Tiny Tapeout GF 26b Datasheet

github.com/TinyTapeout/tinytapeout-gf-26b

June 22, 2026

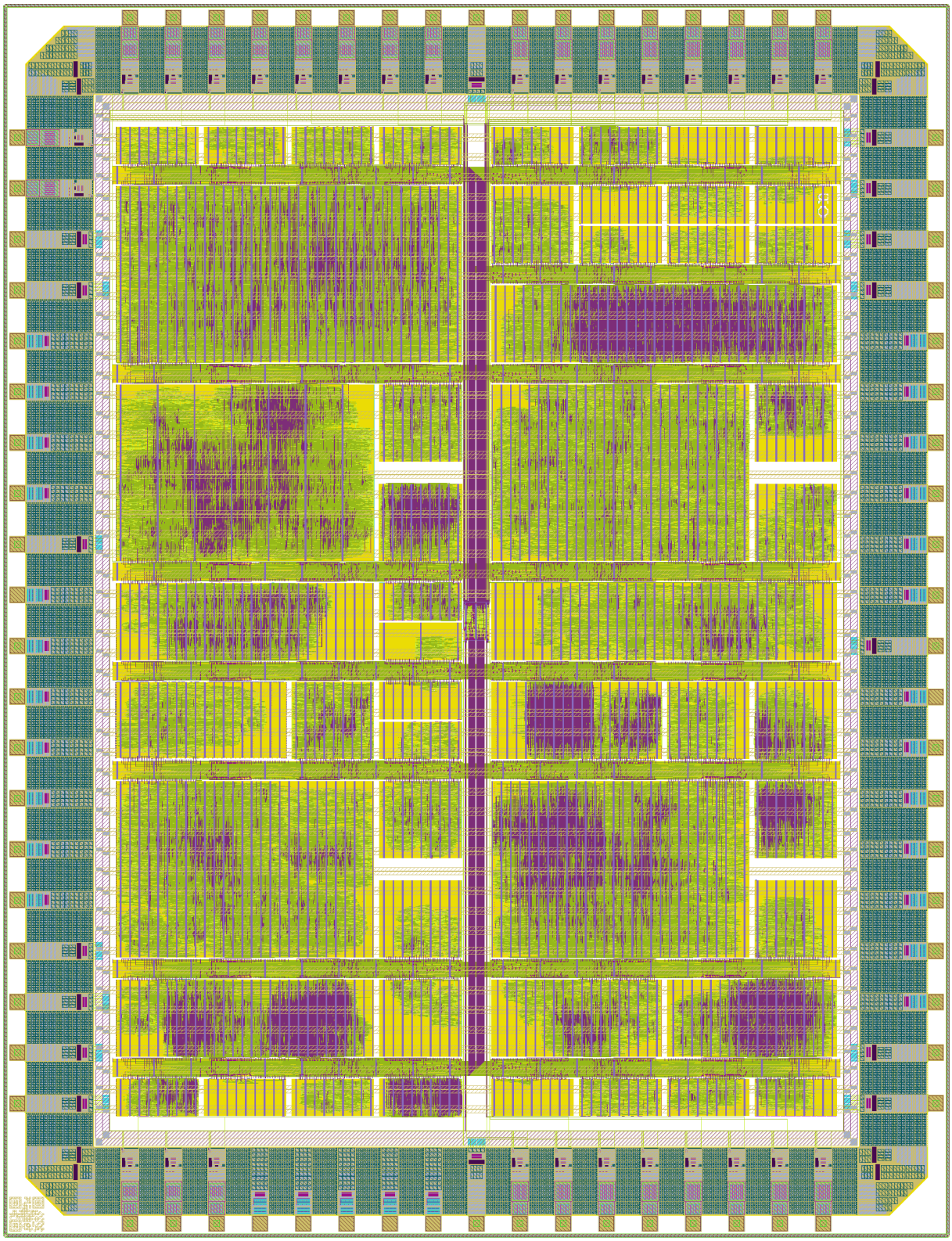
Table of Contents

Chip Renders

Online chip viewer

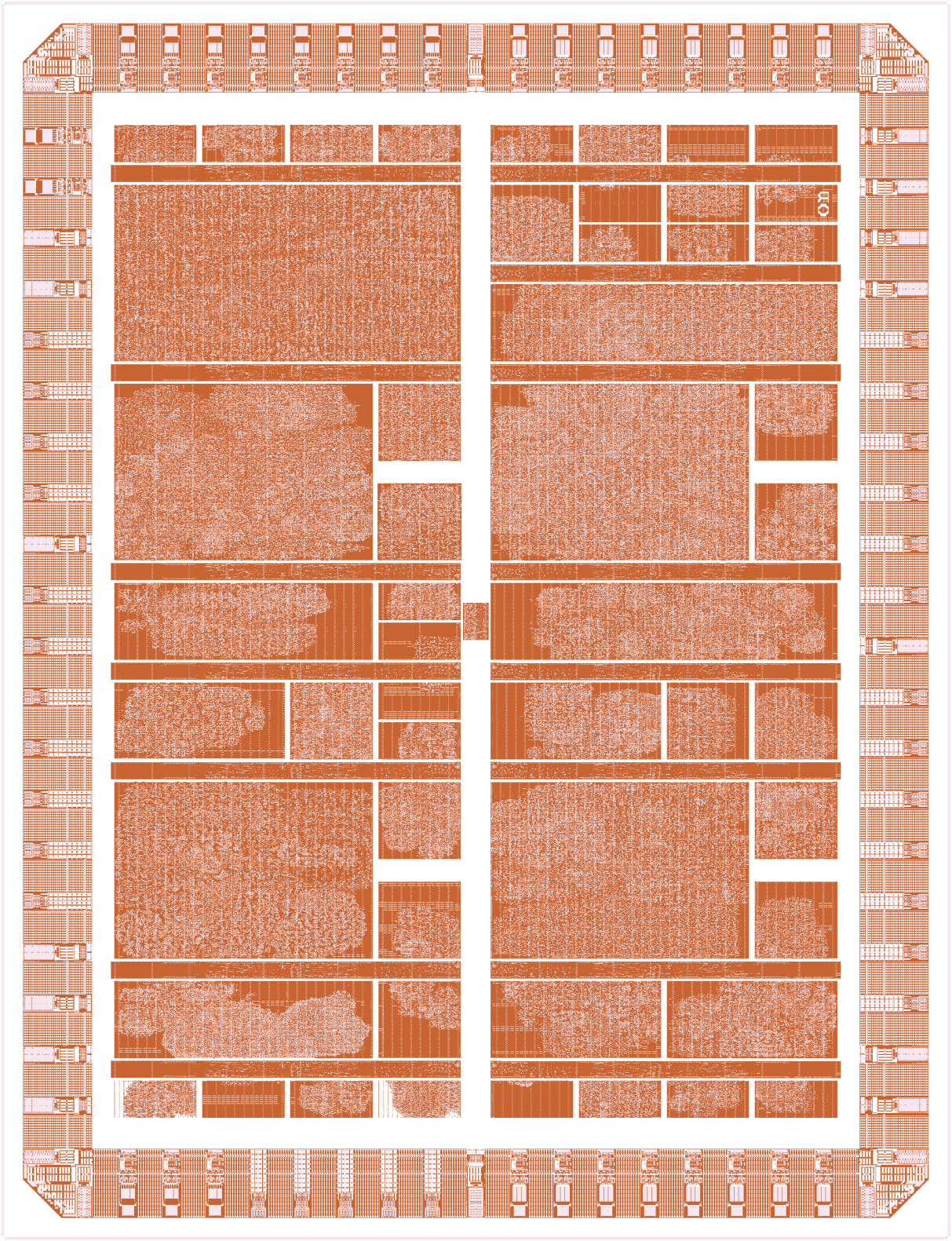
tinytapeout.com/
decap/ttgf26b

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]

TT-Arrakeen-SPSRAM-direct-sramrules

by **Staf Verhaegen**

0003 66 MHz HDL Project

github.com/FibraServiTT/TTGF26b_Arrakeen_SPSRAM_direct_sramrules

"Single port SRAM with pins connected directly to TT tile pins."

How it works

This design contains a single port SRAM block with pins connected directly to TT tile pins. This allows to use this design directly as a SRAM block. This design is for 3.3V and using the SRAM design rules for the bit cell.

The included block has 128 words of 8 bits. The dimension is 169.80µm by 83.42µm. These are the pins for the block:

- a (7 bit): address
- we (1 bit): write enable signal indicating a read or write operation
- d (8 bit): input data
- q (8 bit): output data
- clk: clock for performing an operation

On each rising edge of the clock an operation is performed on the memory. A read is done when `we` is 0, while a write operation is done when it is 1. On the rising edge of the clock the `a` and `d` signals are latched into an internal buffer. For a read operation the data for the provided address is put into the `q` signal, the `d` signal is ignored. For a write operation the value of the `d` signal is put in the given address. The write operation is write-through meaning that also `q` will get the value of `d` during the operation.

How to test

You can test the block yourself by providing the right inputs for a read or write operation. One can check if data written to a certain location is later on read back with a read operation on the same address.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a[0]	q[0]	d[0]
1	a[1]	q[1]	d[1]
2	a[2]	q[2]	d[2]

#	Input	Output	Bidirectional
3	a[3]	q[3]	d[3]
4	a[4]	q[4]	d[4]
5	a[5]	q[5]	d[5]
6	a[6]	q[6]	d[6]
7	we	q[7]	d[7]

spiPWMio

by **Sönke Appel**

0005 | 12 MHz | HDL Project

`github.com/FHW-Appel/TTgf26a-spiPWMio`

> *"A PWM input and PWM generator are configurable via input pins, readable via output pins, and accessible via SPI for read and write operations."*

How it works

This module provides a PWM generator on output pin 7 and a PWM reader on input pin 7. The PWM generator duty cycle is set by input pins [6:0]. The PWM reader output is reflected on output pins [6:0]. The period is assumed to be 20 ms, and the duty cycle can be adjusted between 1 ms and 2 ms.

When the SPI interface is not used, the module behaves as described above. When SPI is used, its behavior can be configured as described in `docs/specification.md`.

How to test

Loop back the generated PWM signal to the PWM input pin, then verify that the PWM configuration is reflected on the output pins.

External hardware

The design can be tested using a development kit. Optionally, use an oscilloscope to monitor the PWM signal and a signal generator to provide a test input on the PWM input pin.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ipins[0]	opins[0]	—
1	ipins[1]	opins[1]	—
2	ipins[2]	opins[2]	—
3	ipins[3]	opins[3]	—
4	ipins[4]	opins[4]	spi_cs_n
5	ipins[5]	opins[5]	spi_mosi
6	ipins[6]	opins[6]	spi_miso

#	Input	Output	Bidirectional
7	pwm_in	pwm_sig	spi_sck

ECC Scalar Multiplication

by **Dhanush Kulkarni**

0007 | 50 MHz | HDL Project

`github.com/CambridgeinstitutetechnologyBLR-Dhanush/ECC_Scalar_Multiplication`

"Compact 8-bit ECC Scalar Multiplication Engine demonstrating scalar-point arithmetic for Tiny Tapeout."

How it works

This project implements a compact ECC (Elliptic Curve Cryptography) Scalar Multiplication engine for Tiny Tapeout.

The design accepts an 8-bit scalar value (k) through the dedicated input pins and an 8-bit point coordinate (P) through the bidirectional input pins. On every rising edge of the clock, the circuit performs a simplified scalar multiplication operation:

$Q = k \times P$

The result is stored in an internal register and provided on the output pins. This project demonstrates the core concept of scalar multiplication used in ECC systems while remaining small enough to fit within a 1×1 Tiny Tapeout tile.

Inputs

- `ui_in[7:0]` : Scalar value (k)
- `uio_in[7:0]` : Point coordinate (P)

Outputs

- `uo_out[7:0]` : Result of scalar multiplication (Q)

The computation is synchronized with the clock and can be reset using the active-low reset signal.

How to test

1. Apply reset by setting `rst_n = 0`.
2. Wait a few clock cycles.
3. Release reset by setting `rst_n = 1`.
4. Apply an 8-bit scalar value on `ui_in`.
5. Apply an 8-bit point value on `uio_in`.
6. Wait for one clock cycle.
7. Observe the result on `uo_out`.

Example Test Cases

Scalar (k)	Point (P)	Output (Q)
5	3	15
10	4	40
20	30	88

Note: Outputs are limited to 8 bits.

External hardware

No external hardware is required.

The project can be tested entirely using Tiny Tapeout simulation, gate-level simulation, or FPGA-based verification environments.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Scalar bit 0	Result bit 0	Point bit 0
1	Scalar bit 1	Result bit 1	Point bit 1
2	Scalar bit 2	Result bit 2	Point bit 2
3	Scalar bit 3	Result bit 3	Point bit 3
4	Scalar bit 4	Result bit 4	Point bit 4
5	Scalar bit 5	Result bit 5	Point bit 5
6	Scalar bit 6	Result bit 6	Point bit 6
7	Scalar bit 7	Result bit 7	Point bit 7

8-bit Interactive ALU

by **Matanel Kadosh**

0032 10 MHz HDL Project

github.com/matanelk96/ASIC-ALU-MK

"An 8-bit Arithmetic Logic Unit with internal registers and multiplexed output for result/carry."

How it works

This project is an **8-bit Interactive Arithmetic Logic Unit (ALU)** designed for real-time calculation and hardware verification. The core is built around a two-register architecture (Register A and Register B) and can execute 11 different mathematical and logical operations based on a 4-bit opcode.

To maximize the efficiency of the physical pins and fit within the standard Tiny Tapeout 3x PMOD footprint, the design implements an **Output Multiplexer**:

- The internal ALU calculates a 9-bit result to capture overflow/carry.
- The 8-bit output bus (`uo_out`) is multiplexed.
- By toggling the `Out_Sel` control pin, the user can switch the output bus to display either the 8-bit numerical result or the 9th bit (Carry Flag) on the LSB of the output bus.

The control bus is routed through the bidirectional pins (`uio_in`), which are hardware-locked to act strictly as inputs for safety.

How to test

The ALU is designed to be driven by a microcontroller or a Raspberry Pi.

1. **Initialization:** Assert the `rpi_reset` pin (`uio_in[7]`) HIGH to clear the internal registers.
2. **Load Data:** Apply an 8-bit value to the input data bus (`ui_in`). Pulse `Load A` (`uio_in[0]`) or `Load B` (`uio_in[1]`) HIGH then LOW to latch the data into the respective internal registers.
3. **Execute:** Apply a 4-bit opcode to `ALU_OP` (`uio_in[5:2]`). The ALU processes the operation combinationally.
4. **Read Result:** * Set `Out_Sel` (`uio_in[6]`) to `0` and read the 8-bit result on the output bus (`uo_out`).
    - Set `Out_Sel` to `1` and read `uo_out[0]` to check for an Arithmetic Carry or Overflow.

Supported Opcodes:

- `0000`: AND
- `0001`: OR
- `0010`: ADD (with Carry Out)
- `0011`: SUB
- `0100`: XOR
- `0101`: NAND
- `0110`: NOR
- `0111`: Shift Left
- `1000`: Shift Right
- `1001`: NOT A
- `1010`: NEG A (2's Complement)

External hardware

To fully interact with and verify the ALU, the following external hardware is recommended:

- **Host Controller:** A Raspberry Pi, Arduino, or any 3.3V microcontroller to drive the input data and control pins.
- **Logic Analyzer:** Highly recommended for debugging and verifying the timing of control signals (Load triggers, Opcodes) and the output bus during operation.
- **Connectivity:** Standard PMOD cables/jumpers to connect the host controller and testing equipment to the Tiny Tapeout demo board.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	Data In Bit 0 (LSB)	Result Bit 0 / Carry Flag (When Out_Sel=1)	Load A (Trigger)
1	Data In Bit 1	Result Bit 1	Load B (Trigger)
2	Data In Bit 2	Result Bit 2	ALU_OP [0]
3	Data In Bit 3	Result Bit 3	ALU_OP [1]
4	Data In Bit 4	Result Bit 4	ALU_OP [2]
5	Data In Bit 5	Result Bit 5	ALU_OP [3]
6	Data In Bit 6	Result Bit 6	Out_Sel (0=Result, 1=Flags)
7	Data In Bit 7 (MSB)	Result Bit 7 (MSB)	Reset (Active High)

60 Hz Grid-Forming ASIC with Dump-Load Control

by **Eric Pearson**

0033 | 48 MHz | HDL Project

`github.com/ericpearson1313/tt_grid_tie_load`

"a CORDIC-locked grid-forming reference + linear dump-load controller ASIC for driving micro-inverters"

A TinyTapeout ASIC for grid-aware AC generation and power balancing

This ASIC implements a **self-contained, grid-aware control loop** capable of generating a clean 60 Hz reference waveform while simultaneously regulating real-power flow using a **DC-link dump load**. It is designed for small AC micro-systems where PV inverters, a low-power grid-former, and a resistive dump load must coexist without external controllers.

The chip senses the AC and DC waveforms, compares DC to a Vref and AC to an internal CORDIC-generated reference, and adjusts a DC-side dump FET to maintain long-term phase and amplitude stability. All control is performed on-chip using add/shift arithmetic, a fast error accumulator, a slow IIR loop, and a minimum-pulse-width PWM engine.

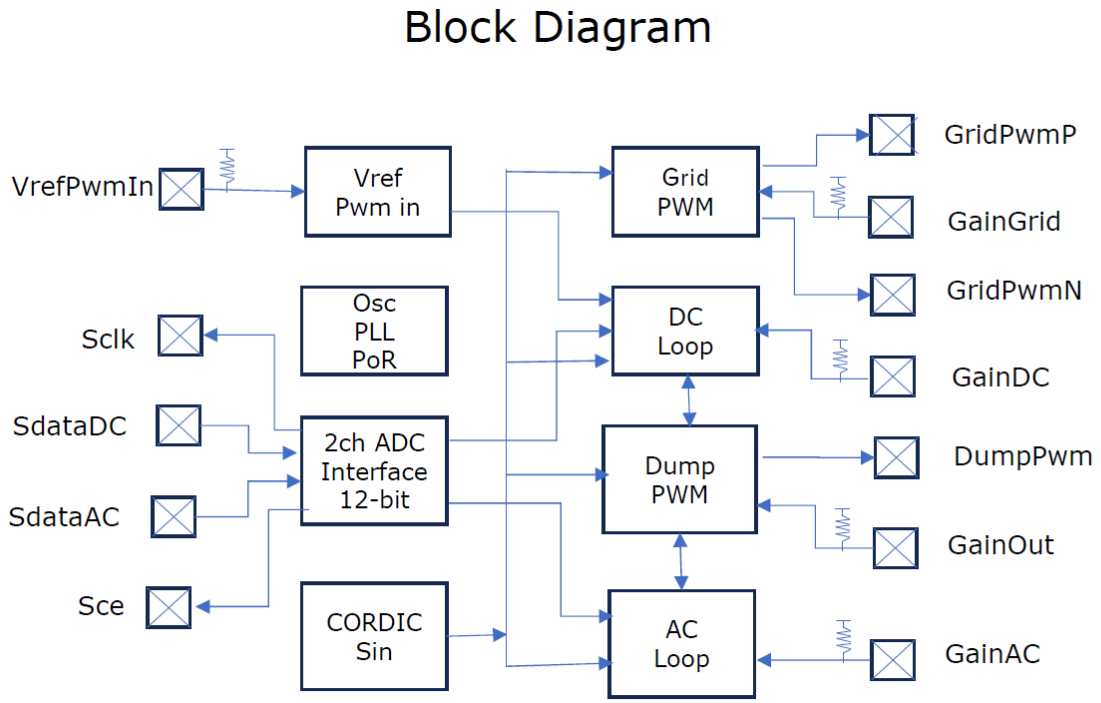


Figure 33.1: Block Diagram

-- ## **Core Features**

- **CORDIC-locked 60 Hz sine generator**
  Produces a stable, phase-accurate reference for a low-power H-bridge grid-former.
- **AC/DC sensing (dual ADC input)**
  Samples the AC and DC waveforms at 3 MHz and compares it to the internal reference.
- **AC/DC Dual Loop control**
  Forms linear and stable control loops without multipliers.
- **DC-link dump-load PWM output**
  Drives a single high-voltage FET with enforced **4 µs minimum ON/OFF** times.
- **Four real-time tuning gates**
  External PWM or logic-level inputs adjust loop behavior on the fly:
  - `dc_vref` — DC Link Reference voltage input
  - `gain_sine` — trims generated sine amplitude
  - `gain_out` — Output gain trim
  - `gain_ac` — AC gain
  - `gain_dc` — DC gain
  - `mode_ac` — select 1/4 cycle AC operation
- **Safe, simple power topology**
  Intended for use with a **rectified 240 V AC DC-link** (VFD-style front end) and a resistive dump load such as a water heater.

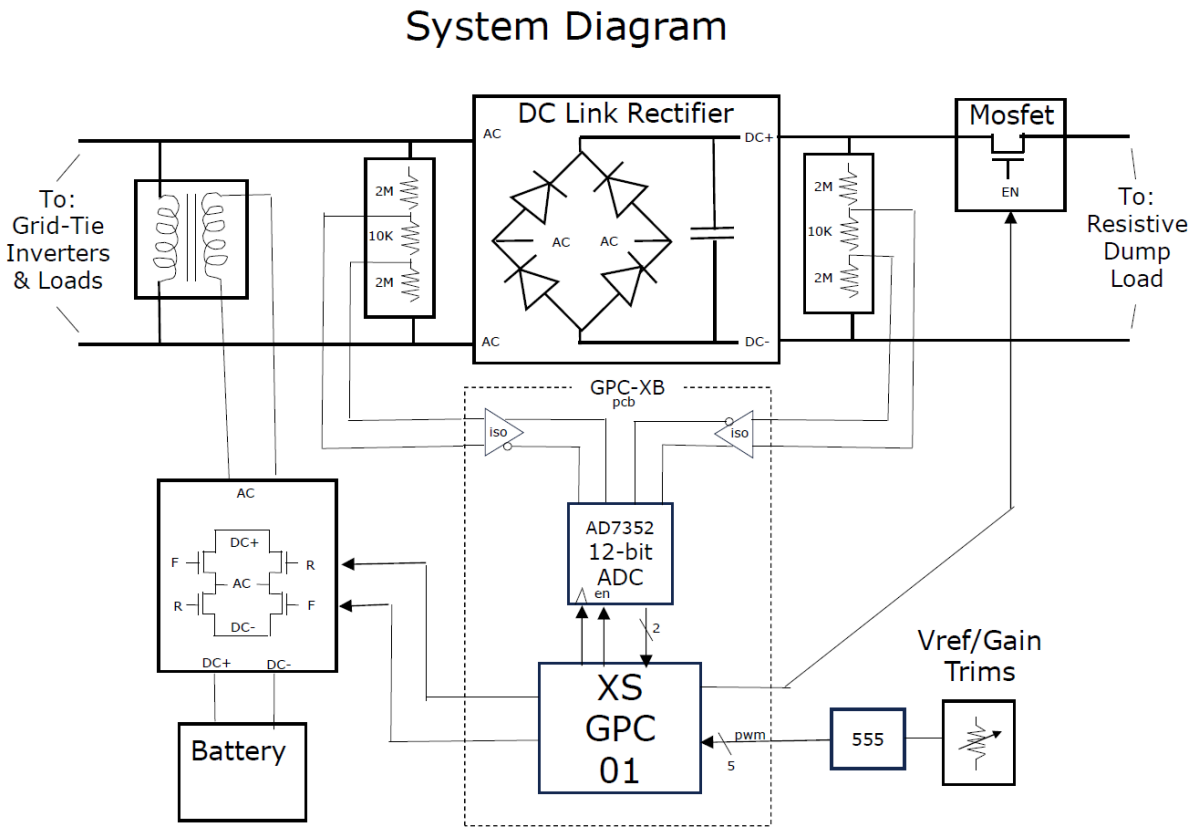


Figure 33.2: System Diagram

--

I/O Summary

Inputs (ui)

```
ui[0]  ac_sdata      # AC ADC serial data input (3 MHz sample
stream)
ui[1]  dc_sdata      # DC ADC serial data input (3 MHz sample
stream)
ui[2]  dc_vref       # DC Vref target voltage for DC Link
ui[3]  gain_sine     # Sine amplitude trim (generation gain)
ui[4]  gain_out      # Dump-PWM gain trim (max dump power)
ui[5]  gain_ac       # AC Gain trim
ui[6]  error_dc      # DC Gain trim
ui[7]  ac_mode       # Select 1/4 cycle ac mode
```

Outputs (uo)

```
uo[0]  adc_cs          # ADC chip-select / sample strobe
uo[1]  gen_pwm_p       # Grid-former PWM (positive leg)
uo[2]  gen_pwm_n       # Grid-former PWM (negative leg)
uo[3]  dump_pwm        # DC-link dump FET PWM (4 µs min pulse
width)
uo[4]  ac_underflow    # phase error
uo[5]  ac_overflow     # ac gain too high
```

```
uo[6]  dc_underflow    # Insufficent power
uo[7]  dc_overflow     # Overpower, dc gain too high
```

--

Intended Use Case

This ASIC is designed for experimental AC micro-systems where:

- a **low-power grid-former** establishes the AC waveform
- **PV inverters** inject unpredictable power
- **AC Appliances** use less energy than available
- a **DC-link dump load** must absorb surplus energy
- the system must remain stable without external controllers

The chip maintains long-term phase and amplitude alignment by modulating the dump load based solely on AC-side sensing.

--

Status

- Preliminary Datasheet
- RTL complete
- Clean synthesis
- Verified P&R on **1×2 tile**
- Ready for TinyTapeout submission
- verification tests
- Ported to Forge 1k fpga
- Max10 Fpga Simulation Environment

The Experimental Board (XB) makes building practical systems easier

Experiment Board

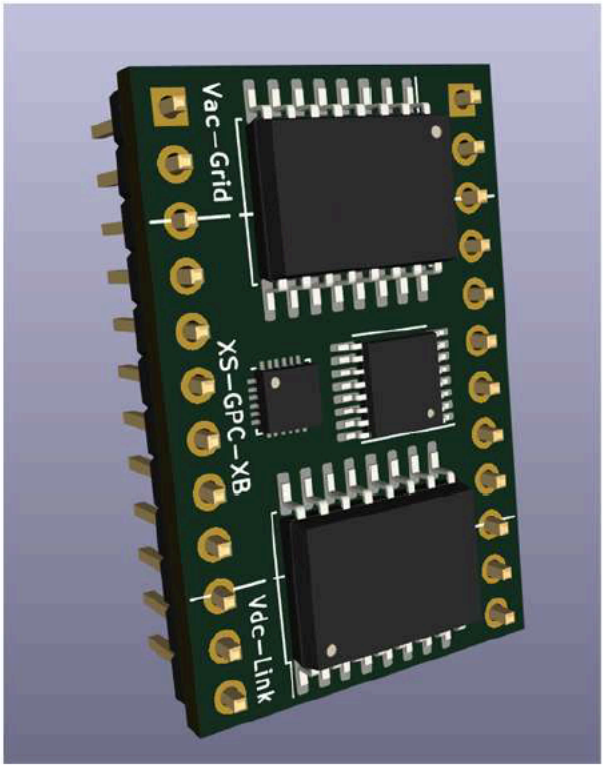


Figure 33.3: XB - Experiment Board

The Development Board (DB) is used to develop and bring up the GPC chip and XB module by giving full chip IO access, as well as an example system design suitable for benchtop experiments with provisions to scale up in power with detachable AC and DC power modules. Pmods mapped to the Tiny Tapeout pinout provide a direct plug in path for when tapeouts return.

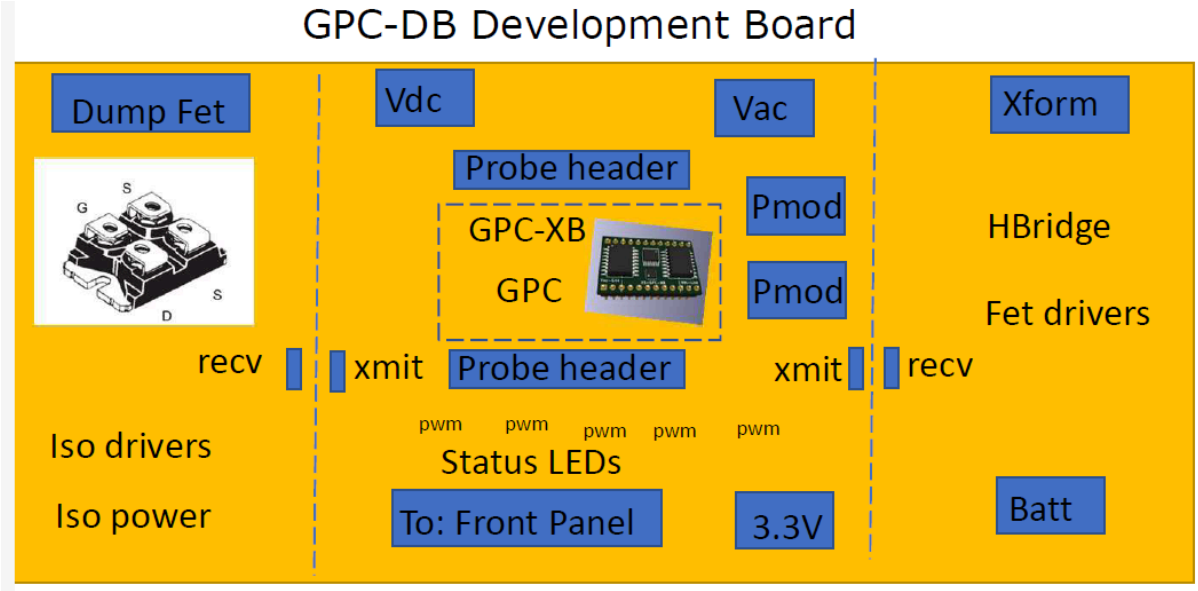


Figure 33.4: DB - Delvelopment Board

Why is it?

Use my 10kw grid-tied solar system to power my home on the second day of a power outage.

A grid tied solar system becomes useless without the grid. Hybrid systems involve using batteries fix this, but are a big expense. If a grid tied solar system is disconnected from the grid and provided with a simulated grid the solar system will generate hydro AC. The issue is that grid tied inverters work by maximizing the energy delivery without restriction knowing that the grid can accept it, and it will vary with available sunlight. If the energy is not dissipated the voltage and frequiency of the simulated grid will be driven out of spec and the grid tied inverters will shutdown (usually for at least 5 min).

The energy from the sun needs to be always and exactly dissipated. This dissipation can be partially done by any electrical devices in the home, but something else needs dissipate the remainder. Heating water is a good way of dumping energy.

A semiconductor chip is proposed which will generate a reference 60Hz AC, and control dumping extra energy into a resistive load without needed a battery system (ref Datasheet). I design up a minimum usable Schematic and PCB board (kicad files in /pcb).

Its a good fit for a tiny tapeout chip with low I/O count PWM and serial data, and 20ns PWM edge resolution gives fine control, while maintaiing minimum pulse widths. It also fits a forge 1k otp part.

Fitting this device into a tiny cost would remove all cost from the control part of the problem. Mounted on a little experimental PCB board it would be a $30 control solution.

How it works

A free running angle counter is input into a cordic rotational block and polarity corrected to calculate a 60Hz sine wave. The sine wave is gated and then accumulated in PWM modulator produce bi-polar PWM signals which can be used to drive an H Bridge and the low side of a transformer, with the high side providing the grid reference. The ‘grid’ is rectified into a DC Link, with PWM switching into a resistive load. The DC and AC ‘grid’ voltages are sampled by ADC. AC is compared to the sine wave while DC is compared to a PWM provided DC Vref. AC and DC Loops accumulate the pseudo-energy (dv*dt) error, compared against a positive thresholds and used to gate |sin| to drive a dump FET while guaranteeing minimum PWM pulse widths (4us).

How to test

```
make -B FST=
```

External hardware

It will need a real or model system to test:

- Grid-tied solar system, sunlight
- Hbridge and drivers
- Step up transformer
- Bridge Rectifier, DCLink Capactors
- Dump FET and driver
- Resistive Water Heater, water
- ADC with isolated instrumentation.
- external control panel (pwm base loop controls))

disclaimer

This ASIC is an experimental research design and is not intended for direct connection to mains power or use in certified grid-tie systems.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	adc_sdata0	adc_cs	—
1	adc_sdata1	grid_pwm_p	—
2	gain_sine	grid_pwm_n	—
3	gain_out	dump_pwm	—
4	gain_ac	ac_underflow	—
5	gain_dc	ac_overflow	—
6	gain_vdc	dc_underflow	—

#	Input	Output	Bidirectional
7	mode_ac	dc_overflow	—

Neuromorphic Spike Codec (GF180)

by **Prof. Santhosh Sivasubramani** & **IIT Delhi**

0034 | 25 MHz | HDL Project

github.com/SanthoshSivasubramani/tt_um_santhosh_spike_codec_gf_pub

> "*GF180mcuD digital neuromorphic spike codec: three encoders (rate via 8-bit Fibonacci LFSR, latency/time-to-first-spike, delta with programmable threshold) and three symmetric decoders (rate count, latency-to-value, signed delta-sigma accumulator with leak) in one SPI-programmable block; 1x2 tile, single-spike and bipolar spike outputs, window tick, busy/irq, 25 MHz signoff clock*"

A compact 1×1-tile **neuromorphic spike codec** for TinyTapeout GF 26a that combines three encoders and three symmetric decoders in a single SPI-programmable block.

The block operates on a shared 8-bit window counter (`WINDOW_N`, 1..255). Encoder and decoder modes are selected independently via the CTRL register; both sides may be enabled simultaneously, e.g. an encoder upstream of an external interconnect under test and a decoder on the return path.

Encoders (`enc_mode`)

- **Rate (2'b00)** — each cycle of the active window, an 8-bit maximal Fibonacci LFSR (taps 7, 5, 4, 3) is compared against `enc_in`; if `lfsr < enc_in`, `uo_out[0] = spike_pos` fires for that cycle. Produces a Bernoulli rate code whose expected spike count is `WINDOW_N × enc_in / 256`.
- **Latency (2'b01)** — on `enc_in` write (or external `start_ext`), the block loads a countdown of `255 − enc_in` cycles. A single `spike_pos` fires when the countdown reaches zero, yielding a time-to-first-spike code where larger values fire earlier.
- **Delta (2'b10)** — maintains a running 8-bit reference. Writing `enc_in` triggers a comparison: if the signed difference exceeds `+DELTA_THR`, a `spike_pos` fires and the reference moves up by one threshold; if it is below `−DELTA_THR`, a `spike_neg` fires and the reference moves down.

Decoders (`dec_mode`)

- **Rate (2'b00)** — counts rising edges of `ui_in[0]` over the window, saturates at 255, latches into DEC_OUT at window rollover.
- **Latency (2'b01)** — records the cycle index of the first `ui_in[0]` rising edge in the window and publishes `255 − index` (0 if no spike seen).
- **Delta (2'b10)** — maintains a signed 16-bit accumulator driven by `spike_pos = ui_in[0]` and `spike_neg = uio_in[6]`, adding / subtracting

DELTA_STEP per spike and applying DELTA_LEAK per cycle toward zero. DEC_OUT is the offset-binary saturated high byte of the accumulator.

Both encoder and decoder provide sticky status latches (W1C on STATUS) plus a combined irq output so a small host can service spikes only at window boundaries.

