CEN input is High, the memory is deactivated but internal states are retained. CEN must be high before 1st running cycle.
A[7:0]	Input	Address Input. This Address input port is used to address the location to be written during the write cycle and read during the read cycle.
GWEN	Input	Write Enable Input. The RAM is in write cycle when GWEN is low. The RAM is in read cycle when GWEN is high.
WEN[7:0]	Input	Bit Write Mask, Active Low. When the memory is in the write cycle, selectively write into individual outputs are controlled by WEN[7:0]. For example, if

		CEN, GWEN, WEN[0] are low and WEN[7:1] are high, only D[0] will write into the addressed location and D[7:1] will be ignored during CLK low to high transition.
D[7:0]	Input	Data input bus. The data input bus is used to write data into the memory location specified by address input port during the write cycle.
Q[7:0]	Output	Data output bus. It outputs the contents of the memory location addressed by the Address Input signals.
VDD	Power	Power pin.
VSS	Ground	Ground pin.

How to test

See above.

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	addr[0]	data_rd[0]	data_wr[0]
1	addr[1]	data_rd[1]	data_wr[1]
2	addr[2]	data_rd[2]	data_wr[2]
3	addr[3]	data_rd[3]	data_wr[3]
4	addr[4]	data_rd[4]	data_wr[4]
5	addr[5]	data_rd[5]	data_wr[5]
6	addr[6]	data_rd[6]	data_wr[6]
7	gwen	data_rd[7]	data_wr[7]

Abad2048

by **Alexander Co Abad**

0258 | 25.175 MHz | HDL Project

`github.com/alexandercoabad/Abad_2048`

"Colorful 2048 game uploaded successfully to the TT FPGA with game pmod!"

How it works

2048 game Video: https://www.linkedin.com/posts/alexander-co-abad-79445767_i-managed-to-replicate-the-2048-game-and-activity-7471442185091981313-8tYN?utm_source=share&utm_medium=member_desktop&rcm=ACoAAA4gaKABRk1KxBdDQFTnsu-IalkZq-7v9R4

Link to the VGA Playground: https://vga-playground.com/?repo=https://github.com/alexandercoabad/Abad_2048

How to test

Use the arrow keys on your keyboard or your game controller if you use the FPGA! Link to the VGA Playground: https://vga-playground.com/?repo=https://github.com/alexandercoabad/Abad_2048

External hardware

VGA FPGA

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	left	R1	—
1	right	G1	—
2	up	B1	—
3	down	VSync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	start/reset	HSync	—

GF BGR

by **Rahul Bhagwat**

0260 Analog Project

github.com/bhagwat-rahul/ttgf180-bandgap-reference

"*Bandgap Reference*"

How it works

Outputs a stable reference voltage over a temperature range of −40C - 125C (varies by about 1% over that range in simulation)

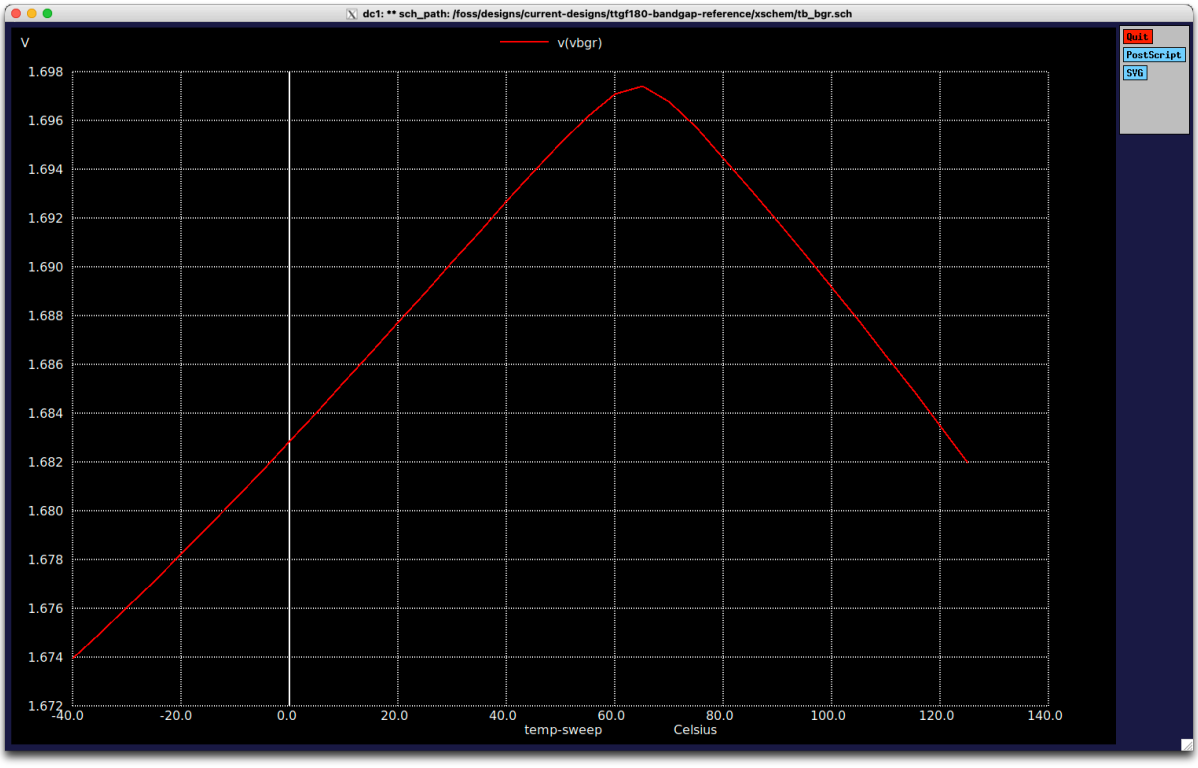
There is no supply noise rejection or trimming in this design, it's a very simple design for providing a stable reference voltage.

How to test

Apply 3.3V VAPWR and plot a graph of voltage over temperature for analog out[0] i.e. VBGR.

Confirm that variability is under 1% from −40C - 125C. Under simulation this range is from 1.674V - 1.697V for the typical corner.

Here's how the graph for vbgr looks.



External hardware

1. 3.3V bench power supply
2. Digital Multimeter (6.5 digit)
3. Thermocouple (To measure temp)
4. Temperature chamber (Some way to vary chip temperature)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	16	VBGR

VGA clock render – Designed by Matt Venn. Illustrated by Máximo Balestrini.

Quadrature VCO

by **Sylvain Munaut**

0262 Analog Project

github.com/smunaut/tt_tnt_gf_vco

"Quadrature VCO with nominal target frequency of 240 MHz"

How it works

It's a 4 stage differential delay VCO. A comparator taps two of the stage to create a 4 phase single ended output.

Design nominal frequency is 240 MHz although the typical tuning range should be from 120 MHz to 500 MHz.

How to test

Power up, apply control voltage, observe oscillation.

External hardware

- Programmable precision voltage source
- Oscilloscope

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	out_0	—
1	—	out_270	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	18	out_180
1	19	out_90
2	20	control

Philippine flag waving

by **Alexander Co Abad**

0288 | 25.175 MHz | HDL Project

`github.com/alexandercoabad/PILIPINAS_waving`

"Philippine flag waving in a DVD-like screensaver with PILIPINAS text in a 7-segment style at the bottom"

How it works

Philippine flag waving

How to test

Philippine flag waving

External hardware

VGA

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	R1	—
1	—	G1	—
2	—	B1	—
3	—	VSync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	—	HSync	—

Analog 4 bit Current DAC

by **Kiterai**

0290 Analog Project

github.com/Kiterai/ttgf-practice1

"Simple 4 bit current output DAC"

How it works

A simple 4bit current DAC

How to test

input 4 bit signal into digital 0-3 pin to choose output current

External hardware

Resistive load on analog 0 pin

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	bit 0	—	—
1	bit 1	—	—
2	bit 2	—	—
3	bit 3	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	13	DAC current sink output

VGA clock render – Designed by Matt Venn. Illustrated by Máximo Balestrini.

3.3V Folded Cascode OTA

by **Pranay Arvind Patil** & **Nithin Purushothama**

0292 Analog Project

github.com/chennakeshavadasa/LPCAS_TTGF0P3_TP1

"NMOS Input pair based Folded Cascode OTA"

TTGF0P3_LPCAS_FC_OTA

Overview

TTGF0P3_LPCAS_FC_OTA is a low-power Folded Cascode Operational Transconductance Amplifier (OTA) designed in the TTGF0P3 process. The amplifier is optimized for low-current operation while maintaining a large input common-mode range and output swing, making it suitable for a wide variety of analog signal-processing applications.

Key Features

- Folded Cascode architecture
- Low power consumption (15 µA typical)
- Typical Unity Gain Bandwidth (UGB): 100 kHz
- Wide input common-mode range
- Output swing approximately 0.5 V to 3.0 V
- Single-ended output
- Bias current programmable through external resistor
- Stable for capacitive loads up to approximately 50 pF
- Suitable building block for amplifiers, filters, DAC buffers, sensor interfaces, and analog computation blocks

---

Architecture and Circuit Operation

Folded Cascode Topology

The OTA uses a PMOS differential input pair combined with an NMOS folded-cascode structure.

Input Stage

The differential pair consists of:

- VINP (positive input)
- VINN (negative input)

The differential pair converts the input voltage difference into differential currents.

Advantages of using a PMOS input pair:

- Improved operation near ground
- Larger usable common-mode input range
- Lower flicker noise contribution

Folded Branch

The differential currents generated by the PMOS input pair are folded into NMOS devices.

This folded structure provides:

- High gain
- Increased output resistance
- Better headroom compared to telescopic cascode architectures
- Operation at relatively low supply voltages

The folded nodes transfer the signal current from the input stage into the output branch while preserving gain.

Cascode Gain Enhancement

Multiple cascode devices are used throughout the signal path.

Benefits include:

- Increased output resistance
- Increased intrinsic gain
- Better PSRR
- Improved isolation between stages

The gain enhancement is achieved without requiring additional gain stages, simplifying frequency compensation.

Current Mirror Load

The folded currents are mirrored and combined to produce a single-ended output at:

- VOUT

The current mirrors perform:

- Differential-to-single-ended conversion
- Gain generation
- Bias current distribution

Bias Network

The OTA uses an externally generated bias current.

The user must connect:

- A resistor between VDD and VBIAS

Recommended value:

- 2 MΩ

This resistor generates the reference current which is mirrored throughout the OTA.

Typical operating current:

- 15 µA

Changing the resistor changes the OTA bias current:

Bias Resistor	Approximate Effect
2 MΩ	Nominal operation
1 MΩ	Higher bias current
500 kΩ	Maximum recommended bias current

Increasing bias current generally results in:

- Higher bandwidth
- Higher slew rate
- Higher power consumption

Pin Description

Pin	Name	Description
ua[0]	VINN	Inverting input
ua[1]	VINP	Non-inverting input
ua[2]	VBIAS	External bias resistor connection
ua[3]	VOUT	OTA output

Power Supply

Recommended operating supply:

Parameter	Value
VDD	3.3 V
GND	0 V

Typical Connections

Biasing

Connect:

- 2 MΩ resistor between VDD and VBIAS

Example:

```
VDD
 |
[2 MΩ]
 |
VBIAS
```

Capacitive Load

Maximum recommended capacitive load:

- 50 pF

For characterization purposes:

- External capacitor ≈ 20 pF
- Probe/parasitics ≈ 30 pF

Total load:

```
CL ≈ 50 pF
```

Important Notes

Do NOT

- Short VOUT directly to GND
- Exceed recommended capacitive loading
- Leave VBIAS floating
- Apply input voltages outside supply rails

Recommended

- Set current limits on laboratory SMUs
- Start with a 2 MΩ bias resistor
- Verify supply current before performing measurements

Basic Functional Tests

1. Voltage Follower (Buffer)

Connect:

```
VINP = Input Signal
VINN = VOUT
```

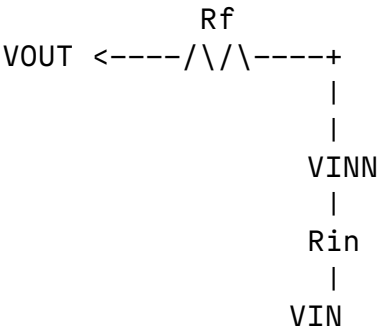
Expected:

- Gain ≈ 1
- Demonstrates stability and bandwidth

---

2. Inverting Amplifier

Connect:



Gain:

```
A_v = −Rf/Rin
```

---

3. Non-Inverting Amplifier

Gain:

```
A_v = 1 + Rf/Rg
```

Useful for:

- Sensor signal conditioning
- Analog front ends

---

4. Comparator Experiment

Operate OTA open-loop:

```
VINP > VINN  → VOUT rises
VINP < VINN  → VOUT falls
```

Useful for:

- Offset measurements
- Switching behavior characterization

Advanced Characterization Tests

5. Open-Loop Gain Measurement

Measure:

- DC gain
- Gain roll-off
- Dominant pole

Useful plots:

- Magnitude response
- Phase response

---

6. Unity-Gain Stability

Configure as a voltage follower.

Measure:

- Phase margin
- Settling time
- Ringing

---

7. Common-Mode Sweep

Sweep:

```
VINP = VINN
```

Measure:

- Input common-mode range
- Output swing limits

---

8. Output Swing Measurement

Sweep differential input slowly and record:

```
VOUT_MIN
VOUT_MAX
```

Expected output swing:

```
~0.5 V to ~3 V
```

---

9. Monte Carlo Analysis

Characterize:

- Offset voltage

- Gain variation
- GBW variation
- Bias current variation

Useful for mismatch verification.

---

10. Process Corner Verification

Run:

- TT
- FF
- SS
- FS
- SF

Verify:

- Gain
- UGB
- Phase Margin
- Current Consumption

---

11. Temperature Sweep

Recommended:

```
−40°C
25°C
85°C
125°C
```

Measure:

- Offset drift
- Gain drift
- Bandwidth variation

---

12. PSRR Measurement

Inject ripple into:

- VDD

Measure:

```
PSRR+ = ΔVDD / ΔVOUT
```

Useful plots:

- PSRR vs Frequency

---

13. CMRR Measurement

Apply common-mode excitation:

```
VINP = VINN
```

Measure:

- Common-mode gain
- Differential gain

Compute:

```
CMRR = Ad / Acm
```

---

14. Noise Analysis

Measure:

- Input referred noise
- Output noise density
- Integrated RMS noise

Useful for sensor applications.

Practical Application Examples

This OTA can be used as the active element in many analog circuits.

Filters

Active Low-Pass Filter
Applications:

- Sensor conditioning
- Audio filtering
- Anti-aliasing

Active High-Pass Filter
Applications:

- DC removal
- AC coupling

Band-Pass Filter
Applications:

- Communication systems

- Tone detection

State Variable Filter
Applications:

- Analog signal processing
- Tunable filtering

Data Converters

R-2R DAC Buffer
Use the OTA as:

- Output buffer
- Reconstruction amplifier

Benefits:

- High input impedance
- Improved DAC drive capability

SAR ADC Front-End Buffer
Applications:

- Sample-and-hold driving
- Capacitive DAC buffering

Sensor Interfaces

Suitable for:

- Temperature sensors
- Resistive sensors
- Photodiodes
- Capacitive sensing systems

Signal Conditioning

Applications:

- Instrumentation amplifiers
- Differential-to-single-ended conversion
- Analog preprocessing

Peak Detector

Build:

- Precision peak detector
- Envelope detector

Applications:

- AM demodulation
- Signal monitoring

Integrator

Applications:

- Analog computation
- PID controllers
- Sigma-delta loops

Differentiator

Applications:

- Edge detection
- High-frequency enhancement

Oscillator Building Block

Can be used in:

- Wien bridge oscillators
- Quadrature oscillators
- Relaxation oscillators

Current-to-Voltage Converter

Useful for:

- Photodiodes
- Current-output sensors
- Electrochemical sensors

Suggested Characterization Results to Include

The following plots are highly recommended for documentation:

DC Characterization

- Transfer curve
- Input common-mode sweep
- Output swing plot

AC Characterization

- Open-loop gain
- UGB
- Phase margin

Transient Characterization

- Step response
- Settling behavior
- Slew rate

Statistical Verification

- Monte Carlo offset histogram
- Gain distribution
- Current distribution

PVT Verification

- Corner sweeps
- Temperature sweeps

Supply Rejection

- PSRR vs frequency

Common Mode Performance

- CMRR vs frequency

Noise

- Input referred noise
- Output noise density

Typical Operating Conditions

Parameter	Typical Value
Supply Voltage	3.3 V
Bias Resistor	2 MΩ

Current Consumption	15 µA
Capacitive Load	≤ 50 pF
Output Swing	0.5 V to 3.0 V
Topology	Folded Cascode OTA
Output Type	Single Ended
UGB	100 kHz

Authors

- Nithin P
- Pranay Patil
- LPCAS Group, IIT Gandhinagar

This OTA is intended as a reusable low-power analog building block for educational, research, and mixed-signal integrated circuit design projects.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	12	VINN
1	13	VINP
2	14	VBIAS
3	15	VOUT

Oscillating Bones

by **Uri Shaked**

0294 Analog Project

`github.com/urish/ttgf-oscillating-bones`

"A stylish SkullFET ring oscillator (~120 MHz) with an 8-bit /2../256 divider"

How it works

A stylish ring oscillator built from **SkullFET** transistors — MOSFETs hand-drawn in the shape of skulls. A chain of 21 SkullFET inverters forms a ring oscillator that generates a 120 MHz square wave; an **8-bit ripple divider** then produces /2 through /256 taps on `uo_out[0..7]`. The raw oscillation is brought out on the analog pin `ua[0]` through **one more SkullFET wired as a buffer**, so an external probe/load on `ua[0]` can't pull the ring frequency.

This is the **gf180mcu** (GlobalFoundries 180nm) port of the design, migrated from the original IHP sg13g2 version. The SkullFETs are **3.3V devices** running directly on the 3.3V core supply (VDPWR). The skull artwork is preserved verbatim from the original; only the layer stack, device implants and feature sizes were retargeted to gf180mcuD (a uniform 1.45× scale clears the 180nm width/spacing/gate rules; grid-snap, exact-cut and metal-slot passes then make it KLayout sign-off-clean — see `scripts/remap_to_gf180.py`).

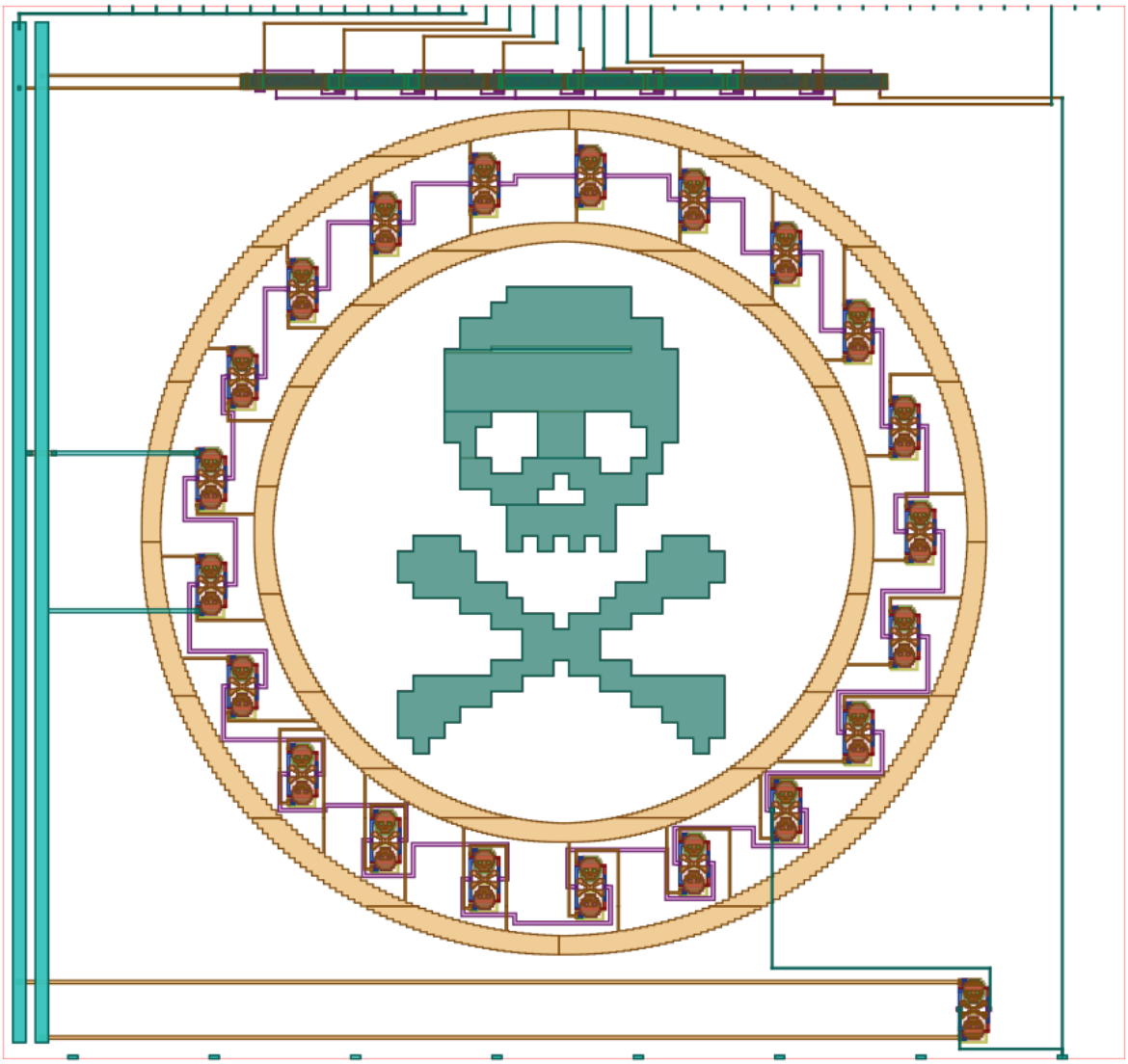


Figure 294.1: Layout

Pin	Signal	Post-layout frequency
ua[0]	osc_out (buffered raw 3.3V oscillation)	120 MHz
uo_out[0]	osc_div_2	60 MHz
uo_out[1]	osc_div_4	30 MHz
uo_out[2]	osc_div_8	15 MHz
uo_out[3]	osc_div_16	7.5 MHz
uo_out[4]	osc_div_32	3.7 MHz
uo_out[5]	osc_div_64	1.9 MHz
uo_out[6]	osc_div_128	0.94 MHz
uo_out[7]	osc_div_256	0.47 MHz

(`uo_out[0]` is the LSB / ÷2, `uo_out[7]` the MSB / ÷256.) All unused outputs — `uio_out[7:0]` and the output-enables `uio_oe[7:0]` — are tied low in the macro, so the bidirectional pads stay in input mode.

Post-layout simulation (extract the hardened GDS with magic, simulate with the gf180mcuD ngspice models — run `make sim`): the ring oscillates **rail-to-rail at 120 MHz** and the 8-stage divider produces clean **/2 .. /256** taps. The testbench supplies only VDPWR/VGND and the substrate bias — it does **not** force any std-cell rail or device well, so the result reflects the actual extracted connectivity (every pfet body is tied to VDPWR through its n-well tap, every std-cell rail is strapped to VDPWR/VGND — nothing floats). With the buffer, a swept capacitive load on `ua[0]` (0→10 pF) leaves the ring frequency **unchanged** at 120 MHz; without it the same load would drop the ring 25–60 %.

Simulation results

`osc_out` (the buffered 120 MHz oscillation on `ua[0]`) and the first three divider taps, captured from the post-layout netlist (`make plot`):

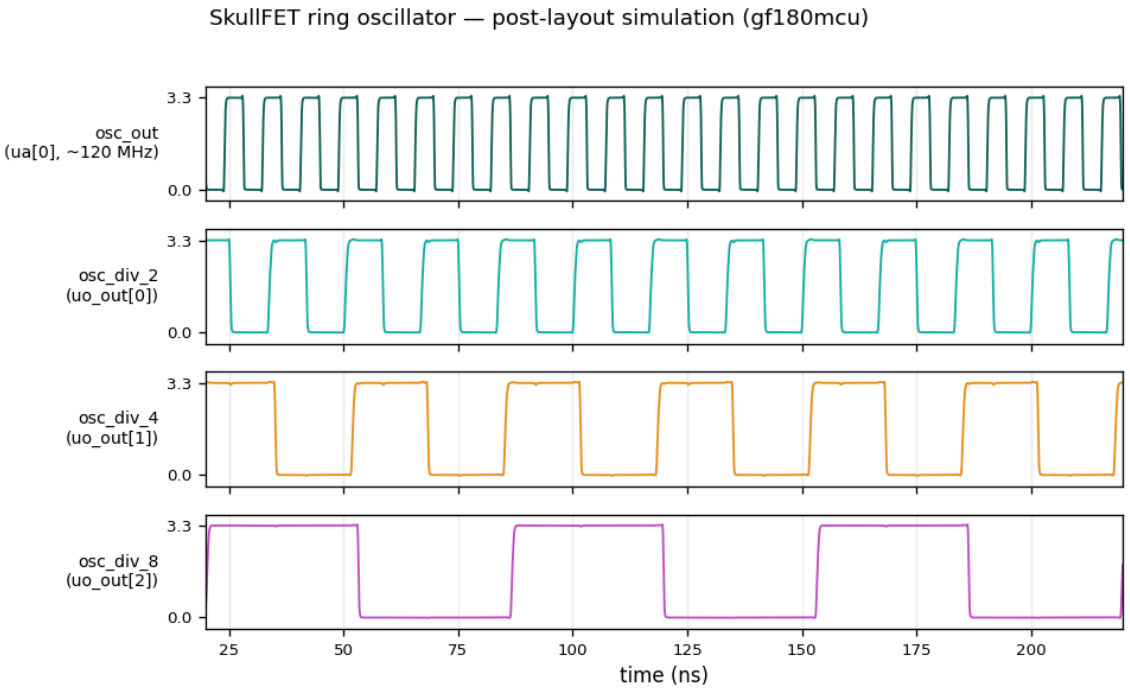


Figure 294.2: Simulation results

Reset

`rst_n` (active-low) resets **only the divider** — it asynchronously clears all eight flip-flops, so while `rst_n` is held low the divided taps `uo_out[0..7]` sit at 0. The 21-stage ring oscillator has no reset and free-runs whenever the design is powered, so `ua[0]` keeps oscillating at full speed even during reset. Release

`rst_n` (high) and the divider starts counting from a known phase. `clk` and `ena` are not used — the ring self-clocks the divider.

How to test

Connect an oscilloscope to a low divider tap such as **`uo_out[7]` (÷256, 0.5 MHz)** or **`uo_out[4]` (÷32, 3.7 MHz)** — scope-friendly. The fast taps and the buffered `ua[0]` (120 MHz) exceed typical GPIO bandwidth, so the analog pin **`ua[0]`** is the place to look for the raw oscillation. Note that `uo_out[0..7]` stay at 0 until you release `rst_n`.

External hardware

None — just an oscilloscope.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	osc_div_2	—
1	—	osc_div_4	—
2	—	osc_div_8	—
3	—	osc_div_16	—
4	—	osc_div_32	—
5	—	osc_div_64	—
6	—	osc_div_128	—
7	—	osc_div_256	—

Analog Pins

ua#	analog#	Description
0	14	osc_out

2048 sliding tile puzzle game (VGA)

by **Uri Shaked**

`0356` `25.175 MHz` `HDL Project`

`github.com/urish/tt-2048-game`

"Slide numbered tiles on a grid to combine them to create a tile with the number 2048."

How it works

2048 is a single-player sliding tile puzzle video game. Your goal is to slide numbered tiles on a grid to combine them and create a tile with the number 2048. The game is won when a tile with the number 2048 appears on the board, hence the name of the game. The game is lost when the board is full and no more moves can be made.

The game is played on a 4x4 grid, with numbered tiles that slide when a player moves them using `ui_in` pins or using a SNES compatible controller along with the Gamepad Pmod.

The game starts with two tiles with the number 2 on the board. The player can move the tiles in four directions: up, down, left, and right. When the player moves the tiles in a direction, the tiles slide as far as they can in that direction until they hit the edge of the board or another tile. If two tiles with the same number collide, they merge into a single tile with the sum of the two numbers. The resulting tile cannot merge with another tile again in the same move.

How to test

Use the `ui_in` pins to move the tiles on the board:

`ui_in` pin	Direction
0	Up
1	Down
2	Left
3	Right

Or use a SNES compatible controller along with the Gamepad Pmod. Both the d-pad and the face buttons can be used for movement:

D-pad	Face button	Direction
Up	X	Up

Down	B	Down
Left	Y	Left
Right	A	Right

After resetting the game, you will see a jumping “2048” animation on the screen. Press any of the `ui_in[3:0]` pins (or the gamepad buttons) to start the game. The game will start with two tiles with the number 2 on the board. Use the `ui_in` pins (or the gamepad buttons) to move the tiles in the desired direction. The game will end when the board is full and no more moves can be made.

The game offers two color themes: modern and retro. You can switch between the two themes using the `select` button on the gamepad or by setting both `ui_in[4]` and `ui_in[5]` to 1.

Setting `ui_in[7]` to 1 will enter unit test mode. In this mode, the game displays a colorful rectangle on the top of the screen, and accepts debug commands on the `uio` pins. Check out the test bench for more information.

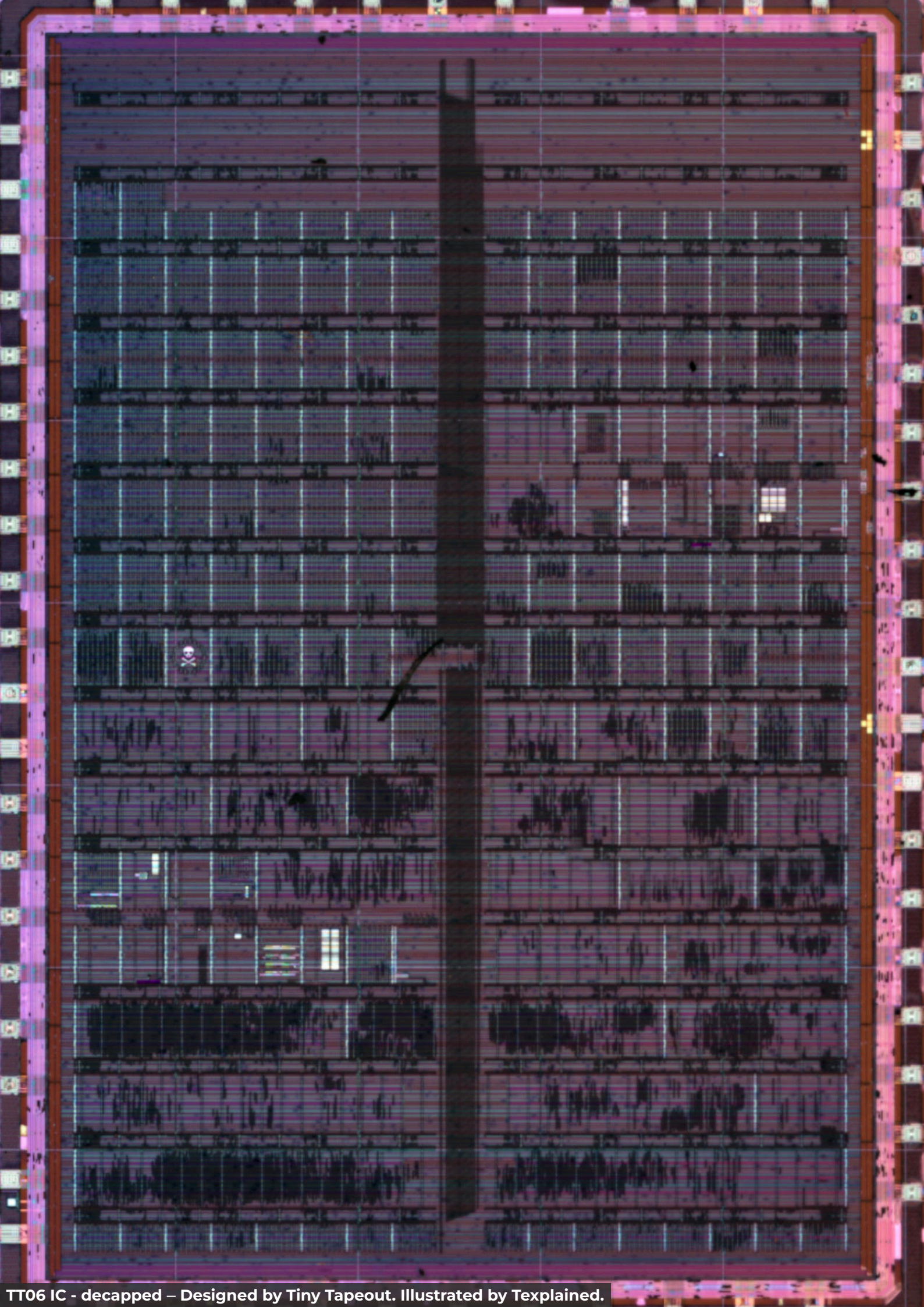
External hardware

- TinyVGA Pmod
- Optional: Gamepad Pmod

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	btn_up	R1	debug_cmd
1	btn_down	G1	debug_cmd
2	btn_left	B1	debug_cmd
3	btn_right	VSync	debug_cmd
4	gamepad_latch	R0	debug_data
5	gamepad_clk	G0	debug_data
6	gamepad_data	B0	debug_data
7	debug_mode	HSync	debug_data



TT06 IC - decapped – Designed by Tiny Tapeout. Illustrated by Texplained.

Hardware Entropy Explorer: UART/SPI TRNG and PUF (Analog)

by **gojimmypi**

358 | 25 MHz | Analog Project

`github.com/gojimmypi/ttgf0p3-analog-UART-FSM-TRNG-Lab`

"UART/SPI-controlled GF180 analog lab with DAC, threshold sampling, TRNG monitors, and measured ua5 passive probe."

How it works

This is the experimental analog project version that enables `ua[0..5]`.

A ring oscillator is implemented at the core of this project as an entropy source for a TRNG (True Hardware Random Number Generator).