How to test

1. **Reset** (rst_n = 0 for ≥ 10 clocks).
2. Program over SPI (16-bit frame {R/Wn, 7-bit addr, 8-bit data}, mode 0):
   - WINDOW_N (0x01): desired window size.
   - DELTA_THR (0x02), DELTA_STEP (0x03), DELTA_LEAK (0x04) as needed for delta modes.
   - LFSR_SEED (0x05) for a deterministic rate code (0 clamps to 1).
3. Write CTRL (0x00) with enc_en=1 / dec_en=1 and the desired mode bits to start the block.
4. **Encode**: stream values into ENC_IN (0x06). Observe spikes on uo_out[0] (pos) and uo_out[1] (neg). uo_out[2] = enc_valid pulses at window end or latency fire.
5. **Decode**: feed spikes into ui_in[0] (and uio_in[6] for delta neg). Poll DEC_OUT (0x13) after uo_out[3] = dec_valid pulses, or freeze the decoder by writing dec_en = 0 to snapshot the in-flight state into DEC_OUT.
6. Poll STATUS (0x20) for sticky spike / valid flags and write 1 to a bit to clear it (W1C semantics).

External hardware

- SPI master (Raspberry Pi / microcontroller) for register programming and readback of DEC_OUT / STATUS.
- Any spike source on ui_in[0] (and optionally uio_in[6] for bipolar delta decoding) — logic analyser, function generator, or another TTGF tile.
- Optional: pair with tt_um_santhosh_snn_wta_gf (tile 4) — route spike-codec encoder outputs into the SNN inference core, decode the winner spikes back into an 8-bit signed value.

Register summary

Addr	Name	R/W
0x00	CTRL	W
0x01	WINDOW_N	W
0x02	DELTA_THR	W
0x03	DELTA_STEP	W
0x04	DELTA_LEAK	W

0x05	LFSR_SEED	W
0x06	ENC_IN	W
0x07	DEC_RESET	W
0x10	ENC_PULSE	R
0x11	ENC_STATE	R
0x12	ENC_WIN_CNT	R
0x13	DEC_OUT	R
0x14	DEC_STATE_L	R
0x15	DEC_STATE_H	R
0x20	STATUS	R/W1C

CTRL bits

Bit	Name	Meaning
0	enc_en	Enable encoder
1	dec_en	Enable decoder (falling edge snapshots DEC_OUT)
3:2	enc_mode	00=rate, 01=latency, 10=delta, 11=reserved
5:4	dec_mode	00=rate, 01=latency, 10=delta, 11=reserved
6	start_enc	Software re-arm of latency / delta encoder

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	spike_in (decoder external spike)	spike_pos	spi_cs_n
1	start_ext (external encoder re-arm pulse)	spike_neg	spi_mosi
2	reserved	enc_valid	spi_miso
3	reserved	dec_valid	spi_sck
4	reserved	busy	overflow_ever
5	reserved	any_spike	irq
6	reserved	window_tick	spike_neg_in (delta decoder negative channel)
7	reserved	lfsr_msb	reserved

Neural Spike Detector

by **Layla Adeli**

0036 | 10 MHz | HDL Project

`github.com/laylaadeli7/tt_neural_spike_detection`

"Adaptive neural spike detector with IIR bandpass filter (300-3000 Hz), Nonlinear Energy Operator (NEO), and EWMA adaptive threshold. Digital backend for neural recording AFEs."

How it works

The design implements a real-time neural spike detector. The 8-bit signed input sample is passed through a 2nd-order IIR bandpass filter (300-3000 Hz) which then isolate the neural spike band. The filtered signal feeds a Nonlinear Energy Operator (NEO), which finds psi[n] = x[n-1]^2 - x[n]*x[n-2], amplifying spike energy relative to background noise. An adaptive threshold is computed as k* sigma^2, and sigma is tracked via an exponential weighted moving average (EWMA) of the signal's absolute value, and k is a configurable multiplier (default is set to 5) set via a simple SPI interface. When the NEO output crosses the adaptive threshold, a spike is flagged and a 7-bit rolling timestamp is output, followed by a configurable refractory period (default 50 samples) to prevent re-triggering on the same spike waveform. This is the very first revision, and my first experience with not just Tiny Tapeout, but also with Verilog for a project such as this. If there are bugs, I am planning on submitting a more in depth version in a future tapeout once I have gained more exposure. :)

How to test

Want to apply an 8-bit signed sample to ui_in on each clock_en pulse (internally divided down from the system clock to 10 kHz). Then, feed in real or synthetic neural recording data containing spike waveforms superimposed on background noise, and monitor uo_out[7] for the spike-detected pulse and uo_out[6:0] for the timestamp. Configure the threshold multiplier and refractory period via the 3-wire SPI interface on uio[2:0] (CS_n, SCLK, MOSI). A cocotb testbench (test/test_spike_detector.py) verifies spike detection, false-positive rejection on pure noise, refractory period behavior, and SPI configuration.

External hardware

External hardware can include ADC or microcontroller that sends data for the input. Also would want either external microcontroller or use the TT board

given to capture the pulses at the output. Overall, some sort of analog to digital block, and signal generator would be ideal. In initial testing I used a Python script to generate synthetic neural data, so that could be another solution without an external ADC.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ADC sample bit 0 (LSB)	Timestamp bit 0 (LSB)	SPI SCLK (input)
1	ADC sample bit 1	Timestamp bit 1	SPI MOSI (input)
2	ADC sample bit 2	Timestamp bit 2	SPI CS_n (input, active low)
3	ADC sample bit 3	Timestamp bit 3	Debug: filtered signal MSB (output)
4	ADC sample bit 4	Timestamp bit 4	—
5	ADC sample bit 5	Timestamp bit 5	—
6	ADC sample bit 6	Timestamp bit 6 (MSB)	—
7	ADC sample bit 7 (MSB, sign)	Spike detected (1-cycle pulse)	—

Lightscan

by **Emilio Perez Juarez**

0038 | 12 MHz | HDL Project

`github.com/EmilioPeJu/ttgf-lightscan`

> "*BISS-C Position acquisition system for a scan with pulsed output.*"

How it works

It contains a BISS-C master to obtain position from an absolute encoder, it also produces a pulse output to synchronize an external equipment trigger with the position acquisition. The SPI register interface must be used with CPOL=1 and CPHA=1, the frame structure is cmd(8 bits) + data(32 bits), in which the top bit of cmd indicates if it is a read(0) or a write(1), the remaining bits of cmd indicates the register number.

Some extra features available are:

- BISS-C delay compensation.
- Spare UIO control via registers which can be synchronized with the trigger.

Registers

REG0(RO): Magic ID 0xCAFEA51C

REG1(RO): Counters, number of positions acquired(bits 15 downto 0), number of position errors(bits 23 downto 16), number of position overruns(bits 31 downto 24).

REG2(RO): Last position acquired.

REG3(RW): Trigger period(minus one), separation in ticks between triggers.

REG4(RW): Number of triggers(minus one).

REG5(RW): Pulse width(minus two).

REG6(RW): BISS-C half clock period(minus one).

REG7(RW): BISS-C clock number of rising edges(minus one).

REG8(RO): High bits of last position acquired.

REG9(WO): Action register, bit 0 gets a single position, bit 1 starts acquisition, bit 2 produces a single trigger.

REG10(RW): When read, spare IO Input value during last trigger, when written, it controls outputs(set after next trigger), the value is set in the first byte, the direction is set in the second byte(0=input, 1=output).

REG11(RW): When read, spare IO Input value during, when written, it controls outputs, the value is set in the first byte, the direction is set in the second byte(0=input, 1=output).

How to test

1. Plug a BISS-C encoder via a RS422 transceiver (with clock as output and data as input).
2. Optionally plug an external instrument to the pulse output.
3. Configure register, for example, at frequency 12MHz, the following register writes configures a 10s acquisition at 0.1s intervals, BISS-C bitrate at 1MHz and an encoder that has a 14 bits position: REG3 = 1199999; REG4 = 99; REG6 = 5; REG7 = 26;
4. Start the acquisition by writing REG9 = 2
5. Collect the position from REG2 whenever there is a position ready event, this can be detected using an interrupt.

External hardware

- A RS422 transceiver for the BISS-C absolute encoder.
- Anything that could be triggered by a pulse, e.g. a camera.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SPI CS_N	RUNNING	BISS-C CLK
1	SPI CLK	PULSE	BISS-C DATA
2	SPI MOSI	POSITION READY EVENT	IO2
3	—	SPI MISO	IO3
4	—	—	IO4
5	—	—	IO5
6	—	—	IO6
7	—	—	IO7

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

ChaCha20

by **Raphael Eguchi**

0039 35 MHz HDL Project

`github.com/egurapha/ttgf-chacha20`

"RFC 8439 ChaCha20 stream-cipher core with a UART or parallel host interface"

How it works

This is a hardware implementation of the **ChaCha20 stream cipher** as specified in RFC 8439: a 256-bit key, a 96-bit nonce, and a 32-bit block counter. ChaCha20 produces a keystream that is XORed with the data, so encryption and decryption are the same operation.

The design is a command-driven peripheral. A host loads the key, nonce, and counter, then issues one of two operations:

- **GEN**: emit raw keystream bytes.
- **CRYPT**: XOR a stream of data bytes with the keystream (encrypt or decrypt).

It has three blocks:

- **`chacha20_core`** computes the ChaCha20 block function. It holds the 16-word (512-bit) state, runs the 20 rounds through four parallel quarter-round units (one Add-Rotate-XOR step per clock), then adds the original state in. The controller reads the result one 32-bit word at a time, so no wide keystream bus is materialised.
- **`chacha20_controller`** is the command FSM. It decodes the command byte, collects the payload, drives the core, and streams keystream/ciphertext bytes back out. It speaks a transport-agnostic byte interface (one byte in with a valid strobe, one byte out with a busy/send handshake).
- **A host front-end**, selected at runtime by the **MODE** pin (`uio[3]`):
  - `MODE = 0`: a **UART** (8N1) at baud = clock / 200.
  - `MODE = 1`: a synchronous **parallel byte bus** (one byte per clock).

Both front-ends present the identical byte interface to the controller, so the core and protocol are the same regardless of which one is used.

Performance

At the 35 MHz target clock, the core computes a 64-byte block in about 84 cycles (0.76 bytes/cycle), a theoretical ceiling of 27 MB/s. Deliverable throughput is set by the host interface:

- **Parallel (MODE = 1):** an estimated 8 MB/s for `GEN` at `HOLD_SEL = 0`, 6.5 MB/s at the default `HOLD_SEL = 1`. The workable `HOLD_SEL`, and hence the real rate, depends on output-pad settling and how fast the host reads. `CRYPT` is lower (each byte is a round trip).
- **UART (MODE = 0):** 175000 baud, giving 17.5 KB/s for `GEN` and 8.5 KB/s for `CRYPT`.

Functionally validated on the Tiny Tapeout FPGA breakout (iCE40 UP5K) over both interfaces: single- and multi-block `GEN`, `CRYPT`, decrypt round-trip, and command-error handling, checked against the reference model.

Command protocol

Every command is a single command byte, optionally followed by a fixed-size payload. After a command completes, **BUSY** returns low; wait for that before sending the next command.

Command	Byte	Payload	Effect
`LOAD_KEY`	0x01	32 bytes	Load the 256-bit key.
`LOAD_NONCE`	0x02	12 bytes	Load the 96-bit nonce.
`LOAD_CTR`	0x03	4 bytes (little-endian)	Load the 32-bit block counter.
`GEN`	0x04	1 byte `N`	Emit `N` × 64 keystream bytes.
`CRYPT`	0x05	2 bytes length `L` (little-endian) + data	For each of `L` data bytes in, return one XORed byte.

Key and nonce bytes are sent in natural order (byte 0 first); they map directly onto the RFC 8439 little-endian state layout. The block counter advances automatically across multiple 64-byte blocks within one `GEN` or `CRYPT`.

For `CRYPT`, the data phase is interleaved: send one plaintext byte, read one ciphertext byte, repeat for all `L` bytes. Decryption is the same command run on the ciphertext (and the same key/nonce/counter).

Status outputs

- **BUSY**: high while the controller is not idle.
- **ERR**: latches high if an unrecognised command byte is received; clears on reset.

How to test

Reset the chip by holding `rst_n` low for at least a few clock cycles, then releasing it. Pick an interface with the MODE pin and talk to it with the command protocol above.

A minimal `GEN` run (using a known key/nonce/counter) is:

1. `LOAD_KEY`: send `0x01` then the 32 key bytes.
2. `LOAD_NONCE`: send `0x02` then the 12 nonce bytes.
3. `LOAD_CTR`: send `0x03` then the 4 counter bytes (little-endian).
4. `GEN`: send `0x04` then `0x01` to request one block.
5. Read the 64 keystream bytes that stream back.

The output matches the ChaCha20 keystream for that key/nonce/counter (see the RFC 8439 test vectors, or `test/chacha20_ref.py` in the repository, which is the bit-exact reference the test suite checks against). To encrypt, use `CRYPT` (`0x05`, the 2-byte length, then the data); to decrypt, run `CRYPT` again on the ciphertext.

UART mode (MODE = 0)
The default interface. 8 data bits, no parity, 1 stop bit; baud = clock / 200 (175000 baud at the 35 MHz default clock). On the Tiny Tapeout demo board this connects to the RP2040 USB-serial bridge, so a PC can drive it directly.

- **RX** = `ui[3]` (host → chip)
- **TX** = `uo[4]` (chip → host)
- **BUSY** = `uo[0]`, **ERR** = `uo[1]`

Parallel mode (MODE = 1)
A faster byte-at-a-time interface for a host that shares the chip's clock.

- **Data in** = `ui[7:0]`; pulse **WR** (`uio[0]`) high for one cycle to write a byte. Gaps between bytes are fine: only WR-high cycles capture data.
- **Data out** = `uo[7:0]`; read it while **VALID** (`uio[1]`) is high.
- **BUSY** = `uio[2]`, **ERR** = `uio[6]`.
- **HOLD_SEL** (`uio[5:4]`) sets how long each output byte is held: `HOLD_SEL` + 1 clock cycles (1–4). Use a longer hold to give a latency-bound reader and the output pad more time to settle. The GF180 output pad's maximum toggle rate is not yet characterised, so raise HOLD_SEL on real silicon if a fast reader misses bytes.

Host requirements in parallel mode: drive MODE high before operating; pulse WR for exactly one cycle per byte; and wait for BUSY low between commands. The `CRYPT` data phase can be streamed back to back (one plaintext byte in, one ciphertext byte out) with no pacing at the 64-byte block boundaries: the controller holds a byte that arrives while it is recomputing the next keystream block.

External hardware

None required. In UART mode the design is driven over the Tiny Tapeout demo board's RP2040 USB-serial bridge from a PC. Parallel mode is optional

and is intended for a clock-synchronous host such as an RP2040 PIO program or an FPGA (for example via the Tiny Tapeout FPGA dev board or a PMOD).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PDIN0	BUSY / PDOUT0	PAR_WR (in)
1	PDIN1	ERR / PDOUT1	PAR_VALID (out)
2	PDIN2	PDOUT2	PAR_BUSY (out)
3	RX / PDIN3	PDOUT3	MODE (in: 1=parallel)
4	PDIN4	TX / PDOUT4	HOLD_SEL0 (in)
5	PDIN5	PDOUT5	HOLD_SEL1 (in)
6	PDIN6	PDOUT6	PAR_ERR (out)
7	PDIN7	PDOUT7	—

TT-Arrakeen-SPSRAM-direct-5V

by **Staf Verhaegen**

0065 66 MHz HDL Project

`github.com/FibraServiTT/TTGF26b_Arrakeen_SPSRAM_direct_5V`

"Single port SRAM with pins connected directly to TT tile pins."

How it works

This design contains a single port SRAM block with pins connected directly to TT tile pins. This allows to use this design directly as a SRAM block. It is the 5V version of the block but also compatible with 3.3V.

The included block has 128 words of 8 bits. The dimension is 224.72µm by 122.72µm. These are the pins for the block:

- a (7 bit): address
- we (1 bit): write enable signal indicating a read or write operation
- d (8 bit): input data
- q (8 bit): output data
- clk: clock for performing an operation

On each rising edge of the clock an operation is performed on the memory. A read is done when `we` is 0, while a write operation is done when it is 1. On the rising edge of the clock the `a` and `d` signals are latched into an internal buffer. For a read operation the data for the provided address is put into the `q` signal, the `d` signal is ignored. For a write operation the value of the `d` signal is put in the given address. The write operation is write-through meaning that also `q` will get the value of `d` during the operation.

How to test

You can test the block yourself by providing the right inputs for a read or write operation. One can check if data written to a certain location is later on read back with a read operation on the same address.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a[0]	q[0]	d[0]
1	a[1]	q[1]	d[1]
2	a[2]	q[2]	d[2]

#	Input	Output	Bidirectional
3	a[3]	q[3]	d[3]
4	a[4]	q[4]	d[4]
5	a[5]	q[5]	d[5]
6	a[6]	q[6]	d[6]
7	we	q[7]	d[7]

Car Trip

by **Luke Silva**

0067 25.175 MHz HDL Project

github.com/LukeSilva/tt-gf-car-trip

"Let's go on a car trip (vga demo)"

How it works

It works by calculating stuff...

How to test

I dunno, I haven't written it yet...

External hardware

The VGA PMod

Project Pinout

Digital Pins

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

triad01

by **Duan Yihe**

0069 | 50 MHz | HDL Project

github.com/Alanduan21/ttgf-triad01

> *"A clocked failsafe arbitration block for autonomous drone control paths. Evaluates RC and FC health signals through temporal filtering and ternary inference to select PRIMARY / FALLBACK / SAFE_HOLD, driving a 50 Hz PWM output accordingly. Includes a scan chain for counter observability and lightweight LBIST for post-silicon diagnosis. Four operating modes: normal, PWM-test (pad loopback), scan, and LBIST."*

How it works

A clocked failsafe arbitration block for autonomous drone control paths. Triad01 evaluates RC and FC health signals through temporal filtering and ternary inference to select PRIMARY / FALLBACK / SAFE_HOLD, driving a 50 Hz PWM output accordingly. Includes a scan chain for counter observability and lightweight LBIST for post-silicon diagnosis. Four operating modes: normal, PWM-test (pad loopback), scan, and LBIST.

How to test

temp placeholder

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	fc_live	decision[0]	safehold_active
1	rc_live	decision[1]	scan_out
2	fc_conf	decision_valid	bist_done
3	rc_conf	fc_score[0]	bist_pass
4	scan_in	fc_score[1]	raw_decision[0]
5	—	rc_score[0]	raw_decision[1]
6	mode[0]	rc_score[1]	—
7	mode[1]	PWM_out/loopback	—

4-Neuron LIF Spiking Neural Network

by **Ilim Stankulov**

0071 10 MHz HDL Project

github.com/ilim-stankulov/lif-spiking-neural-network

"A simple 4-neuron spiking neural network implementing the LIF model"

How it works

This project implements the Leaky Integrate-and-Fire model in a simple 4-neuron spiking neural network chip designed for TinyTapeout GF180nm

Every neuron has a membrane potential and an incoming current. Each clock cycle, the neuron receives input current from neurons that fired in the previous cycle and leaks a fraction of its current voltage (1/8 of the current voltage) according to the following equation:

V[t+1] = V[t] - V[t]/8 + I[t]

Once the voltage inside a neuron reaches 64, it fires a spike and resets its membrane potential to 0.

Every neuron's spike feeds back into other neurons through a weight matrix. If the connection weights are positive, they push the voltage towards the threshold, if they are negative, they pull the voltage away from the threshold.

How to test

1) Hold rst_n low for at least 3 clock cycles, then release. All membrane potentials and spike outputs clear to zero.

2) Assert ui[2] high (mode_prog = 1). Shift in 128 bits MSB-first by toggling ui[1] (s_clk_en) while driving ui[0] (s_data) with each bit. After 128 bits, uo[4] (prog_done) goes high confirming the weight matrix is fully loaded. De-assert ui[2].

3) Assert ena = 1. Spike outputs appear on uo[3:0], one bit per neuron. Behavior depends entirely on the programmed weights — zero weights produce silence, positive weights produce excitation, negative weights produce inhibition.

External hardware

No external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	s_data — serial weight bit input	spike_0 — neuron 0 spike output	—
1	s_clk_en — serial clock enable	spike_1 — neuron 1 spike output	—
2	mode_prog — 1=program weights, 0=run	spike_2 — neuron 2 spike output	—
3	—	spike_3 — neuron 3 spike output	—
4	—	prog_done — all 128 weight bits loaded	—
5	—	—	—
6	—	—	—
7	—	—	—

Asynchronous-AER Spike Router (4-phase REQ/ACK, 16-entry routing table, GF180)

by **Prof. Santhosh Sivasubramani** & **IIT Delhi**

0096 25 MHz HDL Project

github.com/SanthoshSivasubramani/tt_um_santhosh_aer_router_gf_pub

> *"GF180mcuD synchronous implementation of a 4-input / 4-output AER (address-event-representation) spike router. Time-multiplexed AER input and output buses each carry one event at a time as a 2-bit channel tag plus 6-bit address. Each incoming event is captured using a double-flop-synchronised 4-phase bundled-data REQ/ACK handshake, looked up in a 16-entry {dst_ch, dst_addr} routing table (or bypassed to pass through unchanged), queued in a 4-deep FIFO, and emitted out the output port with another 4-phase REQ/ACK handshake. Drops on FIFO overflow are counted in a 16-bit saturating drop counter with a sticky overflow_ever bit. Per-input 8-bit saturating event counters and IN_LAST/OUT_LAST debug latches are readable over SPI. 25 MHz signoff clock."*

A **4-input / 4-output AER (Address-Event-Representation) spike router** implemented entirely in GF180mcuD digital logic. The external AER handshakes are **4-phase bundled-data REQ/ACK**, double-flop-synchronised into a single clock domain for STA cleanliness.

Event flow

```
 (upstream sender)                                  (downstream
receiver)
   ui_in[7:6] = src_ch      +--------------+       uo_out[7:6] =
dst_ch
   ui_in[5:0] = src_addr    |              |       uo_out[5:0] =
dst_addr
   uio[4]     = in_req  --> |    ROUTER    | -->   uio[6]      =
out_req
   uio[5]     = in_ack  <-- |              | <--   uio[7]      =
out_ack
                            +------+------+
                                   |
                               SPI slave
                            (uio[0..3] = cs/mosi/miso/sck)
```

Internal pipeline per event:

```
 IS_IDLE  --(in_req rising, global_en=1)--> IS_LOOKUP
 IS_LOOKUP--(compute route_out)          --> IS_PUSH
 IS_PUSH  --(push FIFO, or drop++)       --> IS_ACK
 IS_ACK   --(drive in_ack=1, wait in_req=0)--> IS_IDLE
```

Output side drains the FIFO with a symmetric 4-phase handshake:

```
 OS_IDLE     --(fifo non-empty)--> OS_DRIVE     (load uo_out, raise
out_req)
 OS_DRIVE    --(out_ack=1)     --> OS_WAIT_REL  (drop out_req, pop
FIFO)
 OS_WAIT_REL --(out_ack=0)     --> OS_IDLE
```

Routing table

16-entry lookup table, each entry 8 bits: `{dst_ch[1:0], dst_addr[5:0]}`. The index is the **low 4 bits of the source address** (`src_addr[3:0]`). This is a deliberate design simplification: 4 source channels × 64 addresses is a 256-entry space, but only 16 routing slots exist on-chip. The low-nibble hash lets any source whose address ends in the same nibble share a routing slot, which is enough for most AER test workloads (neuron cluster → neuron cluster remapping) without blowing the area budget.

With `CTRL.bypass = 1` the table is skipped and the output payload is exactly the input payload (`{src_ch, src_addr}`).

Back-pressure — drops and drop counter

The output FIFO is 4 events deep. If a new event arrives while the FIFO is full, it is **dropped**:

- `drop_cnt` (16-bit saturating, regs `0x02`..`0x03`) increments.
- `STATUS.overflow_ever` (sticky bit) is latched.
- The per-input `evt_cnt` counter is still incremented (the event was captured; it just didn't fit).

Both counters are cleared by writing `1` to `CTRL.clear_drop` / `CTRL.clear_evt` (those bits are 1-cycle pulses, self-clearing).

Register map

Addr	Name	Description
`0x00`	`CTRL`	`{4'd0, clear_drop, clear_evt, bypass, global_en}`
`0x01`	`STATUS`	`{fifo_full, fifo_empty, overflow_ever, out_busy, in_busy, fifo_cnt[2:0]}`
`0x02`	`DROP_LO`	drop counter low byte
`0x03`	`DROP_HI`	drop counter high byte

0x04..0x07	EVT_CNT[0..3]	8-bit saturating per-input event counters
0x08	IN_LAST	{src_ch, src_addr} of most recent captured event
0x09	OUT_LAST	{dst_ch, dst_addr} of most recent emitted event
0x10..0x1F	ROUTE[0..15]	8-bit {dst_ch, dst_addr}; index = src_addr[3:0]

Pinout

- ui_in[5:0] — source address (6 bits)
- ui_in[7:6] — source channel tag (2 bits, selects which of 4 inputs)
- uo_out[5:0] — destination address (6 bits)
- uo_out[7:6] — destination channel tag (2 bits)
- uio[0] — spi_cs_n
- uio[1] — spi_mosi
- uio[2] — spi_miso
- uio[3] — spi_sck
- uio[4] — in_req
- uio[5] — in_ack
- uio[6] — out_req
- uio[7] — out_ack

How to test

```
# 1. Enable, route mode
spi_write(R_CTRL, 0x01)              # global_en=1, bypass=0
# 2. Program a route: src_addr[3:0]=5 -> {dst_ch=1, dst_addr=0x2A}
spi_write(R_ROUTE_BASE + 5, 0x6A)    # (1<<6)|0x2A
# 3. Host 4-phase send on input port:
#    drive {src_ch=3, src_addr=0x25} on ui_in, raise in_req;
#    when chip asserts in_ack on uio[5], drop in_req; when in_ack
drops, done
# 4. Consume on output port:
#    wait for chip to raise out_req on uio[6]; read uo_out = 0x6A;
#    raise out_ack on uio[7]; when chip drops out_req, drop out_ack
```

External hardware

Any AER-compliant source (silicon retina, spiking sensor emulator, FPGA event generator) and any AER-compliant sink. Upstream and downstream use a 4-phase bundled-data REQ/ACK protocol; the router tolerates arbitrary handshake timing on either side because both REQ and ACK are double-flop-synchronised at the input boundary.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	src_addr[0] (AER in, LSB)	dst_addr[0] (AER out, LSB)	spi_cs_n (in)
1	src_addr[1]	dst_addr[1]	spi_mosi (in)
2	src_addr[2]	dst_addr[2]	spi_miso (out)
3	src_addr[3]	dst_addr[3]	spi_sck (in)
4	src_addr[4]	dst_addr[4]	in_req (AER in, REQ from upstream sender)
5	src_addr[5] (MSB of 6-bit src addr)	dst_addr[5] (MSB of 6-bit dst addr)	in_ack (AER in, ACK to upstream sender)
6	src_ch[0] (AER in channel tag LSB)	dst_ch[0] (AER out channel tag LSB)	out_req (AER out, REQ to downstream receiver)
7	src_ch[1] (AER in channel tag MSB)	dst_ch[1] (AER out channel tag MSB)	out_ack (AER out, ACK from downstream receiver)

CIM Controller with BIST and Fault Map (GF180)

by **Prof. Santhosh Sivasubramani** & **IIT Delhi**

0098 | 25 MHz | HDL Project

github.com/santhoshs93/tt_um_santhosh_cim_bist_gf_pub

> *"GF180mcuD digital compute-in-memory peripheral: 8x8 1-bit cell array with built-in self-test (march/checkerboard/diagonal patterns), 64-bit sticky fault map, 16-bit dot-product accumulator over user-programmable row activations, SPI-programmable, irq output, 25 MHz signoff clock, 1x2 tile"*

Tile 6 of TTGF26a Scenario B (GF180mcuD, 1x1, 25 MHz).

An SPI-programmable compute-in-memory (CIM) controller with built-in self-test (BIST) and a persistent fault map.

Architecture

The block holds an 8x8 1-bit cell array (`cmem`), the digital stand-in for a memristive or SRAM CIM crossbar. Two orthogonal finite-state machines run against that array:

BIST FSM

On `start_bist` (CTRL[0] or rising edge on `ui_in[0]`):

1. **WRITE** — walks rows 0..7, writing the pattern-specific expected byte into each row (march phase 0, checkerboard, or diagonal).
2. **READ** — walks rows 0..7, XOR-ing the actual cell value against the expected byte. The mismatch mask is OR-merged into `fault_row[r]`. `ui_in[1]=1` forces a mismatch on `cell[0][0]` for silicon-side testing.
3. For the march pattern, the BIST re-runs WRITE then READ with the inverted phase (all-ones) so stuck-at-0 faults are also detected.
4. **DONE** — sets sticky `bist_done`, clears `bist_running`.

A 4-bit zero-extended popcount of each mismatch byte is accumulated into a saturating 8-bit `FAULT_COUNT`, and any non-zero mismatch latches `fault_ever`.

CIM FSM

On `start_cim` (CTRL[1]):

1. Walks rows 0..7. For each row where `INPUT_VEC[r]=1`, adds the 4-bit zero-extended popcount of `cmem[r]` into a 16-bit accumulator.

2. **DONE** — sets sticky `cim_done`, clears `cim_running`.

This models a 1-bit-weight column sum: activated rows contribute their total "on" cells into the accumulator.

The two FSMs are mutually exclusive: CIM waits while BIST is running and vice versa.

SPI protocol

Shared `spi_slave` (16-bit frame `{R/Wn, 7-bit addr, 8-bit data}`, mode 0, MSB first). The full register map is in the main README.

Sticky flags (`bist_done`, `cim_done`, `fault_ever`) are cleared by writing 1 to the corresponding bit of STATUS (W1C). `clear_fault_map` in CTRL is a self-clearing pulse that zeros the fault map, fault counter, and `fault_ever` without disturbing the cell contents — useful for running BIST periodically as a watchdog without losing the stored weights.

How to test

1. Via SPI, write the 8 rows of `cmem` (0x03 after setting ROW_SEL at 0x02) to load weights.
2. Write `INPUT_VEC` (0x04) with the activation vector.
3. Write CTRL (0x00) = 0x02 to start CIM. Poll `cim_done` (uo[2] or STATUS bit 6); read `ACCUM_LO/HI` (0x05/0x06).
4. For BIST: set PATTERN (0x01), write CTRL = 0x01, poll `bist_done`, read `FAULT_COUNT` (0x09) and fault rows (0x08 after setting 0x07).

External interfaces

Pin	Purpose
`ui_in[0]`	External BIST start (rising edge)
`ui_in[1]`	Inject fault (forces cell[0][0] mismatch)
`uo_out[0]`	Busy
`uo_out[1]`	`bist_done` sticky
`uo_out[2]`	`cim_done` sticky
`uo_out[3]`	`fault_ever` sticky
`uo_out[4]`	`cim_done && accum == 0`
`uio[6]`	`irq` (when `irq_en` and either sticky done is set)

Novelty

First TTGF tile to expose persistent cell-level defect data over a host interface — pairs with `tt_um_santhosh_xbar_ctrl_gf` (Tile 3) and

`tt_um_santhosh_rsd_char_gf` (Tile 8) for a full memristive-array diagnostic flow.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	start_ext (external BIST trigger; rising edge)	busy (bist_running | cim_running)	spi_cs_n
1	inject_fault (BIST test aid: forces cell[0][0] mismatch)	bist_done (sticky)	spi_mosi
2	reserved	cim_done (sticky)	spi_miso
3	reserved	fault_ever (sticky)	spi_sck
4	reserved	accum_zero (cim_done && accum==0)	bist_running
5	reserved	accum[8]	cim_running
6	reserved	accum[9]	irq
7	reserved	accum[10]	reserved

Neuromorphic PUF (distinct-tap LFSR arbiter + memristor XOR, GF180)

by **Prof. Santhosh Sivasubramani** & **IIT Delhi**

100 | 25 MHz | HDL Project

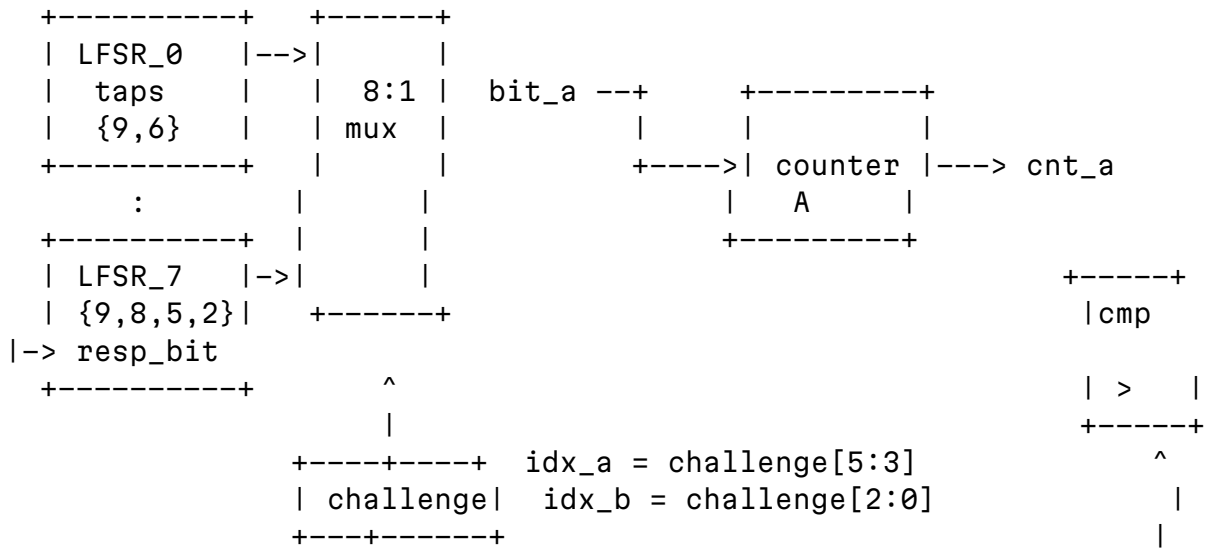
github.com/santhoshs93/tt_um_santhosh_neuro_puf_gf_pub

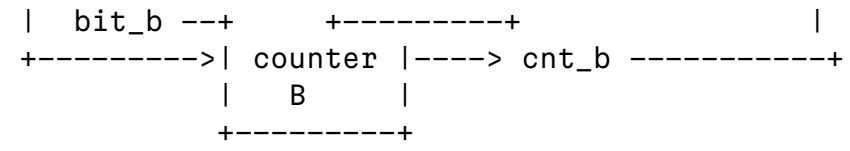
> *"GF180mcuD digital Physical Unclonable Function: 8 distinct-tap maximal-length 10-bit LFSRs act as synthesizable virtual ring-oscillators, a 6-bit challenge selects a pair, and an arbiter FSM compares their popcounts over a programmable 2^N-cycle window (N in 4..12) to produce a response bit. 64 challenges accumulate into a 64-bit response that can be XORed with a host-loaded memristor vector for dual-entropy mode. SPI-programmable, IRQ output, 25 MHz signoff clock, 1x2 tile (upsized from 1x1: global-placement utilization 103.6% on 1x1)."*

A compact **digital Physical Unclonable Function (PUF)** for TinyTapeout GF 26a that derives a 64-bit device-unique signature by running an arbiter between **eight distinct-tap maximal-length LFSRs** and optionally XORing the result with a host-loaded 64-bit "memristor vector" for dual-entropy mode.

Because free-running combinational ring oscillators are rejected by the standard-cell ASIC flow (Yosys logic-loop check), this design replaces them with eight clocked 10-bit LFSRs, each with a distinct maximal-length tap polynomial. Their output bits have data-dependent popcounts over a given window, giving the challenge-response asymmetry of an RO-PUF without the non-synthesizable feedback loop.