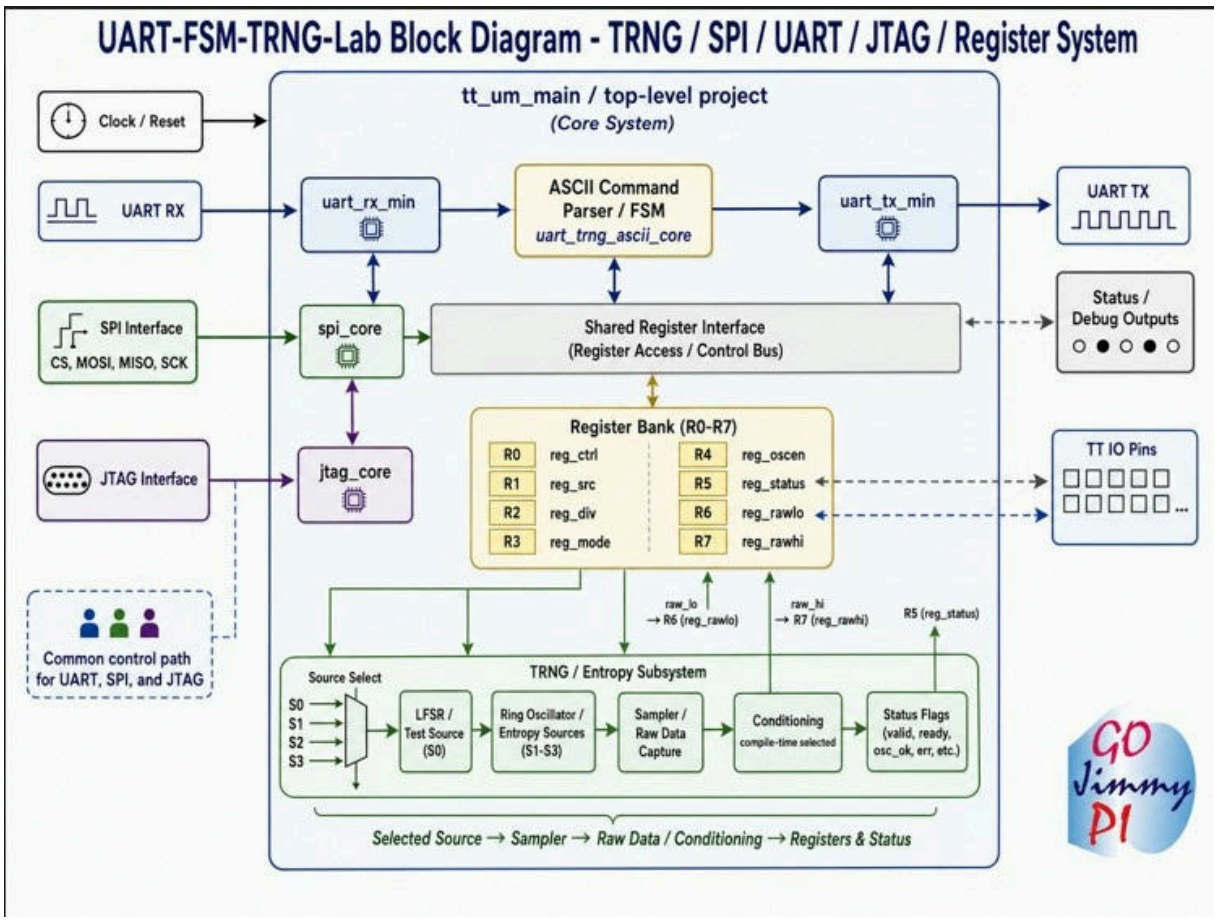


Figure 358.1: UART-FSM-TRNG-Lab-block-diagram.png

This project exposes a UART-controlled interface to a ring-oscillator-based entropy source. A host such as a PC, ESP32, or test script can send simple

ASCII commands over UART to configure internal registers, control the oscillator network, and read back raw entropy samples.

At a high level:

- A bank of ring oscillators generates jitter-based entropy
- A sampling clock (controlled by a divider) captures this behavior
- Control and configuration are managed through memory-mapped registers
- Data and status are read back over the same UART interface

GF 0p3 analog experiment update:

- `info.yaml` enables `analog_pins: 6` so all available Tiny Tapeout analog pins are requested.
- `ua[0]` is an external analog stimulus/noise input sampled by a CMOS threshold.
- `ua[1]` is a 1-bit sigma-delta DAC output. Add an external RC filter to observe an analog voltage.
- `ua[2]` is an external comparator/reference-style input sampled by a CMOS threshold.
- `ua[3]` is a digital monitor mux output for DAC/comparator/probe/TRNG/status observation.
- `ua[4]` is a divider or TRNG-bit monitor output for scope/frequency tests.
- `ua[5]` is a charge/release/sample probe pad for RC, touch, leakage, and PUF-style experiments. The analog GDS patch flow adds a small on-chip Metal4 fringe/pickup structure tied to this pad so the probe has real silicon capacitance/noise/leakage behavior to exercise.
- `R14`/`0xE` reads the sampled analog experiment status through the same UART/SPI register bank.
- `R15`/`0xF` reads the latest `ua[5]` passive-structure threshold/decay timing sample measured by RTL during the release phase.
- The current RTL includes a digital/FPGA-safe analog pad exerciser. The GDS adds one small passive on-chip analog structure on `ua[5]`. This is useful for control-plane testing and post-silicon experiments with external RC/test equipment, but it is not a precision analog macro and cannot be validated by the FPGA bitstream.

Why? The National Institute of Standards and Technology (NIST) notes that random numbers are essential for cryptographic and security applications, and that cryptography makes extensive use of random numbers and random bits, particularly for generating cryptographic keying material.

See presentations:

- NIST Standards on Random Bit Generation slides.
- Why Random Numbers for Cryptography?

SPI vs JTAG Special Note

JTAG is experimental only.

Build / board	Physical setting	`ui_in[4]`	`debug_is_jtag`	Active interface
TT Demoboard	INPUT `SW4` / `IN4` up/off	`0`	`0`	SPI
TT Demoboard	INPUT `SW4` / `IN4` down/on	`1`	`1`	JTAG
ULX3S	`gp4` high / unconnected pull-up	`1`	`0`	SPI
ULX3S	`gp4` pulled low	`0`	`1`	JTAG

For additional related information:

https://gojimmypi.github.io/trng/

https://gojimmypi.github.io/tinytapeout/

External Hardware

It can be helpful to have a TTY-UART USB adapter on hand to interact with the FSM and TRNG on the FPGA or ASIC. This can be used to send commands and read responses from the FSM and TRNG.

Most of the scripts to test assume the external UART. Testing and interactive commands could still be entered via the TT prompt.

FPGA Tests

This project can be tested on an FPGA such as these examples:

- Tiny Tapeout FPGA Development Kit Demoboard in the `ice40` directory.
- ULX3S ECP5 + ESP32 FPGA Development Board in the `ulx3s` directory, and ESP32 SPI Example.

Note that the ring oscillators will not be implemented on the FPGA builds, rather a deterministic Linear-Feedback Shift Register (LFSR) is used in trng_lab_core.v to simulate the TRNG bitstream. In the TT FPGA and ULX3S wrappers the `ua` pins are locally wired into the design but are not routed to physical analog pads, so FPGA testing validates the digital control plane, register control, and deterministic surrogate behavior. Real GF180 pad behavior still needs ASIC silicon or extracted analog simulation.

See the `FPGA_NIST_PRNG_SOURCE` and `FPGA_BASIC_LFSR_RO_TAPS` options in `project_config.v` that are disabled for the TT build.

Commander App Tests

Use the commander.tinytapeout.com to connect to the tt-commander-app

---

How to test

The TT projects usually start in a reset mode = `True`. Connect to TT Breakout (or Demoboard) USB.

Once connected, there should be a Python REPL command prompt.

Don't confuse the TT board serial connection with the external UART.

Ensure all the dip input switches are in the `up` default (off) position.

Select the project, set the clock to 25 MHz, and reset. (see project_reset.py):

```
# select project and reset ttgf
tt.shuttle.tt_um_gojimmypi_ttgfa_UART_FSM_TRNG_Lab.enable()

tt.clock_project_PWM(25000000)
tt.reset_project(True)
tt.reset_project(False)
```

Connect a UART terminal (e.g. PuTTY) to the TT Breakout (or Demoboard) I/O pins with the following connections:

- UART/TTY USB Tx to `IN3/Rx`
- UART/TTY USB Rx to `OUT4/Tx`
- GND to `GND`

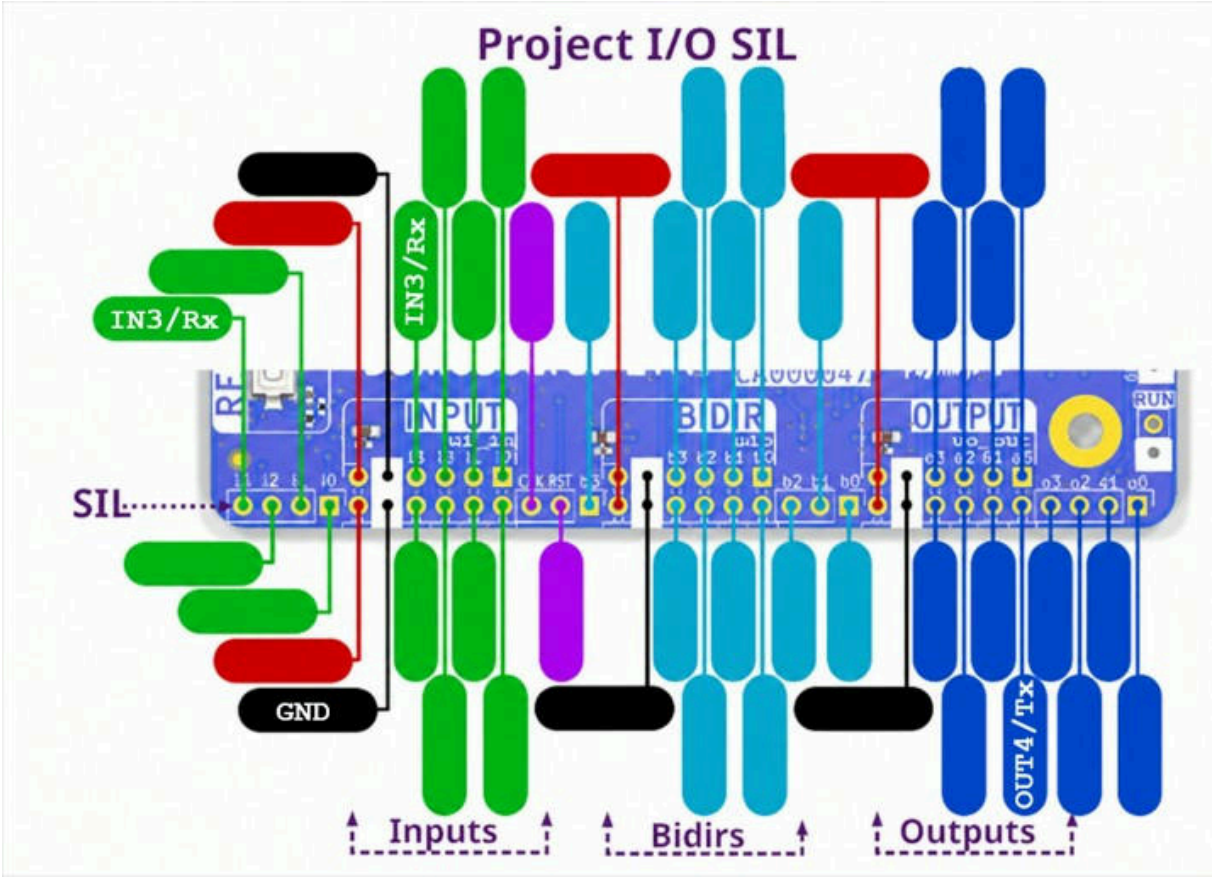


Figure 358.2: PMOD-connector-test1.png

⚠️ ** **CAUTION:** ** Pins are 3v3 and NOT expected to be 5v tolerant.

⚠️ ** **CAUTION:** ** TT IO pins such as `Tx` and `Rx` are likely ** **NOT** ** tolerant to reversal. See TT Discord.

That's the same as shorting them. They're definitely not designed for it, but they won't die immediately either.

Note: `IN3` and `OUT4` are Tiny Tapeout logical signal names, not PMOD physical pin numbers. On the shown PMOD adapter:

- `in3` is PMOD `IO4` /physical pin 4.
- `out4` is PMOD `IO5` / physical pin 7.

Project config:

- `clock_hz: 25000000` in `info.yaml`
- `define PROJECT_CLOCK_HZ 32'd25_000_000` in `src/project_config.v`
- `define PROJECT_UART_BAUD 32'd115_200` in `src/project_config.v`

At a 25 MHz project clock with `PROJECT_UART_BAUD = 115_200`:

- `CLKS_PER_BIT = 25_000_000 / 115_200` = 217
- Terminal baud rate: 115,200

At a 50 MHz project clock, if the design is rebuilt with `PROJECT_CLOCK_HZ = 50_000_000`:

- `CLKS_PER_BIT = 50_000_000 / 115_200` = 434
- Terminal baud rate: 115,200

If the bitstream was built for 25 MHz but the board is actually clocked at 50 MHz, the effective UART baud rate doubles to approximately 230,400 baud.

Terminal session at 25 MHz clock is

- 115,200 baud
- 8 data bits
- No parity
- 1 Stop
- No flow control (Although the default XON/XOFF should also work, but ignored)

Or:

```
stty -F "$PORT" "$BAUD" cs8 -cstopb -parenb -ixon -ixoff -crtscts
raw -echo min 0 time 5
```

Type `V` and press `Enter` to query the version string (if enabled in the build, on by default for TT). Then you can send commands to configure the TRNG and read back entropy samples.

⚠️ The TT Build is Case Sensitive. Although there are case-insensitive settings available for local FPGA builds, they have been disabled for TT ASIC due to observed increased slew and setup violations.

Type `RD` and press enter to view the Build Target ID. The expected value for GF180 ASIC is 42.

Send the appropriate commands to configure and read from the TRNG core. See Register Overview, below.

NIST Validation

NIST has a Resource for Random Bit Generation testing:

Overview

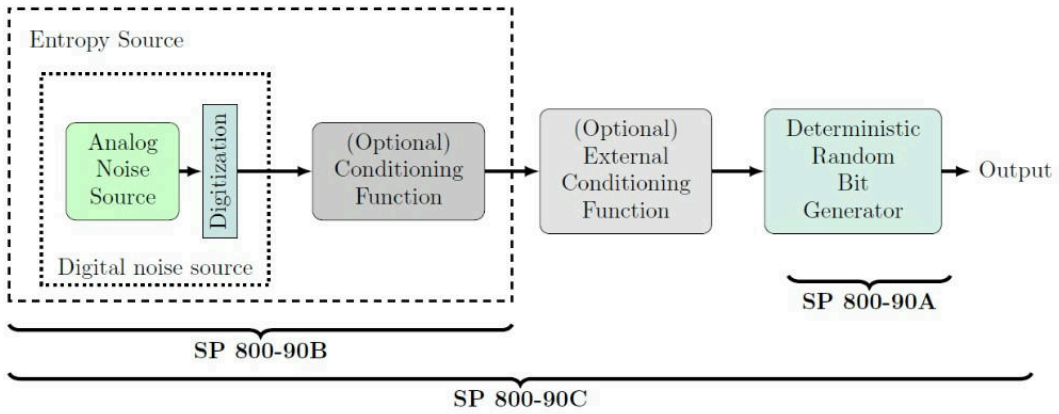
The National Institute of Standards and Technology (NIST) Random Bit Generation (RBG) project focuses on the development and validation of generating random numbers that are essential for cryptographic and security applications.

SP 800-90 Series

The project provides guidelines through the SP 800-90 series, which includes recommendations on deterministic random bit generator (DRBG) mechanisms, entropy sources, and construction principles for RBGs, and has three parts:

- SP 800-90A, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, specifies mechanisms for generating random bits using deterministic methods. NIST is revising SP 800 90A to be consistent with SP 800-90C.
- SP 800-90B, *Recommendation for the Entropy Sources Used for Random Bit Generation*, specifies the design principles and requirements for the entropy sources used by RBGs and the tests for the validation of entropy sources.
- SP 800-90C, *Recommendation for Random Bit Generator (RBG) Constructions*, specifies constructions for the implementation of RBGs.

The following figure explains the relationship of the three parts of the series.



NIST IR 8427, *Discussion on the Full Entropy Assumption of the SP 800 90 Series*, provides technical discussions to support the full entropy definition used in the SP 800 90 series.

Image credit: screen snip from csrc.nist.gov/Projects/random-bit-generation

See the `capture_trng_raw_uart.py` script to capture a binary file of random data from this project, large enough for 100 runs of 1,000,000-bit NIST-style tests:

```
# WSL /dev/ttyS[n] == COM[n] on Windows, other Linux: /dev/
ttyUSB[n], /dev/ttyACM[n], etc

./capture_trng_raw_uart.py  --port /dev/ttyS12  --bytes 16777216
--out trng_raw.bin
```

This script requires a build with `TRNG_BINARY_STREAM` enabled.

The raw output is intended for experimentation and characterization. It is not a certified cryptographic random number generator.

When the optional `define TRNG_CONDITIONED_STREAM` is used in `project_config.v`, the conditioned output can be generated with the `--conditioned` option:

```
./capture_trng_raw_uart.py \
    --port /dev/ttyS12 \
    --bytes 16777216 \
    --out trng_conditioned.bin \
    --fast-baud \
    --conditioned
```

See also:

```
# The official STS package from NIST CSRC:
# https://csrc.nist.gov/CSRC/media/Projects/Random-Bit-Generation/
documents/sts-2_1_2.zip

unzip sts-2_1_2.zip
cd sts-2.1.2
make

#
# or this UNOFFICIAL mirror:
# https://github.com/terrillmoore/NIST-Statistical-Test-Suite.git

cd NIST-Statistical-Test-Suite
./setup.sh
cd sts
make
```

For further testing information see NIST Random Bit Generation RBG - Guide to the Statistical Tests.

Quickstart Simulation

```
cd /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/test

./my_test.sh

./jtag_test.sh
```

Quickstart Testing on TT Demoboard

If all the toolchains are installed:

```
cd /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ice40

source ./env_ice40.sh
./build_and_flash.sh
./project_reset.sh
./run_tests.sh
```

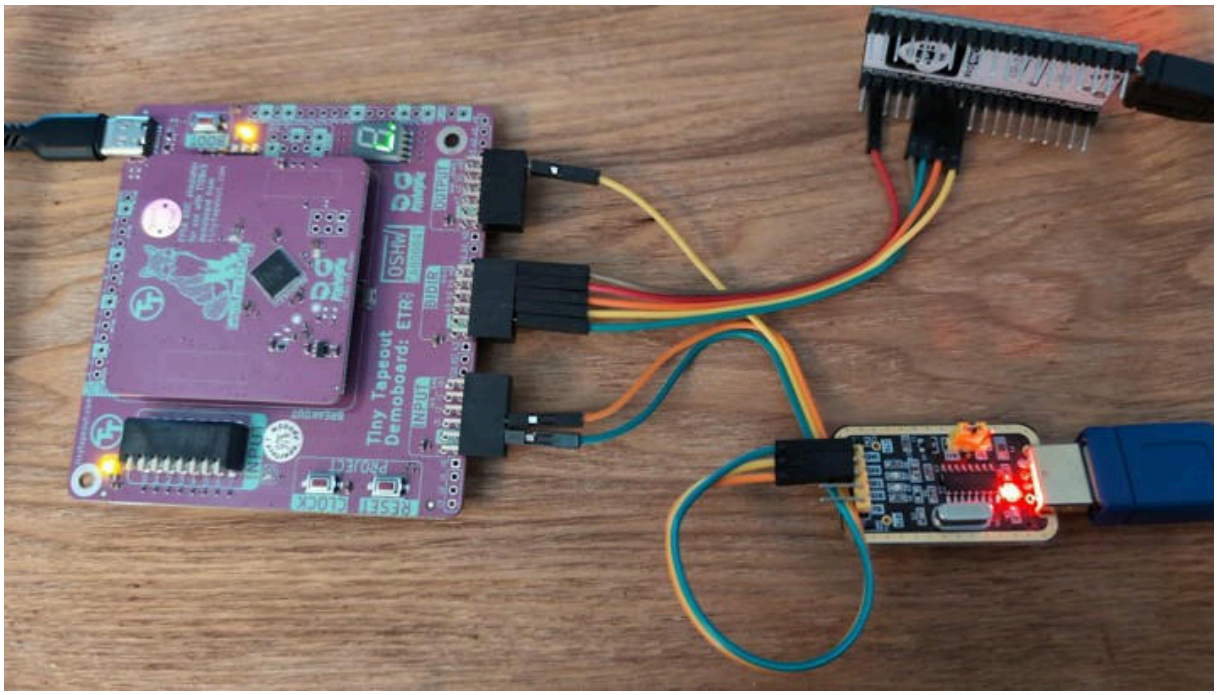


Figure 358.3: TT-Demoboard-SOFT_UART-SOFT-SPI-Wiring.jpg

Sample Soft SPI connected to ESP32 and Soft UART connected to external USB/TTY UART.

Despite the “F” that may be repeatedly displayed on the 7-segment display during testing, that does not indicate failure:

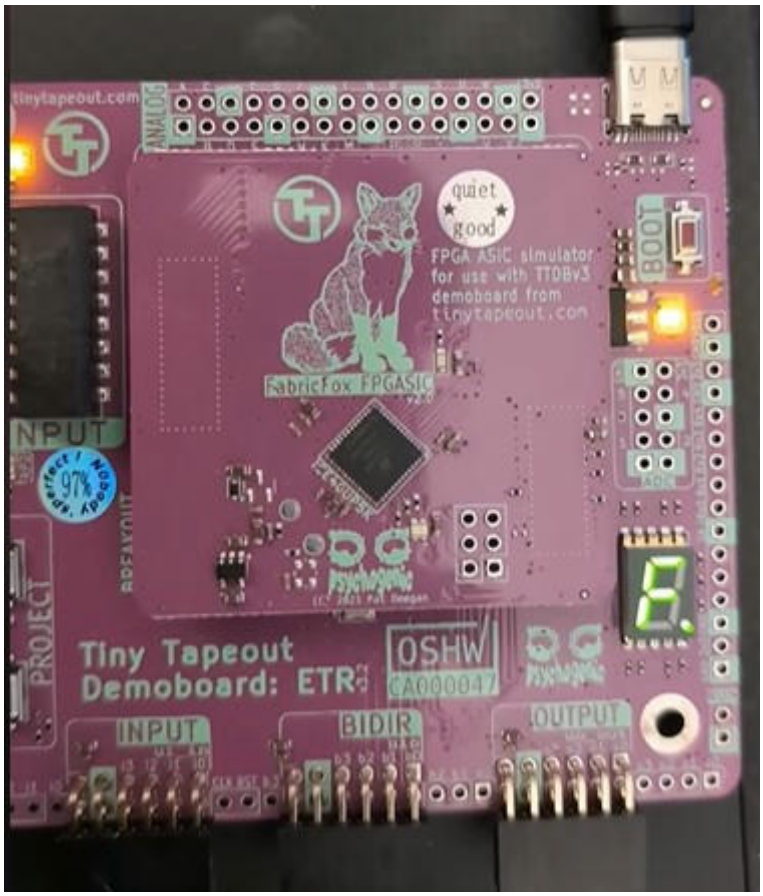


Figure 358.4: Demoboard_F_is_for_Fun_Success.jpg

From youtube.com/shorts/zFnfsl1DQHE

Quickstart Testing on ULX3S

See the `[project]/ulx3s` and `[project]/test-hw` directories.

ULX3S Connections

All pins are 3v3 and assumed to NOT be 5v tolerant.

Soft External UART

⚠️ Do not connect to 5V TTY

- `GND` on `J1` pin 4; Ground connection. Beware of adjacent `3v3` on `J1` pins 1 and 2.
- `GP0` for `Rx` on `J1` pin 6 (connect to external USB/TTY UART `Tx`)
- `GP1` for `Tx` on `J1` pin 8 (connect to external USB/TTY UART `Rx`)

Soft SPI

Select SPI by leaving TT `IN4` up/off, or leaving ULX3S `gp4` high/unconnected/pull-up.

- For TT boards, `INPUT` Dip Switch `IN4` up/off gives `ui_in[4] = 0`, selecting SPI.

- For ULX3S, `gp4` high/unconnected/pull-up gives `shared_spi_jtag_select = 1`, selecting SPI.

Pins are already connected to the on-board ESP32 - but for debugging reference:

- `GND` on `J1` pin 5; Ground connection. Beware of adjacent `3v3` on `J1` pins 1 and 2.
- `GN2` -> (TT `uio[0]`) TMS
- `GP2` -> (TT `uio[1]`) TDI
- `GN3` <- (TT `uio[2]`) TDO
- `GP3` -> (TT `uio[3]`) TCK

See `/ulx3s/ESP32/main/ulx3s_spi_lib.c`

⚠ Do not accidentally wire ESP32 `GPIO2` to PMOD `GP2`. `GPIO2` goes to `GN3`, because it is `MISO/TDO`. Also be careful around `J1`: use pin 5 `GND`, not the adjacent `3v3` pins 1/2.

ULX3S ESP32 SPI Pins

```
#define PIN_NUM_MISO        2
#define PIN_NUM_MOSI        15
#define PIN_NUM_CLK         14
#define PIN_NUM_CS          13
#define SPI_CLOCK_HZ        1000000
```

ESP32 signal	ESP32 GPIO	TT/ PMOD pin	TT signal	JTAG-style name	Direction	Wire
PIN_NUM_CS	GPIO13	GN2	uio[0]	TMS	ESP32 -> TT	Brown
PIN_NUM_MOSI	GPIO15	GP2	uio[1]	TDI	ESP32 -> TT	Red
PIN_NUM_MISO	GPIO2	GN3	uio[2]	TDO	TT -> ESP32	Orange
PIN_NUM_CLK	GPIO14	GP3	uio[3]	TCK	ESP32 -> TT	Yellow
GND	ESP32 GND	J1 pin 5	GND	-	common ground	Green

Stand-alone ESP32 SPI Pins

⚠ Do not use these pins on the ULX3S ESP32.

Disable `IS_ULX3S_ESP32` macro in `ulx3s_spi_lib.c` to use external stand-alone ESP32:

```
#define PIN_NUM_MISO        19
#define PIN_NUM_MOSI        23
#define PIN_NUM_CLK         18
#define PIN_NUM_CS          21
#define SPI_CLOCK_HZ        1000000
```

ESP32 signal	ESP32 GPIO	TT/PMOD pin	TT signal	Direction	Wire
`PIN_NUM_CS`	GPIO21	`GN2`	`uio[0]` / CS / TMS	ESP32 -> TT	Brown
`PIN_NUM_MOSI`	GPIO23	`GP2`	`uio[1]` / MOSI / TDI	ESP32 -> TT	Red
`PIN_NUM_MISO`	GPIO19	`GN3`	`uio[2]` / MISO / TDO	TT -> ESP32	Orange
`PIN_NUM_CLK`	GPIO18	`GP3`	`uio[3]` / SCK / TCK	ESP32 -> TT	Yellow
`GND`	ESP32 GND	`J1` pin 5	`GND`	common ground	Green

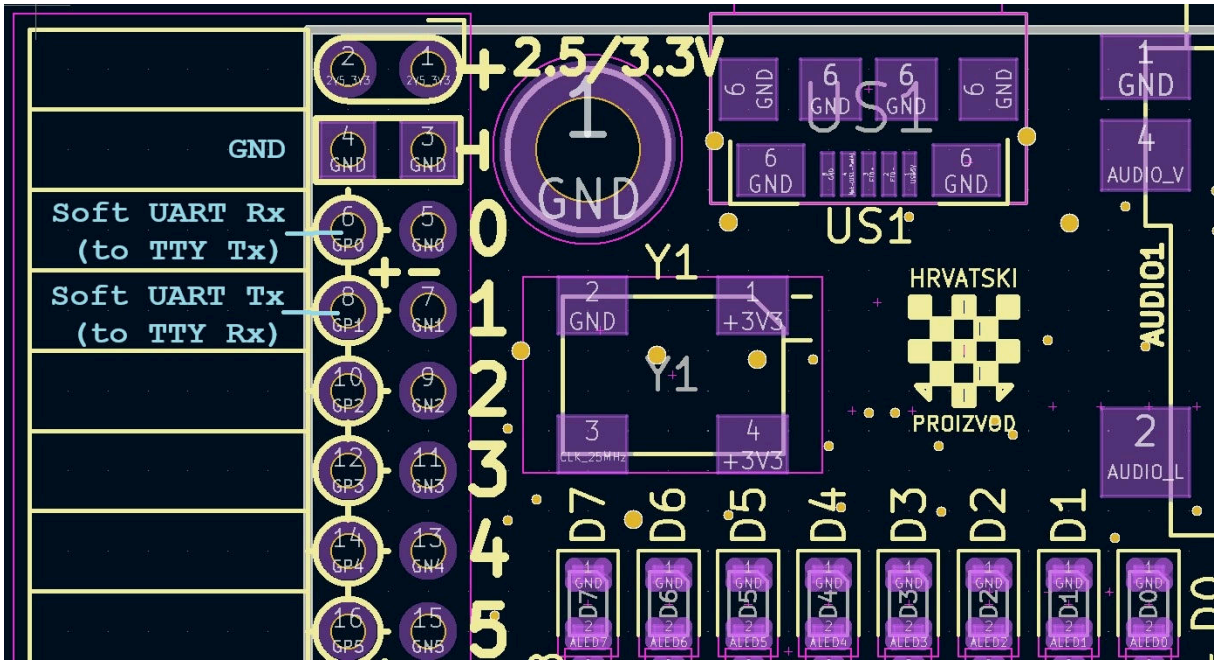


Figure 358.5: ULX3S-Pin-Connections.jpg

Build and run tests from the `./test-hw` directory.

```
cd /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/test-hw

# may need to remove generated file
rm  ../src/_tt_fpga_top.v
```

```
# Edit board version as needed, tested on older v3.0.7:
./run_tests.sh  --with-build  --ulx3s-board-version v307  --ignore-
combinational-warning  --no-warning-pause  --port /dev/ttyS12 --
pause-for-test
```

Quickstart on ULX3S ESP32

The onboard ESP32 is pre-configured to work with this TT project. No external wiring is needed.

```
# [project]/ulx3s/ESP32
cd /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ESP32

PORT=/dev/ttyS3

idf.py -p $PORT -b 115200 flash
idf.py -p $PORT -b 115200 monitor
```

See also Comprehensive Testing below and the TT MicroPython SDK v3.

Register Overview

Register	Description
`reg_ctrl`	Global control bits (enable, feature flags)
`reg_src`	Selects entropy source or oscillator group
`reg_div`	Clock divider controlling sampling rate
`reg_mode`	Operating mode configuration
`reg_oscen`	Bitmask enabling individual oscillators
`reg_status`	Status flags (data ready, internal state)
`reg_rawlo`	Low byte of raw sampled entropy
`reg_rawhi`	High byte of raw sampled entropy

Key Concepts

- **Enable (`E`)**
  Must typically be cleared (`E0`) before changing configuration, then set (`E1`) to run.
- **Oscillator Control (`O`)**
  Enables one or more ring oscillators. More oscillators can improve entropy but may affect stability.
- **Sampling (`D`)**
  The divider controls how frequently entropy is sampled. This impacts randomness quality and bias.

- **Source Selection (`S`)**
  Allows switching between different entropy paths or test modes (implementation-specific).
- **Raw Data (`R6`, `R7`)**
  Returns unprocessed entropy bytes. These are not whitened and may require post-processing.

Typical Flow

1. Disable the core (`E0`)
2. Configure source, divider, mode, and oscillators
3. Enable the core (`E1`)
4. Read entropy and status via `R6`, `R7`, `R5`

This simple interface allows interactive exploration of TRNG behavior directly from a terminal.

UART TRNG Command Interface

All commands are ASCII and terminated with `\r`.
Responses are ASCII for normal register/configuration commands, typically:

`R<n>=<value>`

The optional `Bxx` raw stream command returns binary bytes and does not append `OK<CR>`.

Write Commands

Cmd	Description
`E<n>`	Write enable bit (0=disable, 1=enable)
`S<n>`	Write source select
`V<n>`	Write control bit 1
`W<n>`	Write control bit 2
`D<hex>`	Write divider
`M<hex>`	Write mode
`O<hex>`	Write oscillator enable mask

Special:

- `V\r` -> returns version string (if enabled in build)

Read Commands

Cmd	Description
R0	Read reg_ctrl
R1	Read reg_src
R2	Read reg_div
R3	Read reg_mode
R4	Read reg_oscen
R5	Read reg_status
R6	Read reg_rawlo
R7	Read reg_rawhi
R8	Read `ui_in` snapshot when `BIG16_SPI_REG` is enabled
R9	Read `uo_out` snapshot when `BIG16_SPI_REG` is enabled
RA	Read `uio_in` snapshot when `BIG16_SPI_REG` is enabled
RB	Read `uio_out` snapshot when `BIG16_SPI_REG` is enabled
RC	Read `uio_oe` snapshot when `BIG16_SPI_REG` is enabled
RD	Read build target ID when `BIG16_SPI_REG` is enabled
RE	Read analog status when `BIG16_SPI_REG` is enabled

Examples

Enable and configure:

```
E0\r
V0\r
W0\r
S0\r
D10\r
M00\r
O01\r
E1\r
```

Read back registers:

```
R0\r -> R0=01
R2\r -> R2=10
R6\r -> R6=7B
R7\r -> R7=3C
```

Version query:

```
V\r -> Version x.x.x <date>
```

Binary raw stream, when enabled:

```
B10<CR> -> 16 raw binary bytes
B64<CR> -> 100 raw binary bytes
BFF<CR> -> 255 raw binary bytes
B00<CR> -> ?<CR>
```

The xx byte count is hexadecimal, not decimal.

Do not use a normal terminal to view `Bxx` output. The response may contain arbitrary byte values, including control characters. Use `capture_trng_raw_uart.py` or another binary-safe capture tool.

Notes

- Commands are stateful; configure with `E0` before changes
- `R6`/`R7` provide raw entropy bytes
- `O` controls active oscillators (entropy source)
- `D` affects sampling rate and bias

UART

Connect with your favorite terminal program such as putty.

For the ULX3S FPGA, the UART is connected to pins `gp0` and `gp1` The default baud rate is 115200.

See the default reference ULX3S `ulx3s_v20.lpf` restraint file.

The `B11` (aka `gp[0]` or `gp0`) is Rx, to UART Tx. The `A10` (aka `gp[1]` or `gp1`) is Tx, to UART Rx.

```
# UART pins for testing

LOCATE COMP "uart_rx_pin" SITE "B11"; # formerly "gp[0]"; # J1_5+
GP0 PCLK
IOBUF PORT "uart_rx_pin" IO_TYPE=LVCMOS33;

LOCATE COMP "uart_tx_pin" SITE "A10"; # formerly "gp[1]"; # J1_7+
GP1 PCLK
IOBUF PORT "uart_tx_pin" IO_TYPE=LVCMOS33;
```

Comprehensive Testing

There are TT simulation tests and local ULX3S FPGA tests.

Set the `TT_PROJECT_ROOT` environment variable to the root of the project directory before running the tests or other scripts.

```
export TT_PROJECT_NAME="ttgf0p3-UART-FSM-TRNG-Lab"
export TT_PROJECT_ROOT="/mnt/c/workspace/$TT_PROJECT_NAME"
```

Testing on the Tiny Tapeout FPGA Development Kit

See the overview video for the FPGA Development Kit.

Testing ULX3S / TT

First run this script in one bash terminal, note test pause “Press Enter to continue...” (see concurrent `Testing ESP32`, below)

```
cd "$TT_PROJECT_ROOT/test-hw"

./run_tests.sh --with-build --ulx3s-board-version v307 --ignore-combinational-warning --no-warning-pause --port /dev/ttyS12 --pause-for-test
```

Testing SPI with ESP32

The ULX3S has a built-in ESP32, but a standalone ESP32 can also be used to test the SPI interface.