Architecture






```
  Every 2^N cycles (N in 4..12) one response bit is latched into
resp[63:0]
  at position `challenge`. Optional XOR with mem_vec[63:0] for
dual-entropy.
```

LFSR taps (all maximal-length, Xilinx XAPP052)

LFSR	Taps	Seed
0	{9, 6}	0x001
1	{9, 5}	0x002
2	{9, 4}	0x004
3	{9, 2}	0x008
4	{9, 7, 6, 5}	0x010
5	{9, 8, 7, 4}	0x020
6	{9, 8, 6, 3}	0x040
7	{9, 8, 5, 2}	0x080

SPI register map (16-bit frame, mode-0, MSB first)

Addr	Name	Description
0x00	R_CTRL	{irq_en, —, —, clear_resp, mem_xor_en, —, start, global_en}
0x01	R_CHAL	6-bit current challenge (idx_a = [5:3], idx_b = [2:0])
0x02	R_CYCLES	{—, —, —, run_sweep, cyc_exp[3:0]} — effective exp clamped 4..12
0x03	R_CHAL_MAX	6-bit max challenges in sweep (1..64)
0x04..0x0B	R_RESP0..7	64-bit response (LSB byte at 0x04)
0x0C..0x13	R_MVEC0..7	64-bit memristor vector (host write)
0x1E	R_STATUS	{busy, done_sticky, irq_sticky, ...}
0x1F	R_CHAL_IDX	current challenge index within the sweep
0x20	R_W1C	write 1<<6 to clear done, 1<<5 to clear irq

Degenerate challenges

Challenges where `idx_a == idx_b` (8 of the 64: `0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F`) would always produce `cnt_a == cnt_b` and thus a deterministic response bit of 0 — useless for PUF entropy. The arbiter substitutes the corresponding mem_vec bit instead, so the host can either skip these positions (treat them as the loaded fingerprint) or dual-entropy them by setting `mem_xor_en=1`.

Pinout

- `ui_in[7:0]` — reserved (tied off)
- `uo_out[7:0]` — live mirror of `resp[0]` (low byte of response)
- `uio[0..3]` — SPI (cs_n, mosi, miso, sck)
- `uio[4]` — busy
- `uio[5]` — done (sticky)
- `uio[6]` — irq (sticky)
- `uio[7]` — reserved

How to test

```
# generate response
spi_write(R_CYCLES, 0x14)          # run_sweep=1, cyc_exp=4 (16
cycles/chal)
spi_write(R_CHAL_MAX, 64)
spi_write(R_CTRL, 0x03)            # global_en=1, start=1
wait (status.done == 1)
read R_RESP0..R_RESP7              # 64-bit PUF response
spi_write(R_W1C, 0x40)             # clear done
```

Apply a mem_vec and retry with `R_CTRL = 0x0B` (global_en + start + mem_xor_en) to fold in the host-supplied entropy.

External hardware

None required. The PUF derives its entropy from the LFSR arithmetic; the `mem_vec` input lets an external memristor / OTP array or MCU contribute additional entropy through normal SPI writes.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	reserved (tied-off, available for future memristor-byte live override)	resp[0][0] (live mirror of R_RESP0 bit 0)	spi_cs_n

#	Input	Output	Bidirectional
1	reserved	resp[0][1]	spi_mosi
2	reserved	resp[0][2]	spi_miso
3	reserved	resp[0][3]	spi_sck
4	reserved	resp[0][4]	busy (out)
5	reserved	resp[0][5]	done (out, sticky)
6	reserved	resp[0][6]	irq (out, sticky)
7	reserved	resp[0][7]	reserved (driven low)

AER Reflex Chip - MCP2515 CAN gateway

by **khu-haeun**

`0101` `20 MHz` `HDL Project`

`github.com/khu-haeun/aer_reflex`

> "*Robot-arm reflex core: relays joint commands to CAN via an MCP2515, blocks the stream on a danger trigger and injects a reflex pose. Tuning values (recoil/speed/debounce) hardcoded; rules/threshold/flinch programmable.*"

How it works

This is the **decision/relay core of a robot-arm reflex system** (AgileX Piper 6-DOF arm). The chip is the **only transmitter on the CAN bus**: it forwards the host's normal joint commands to the robot via an external **MCP2515 (SPI↔CAN controller)**, and when a *danger* trigger fires it **blocks the normal stream and injects a reflex pose** instead. Because the chip is the sole sender, the "gate" is not a cross-module signal — it is a single internal mux.

Two SPI ports:

- `uio[0:3]` = **SPI slave** (the FPGA's PL writes config registers + relays normal CAN frames into the chip).
- `uio[4:6] + ui[3]` = **SPI master** that drives the MCP2515 directly (init, transmit, read-back).

Data path:

- *Normal:* PL → SPI-slave mailbox (regs `0x50–0x55`) → mux(normal) → MCP2515 → CAN.
- *Reflex:* trigger (DIP / soft / XADC-threshold) → `reflex_core_c` (rule table + priority + debounce) → `reflex_pose_gen` / `reflex_tx_src` → mux(reflex, gate closed) → MCP2515 → CAN.
- *RX:* MCP2515 RX → `mcp_rx_recv` (decodes feedback `0x2A5–7` current pose) → used by the "flinch-from-current-pose" reflex.

Reflex actions (`action_id`):

1. **freeze** — block normal commands, hold the current pose (the real arm's e-stop releases torque and droops, so freeze is used instead).
2. **duck/home** — drive to the home (0) pose, level-held while the trigger is active.
3. **flinch-home** — one-shot: jump toward home for `FLINCH_TICKS`, then auto-release (re-arm on sensor release).

4. **flinch-current** — one-shot: current pose + a J5 delta, then auto-release.

Programmable reflex (rule encoding, register `0x10–0x13`): each 16-bit rule is `[2:0]=action [3]=enable [5:4]=priority [6]=source(0=digital pin / 1=XADC threshold)`. The danger source is `xadc_val >= threshold` (FSR) or a digital pin. Highest-priority active rule wins. A debounce (`0x49`) rejects noise. **MCP2515 read-back** (regs `0x21–0x28`: CANSTAT/CNF/TEC/REC/...) is exposed so the host can see how the chip configured/drove the CAN controller (black-box-free debugging).

Note: timing constants (MCP oscillator-settle delay, SPI pacing, flinch ticks) are sized for the **20 MHz** clock via the module's default parameters. Run the design at 20 MHz.

How to test

The chip is a **clock-synchronous SPI peripheral** that also masters an MCP2515. To exercise it:

1. **Clock/reset:** drive `clk` at 20 MHz, pulse `rst_n` low then high, keep `ena`=1, `ui[7]`(arm_enable)=1.
2. **Wait for MCP init:** after reset the chip auto-runs the MCP2515 startup (soft-reset `0xC0`, oscillator-settle delay, `CNF1/2/3`=00/C0/80 for 1 Mbps, normal mode). `STATUS`(`0x20`) `init_done` goes high.
3. **Program (via SPI slave on `uio[0:2]`, 24-bit txn = 8-bit cmd `{rw,addr[6:0]}` + 16-bit data):** set a rule (e.g. `0x12`=FSR rule), threshold (`0x1A`), flinch ticks (`0x46/0x47`), speed (`0x48`), debounce (`0x49`).
4. **Relay a normal frame:** write `0x50`(id) + `0x51–0x54`(8 bytes) then `0x55` (send) → appears on CAN.
5. **Fire a reflex:** raise `ui[0]`(DIP) or drive the XADC input above threshold → `uo[5]`(reflex_active) goes high, the normal frame is blocked, and the reflex pose (`0x150/0x155–7`) goes out on CAN instead.
6. **Observe:** `uo[7]`=heartbeat, `uo[4:2]`=action_id, `uo[1]`=fire; MCP regs read-back via SPI `0x21–0x28`.

A full reference Verilog integration test (chip + an MCP2515 model, exercising pass-through + the current-pose flinch) is in `test/tb_reference_full.v` (+ `mcp2515_model_v2.v`). The `test/` cocotb harness runs a minimal reset/heart-beat sanity check for the GDS CI.

External hardware

- **MCP2515** SPI↔CAN controller (8 MHz crystal) on `uio[4:6]` (SCLK/MOSI/CSn) + `ui[3]` (MISO) + `ui[2]` (INT).
- **CAN transceiver** (e.g. TJA1050/SN65HVD230) from the MCP2515 to the robot's CAN bus (1 Mbps).

- **DIP switch** on `ui[0]` (estop/freeze trigger).
- **FSR (force sensor)** into the FPGA **XADC** analog input; the PL feeds the digitized value to the chip as the threshold-compare source (the reflex trigger).
- The chip is intended to sit between a Zynq PL (which does the SPI plumbing + XADC) and the robot's CAN bus.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	dip estop trigger	valid	SPI slave SCLK in
1	danger1 soft trigger	fire	SPI slave MOSI in
2	mcp_int	action_id0	SPI slave CSn in
3	mcp_miso	action_id1	SPI slave MISO out
4	—	action_id2	SPI master SCLK out
5	—	reflex_active gate	SPI master MOSI out
6	—	mcp_rst	SPI master CSn out
7	arm_enable	heartbeat	—

Wafer.space Logo VGA Screensaver

by **Uri Shaked**

0102 | 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 102.1: 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	—

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

Music for ASICs

by **d5smith**

`0103` `25 MHz` `HDL Project`

`github.com/d5smith/tt_um_d5smith_mfa`

"Autonomous generative ambient music chip using LFSR composition, scale quantization, and DDS synthesis"

How it works

Music for ASICs is an autonomous generative ambient music chip. It creates musical structure with pseudo-random LFSRs, maps the resulting note values onto musical scales, and renders the result with two direct digital synthesis voices.

The audio pipeline is:

```
LFSR composition engine -> scale quantizer -> DDS sound engine -> DAC/PWM output
```

Three maximal-length LFSRs, 5-bit, 7-bit, and 12-bit, produce deterministic pseudo-random sequences with co-prime periods. The short LFSRs shape note choice and melodic variation, while the 12-bit LFSR adds slower movement. A combinational quantizer maps raw note values onto pentatonic, minor 7th, blues, or fallback tuning tables.

The sound engine contains two DDS oscillators: a bass voice and a melody voice. Each voice supports square, triangle, and sawtooth waveforms. The voices are envelope-smoothed, mixed digitally, and exposed as both a 4-bit parallel DAC output and a single-pin PWM audio output.

How to test

1. Apply the Tiny Tapeout 25 MHz clock.
2. Hold `rst_n` low, then release it.
3. Music starts automatically after reset.
4. Connect an R-2R ladder to `uo_out[3:0]` for 4-bit DAC audio, or low-pass filter `uio_out[0]` for PWM audio.
5. Use `ui_in[1:0]` to select scale: `00` pentatonic, `01` minor 7th, `10` blues, `11` fallback.
6. Use `ui_in[3:2]` to select tempo: `00` slow, `01` medium, `10` fast, `11` very fast.
7. Use `ui_in[5:4]` to select the bass waveform: `00`/`11` square, `01` triangle, `10` sawtooth.
8. Use `ui_in[7:6]` to select the melody waveform: `00`/`11` square, `01` triangle, `10` sawtooth.

9. Optional: connect LEDs to `uo_out[7:4]` to view debug activity.

External hardware

- 4-bit R-2R DAC ladder on `uo_out[3:0]`, for example using 10k and 20k resistors.
- Or an RC low-pass filter on `uio_out[0]` before a high-impedance audio input or amplifier.
- Optional LEDs on `uo_out[7:4]` for sample tick, voice-active, and LFSR debug signals.

Do not connect low-impedance headphones directly to a GPIO pin. Use an amplifier or a suitable high-impedance input stage.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	scale_sel[0]	DAC bit 0 (LSB)	PWM audio out
1	scale_sel[1]	DAC bit 1	—
2	tempo_sel[0]	DAC bit 2	—
3	tempo_sel[1]	DAC bit 3 (MSB)	—
4	wave_bass[0]	Debug: sample tick	—
5	wave_bass[1]	Debug: bass active	—
6	wave_melody[0]	Debug: melody active	—
7	wave_melody[1]	Debug: LFSR-12 MSB	—

UART_SOC

by **Siri Jagadeesh**

0128 HDL Project

`github.com/sirijagadeesh4-hub/SIRI_UART_SOC`

"Verifies firmware signature using a secure hash comparison module"

Description

SIRI UART SOC is a simple UART receiver System-on-Chip designed in Verilog for Tiny Tapeout.

The design receives serial UART data through the input pin and converts it into parallel data output. After successful reception of data bits, a done signal is generated.

Features

- UART serial receiver
- 7-bit serial data reception
- Parallel data output
- Done signal generation
- Tiny Tapeout compatible
- Simple RTL architecture

How it Works

The UART receiver continuously samples serial data from `ui_in[0]`.

Each clock cycle:

- One serial bit is shifted into an internal shift register.
- A bit counter tracks the number of received bits.
- After receiving 7 bits, the data is transferred to the output register.
- A done signal is generated on `uio_out[0]`.

The received data appears on `uo_out[6:0]`.

Inputs

Pin	Description
`ui_in[0]`	UART RX Serial Input

`clk`	System Clock
`rst_n`	Active Low Reset

Outputs

Pin	Description
`uo_out[6:0]`	Received UART Data
`uo_out[7]`	Unused
`uio_out[0]`	Data Ready / Done Signal

Reset Behavior

When `rst_n = 0`:

- Shift register is cleared
- Bit counter resets to zero
- Output data clears
- Done signal resets

How to Test

1. Apply reset:
   - Set `rst_n = 0`
   - Wait one clock cycle
   - Set `rst_n = 1`
2. Send serial data through `ui_in[0]`
   - One bit per clock cycle
3. After 7 received bits:
   - Output data appears on `uo_out[6:0]`
   - `uio_out[0]` goes HIGH for one clock cycle

Testbench

The project includes a Verilog testbench located in the `test` directory.

Simulation verifies:

- UART bit shifting
- Bit counting
- Parallel output generation
- Done signal assertion

Tiny Tapeout

This project is designed for the Tiny Tapeout open-source ASIC flow.

More information:

- https://tinytapeout.com

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in_0	verified	—
1	data_in_1	status_led	—
2	data_in_2	—	—
3	data_in_3	—	—
4	signature_bit_0	—	—
5	signature_bit_1	—	—
6	signature_bit_2	—	—
7	signature_bit_3	—	—

CWRU CHIPS Simple Counter with 7-segment Display

by **Andrew Chen** & **John Paul Magbitang**

`0129` `5 Hz` `HDL Project`

`github.com/john-paul-sm/ASICWRU_SimpleCounter`

> "*Simple counter demo on a RISC-V CPU with 7-segment display output*"

Credit: Patterson & Hennessy, Computer Organization and Design: RISC-V Edition

How It Works

This is an educational technical project aimed at teaching CWRU computer engineering students how to design a single-cycle RISC-V processor in Verilog. This `info.md` is meant to serve as pedagogical material for those also interested in learning computer architecture concepts. The deliverable will be a complete processor that executes a Fibonacci sequence using RISC-V assembly.

Pipeline (Big Picture)

The CPU has five stages that it must go through to complete a single instruction:

1. `IF` / `Instruction Fetch`: Fetches the current cycle's instruction from memory
2. `ID`/ `Instruction Decode`: Converts instructions into register addresses, opcodes, etc.
3. `EX` / `Execute`: Perform operations with those opcodes/register values
4. `MEM` / `Memory`: Writes/loads data to memory, more important for load/store instructions
5. `WB` / `Writeback`: Writes result into the register file for next cycle, repeat

In a **single-cycle** design, all five stages happen within one clock period. Every component is connected combinationally from the output of instruction memory through to the writeback mux, with the only registered state being the program counter (updated on each clock edge) and the register file / data memory (written on the clock edge when their respective write enables are asserted).

RISC-V Opcode Reference

These are the standard base integer (RV32I) opcodes. The `opcode` field occupies bits [6:0] of every instruction.

Instruction Type	opcode (binary)	opcode (hex)	Example Instructions
R-type	`0110011`	`0x33`	`add`, `sub`, `and`, `or`, `slt`
I-type (ALU)	`0010011`	`0x13`	`addi`, `andi`, `ori`, `slti`
I-type (Load)	`0000011`	`0x03`	`lw`, `lh`, `lb`
I-type (JALR)	`1100111`	`0x67`	`jalr`
S-type	`0100011`	`0x23`	`sw`, `sh`, `sb`
B-type	`1100011`	`0x63`	`beq`, `bne`, `blt`, `bge`
U-type (LUI)	`0110111`	`0x37`	`lui`
U-type (AUIPC)	`0010111`	`0x17`	`auipc`
J-type	`1101111`	`0x6F`	`jal`

funct3 Reference

`funct3` (bits [14:12]) disambiguates instructions that share the same opcode.

funct3	R-type (`0x33`)	I-type ALU (`0x13`)	Load (`0x03`)	Store (`0x23`)	Branch (`0x63`)
`000`	`add` / `sub`*	`addi`	`lb`	`sb`	`beq`
`001`	`sll`	`slli`	`lh`	`sh`	`bne`
`010`	`slt`	`slti`	`lw`	—	—
`011`	`sltu`	`sltiu`	—	—	—
`100`	`xor`	`xori`	`lbu`	—	`blt`
`101`	`srl` / `sra`*	`srli` / `srai`*	`lhu`	—	`bge`
`110`	`or`	`ori`	—	`sw`	`bltu`
`111`	`and`	`andi`	—	—	`bgeu`

* `funct7[5]` *distinguishes these pairs:* `0` → `add` / `srl` / `srli`, `1` → `sub` / `sra` / `srai`

Components

Program Counter

The program counter acts as a tracker for the CPU. Namely, what instruction are we executing right now? It is a register that holds the address of the instruction that's currently being executed. This number can be updated based on normal sequencing (+4 bytes per cycle for 4-byte instructions) or

conditional branches and jumps that can happen in the code. The pins you'll need are:

- `clk`: program counter needs to update on the clock, keeping it synchronized with the rest of the CPU
- `pc_in`: this is a little tricky but this is the next address to execute, it may come from a branch
- `pc_out`: this is the current address being executed, and `pc_in` will propagate on the next edge

Essentially, the program counter is just a large 32-bit flip-flop if that helps the Verilog code. As for the lack of reset pins for this program counter, it's cleaner to have a separate register in the top module that's sensitive to the reset and propagate it into the program counter, as opposed to having reset here and needing to reset the `pc` signal in the top module anyway.

Instruction Memory

Instruction memory is a piece of RAM that stores your instructions for the processor to execute. It may be more intuitive to think of an instruction memory as an array:

```
// this array would be your instruction memory
unsigned char instructions[256] ={0x00000013, 0x00000013,
0x00000013, ...};
// to access the array, you need to do some indexing
// normal cpu operation

// we have a register named curr_instr
unsigned char curr_instr;

// we use a for loop here, the counter variable
// i is just like our program counter
for (int i = 0; i < 256; i++) {
    curr_instr = instructions[i];
}
```

For those who are technically sharp, this isn't 100% accurate because of possible branch/jump instructions, but for 99% of cases when the CPU is running sequentially, this is a perfect parallel. The pins you'll need are:

- `pc`: the output of your program counter (incrementer)
- `instr`: whatever instruction needs to be executed

Hint: to instantiate an array block in Verilog, you can do this:

```
wire [WIDTH-1:0] mem [0:DEPTH-1];
```

- Just remember if you're updating it in a process, use `reg` instead
- `mem` is just the assigned name, you can name it anything
- `WIDTH` is how wide each word is, maybe let's say 32 bits wide

- `DEPTH` is how many words can be stored at a given time in `mem`

You can pre-load instruction memory using `$readmemh` in a Verilog initial block, which reads a hex file of encoded instructions directly into your memory array:

```
initial begin
    $readmemh("program.hex", mem);
end
```

Register File

The `register file` is a module that, in RISC-V specification, contains 32 distinct registers. Each register itself can store a 32-bit word, which can then be used as operands within the ALU. Some design questions:

- Does the register file need a `clk` signal? What are the advantages/trade-offs of having a clocked register file?
- We will need read and write ports for the register file. For 32 registers, how many bits do we need to identify a unique register? We will need three addresses: `rd_addr1`, `rd_addr2`, and `wr_addr`.

> **Important**: In RISC-V, register `x0` is hardwired to zero — writes to it must be ignored and reads from it must always return `0`. Make sure your register file enforces this.

Immediate Generator

The immediate generator, like its name suggests, generates an immediate value. But what is an immediate value? Take a look at the following assembly instruction:

```
add x3, x1, x2
```

In RISC-V, there are six types of instructions: R-type, I-type, S-type, B-type, U-type, and J-type instructions. This `add` instruction is an R-type instruction, as it adds the contents of two registers (namely `x1` and `x2`) and saves the result in `x3`. There is no immediate here. Now let's take a look at an I-type instruction:

```
addi x3, x1, 1
```

With a blank register file that will initiate all register values to 0 on power up, you would be pretty limited in what you can do with just zeroes. This "add immediate" instruction allows you to add any integer with x1, and save the result to x3.

Source: RISC-V International, The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2.

Refer to this diagram to see how the immediates are encoded in each instruction. You can determine based on the opcodes what type of instruction it is. As for designing/writing the RTL code, treat the immediate generator

as a switch case that based on the opcode, decodes the immediate value from the instruction. For simplicity, make the immediate a 32-bit logic value to keep things consistent.

All immediates must be **sign-extended** to 32 bits. This means replicating the most significant bit of the immediate field across all upper bits when producing the 32-bit output. In Verilog:

```
// Example: sign-extending a 12-bit I-type immediate
assign imm_out = {{20{instr[31]}}, instr[31:20]};
```

Control Unit

The control unit asserts flags that help facilitate the `execute` stage of the CPU. Generally, it needs to take in the opcode, funct3, and funct7 as inputs, and it outputs the following values:

- `RegWrite`: are we writing the result to reg?
- `MemRead`: are we reading from memory?
- `MemWrite`: are we writing to memory?
- `BranchEq`: is the `beq` condition satisfied?
- `MemToReg`: are we loading from memory to reg?
- `ALUSrc`: which operand are we using? (`0` = rs2, `1` = immediate)
- `ALUCont`: what type of ALU operation are we doing? (passed to ALU control)
- `JMP`: did a jump occur?

This can be implemented using a really large switch case. Recall that we have 6 types of instructions (R, I, S, B, U, J) all with different opcodes and funct codes.

The table below summarizes what each control signal should be asserted to for each major instruction type:

Signal	R-type	I-type (ALU)	I-type (Load)	S-type	B-type	J-type
`RegWrite`	1	1	1	0	0	1
`ALUSrc`	0	1	1	1	0	—
`MemRead`	0	0	1	0	0	0
`MemWrite`	0	0	0	1	0	0
`MemToReg`	0	0	1	—	—	0
`BranchEq`	0	0	0	0	1	0
`JMP`	0	0	0	0	0	1

R-type

R-type (R for register) instructions are one of the common types of instructions that you will encounter in the assembly language. R-type operations

always take two values from the register file, perform an operation (addition, subtraction, multiplication, division), then writes directly back to the register file.

I-type

I-type instructions are also extremely common but slightly different from R-type instructions, where the I-part stands for “immediate”. The I-type instruction typically takes only one register and a user-defined constant (0-4095) as operands, making it useful for easy increments without having to explicitly reference a second register operand. Load instructions (`lw`, `lh`, `lb`) are also encoded as I-type: the immediate serves as a byte offset added to `rs1` by the ALU to compute the memory address.

S-type

S-type (store) instructions write a register value into data memory. The destination address is computed as `rs1 + imm`, where the 12-bit immediate is split across two non-contiguous fields in the instruction encoding — bits [11:5] in `instr[31:25]` and bits [4:0] in `instr[11:7]`. Your immediate generator must reassemble these pieces. Common S-type instructions: `sw` (store word), `sh` (store halfword), `sb` (store byte).

B-type

B-type (branch) instructions conditionally redirect the program counter based on a comparison between two registers. The branch target is PC-relative: `PC + imm`, where `imm` is a signed 13-bit offset (always even, since instructions are 4-byte aligned). The immediate is also scrambled across the instruction word — refer to the encoding diagram above carefully. The ALU computes `rs1 - rs2` and the `zero_flag` tells the control unit whether the branch condition is satisfied. Common B-type instructions: `beq`, `bne`, `blt`, `bge`.

U-type

U-type (upper immediate) instructions load a 20-bit immediate into the upper 20 bits of a destination register (bits [31:12]), zeroing the lower 12 bits. This is useful for constructing large constants in combination with an I-type instruction. For example, `lui x1, 0x12345` followed by `addi x1, x1, 0x678` builds the full 32-bit constant `0x12345678` in `x1`. The two U-type instructions are `lui` (Load Upper Immediate) and `auipc` (Add Upper Immediate to PC, useful for position-independent addressing).

J-type

J-type (jump) instructions unconditionally redirect the program counter. Like B-type, the target is PC-relative, but the immediate is 21 bits, allowing larger jumps. The return address (`PC + 4`) is saved into `rd`, enabling function calls and returns. The immediate bits are scrambled across the instruction word — be careful to reassemble them correctly in your immediate generator. The

primary J-type instruction is `jal` (Jump and Link). Its register-indirect cousin, `jalr` (Jump and Link Register), is actually encoded as an I-type with opcode `0x67`.

Arithmetic Logic Unit

The arithmetic logic unit (also known as the ALU) is the calculator that performs mathematical operations on your operands. The ALU accepts the following pins:

- `op1`: your first operand (always `rs1` from the register file)
- `op2`: second operand — either `rs2` (R-type) or the sign-extended immediate (I/S/B-type), selected by the `ALUSrc` control signal upstream
- `alu_ctrl`: the specific operation to perform (add, sub, AND, OR, SLT, etc.)
- `res`: the 32-bit result of the operation
- `zero_flag`: asserted when `res == 0`; used by the control unit to evaluate branch conditions

The `alu_ctrl` lines are typically driven by a small sub-module called the **ALU Control Unit**, which takes `ALUCont` (from the main control unit) plus `funct3` and `funct7` to determine the exact operation. For example, both `add` and `sub` share the R-type opcode `0x33` — it is `funct7[5]` that distinguishes them.

alu_ctrl	Operation	Used by
0000	AND	and, andi
0001	OR	or, ori
0010	ADD	add, addi, lw, sw
0110	SUB	sub, beq (zero check)
0111	Set Less Than (SLT)	slt, slti
1100	NOR	(less common)
1101	XOR	xor, xori
1110	Shift Left Logical	sll, slli
1111	Shift Right Logical	srl, srli

> Note: the `alu_ctrl` encoding above is one common convention — what matters is internal consistency between your ALU and ALU control unit.

A minimal Verilog skeleton for the ALU:

```
module alu (
    input  [31:0] op1,
    input  [31:0] op2,
    input  [3:0]  alu_ctrl,
    output reg [31:0] res,
    output zero_flag
```

```
);
    always @(*) begin
        case (alu_ctrl)
            4'b0000: res = op1 & op2;           // AND
            4'b0001: res = op1 | op2;           // OR
            4'b0010: res = op1 + op2;           // ADD
            4'b0110: res = op1 - op2;           // SUB
            4'b0111: res = (op1 < op2) ? 1 : 0; // SLT
            4'b1101: res = op1 ^ op2;           // XOR
            4'b1110: res = op1 << op2[4:0];     // SLL
            4'b1111: res = op1 >> op2[4:0];     // SRL
            default: res = 32'b0;
        endcase
    end
    assign zero_flag = (res == 32'b0);
endmodule
```

Data Memory

The data memory acts as a secondary RAM that stores runtime data (not instructions). Specifically, `STORE` instructions write data from the register file into data memory, and `LOAD` instructions read data back from memory into the register file.

The pins you'll need are:

- `clk`: data memory writes are synchronous — they happen on the clock edge
- `addr`: the memory address to read from or write to (comes from the ALU result)
- `wr_data`: the data to be written (comes from `rs2` in the register file)
- `MemRead`: control signal asserted for load instructions (`lw`)
- `MemWrite`: control signal asserted for store instructions (`sw`)
- `rd_data`: the data read out of memory (routed to the writeback mux)

A few design notes:

- **Address alignment**: RISC-V word addresses are 4-byte aligned. The ALU computes the full byte address, so index into your word-addressed memory array using `addr[N:2]` (dropping the two LSBs).
- **Read vs. Write**: Writes must be clocked (registered) so data is only committed on the clock edge. Reads can be combinational in a single-cycle design so the data is available immediately within the same cycle.
- **MemToReg mux**: The `MemToReg` control signal selects whether the write-back value into the register file comes from the ALU result (`MemToReg = 0`) or from `rd_data` (`MemToReg = 1`). This 2:1 mux lives between data memory and the register file write port, typically in your top-level datapath.

```
module data_memory (
    input         clk,
```

```
    input  [31:0] addr,
    input  [31:0] wr_data,
    input         MemRead,
    input         MemWrite,
    output [31:0] rd_data
);
    reg [31:0] mem [0:255];  // 256 words of data memory

    always @(posedge clk) begin
        if (MemWrite)
            mem[addr[9:2]] <= wr_data;  // word-aligned write
    end

    assign rd_data = MemRead ? mem[addr[9:2]] : 32'b0;
endmodule
```

Top-Level Datapath

With all components implemented, the top-level module wires them together. The signal flow for a typical R-type instruction looks like this:

```
PC → Instruction Memory → [opcode/rs1/rs2/rd/funct3/funct7]
                                ↓                ↓
                          Control Unit    Register File (read rs1,
rs2)
                                ↓                ↓
                           ALUSrc mux ← rs2 or Immediate Generator
                                ↓
                              ALU → res → MemToReg mux → Register
File (write rd)
                                                ↑
                                          Data Memory (if Load)
```

For branch instructions, the ALU `zero_flag` is AND-ed with `BranchEq` to decide whether `pc_in` comes from `PC + 4` (sequential) or `PC + imm` (branch taken). For jump instructions, `pc_in` is driven directly to `PC + imm` when `JMP` is asserted.

Target Program: Fibonacci Sequence

The goal of this project is to execute the following Fibonacci sequence program in RISC-V assembly. Use this as your integration test — if your CPU correctly drives the seven-segment display through all Fibonacci values, all five pipeline stages and all major instruction types are working.

```
# Fibonacci sequence in RISC-V assembly
# x1 = F(n-2), x2 = F(n-1), x3 = F(n), x10 = loop counter
```

```
    addi x1, x0, 0        # F(0) = 0
    addi x2, x0, 1        # F(1) = 1
    addi x10, x0, 10      # compute 10 terms

loop:
    add  x3, x1, x2       # F(n) = F(n-1) + F(n-2)
    add  x1, x0, x2       # x1 = old x2
    add  x2, x0, x3       # x2 = new F(n)
    sw   x3, 0(x0)        # store result to data memory (addr 0)
    addi x10, x10, -1     # decrement counter
    bne  x10, x0, loop    # if counter != 0, loop
```

This program exercises `addi` (I-type), `add` (R-type), `sw` (S-type), and `bne` (B-type) — a solid cross-section of the instruction set that validates your control unit, ALU, register file, data memory, and branch logic all at once.

How to Test

Simulation (Icarus Verilog or Vivado XSim)

1. Pre-load instruction memory using `$readmemh("fibonacci.hex", mem)` in an `initial` block inside your instruction memory module.
2. Write a testbench that instantiates your top-level CPU, drives `clk`, and asserts reset for the first few cycles.
3. Use `$monitor` or `$dumpvars` to observe register file contents and data memory writes over time.
4. Verify that after 10 loop iterations, the sequence `0, 1, 1, 2, 3, 5, 8, 13, 21, 34` appears at memory address 0 across successive cycles.

On Hardware (Seven-Segment Display)

The CPU will drive a seven-segment display showing each Fibonacci value in sequence. Confirm that the display cycles through the correct values without freezing or producing garbage output. Common failure modes to watch for:

- Display stuck on one value → branch logic not redirecting the PC correctly
- Display shows wrong values → ALU or immediate generator bug
- Display cycles too fast / too slow → clock domain issue at the top level

---

External Hardware

There's no external hardware needed to run this ASIC! It will run a Fibonacci sequence on the seven-segment display using the assembly instructions above.

References

- Patterson & Hennessy, *Computer Organization and Design: RISC-V Edition*, Elsevier
- RISC-V International, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	SEG_A	ADE
1	—	SEG_B	—
2	—	SEG_C	—
3	—	SEG_D	—
4	—	SEG_E	—
5	—	SEG_F	—
6	—	SEG_G	—
7	—	—	—

Raksha

by **Sinchana**

`0130` `1 MHz` `HDL Project`

`github.com/shanthkumars165-commits/secure_packet_authentication`

"Secure Packet Authentication Engine for V2X Communication"

How it works

Raksha is a lightweight Secure Packet Authentication Engine designed for secure V2X communication systems.

The design checks whether an incoming packet is authenticated using a predefined secret key.

The hardware performs an XOR operation between:

- Incoming packet data (`ui_in`)
- Secret key (`0xA5`)

Authentication condition:

5A XOR A5 = FF

If the result matches the expected value:

- Authentication OK signal becomes HIGH.

Otherwise:

- Alert signal becomes HIGH.

Outputs:

- `uo_out[0]` → Authentication Success
- `uo_out[1]` → Alert Detection

The design demonstrates basic hardware-assisted packet authentication suitable for low-latency vehicular communication systems.

How to test

1. Apply clock and reset signals.
2. Provide packet data through `ui_in[7:0]`.
3. Observe output signals.

Valid Packet Test

Input:

- `ui_in = 0x5A`

Expected Output:

- `uo_out[0] = 1`
- `uo_out[1] = 0`

Invalid Packet Test
Input:

- `ui_in = 0x12`

Expected Output:

- `uo_out[0] = 0`
- `uo_out[1] = 1`

The simulation testbench automatically verifies both valid and invalid authentication cases.

External hardware

The project can interface with:

- FPGA development boards
- Vehicle communication modules
- Microcontrollers
- Embedded V2X systems
- LEDs for authentication status indication

Possible hardware usage:

- Green LED → Authentication Successful
- Red LED → Unauthorized Packet Alert

The design is compatible with Tiny Tapeout ASIC flow and can be integrated into lightweight secure communication hardware systems.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Packet Bit 0	Authentication OK	—
1	Packet Bit 1	Alert	—
2	Packet Bit 2	—	—
3	Packet Bit 3	—	—
4	Packet Bit 4	—	—
5	Packet Bit 5	—	—
6	Packet Bit 6	—	—

#	Input	Output	Bidirectional
7	Packet Bit 7	—	—

8-bit WNN Pattern Recognizer

by **Your Name**

`0132` `HDL Project`

`github.com/cambridgeinstitutetechnology-blr-venkat/cit_project-2`

"8-bit Weightless Neural Network for pattern recognition using pattern matching."

This project implements the required digital logic circuit using Verilog. The circuit processes the input signals and generates outputs according to the designed functionality.

How to test

1. Apply input values using the Tiny Tapeout input pins.
2. Observe the output pins.
3. Verify that the outputs match the expected behaviour of the circuit.

External hardware

No external hardware required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input bit 0	Pattern match result	unused
1	Input bit 1	Bit 1 compare	unused
2	Input bit 2	Bit 2 compare	unused
3	Input bit 3	Bit 3 compare	unused
4	Input bit 4	Bit 4 compare	unused
5	Input bit 5	Bit 5 compare	unused
6	Input bit 6	Bit 6 compare	unused
7	Input bit 7	Bit 7 compare	unused

Secure TRNG Entropy Generator

by **Likitha**

0134 HDL Project

github.com/Cambridgeinstituteoftechnology-likitha/likii

"Hardware entropy mixing circuit generating randomized outputs using XOR feedback paths"

How it works

The Secure TRNG (True Random Number Generator) Entropy Generator is a lightweight hardware design that demonstrates entropy mixing and randomized output generation using XOR-based feedback logic.

The design accepts an 8-bit entropy input through the Tiny Tapeout user input pins. Internal feedback paths are generated from selected entropy bits and combined with the incoming entropy data to produce an 8-bit randomized output stream.