Current testing scripts:

```
# change directory to your ESP-IDF directory:
cd /mnt/c/SysGCC/esp32-master/esp-idf/v5.5

source ./export.sh

cd "$TT_PROJECT_ROOT/ulx3s/ESP32"
idf.py build

idf.py -p /dev/ttyS3 -b 115200 flash

idf.py -p /dev/ttyS3 -b 115200 monitor
```

There should be output from the ESP32 showing the SPI transactions and register values. This can be used to verify that the SPI interface is working correctly and that the TRNG lab core is responding to commands. (See example output):

```
gojimmypi:/mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ESP32
$ idf.py -p /dev/ttyS3 -b 115200 monitor
Executing action: monitor
Running idf_monitor in directory /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ESP32
Executing "/home/gojimmypi/.espressif/python_env/idf5.5_py3.10_env/bin/python /mnt/c/SysGCC/esp32-master/esp-idf/v5.5/tools/idf_monitor.py -p /dev/ttyS3 -b 115200 --toolchain-prefix xtensa-esp32-elf- --target esp32 --revision 0 /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ESP32/build/ulx3s_esp32.elf /mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ESP32/build/bootloader/bootloader.elf -m '/home/gojimmypi/.espressif/python_env/idf5.5_py3.10_env/bin/python' '/mnt/c/SysGCC/esp32-master/esp-idf/v5.5/tools/idf.py' '-p' '/dev/ttyS3' '-b' '115200'"...
--- esp-idf-monitor 1.6.2 on /dev/ttyS3 115200
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
I (13) boot: ESP-IDF v5.5 2nd stage bootloader
```

```
  [... snip. etc ... ]
I (350) main: SPI write mode: boot config once
I (350) ulx3s_spi: SPI regs: R0=00 R1=00 R2=10 R3=00 R4=01 R5=04
R6=3E R7=84 raw=0x843E status=0x04 src=0 div=0x10 mode=0x00
oscen=0x01
I (460) main: TRNG deterministic LFSR test
I (460) main: lfsr test sample 00: raw=0x7F2E status=0x00
I (460) main: lfsr test sample 01: raw=0x9F33 status=0x00


  [... snip. etc ... ]

I (740) ulx3s_spi: SPI regs: R0=00 R1=00 R2=10 R3=00 R4=01 R5=04
R6=3E R7=84 raw=0x843E status=0x04 src=0 div=0x10 mode=0x00
oscen=0x01
I (740) ulx3s_spi: SPI regs: R0=00 R1=00 R2=10 R3=00 R4=01 R5=04
R6=3E R7=84 raw=0x843E status=0x04 src=0 div=0x10 mode=0x00
oscen=0x01
```

If the `./run_tests.sh` was left at the `Press Enter to continue...` prompt, press `Enter` to continue with the next set of tests. The output should look something like this example:

```
Build PASSED
Flash...
Flashing file:
-rw-r--r-- 1 gojimmypi gojimmypi 294455 Jun  4 08:22 /mnt/c/
workspace/ttgf0p3-UART-FSM-TRNG-Lab/ulx3s/ulx3s.bit
ULX2S / ULX3S JTAG programmer v4.8 (git 96ebb45 built Oct  7 2020
22:42:00)
Copyright (C) Marko Zec, EMARD, gojimmypi, kost and contributors
Using USB cable: ULX3S FPGA 12K v3.0.3
Programming: 100%
Completed in 12.78 seconds.
/mnt/c/workspace/ttgf0p3-UART-FSM-TRNG-Lab/test-hw
Press Enter to continue...

Skipping register reset. Use --reset-registers to start from
configured defaults.

Running: version_if_present
Version probe response: b'Version 0.1.5d 6/3/2026\r'
PASS: Version command

  [... snip. etc ... ]
```

The continued output from the ESP32 in the separate TTY window should look something like this:

```
I (186760) ulx3s_spi: SPI regs: R0=06 R1=00 R2=10 R3=00 R4=01 R5=00
R6=00 R7=00 raw=0x0000 status=0x00 src=0 div=0x10 mode=0x00
```

```
oscen=0x01
I (187760) ulx3s_spi: SPI regs: R0=06 R1=00 R2=2A R3=5C R4=0F R5=00
R6=00 R7=00 raw=0x0000 status=0x00 src=0 div=0x2A mode=0x5C
oscen=0x0F
I (188760) ulx3s_spi: SPI regs: R0=00 R1=00 R2=10 R3=00 R4=01 R5=04
R6=00 R7=00 raw=0x0000 status=0x04 src=0 div=0x10 mode=0x00
oscen=0x01
```

TT Simulation tests

Commit changes. See results in actions.

In particular, note the output of the gds workflow:

- Linter output
- Routing Stats
- Cell usage by Category
- Tiny Tapeout Precheck Results
- Viewer summary

Test on ULX3S FPGA

Build and flash the bitstream to the FPGA, then run the test script. The test script will print the output of the FSM and TRNG.

Test locally with ULX3S ECP5 FPGA in /ulx3s/ directory.

- verilator_lint.sh
- ulx3s_build.sh
- ulx3s_flash.sh

Example:

```
cd ulx3s

./ulx3s_build.sh
./ulx3s_flash.sh
```

Connect to the FPGA using a serial terminal (e.g., `putty` or `minicom`) to view the output of the FSM and TRNG.

Local Loopback Test

There are two loopback tests: a basic loopback test and a deep loopback test. The basic loopback test verifies that the UART is functioning correctly by sending data from the FPGA to the host and back. The deep loopback test verifies that the FSM and TRNG are functioning correctly by sending commands to the FPGA and reading the responses.

Basic Loopback Test

The basic loopback assigns `Tx` to `Rx` in `top_ulx3s.v`.

```
    assign uart_tx_pin = uart_rx_sync;
```

All characters should be echoed back in the terminal when you type. This verifies that the UART is working correctly.

Sample loopback build defines `FORCE_LOOPBACK=1` macro in `ulx3s_build.sh`:

```
./ulx3s_build.sh --loopback --ignore-combinational-warning --no-warning-pause
./ulx3s_flash.sh
```

Deep Loopback Test

```
./ulx3s_build.sh --deep-loopback --ignore-combinational-warning --no-warning-pause
./ulx3s_flash.sh
```

Extensive loopback tests

Additional loopback tests:

```
    # The safest test to start (default write_with_delay when --bulk not specified)
    python ./loopback_test.py --port $PORT -b 115200 || exit 1

    echo "Test non-bulk mode, delay = 0.005"
    python ./loopback_test.py --port $PORT -b 115200 --tx-delay 0.005 || exit 1

    echo "Test non-bulk mode, delay = 0.001"
    python ./loopback_test.py --port $PORT -b 115200 --tx-delay 0.001 || exit 1

    echo "Test non-bulk mode, delay = 0.000"
    python ./loopback_test.py --port $PORT -b 115200 --tx-delay 0.000 || exit 1

    echo "Test bulk mode most challenging"
    python ./loopback_test.py --port $PORT -b 115200 --bulk || exit 1
```

The `run_tests.sh` can be used to run the loopback tests with the appropriate flags:

```
cd "$TT_PROJECT_ROOT/test-hw

./run_tests.sh --with-build --ignore-combinational-warning --no-warning-pause --loopback
./run_tests.sh --with-build --ignore-combinational-warning --no-warning-pause --deep-loopback
./run_tests.sh --with-build --ignore-combinational-warning --no-warning-pause
```

Local Automated Hardware Operation Tests

Generic local hardware operation tests in /test-hw/.

- tt_uart_test.py - Python script to test the UART functionality of the FSM and TRNG on the ULX3S FPGA. It sends commands to the FPGA and reads the responses to verify correct operation.
- run_tests.sh - Shell script to run the hardware tests. It can be configured to build the FPGA bitstream, flash it to the FPGA, and run the Python test script.

```
cd test-hw

 ./run_tests.sh --with-build --ignore-combinational-warning --no-
warning-pause
```

UART FSM TRNG Lab Datasheet

Document revision: 1.1.0 RTL revision string: `Version 1.2.5 7/2/2026`
Project family: Tiny Tapeout UART/SPI configurable TRNG experiment
Primary top modules: `tt_um_gojimmypi_ttgfa_UART_FSM_TRNG_Lab` (conditional based on build) License: Apache-2.0, as declared in the source files

1. Overview

The UART FSM TRNG Lab is a Tiny Tapeout compatible experimental random-number and entropy-source project. It exposes a small register bank through an ASCII UART command interface and, when enabled, an SPI mode 0 register-access interface. The design is intended for laboratory bring-up, education, FPGA validation, and ASIC ring-oscillator entropy experiments.

The core supports multiple sample sources:

- Deterministic LFSR source for repeatable tests
- Single ring-oscillator sample source
- XOR of multiple ring-oscillator sources
- Mixed source combining ring-oscillator and LFSR-derived state

The design is not a certified cryptographic random number generator. Raw output should be characterized, health-tested, and conditioned before use in security-sensitive systems.

2. Key Features

- Tiny Tapeout standard digital pin interface
- UART RX/TX control path using ASCII commands
- Optional SPI register-access slave
- SPI mode 0, MSB first
- Shared UART and SPI register bank
- 8-byte logical register map
- Configurable sample divider
- Selectable entropy/sample source
- Ring oscillator enable mask

- Deterministic single-step mode for test reproducibility
- Reset control through a configuration bit
- Default 25 MHz project clock
- Default 115200 baud UART
- ASIC real ring-oscillator path for selected PDK builds
- FPGA/simulation-safe LFSR tap substitute when real ring oscillators are disabled

3. Design Status and Intended Use

This block is intended as an experimental TRNG lab core. It is suitable for:

- Tiny Tapeout project demonstration
- FPGA bring-up on ULX3S or similar wrappers
- UART command parser testing
- SPI register interface testing
- Ring oscillator experimentation in supported ASIC flows
- Deterministic regression testing using the LFSR source

It is not, by itself, suitable as a drop-in cryptographic RNG. The raw output is unconditioned and no formal entropy claim is made in this datasheet.

4. Source Files

The main RTL files are:

File	Purpose
`project.v`	Top-level Tiny Tapeout project wrapper and project feature defines
`project_config.v`	Project clock and UART baud configuration
`target_pdk.v`	PDK target selection
`tt_um_main.v`	Tiny Tapeout pin mapping and UART/SPI/TRNG integration
`UART/uart_rx_min.v`	Minimal UART receiver
`UART/uart_tx_min.v`	Minimal UART transmitter
`UART/uart_trng_ascii_core.v`	UART and TRNG integration core
`TRNG/trng_cfg_ascii_core.v`	ASCII command parser and register bank
`TRNG/trng_lab_core.v`	Experimental TRNG/lab source logic
`TRNG/trng_stub.v`	Stub/test TRNG replacement when the lab core is not enabled
`SPI/spi_slave.v`	SPI mode 0 register-access slave
`JTAG/jtag_core.v`	Optional JTAG-related logic

5. Top-Level Parameters

Parameter	Default	Description
CLOCK_HZ	25000000	Project clock frequency in Hz
UART_BAUD	115200	UART baud rate

The source contains parameter checks that intentionally fail elaboration if either parameter is zero or if `CLOCK_HZ / UART_BAUD` would be zero.

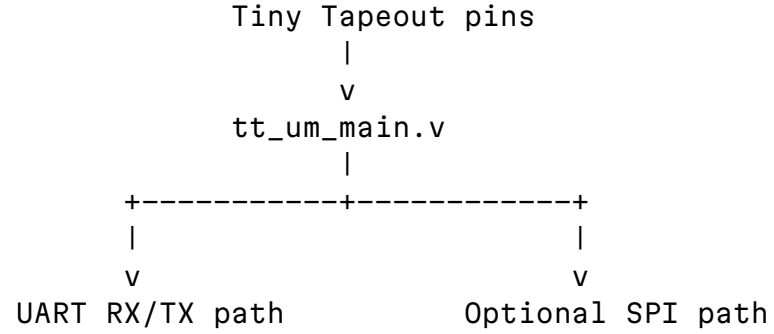
The `ULX3S_USE_GN12_50MHZ` configuration path can set `PROJECT_CLOCK_HZ` to 50 MHz for the optional ULX3S `gn12` clock path. Normal builds use the 25 MHz project clock.

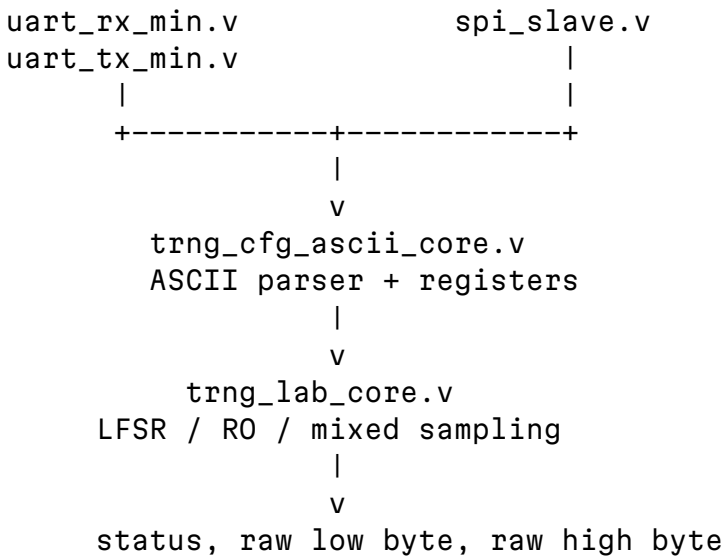
6. Build-Time Feature Defines

Define	Purpose
UART_ENABLED	Enables UART-related integration
SPI_ENABLED	Enables the SPI slave pin path
SPI_REG_ACCESS	Enables SPI access to the shared register bank
TRNG_ENABLED	Selects the TRNG lab core instead of the stub core
JTAG_ENABLED	Enables JTAG-related build integration
USE_LONG_STRINGS	Enables the long version string reply path
TRNG_USE_RO	Requests the real ring-oscillator TRNG path
TRNG_ALLOW_REAL_RO	Explicit guard required with `TRNG_USE_RO`
TRNG_BINARY_STREAM	Enables the UART `Bxx` raw binary byte-stream command
PDK_TARGET_SKY130	Selects SKY130 inverter cell instantiation
PDK_TARGET_GF180	Selects GF180 inverter cell instantiation
ULX3S	Selects ULX3S FPGA wrapper/build behavior

For ULX3S builds, the source intentionally rejects `TRNG_USE_RO` and `TRNG_ALLOW_REAL_RO`. FPGA and normal simulation paths use deterministic LFSR-derived substitutes for the ring oscillator signals.

7. Functional Block Diagram





8. Clock and Reset

Signal	Active level	Description
`clk`	Rising edge	Main synchronous project clock
`rst_n`	Low	Global reset

On reset, the register bank is initialized as shown below:

Register	Reset value
`reg_ctrl`	`0x00`
`reg_src`	`0x00`
`reg_div`	`0x10`
`reg_mode`	`0x00`
`reg_oscen`	`0x01`

The TRNG lab core internally resets the LFSR to `0x1ACE`, clears `sample_shift`, clears the sample counter, clears status, and clears raw output registers.

9. Tiny Tapeout Pin Map

Dedicated inputs: `ui_in[7:0]`

Pin	Direction	Function
`ui_in[7:5]`	Input	Reserved / unused
`ui_in[4]`	Input	SPI/JTAG select, 0 = SPI, 1 = JTAG (`INPUT` Dip Switch `SW4` down; when JTAG_ENABLED is defined)
`ui_in[3]`	Input	UART RX
`ui_in[2:0]`	Input	Reserved / unused

The UART RX input is synchronized through a two-stage synchronizer before it enters the UART receive logic.

Dedicated outputs: `uo_out[7:0]`

Pin	Direction	Function
`uo_out[0]`	Output	Debug visibility: `trng_bit`
`uo_out[1]`	Output	Debug visibility: `reg_status[0]`
`uo_out[2]`	Output	Debug visibility: `reg_status[1]`
`uo_out[3]`	Output	Debug visibility: `reg_status[2]`
`uo_out[4]`	Output	UART TX
`uo_out[5]`	Output	`reg_rawlo[0]`
`uo_out[6]`	Output	`reg_rawlo[1]`
`uo_out[7]`	Output	`reg_rawlo[2]`

Bidirectional IO: `uio[7:0]` **when SPI is enabled**

Pin	Direction	Function
`uio[0]`	Input	SPI CS_N
`uio[1]`	Input	SPI MOSI
`uio[2]`	Output	SPI MISO
`uio[3]`	Input	SPI SCK
`uio[7:4]`	Output	`reg_rawhi[7:4]` debug visibility

When SPI is enabled, `uio_oe` is driven as `0xF4`, making `uio[2]` and `uio[7:4]` outputs while leaving `uio[0]`, `uio[1]`, and `uio[3]` as inputs.

Bidirectional IO when SPI is disabled

When SPI is not enabled, `uio_out[7:0]` drives the full `reg_rawhi` byte and `uio_oe` is driven as `0xFF`.

10. UART Interface

UART settings

Setting	Value
Baud rate	`UART_BAUD`, default 115200
Data bits	8
Parity	None
Stop bits	1
Byte order	ASCII command bytes

Command terminator	Carriage return, `0x0D`

Line feed, `0x0A`, is ignored in command wait states, allowing common CRLF terminal behavior.

UART command summary

Command	Arguments	Effect	Reply
`Bxx`	2 hex nibbles, `01..FF`	Stream `xx` raw binary bytes from `reg_rawlo`/`reg_rawhi` alternately, when `TRNG_BINARY_STREAM` is enabled	Binary bytes, no `OK<CR>`
`E0` / `E1`	1 hex nibble	Write `reg_ctrl[0]`, TRNG enable	`OK<CR>`
`Sx`	1 hex nibble	Write `reg_src[1:0]`	`OK<CR>`
`Vx`	1 hex nibble	Write `reg_ctrl[1]`, deterministic single-step request	`OK<CR>`
`Wx`	1 hex nibble	Write `reg_ctrl[2]`, TRNG reset control	`OK<CR>`
`Dxx`	2 hex nibbles	Write `reg_div[7:0]`	`OK<CR>`
`Mxx`	2 hex nibbles	Write `reg_mode[7:0]`	`OK<CR>`
`Oxx`	2 hex nibbles	Write `reg_oscen[7:0]`	`OK<CR>`
`Rn`	`n = 0..7`	Read register `n`	`Rn=HH<CR>`
`V`	None	Version query	Version string + `<CR>`

Invalid syntax returns `?<CR>`.

UART command examples

```
V<CR>       -> Version 1.2.5 7/2/2026<CR>
R2<CR>      -> R2=10<CR>
E1<CR>      -> OK<CR>
D10<CR>     -> OK<CR>
S0<CR>      -> OK<CR>
```

```
001<CR>      -> OK<CR>
R6<CR>       -> R6=HH<CR>
R7<CR>       -> R7=HH<CR>
```

To reconstruct the current 16-bit raw sample from UART register reads:

```
raw16 = (R7 << 8) | R6
```

11. SPI Interface

The SPI interface is available when `SPI_ENABLED` and `SPI_REG_ACCESS` are enabled.

SPI electrical/protocol settings

Setting	Value
Mode	SPI mode 0
CPOL	0
CPHA	0
Bit order	MSB first
Chip select	Active low, `CS_N`
Register address width	3 .. 7 bits (see project_config.v)

SPI command byte

Bit field	Description
`bit[7]`	`1` = read, `0` = write
`bit[6:3]`	Ignored
`bit[2:0]`	Register address `0..7` or `0..15` or `0..127` (see project_config.v)

SPI read transaction

```
byte 0: 0x80 | addr
byte 1: dummy byte; returned MISO byte is the register value
```

For example, reading `reg_rawlo` at address 6:

```
TX: 86 00
RX: xx HH
```

The useful read value is the second received byte.

SPI write transaction

```
byte 0: addr
byte 1: data byte
```

Only addresses 0 through 4 are writable. Writes to addresses 5 through 7 are ignored by the register bank.

For example, writing divider register `R2 = 0x10`:

```
TX: 02 10
```

12. Register Map

Addr	UART read	Name	Access	Reset	Description
0	R0	reg_ctrl	R/W	0x00	Control bits
1	R1	reg_src	R/W	0x00	Source selection
2	R2	reg_div	R/W	0x10	Sample divider
3	R3	reg_mode	R/W	0x00	Mode/ debug field
4	R4	reg_oscen	R/W	0x01	Ring oscillator enable mask
5	R5	reg_status	Read-only	0x00	Status mirror
6	R6	reg_rawlo	Read-only	0x00	Raw sample low byte
7	R7	reg_rawhi	Read-only	0x00	Raw sample high byte
8	R8	ui_in	Read-only	board-dependent	Dedicated input snapshot
9	R9	uo_out	Read-only	board-dependent	Dedicated output snapshot
10	RA	uio_in	Read-only	board-dependent	Bidirectional input snapshot
11	RB	uio_out	Read-only	board-dependent	Bidirectional output snapshot
12	RC	uio_oe	Read-only	board-dependent	Bidirectional output-enable snapshot
13	RD	BUILD_TARGET_ID	Read-only	target-dependent	Build target ID

14	RE	analog_status	Read-only	0x00	Sampled analog experiment status
15	RF	analog_measure	Read-only	0x00	Latest ua[5] threshold/ decay timing sample

13. Control Register: `reg_ctrl`, Address 0

Bit	Name	Description
0	enable	Enables periodic sampling when set
1	step	Deterministic single-step request. A rising edge creates one sample event
2	reset	Resets the TRNG lab core while asserted
7:3	Reserved	Currently unused

UART aliases:

Command	Field
E0 / E1	reg_ctrl[0]
V0 / V1	reg_ctrl[1]
W0 / W1	reg_ctrl[2]

Note: bare V<CR> is the version query. V0<CR> and V1<CR> are control writes.

14. Source Register: `reg_src`, Address 1

Only `reg_src[1:0]` is used.

Value	Source	Description
0	SRC_LFSR	LFSR bit source, deterministic and repeatable
1	SRC_RO0	Sampled ring oscillator 0 source
2	SRC_ROX	XOR of the ring oscillator raw bits
3	SRC_MIX	Mixed source using RO XOR, LFSR taps, and sample history

UART alias: Sx<CR> writes `reg_src[1:0]`.

15. Divider Register: `reg_div`**, Address 2**

`reg_div` controls the periodic sample interval when `reg_ctrl[0]` is enabled. The internal sample counter increments while enabled. A sample event is generated when:

```
sample_ctr >= reg_div
```

A single-step event through `reg_ctrl[1]` can also generate a sample event without waiting for the periodic divider.

UART alias: `Dxx<CR>` writes the full divider byte.

16. Mode Register: `reg_mode`**, Address 3**

`reg_mode` is a full 8-bit writable register. In the current TRNG lab core, `reg_mode[2:0]` is mirrored into `reg_status[7:5]`. Other bits are reserved for future use.

UART alias: `Mxx<CR>` writes the full mode byte.

17. Oscillator Enable Register: `reg_oscen`**, Address 4**

`reg_oscen` is an 8-bit enable mask for the ring oscillator instances in real RO builds.

There's a conditional `BASIC_RO_SET` (not defined) in `trng_lab_core.v` for RO states 3 .. 17.

The default build contains 7 .. 21 stages:

Bit	Real RO instance	Stage count
0	`u_ro0`	7
1	`u_ro1`	9
2	`u_ro2`	11
3	`u_ro3`	13
4	`u_ro4`	15
5	`u_ro5`	17
6	`u_ro6`	19
7	`u_ro7`	21

In FPGA and normal simulation builds, the RO raw bits are derived from LFSR taps instead of real ring oscillators.

UART alias: `Oxx<CR>` writes the full oscillator enable mask.

18. Status Register: `reg_status`**, Address 5**

Bit field	Description
`bit[0]`	Mirrors TRNG enable

bit[1]	Mirrors sample tick condition
bit[2]	Indicates at least one oscillator enable bit is set
bits[4:3]	Mirrors source selection
bits[7:5]	Mirrors reg_mode[2:0]

reg_status is read-only from the external UART/SPI register interfaces.

19. Analog Status Register: analog_status**, Address 14 / R14 / 0xE**

analog_status is read-only and is intended for post-silicon analog bring-up. It does not add a new command parser path; it reuses the existing RE<CR>/ SPI register read mechanism available when BIG16_SPI_REG is enabled. In the patched GDS, ua[5] also has a small top-metal fringe/pickup passive tied to the pad; bits 3 and 4 are the main readback hooks for that experiment.

Bit	Description
0	Synchronized sample of ua[0] (ain_ext)
1	Synchronized sample of ua[2] (cmp_ref_ext)
2	Threshold compare helper, ua[0] & ~ua[2] after synchronization
3	Live synchronized sample of ua[5] (puf_probe)
4	Latched ua[5] probe sample from the charge/release/sample sequence
5	Current sigma-delta DAC bit driving ua[1] when enabled
6	Current oscillator/TRNG monitor bit driving ua[4] when enabled
7	ua[5] probe driver output-enable state

19.1 Analog Measurement Register: analog_measure**, Address 15 / R15 / 0xF**

analog_measure is read-only and captures the latest ua[5] passive-structure threshold/decay timing sample. When the probe sequence is enabled, the RTL drives ua[5], releases it to high-Z, counts clock cycles until the synchronized pad input crosses the ordinary CMOS threshold, and latches the count for readback. This gives the GDS-level Metal4 fringe/pickup structure a direct digital measurement path without adding a full analog macro.

Value	Meaning
0x00	No threshold crossing observed yet, or immediate crossing
0x01..0xFE	Clock cycles from release to threshold crossing
0xFF	Saturated/no crossing before timeout

Basic bring-up sequence:

```
E1<CR>
M18<CR>
O08<CR>
RE<CR>
RF<CR>
```

20. Raw Output Registers: `reg_rawlo` and `reg_rawhi`

The TRNG lab core maintains a 16-bit sample shift register. On each sample event, the selected source bit is shifted into the sample history and the raw output registers are updated.

Register	Description
`reg_rawlo`	Low byte of the latest raw sample history
`reg_rawhi`	High byte of the latest raw sample history

The current 16-bit raw sample value is reconstructed as:

```
raw16 = (reg_rawhi << 8) | reg_rawlo
```

20. Sampling Behavior

A sample event occurs when either condition is true:

```
do_sample = (enable && sample_tick) || step_pulse
```

Where:

```
sample_tick = sample_ctr >= reg_div
step_pulse  = reg_ctrl[1] && !previous_reg_ctrl_bit_1
```

On each sample event:

- `sample_ctr` is cleared to zero
- The 16-bit LFSR advances
- The selected source bit shifts into `sample_shift`
- `reg_rawlo` and `reg_rawhi` are updated

When enable is deasserted, the sample counter is held at zero. A single-step pulse can still advance one sample.

21. LFSR Details

The deterministic LFSR path is used for repeatable tests and for FPGA/simulation-safe substitute RO signals. On TRNG reset, the LFSR seed is:

```
0x1ACE
```

The next LFSR bit is computed as:

```
lfsr_next_bit = lfsr[15] ^ lfsr[13] ^ lfsr[12] ^ lfsr[10]
```

The LFSR then shifts as:

```
lfsr <= {lfsr[14:0], lfsr_next_bit}
```

This path is deterministic and should not be treated as an entropy source.

22. Ring Oscillator Path

In real ASIC RO builds, the default design instantiates eight odd-length ring oscillators with stage counts from 7 to 21. The oscillator outputs feed the source selection logic through synchronizers.

In FPGA and normal simulation builds, real ring oscillators are not instantiated. Instead, the RO raw signals are mapped to LFSR taps so the rest of the design can be tested safely without combinational oscillator loops.

Supported real RO PDK cell paths in the current source are:

PDK define	Inverter cell
`PDK_TARGET_SKY130`	`sky130_fd_sc_hd__inv_2`
`PDK_TARGET_GF180`	`gf180mcu_fd_sc_mcu7t5v0__inv_2`

23. Recommended Bring-Up Sequence

A conservative UART bring-up sequence is:

```
V<CR>       Read version string
R0<CR>      Confirm control reset value
R1<CR>      Confirm source reset value
R2<CR>      Confirm divider reset value, expected 10
R3<CR>      Confirm mode reset value
R4<CR>      Confirm oscillator enable reset value, expected 01
S0<CR>      Select deterministic LFSR source
D10<CR>     Set divider to 0x10
M00<CR>     Clear mode
O01<CR>     Enable oscillator mask bit 0
E1<CR>      Enable sampling
R6<CR>      Read raw low byte
R7<CR>      Read raw high byte
```

For deterministic regression testing, use source `S0`, assert and release reset through `W1` and `W0`, then issue single-step pulses through `V1` and `V0` as required by the test harness.

24. Known Deterministic Regression Sequence

With the deterministic LFSR path and the established single-step/reset test flow, the known-good 16-bit sample sequence used in current hardware regression is:

```
sample 01: 0x7F2E
sample 02: 0x9F33
sample 03: 0xFC1C
sample 04: 0x6F03
sample 05: 0x4B7D
sample 06: 0x52C8
```

```
sample 07: 0xD6B7
sample 08: 0xEF2A
```

This sequence is a reproducibility check for the deterministic path. It is not an entropy-quality claim.

25. UART and SPI Concurrency Notes

UART and SPI share the same logical register bank. SPI writes are applied when `spi_reg_wr_en` is asserted. UART commands also update the same configuration registers.

When using SPI as a passive monitor while UART is active, individual register reads are separate transactions. Reading `reg_rawlo` and `reg_rawhi` separately is not atomic, so the two bytes may occasionally come from adjacent sample updates. For exact sample capture, add an atomic snapshot/latch mechanism or temporarily stop sampling before reading both bytes.

26. Limitations

- No cryptographic certification is claimed.
- No built-in conditioner, extractor, or DRBG is provided by this RTL block.
- Raw RO entropy quality must be measured on the actual ASIC implementation.
- FPGA and normal simulation builds do not use real ring oscillators.
- SPI multi-byte reads of raw low/high registers are not atomic.
- UART command parsing is intentionally small and accepts only the documented command forms.
- Register addresses 5 through 7 are read-only through SPI writes and UART write aliases do not target them.

27. Characterization Recommendations

Before using ASIC RO output as a source of entropy, characterize at minimum:

- Raw bit bias for each source selection
- Bit transition rate
- Autocorrelation
- Per-oscillator behavior across voltage and temperature
- Behavior across process corners and multiple chips
- Startup behavior after reset
- Sensitivity to `reg_div` and `reg_oscen`
- Health test behavior under stuck oscillator conditions

For security use, add a conditioning function and a health-test strategy appropriate for the target application.

28. Revision History

Datasheet rev	Date	Notes
0.1	2026-05-23	Initial datasheet generated from current TRNG source package

Example Outputs

- Example ESP32 Output
- Example Text Results

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	reserved_in0	trng_bit	spi_cs_n_jtag_tms
1	reserved_in1	status0	spi_mosi_jtag_tdi
2	reserved_in2	status1	spi_miso_jtag_tdo
3	uart_rx	status2	spi_sck_jtag_tck
4	spi_jtag_sel	uart_tx	rawhi4
5	reserved_in5	rawlo0	rawhi5
6	reserved_in6	rawlo1	rawhi6
7	reserved_in7	rawlo2	rawhi7

Analog Pins

ua#	analog#	Description
0	6	ain_ext
1	11	dac_out
2	7	cmp_ref_ext
3	10	amon_out
4	8	osc_out
5	9	puf_probe

Tiny FABulous FPGA (3.3V version)

by **Leo Moser**

359 HDL Project

github.com/mole99/tt-fabulous-gf-0p3-3v3

"A tiny FABulous FPGA fabric, ready for use with Yosys and nextpnr."

How it works

Tiny FABulous FPGA for GF0p3 (3.3V version).

This design implements a tiny FPGA with 64 LUT4+FF. The FPGA fabric is 6x4 tiles in size, of which 4x2 are LUT4x8_ha tiles. The logic cells include a vertical carry-chain in upwards direction, allowing for fast additions up to 15-bits.

The I/Os resemble the Tiny Tapeout interface, allowing for clk, rst_n, uo, ui and uio signals. This enables to directly implement simple Tiny Tapeout designs on the FPGA.

The user design is synthesized using Yosys and implemented using nextpnr (currently forks are required to be used, but the changes will be upstreamed).

The bitstream is uploaded to the fabric using a bitbang interface (see how to test). The bitbang interface is active while reset is applied, this ensures that all I/Os are available for the active user design.

The exact available resources can be seen in this table:

Primitive	Available	Description
FABULOUS_LC	64	Logic cells with LUT4+FF and carry-chain.
IOBUF	26	Input/output buffers.
GBUF	4	Global buffers to supply clock, reset and enable to the flip-flops.
SYS_RESET	1	Can be used to reset the design after configuration.

Even though there are 26 IOBUF are available, only the uio signals are actually bidirectional. uo will always read zero when read from, and writing to clk, rst_n and ui has no effect.

The GBUFs are used for high-fanout signals. Their use is mandatory for the clock signal of flip-flops to ensure a balanced clock network. This means up to 4 clock domains are possible. The GBUFs can also be used for reset and enable of the FFs, although those can also be routed through "normal" fabric routing.

SYS_RESET applies a reset during fabric reconfiguration and can only be directly connected to a GBUF.

How to test

First, compile a bitstream for your user design. The bitstream is big-endian with 32-bit words.

1. Set rst_n to 1 to reset the configuration interface.
2. Set rst_n to 0 to enable the configuration interface.
3. Write the bitstream bits to ui[1] (MSB first) and the sample signal on ui[0].

The data is sampled on a rising edge of the sample signal. The interface is synchronous, so ensure that the clk signal is toggling faster than the sample signal. Is anything is unclear, have a look at the top-level cocotb tests.

Finally, set rst_n to 1 and enjoy your design on Tiny FABulous FPGA!

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ui[0] or sample_i	uo[0]	uio[0]
1	ui[1] or data_i	uo[1]	uio[1]
2	ui[2]	uo[2]	uio[2]
3	ui[3]	uo[3]	uio[3]
4	ui[4]	uo[4]	uio[4]
5	ui[5]	uo[5]	uio[5]
6	ui[6]	uo[6]	uio[6]
7	ui[7]	uo[7]	uio[7]

Wafer.space Logo VGA Screensaver

by **Uri Shaked**

448 | 25.175 MHz | HDL Project

`github.com/TinyTapeout/tt-waferspace-vga-screensaver`

"Wafer.space Logo bouncing around the screen (640x480, TinyVGA Pmod)"

How it works

Displays a bouncing Wafer.space logo on the screen, with an animated color gradient.

Figure 448.1: Wafer.space VGA screensaver

How to test

Connect to a VGA monitor. Set the following inputs to configure the design:

- `tile` (ui_in[0]) to repeat the logo and tile it across the screen,
- `solid_color` (ui_in[1]) to use a solid color instead of an animated gradient.