Architecture

```
Entropy Input (8-bit)
        |
        v
Feedback Path Generation
        |
        v
XOR Mixing Logic
        |
        v
Randomized Output (8-bit)
```

Working Principle

1. Entropy Collection
   - Raw entropy bits are applied through the 8-bit input interface.
2. Feedback Generation
   - Feedback paths are generated using XOR operations between selected input bits.
3. Entropy Mixing
   - The generated feedback paths are combined with the input entropy bits.
4. Random Output Generation
   - The processed values are driven to the output pins as an 8-bit randomized output.

The design provides a compact hardware entropy processing architecture suitable for studying randomness generation techniques in FPGA and ASIC implementations.

How to test

Apply any 8-bit value to the input pins:

Input	Description
ui[7:0]	Entropy Input

Observe the generated output:

Output	Description
uo[7:0]	Randomized Output

Example

Input:

`10101100`

Output:

`11001011`

Different input patterns will produce different output patterns based on the implemented XOR feedback network.

Pin Mapping

Pin	Function
ui[0]	Entropy Input Bit 0
ui[1]	Entropy Input Bit 1
ui[2]	Entropy Input Bit 2
ui[3]	Entropy Input Bit 3
ui[4]	Entropy Input Bit 4
ui[5]	Entropy Input Bit 5
ui[6]	Entropy Input Bit 6
ui[7]	Entropy Input Bit 7
uo[0]	Random Output Bit 0
uo[1]	Random Output Bit 1
uo[2]	Random Output Bit 2

uo[3]	Random Output Bit 3
uo[4]	Random Output Bit 4
uo[5]	Random Output Bit 5
uo[6]	Random Output Bit 6
uo[7]	Random Output Bit 7

External hardware

No external hardware is required.

The entropy inputs are supplied through the Tiny Tapeout input pins and the randomized outputs can be observed through the Tiny Tapeout output pins.

Applications

- Hardware Security
- Random Number Generation
- Cryptographic Systems
- Entropy Processing
- FPGA Prototyping
- ASIC Design Demonstration

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	entropy_in_0	random_out_0	—
1	entropy_in_1	random_out_1	—
2	entropy_in_2	random_out_2	—
3	entropy_in_3	random_out_3	—
4	entropy_in_4	random_out_4	—
5	entropy_in_5	random_out_5	—
6	entropy_in_6	random_out_6	—
7	entropy_in_7	random_out_7	—

2048 sliding tile puzzle game (VGA)

by **Uri Shaked**

0135 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

ECDSA Verification

by **Lyra**

0192 | 10 MHz | HDL Project

`github.com/santoshism2006-pixel/vtox`

> "*Simplified ECDSA digital signature verification module using Verilog*"

Firmware Signature Verification Accelerator

How it works

The Firmware Signature Verification Accelerator is a hardware security IP designed for secure V2X (Vehicle-to-Everything) communication systems. This project focuses on verifying the authenticity and integrity of firmware before execution inside automotive electronic systems.

The accelerator uses a cryptographic signature verification process to validate firmware updates.
It prevents unauthorized or tampered firmware from running in the system, improving security against cyber attacks.

Main Functional Blocks

1. **Hash Engine**
   - Generates a secure hash value from incoming firmware data using SHA algorithms.
   - Ensures firmware integrity.
2. **Signature Verification Core**
   - Verifies the digital signature using ECC/ECDSA cryptographic methods.
   - Confirms firmware authenticity.
3. **Key Storage**
   - Stores trusted public keys used for verification.
4. **Control Unit**
   - Manages verification sequence and operation control.
5. **Status Output**
   - Displays whether firmware is valid or invalid.

Tile Layout Information

The Tiny Tapeout layout contains multiple standard-cell tiles used for implementing:

- Hash computation logic
- ECC arithmetic blocks
- Control FSM

- Memory interface
- Signature verification engine

The total number of tiles depends on synthesis and place-and-route results generated during the OpenLane RTL-to-GDSII flow.

Inputs

Signal	Description
ui_in[7:0]	Firmware data / control input
ena	Enable signal
clk	System clock
rst_n	Active-low reset

Outputs

Signal	Description
uo_out[7:0]	Verification status output
uio_out	Debug/status signals

How to test

1. Apply reset signal (`rst_n = 0`) for initialization.
2. Enable the module using `ena = 1`.
3. Provide firmware data through `ui_in`.
4. Start signature verification process.
5. Observe output signals:
    - Valid signature → success indication on `uo_out`
    - Invalid signature → failure indication on `uo_out`

Simulation Steps

```bash make clean make make sim
```

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input bit 0	Output bit 0	Bidirectional IO 0
1	Input bit 1	Output bit 1	Bidirectional IO 1
2	Input bit 2	Output bit 2	Bidirectional IO 2
3	Input bit 3	Output bit 3	Bidirectional IO 3
4	Input bit 4	Output bit 4	Bidirectional IO 4

#	Input	Output	Bidirectional
5	Input bit 5	Output bit 5	Bidirectional IO 5
6	Input bit 6	Output bit 6	Bidirectional IO 6
7	Input bit 7	Output bit 7	Bidirectional IO 7

ECC Processor

by **Dhanush Kulkarni**

`0194` `50 MHz` `HDL Project`

`github.com/CambridgeinstitutetechnologyBLR-Dhanush/_ECC_Processor`

"Compact 8-bit ECC Processor demonstrating scalar-point arithmetic for TinyTapeout."

How it works

The ECC Processor is a compact hardware accelerator implemented in Verilog for TinyTapeout.

The design accepts two 8-bit input values:

- `ui_in[7:0]` : ECC scalar value (k)
- `uio_in[7:0]` : ECC point coordinate

On every rising edge of the clock, the processor performs a simplified ECC arithmetic operation:

```
Result = (Scalar × Point) + Scalar
```

The computed result is stored in an internal register and presented on the output bus `uo_out[7:0]`.

The design demonstrates the basic concept of scalar multiplication used in Elliptic Curve Cryptography (ECC) while remaining small enough to fit within a TinyTapeout 1x1 tile.

How to test

1. Apply reset (`rst_n = 0`) for one clock cycle.
2. Release reset (`rst_n = 1`).
3. Apply an 8-bit scalar value on `ui_in`.
4. Apply an 8-bit point value on `uio_in`.
5. Provide clock pulses on `clk`.
6. Observe the computed result on `uo_out`.

Example:

Scalar (ui_in)	Point (uio_in)	Output (uo_out)
5	3	20
10	4	50
7	2	21

Calculation:

```
Output = (Scalar × Point) + Scalar
```

External hardware

No external hardware is required.

The design uses only the standard TinyTapeout I/O interface and can be tested directly through simulation or on TinyTapeout demonstration boards.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Scalar bit 0	Result bit 0	Point bit 0
1	Scalar bit 1	Result bit 1	Point bit 1
2	Scalar bit 2	Result bit 2	Point bit 2
3	Scalar bit 3	Result bit 3	Point bit 3
4	Scalar bit 4	Result bit 4	Point bit 4
5	Scalar bit 5	Result bit 5	Point bit 5
6	Scalar bit 6	Result bit 6	Point bit 6
7	Scalar bit 7	Result bit 7	Point bit 7

LEA-128

by **awayzer JCT**

`0195` `1 MHz` `HDL Project`

`github.com/awayzer/ttgf-jct-LEA128`

> "*iterative approach implementation of the LEA128 block cipher*"

About

The design was implemented as part of an undergraduate academic project.

This design implements the LEA-128 cryptographic algorithm according to the specifications presented in the original paper by its developers [1] and is based on the ideas and architectures proposed in [2] and [3] for an area-efficient implementation.

The design supports both encryption and decryption operations using a very simple handshake-based byte-oriented interface.

How it works

The module receives:

- a 128-bit master key (16 bytes)
- a 128-bit data block (16 bytes)

The bytes are loaded serially through `ui_in` using an input handshake protocol.

After all bytes are loaded:

- `uio_in[3]` starts encryption
- `uio_in[4]` starts decryption

The processed output block is then transmitted byte-by-byte through `uo_out` using an output handshake protocol.

Pin usage

Pin	Function
`ui_in[7:0]`	Input byte bus
`uo_out[7:0]`	Output byte bus
`uio_in[0]`	request (input handshake)
`uio_in[1]`	request (output handshake)
`uio_in[2]`	acknowledge (output handshake)

uio_in[3]	Start encryption
uio_in[4]	Start decryption
uio_out[6]	valid (output handshake)
uio_out[7]	acknowledge (input handshake)

The remaining bidirectional pins are unused.

Input protocol

To send a byte into the design:

1. Place the byte on ui_in.
2. Set uio_in[0] high.
3. Wait until uio_out[7] becomes high.
4. Clear uio_in[0].
5. Wait until uio_out[7] returns low.

The first 16 transmitted bytes are interpreted as the encryption key. The next 16 transmitted bytes are interpreted as the input data block.

Output protocol

To receive the plaintext/ciphertext from the design:

1. Set uio_in[1] high.
2. Wait until uio_out[6] becomes high.
3. Read the byte from uo_out.
4. Pulse uio_in[2] high .
5. after repeating for all 16 bytes, Clear uio_in[1].

The output block is transmitted one byte at a time.

How to test

The test vector was sourced from Wikipedia. [4]

Encryption test

1. Reset the design by setting rst_n low.
2. Send the following 16-byte key:

```
text id="k1" 0f 1e 2d 3c 4b 5a 69 78 87 96 a5 b4 c3 d2 e1 f0
```

3. Send the following plaintext block:

```
text id="k2" 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
```

4. Pulse uio_in[3] high to start encryption.
5. Wait for the operation to complete.
6. Read 16 output bytes using the output handshake.

Expected ciphertext:

```
text id="k3" 9f c8 4e 35 28 c6 c6 18 55 32 c7 a7 04 64 8b fd
```

Decryption test

1. Reset the design.
2. Send the same 16-byte key.
3. Send the ciphertext block shown above.
4. Pulse `uio_in[4]` high to start decryption.
5. Read 16 output bytes.

Expected plaintext:

```
text id="k4" 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
```

References

[1] D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K. H. Ryu, and D.-G. Lee, "LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors," in Information Security and Cryptology – ICISC 2013, Springer, 2014.

[2] D. Lee, D.-C. Kim, D. Kwon, and H. Kim, "Efficient hardware implementation of the lightweight block encryption algorithm LEA," Sensors, vol. 14, no. 1, pp. 975–994, Jan. 2014. doi: 10.3390/s140100975.

[3] M.-J. Sung, G.-C. Bae, and K.-W. Shin, "Implementation of Lightweight Encryption Algorithm LEA," IDEC Journal of Integrated Circuits and Systems, vol. 2, no. 2, Jul. 2016.

[4] Wikipedia contributors, "LEA (cipher)," Wikipedia, The Free Encyclopedia. Available: https://en.wikipedia.org/wiki/LEA_(cipher). Licensed under CC BY-SA 4.0.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	req_rx
1	data_in[1]	data_out[1]	req_tx
2	data_in[2]	data_out[2]	ack_tx
3	data_in[3]	data_out[3]	enable_encryption
4	data_in[4]	data_out[4]	enable_decryption
5	data_in[5]	data_out[5]	—
6	data_in[6]	data_out[6]	valid_tx
7	data_in[7]	data_out[7]	ack_rx

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

Fast Authentication Accelerator

by **Nagaraj**

`0196` `50 MHz` `HDL Project`

`github.com/CambridgeinstitutetechnolgyBLR-Harshini/v2x_accelator`

> *"A hardware IP block that performs low-latency ECDSA verification for V2X communication. Authenticates digital signatures in under 1 ms, meeting IEEE 1609.2 automotive security requirements."*

Fast Authentication Accelerator

Low-Latency Digital Signature Verification for V2X Communication

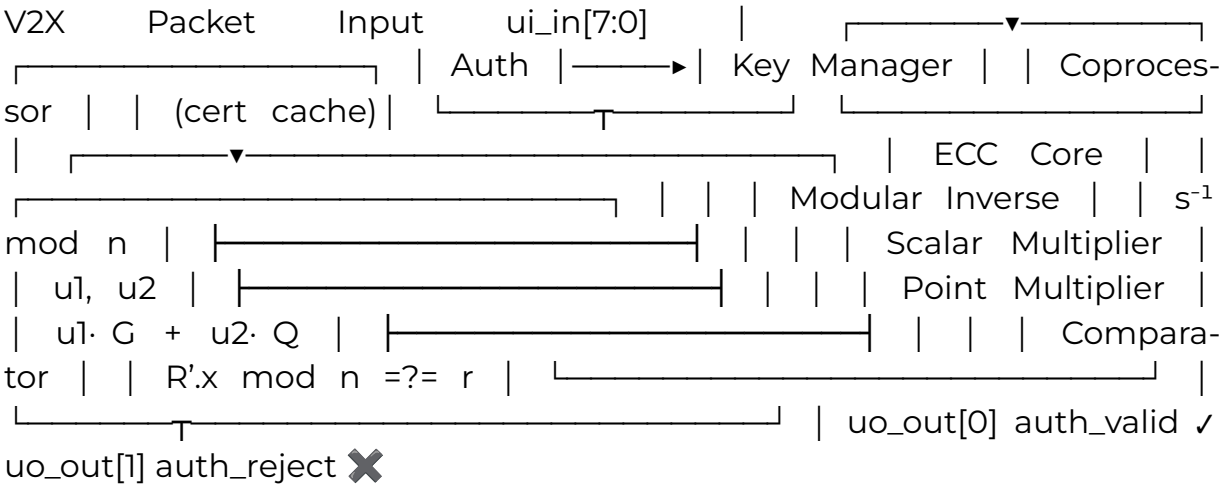
How it works

V2X (Vehicle-to-Everything) communication requires every transmitted message to be digitally authenticated in real time. At highway speeds this must happen in under **1 millisecond** — far too fast for general-purpose software.

This project implements a dedicated hardware IP block that performs **ECDSA (Elliptic Curve Digital Signature Algorithm)** verification, the industry standard for V2X security defined in **IEEE 1609.2**.

How to test

Architecture



Block	Function
Auth Coprocessor	Protocol parsing, FSM control, IEEE 1609.2 packet handling
Key Manager	Certificate and session key cache

Modular Inverse	Computes s^{-1} mod n using Fermat's Little Theorem
Scalar Multiplier	Computes u1 = e·w mod n and u2 = r·w mod n
Point Multiplier	Computes u1·G + u2·Q using double-and-add
Comparator	Checks R'.x mod n == r to accept or reject

ECDSA Verification Steps

1. Receive V2X packet: message hash `e`, signature `(r, s)`, public key `Q`
2. Compute `w = s⁻¹ mod n`
3. Compute `u1 = e·w mod n` and `u2 = r·w mod n`
4. Compute point `R' = u1·G + u2·Q`
5. Compare `R'.x mod n` with `r` — match = **VALID**, mismatch = **REJECT**

How to test

Pin Usage

Pin	Name	Description
`ui_in[7:0]`	`data_in`	Serialised packet/key bytes
`ui_in[0]`	`start`	Pulse HIGH to begin (in control mode)
`ui_in[1]`	`soft_rst`	Soft reset
`ui_in[3:2]`	`mode`	`00`=verify, `01`=load_key
`uo_out[0]`	`auth_valid`	HIGH = accepted
`uo_out[1]`	`auth_reject`	HIGH = rejected
`uo_out[2]`	`busy`	HIGH = computing
`uo_out[3]`	`ecc_done`	Pulse on each ECC step done
`uo_out[4]`	`key_loaded`	HIGH = key ready
`uo_out[5]`	`packet_ready`	HIGH = packet received

Operation Sequence

1. Apply `rst_n` reset
2. Set `ui_in[3:2]` = `01` (load_key mode), pulse `ui_in[0]` (start)
3. Stream 64 key bytes on `ui_in[7:0]`
4. Wait for `uo_out[4]` (key_loaded) = HIGH
5. Set `ui_in[3:2]` = `00` (verify mode), pulse `ui_in[0]` (start)
6. Stream packet bytes on `ui_in[7:0]`
7. Wait for `uo_out[2]` (busy) = LOW
8. Read `uo_out[0]` (auth_valid) or `uo_out[1]` (auth_reject)

Simulation

```
cd test/
make
```

External hardware

None required.

References

- IEEE 1609.2 — V2X Security Services
- FIPS 186-4 — Digital Signature Standard, NIST
- AEC-Q100 — Automotive IC reliability
- RFC 6979 — Deterministic ECDSA

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	start - Pulse HIGH to begin	auth_valid - HIGH when signature verified	—
1	soft_rst - Soft reset active HIGH	auth_reject - HIGH when signature rejected	—
2	mode0 - Mode select bit 0	busy - HIGH while processing	—
3	mode1 - Mode select bit 1	ecc_done - Pulses when ECC step done	—
4	data_in bit 4	key_loaded - HIGH when key loaded	—
5	data_in bit 5	packet_ready - HIGH when packet received	—
6	data_in bit 6	—	—
7	data_in bit 7	—	—

TRNG using Ring Oscillator

by **Karthik**

0198 HDL Project

github.com/cambridgeinstituetechnologyBLR-karthikp/trng_

"True Random Number Generator using ring oscillator"

TRNG using Ring Oscillator

How it works

This project implements a True Random Number Generator (TRNG) using a Ring Oscillator.

The ring oscillator continuously oscillates because of inverter delay. A sampler captures the oscillating signal using a clock signal. The sampled bits are checked using a health checker to detect repeated patterns. Post-processing improves randomness and generates the final random output bits.

Main modules used:

- Ring Oscillator
- Sampler
- Health Checker
- Post Processing Unit

How to test

1. Run Verilog simulation using Icarus Verilog.
2. Open waveform using GTKWave.
3. Observe oscillation in the ring oscillator.
4. Observe sampled random bits and clean output bits.

Simulation commands:

```bash iverilog -o trng_out *.v vvp trng_out gtkwave trng.vcd

## Project Pinout

### Digital Pins

| # | Input | Output | Bidirectional |
|---|---|---|---|
| 0 | Input 0 | Output 0 | Unused |
| 1 | Input 1 | Output 1 | Unused |
| 2 | Input 2 | Output 2 | Unused |
| 3 | Input 3 | Output 3 | Unused |
```

#	Input	Output	Bidirectional
4	Input 4	Output 4	Unused
5	Input 5	Output 5	Unused
6	Input 6	Output 6	Unused
7	Input 7	Output 7	Unused

100Mpbs 3 port Ethernet switch

by **Julia Desmazes**

`0199` `50 MHz` `HDL Project`

`github.com/Essenceia/ethernet_switch`

> *"Unmanaged cut-though 3 port 100Mbps ethernet switch"*

3 port cut-through 100Mbps ethernet switch ASIC targetting the Global Foundary 180nm MCU node.

Overview

This is an simple 3-port unmanaged 100Mbps Ethernet Switch, just plug in the ethernet and let it route packets.

Packets will be routed based on there mac addresses and the switch will keep track of the addresses of the devices connected to each port by tracking the `source address` field in the ethernet frame header of incomming ethernet packets.

Setup

Connect the 3 RMII PHY interfaces to RMII compliant PHY chips such as the LAN8720A, all the PHYs and the ASIC should be using the same external 50MHz reference clock.

If your PHY board doesn't expose the `rx_err` signal connect tie the pins to ground.

Similarly to the Ethernet accelerator wrapper ASIC project, this ASIC features a `tx_phase` signal to select the phase shift between the ASIC internal reference clock on the tx output data 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

A link to a test PCB design will be provided at a later date.

Testing

One the ASIC has been connected to the PHY and one ethernet port is connected to your computer and the other is connected to another piece of

networking equipement connected to the internet like a router or another switch, try pining google servers to quickly confirm traffic is being routed through the switch.

In your computer's terminal:

```
ping -I <wire_ethernet_interface> 8.8.8.8
```

Expected behavior:

```
ping -I enp2s0 8.8.8.8
PING 8.8.8.8 (8.8.8.8) from 192.168.0.119 enp2s0: 56(84) bytes of
data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=116 time=23.7 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=116 time=18.9 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=116 time=21.1 ms
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 18.872/21.242/23.730/1.985 ms
```

Diagnosing issues

If you're ping is not going through you can start by checking if packets are being received by your computer over your wired interface using `ethtool`:

```
sudo ethtool -S <wired_ethernet_interface>
```

Example output :

```
sudo ethtool -S enp2s0
NIC statistics:
     tx_packets: 549079
     rx_packets: 433146 <-----
     tx_errors: 0
     rx_errors: 4106 <----
     rx_missed: 0
     align_errors: 1370
     tx_single_collisions: 0
     tx_multi_collisions: 0
     unicast: 427139
     broadcast: 1090
     multicast: 4917
     tx_aborted: 0
     tx_underrun: 0
```

External hardware

Ethernet 100BASE-T Pmod connector:

- 3x100Mbps RMII compliant PHYs (LAN8720A)
- 1x50MHz oscillator

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	phy0_rx_data[0]	phy0_tx_data[0]	phy2_rx_data[0]
1	phy0_rx_data[1]	phy0_tx_data[1]	phy2_rx_data[1]
2	phy0_rx_v	phy0_tx_v	phy2_rx_v
3	phy0_rx_err	—	phy2_rx_err
4	phy1_rx_data[0]	—	tx_phase
5	phy1_rx_data[1]	phy1_tx_data[0]	phy2_tx_data[0]
6	phy1_rx_v	phy1_tx_data[1]	phy2_tx_data[1]
7	phy1_rx_err	phy1_tx_v	phy2_tx_v

100Mbps Ethernet Accelerator Wrapper

by **Julia Desmazes**

`0288` `50 MHz` `HDL Project`

`github.com/Essenceia/Teapot`

"Network connected accelerator wrapper, connected to custom bfloat16 multiplier."

ASIC ethernet accelerator wrapper.

How it works

TODO

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).

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_sender wlp3s0
```

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

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

The ASIC will only respond to application packet (ethtype:`0x88b5`) requests sent to it, a packet must me sent first to initiate a response.

External hardware

Ethernet 100BASE-T Pmod connector, featuring:

- LAN8720A PHY
- 50MHz oscillator

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	—	—

ASCON Integrated Crypto Processor

by **Dr. Mohamed El-Hadedy**

0295 50 MHz HDL Project

`github.com/mealycpp/tt-ascon-full`

> *"AEAD-only SDMC ASCON cryptographic processor with UART command/control, using the CISC-style AEAD core. HASH, XOF, CXOF, and chain modes are intentionally excluded from this branch."*

How it works

Reconfigurable hardware implementation of the ASCON cryptographic family standardized in NIST SP 800-232. The chip supports ASCON-AEAD-128 (encryption and decryption), ASCON-Hash256, ASCON-XOF128 (single and chained), and ASCON-CXOF128 (single and chained). A single shared ASCON-p[12]/p[8] permutation core serves all modes through a mode controller.

The design also integrates an on-die research entropy source: a ring oscillator with NIST SP 800-90B health tests (Repetition Count, Adaptive Proportion), conditioned through ASCON-Hash256, and fed into an ASCON Hash_DRBG (SP 800-90A) for on-chip key and nonce generation.

A single UART carries logical channels for control, key, nonce, seed, associated data, customization string, plaintext, ciphertext, digest, and XOF output.

How to test

Connect a host (PC, microcontroller, FPGA) to the UART pins at the chip clock rate. Frame format: SOF, channel, mode, length, payload, CRC16, EOF. Channel 0 = control/key/nonce/seed. Channel 1 = AD/customization. Channel 2 = plaintext/ciphertext/digest/output. Mode byte selects the ASCON variant. Verification is byte-exact against pyascon reference across all NIST KAT vectors.

External hardware

Optional external entropy source can be supplied via `ext_entropy_in`. A `trng_raw_dbg` debug pin exposes raw ring-oscillator bits for post-silicon NIST SP 800-90B entropy characterization.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	uart_rx_cmd_data	uart_tx_result	—
1	unused	constant_high	—
2	unused	uart_tx_result_mirror	—
3	reserved	busy	—
4	—	done	—
5	—	error_sticky	—
6	—	auth_ok	—
7	—	heartbeat	—

Procedural ASIC

by **Conner Daehler**

`0352` `10 MHz` `HDL Project`

`github.com/wiredlab/daehler-procedural_graphics_core`

> "*animated procedural graphics using XOR and arithmetic patterns*"

How it works

This project implements a **scan-based procedural graphics generator** that produces an 8-bit grayscale pixel stream using arithmetic and bitwise logic. The design behaves like a simplified fragment shader pipeline, where each pixel is computed on-the-fly from its current position rather than being stored in a framebuffer.

Internally, the system contains a **raster scan generator** (x/y counters) that continuously iterates over a virtual 256×256 image grid. For each `(x, y)` coordinate, a **mode-selectable combinational shader core** computes the output pixel value using different mathematical operations such as gradients, distance-based lighting, and XOR-based procedural noise.

A small temporal counter is optionally used to introduce animation in some modes, and a final output register ensures stable pixel output timing. Optional control signals allow the output to be inverted or animation to be frozen.

Overall, this design functions as a **streaming procedural fragment processor without framebuffers or external memory**.

How to test

The design is tested using a **cocotb-based Python testbench** that reconstructs full images from the continuous pixel output stream.

The testbench:

- Simulates a scan over a full frame
- Captures sequential pixel values
- Reconstructs a 2D image buffer for visualization
- Generates one output image per mode (0–3)

This allows verification of:

- spatial gradients
- lighting behavior
- procedural noise stability
- mode switching correctness

External hardware

No external hardware is required. The design operates entirely in digital logic and produces a continuous pixel stream suitable for simulation and FPGA/ASIC integration.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	GFX_MODE_0	PIXEL_OUT_0	—
1	GFX_MODE_1	PIXEL_OUT_1	—
2	FREEZE_ANIMATION	PIXEL_OUT_2	—
3	INVERT_COLORS	PIXEL_OUT_3	—
4	—	PIXEL_OUT_4	—
5	—	PIXEL_OUT_5	—
6	—	PIXEL_OUT_6	—
7	—	PIXEL_OUT_7	—

ttgf jct PoC

by **awayzer JCT**

`0353` `500 kHz` `HDL Project`

`github.com/awayzer/ttgf-jct-PoC`

> "*Naive and very simple implementation of a stream cipher*"

How it works

This project is a naive and very simple implementation of a stream cipher demonstration.

The design contains two internal 8-bit registers:

- `data_mem` — stores the input data.
- `key_mem` — stores the encryption key.

The output is generated by XORing the stored data and key values:

```
uo_out = data_mem ^ key_mem
```

The module uses `uio_in[0]` as a synchronization/control signal. A rising edge on this pin triggers a register update.

The pin `uio_in[1]` selects which register is updated:

`uio_in[1]`	Action
0	Load `ui_in` into `data_mem`
1	Load `ui_in` into `key_mem`

The bidirectional pins are configured as inputs only, and are never driven by the design.

Pin usage

Pin	Function
`ui_in[7:0]`	8-bit input data/key bus
`uio_in[0]`	Load trigger (rising edge sensitive)
`uio_in[1]`	Register select
`uo_out[7:0]`	XOR result (`data_mem ^ key_mem`)

How to test

1. Reset the design by setting `rst_n` low.
2. Set `uio_in[1] = 0`.

3. Place a byte on `ui_in`.
4. Generate a rising edge on `uio_in[0]` to load the value into `data_mem`.
5. Set `uio_in[1] = 1`.
6. Place another byte on `ui_in`.
7. Generate another rising edge on `uio_in[0]` to load the value into `key_mem`.
8. Observe that `uo_out` equals:

```
data_mem XOR key_mem
```

Example:

data_mem	key_mem	uo_out
8'hAA	8'hFF	8'h55

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]/key_in[0]	output[0]	valid
1	data_in[1]/key_in[1]	output[1]	is_key
2	data_in[2]/key_in[2]	output[2]	—
3	data_in[3]/key_in[3]	output[3]	—
4	data_in[4]/key_in[4]	output[4]	—
5	data_in[5]/key_in[5]	output[5]	—
6	data_in[6]/key_in[6]	output[6]	—
7	data_in[7]/key_in[7]	output[7]	—

PWM-Analyser

by **Pauline Kreis** & **Muhammad Saim Bilal**

`0354` `62.5 MHz` `HDL Project`

`github.com/PaulineKreis/TTGF26b_PWM-Analyser`

"Analyses PWM signal and delivers duty Cycle and frequency measurements on a 7-segment display"

How it works

This project is meant to analyse the key characteristics of an input PWM signal, namely its **duty cycle** and **frequency**. It outputs these characteristics to a 4-digit 7-segment LED display. An additional mode-switch signal can be used to toggle what property is being displayed, *LO: Duty Cycle, HI: Frequency*. The frequency counter module is designed for measuring frequencies between **1 and 9999 KHz**, any lower or higher frequency signal shall result in a **LO or HI** message on the Display. If the PWM signal is faulty and stays constant for too long the the frequency display shall output an **ERR** message.

How to test

This design can be tested using another either a microcontroller or signal generator to generate a PWM signal. Please note that the voltage of said signal is appropriate so as to not destroy the device.

External hardware

- 4-digit 7-segment display with decimal point
- A PWM Signal Source

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PWM_INPUT	G	DIGIT_EN_3
1	MODE_SWITCH	F	DIGIT_EN_2
2	NC	E	DIGIT_EN_1
3	NC	D	DIGIT_EN_0
4	NC	C	NC
5	NC	B	NC
6	NC	A	NC

#	Input	Output	Bidirectional
7	NC	DP	NC

CORDIC sin/cos generator

by **Jacques Te**

`0355` `50 MHz` `HDL Project`

`github.com/JacquesBTe/tt_um_jte_cordic`

“ Computes the sine and cosine of an 8-bit input angle using an iterative 8-stage CORDIC algorithm in Q2.6 fixed-point.”

How it works

This project computes the **sine and cosine** of an input angle using an iterative **CORDIC** (COordinate Rotation DIgital Computer) algorithm.

The angle is supplied as a 7-bit value on `ui[6:0]`, where `0`–`127` maps linearly to `0°`–`360°` (so each step is `360 / 128 ≈ 2.81°`). Asserting `start` on `ui[7]` begins a computation.

Internally the design works in **Q2.6 fixed-point** (8-bit two’s-complement, where `1.0 = 64 = 0x40`):

1. **Quadrant fold** — the angle is reduced to the first quadrant (0°–90°), and per-quadrant sign flags are recorded so the result can be reflected back into the correct quadrant.
2. **Rotation** — a vector is pre-scaled by the CORDIC gain (`1/K ≈ 0.607`) and rotated through **8 fixed micro-rotations** of `arctan(2^-i)`, each taken from a small lookup table. A small FSM (`IDLE → LOAD → 8× COMPUTE → DONE`) sequences the iterations; each iteration drives a residual-angle accumulator toward zero while the vector’s X and Y components converge to `cos` and `sin`.
3. **Sign correction** — the recorded quadrant flags are applied, producing the final signed `sin` and `cos`.

A result is ready roughly **11 clock cycles** after `start` is asserted. `start` is a level: hold it high to recompute continuously, release it to freeze and hold the last result.

The outputs are 8-bit two’s-complement Q2.6 values:

Raw (hex)	Decimal	Real value
`0x40`	+64	+1.0
`0x00`	0	0.0
`0xC0`	−64	−1.0

(Convert any output to its real value with `raw / 64`.)

`sin` appears on `uo[7:0]`, `cos` on `uio[7:0]` (the bidirectional pins, configured as outputs).

How to test

1. Put a 7-bit angle on `ui[6:0]` (0–127 = 0°–360°).
2. Drive `start` (`ui[7]`) high to trigger a computation, then low to latch the result.
3. After 11 clock cycles, read `sin` on `uo[7:0]` and `cos` on `uio[7:0]`. Interpret both as signed Q2.6 (`value = raw / 64`).

Example angles (input code → angle → expected outputs):

`ui[6:0]`	Angle	`sin` (uo)	`cos` (uio)
0	0°	`0x00` (0)	`0x40` (+1.0)
16	45°	`0x2E` (≈0.72)	`0x2C` (≈0.69)
32	90°	`0x41` (≈+1.0)	`0x01` (≈0)
64	180°	`0x00` (≈0)	`0xC0` (−1.0)
96	270°	`0xBF` (≈−1.0)	`0x01` (≈0)

Outputs are within ±a few LSB of the ideal due to the 8-iteration, 8-bit fixed-point approximation. The included cocotb test (`test/test.py`) sweeps all 128 angles and checks them bit-exactly against a golden CORDIC model.

External hardware

No external hardware is required to *drive* the design — the angle and `start` come from the demo board's input switches.

To *view* the outputs:

- **`sin`** is on `uo[7:0]`, which drives the demo board's 8 on-board LEDs directly.
- **`cos`** is on `uio[7:0]` (the bidirectional/PMOD pins). To see it, attach an **8-bit LED PMOD** (or any 8-LED breakout) to the `uio` header. This is optional — `cos` is still readable via the RP2040/test harness without it.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	angle[0]	sin[0]	cos[0]
1	angle[1]	sin[1]	cos[1]
2	angle[2]	sin[2]	cos[2]

#	Input	Output	Bidirectional
3	angle[3]	sin[3]	cos[3]
4	angle[4]	sin[4]	cos[4]
5	angle[5]	sin[5]	cos[5]
6	angle[6]	sin[6]	cos[6]
7	start	sin[7]	cos[7]

Arctic0 16-bit CPU

by **Einosuke Okazaki**

`0356` `40 MHz` `HDL Project`

`github.com/PenguinEino/Arctic0`

"16-bit multi-cycle CPU: SPI flash instructions, SPI SRAM data, UART TX/RX, GPIO and register dump"

How it works

Arctic0 16-bit CPU is a multi-cycle CPU wrapped for Tiny Tapeout. The CPU sees one 16-bit address space. `MEM_CTRL` decodes each access and stalls the CPU clock while external SPI or UART work completes.