If you have a Gamepad Pmod connected, you can also use the following controls:

- Start button: start/pause bouncing
- Left/right/up/down: change the bouncing direction (if bouncing) or move the logo around the screen (if paused)

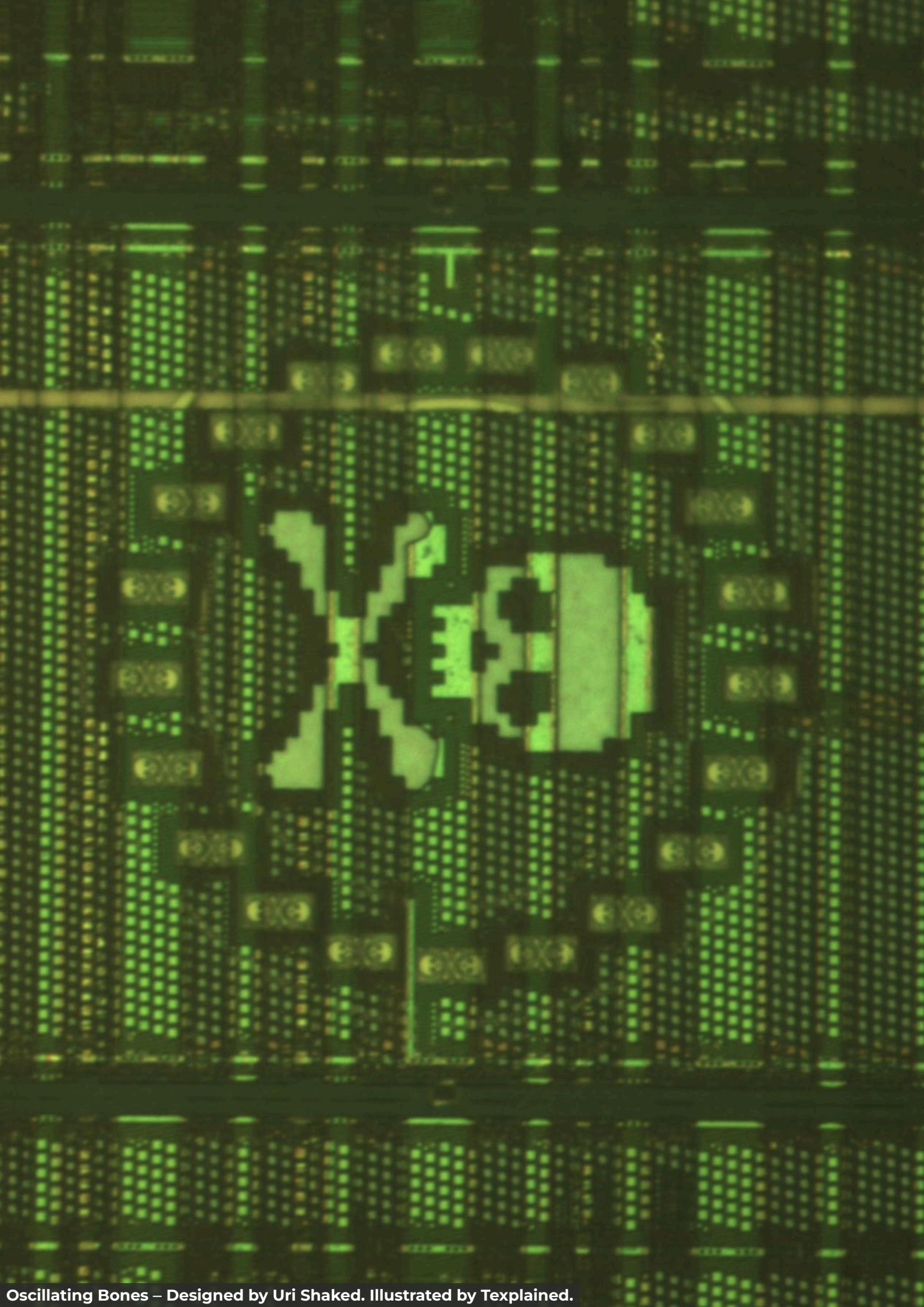
External hardware

- Tiny VGA Pmod
- Optional: Gamepad Pmod

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	tile	R1	—
1	solid_color	G1	—
2	—	B1	—
3	—	VSync	—
4	gamepad_latch	R0	—
5	gamepad_clk	G0	—
6	gamepad_data	B0	—
7	—	HSync	—



Oscillating Bones – Designed by Uri Shaked. Illustrated by Texplained.

USB CDC (Serial) Device

by **Uri Shaked**

`0450` `48 MHz` `HDL Project`

`github.com/urish/tt-usbcdc-device`

"USB to UART bridge, 115200 baud rate"

How it works

A USB CDC to UART bridge, based on tinyfpga_bx_usbserial.

How to test

1. Connect `usb_p` and `usb_n` pins to D+ / D- USB pins either through 68 ohm resistors or directly (the resistors are recommended, but not mandatory).
2. Connect a 1.5k ohm resistor between `dp_pu_o` and `usb_p` (D+).
3. Connect the RX and TX pins to a UART device or to a logic analyzer.
4. Set the clock frequency to 48 MHz.

The device should appear as a serial port on your computer, with vendor_id=1209 and product_id=5454 (https://pid.codes/1209/5454/). The baud rate for the UART interface is hardcoded at 115200.

Demo mode

Set `ui_in[0]` high to enable demo mode. In this mode, the device sends "Tiny Tapeout!" over USB once per second. UART RX input is ignored while demo mode is active.

External Hardware

USB breakout board, 1.5k ohm resistor

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	demo_mode	—	usb_p
1	—	—	usb_n
2	—	—	dp_pu_o
3	RX	—	—
4	—	TX	—
5	—	—	—

#	Input	Output	Bidirectional
6	—	—	—
7	—	configured	—

PolyWave

by **Riccardo Pellegrini**

`0452` `50 MHz` `Analog Project`

`github.com/ODGrip/ttgf-polywave`

"Mixed-signal four-channel waveform source with a compact digital engine driving four 16-bit R-2R DACs."

PolyWave is a four-channel mixed-signal waveform source. A compact digital engine generates four 16-bit sample streams, and the custom GF180 layout converts those streams to analog voltages with four R-2R DAC ladders.

The design is intended as a small multi-output waveform lab for synchronized signals, modulation experiments, DAC characterization and mixed-signal Tiny Tapeout demonstrations.

How it works

The digital engine contains four 20-bit phase accumulators, a shared 16-bit LFSR/noise source, a 4-bit sequencer and four 16-bit DAC sample registers. It can run autonomously in generated-waveform mode, or it can accept direct byte writes to any DAC channel.

The four DAC sample registers drive inverted 16-bit layout buses named `R2R_Bn_0`, `R2R_Bn_1`, `R2R_Bn_2` and `R2R_Bn_3`. These buses feed the four physical R-2R ladders in the custom GDS. The analog DAC outputs are exposed on `ua[0]` to `ua[3]`.

This repository uses the Tiny Tapeout custom GDS path. The prebuilt `gds/` and `lef/` files are the physical source for the submitted layout. `src/project.v` documents and syntax-checks the mixed-signal partition used by the layout; it is not synthesized by the submission action into a new layout. The public `tt_um_odgrip_polywave` module keeps the standard Tiny Tapeout analog port list, instantiates `polywave_top_digital` for the digital waveform engine, and instantiates `polywave_top_analog_stub` as the analog boundary that represents the custom GDS. The R-2R buses are internal signals between those two blocks, not top-level ports.

The submitted analog frame uses the GF180 pgvaa Tiny Tapeout interface, so `VAPWR` remains part of the project interface even though the current digital engine does not consume the analog supply internally.

Digital controls

Generated mode

Set `ui_in[7]` low to use the autonomous waveform generator.

Signal	Function
`ui_in[2:0]`	Extra live modulation bits and low-cost phase nudges
`ui_in[3]`	Fast internal tempo select
`ui_in[5:4]`	Scene selection for the waveform formulas
`ui_in[6]`	Freeze phase accumulator updates while held high
`uio_in[7:0]`	Live modulation/control bus used by the waveform engine

In this mode the four phase accumulators, cross-modulation paths, sequencer, noise source, saw/triangle/PWM/stepped waveforms and live controls generate the four DAC values.

Direct DAC mode

Set `ui_in[7]` high to stop generated DAC updates and write DAC codes directly.

Signal	Function
`ui_in[5:4]`	DAC channel select: 0 to 3
`ui_in[3]`	Byte select: 0 for low byte, 1 for high byte
`uio_in[7:0]`	Byte value to load
`ui_in[6]`	Rising-edge load strobe

To write a full 16-bit DAC code, load the low byte and high byte separately with two pulses on `ui_in[6]`.

Outputs

Signal	Function
`ua[0]`	DAC A analog output
`ua[1]`	DAC B analog output
`ua[2]`	DAC C analog output
`ua[3]`	DAC D analog output
`uo_out[2:0]`	Mirror of `ui_in[2:0]` live modulation bits
`uo_out[6:3]`	Current 4-bit sequencer step
`uo_out[7]`	Tempo tick
`uio_out[7:0]`	Always zero

`uio_oe[7:0]`	Always zero, so the bidirectional pins are input-only controls

How to test

1. Apply power to the design.
2. Provide the system clock and release reset.
3. Hold `ui_in[7]` low for generated-waveform mode.
4. Observe `ua[0]` to `ua[3]` with an oscilloscope or data acquisition system.
5. Change `ui_in[5:4]`, `ui_in[3]`, `ui_in[2:0]` and `uio_in[7:0]` to exercise the live waveform controls.
6. Set `ui_in[7]` high and use direct DAC mode to write known 16-bit codes to each channel.
7. Check `uo_out[7]` and `uo_out[6:3]` for the tempo tick and sequencer state.

External hardware

Use a clock/reset source, digital control source for `ui_in` and `uio_in`, and an oscilloscope, logic analyzer or data acquisition system to observe the analog and status outputs.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	MOD0	MOD0_MON	LIVE0
1	MOD1	MOD1_MON	LIVE1
2	MOD2	MOD2_MON	LIVE2
3	TEMPO_HI	SEQ0	LIVE3
4	SCENE0	SEQ1	LIVE4
5	SCENE1	SEQ2	LIVE5
6	FREEZE/LOAD	SEQ3	LIVE6
7	DIRECT	TEMPO_TICK	LIVE7

Analog Pins

ua#	analog#	Description
0	4	DAC_A
1	1	DAC_B
2	3	DAC_C
3	2	DAC_D

MCML VCO

by **Namibj**

0454 Analog Project

`github.com/namibj/tt_gf-0p3_mcml`

"An MCML VCO"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Analog Pins

ua#	analog#	Description
0	0	vco_takeoff_tail_bias
1	5	vco_tail_bias
2	2	clk_0_p
3	3	clk_0_n
4	1	clk_1_p
5	4	clk_1_n

KianV uLinux SoC

by **Hirosh Dabui**

518 16 MHz HDL Project

github.com/TinyTapeout/KianV-RV32IMA-RISC-V-uLinux-SoC

"A RISC-V ASIC that can boot µLinux."

How it works

32-bit RISC-V IMA processor, capable of booting Linux. Features 16 MiB of external SPI flash memory, 16 MiB of external PSRAM (8 MiB per bank), a UART peripheral, and a SPI peripheral.

System Memory Map

The system memory map is as follows:

Address	Size	Purpose
0x10000000	0x14	UART Peripheral
0x10500000	0x14	SPI Peripheral
0x10600000	0x0c	GPIO Peripheral
0x11100000	0x04	Reset / HALT control
0x20000000	16 MiB	SPI Flash
0x80000000	16 MiB	PSRAM

The system boots from the SPI flash memory. After reset, the CPU starts executing code from 0x20100000 (corresponding to the offset 0x100000 into the SPI flash memory), where the bootloader is expected to be.

UART Peripheral registers

Address	Name	Description
0x10000000	UART_DATA	Write to transmit, read to receive
0x10000005	UART_LSR	UART line status register
0x10000010	UART_DIV	Clock divider for UART baud rate

SPI Peripheral registers

Address	Name	Description
0x10500000	SPI_CTRL0	SPI Peripheral Control
0x10500004	SPI_DATA0	SPI Data

0x10500010	SPI_DIV	Clock divider for SPI peripheral

GPIO Peripheral registers

Address	Name	Description
0x10600000	GPIO_UO_EN	Enable bits for uo_out pins
0x10600004	GPIO_UO_OUT	Write to uo_out pins
0x10600008	GPIO_UI_IN	Read from ui_in pins(read-only)

CPU control register

Address	Name	Description
0x11100000	CPU_RESET	Write 0x7777 to reset the CPU, 0x5555 to halt the CPU.

How to test

Build the system image as described in the kianRiscV repo and load it into the SPI flash memory:

Flash offset	File name	Description
0x100000	`bootloader.bin`	Bootloader
0x180000	`kianv.dtb`	Device Tree Blob
0x200000	`Image`	Linux kernel + rootfs

The system runs at 16 MHz.

External hardware

QSPI Pmod - can be purchased from the Tiny Tapeout store.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	gpio[0]	spi_cen0	ce0 flash
1	gpio[1]	spi_sclk0	sio0
2	spi_sio1_so_miso0	spi_sio0_si_mosi0	sio1
3	uart_rx	gpio[3]	sck
4	gpio[4]	uart_tx	sd2
5	gpio[5]	gpio[5]	sd3
6	gpio[6]	gpio[6]	cs1 psram

#	Input	Output	Bidirectional
7	gpio[7]	gpio[7]	cs2 psram

Simon Says memory game

by **Uri Shaked**

519 | 50 kHz | HDL Project

github.com/urish/tt-simon-game

"*Repeat the sequence of colors and sounds to win the game*"

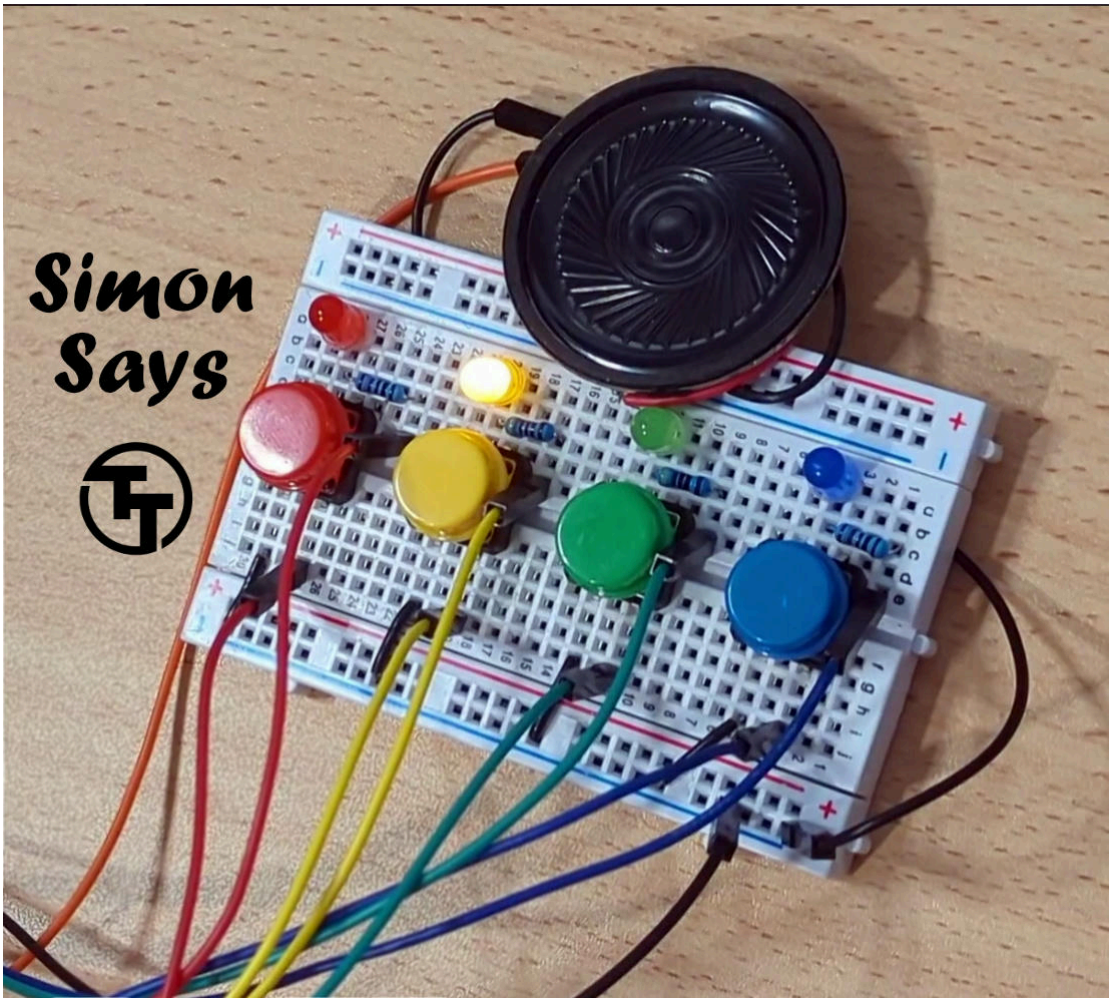


Figure 519.1: Simon Says Game

How it works

Simon says is a simple electronic memory game: the user has to repeat a growing sequence of colors. The sequence is displayed by lighting up the LEDs. Each color also has a corresponding tone.

In each turn, the game will play the sequence, and then wait for the user to repeat the sequence by pressing the buttons according to the color sequence. If the user repeated the sequence correctly, the game will play a

“leveling-up” sound, add a new color at the end of the sequence, and move to the next turn.

The game continues until the user has made a mistake. Then a game over sound is played, and the game restarts.

Check out the online simulation at https://wokwi.com/projects/408757730664700929 (including wiring diagram).

Clock settings

The `clk_sel` input selects the clock source:

- `0`: external 50 KHz clock, provided through the `clk` input.
- `1`: internal clock, generated by the `ring_osc` module, at 50 kHz (to be confirmed on TTGF26a silicon).

The internal clock is generated by a 13-stage ring oscillator (101 MHz in GF180 SPICE at 3.3 V), divided by 2048 to land near 50 kHz, matching the external-clock path. The divide ratio will be confirmed against the post-layout PEX and silicon.

When using the internal clock, its signal is also output on the `uo_out[7]` pin for debugging purposes.

How to test

Use a Simon Says Pmod to test the game.

Provide a 50 KHz clock input, reset the game, and enjoy!

If you don’t have the Pmod, you can still connect the hardware manually as follows:

1. Connect the four push buttons to pins `btn1`, `btn2`, `btn3`, and `btn4`. Also connect each button to a pull down resistor.
2. Connect the LEDs to pins `led1`, `led2`, `led3`, and `led4`, matching the colors of the buttons (so `led1` and `btn1` have the same color, etc.). Don’t forget current-limiting resistors!
3. Connect the speaker to the `speaker` pin (optional).
4. Connect the seven segment display as follows: `seg_a` through `sev_g` to individual segments, `dig1` to the common pin of the tens digit, `dig2` to the common pin of the ones digit. Set `seginv` according to the type of 7 segment display you have: high for common anode, low for common cathode.
5. Reset the game, and then press any button to start it. Enjoy!

External Hardware

Simon Says Pmod or four push buttons (with pull-down resistors), four LEDs, and optionally a speaker/buzzer and two digit 7-segment display.