```
CPU -- 16-bit bus --> MEM_CTRL -- SPI --> Flash / SRAM
                              |-- UART TX/RX
                              `-- GPIO
```

Memory map

Address range	Use	Device
`0x0000–0x7EFF`	Instructions and constants	SPI NOR Flash, read only
`0x7F00–0x7FFF`	I/O registers	UART, dump, GPIO
`0x8000–0xFFFF`	Data and stack	SPI SRAM

External SPI devices are byte-addressed, while the CPU uses 16-bit words. The controller converts CPU word addresses to byte addresses by shifting left once.

I/O registers

Address	Access	Function
`0x7F00`	W	Transmit `DATA[7:0]` over UART
`0x7F10`	W	Emit a 13-byte register dump over UART
`0x7F11`	R	Status: bit0 = UART TX busy, bit1 = UART RX ready
`0x7F20`	R	UART RX data; read acknowledges and clears ready
`0x7F21`	R	GPIO input from `ui_in[7:0]`
`0x7F30`	W	GPIO output latch to `uo_out[7:0]`

Register dump format:

```
D0 01 PC_hi PC_lo A_hi A_lo B_hi B_lo SP_hi SP_lo IR PHASE FLAGS
```

`FLAGS = {6'b0, CF, ZF}`.

How to test

Connect an SPI NOR Flash containing a big-endian 16-bit word program, an SPI SRAM, and a USB-serial adapter. The default clock is 40 MHz, giving SCLK = clk/2 = 20 MHz and a UART rate of 115200 8N1 (BAUD_DIV = 347). Override `BAUD_DIV` when running at a different clock.

All simulations live under `test/`:

Directory	Coverage
`test/`	Tiny Tapeout top-level smoke test
`test/cpu_spi/`	CPU + Flash + SRAM + UART dump
`test/flash_prog/`	Execute programs loaded from `prog.hex`
`test/mem_boundary/`	Flash/I/O/SRAM decode boundaries
`test/io/`	UART RX plus GPIO
`test/spi/`	SPI master read/write transactions
`test/isa/`	Instruction-set coverage
`test/uart/`	UART TX/RX framing
`test/fpga/`	FPGA bring-up programs and Python host scripts

Run the top-level smoke test with:

```
PATH=/opt/homebrew/bin:$PATH make -B -C test
```

Run any sub-test the same way, e.g. `make -B -C test/io`.

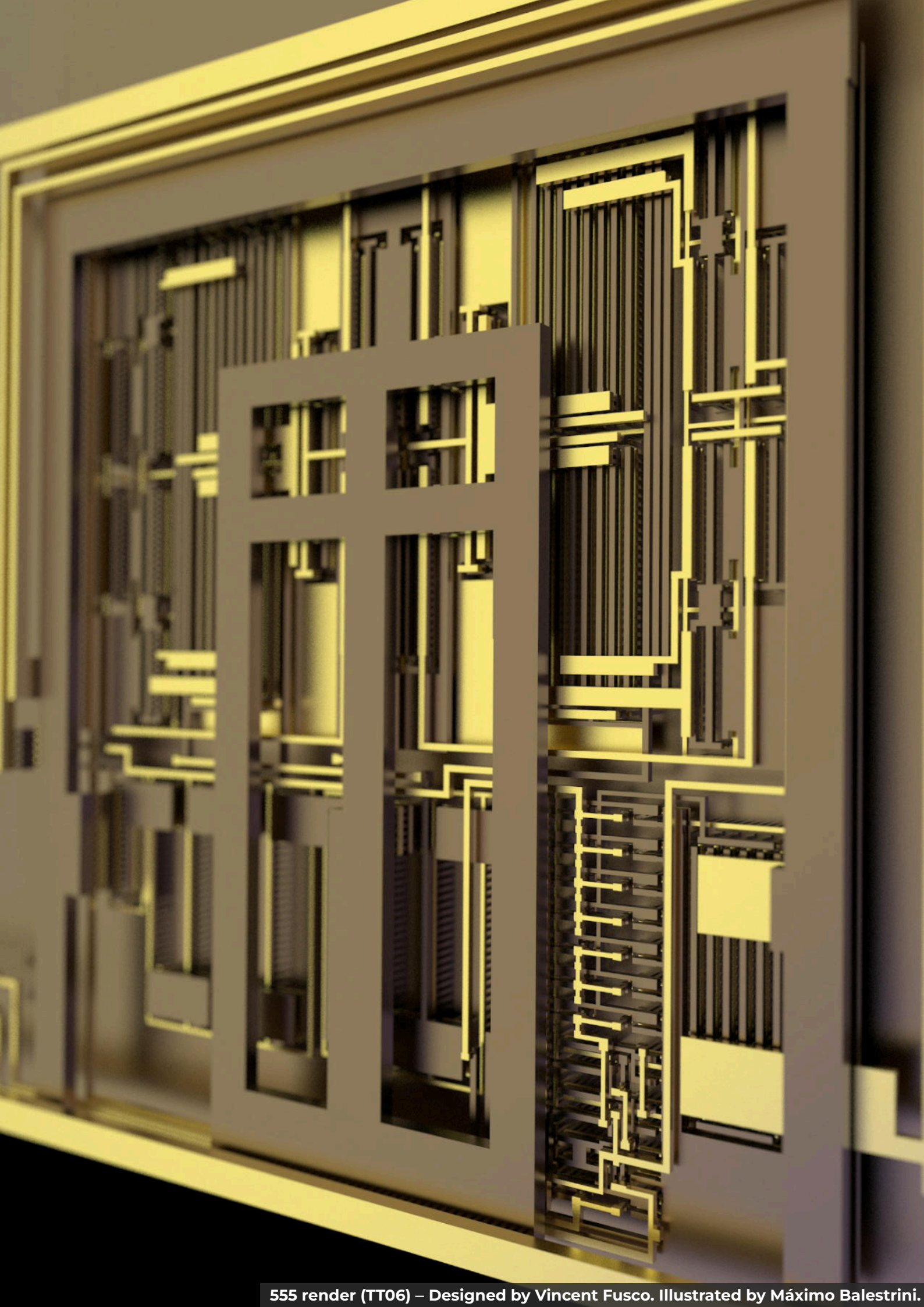
External hardware

Pin	Signal
`ui[7:0]`	GPIO input
`uo[7:0]`	GPIO output
`uio[0]`	Flash CS_n
`uio[1]`	SPI SCLK
`uio[2]`	SPI MOSI
`uio[3]`	SPI MISO
`uio[4]`	SRAM CS_n
`uio[5]`	UART TX
`uio[6]`	UART RX
`uio[7]`	unused

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	GPIO_IN0	GPIO_OUT0	FLASH_CS_N
1	GPIO_IN1	GPIO_OUT1	SPI_SCLK
2	GPIO_IN2	GPIO_OUT2	SPI_MOSI
3	GPIO_IN3	GPIO_OUT3	SPI_MISO
4	GPIO_IN4	GPIO_OUT4	SRAM_CS_N
5	GPIO_IN5	GPIO_OUT5	UART_TX
6	GPIO_IN6	GPIO_OUT6	UART_RX
7	GPIO_IN7	GPIO_OUT7	—



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

Three Channel RGB PWM Controller

by **Aiden Koch**

`0357` `HDL Project`

`github.com/aidenkoch4/ece430-tt-GF26A`

"Red Green Blue compatable Pulse Width Modulator to control effective voltages at an LED to control the color output. User defiend clock divider included to increase range of application usage."

How it works

This project is a three-channel RGB Pulse Width Modulation (PWM) controller designed for TinyTapeout.

The design stores independent brightness values for red, green, and blue LED channels and continuously generates PWM waveforms on the outputs. The PWM duty cycle determines the effective brightness of each color channel.

The 8-bit input bus is divided into:

Bits	Purpose
ui_in[7:6]	Mode selection
ui_in[5:0]	Value data

The mode bits determine which internal register is updated.

Mode	Function
00	Set RED brightness
01	Set GREEN brightness
10	Set BLUE brightness
11	Set PWM clock divider

The PWM engine operates using:

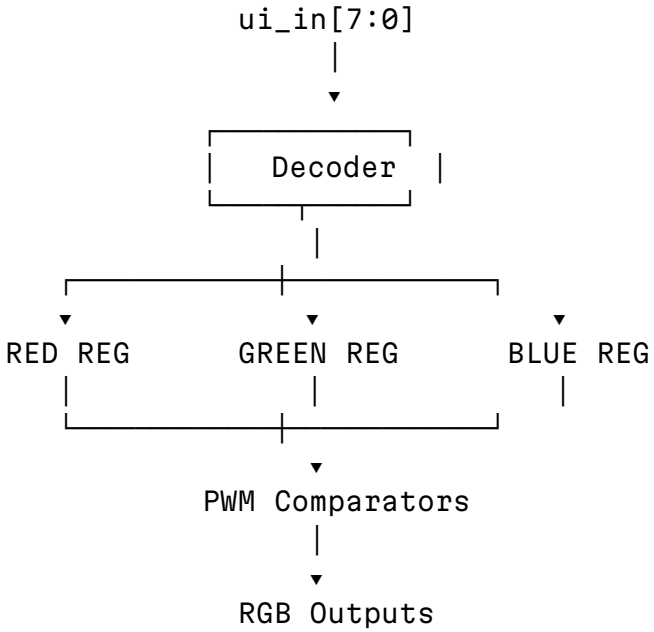
- a programmable clock divider
- a 6-bit PWM counter
- comparator-based PWM generation

Each output compares the PWM counter against its stored duty-cycle register:

```
PWM output HIGH when:
counter < duty_cycle
```

This creates variable duty-cycle square waves suitable for LED brightness control.

Internal Architecture



PWM Generation

A programmable clock divider slows the incoming TinyTapeout clock to generate a visible PWM frequency.

A 6-bit counter continuously ramps:

`0 → 1 → 2 → ... → 63 → repeat`

The RGB outputs turn on whenever the counter value is below the stored duty-cycle value.

Example:

```
Duty = 32

counter < 32 → HIGH
counter ≥ 32 → LOW
```

This creates a 50% duty cycle.

How to test

The design is tested using cocotb and Icarus Verilog simulation.

Simulation Inputs

The user controls the design through the `ui_in[7:0]` bus.

Example Commands

Set RED brightness to maximum

```
mode  = 00
value = 63

ui_in = 00111111
```

Set GREEN brightness to medium brightness

```
mode  = 01
value = 32

ui_in = 01100000
```

Set BLUE brightness to low brightness

```
mode  = 10
value = 8

ui_in = 10001000
```

Change PWM speed

```
mode  = 11
value = 4

ui_in = 11000100
```

Running Simulation

Simulation is performed using:

```
cd test
make
```

Waveforms can then be viewed using GTKWave or Surfer.

Verification Goals

The testbench verifies:

- register updates
- PWM activity
- RGB output generation
- clock divider operation
- reset behavior

External hardware

The outputs are intended to drive:

- RGB LEDs
- transistor driver stages

- logic-level PWM-compatible devices

For direct LED use:

- connect current-limiting resistors in series with each LED channel
- connect PWM outputs to the LED control pins
- ensure current limits are respected

Typical usage:

```
uo_out[0] → Red LED
uo_out[1] → Green LED
uo_out[2] → Blue LED
```

The design may also interface with:

- MOSFET drivers
- LED strips
- external power stages
- digital lighting systems

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	value bit 0	PWM Red	—
1	value bit 1	PWM Green	—
2	value bit 2	PWM Blue	—
3	value bit 3	—	—
4	value bit 4	—	—
5	value bit 5	—	—
6	mode bit 0	—	—
7	mode bit 1	—	—

Pong

by **Lukas Hahne**

`0358` `25.175 MHz` `HDL Project`

`github.com/ljhahne/ttgf-pong`

"Simple implementation of Pong"

How it works

This is a simple implementation of the classic video game Pong, developed and verified on the Analogue Pocket and in the VGA Playground.

The design generates a VGA video signal through the Tiny VGA Pmod, so it needs to be connected to a VGA-capable monitor or capture device. Player input comes from a single Gamepad Pmod: the D-pad's Up/Down buttons move the left paddle, and the A/B buttons move the right paddle, so two players can share one SNES-style controller.

How to test

Connect a Gamepad Pmod and a Tiny VGA Pmod to the project. Use Up/Down on the gamepad to move the left paddle and A/B to move the right paddle, and check that the ball bounces correctly and the score updates for both players on the VGA output. This is my first hardware project, so testing has been visual ("eyeballed") on the Analogue Pocket and in the VGA Playground rather than with an automated cocotb test bench.

External hardware

Gamepad Pmod and Tiny VGA Pmod, both available in the Tiny Tapeout shop.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	R1	—
1	—	G1	—
2	—	B1	—
3	—	VSync	—
4	Gamepad Latch	R0	—
5	Gamepad CLK	G0	—
6	Gamepad Data	B0	—

#	Input	Output	Bidirectional
7	—	HSync	—

Tremolo guitar pedal ASIC

by **Preston Schuetz**

`0359` `50 MHz` `HDL Project`

`github.com/Pschuetz112/tt-guitar-pedal-tremolo_final`

"This is a PWM controller for a tremolo guitar pedal"

How it works

This project implements a digital tremolo modulation controller, intended for guitar pedal applications. This design generates PWM (pulse-width modulated) signal that can be converted into a smooth analog control signal later on with an RC filter. At the center of the design is a LFO (Low Frequency Oscillator) that controls how the volume of the guitar changes over time. Internally there is an 8-bit LFO that counts with a frequency controlled by the rate select logic. This is controlled by rate select inputs [2:1]. Different rate selections change a divider value inside the chip that determines how fast the LFO counts. The current LFO value is then sent into the waveform generator. This block changes the counter value into different modulation shapes. Also fed into this block is the shape select input. The chip has 4 modes: saw up, saw down, square, and triangle. That then passes into the "depth logic". This section controls how intense the actual tremolo effect is. Low depth creates subtle volume changes, while high creates "aggressive" volume swings. The resulting modulation level is then compared against a PWM counter using a comparator and the resulting output is the PWM signal that will control analog gain staging. Full block diagram can be seen in the block diagram text file in this same docs folder.

How to test

The test bench is verified using Cocotb testbenches and the GitHub actions page on my repository. The testbench verifies the following: -reset state -enable/disable functionality -PWM signal generation -rate selection -waveform selection -depth selection -debug outputs

Waveforms can be viewed by using GTKwave

External hardware

Because the chip only outputs a PWM control signal, this project will need quite a bit of other hardware to work:

RC lowpass filtering Op-amp buffer Analog tremolo staging

Possible hardware ideas to try: Vactrol-based tremolo Jfet-based tremolo Operational transconductance/ voltage controlled amplifier tremolo

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Enable Tremolo	PWM tremolo control output	—
1	Rate select bit 0	LFO debug output	—
2	Rate select bit 1	PWM carrier debug output	—
3	Depth select bit 0	Enable status	—
4	Depth select bit 1	Mod level debug bit	—
5	Waveform select bit 0	Mod level debug bit	—
6	Waveform select bit 1	Mod level debug bit	—
7	Test speed select	Mod level debug bit	—

PWM Generator

by **Omar** & **Nitin**

`0385` `10 MHz` `HDL Project`

`github.com/Omar90551/TTGF-PWM`

"A PWM generator with configurable period and duty cycle via 8-bit input pins"

How it works

The PWM Generator is a configurable digital hardware module that produces a continuous pulse-width modulated (PWM) signal. It utilizes a two-process finite state machine structure, consisting of an 8-bit counter and an 8-bit comparator.

The user configures the total cycle period via the 8 dedicated input pins (`ui_in`) and the active duty cycle duration via the 8 bidirectional I/O pins configured as inputs (`uio_in`). The module continuously increments an internal counter and compares it against the duty cycle configuration to drive the output signal HIGH or LOW. The resulting PWM waveform is output on the primary output pin (`uo_out[0]`). The design features both an asynchronous reset and a synchronous enable to provide safe initialization and instant halting.

How to test

To physically verify the chip's functionality, follow this testing sequence:

1. **Initialization:** Power the device and assert the asynchronous reset (`rst_n = 0`) to clear the internal counter, ensuring the output is LOW.
2. **Enable:** Release the reset (`rst_n = 1`) and assert the enable signal (`ena = 1`).
3. **Configure Period:** Apply an 8-bit binary value to the dedicated input pins (`ui_in`) to define the total duration of one PWM cycle (e.g., `00001010` for 10 clock cycles).
4. **Configure Duty Cycle:** Apply an 8-bit binary value to the bidirectional input pins (`uio_in`) to define how long the signal remains HIGH during the period.
   - Setting `uio_in` to exactly half of `ui_in` will generate a 50% duty cycle.
   - Setting `uio_in` to `00000000` will force the output to 0% (always LOW).
   - Setting `uio_in` to a value equal to or greater than `ui_in` will saturate the output to 100% (always HIGH).

5. **Observation:** Observe the output on `uo_out[0]`. If connected to an LED, the visual brightness of the LED will scale linearly as you adjust the duty cycle input.

External hardware

To visually observe the PWM generation, the following external hardware is recommended:

- **LED / PMOD LED Module:** Connect an LED (with an appropriate current-limiting resistor) to the `uo_out[0]` pin to observe the varying brightness levels corresponding to different duty cycle configurations.
- **Oscilloscope or Logic Analyzer (Optional):** For precise timing verification, connect a logic probe to `uo_out[0]` to measure the exact frequency and pulse width of the generated high/low phases.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	period_cfg bit 0	pwm_out	duty_cfg bit 0
1	period_cfg bit 1	Unused (Grounded)	duty_cfg bit 1
2	period_cfg bit 2	Unused (Grounded)	duty_cfg bit 2
3	period_cfg bit 3	Unused (Grounded)	duty_cfg bit 3
4	period_cfg bit 4	Unused (Grounded)	duty_cfg bit 4
5	period_cfg bit 5	Unused (Grounded)	duty_cfg bit 5
6	period_cfg bit 6	Unused (Grounded)	duty_cfg bit 6
7	period_cfg bit 7	Unused (Grounded)	duty_cfg bit 7

Elemental Harmony Game

by **Aakarshitha Suresh**

`386` `10 MHz` `HDL Project`

`github.com/Aakarshitha/Elemental_Harmony_GF26b`

*"Elemental Harmony is a silicon-based 4 * 4 strategy game, like Tic-Tac-Toe, where players compete against a logic engine to score points by matching elemental patterns across a digital grid."*

How it works

🌟 Elemental Harmony A TinyTapeout Puzzle of Balance and Chaos

🎯 Objective Bring balance to a 4×4 elemental board by placing patterns (Air, Water, Fire, Earth) such that harmony outnumbers chaos. Win by creating rows or columns that embody elemental cooperation. Lose if conflict overtakes the board — or if you exhaust your allowed retries.

🌍 The Elements & Their Patterns

🌍 The Elements & Their Patterns

Element	Patterns	Symbol	Meaning
Air	AB (Breeze), AG (Gust)	💨	Movement & flow
Water	WD (Drop), WR (Ripple)	💧	Calm & renewal
Fire	FB (Blaze), FS (Spark)	🔥	Energy & creation
Earth	EG (Grain), EP (Pebble)	🌿	Stability & grounding

Each pattern has its own personality — some blend, others clash.

Each pattern has its own personality => some blend, others clash.

Elemental Harmony is a silicon-native strategy game implemented as a TinyTapeout-compatible hardware module. It utilizes an internal 16-bit occupancy register to manage a 4 * 4 grid, where a Finite State Machine (FSM) coordinates turns between a human player and an internal logic engine. The engine uses a Linear Feedback Shift Register (LFSR) to search for empty tiles and calculates scores based on adjacent "elemental" pattern matches using a combinational adder tree. A unique feature of the architecture is its tri-state error reporting system: if a player attempts to place a pattern on an occupied tile, the FSM transitions through a sequence of diagnostic states (ERROR1

through ERROR3) that stream the current board occupancy and fill-count back to the interface, allowing for external recovery and move retry without a full system reset.

PAIRWISE PATTERN SCORE TABLE As created in the specifications of the design Elemental Harmony. Here is the complete 8×8 Pairwise Pattern Score Table exactly as defined in the spec. Definition: Row = newly placed pattern Column = existing neighboring pattern (N/E/S/W) Values: +2 (strong), +1 (mild), 0 (same), −2 (conflict)

8×8 Pairwise Score LUT

New \ Nbr	AB	AG	WD	WR	FB	FS	EG	EP
AB (0)	0	+1	−2	−2	−2	+2	−2	+2
AG (1)	+1	0	+2	−2	+2	−2	−2	+2
WD (2)	−2	+2	0	+1	−2	+2	+2	+2
WR (3)	−2	−2	+1	0	−2	+2	+2	+2
FB (4)	−2	+2	−2	−2	0	+1	−2	+2
FS (5)	+2	−2	+2	+2	+1	0	−2	+2
EG (6)	−2	−2	+2	+2	−2	−2	0	+1
EP (7)	+2	+2	+2	+2	+2	+2	+1	0

Figure 386.1: Elemental Patterns

How to test

To verify the design, pull rst_n low to initialize the grid, then provide a 4-bit tile address and 3-bit pattern on ui_in before pulsing the Start bit (ui_in[7]). Monitor the uo_out port for the resulting Harmony score, which is valid when the strobe bit (uio_out[0]) is high. To test the robustness of the error-handling logic, intentionally attempt to place a pattern on a previously occupied tile; observe the FSM transition into the error states and verify that uo_out sequentially outputs the fill status and the 16-bit occupancy map (split into two 8-bit chunks). The test is successful if the system returns to the IDLE state after an error, ready to accept a corrected move at a valid, non-colliding position.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	h_pos[0]	uo_out[0]	design_done_strobe
1	h_pos[1]	uo_out[1]	fsm_state_out[0]
2	h_pos[2]	uo_out[2]	fsm_state_out[1]
3	h_pos[3]	uo_out[3]	fsm_state_out[2]

#	Input	Output	Bidirectional
4	h_pat[0]	uo_out[4]	fsm_state_out[3]
5	h_pat[1]	uo_out[5]	unused_input[0]
6	h_pat[2]	uo_out[6]	unused_input[1]
7	start_pulse	uo_out[7]	unused_input[2]

WashingMachine_FSM

by **Charithma Perera** & **Kim Luu**

0387 | 100 MHz | HDL Project

github.com/KimLuu02/TTGF26b_WashingMachine-FSM

"Washing Machine Control"

How it works

The washing machine controller is a modular digital system implemented in Verilog using a finite-state-machine (FSM). The system simulates a simplified washing process and automatically progresses through different washing stages. If the door is closed, which should be simultaed through a lever, the start button can be pressed. If wanted, the duration of the washing process can be changed, by using the lever.

How to test

Use 1 button for Start and 2 lever for mode_select and for door_closd. Use LEDs to see the output.

External hardware

- LED 4
- Debounced Button 1
- Lever 2

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	START	WATER_VALVE	NC
1	MODE_SELECT	WASH_MOTOR	NC
2	DOOR_CLOSED	SPIN_MOTOR	NC
3	NC	DONE_LED	NC
4	NC	NC	NC
5	NC	NC	NC
6	NC	NC	NC
7	NC	NC	NC

Traffic Light FSM

by **Eric Kiecksee**

`389` `12 MHz` `HDL Project`

`github.com/Kieckenwama/TTGF26b_Traffic_Ligth_FSM`

"Simple traffic light FSM for intersection with main road, side road and pedestrian phases"

How it works

A digital traffic light controller implemented as a finite state machine (FSM) in Verilog. The design cycles through RED, YELLOW, and GREEN phases with durations controlled by an internal timer module.

How to test

The design implements a traffic light FSM with five states (S0–S4) controlling three sets of lights: main road, side road, and pedestrian.

Inputs

Signal	Description
`clk`	12 MHz system clock
`rst_n`	Asynchronous reset, active low
`ped_req`	Pedestrian request, active high

Outputs

Signal	Description
`main_green`	Main road green light
`main_yellow`	Main road yellow light
`main_red`	Main road red light
`side_green`	Side road green light
`side_yellow`	Side road yellow light
`side_red`	Side road red light
`ped_green`	Pedestrian green light
`ped_red`	Pedestrian red light

Test Procedure

Test 1 — Reset: Assert `rst_n = 0` for at least two clock cycles, then release. Verify that `main_green = 1`, `side_red = 1`, `ped_red = 1` within one clock cycle after release. All other outputs must be low.

Test 2 — Normal state sequence: Verify the FSM cycles through S0 (10 s) → S1 (2 s) → S2 (5 s) → S3 (2 s) → S0 without entering S4. When `ped_req = 0`. Verify that `main_yellow = 1` immediately before `main_red` becomes active, and `side_yellow = 1` immediately before `side_red` becomes active.

Test 3 — Pedestrian request: Assert `ped_req = 1` briefly during S0, then release before S1. Verify the FSM follows S0 → S1 → S4 (3 s) → S2 → S3 → S0. During S4, `ped_green = 1` and `main_red = side_red = 1`.

Test 4 — Mutual exclusion: At every clock cycle, verify:

- `main_green AND side_green` is never high
- `ped_green AND main_green` is never high
- `ped_green AND side_green` is never high

External hardware

8 LEDs connected to the output signals main_green, main_yellow, main_red, side_green, side_yellow, side_red, ped_green, ped_red with appropriate current-limiting resistors (330 Ω recommended).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Pedestrian Button	Main Green	NC
1	NC	Main Yellow	NC
2	NC	Main Red	NC
3	NC	Side Green	NC
4	NC	Side Yellow	NC
5	NC	Side Red	NC
6	NC	Pedestrian Green	NC
7	NC	Pedestrian Red	NC

Smerity-Mandelbrot

by **Stephen Merity**

`0390` `10 MHz` `HDL Project`

`github.com/Smerity/tt_um_smerity_mandelbrot`

"An interactive fixed-point Mandelbrot iterator with a sequential (bit-serial) multiplier. Tested on iCEBreaker FPGA V1.1a."

How it works

A fixed-point **Mandelbrot** renderer. The chip walks a 320×240 raster; for each pixel it iterates `z = z² + c` in **Q4.28** (32-bit signed) and outputs the iteration count (0 = "in the set"). To keep the area small it uses a **single sequential bit serial shift-add multiplier**, shared across the three products per iteration.

The host owns the view, the chip is a pure pixel engine.

- **Input — view packet.** The host sends the **top-left corner** `c0` and the **step** (complex units per pixel) as three signed Q4.28 fields, plus a 1-byte **maxit** (runtime iteration limit) — 13 bytes total, MSB-first on `ui_in`, framed by `param_frame` and strobed by `param_valid`. The chip latches it and commits at the next `frame_start`, so a view (or maxit) change never tears a frame. A coordinate generator (DDA, pure adds) walks `c` across the frame. A default full view and maxit=100 are loaded on reset.
- **Output — pixel stream.** `uo_out` carries the pixel value; `pix_valid`, `frame_start`, and `line_end` strobes (on the bidir pins) delimit the stream. Because the markers are their own wires, pixel values use the full range.

The whole datapath is reset by `rst_n` (no power-on register state).

How to test

The chip free-runs, rendering the current view forever. On a TinyTapeout demo board, bridge the parallel pins to USB with an RP2040 PIO program: emit `0xFF` on `frame_start`, `0xFE` on `line_end`, and `uo_out` on `pix_valid`. Since pixel values stay below `0xFE`, this reproduces a simple framed byte stream that a desktop viewer can draw directly (map the iteration count to a grayscale/colour palette; 0 = in-set = dark).

To change the view (zoom/pan/iterations), compute `c0`, `step`, and `maxit` on the host and send the 13-byte packet over `ui_in` with the `param_valid` / `param_frame` handshake.

The included cocotb test resets the design, checks the bidirectional directions, and verifies the first pixels of the default view against a bit-exact reference.

External hardware

- An RP2040 (the TinyTapeout demo board's microcontroller) to bridge the parallel pixel bus + view-packet bus to USB
- A host running a small viewer (pygame) to display the stream and compute view packets

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	view-packet data 0	pixel value 0	pix_valid (out)
1	view-packet data 1	pixel value 1	frame_start (out)
2	view-packet data 2	pixel value 2	line_end (out)
3	view-packet data 3	pixel value 3	param_valid (in)
4	view-packet data 4	pixel value 4	param_frame (in)
5	view-packet data 5	pixel value 5	—
6	view-packet data 6	pixel value 6	—
7	view-packet data 7	pixel value 7	—

fibbonaci_tt

by **Jake Sabree**

0391 HDL Project

`github.com/wiredlab/sabree-fibonacci_tt`

"Checks if a user input number is in the fibbonaci sequence"

How it works

The design checks whether the 8-bit input value on `ui_in[7:0]` is part of the Fibonacci sequence. It compares the input against the Fibonacci numbers that fit in 8 bits: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, and 233. If the input matches one of these values, `uo_out[0]` is set to 1. Otherwise, `uo_out[0]` is set to 0.

How to test

The cocotb testbench applies known Fibonacci and non-Fibonacci values to `ui_in[7:0]`. Values such as 2, 3, 5, and 8 should produce a 1 on `uo_out[0]`, while values such as 4, 6, and 7 should produce a 0. The test passes when the output matches the expected boolean 1/0 result for every case

External hardware

No external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input bit 0	1 if input is Fibbonaci number	—
1	Input bit 1	—	—
2	Input bit 2	—	—
3	Input bit 3	—	—
4	Input bit 4	—	—
5	Input bit 5	—	—
6	Input bit 6	—	—
7	Input bit 7	—	—

Simple Sprinkler

by **Noah Perez**

0449 Wokwi Project

github.com/noahzperez29/Noah_Binary_Hex_Decoder

wokwi.com/projects/466666882406199297

"I designed a Binary to Hexidecimal Decoder using a seven-segment LED display that takes inputs A - D and outputs a hexidecimal value between 1 - F."

How it works

The decoder takes a 4-bit binary input from the user and translates it to hexidecimal. This is shown via the seven segment display. I developed combinational logic, controlling each of the 7 segments and sectioned them for easy accessibility and troubleshooting. I color coded the truth table and K-maps to match that of the wires on the Wokwi project.

How to test

Toggle inputs A - D (1 - 4) to change the state of the seven segment display which displays a number 0 - 9 or a letter A - F.

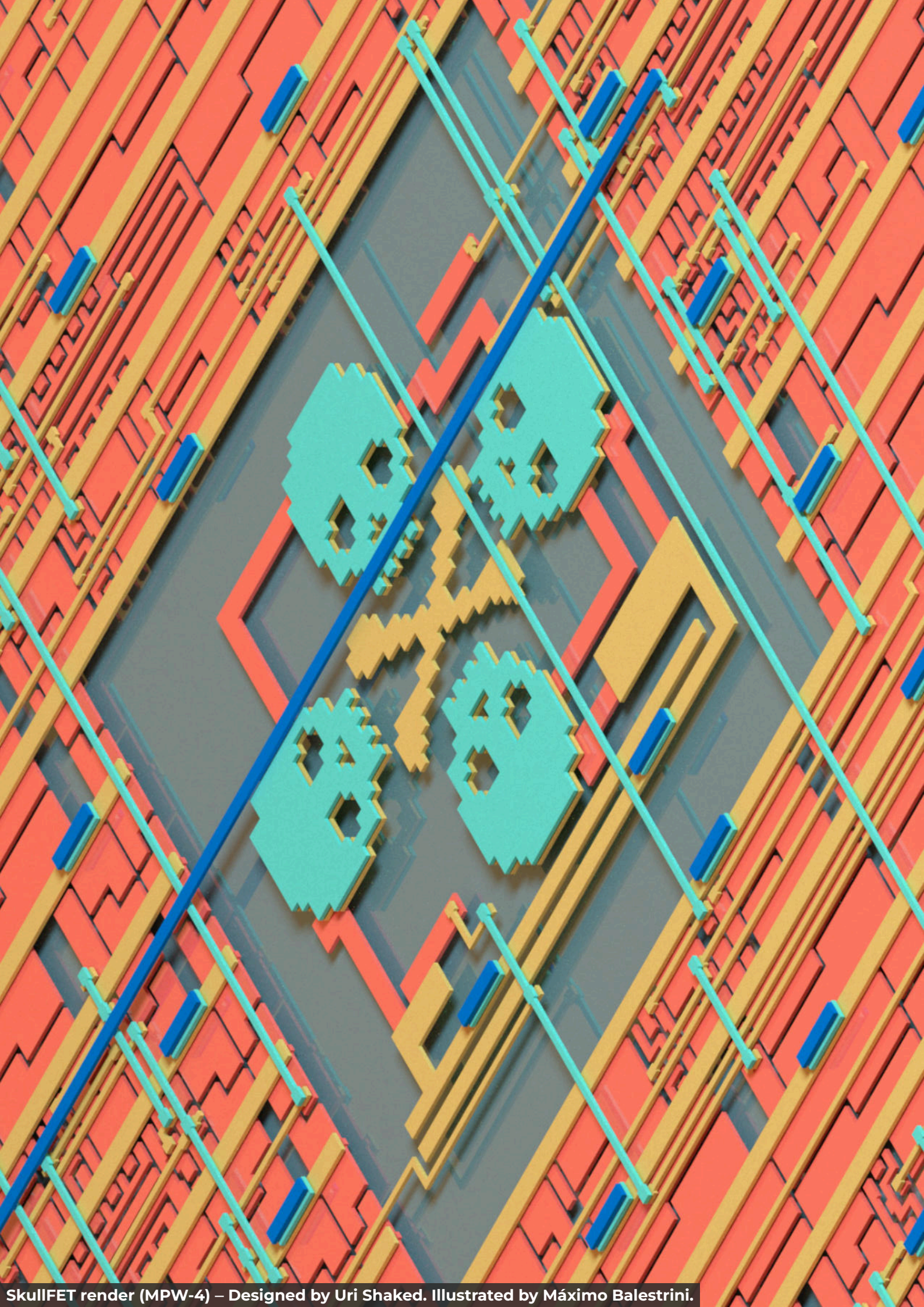
External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A	A	—
1	B	B	—
2	C	C	—
3	D	D	—
4	—	E	—
5	—	F	—
6	—	G	—
7	—	—	—



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

Programmable Waveform and PWM Generator

by **silicon_nomad**

450 | 10 MHz | HDL Project

github.com/silicon-nomad/tt-programmble-tiny-pattern-gen

"UART-programmable waveform and PWM generator. Send samples over UART, stored in 256-byte RAM, played back as 8-bit parallel DAC output or 8 independent PWM channels."

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	UART_RX	DAC_D0 / PWM_CH0	DAC_WR
1	MODE	DAC_D1 / PWM_CH1	—
2	LOAD	DAC_D2 / PWM_CH2	—
3	—	DAC_D3 / PWM_CH3	—
4	—	DAC_D4 / PWM_CH4	—
5	—	DAC_D5 / PWM_CH5	—
6	—	DAC_D6 / PWM_CH6	—
7	—	DAC_D7 / PWM_CH7	—

Universal Binary to Segment Decoder

by **Rebecca G. Bettencourt**

0451 HDL Project

`github.com/RebeccaRGB/ttgf-ubcd`

"*Decodes various binary codes to various segmented displays.*"

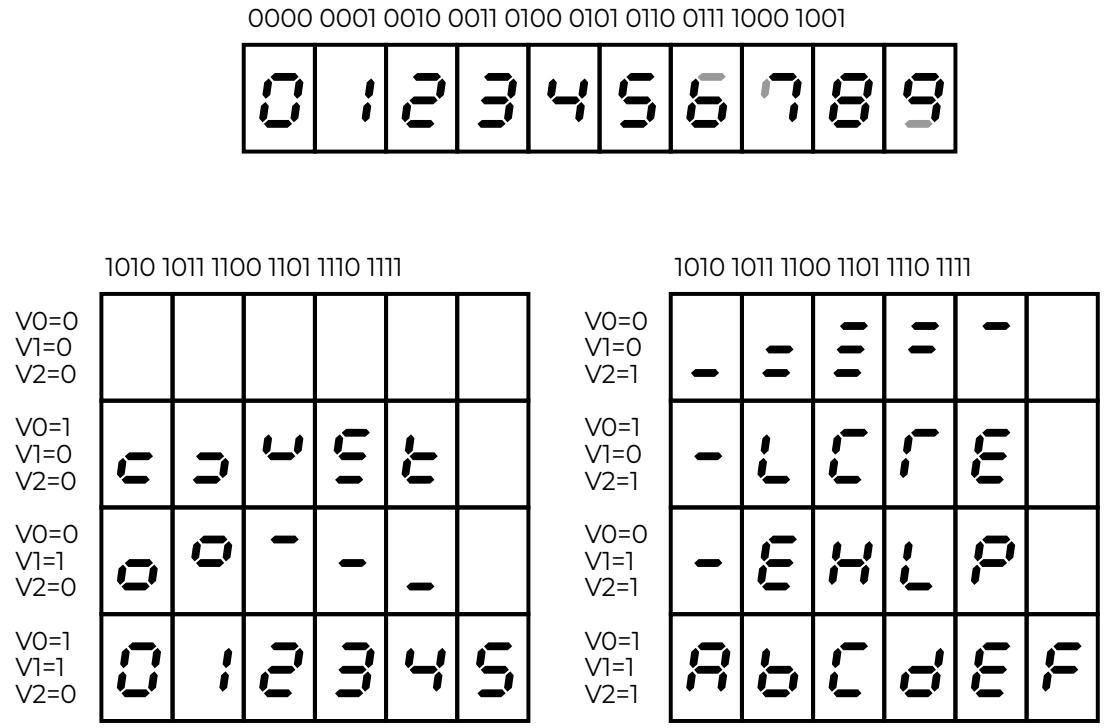
How it works

This project is composed of four modules:

- A BCD to seven segment decoder with a wide variety of options for customizing the appearance of digits
- An ASCII to seven segment decoder with two different "fonts"
- A dual BCD to Cistercian numeral decoder
- A BCV (binary-coded *vigesimal*) to Kaktovik numeral decoder

BCD to seven segment decoder

This mode converts a decimal digit in BCD to its representation on a standard seven segment display. There are inputs that affect the display of the digits 6, 7, and 9, and eight different options for handling out-of-range values. These inputs allow this decoder to match the behavior of just about any other BCD to seven segment decoder, making it *universal*.



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCD value is zero and **/RBI** is LOW, all segments will be blank.
- **V0**, **V1**, **V2** - Selects the output when the BCD value is out of range.
- **X6** - When HIGH, the extra segment **a** will be lit on the digit 6.
- **X7** - When HIGH, the extra segment **f** will be lit on the digit 7.
- **X9** - When HIGH, the extra segment **d** will be lit on the digit 9.
- **A**, **B**, **C**, **D** - BCD input from least significant bit **A** to most significant bit **D**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g** - Outputs for a seven segment display.
- **/RBO** - Ripple blanking output. HIGH when BCD value is nonzero or **/RBI** is HIGH.

The pin assignments in this mode are:

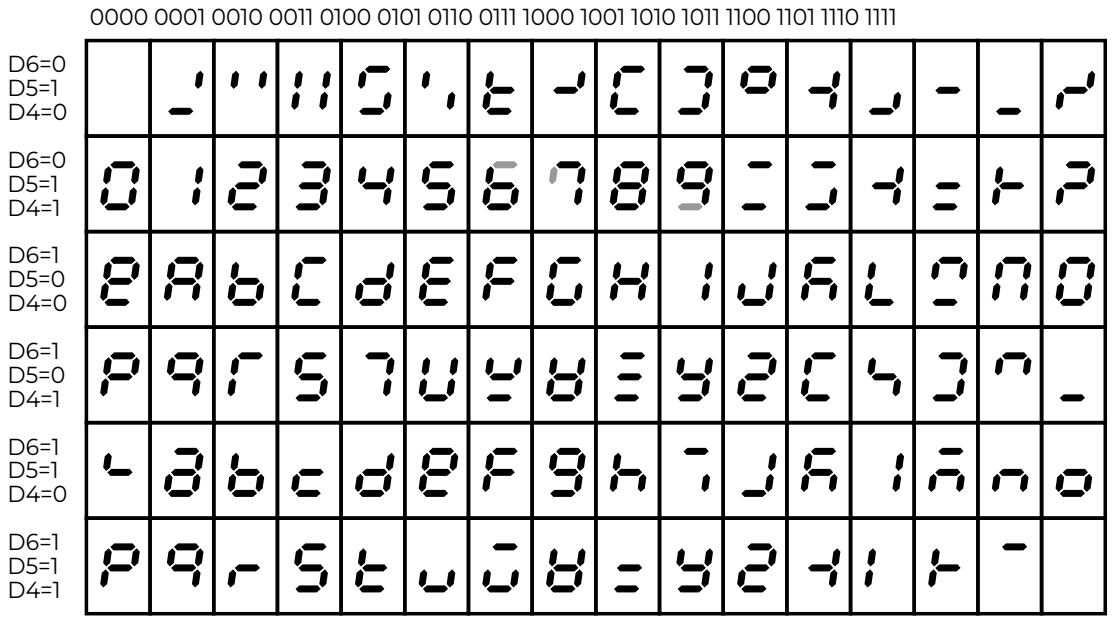
—	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Input - X6
1	B	Segment b	Input - X7

2	C	Segment c	Input - X9
3	D	Segment d	Input - /LT
4	V0	Segment e	Input - /BI
5	V1	Segment f	Input - /AL
6	V2	Segment g	Input - LOW
7	/RBI	/RBO	Input - LOW

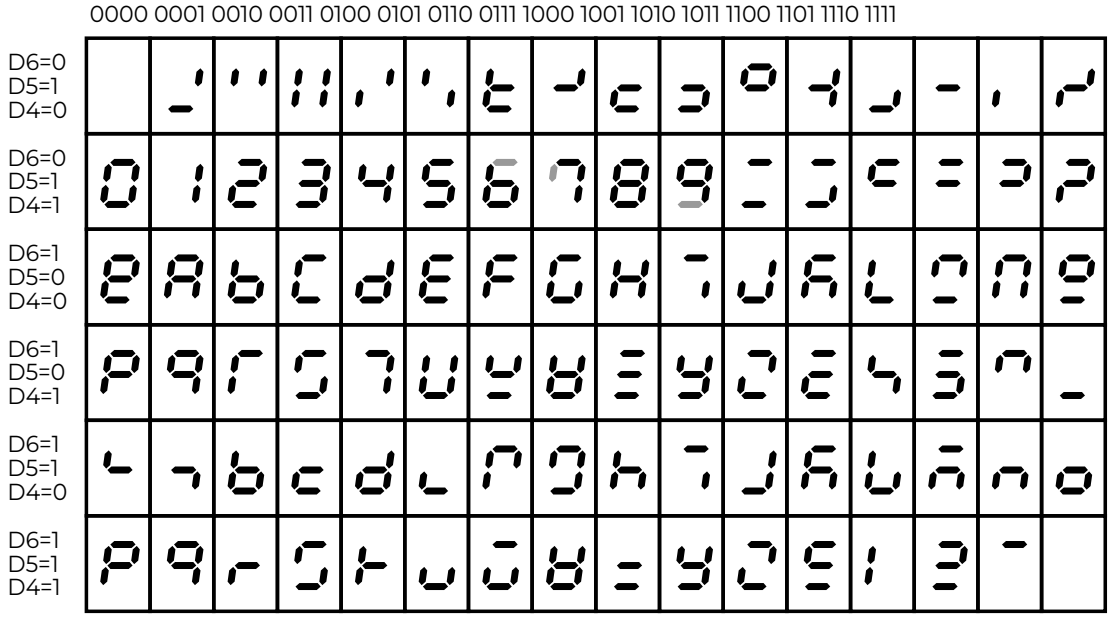
ASCII to seven segment decoder

This mode converts an ASCII character to a representation on a standard seven segment display. Like with the BCD decoder, there are inputs that affect the display of the digits 6, 7, and 9. There are also two choices of “font” and the option to display lowercase letters as uppercase or as lowercase.

FS=0:



FS=1:



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs.
- **FS** - Font select. Selects one of two “fonts.”
- **LC** - Lower case. If LOW, lowercase letters will appear as uppercase.
- **X6** - When HIGH, the extra segment **a** will be lit on the digit 6.
- **X7** - When HIGH, the extra segment **f** will be lit on the digit 7.
- **X9** - When HIGH, the extra segment **d** will be lit on the digit 9.
- **D0**...**D6** - ASCII input from least significant bit **D0** to most significant bit **D6**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g** - Outputs for a seven segment display.
- **/LTR** - Letter. LOW when the input is a letter (A...Z or a...z).

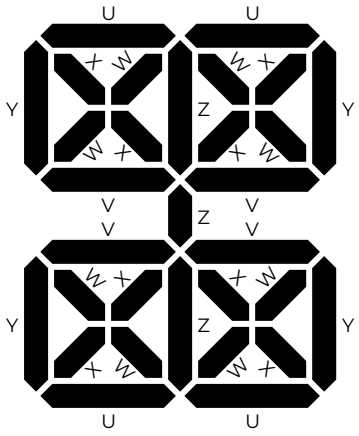
The pin assignments in this mode are:

—	Dedicated Input	Dedicated Output	Bidirectional
0	D0	Segment a	Input - X6
1	D1	Segment b	Input - X7
2	D2	Segment c	Input - X9
3	D3	Segment d	Input - FS
4	D4	Segment e	Input - /BI

5	D5	Segment f	Input - /AL
6	D6	Segment g	Input - HIGH
7	LC	/LTR	Input - LOW

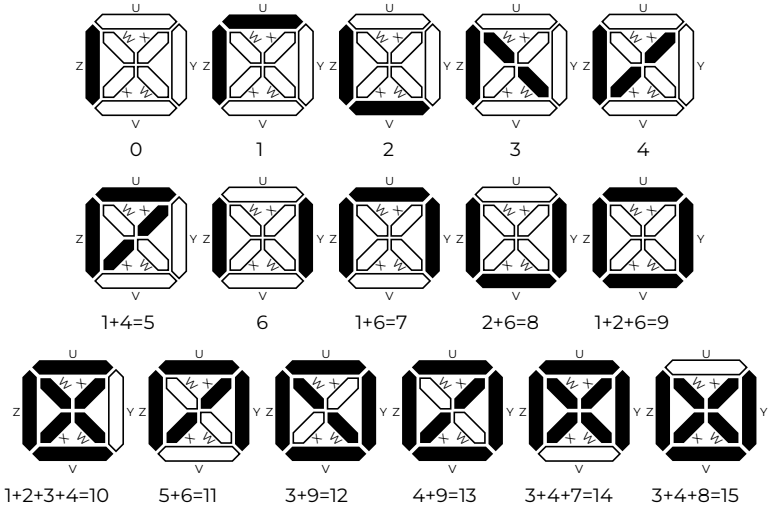
Dual BCD to Cistercian numeral decoder

This mode converts two decimal digits in BCD to their representations on the segmented display for Cistercian numerals shown below.



The patterns produced for each input value are shown below.

Patterns as seen in top right (units) position:



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.

- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT1** and **/LT2**.
- **/LT1** - Lamp test for digit 1. When **/BI** is HIGH and **/LT1** is LOW, all segments for digit 1 will be lit.
- **/LT2** - Lamp test for digit 2. When **/BI** is HIGH and **/LT2** is LOW, all segments for digit 2 will be lit.
- **A1**, **B1**, **C1**, **D1** - BCD input for digit 1 from least significant bit **A1** to most significant bit **D1**.
- **A2**, **B2**, **C2**, **D2** - BCD input for digit 2 from least significant bit **A2** to most significant bit **D2**.
- **U1**, **V1**, **W1**, **X1**, **Y1** - Outputs for digit 1 on a Cistercian segmented display.
- **U2**, **V2**, **W2**, **X2**, **Y2** - Outputs for digit 2 on a Cistercian segmented display.

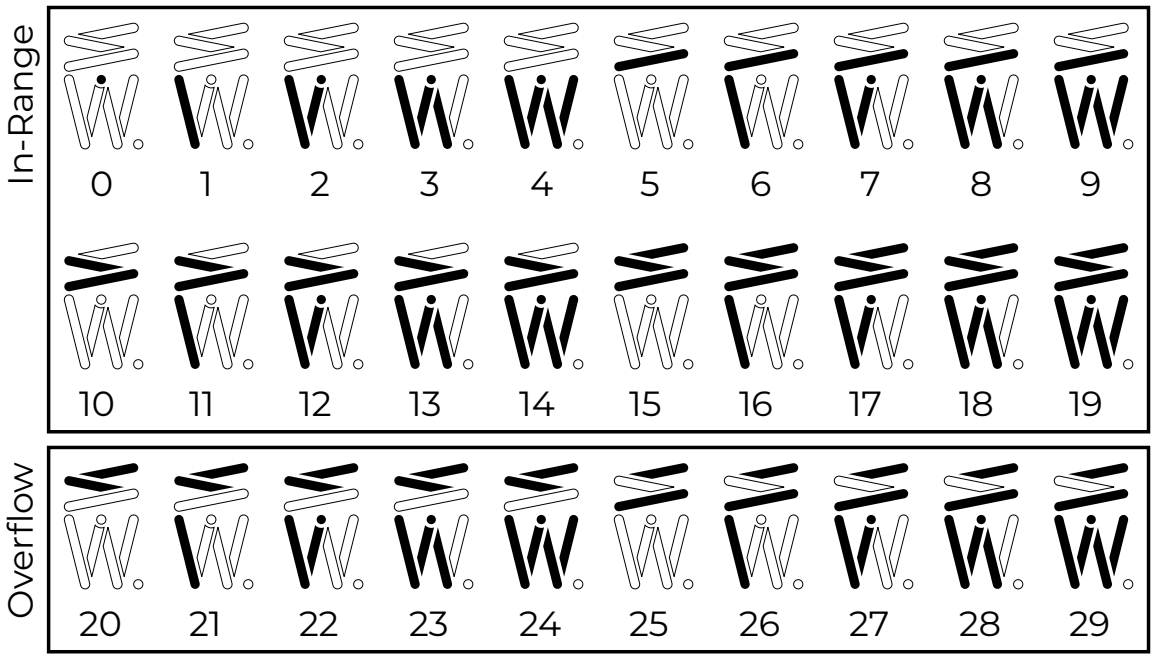
The pin assignments in this mode are:

—	Dedicated Input	Dedicated Output	Bidirectional
0	A1	Segment U1	Output - Y1
1	B1	Segment U2	Output - Y2
2	C1	Segment V1	Input - /LT1
3	D1	Segment V2	Input - /LT2
4	A2	Segment W1	Input - /BI
5	B2	Segment W2	Input - /AL
6	C2	Segment X1	Input - LOW
7	D2	Segment X2	Input - HIGH

BCV to Kaktovik numeral decoder

This mode converts a *vigesimal* (base 20) digit in BCV (binary-coded vigesimal) to its representation on the segmented display for Kaktovik numerals shown below.

The patterns produced for each input value are shown below.



The signals used in this mode are:

- **/AL** - Active low. If HIGH, outputs will be HIGH when lit. If LOW, outputs will be LOW when lit.
- **/BI** - Blanking input. If LOW, all segments will be blank regardless of other inputs, including **/LT**.
- **/LT** - Lamp test. When **/BI** is HIGH and **/LT** is LOW, all segments will be lit.
- **/RBI** - Ripple blanking input. If the BCV value is zero and **/RBI** is LOW, all segments will be blank.
- **/VBI** - Overflow blanking input. If the BCV value is out of range and **/VBI** is LOW, all segments will be blank.
- **A**, **B**, **C**, **D**, **E** - BCV input from least significant bit **A** to most significant bit **E**.
- **a**, **b**, **c**, **d**, **e**, **f**, **g**, **h** - Outputs for a Kaktovik segmented display.
- **/RBO** - Ripple blanking output. HIGH when BCV value is nonzero or **/RBI** is HIGH.
- **V** - Overflow. HIGH when BCV value is out of range (greater than or equal to 20).

The pin assignments in this mode are:

—	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Output - h
1	B	Segment b	Output - V
2	C	Segment c	—
3	D	Segment d	Input - /LT

4	E	Segment e	Input - /BI
5	—	Segment f	Input - /AL
6	/VBI	Segment g	Input - HIGH
7	/RBI	/RBO	Input - HIGH

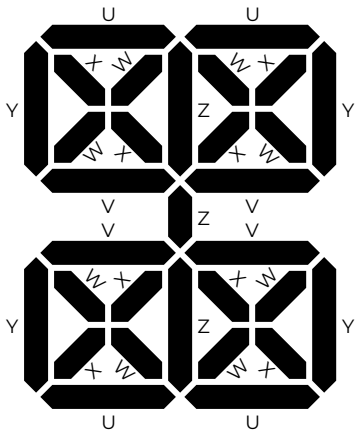
How to test

The `test` directory includes extensive tests for each of the four modules.

External hardware

For the BCD and ASCII modes, a standard seven-segment display is used.

For the Cistercian mode, a segmented display like the one below is used. There are design files for such a display here.



For the Kaktovik mode, a segmented display like the one below is used. There are design files for such a display here.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A; D0; A1; A	Segment a; U1; a	X6; X6; Y1; h
1	B; D1; B1; B	Segment b; U2; b	X7; X7; Y2; V
2	C; D2; C1; C	Segment c; V1; c	X9; X9; /LT1; -
3	D; D3; D1; D	Segment d; V2; d	/LT; FS; /LT2; /LT
4	V0; D4; A2; E	Segment e; W1; e	/BI (blanking input)
5	V1; D5; B2; -	Segment f; W2; f	/AL (active low)
6	V2; D6; C2; /VBI	Segment g; X1; g	M0 (mode select)
7	/RBI; LC; D2; /RBI	/RBO; /LTR; X2; /RBO	M1 (mode select)

Hardware UTF Encoder/Decoder

by **Rebecca G. Bettencourt**

0453 HDL Project

`github.com/RebeccaRGB/ttgf-hardware-utf8`

"Converts Unicode code points between UTF-8, UTF-16, and UTF-32."

How it works

This project contains hardware logic to convert between the UTF-8, UTF-16, and UTF-32 encodings for Unicode text.

It will detect and raise an error signal on overlong encodings, out of range code point values, and invalid byte sequences.

(You can optionally disable range checking if you wish to use the original UTF-8 spec that supports values up to 0x7FFFFFFF.)

Basic operation

- In the initial state, all dedicated inputs should be set HIGH.
- At any time, set /RESET (rst_n) LOW and pulse CLK to reset all inputs and outputs to initial state.
- At any time, set /ROUT (input 0) LOW and pulse CLK to seek to the beginning of the output.
- You can set ERRS or /PROPS (input 1) HIGH to get an error status on the dedicated outputs.
- You can set ERRS or /PROPS (input 1) LOW to get character properties on the dedicated outputs.
- You can set CHK (input 2) HIGH to raise an error signal when the code point value is out of range (≥0x110000).
- You can set CHK (input 2) LOW to ignore out of range code point values and encode/decode values up to 0x7FFFFFFF.
- You can set CBE (input 3) HIGH to specify big endian order for UTF-32 and UTF-16 input and output.
- You can set CBE (input 3) LOW to specify little endian order for UTF-32 and UTF-16 input and output.

Inputting UTF-32

1. Set READ or /WRITE (input 4) LOW.
2. Set /CIO (input 5, character I/O) LOW.
3. Set bidirectional I/O to the first byte of the UTF-32 word and pulse CLK.

4. Set bidirectional I/O to the second byte of the UTF-32 word and pulse CLK.
5. Set bidirectional I/O to the third byte of the UTF-32 word and pulse CLK.
6. Set bidirectional I/O to the fourth byte of the UTF-32 word and pulse CLK.
7. Set /CIO (input 5, character I/O) HIGH.
8. Set READ or /WRITE (input 4) HIGH.
9. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
10. If READY (output 0) is LOW or ERROR (output 5) is HIGH, the input was out of range (≥0x110000 or, if CHK is LOW, ≥0x80000000).

Inputting UTF-16

1. Set ERRS or /PROPS (input 1) LOW.
2. Set READ or /WRITE (input 4) LOW.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Set bidirectional I/O to the first byte of the first UTF-16 word and pulse CLK.
5. Set bidirectional I/O to the second byte of the first UTF-16 word and pulse CLK.
6. If HIGHCHAR (output 3) is LOW, skip to step 9.
7. Set bidirectional I/O to the first byte of the second UTF-16 word and pulse CLK.
8. Set bidirectional I/O to the second byte of the second UTF-16 word and pulse CLK.
9. Set /UIO (input 6, UTF-16 I/O) HIGH.
10. Set READ or /WRITE (input 4) HIGH.
11. Set ERRS or /PROPS (input 1) HIGH.
12. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.
13. If RETRY (output 1) is HIGH, the first word was a high surrogate but the second word was not a low surrogate. The output will be the high surrogate only; the last word will need to be processed again.

Inputting UTF-8

1. Set READ or /WRITE (input 4) LOW.
2. Set /BIO (input 7, byte I/O) LOW.
3. Set bidirectional I/O to the current byte of the UTF-8 sequence and pulse CLK.
4. Repeat step 3 until READY (output 0) or ERROR (output 5) is HIGH.
5. If READY (output 0) is HIGH and ERROR (output 5) is LOW, the input and output are both valid.

6. If RETRY (output 1) is HIGH, the UTF-8 sequence was truncated (not enough continuation bytes). The output will be the truncated sequence only; the last byte will need to be processed again.
7. If INVALID (output 2) is HIGH, the UTF-8 sequence was a single continuation byte or invalid byte (0xFE or 0xFF).
8. If OVERLONG (output 3) is HIGH, the UTF-8 sequence was an overlong encoding.
9. If NONUNI (output 4) is HIGH, the UTF-8 sequence was out of range (≥0x110000).

Outputting UTF-32

1. Set READ or /WRITE (input 4) HIGH.
2. Set /CIO (input 5, character I/O) LOW.
3. Pulse CLK and read the first byte of the UTF-32 word from the bidirectional I/O.
4. Pulse CLK and read the second byte of the UTF-32 word from the bidirectional I/O.
5. Pulse CLK and read the third byte of the UTF-32 word from the bidirectional I/O.
6. Pulse CLK and read the fourth byte of the UTF-32 word from the bidirectional I/O.
7. Set /CIO (input 5, character I/O) HIGH.
8. If the UTF-32 word is within range, the input and output are both valid.
9. If the UTF-32 word is not within range, then the input was either incomplete or invalid.

Outputting UTF-16

1. Set READ or /WRITE (input 4) HIGH.
2. If UEOF (output 6) is HIGH, then the input was either incomplete or invalid.
3. Set /UIO (input 6, UTF-16 I/O) LOW.
4. Pulse CLK and read the next byte of the UTF-16 sequence from the bidirectional I/O.
5. Repeat step 4 until UEOF (output 6) is HIGH.
6. Set /UIO (input 6, UTF-16 I/O) HIGH.

Outputting UTF-8

1. Set READ or /WRITE (input 4) HIGH.
2. If BEOF (output 7) is HIGH, then the input was either incomplete or invalid.
3. Set /BIO (input 7, byte I/O) LOW.

4. Pulse CLK and read the next byte of the UTF-8 sequence from the bidirectional I/O.
5. Repeat step 4 until BEOF (output 7) is HIGH.
6. Set /BIO (input 7, byte I/O) HIGH.

Error status

When ERRS or /PROPS (input 1) is HIGH, the dedicated outputs will be:

#	Name	Meaning
0	READY	The input and output are complete sequences.
1	RETRY	The previous input was invalid or the start of another sequence and was ignored. Process the output, reset, and try the previous input again.
2	INVALID	The input and output are invalid.
3	OVERLONG	The UTF-8 input was an overlong sequence.
4	NONUNI	The code point value is out of range (≥0x110000). (This is set independently of the CHK input; the CHK input only changes whether this counts as an error.)
5	ERROR	Equivalent to (RETRY or INVALID or OVERLONG or (NONUNI and CHK)).

If all of these outputs are LOW, the accumulated input is incomplete and more input is required (underflow).

Character properties

When ERRS or /PROPS (input 1) is LOW, the dedicated outputs will be:

#	Name	Meaning
0	NORMAL	The code point value is valid and not a C0 or C1 control character, surrogate, private use character, or noncharacter.
1	CONTROL	The code point value is valid and a C0 or C1 control character (0x00-0x1F or 0x7F-0x9F).
2	SURROGATE	The code point value is valid and a UTF-16 surrogate (0xD800-0xDFFF).
3	HIGHCHAR	The code point value is valid and either a high surrogate (0xD800-0xDBFF) or a non-BMP character (≥0x10000).

4	PRIVATE	The code point value is valid and either a private use character (0xE000-0xF8FF, ≥0xF0000) or the high surrogate of a private use character (0xDB80-0xDBFF).
5	NONCHAR	The code point value is valid and a noncharacter (0xFDD0-0xFDEF or the last two code points of any plane).

If all of these outputs are LOW, there is no valid code point in the output.

How to test

The `test.py` file covers a comprehensive set of test cases which are listed in a separate file to avoid bloating the TT09 manual.

External hardware

Any device that needs to process Unicode text.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	/ROUT	READY; NORMAL	I/O LSB
1	ERRS, /PROPS	RETRY; CONTROL	I/O
2	CHK	INVALID; SURROGATE	I/O
3	CBE, /CLE	OVERLONG; HIGHCHAR	I/O
4	READ, /WRITE	NONUNI; PRIVATE	I/O
5	/CIO	ERROR; NONCHAR	I/O
6	/UIO	UEOF	I/O
7	/BIO	BEOF	I/O MSB

Spiking Neural Network WTA Inference Engine (GF180)

by **Prof. Santhosh Sivasubramani** & **IIT Delhi**

454 | 25 MHz | HDL Project

github.com/ragansanthosh/tt_um_santhosh_snn_wta_gf_pub

> "*GF180mcuD digital spiking neural network inference engine: 4 LIF neurons with programmable 4x4 signed synaptic weight matrix, per-neuron bias, programmable global threshold/leak/refractory, hard winner-take-all lateral inhibition, optional self-recurrence, SPI-configurable 16-bit register file, 4-channel 2-bit input current coding, 4 spike outputs, 2-bit winner ID output, any-spike and busy flags; deterministic single-cycle neuron update*"

A 4-neuron digital Spiking Neural Network (SNN) inference engine with hard Winner-Take-All lateral inhibition for TinyTapeout GF 26a (1×2 tile).

Each clock tick (rate set by an 8-bit tick divider) every non-refractory neuron accumulates a MAC of its bias, the dot product of the 4-channel 2-bit input current vector with its row of the 4×4 signed weight matrix, and an optional self-recurrence term. The resulting increment is added to the 16-bit signed membrane (saturating on overflow). If the updated membrane crosses the global threshold the neuron fires a 1-tick spike and resets; if `hard_wta=1`, the lowest-index firing neuron is the sole winner and all other firing neurons are forced into the `refractory_cnt` inhibition window for `refractory` cycles.

All datapath parameters (threshold, leak, refractory, tick_div, bias, weights, recur_gain) are programmable over the SPI register file. Membrane and refractory counters are memory-mapped for debug.

How to test

1. **Reset** (`rst_n=0` ≥ 10 clocks).
2. Program via SPI:
   - threshold, leak, refractory, tick_div, recur_gain
   - per-neuron bias[0..3]
   - weight matrix w[0..15]
3. Write `ctrl = {recur_en, hard_wta, 0, global_en=1, irq_en}` to start.
4. Drive `ui_in[7:0]` with the 4×2-bit input current vector.
5. Observe spikes on `uo_out[3:0]` and winner ID on `uo_out[5:4]`; `uio_out[5]` pulses every tick for oscilloscope triggering.
6. Poll `status (0x20)` for latest spike vector + sticky overflow flag.

External hardware

- An SPI master (any Raspberry Pi / microcontroller) for register programming.
- A 4×2-bit input pattern generator on `ui_in[7:0]` — could be raw GPIO, a small logic-analyser pattern, or an external event source.
- Oscilloscope or logic analyser on `uo_out` / `uio_out` for spike capture.

Register summary

Addr	Name
0x00	ctrl
0x01	threshold
0x02	leak
0x03	refractory
0x04	recur_gain
0x05	tick_div
0x08..0x0B	bias[0..3]
0x10..0x1F	w[0..15]
0x20	status
0x30..0x37	membrane lo/hi (read-only)
0x38..0x3B	refractory_cnt (read-only)

ctrl bits

Bit	Name	Meaning
0	irq_en	gate `uio[7]` IRQ pulse on any spike
1	global_en	master enable (also drives `busy` output)
2	learn_frozen	reserved for future on-line STDP
3	hard_wta	1 = lateral inhibition of losers; 0 = soft WTA
4	recur_en	1 = add `recur_gain` when `fired_prev[i]=1`

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in_cur0[0]	spike[0]	spi_cs_n
1	in_cur0[1]	spike[1]	spi_mosi
2	in_cur1[0]	spike[2]	spi_miso
3	in_cur1[1]	spike[3]	spi_sck
4	in_cur2[0]	winner_id[0]	overflow_ever

#	Input	Output	Bidirectional
5	in_cur2[1]	winner_id[1]	tick
6	in_cur3[0]	busy	winner_valid
7	in_cur3[1]	any_spike	irq

SPI Master Slave Communication

by **Nevin Philip**

0455 HDL Project

`github.com/nev-1910/SPI`

> "*8-bit Full Duplex SPI Communication using Master and Slave*"

How it works

This project implements an 8-bit SPI Master Controller using Verilog.

The design accepts 8-bit parallel data through `ui_in[7:0]`. When the START signal (`uio_in[0]`) is asserted, the data is loaded into an internal shift register.

A finite state machine (FSM) controls the transmission process using four states:

- IDLE
- LOAD
- TRANSFER
- DONE

The shift register transmits data serially through the MOSI line while generating the SPI clock (SCLK). A chip select (CS) signal is used to indicate active communication. A BUSY signal indicates that transmission is in progress.

How to test

1. Apply reset by driving `rst_n = 0`.
2. Load an 8-bit value on `ui_in[7:0]`.
3. Release reset by setting `rst_n = 1`.
4. Assert the START signal using `uio_in[0]`.
5. Observe MOSI output on `uo_out[0]`.
6. Observe SPI clock on `uo_out[1]`.
7. Observe chip select on `uo_out[2]`.
8. Observe busy status on `uo_out[3]`.
9. After 8 bits are transmitted, the FSM returns to IDLE.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	START	MOSI	—
1	—	MISO	—

#	Input	Output	Bidirectional
2	—	SCLK	—
3	—	CS	—
4	—	DONE	—
5	—	MASTER_RX0	—
6	—	SLAVE_RX0	—
7	—	—	—

Configurable 8-bit PWM Generator

by **Vinuta**

`480` `50 MHz` `HDL Project`

`github.com/Cambridge-CIT-ASIC-Tapeouts/PWM`

> "*Generates a variable duty cycle PWM signal using an 8-bit input.*"

Author

Raksha S S

Description

This project implements an 8-bit PWM (Pulse Width Modulation) generator using Verilog. The PWM duty cycle is controlled using the 8-bit input `ui_in`.

How it works

An internal 8-bit counter continuously increments with every clock cycle.

The counter value is compared with the input duty cycle value:

- If counter < ui_in → output is HIGH
- Else → output is LOW

This generates a PWM waveform on `uo_out[0]`.

How to test

1. Apply clock signal.
2. Release reset (`rst_n = 1`).
3. Give different values to `ui_in`.
4. Observe PWM waveform at `uo_out[0]`.

Examples:

- `ui_in = 64` → 25% duty cycle
- `ui_in = 128` → 50% duty cycle
- `ui_in = 192` → 75% duty cycle

External hardware

No external hardware required.

Pinout

Pin	Direction	Description

ui_in[7:0]	Input	PWM duty cycle input
uo_out[0]	Output	PWM output signal
clk	Input	Clock signal
rst_n	Input	Active-low reset

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Duty Cycle Bit 0	PWM Output	Unused
1	Duty Cycle Bit 1	Unused	Unused
2	Duty Cycle Bit 2	Unused	Unused
3	Duty Cycle Bit 3	Unused	Unused
4	Duty Cycle Bit 4	Unused	Unused
5	Duty Cycle Bit 5	Unused	Unused
6	Duty Cycle Bit 6	Unused	Unused
7	Duty Cycle Bit 7	Unused	Unused

Secure V2X Mini Demonstrator

by **Pushkar Kulkarni**

0481 | 50 MHz | HDL Project

github.com/push-777/Sanpush_ALU

"8-bit LFSR based data masking for secure V2X communication"

How it works

This project implements a Secure V2X (Vehicle-to-Everything) Mini Demonstrator using an 8-bit Linear Feedback Shift Register (LFSR).

The design accepts an 8-bit plaintext input through the ui_in pins. An 8-bit pseudo-random key stream is generated internally using the LFSR. The plaintext data is XORed with the generated key stream to produce masked output data.

The LFSR updates on every rising edge of the clock and continuously generates a changing sequence. This demonstrates a simple hardware-based data masking technique that can be used in secure communication systems.

Input:

- ui_in[7:0] : 8-bit plaintext data

Output:

- uo_out[7:0] : 8-bit masked data

The bidirectional pins are not used in this design.

How to test

1. Apply an 8-bit value to ui_in.
2. Provide a clock signal to clk.
3. Release reset by setting rst_n high.
4. Observe the generated masked output on uo_out.
5. Change the input data and continue clocking the design.
6. Verify that the output changes according to the XOR operation between the plaintext input and the LFSR-generated key stream.

Expected Result:

- The output should be different from the input due to data masking.
- The output pattern changes as the LFSR state changes.

External hardware

No external hardware is required for this project.

Project Pinout

Digital Pins

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

Digital Door Lock

by **Rameshwar**

482 HDL Project

`github.com/Rameshwar-12/Door_lock_system`

"*Digital door lock system using password verification.*"

How it works

This project implements a simple Digital Door Lock System.

The user enters a 4-bit password through the input pins.

Stored Password: 1010

Operation:

- If the entered password matches 1010, the lock is unlocked.
- If the entered password is incorrect, an error signal is generated and the lock remains locked.

Modules Used:

1. password_checker
   - Compares entered password with stored password.
   - Generates match signal.
2. lock_controller
   - Controls unlock and error outputs based on match signal.
3. digital_door_lock
   - Top module integrating all submodules.

How to test

Test Case 1: Input Password = 1010 Expected Output: unlock = 1 error = 0

Test Case 2: Input Password = 1111 Expected Output: unlock = 0 error = 1

Testbench file: test/tb.v

Simulation: iverilog -o sim src/project.v test/tb.v vvp sim

External hardware

No external hardware is required.

Inputs:

- 4-bit password

Outputs:

- unlock signal
- error signal

Project Pinout

Digital Pins

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

AES S-Box Accelerator

by **Santosh**

0483 HDL Project

`github.com/cambridgeinstitutetechnologyblr-santosh/sanpush_aes`

"Design and Physical Implementation of AES S-Box Accelerator using SKY130 OpenLane Flow"

How it works

This project implements an AES (Advanced Encryption Standard) S-Box accelerator using Verilog HDL.

The AES S-Box is a nonlinear substitution block used in the SubBytes stage of AES encryption. It takes an 8-bit input byte and produces a corresponding 8-bit substituted output byte according to the AES substitution table.

In this design:

- `ui_in[7:0]` is the AES input byte.
- `uo_out[7:0]` is the AES S-Box output byte.
- The design is implemented as a combinational lookup table.
- No clocked logic is required.
- Unused bidirectional I/O pins are tied to zero.

Example:

Input (Hex)	Output (Hex)
00	63
01	7C
02	77
03	7B

The design demonstrates a hardware implementation of a cryptographic primitive that is widely used in secure communication systems, cybersecurity applications, and embedded security hardware.

How to test

Apply an 8-bit value to `ui_in[7:0]`.

Observe the corresponding AES S-Box output on `uo_out[7:0]`.

Example test vectors:

Input	Expected Output
0x00	0x63
0x01	0x7C
0x02	0x77
0x03	0x7B
0x04	0xF2

The supplied Cocotb testbench automatically verifies these mappings.

External hardware

No external hardware is required.

The design is fully self-contained and can be simulated using the Tiny Tape-out verification flow.

Project Pinout

Digital Pins

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

8-bit LFSR Circuit

by **Satya Roop Bankuru**

484 | 50 MHz | HDL Project

`github.com/CambridgeinstitutetechnologyBLR-satya/lfsrvenkys`

"*Sequential PRNG block using Verilog.*"

Designed for digital VLSI training module parameters.

Core Topology

This design generates pseudo-random sequences using an active feedback polynomial shift mechanism:

$$x^8 + x^6 + x^5 + x^4 + 1$$

Validation Guide

1. Hold `rst_n` low to inject the `0x01` processing seed.
2. Assert `rst_n` high to begin random cycling.
3. Observe the generated unique sequences change across the 8-bit `uo_out` pins on each clock cycle edge.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	unused	out0	unused
1	unused	out1	unused
2	unused	out2	unused
3	unused	out3	unused
4	unused	out4	unused
5	unused	out5	unused
6	unused	out6	unused
7	unused	out7	unused

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

Hardware Anomaly Detection

by **Vinayak**

`0485` `50 MHz` `HDL Project`

`github.com/vinayakalk25/Hardware_abnormal_detection`

"ML-based V2X hardware anomaly detection engine."

How it works

This project implements a lightweight, Machine Learning-based Hardware Anomaly Detection engine designed to analyze incoming V2X (Vehicle-to-Everything) packet data for malicious or anomalous behavior in real time. The architecture is deeply pipelined and operates in six stages:

1. **SIPO Ingestion:** Incoming data is fed serially (LSB first) into a 64-bit shift register via the `ui_in[0]` pin, controlled by the `ui_in[1]` `bit_valid` strobe.
2. **Feature Extraction:** Once 64 bits are captured, the core extracts two key 8-bit signed features: **Velocity** (bits 31:24) and **Heading** (bits 23:16).
3. **DMA Flow Control:** Extracted features are passed into a pipeline register to decouple the serial ingestion from the math core.
4. **ML Inference Core:** A highly optimized MAC (Multiply-Accumulate) unit applies hardcoded model weights to the inputs. It computes: `(12 * Velocity) + (88 * Heading)`.
5. **Anomaly Scoring:** The MAC result passes through a ReLU activation function (negative values are clamped to 0) and is evaluated against predefined thresholds:
   - **< 1024:** SAFE (`0x00`)
   - **>= 1024:** CAUTION (`0x55`)
   - **>= 2048:** WARNING (`0xAA`)
   - **>= 4096:** CRITICAL / ATTACK (`0xFF`)
6. **Output Endpoints:** The 8-bit threat score is output on `uo_out`. A 1-cycle `done_flag` pulses on `uio_out[0]`, and if the threat is critical, a hardware interrupt `irq_flag` pulses on `uio_out[1]`.

How to test

To test the anomaly detection engine, you will need to simulate a serial data feed representing a 64-bit V2X packet:

1. **Reset the Core:** Assert `rst_n` (active low) to clear all internal shift registers and pipelines.
2. **Feed Data:** Drive `ui_in[0]` with your serial data stream (LSB first) while holding `ui_in[1]` (bit_valid) HIGH for exactly 64 clock cycles.

3. **Wait for Inference:** After the 64th bit, drop `ui_in[1]` LOW. Wait a few clock cycles for the data to propagate through the MAC and scoring pipelines.
4. **Monitor Outputs:** * Watch `uio_out[0]` for a 1-clock-cycle HIGH pulse. This indicates the evaluation is complete.
    - At that exact cycle, read the 8-bit threat score on the `uo_out` pins.
    - Check if `uio_out[1]` pulsed HIGH, which indicates an interrupt triggered by a CRITICAL score (`0xFF`).

Example Test Vectors:

- **Velocity = 10, Heading = 5:** Engine score should read `0x00` (Safe).
- **Velocity = 120, Heading = 40:** Engine score should read `0xFF` (Critical) and `uio_out[1]` will pulse.

External hardware

- **Microcontroller (e.g., RP2040 / Raspberry Pi Pico):** Required to bit-bang the serial data into the input pins (`ui_in[0]` and `ui_in[1]`) and monitor the output interrupt flag. The standard Tiny Tapeout demo board provides this capability.
- **(Optional) 8x LEDs:** Connected to `uo_out[7:0]` to provide a visual, real-time representation of the threat level.
- **(Optional) Logic Analyzer:** To visualize the serial ingestion and accurately capture the 1-cycle `done_flag` and `irq_flag` pulses.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	serial_bit_in	threat_score[0]	done_flag (out)
1	bit_valid_strobe	threat_score[1]	irq_flag (out)
2	mode_sel (reserved)	threat_score[2]	uart_tx (out)
3	—	threat_score[3]	—
4	—	threat_score[4]	—
5	—	threat_score[5]	—
6	—	threat_score[6]	—
7	—	threat_score[7]	—

V2X Collision Warning

by **PM Likhith Kumar**

0486 HDL Project

github.com/Cambridge-CIT-ASIC-Tapeouts/V2X_Collision_Warning_system

"Simple V2X collision warning system using Verilog"

How it works

Vehicle to vehicle warning system

How to test

Keep the module in both the cars and try to communicate with each other.

External hardware

PCB

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Vehicle1 bit0	Collision Warning	—
1	Vehicle1 bit1	—	—
2	Vehicle1 bit2	—	—
3	Vehicle1 bit3	—	—
4	Vehicle2 bit0	—	—
5	Vehicle2 bit1	—	—
6	Vehicle2 bit2	—	—
7	Vehicle2 bit3	—	—

Multi-Protocol Communication Controller

by **Chirantan J Tilavalli**

487 | 1 MHz | HDL Project

github.com/CambridgeinstituteoftechnologyBLR-chiru/Communication-protocol

"Supports UART, SPI and I2C transmission modes using a compact single-module design."

Description

The Multi-Protocol Communication Controller is a compact digital communication module designed for Tiny Tapeout. The controller supports three commonly used serial communication protocols: UART, SPI, and I2C. A protocol is selected using dedicated mode control inputs, allowing the same hardware to operate in different communication modes.

This project demonstrates finite state machine (FSM) based protocol control, serial data transmission, and multiplexed protocol selection while fitting within a Tiny Tapeout 1x1 tile.

How it works

The design accepts a 4-bit input data value, a start signal, and two mode selection bits.

Protocol Selection

MODE[1:0]	Protocol
00	UART
01	SPI
10	I2C
11	Reserved

When the START signal is asserted, the controller loads the input data and begins transmission according to the selected protocol.

Inputs

Input	Description
DATA0-DATA3	4-bit data input
START	Starts transmission

MODE0	Protocol select bit 0
MODE1	Protocol select bit 1
RESET	Resets the controller

Outputs

Output	Description
UART_TX	UART transmit output
SPI_MOSI	SPI master output
SPI_SCLK	SPI serial clock
SPI_CS	SPI chip select
I2C_SDA	I2C data line
I2C_SCL	I2C clock line
BUSY	Transmission in progress
DONE	Transmission completed

The controller uses a simple state machine consisting of IDLE, LOAD, SEND, and DONE states to manage the communication process.

How to test

UART Test

1. Apply RESET.
2. Set MODE = 00.
3. Load DATA[3:0] with a test value.
4. Assert START for one clock cycle.
5. Observe UART_TX output.
6. Verify BUSY becomes high during transmission.
7. Verify DONE becomes high after transmission completes.

SPI Test

1. Apply RESET.
2. Set MODE = 01.
3. Load DATA[3:0] with a test value.
4. Assert START.
5. Observe SPI_MOSI, SPI_SCLK, and SPI_CS.
6. Verify BUSY and DONE operation.

I2C Test

1. Apply RESET.
2. Set MODE = 10.
3. Load DATA[3:0] with a test value.

4. Assert START.
5. Observe I2C_SDA and I2C_SCL outputs.
6. Verify BUSY and DONE operation.

Simulation

Run the Tiny Tapeout cocotb tests:

```
cd test
make -B
```

The supplied cocotb testbench automatically verifies UART, SPI, and I2C operating modes.

External hardware

No external hardware is required. The design can be fully verified through simulation using the provided cocotb testbench.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA0	UART_TX	—
1	DATA1	SPI_MOSI	—
2	DATA2	SPI_SCLK	—
3	DATA3	SPI_CS	—
4	START	I2C_SDA	—
5	MODE0	I2C_SCL	—
6	MODE1	BUSY	—
7	RESET	DONE	—

SPI Slave with 8-Register File

by **Usha**

512 | 10 MHz | HDL Project

`github.com/Ushaop-op/ttgf-spi`

"An industry-grade SPI slave peripheral with an internal 8-byte register file"

How it works

This design implements a standard 4-wire SPI (Serial Peripheral Interface) slave targeting the GlobalFoundries 180nm platform. It contains a synchronous Finite State Machine (FSM) that decodes incoming SPI transactions on `SCLK`, `MOSI`, and `CS_N`.

The core features an internal 8-byte register file. Transactions consist of a 1-byte command phase (specifying a Read/Write bit and a 3-bit register address) followed by a 1-byte data transfer phase. On a write transaction, data is sampled from `MOSI` and written to the selected register. On a read transaction, data from the internal register is driven out onto `MISO`. The content of Register 0 is continuously driven out to the dedicated output pins (`uo_out`) for real-time hardware monitoring.

How to test

To test this design, keep `CS_N` high initially. Toggle `SCLK` and ensure `MISO` remains in a high-impedance or idle state.

1. Pull `CS_N` low to initiate a transaction.
2. Stream 8 bits on `MOSI` synchronized to `SCLK` rising edges: Set the first bit to `0` (Write command) followed by `0000000` (Address 0).
3. Immediately stream another 8 bits containing the data payload (e.g., `0xA5`) on `MOSI`.
4. Pull `CS_N` back high to commit the byte.
5. Verify that the external output pins `uo_out[7:0]` update to show `0xA5`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	unused	reg0_out[0]	SCLK (Input)
1	unused	reg0_out[1]	MOSI (Input)
2	unused	reg0_out[2]	CS_N (Input)

#	Input	Output	Bidirectional
3	unused	reg0_out[3]	MISO (Output)
4	unused	reg0_out[4]	unused
5	unused	reg0_out[5]	unused
6	unused	reg0_out[6]	unused
7	unused	reg0_out[7]	unused

Project

by **DHR**

0513 HDL Project

github.com/DRosen766/ttgf-26b

"*Description of project*"

How it works

This is how my project works

How to test

This is how to test my project

External hardware

No external hardware

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A	I	Q
1	B	J	R
2	C	K	S
3	D	L	T
4	E	M	U
5	F	N	V
6	G	O	W
7	H	P	X

4 bit Accumulator CPU

by **Darshan**

`514` `100 MHz` `Wokwi Project`

`github.com/Cambridge-CIT-ASIC-Tapeouts/ttgf-4b-accumulator-cpu`

`wokwi.com/projects/0`

> *"A minimal 4-bit accumulator-driven CPU where each instruction from ROM operates on the accumulator via an ALU (ADD, SUB, AND, OR), with the accumulator storing the result of each operation for the next cycle."*

What is this design

This is a simple CPU (Central Processing Unit) built in Verilog, which is a hardware description language. Think of this as the brain of a small computer. It's a tiny processor that can perform basic operations like adding, subtracting, and doing simple logic operations (AND, OR).

The CPU has a few important parts:

Accumulator (acc) – This is like a temporary storage space where the CPU keeps results of calculations. It holds values that the CPU works with.

Program Counter (pc) – This keeps track of what instruction the CPU is working on right now, like a bookmark that tells the CPU which part of its instruction list to read next.

ALU (Arithmetic Logic Unit) – This is where the actual math and logic operations happen, like addition or subtraction.

ROM (Read-Only Memory) – A memory space that holds a list of instructions for the CPU to follow. It's "read-only" because it doesn't change during normal operation, just like a recipe you follow step by step.

How it works

Start-up: When you turn it on (or reset it), everything gets set to zero: the Accumulator (where the results are stored), the Program Counter (which keeps track of which instruction is next), and the Immediate Value (a small number used in calculations).

Getting Instructions: The Program Counter decides what instruction to fetch next from the ROM. It's like looking at the next step in a recipe.

Executing Instructions: Once the CPU gets an instruction, it decodes what to do (like whether to add or subtract) based on the instruction it fetched.

If the CPU is in “execute mode,” it’ll perform a calculation using the Accumulator (which holds the current value) and an Immediate Value (a small number it got from the instruction). The result of the operation is stored back in the Accumulator.

Repeat: After doing the calculation, the Program Counter moves forward to the next instruction. This keeps happening until the CPU has finished all its instructions. Explain how your project works

How to test

Clock:

The CPU needs a clock to work, just like how a heart beats at regular intervals. Every time the clock ticks, the CPU moves to the next step.

Starting the Test:

First, we reset the CPU to make sure everything starts fresh, just like turning a calculator off and on to clear it.

Feeding Instructions:

We give the CPU a bunch of test instructions (like “add 5 to this” or “subtract 3 from that”) and let it go through them step by step.

Checking Results:

We watch how the Accumulator changes. It should hold the right result after every operation (like if we told it to add, the Accumulator should show the sum). We also watch the Program Counter to see if it’s correctly moving from one instruction to the next.

Final Check:

At the end, the CPU should have gone through all the instructions correctly, and the Accumulator should hold the right results after each operation. ## External hardware

Non

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Data Input 1	Output data 1	Operation mode check
1	Data Input 2	Output data 2	—
2	Data Input 3	Output data 3	—
3	Data Input 4	Output data 4	—

#	Input	Output	Bidirectional
4	Opcode Input 1	ALU out 1	—
5	Opcode Input 2	ALU out 2	—
6	Opcode Input 3	ALU out 3	—
7	Opcode Input 4	ALU out 4	—

Simple SPI configuration for analog designs

by **Alan G**

515 HDL Project

`github.com/agurlask/ttgf26b-spi-config-reg`

"Simple 8-bit unidirectional SPI register for external configuration of future analog/mixed-signal designs"

How it works

The 8-bit SPI configuration register consists of a shift register and an output latch.

The SPI interface shifts in the next value of the register on MOSI when chip select (CS) is low (shifting is done MSB-first). Once CS is high, the register outputs update to the current value of the shift register (even if less/more than 8 clocks were received).

The design also includes an active-high asynchronous clear signal which is used to reset the state of the register to a default value.

The 8-bit parallel output of the register is accessible from the output pins.

The IP is intended to be eventually used to write configuration bits to analog/mixed-signal designs with limited pin access.

How to test

Note: see pinout table for pin location information.

By default, the register should read the value 0xAA.

Use the protocol stated above to write to the 8-bit register and check the parallel output.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SCLK	CONFIG[0]	—
1	MOSI	CONFIG[1]	—
2	CSN	CONFIG[2]	—
3	CLR	CONFIG[3]	—

#	Input	Output	Bidirectional
4	—	CONFIG[4]	—
5	—	CONFIG[5]	—
6	—	CONFIG[6]	—
7	—	CONFIG[7]	—

Programmable Chaotic NLFSR

by **Aishu**

0516 | 10 MHz | HDL Project

github.com/CambridgeinstitutetechnologyBLR-Aishu/CIT_Programmable-NLFSR

"A 16-bit NLFSR with two switchable non-linear feedback functions for cryptographic chaos."

How it works

This project implements a 16-bit Non-Linear Feedback Shift Register (NLFSR) designed for cryptographic primitives and pseudo-random number generation. Unlike a standard LFSR, this design utilizes non-linear logic gates (AND/OR) in the feedback path to increase algebraic complexity. The design is programmable, allowing the user to switch between two distinct non-linear feedback functions in real-time by toggling the `mode_select` input (ui[0]). Function A (Mode 0) focuses on high algebraic degree using AND/XOR logic, while Function B (Mode 1) provides a chaotic variant using an alternate tap configuration.

How to test

1. Apply a clock signal to the `clk` pin.
2. Set the `ena` pin to high and pulse the `rst_n` pin low to load the internal 16-bit seed (0xACE1).
3. Observe the resulting bitstream on the 8 dedicated output pins (`uo_out` for bits 15:8) and the 8 bidirectional pins (`uio_out` for bits 7:0).
4. To verify the programmable logic, toggle the `ui[0]` input and observe the output sequence divergence. In Gate Level (GL) simulation or hardware, allow at least 30 clock cycles for the two modes to produce unique, non-overlapping sequences.

External hardware

This project is self-contained and does not require complex external hardware. It can be tested using the standard TinyTapeout Demo Board switches to control input pins and a logic analyzer or oscilloscope to capture the 16-bit output state from the `uo_out` and `uio_out` pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	mode_select	nlfsr_8	nlfsr_0
1	—	nlfsr_9	nlfsr_1
2	—	nlfsr_10	nlfsr_2
3	—	nlfsr_11	nlfsr_3
4	—	nlfsr_12	nlfsr_4
5	—	nlfsr_13	nlfsr_5
6	—	nlfsr_14	nlfsr_6
7	—	nlfsr_15	nlfsr_7

I2C Slave Template with Emulated Sensor

by **Sebastian Budde** & **Janina Speckmann**

`517` `25 MHz` `HDL Project`

`github.com/BastiBudde/TTSky26b_I2C-Slave-Sensor`

> *"I2C slave (tested up to FM+) with eight writable registers and eight read-only registers fed by an internal LFSR that emulates a sensor - a reusable template for a real I2C sensor."*

How it works

This project implements an I2C slave that emulates a real-world sensor. It exposes two register banks of eight 8-bit registers each, addressed by the I2C master via register indices `0x00`–`0x0F`.

Block A (`0x00`–`0x07`) is writable by the master and serves as configuration registers - a placeholder for the configuration interface a real sensor would expose. Block B (`0x08`–`0x0F`) is read-only for the master and is driven by an internal LFSR (linear-feedback shift register) that cyclically updates each register with pseudo-random values. The master can reads these as simulated sensor data that changes over time.

A constant device signature is available at addresses `0xF8`–`0xFF`. Reads from any unmapped address return `0x00`, and writes to read-only or unmapped addresses are acknowledged on the bus but have no effect. The slave operates at all three standard bus speeds: Standard Mode (100 kHz), Fast Mode (400 kHz) and Fast Mode Plus (1 MHz), with a 25 MHz system clock.

The design is a reusable template: replacing the LFSR with a real sensor frontend turns it into a functional I2C sensor.

How to test

The device responds at 7-bit I2C address `0x55`. To read a register, write the register index, then issue a repeated START and read one or more bytes (the register pointer auto-increments for bulk reads). To write Block A, send the register index followed by one or more data bytes (the register pointer auto-increments for bulk writes).

The design was verified on hardware using a Sipeed Tang Primer 25K FPGA with an ESP32-C6 acting as I2C master, running a test suite at all three bus speeds. The full setup, firmware and results are documented in the FPGA test report:

External hardware

External pull-up resistors (e.g. 4.7 kΩ to 3.3 V) on the SCL pin (uio[0]) and the SDA pin (uio[3]). No other external hardware is required.

Project Pinout

Digital Pins

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

PQC NTT Butterfly Core

by **Aishwarya Gadela** & **Bojji Reddy Rishika**

518 | 10 MHz | HDL Project

github.com/bojjireddyrishika12-web/butterfly

"Hardware-accelerated Cooley-Tukey NTT Butterfly processing block for PQC Kyber"

How it works

This project implements a hardware-accelerated Cooley-Tukey Number Theoretic Transform (NTT) Butterfly Core optimized for CRYSTALS-Kyber post-quantum cryptography ($q = 3329$). It performs fast modular addition and subtraction arithmetic arrays using specialized combinational subtraction matrix logic blocks without slow division operations.

How to test

Provide input arrays for A_in on the primary input bus, and seed values for B_in and twiddle factors W_in on the split bidirectional bus lines. Drive clock transitions to observe calculated modular transformation bounds on X_out and Y_out.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A_in[0]	X_out[0]	B_in[0]
1	A_in[1]	X_out[1]	B_in[1]
2	A_in[2]	X_out[2]	B_in[2]
3	A_in[3]	X_out[3]	B_in[3]
4	A_in[4]	X_out[4]	Y_out[4]
5	A_in[5]	X_out[5]	Y_out[5]
6	A_in[6]	X_out[6]	Y_out[6]
7	A_in[7]	X_out[7]	Y_out[7]

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

BSD Convolution Adder Tree

by **Vincent Mertens**

519 | 100 kHz | HDL Project

github.com/Vincent2405/BSD-Convolution-Adder-Tree-TinyTapeout

"Multiplierless shift-add tree for image convolution"

How it works

SD format: 00 => −1 01 => 0 10 => 0 11 => 1

reads 8 values in Registers R0-R7, adds 8 values in BSD format where each value is shifted by 1 digit resulting in: Y = R0 · 1 + R1 · 2 + R2 · 4 + R3 · 8 + R4 · 16 + R5 · 32 + R6 · 64 + R7 · 128

every Addition is performed in BSD format either BSD + TC => BSD or BSD + BSD=> BSD

output given in BSD format.

How to test

Because Tiny Tapeout provides only 8 dedicated input bits and 8 dedicated output bits, the register inputs and result output are multiplexed.

Control signals on uio_in uio_in[7] => write enable uio_in[2:0] => register select for R0 to R7 uio_in[4:3] => output chunk select

During writing, uio_in[7] must be set to 1. This enables writing to the selected register.

uio_in[7] = 1 => write enabled uio_in[7] = 0 => write disabled

After all registers have been loaded, uio_in[7] must be set back to 0. This prevents accidental overwriting of registers during readout.

Writing registers:

To write a value into a register:

Put the input value on ui_in[7:0]. Set uio_in[7] = 1. Set uio_in[2:0] to the target register index. Apply one clock cycle.

Register selection:

uio_in[2:0] = 000 => R0 uio_in[2:0] = 001 => R1 uio_in[2:0] = 010 => R2 uio_in[2:0] = 011 => R3 uio_in[2:0] = 100 => R4 uio_in[2:0] = 101 => R5 uio_in[2:0] = 110 => R6 uio_in[2:0] = 111 => R7

Example input values:

[2, 4, 8, 16, 8, 4, 2, 0]

Write sequence:

ui_in = 00000010 uio_in = 10000000 clock

ui_in = 00000100 uio_in = 10000001 clock

ui_in = 00001000 uio_in = 10000010 clock

ui_in = 00010000 uio_in = 10000011 clock

ui_in = 00001000 uio_in = 10000100 clock

ui_in = 00000100 uio_in = 10000101 clock

ui_in = 00000010 uio_in = 10000110 clock

ui_in = 00000000 uio_in = 10000111 clock

-writing done

uio_in = 00000000 clock Reading the output

The result ist lenght 26 bit so we need to multiplex through uo_out[7:0].

The output chunk is selected using uio_in[4:3]

uio_in[4:3] = 00 => first 8 bits are laying on uo_out[0:7] : o0 uio_in[4:3] = 01 => second 8 bits ": o1 uio_in[4:3] = 10 => third 8 bits": o2 uio_in[4:3] = 11 => fourth 8 bits ": o3

Read sequence:

uio_in = 00000000 uo_out now contains o0

uio_in = 00001000 uo_out now contains o1

uio_in = 00010000 uo_out now contains o2

uio_in = 00011000 uo_out now contains o3

The full BSD result is reconstructed as:

[o3 | o2 | o1 | o0]

Real convolution usage

For a real 3x3 image convolution, the values loaded into registers R0 to R7 should be the preadded weight sums of the 3x3 filter kernel.

The circuit itself does not contain the full ROM. Instead, the ROM lookup can be simulated or calculated externally. The external ROM implements the lookup table for the fixed 3x3 filter kernel(for example gaussian).

For each bit position of the 8-bit pixel values, one bit from each of the nine pixels is used to form a 9-bit ROM address:

[p1, p2, p3] [p4, p5, p6] => 8 x 9-bit address => 512-entry ROM => 8 preadded values => load into R0 to R7 => BSD Out of complete Convolution [p7, p8, p9]

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input value bit 0	BSD output chunk bit 0	register select bit 0
1	input value bit 1	BSD output chunk bit 1	register select bit 1
2	input value bit 2	BSD output chunk bit 2	register select bit 2
3	input value bit 3	BSD output chunk bit 3	output chunk select bit 0
4	input value bit 4	BSD output chunk bit 4	output chunk select bit 1
5	input value bit 5	BSD output chunk bit 5	—
6	input value bit 6	BSD output chunk bit 6	—
7	input value bit 7	BSD output chunk bit 7	write enable

Quadrature sine generator

by **Framcesco Osti**

0545 | 50 MHz | HDL Project

`github.com/fran-retfie/GF26b-qsdss`

> "*A Quadrature Sine Direct Digital Synthetizer with PCM outputs*"

How it works

This project generates two sinusoidal waveforms with a fixed 90° phase offset (I/Q signals) using Direct Digital Synthesis (DDS) techniques. The output is presented on output in Pulse Code Modulation via two first order Sigma-Delta modulators. DSS block can be prescaled by sending the prescaler value through SPI port.

Oscillator design is based on the following paper: J. Nam, A Study of Sinusoid Generation Using Recursive Algorithms

Block diagram

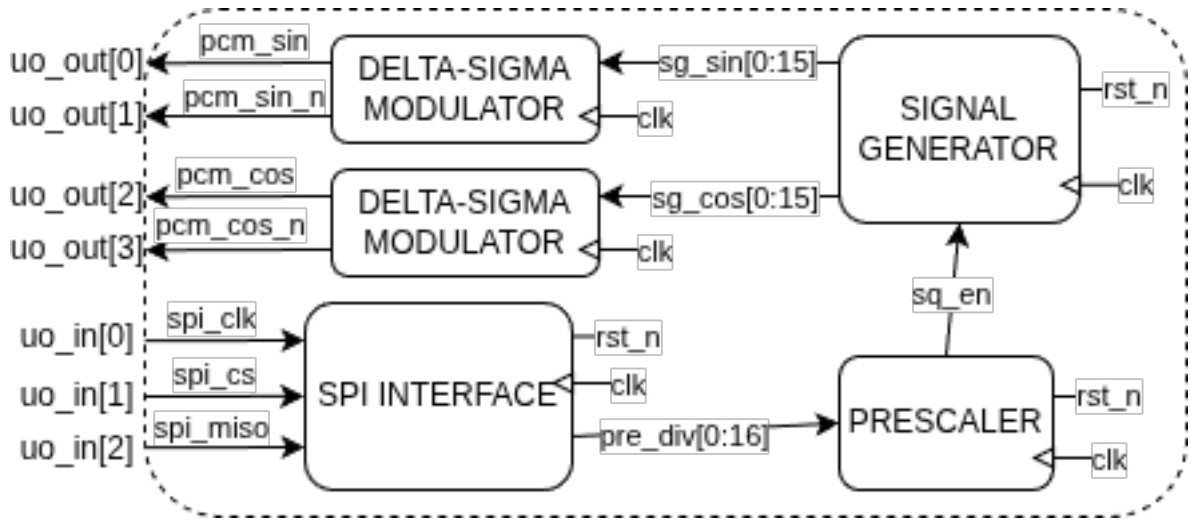


Figure 545.1: image

PINPUT

Inputs:

- `spi_clk = uo_in[0]`
- `spi_cs = uo_in[1]`
- `spi_miso = uo_in[2]`

Outputs:

- `pcm_sin = uo_out[0]`
- `pcm_sin_n = uo_out[1]`
- `pcm_cos = uo_out[2]`

- `pcm_cos_n = uo_out[3]`

How to test

Basic test

- Reset device by putting `rst_n = 0` while providing a clock signal cycle on `clk` pin.
- While keeping `spi_cs = 0` provide 50MHz clock to `clk` pin, a 1kHz PCM sine and cosine signal should be visible respectively on `pcm_sin` and `pcm_cos` pins.
- `pcm_sin_n` and `pcm_cos_n` pins are the negated counterparts. ### Full test
- Buffer the outputs `pcm_sin`, `pcm_cos`, `pcm_sin_n` and `pcm_cos_n` with for example a 74AC244 Digital buffer IC and filter them with a RC filter (suggested cutoff frequecy 50-100 kHz range).
- Observe sine and cosine analog outputs.
- Using SPI port, the prescaler `pre[15:0]` cane be loaded with new value allowing for output signals frequency control. After reset prescaler default value is `0x0124` corresponding to 1kHz. In general output frequency is given by rougthly `f_sig = f_clk / pre[15:0] / 400`

SPI timing diagram

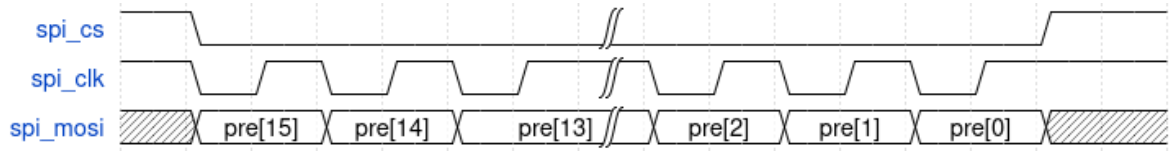


Figure 545.2: image

Output wiring diagram

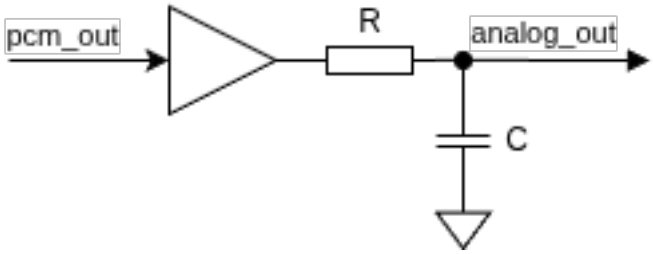


Figure 545.3: image

External hardware

Recomanded hardware:

- Digital buffer IC (e.g: 74AC244)
- n.2 RC filters
  - R = 220 ohm
  - C = 10 nF

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SCK	PCM_SIN	—
1	CS	PCM_COS	—
2	MOSI	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Tiny RIng Oscillator PUF

by **Manuel Roales**

0547 50 MHz HDL Project

`github.com/mRoales/Programmable_Ring_Oscillator_PUF`

"A Ring Oscillators PUF wit programmable characteristics"

How it works

This project implements a **Programmable Length Ring Oscillator** designed as a Physical Unclonable Function (PUF) for hardware security applications. The core architecture consists of an inverter chain whose effective delay path can be dynamically configured using digital selection signals.

By manipulating the multiplexer controls, the user can select different feedback loops and vary the number of active inverting stages. Due to sub-nanosecond propagation delays and microscopic process variations unique to each die during the GlobalFoundries 180nm (`gf180mcu`) manufacturing process, the circuit oscillates at a unique, device-specific frequency for each challenge combination. This chaotic asynchronism acts as a silicon fingerprint.

An integrated control path monitors the state of the circuit, sample capture events, and triggers a telemetry transmission when the output is ready.

Top-Level Port Mapping (Tiny Tapeout Template Integration)

The hardware signals have been packed and mapped into the standard Tiny Tapeout 8-bit buses as follows:

Dedicated Inputs (`ui_in`)

Bit	Signal Name	Type	Description
[1:0]	`sel_mux_0`	Input	Multiplexer 0 selection lines (Challenge bits 0-1)
[3:2]	`sel_mux_1`	Input	Multiplexer 1 selection lines (Challenge bits 2-3)
[6:4]	`n_inv`	Input	Number of active inverter stages selection (Challenge bits 4-6)
[7]	`enable`	Input	Ring Oscillator hardware enable (1 = Run, 0 = Standby/Gated)

Bidirectional Controls (`uio_in` / `uio_out`)

Note: Configured as inputs for control handshake.

Bit	Signal Name	Type	Description
[0]	`tx_ready`	Input	Handshake signal indicating receiver is ready for data
[1]	`op_mode`	Input	Operation Mode configuration bit
[7:2]	*Unused*	Input	Reserved (Driven to 0)

Dedicated Outputs (`uo_out`)
The output ports stream out the digitized frequency signatures or telemetry data generated by the PUF core.

How to test

To prevent excessive power consumption and thermal saturation, the Ring Oscillator features a gated input. Follow these sequential steps to test the project behavior:

1. **System Reset:** Drive `rst_n` LOW for at least 10 clock cycles while keeping `ui_in` and `uio_in` at `0`. Then release the reset by driving `rst_n` HIGH.
2. **Challenge Selection:** Provide the desired delay path configuration by assigning values to `sel_mux_0`, `sel_mux_1`, and `n_inv` using the lower 7 bits of `ui_in`.
3. **Core Activation:** Drive `ui_in[7]` (`enable`) HIGH. The Ring Oscillator will immediately start oscillating at its native physical frequency.
4. **Data Acquisition:** Set `uio_in[0]` (`tx_ready`) HIGH to enable the internal transmission logic. Observe the output data streams or frequency signatures on the dedicated output pins (`uo_out`).
5. **Safe Standby:** Once data collection is completed, immediately drive `ui_in[7]` (`enable`) back to `0` to gate the oscillator loop and release system resources.

External hardware

No external hardware is strictly required for basic evaluation, as the digital control buses can be fully driven using Tiny Tapeout's standard digital input switches.

However, for high-precision validation:

- An **Oscilloscope or Logic Analyzer** can be connected to the dedicated output pins (`uo_out`) to measure raw digital jitter or frequency deviations.
- A **Microcontroller or FPGA board** (via PMOD) can be attached to automate challenge injection loops (`ui_in`) and analyze the uniqueness and reliability metrics of the PUF responses.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sel_mux_0[0]	debug_ro[0]	tx_ready
1	sel_mux_0[1]	debug_ro[1]	op_mode
2	sel_mux_1[0]	debug_ro[2]	—
3	sel_mux_1[1]	debug_ro[3]	—
4	n_inv[0]	valid_out[0]	—
5	n_inv[1]	valid_out[1]	—
6	n_inv[2]	cnt_data_out[0]	debug_done_out[0]
7	puf_enable	cnt_data_out[1]	debug_done_out[1]

Detronyx UART Trace Exerciser

by **Detronyx contributors**

549 | 50 MHz | HDL Project

`github.com/Zhekar1998/tt_um_detronyx_uart_trace_exerciser`

"UART-controlled pattern generator and streaming trace/event analyzer"

How it works

Detronyx UART Trace Exerciser is a 1x1 digital bring-up tile. It exposes eight trace probe inputs on `ui_in[7:0]`, eight generated pattern outputs on `uo_out[7:0]`, and a UART control/status link on `uio[0]`/`uio[1]`.

The trace side samples `ui_in[7:0]` at a programmable interval. When any masked bit changes while tracing is armed and streaming is enabled, the tile emits a four-byte UART trace packet:

```
0xe1 sample delta changed_mask
```

`delta` is an 8-bit saturated count of sample ticks since the previous emitted event. If a second event arrives while a previous trace event is still waiting to be transmitted, the event is dropped and the overflow/drop counters update. This keeps the design small and predictable for a single TTGF tile.

The pattern side drives `uo_out[7:0]` with one of eight modes:

```
0 hold pattern_a
1 incrementing counter
2 walking one
3 8-bit LFSR
4 inverted pattern_a
5 mirror trace probe inputs
6 trace probe inputs XOR pattern_a
7 trace probe inputs XOR pattern_a
```

The project config targets the AvalonSemiconductors `gf180mcu_as_sc_mcu7t3v3` GF180 native-3.3 V standard-cell library.

UART protocol

Default UART settings are 115200 baud, 8 data bits, no parity, 1 stop bit, with a 50 MHz `clk`. The UART bit divider is the top-level `BAUD_RELOAD` parameter; the default is `433`, which gives 434 clock cycles per bit at 50 MHz. For a different system clock, set `BAUD_RELOAD` to `round(clock_hz / baud) - 1`.

Most commands are two bytes: command, then argument. They do not ACK, to avoid polluting the trace stream.