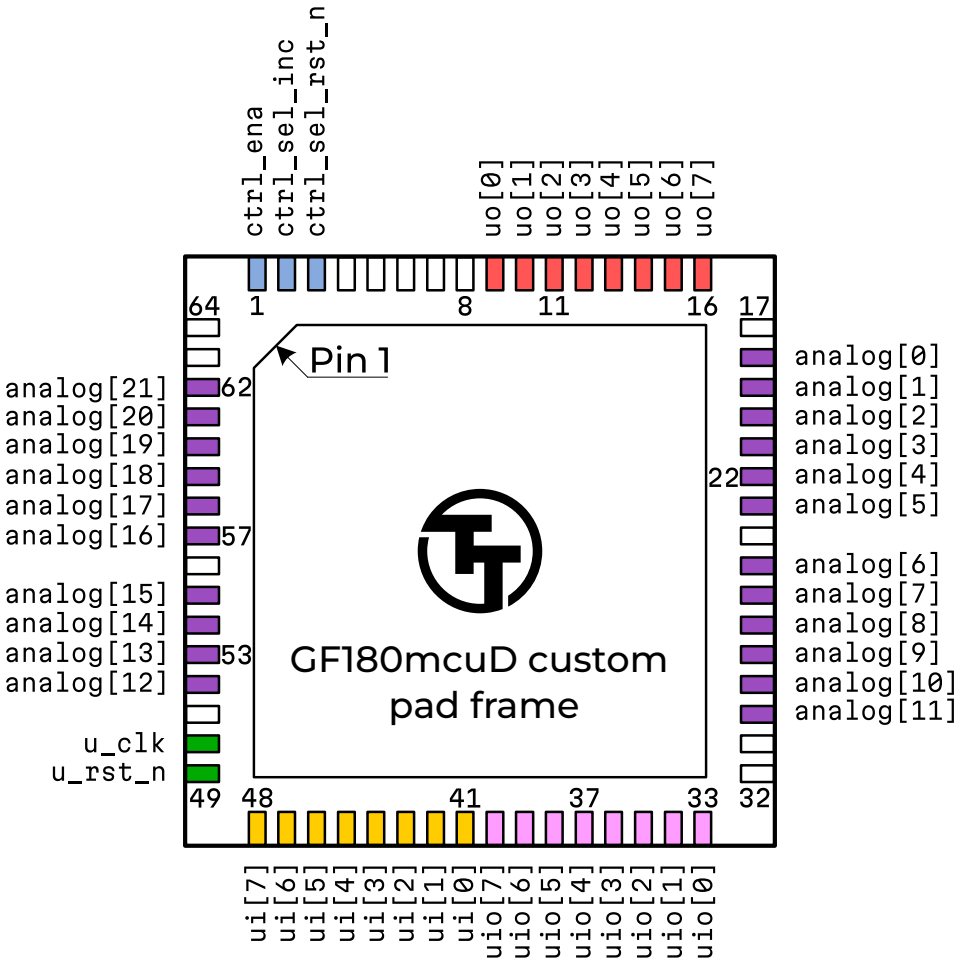
Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	btn1	led1	seg_a
1	btn2	led2	seg_b
2	btn3	led3	seg_c
3	btn4	led4	seg_d
4	seginv	speaker	seg_e
5	—	dig1	seg_f
6	—	dig2	seg_g
7	clk_sel	clk_internal	—

Pinout

The chip is packaged in a 64-pin QFN package. The pinout is shown below.



Bottom View

Note

You will receive the chip mounted on a breakout board (github.com/tinytapeout/breakout-pcb).

The pinout is provided for advanced users, as most users will not need to solder the chip directly.

The Tiny Tapeout Multiplexer

Overview

The Tiny Tapeout Multiplexer distributes a single set of user IOs to multiple user designs. It is the backbone of the Tiny Tapeout chip.

It has the following features:

- 10 dedicated inputs
- 8 dedicated outputs
- 8 bidirectional outputs
- Supports up to 512 user designs (32 mux units, each with up to 16 designs)
- Designs can have different sizes. The basic unit is called a tile, and each design can occupy up to 16 tiles.

Operation

The multiplexer consists of three main units:

1. The controller — used to set the address of the active design
2. The spine — a bus that connects the controller with all the mux units
3. Mux units — connects the spine to individual user designs

The Controller

The mux controller has 3 input lines:

Input	Description
`ena`	Sent as-is (buffered) to the downstream mux units
`sel_rst_n`	Resets the internal address counter to 0 (active low)
`sel_inc`	Increments the internal address counter by 1

It outputs the address of the currently selected design on the `si_sel` port of the spine (see below).

For instance, to select the design at address 12, you need to pulse `sel_rst_n` low, and then pulse `sel_inc` 12 times:

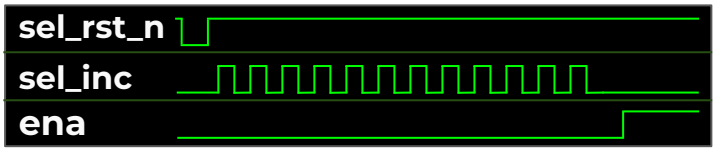


Figure 1: Mux signals for activating the design at address 12

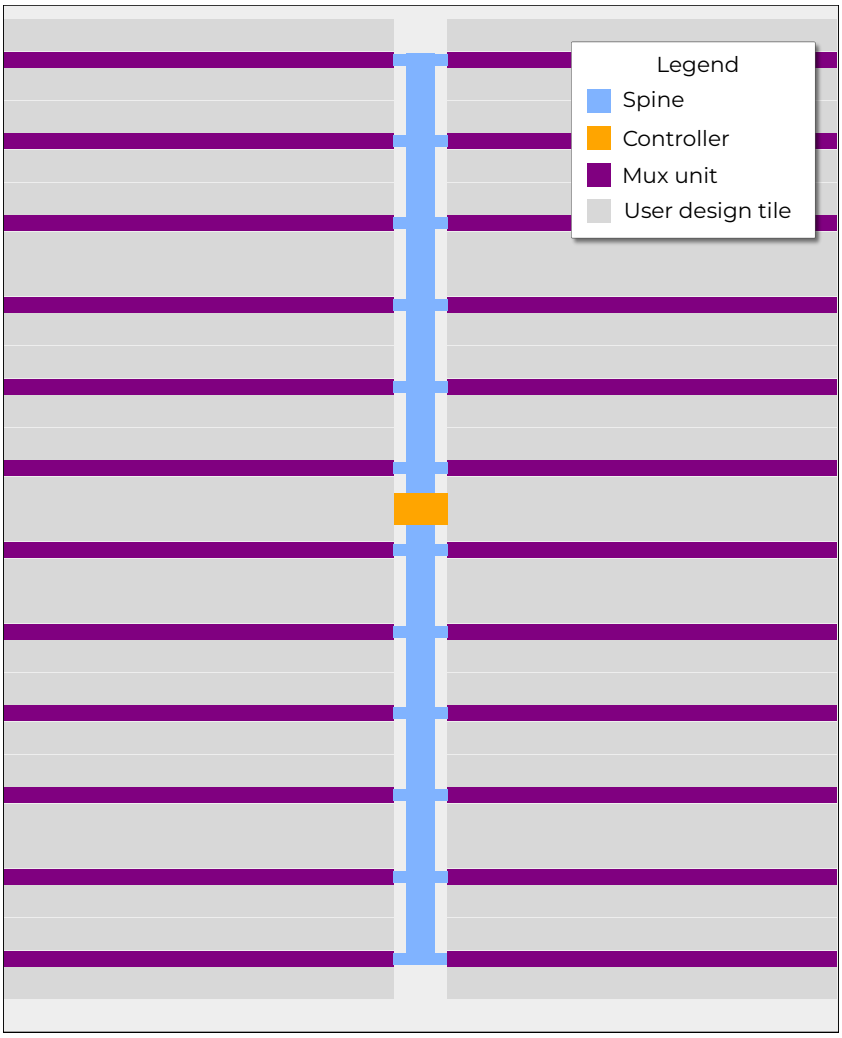


Figure 2: Mux Diagram

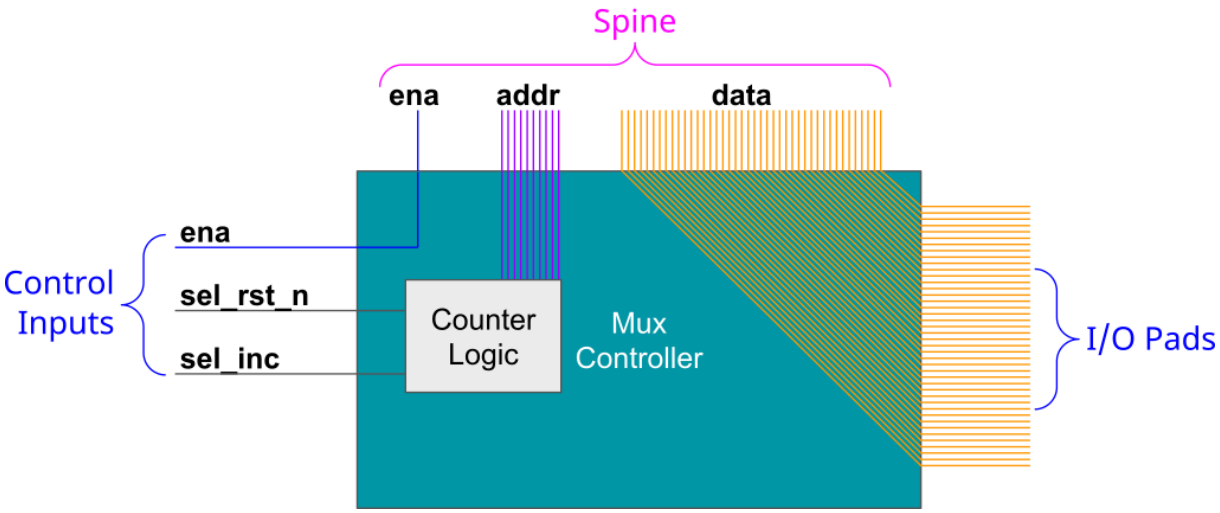


Figure 3: Mux Controller Diagram

Internally, the controller is just a chain of 10 D-flip-flops. The `sel_inc` signal is connected to the clock of the first flip-flop, and the output of each flip-flop is connected to the clock of the next flip-flop. The `sel_rst_n` signal is connected to the reset of all flip-flops.

The following Wokwi project demonstrates this setup: wokwi.com/projects/36434780766. It contains an Arduino Nano that decodes the currently selected mux address and displays it on a 7-segment display. Click on the button labeled `RST_N` to reset the counter, and click on the button labeled `INC` to increment the counter.

The Spine

The controller and all muxes are connected together through the spine. The spine has the following signals going to it:

From controller to mux:

- `si_ena` — the `ena` input
- `si_sel` — selected design address (10 bits)
- `ui_in` — user clock, user `rst_n`, user_inputs (10 bits)
- `uio_in` — bidirectional I/O inputs (8 bits)

From mux to controller:

- `uo_out` — user outputs (8 bits)
- `uio_oe` — bidirectional I/O output enable (8 bits)
- `uio_out` — bidirectional I/O outputs (8 bits)

The only signal which is actually generated by the controller is `si_sel` (using `sel_rst_n` and `sel_inc`, as explained above). The other signals are just going through from/to chip I/O pads.

The Multiplexer

Each mux branch is connected to up to 16 designs. It also has 5 bits of a hard-coded address (each unit gets assigned a different address, based on its position on the die).

The mux implements the following logic: If `si_ena` is 1, and `si_sel` matches the mux address, we know the mux is active. Then, it activates the specific user design port that matches the remaining bits of `si_sel`.

For the active design:

- `clk`, `rst_n`, `ui_in`, `uio_in` are connected to the respective pins coming from the spine (through a buffer)
- `uo_out`, `uio_oe`, `uio_out` are connected to the respective pins going out to the spine (through a tristate buffer)

For all others, inactive designs (including all designs in inactive muxes):

- `clk`, `rst_n`, `ui_in`, `uio_in` are all tied to zero
- `uo_out`, `uio_oe`, `uio_out` are disconnected from the spine (the tristate buffer output enable is disabled)

Pinout

Die Pad	QFN64 pin	Function	Signal
0	1	Mux Control	ctrl_ena
1	2	Mux Control	ctrl_sel_inc
2	3	Mux Control	ctrl_sel_rst_n
3	4	Reserved	—
4	5	Reserved	—
5	6	Reserved	—
6	7	Reserved	—
7	8	Reserved	—
8	EPAD	Ground	GND IO
9	9	Output	uo[0]
10	10	Output	uo[1]
11	11	Output	uo[2]
12	12	Output	uo[3]
13	13	Output	uo[4]
14	14	Output	uo[5]
15	15	Output	uo[6]
16	16	Output	uo[7]
17	17	Power	VDD IO
18	EPAD	Ground	GND IO
19	18	Analog	analog[0]
20	19	Analog	analog[1]
21	20	Analog	analog[2]
22	21	Analog	analog[3]
23	22	Analog	analog[4]
24	23	Analog	analog[5]
25	24	Power	PWR Analog
26	EPAD	Ground	GND Analog
27	25	Analog	analog[6]
28	26	Analog	analog[7]
29	27	Analog	analog[8]
30	28	Analog	analog[9]
31	29	Analog	analog[10]
32	30	Analog	analog[11]
33	EPAD	Ground	GND Core
34	31	Power	VDD Core

Die Pad	QFN64 pin	Function	Signal
35	EPAD	Ground	GND IO
36	32	Power	VDD IO
37	33	Bidirectional	uio[0]
38	34	Bidirectional	uio[1]
39	35	Bidirectional	uio[2]
40	36	Bidirectional	uio[3]
41	37	Bidirectional	uio[4]
42	38	Bidirectional	uio[5]
43	39	Bidirectional	uio[6]
44	40	Bidirectional	uio[7]
45	EPAD	Ground	GND IO
46	41	Input	ui[0]
47	42	Input	ui[1]
48	43	Input	ui[2]
49	44	Input	ui[3]
50	45	Input	ui[4]
51	46	Input	ui[5]
52	47	Input	ui[6]
53	48	Input	ui[7]
54	49	Input	u_rst_n †
55	50	Input	u_clk †
56	EPAD	Ground	GND IO
57	51	Power	VDD IO
58	52	Analog	analog[12]
59	53	Analog	analog[13]
60	54	Analog	analog[14]
61	55	Analog	analog[15]
62	EPAD	Ground	GND Analog
63	56	Power	PWR Analog
64	57	Analog	analog[16]
65	58	Analog	analog[17]
66	59	Analog	analog[18]
67	60	Analog	analog[19]
68	61	Analog	analog[20]
69	62	Analog	analog[21]
70	EPAD	Ground	GND Core
71	63	Power	VDD Core

Die Pad	QFN64 pin	Function	Signal
72	EPAD	Ground	GND IO
73	64	Power	VDD IO

† Internally, there's no difference between `u_clk`, `u_rst_n`, and `ui` pins. They are all just bits in the `pad_ui_in` bus. However, we use different names to make it easier to understand the purpose of each signal.

Sponsors

GF180mcuD support for Tiny Tapeout was funded by Tillitis and Wit.

The manufacturing of Tiny Tapeout GF 0p2 silicon was funded by wafer.space as part of their mission to support open source silicon.

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
- **Patrick Deegan** for PCBs, software, documentation and lots more
- **Sylvain Munaut** for help with scan chain improvements
- **Mike Thompson** and **Mitch Bailey** for verification expertise
- **Tim Edwards** and **Harald Pretl** for ASIC 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 makes**
- **Jeff** and the **Efabless Team** 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

Using This Datasheet

Structure

Projects are ordered by their mux address, in ascending order. Documentation is user-provided from their GitHub repositories and are merged into the final shuttle once the deadline is reached.

In general, each project should contain:

- The user-provided title & a list of authors
- A link to the GitHub repository used for submission
- A link to the Wokwi project (if applicable)
- A "How it works" section
- A "How to test" section
- An "External hardware" section (if applicable)
- A pinout table for both digital & analog designs

Badges

This datasheet uses "badges" to quickly convey some information about the content. These badges are explained in the table below.

Badge	Description
Artwork	Used to showcase artwork from our community.
0123 0423/2	Mux address of the project, in decimal. For microtile designs, their sub-address is placed after the forward slash. In this example, it would be 2.
25.175 MHz	Clock frequency of the project. May be truncated from actual value or omitted completely.
HDL Project Wokwi Project Analog Project	Project type, indicating if it was made with a HDL, Wokwi, or if it is analog.
Medium Danger High Danger	Indicates the risk that the project presents to the ASIC. Medium danger projects can damage the ASIC under certain conditions, whilst high danger projects *will* damage the ASIC.

Callouts

In addition to `Medium Danger` and `High Danger` badges being used, a callout is placed before the project documentation begins to alert the user.

A callout for `Medium Danger` may look something like:

This project will damage the ASIC under certain conditions.

There is an error in the schematic which may lead to ASIC failure under certain clocking conditions.

Similarly, a callout for `High Danger` may look something like:

This project will damage the ASIC.

There is an error in the schematic which may cause permanent damage when powered on in a certain configuration.

Should there be a project that poses a danger, the callout will explain the reasoning behind the danger level.

Callouts may also provide some additional information, and look something like so:

Information

Silicon melts at 1414°C, and boils at 3265°C. Don't let your chip get too hot!

Figures & Footnotes

Numbering for figures and footnotes within the “Project” chapter is formed by combining the address of the project with the current figure number. For example, the second figure for a project with an address of 256 will be captioned with “Figure 256.2”. Likewise, the third footnote for a project of address 128 will be shown as “128.3”.

The numbering outside of the “Project” chapter resumes as normal, being formatted with a simple number, e.g. “Figure 3”.

Updates

This datasheet is intended to be a living and breathing document. Please update your projects’ datasheet with new information if you have it, by creating a pull request against the shuttle repository.

Where is *your* design?

Go from idea to chip design in minutes, without breaking the bank.

Interested and want to get your own design manufactured? Visit our website and check out our educational material and previous submissions!

How?

New to this? Use our basic Wokwi template to see what's possible. If you're ready for more, use our advanced Wokwi template and unlock some extra pins.

Know Verilog and CocoTB? Get stuck in with our HDL templates.

When?

Multiple shuttles are run per year, meaning you've got an opportunity to manufacture your design at any time.

Stuck? Need help? Want inspiration?

Come chat to us and our community on Discord! Scan the QR code below.

Website	Digital design guide	Discord server
tinytapeout.com	tinytapeout.com/digital_design	tinytapeout.com/discord