```
0x10 mask    set trace bit mask
0x11 div     set trace sample divider; low 4 bits used, 0 samples
every clock
0x12 ctrl    bit0 arm, bit1 stream enable, bit2 clear counters/
errors, bit3 snapshot
0x20 mode    set pattern mode, low three bits used
0x21 div     set pattern update divider; low 6 bits used
0x22 value   set pattern_a
```

Single-byte commands:

```
0x30         emit status packet
0x31         emit ping/version packet
```

Status packets are:

```
0xa5 status event_count drop_count
```

Ping packets are:

```
0xd7 0x54 0x01 status
```

The `status` byte is:

```
bit7 overflow
bit6 uart_rx_error
bit5 trace_packet_pending
bit4 stream_enabled
bit3 trace_armed
bit2..0 pattern_mode
```

Pinout

```
ui_in[7:0]   trace probes
uo_out[7:0]  generated pattern
uio[0]       UART RX input
uio[1]       UART TX output
uio[2]       trace armed status
uio[3]       stream enabled status
uio[4]       UART/packet busy status
uio[5]       trace overflow status
uio[6]       UART RX framing error sticky status
uio[7]       toggles on every accepted trace event
```

How to test

Run the local RTL simulation from the TTGF experiment directory:

```
make sim-trace-exerciser
```

The test configures the pattern generator over UART, arms trace streaming, changes a probe input, checks the emitted trace packet, and then requests a status packet.

External hardware

No required external hardware beyond a UART adapter or host MCU. For a simple demo, loop selected `uo_out` pattern pins back into `ui_in` trace probe pins and watch the trace packets on `uio[1]`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	trace_probe[0]	pattern[0]	uart_rx
1	trace_probe[1]	pattern[1]	uart_tx
2	trace_probe[2]	pattern[2]	trace_armed
3	trace_probe[3]	pattern[3]	stream_enabled
4	trace_probe[4]	pattern[4]	uart_busy
5	trace_probe[5]	pattern[5]	trace_overflow
6	trace_probe[6]	pattern[6]	uart_rx_error
7	trace_probe[7]	pattern[7]	event_toggle

BRISQ

by **Poiku**

`0551` `25 MHz` `HDL Project`

`github.com/pratyaygopal/BRISQ`

"BRISQ: Binary Reduced Inverse SQrt"

How it works

BRISQ computes an approximate IEEE-754 single-precision inverse square root:

```
y1 = y0 * (1.5 - 0.5 * x * y0 * y0)
```

The Tiny Tapeout top module is `tt_um_brisq`. The design receives one 32-bit FP32 operand over an 8-bit byte-serial input bus, runs one Newton-Raphson refinement, and returns one 32-bit FP32 result over an 8-bit byte-serial output bus.

The initial estimate is generated in `src/top.sv` with a Quake-style integer seed:

```
seed_y0 = WTF - (input_magnitude >> 1)
```

`WTF` is a localparam in the RTL and is currently `31'h5F3759DF`. The cocotb testbench reads this value from the design, so the expected-value model remains parametric with the RTL seed constant.

The implementation is optimized for small area:

- `src/top.sv` contains the Tiny Tapeout wrapper, byte SerDes, FSM, seed logic, and special-case handling.
- `src/fp32_mul.sv` implements a truncated positive-magnitude FP32-style multiplier using the high `PRECISION_BITS` fraction bits.
- `src/fp32_sub.sv` implements the parametric `1.5 - b` subtractor used by the Newton correction term.

The default precision is 11 high fraction bits. This is an approximate datapath, not a fully rounded IEEE-754 FPU.

I/O protocol

All FP32 values are transferred most-significant byte first.

Pin	Direction	Description
`ui_in[7:0]`	input	Input FP32 byte
`uio_in[0]`	input	Input byte valid

`uo_out[7:0]`	output	Output FP32 byte
`uio_out[7]`	output	Output byte valid
`uio_out[6]`	output	Final output byte
`uio[5:1]`	input/unused	Unused

To send an operand, drive each byte on `ui_in[7:0]` and assert `uio_in[0]` for the clock edge that accepts that byte. The receive FSM waits when `uio_in[0]` is low, so the input stream can pause between bytes.

After the fourth input byte is accepted, the accelerator computes for four clock cycles. It then drives four output bytes on `uo_out[7:0]`. `uio_out[7]` is high while output bytes are valid, and `uio_out[6]` is high with the fourth and final output byte.

Special cases

Input class	Output
zero or subnormal	`+inf`
negative value	quiet NaN
`+inf`	`+0.0`
NaN	quiet NaN

How to test

From the `test` directory, install dependencies once:

```
python3 -m pip install -r requirements.txt
```

Run the self-checking cocotb RTL vector test:

```
make -B
```

Run the sweep test, which generates `sweep_results/isqrt_sweep.csv` and `sweep_results/isqrt_sweep.png`:

```
make sweep
```

Use more sweep samples with:

```
make sweep SWEEP_POINTS=4096
```

For gate-level simulation, copy the hardened netlist to `test/gate_level_netlist.v` and run:

```
make -B GATES=yes
```

The cocotb testbench writes `tb.fst`, which can be opened with GTKWave or Surfer.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	fp32_in_byte[0]	fp32_out_byte[0]	fp32_in_valid
1	fp32_in_byte[1]	fp32_out_byte[1]	—
2	fp32_in_byte[2]	fp32_out_byte[2]	—
3	fp32_in_byte[3]	fp32_out_byte[3]	—
4	fp32_in_byte[4]	fp32_out_byte[4]	—
5	fp32_in_byte[5]	fp32_out_byte[5]	—
6	fp32_in_byte[6]	fp32_out_byte[6]	fp32_out_last
7	fp32_in_byte[7]	fp32_out_byte[7]	fp32_out_valid

4-bit Maximum-Length LFSR

by **Geeta Doddamani**

0576 | 10 MHz | HDL Project

github.com/Geeta-doddamani/ttgf-lfsr

"A 4-bit maximum-length LFSR with seed 0x1 and feedback polynomial x4+x+1. Generates pseudo-random sequence of period 15."

How it works

This project implements a 4-bit maximum-length Linear Feedback Shift Register (LFSR) using the feedback polynomial x4+x+1 with taps at bit positions 3 and 0.

On every rising clock edge the register shifts left by one position. A feedback bit computed as XOR of bits 3 and 0 is inserted at the LSB. When the active-low reset (rst_n) is asserted the register loads seed value 0x1.

The LFSR cycles through all 15 non-zero states before repeating: 1, 3, 7, F, E, D, A, 5, B, 6, C, 9, 2, 4, 8.

The 4-bit output is on uo_out[3:0]. The upper nibble uo_out[7:4] is always 0. All bidirectional IOs are unused.

How to test

Assert reset by driving rst_n = 0 for at least 2 clock cycles. After reset, uo_out[3:0] should read 0x1. Release reset. On each rising clock edge the output advances through the sequence: 1, 3, 7, F, E, D, A, 5, B, 6, C, 9, 2, 4, 8, then repeats. After 15 cycles the output returns to 0x1.

External hardware

None required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	LFSR bit 0 (LSB)	—
1	—	LFSR bit 1	—
2	—	LFSR bit 2	—
3	—	LFSR bit 3 (MSB)	—
4	—	—	—

#	Input	Output	Bidirectional
5	—	—	—
6	—	—	—
7	—	—	—

SPI-CPU

by **MS College of Engineering**

0577 | 30 MHz | HDL Project

`github.com/cyrilprasannarajp/tt_MS_Engg_Clg_SPI`

"A 4-bit CPU using the SPI Flash RAM from the QSPI PMOD to load programs."

How it Works

This project implements a compact, **4-bit microcoded CPU** designed for the **TinyTapeout (GF180MCU)** platform. Rather than using limited on-chip area for program storage, the CPU fetches and executes its instructions directly from an **external SPI RAM** (such as a physical 23LC512 memory chip or an RP2040 microcontroller emulating it).

Hardware Architecture

The design is split into three main functional blocks:

1. **TinyTapeout Wrapper (`tt_um_spi_cpu_top`)**: Handles top-level ASIC pins and bridges them to the internal logic.
2. **SPI Fetch & CPU Wrapper (`spi_wrap`)**: Manages the 12-bit **Program Counter (PC)**, an FSM to decode instructions fetched over SPI, and a byte-wide SPI master engine (`spi_read_byte`).
3. **Execution Unit Datapath (`ExecutionUnit`)**: Based on the Aeolus CPU Core topology, it coordinates an 8-bit Accumulator (ACC), a 4-bit Register File (Registers A, B, and O), an 8-bit shift register with a overflow flag (`SF`), and a 4-bit slice ALU.

Instruction Fetch & Execution Pipeline

To optimize memory bandwidth, **every byte fetched from the SPI RAM packs two 4-bit micro-operations**:

- `opcode1 = spi_data[3:0]` (Executed first)
- `opcode2 = spi_data[7:4]` (Executed second)

The `spi_wrap` controller cycles through a sequential Finite State Machine (FSM):

- **`S_FETCH_START`**: Triggers a memory read when the SPI engine is idle.
- **`S_FETCH_WAIT_OPCODE`**: Waits for the transaction to finish and latches the instruction byte.
- **`S_EXECUTE_1`**: Sets the execution bus to `opcode1` and pulses `cpu_start`.
- **`S_EXECUTE_2`**: Sets the execution bus to `opcode2`, pulses `cpu_start`, increments the PC, and loops back to fetch the next pair.

The underlying `spi_read_byte` module executes a standard **23LC512 READ (0x03)** command sequence, transmitting `{8'h03, 16'b0, pc}` MSB-first over MOSI before shifting in the payload.

The Microprogram (Firmware)
As a proof-of-concept hardware demonstration, the pre-loaded microcode implements a **4×4-bit to 8-bit software binary multiplier** using a shift-and-add algorithm:

- `ui_in[7:4]` = Operand A (4-bit)
- `ui_in[3:0]` = Operand B (4-bit)
- `uo_out[7:0]` = Product Output ($A \times B$)

Conditional instructions like `SNZA` and `SNZS` check the state of the shift register flag, allowing the datapath to selectively add values into the accumulator to dynamically handle binary multiplication without complex, rigid hardware branching paths.

How to Test

Verification workspace parameters rely on **cocotb** coupled with **Icarus Verilog (`iverilog`)**.

Dependencies
Ensure your environment includes Python 3.11+ and the proper HDL toolchain packages: ```sh pip install cocotb sudo apt install iverilog

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA_A[0]	OUT_REG[0]	SPI_CS_N
1	DATA_A[1]	OUT_REG[1]	SPI_MOSI
2	DATA_A[2]	OUT_REG[2]	SPI_MISO
3	DATA_A[3]	OUT_REG[3]	SPI_SCK
4	DATA_B[0]	OUT_REG[4]	—
5	DATA_B[1]	OUT_REG[5]	—
6	DATA_B[2]	OUT_REG[6]	—
7	DATA_B[3]	OUT_REG[7]	CPU_VALID

ECC Scalar Accelerator

by **PM LIKHITH KUMAR AND PRAJWAL T**

0578 HDL Project

github.com/pmlikhithkumar55-ship-it/CIT_AMG

"*Simplified ECC accelerator using Verilog HDL for ASIC implementation*"

How it works

This project implements a simplified ECC (Elliptic Curve Cryptography) Scalar Multiplication Accelerator using Verilog HDL.

The design accepts two 8-bit input values through TinyTapeout GPIO pins. These inputs represent simplified ECC operands. The accelerator performs arithmetic operations similar to ECC point processing by adding the two operands and generating a scaled output.

The architecture contains:

- Input interface
- Arithmetic processing block
- Scalar multiplication logic
- Output interface

The design is optimized for ASIC implementation using the SKY130 process and OpenLane physical design flow.

How to test

Inputs are applied using the TinyTapeout input pins.

Inputs

- ui[7:0] -> Operand A
- uio[7:0] -> Operand B

Operation

1. Apply input values to Operand A and Operand B.
2. The design performs addition of both operands.
3. The result is shifted left by one bit to emulate scalar multiplication behavior.
4. The processed value appears at the output pins.

Outputs

- uo[7:0] -> Processed ECC output

Example:

- Operand A = 5
- Operand B = 3
- Sum = 8
- Output = 16

External hardware

No external hardware is required.

The design operates entirely inside the TinyTapeout ASIC framework using GPIO interfaces.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input A bit 0	ECC output bit 0	Input B bit 0
1	Input A bit 1	ECC output bit 1	Input B bit 1
2	Input A bit 2	ECC output bit 2	Input B bit 2
3	Input A bit 3	ECC output bit 3	Input B bit 3
4	Input A bit 4	ECC output bit 4	Input B bit 4
5	Input A bit 5	ECC output bit 5	Input B bit 5
6	Input A bit 6	ECC output bit 6	Input B bit 6
7	Input A bit 7	ECC output bit 7	Input B bit 7

I2C Master Controller

by **Akshaya Institute of Technology**

0579 | 1 MHz | HDL Project

`github.com/abhilash0419/tt_Akshaya_Ins_of_Tech_I2C`

> "*Simple I2C Master Controller that transmits one byte to a fixed slave address (0x50). Generates START and STOP conditions and provides BUSY, DONE, and ACK status outputs.*"

How it works

This project implements a simple I2C master controller.

When the START input is asserted, the controller begins transmitting an 8-bit data value serially over the SDA line while generating the SCL clock.

The controller generates:

START condition 8-bit serial transmission STOP condition The BUSY output indicates an active transfer.

How to test

Apply reset. Place an 8-bit value on ui_in. Assert START. Observe SCL and SDA activity. Monitor BUSY until transmission completes.

External hardware

An I2C slave device, logic analyzer, or oscilloscope may be connected to observe SDA and SCL signals.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	START	BUSY	SDA
1	DATA_BIT1	DONE	SCL
2	DATA_BIT2	ACK	—
3	DATA_BIT3	—	—
4	DATA_BIT4	—	—
5	DATA_BIT5	—	—
6	DATA_BIT6	—	—
7	DATA_BIT7	—	—

Configurable PWM Generator

by **Shivaranjani GR**, **Ganesh Ragava** & **Sidharth Kamalakkannan**

`0580` `50 MHz` `HDL Project`

`github.com/ShivaranjaniGR/ePWM_Tapeout`

"A PWM generator featuring run-time variable period, duty cycle, and output polarity controls with built-in safety clamps."

How it works

This project is a run-time configurable Pulse Width Modulation (PWM) generator featuring dynamic controls for the period, duty cycle, and output signal polarity.

The architecture consists of three core blocks:

1. **Input Safety Constraints:** The hardware automatically enforces design rules. It ensures the operating period never falls below a safe minimum threshold of 3 clock cycles to protect physical chip output pad bandwidth. Additionally, it clamps the duty cycle ceiling so that it can never exceed the designated period, avoiding mathematical overflow errors.
2. **Up-Counter Engine:** When the system is enabled, an internal tracking counter increments on every clock cycle. Rather than counting to a fixed maximum, it monitors the safe period input value. As soon as the counter reaches the boundary threshold, it smoothly resets to zero and loops.
3. **Output Generation & Polarity Control:** The logic continually compares the tracking counter against the active duty cycle input to determine the raw output state. An inversion toggle layer is included to completely flip the resulting signal polarity if requested. The chip concurrently outputs both the primary PWM signal and its inverted, complementary twin. Disabling the module immediately forces both output tracks to zero.

How to test

To test the design, ensure the clock and active-low reset lines are operating correctly.

1. **Enable the Module:** Drive the `enable` pin high (ui_in[0]).
2. **Set the Frequency (Period):** Apply a target boundary value to the `raw_period` pins (ui_in[4:1]) to set the total loop cycle time.
3. **Set the Pulse Width (Duty Cycle):** Apply a target threshold value to the `raw_duty` pins (uio[3:0]).
   - If this value is `0`, the primary output will remain flatly deactivated.

    - If this value matches or exceeds the period, the primary output will achieve a constant active high state (100% duty cycle).
4. **Test Polarity:** Drive the `invert` pin high (ui_in[5]) to observe the primary and complementary outputs completely swap their logical signal states.
5. **Test Safe Disarm:** Drop the `enable` pin low; both outputs must instantly drive to zero.

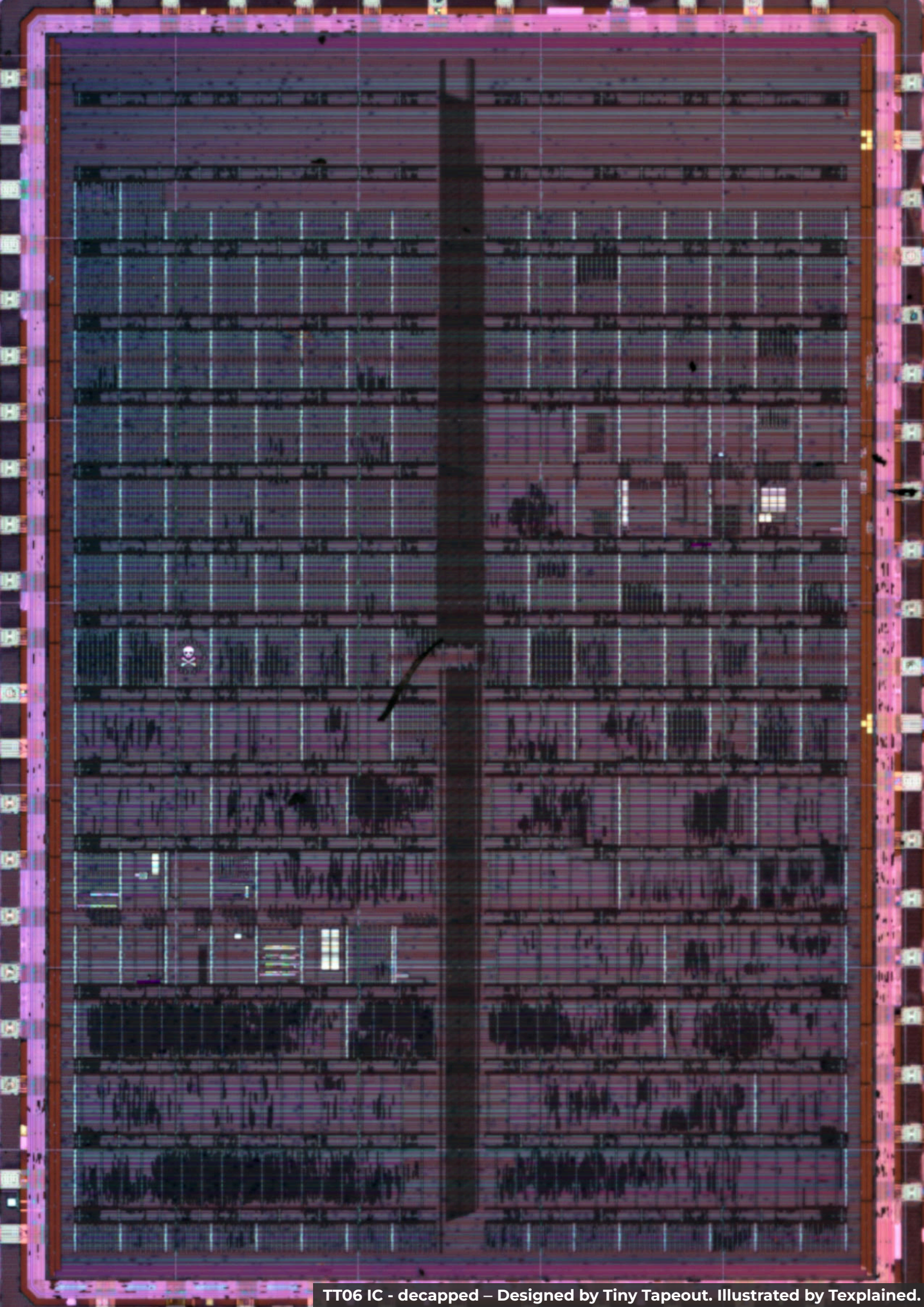
External hardware

- Standard Tiny Tapeout Demo Board input DIP switches (to adjust period, duty cycle, and toggles).
- Standard Tiny Tapeout Demo Board output LEDs (to observe the brightness change from the PWM output).
- (Optional) An oscilloscope or logic analyzer connected to the output PMOD pins to inspect the exact generated waveforms and verify the safety clamp thresholds.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	enable	pwm_out	duty_in[0]
1	period_in[0]	comp_pwm_out	duty_in[1]
2	period_in[1]	unused_out_2	duty_in[2]
3	period_in[2]	unused_out_3	duty_in[3]
4	period_in[3]	unused_out_4	unused_io_4
5	invert	unused_out_5	unused_io_5
6	unused_in_6	unused_out_6	unused_io_6
7	unused_in_7	unused_out_7	unused_io_7



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

Aswarby INT8 MAC

by **Mark Shilton**

`0581` `50 MHz` `HDL Project`

`github.com/markd666/tt-aswarby-mac`

"Byte-serial weight-stationary signed INT8 multiply-accumulate engine"

How it works

This is a **weight-stationary signed-INT8 multiply-accumulate (MAC) engine** — the single compute primitive at the heart of every quantized convolution layer. A weight is loaded once and held "stationary" while a stream of activation bytes is multiplied into a 32-bit accumulator:

```
acc = clamp_int32( acc + weight * activation )
```

Both operands are signed 8-bit (two's complement, range −128..127). The product is signed 16-bit; it is added into a **signed 32-bit accumulator** whose add **saturates** at the INT32 limits (+2147483647 / −2147483648) so overflow is well-defined, exactly as in real fixed-point inference hardware. A sticky `ovf` flag records whether saturation has ever occurred since the last clear.

Internally the design is two small modules:

- `mac_core` — the datapath: weight register, signed 8×8 multiplier, saturating 32-bit accumulator, and a combinational byte-select mux for readout. The MAC is **4-stage pipelined** — split-multiply, reconstruct product, 33-bit add, then saturate/clamp, each in its own cycle — so the result commits to the accumulator four cycles after the command is accepted. (The multiply is split into two parallel nibble-products because a full 8×8 multiply alone is too long a path for 50 MHz on the 180 nm node.)
- `mac_fsm` — a 2-state controller that converts each rising edge of `strobe` into a single-cycle execute pulse (one strobe = exactly one operation, no repeated accumulation while strobe is held high) and raises `done` once the pipelined result has committed.

Everything is fully synchronous to `clk`, single clock domain, active-low reset. The pipeline keeps each stage's logic to a single operation so the design closes timing at the 50 MHz tile target on the 180 nm GF180 process (a single-cycle multiply-and-accumulate does not — see the project notes). Per-operation latency is hidden behind `done`, so the host protocol is unchanged.

Pin map

Pins	Dir	Name	Meaning
`ui_in[7:0]`	in	data	signed INT8 operand (weight or activation)
`uio_in[1:0]`	in	cmd	`00` NOP · `01` load weight · `10` MAC · `11` clear
`uio_in[2]`	in	strobe	rising edge executes one command
`uio_in[4:3]`	in	rd_sel	which accumulator byte appears on `uo_out` (0=LSB … 3=MSB)
`uo_out[7:0]`	out	acc_byte	selected accumulator byte
`uio_out[5]`	out	done	one-cycle completion pulse
`uio_out[6]`	out	ovf	sticky saturation flag

How to test

Each operation is a three-step handshake from the Commander:

1. Drive `ui_in` (data) and `uio_in[1:0]` (command), with `strobe` low.
2. Raise `strobe` (uio_in[2]); the engine executes on the rising edge and pulses `done` (uio_out[5]) one cycle later.
3. Lower `strobe` to re-arm for the next operation.

To read the 32-bit result, set `rd_sel` (uio_in[4:3]) to 0,1,2,3 in turn and read `uo_out` each time; concatenate as little-endian to recover the signed INT32 accumulator.

Worked example (compute 3·5 + 3·5 = 30):

Step	cmd	data	result
load weight	`01`	3	weight = 3
MAC	`10`	5	acc = 15
MAC	`10`	5	acc = 30
read	`00`	rd_sel 0→3	`1E 00 00 00` → 30

The cocotb suite in `test/` drives exactly this protocol and checks every result against a Python golden model, including signed operands, clear, byte-streamed readout, a 150-step randomized sequence, and both saturation directions (the exhaustive saturation tests are gated behind `SKIP_SLOW=1`).

Vectors can be generated with `tools/export_vectors.py`, which can also pull a real INT8 weight/activation row from a quantized detector layer so the silicon is exercised with mission-representative data.

External hardware

None. The design is driven entirely from the Tiny Tapeout demo board (RP2040 + Commander); no PMOD or external parts required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data[0] (signed INT8 in)	acc_byte[0] (selected accumulator byte)	cmd[0] (in)
1	data[1]	acc_byte[1]	cmd[1] (in)
2	data[2]	acc_byte[2]	strobe (in)
3	data[3]	acc_byte[3]	rd_sel[0] (in)
4	data[4]	acc_byte[4]	rd_sel[1] (in)
5	data[5]	acc_byte[5]	done (out)
6	data[6]	acc_byte[6]	ovf (out)
7	data[7]	acc_byte[7]	—

8-bit Comparator

by **Nagaraj**

582 HDL Project

`github.com/Cambridge-CIT-ASIC-Tapeouts/VLSI-project`

"Compares two 8-bit inputs and indicates equal, greater than, or less than."

How it works

The proposed project is an 8-bit comparator designed using Verilog for the Tiny Tapeout platform. The main function of this project is to compare two 8-bit input values and generate an output based on their relationship. The first input is provided through ui_in[7:0] and the second input is provided through uio_in[7:0]. The comparator checks whether both inputs are equal, greater than, or less than each other. If both inputs are equal, the output uo_out becomes 1. If the first input is greater than the second input, the output becomes 2. If the first input is less than the second input, the output becomes 4. This project is a combinational circuit, meaning the output changes immediately whenever the input changes, without requiring a clock signal. The bidirectional pins are used only as additional input pins, while the output enable pins are disabled. This design is simple, compact, and suitable for beginners learning digital design and Tiny Tapeout project implementation.

How to test

To use the 8-bit comparator project, two 8-bit binary numbers must be applied to the input ports ui_in and uio_in. The comparator continuously compares these two input values and produces an output on uo_out. If both inputs are equal, the output becomes decimal 1. If the value applied to ui_in is greater than the value applied to uio_in, the output becomes decimal 2. If the value applied to ui_in is smaller than the value applied to uio_in, the output becomes decimal 4. Since the design is combinational, the output changes immediately whenever the inputs change, without waiting for a clock signal. For example, if ui_in = 10 and uio_in = 3, the output will be 2, indicating that the first input is greater. Similarly, if both inputs are 5, the output becomes 1, indicating equality. This project can be simulated using the provided testbench or implemented on the Tiny Tapeout platform for hardware verification.

External hardware

The 8-bit comparator project requires very minimal external hardware because it is a simple combinational digital circuit. The primary hardware requirements include a power supply, input switches, and output indicators such as LEDs. The two 8-bit input values can be provided using switches, DIP switches, or FPGA/Tiny Tapeout input pins connected to ui_in and uio_in. The output uo_out can be connected to LEDs to visually indicate the comparison result. If the output is 1, it indicates both inputs are equal; if the output is 2, it indicates the first input is greater; and if the output is 4, it indicates the first input is smaller. Since the design does not use sequential logic, external clock generation hardware is not necessary. The project can be simulated in software using Icarus Verilog and GTKWave or implemented on Tiny Tapeout-compatible hardware for real-time verification.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Input A bit 0	Equal output	Input B bit 0
1	Input A bit 1	Greater output	Input B bit 1
2	Input A bit 2	Less output	Input B bit 2
3	Input A bit 3	—	Input B bit 3
4	Input A bit 4	—	Input B bit 4
5	Input A bit 5	—	Input B bit 5
6	Input A bit 6	—	Input B bit 6
7	Input A bit 7	—	Input B bit 7

TT-Arrakeen-SPSRAM-direct

by **Staf Verhaegen**

583 | 66 MHz | HDL Project

`github.com/FibraServiTT/TTGF26b_Arrakeen_SPSRAM_direct`

"Single port SRAM with pins connected directly to TT tile pins."

How it works

This design contains a single port SRAM block with pins connected directly to TT tile pins. This allows to use this design directly as a SRAM block. This design is for 3.3V and uses the regular design rules, the SRAM design rules are not used.

The included block has 128 words of 8 bits. The dimension is 181.32µm by 83.46µm. These are the pins for the block:

- a (7 bit): address
- we (1 bit): write enable signal indicating a read or write operation
- d (8 bit): input data
- q (8 bit): output data
- clk: clock for performing an operation

On each rising edge of the clock an operation is performed on the memory. A read is done when `we` is 0, while a write operation is done when it is 1. On the rising edge of the clock the `a` and `d` signals are latched into an internal buffer. For a read operation the data for the provided address is put into the `q` signal, the `d` signal is ignored. For a write operation the value of the `d` signal is put in the given address. The write operation is write-through meaning that also `q` will get the value of `d` during the operation.

How to test

You can test the block yourself by providing the right inputs for a read or write operation. One can check if data written to a certain location is later on read back with a read operation on the same address.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	a[0]	q[0]	d[0]
1	a[1]	q[1]	d[1]

#	Input	Output	Bidirectional
2	a[2]	q[2]	d[2]
3	a[3]	q[3]	d[3]
4	a[4]	q[4]	d[4]
5	a[5]	q[5]	d[5]
6	a[6]	q[6]	d[6]
7	we	q[7]	d[7]

7SegmentDice

by **Jannis Rohleder** & **Aishwarya Kanade**

0608 | 12 MHz | HDL Project

github.com/elvtide01/7SegmentDice

"6-sided electronic dice with 7-Segment Display output"

How it works

The system is intended to recreate the effect of a rolling dice as an instrument to generate random numbers between 1 to 6 after hitting a input button. After releasing the button, the counter that initally counts in a rapid way decreases it's counting speed to simulate a roll-off effect. After the dice finally finishes counting, an external LED signalizes that the user can start to read the resulting number.

How to test

Press the trigger button to start the counter and keep it pressed for a certain time (for example 2 seconds). Release the button and after approx. 7 seconds the number on the 7-segment display shows you a random number in the range of 1 to 6. Meanwhile LED "Pulsecount" starts to flash each time the counter counts one up. After the process is done, LED "Finish" lights up and signalizes, no further counts can be expected.

External hardware

- 2 LEDs connected as active low, LED1 (Pulsecount) connected to bidirectional pin[0] and LED2 (Finish) to bidirectional pin[1].
- 1 7-segment display in common-anode configuration (or 2-digit 7-segment display connected to PMOD, whereas only one of the digits is used).
- Pin "COMMON" can be ignored if using an general purpose LED Display and only outputs a LOW signallevel.
- 1 button for triggering in active high configuration

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Trigger	SEGA	Pulsecount
1	NC	SEGB	Finish
2	NC	SEGC	NC

#	Input	Output	Bidirectional
3	NC	SEGD	NC
4	NC	SEGE	NC
5	NC	SEGF	NC
6	NC	SEGG	NC
7	NC	COMMON	NC

XORing given bits

by **Jan Cerny**

0609 HDL Project

`github.com/cherny1001/ttgf-26b`

> "*Oputpus is a XOR of ui_in and previously configured address*"

How it works

The curcit inverts bits from ui_in according to preset value. Presetting is done using uio_in[0] bit and setting ui_in to respective value.

How to test

Set ui_in to a value and keep uio_in[0] high for at least two clock cycles. Then set ui_in to any value and read the result from uo_out.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	In bit 0	Out bit 0	Cfg input 0
1	In bit 1	Out bit 1	—
2	In bit 2	Out bit 2	—
3	In bit 3	Out bit 3	—
4	In bit 4	Out bit 4	—
5	In bit 5	Out bit 5	—
6	In bit 6	Out bit 6	—
7	In bit 7	Out bit 7	—

PDM Voice Activity Detector

by **Kevin**

`0610` `1 MHz` `HDL Project`

`github.com/antimatter15/ttgf26-vad`

"A one-tile spectral PDM voice activity score."

How it works

This project is a compact activity-score estimator for a one-bit PDM microphone stream. It counts PDM ones over 64 clock cycles, measures how far that density is from the quiet midpoint of 32 ones, and accumulates 160 of those 64-bit windows into a roughly 10 ms frame at a 1.024 MHz PDM clock.

At the end of each frame, the circuit combines three tile-friendly features: low-weight frame magnitude, frame-to-frame magnitude change, and three square-wave mixer responses. The mixer phase steps are 8, 9, and 24 at the 16 kHz density-window rate, which approximate 500 Hz, 562.5 Hz, and 1.5 kHz spectral probes. These are not a full FFT; they are cheap signed accumulators that add a little frequency selectivity inside a single tile.

The internal score update is `score = score - score / 64 + feature`, with saturation at 255, and `uo[7:0]` reports that score once per frame. Quiet 50 percent-density PDM streams decay the activity toward zero, while speech-like energy changes and spectral responses drive it higher. The bidirectional pins are left unused and configured as inputs.

How to test

Apply a PDM clock, release reset, set `ui[1]` high, and present the PDM bit on `ui[0]`. Watch `uo[7:0]` for the smoothed activity score.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	pdm_data	score_bit0	—
1	sample_enable	score_bit1	—
2	—	score_bit2	—

#	Input	Output	Bidirectional
3	—	score_bit3	—
4	—	score_bit4	—
5	—	score_bit5	—
6	—	score_bit6	—
7	—	score_bit7	—

PolyTrig Digital Waveform Synthesis Core

by **Saeed Seyedfaraji**

611 HDL Project

github.com/SaeedSeyedfaraji/tt_um_polytrig_core_gf

"Compact ASIC-oriented digital waveform synthesis engine supporting LUT-based trigonometric reconstruction, waveform generation, and NCO-style signal synthesis."

How it works

PolyTrig is a TinyTapeout-compatible digital waveform synthesis core designed for compact ASIC implementation.

The design generates multiple waveform types using lookup-table (LUT) based signal reconstruction techniques and phase manipulation methods.

Supported waveform modes include:

- Sine
- Cosine
- Tangent approximation
- Cotangent approximation
- Triangle
- Sawtooth
- Square wave
- Rectified sine
- NCO-style waveform generation

The architecture uses quarter-wave LUT optimization to reduce memory usage while reconstructing complete waveforms through symmetry operations and phase transformations.

Waveform selection and runtime configuration are controlled through the TinyTapeout input interface.

How to test

The project includes a cocotb-based verification environment located in the `test` directory.

Run RTL simulation using:

```
cd test
make -B
```

Run gate-level simulation using:

```
make -B GATES=yes
```

Waveforms can be viewed using GTKWave:

```
gtkwave tb.fst tb.gtkw
```

External hardware

No external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	phase/control bit 0	waveform output bit 0	NCO waveform select bit 0
1	phase/control bit 1	waveform output bit 1	NCO waveform select bit 1
2	phase/control bit 2	waveform output bit 2	NCO amplitude select bit 0
3	phase/control bit 3	waveform output bit 3	NCO amplitude select bit 1
4	phase/control bit 4	waveform output bit 4	auxiliary/debug bit 4
5	phase/control bit 5	waveform output bit 5	auxiliary/debug bit 5
6	phase/control bit 6	waveform output bit 6	auxiliary/debug bit 6
7	phase/control bit 7	waveform output bit 7	auxiliary/debug bit 7

Detronyx Arithmetic Lab Tile

by **Detronyx contributors**

0612 | 50 MHz | HDL Project

`github.com/Zhekar1998/tt_um_arith_lab`

"Register-loaded add/multiply/divide arithmetic lab for TinyTapeout GF"

How it works

This project combines three public arithmetic blocks behind a tiny register-loaded lab interface: the byte adder, the standalone combinational 8x8 multiplier, and the reciprocal-ROM divider. It has four task slots, each with one byte in bank A and one byte in bank B. `ui_in[7:0]` carries data. `uio_in[2:0]` selects a command: `1` writes `A[bank_sel]`, `2` writes `B[bank_sel]`, `3` loads the 2-bit mode from `ui_in[1:0]`, and `4` loads `bank_sel` from `ui_in[1:0]`. Command `5` loads the configuration register. `cfg[0]` selects the add status flag: `0` reports unsigned carry and `1` reports signed overflow. Modes are: `0` add, `1` multiply low byte, `2` quotient, and `3` multiply high byte. Quotient results come from the staged Detronyx MDU byte-divide RTL path and settle after the divider pipeline has accepted the selected slot operands.

How to test

Select a task slot, write A and B, then load the desired mode. `uo_out[7:0]` shows the selected result for the active slot. `uio[7:6]` report the selected bank, `uio[5:4]` report the selected mode, and `uio[3]` is a mode-specific status flag: add carry/overflow selected by `cfg[0]`, divider divide-by-zero, or multiplier high-byte nonzero. `uio[2:0]` remain command inputs.

The cocotb regression covers add carry and signed overflow selection, multiply low/high byte outputs, multiply high-byte-nonzero status, divider quotient, and divider divide-by-zero status. The GF180MCU AS 3.3 V hardening flow uses a project SDC with explicit I/O delay, clock uncertainty/transition, fanout, transition, and capacitance constraints. KLayout DRC is enabled with the generated standalone GF180 5LM runset in `src/klayout_drc/gf180mcu_5lm_full.drc`. The KLayout checker is non-gating for now because the GF180 deck reports `CO.6a` contact/Metal1 end-of-line hits inside the PDK standard cell `gf180mcu_as_sc_mcu7t3v3__aoi211_2`; the report is still emitted for review.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data[0]	result[0]	cmd[0]
1	data[1]	result[1]	cmd[1]
2	data[2]	result[2]	cmd[2]
3	data[3]	result[3]	status
4	data[4]	result[4]	mode[0]
5	data[5]	result[5]	mode[1]
6	data[6]	result[6]	bank[0]
7	data[7]	result[7]	bank[1]

Simon Says memory game

by **Uri Shaked**

`0613` `50 kHz` `HDL Project`

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

"*Repeat the sequence of colors and sounds to win the game*"

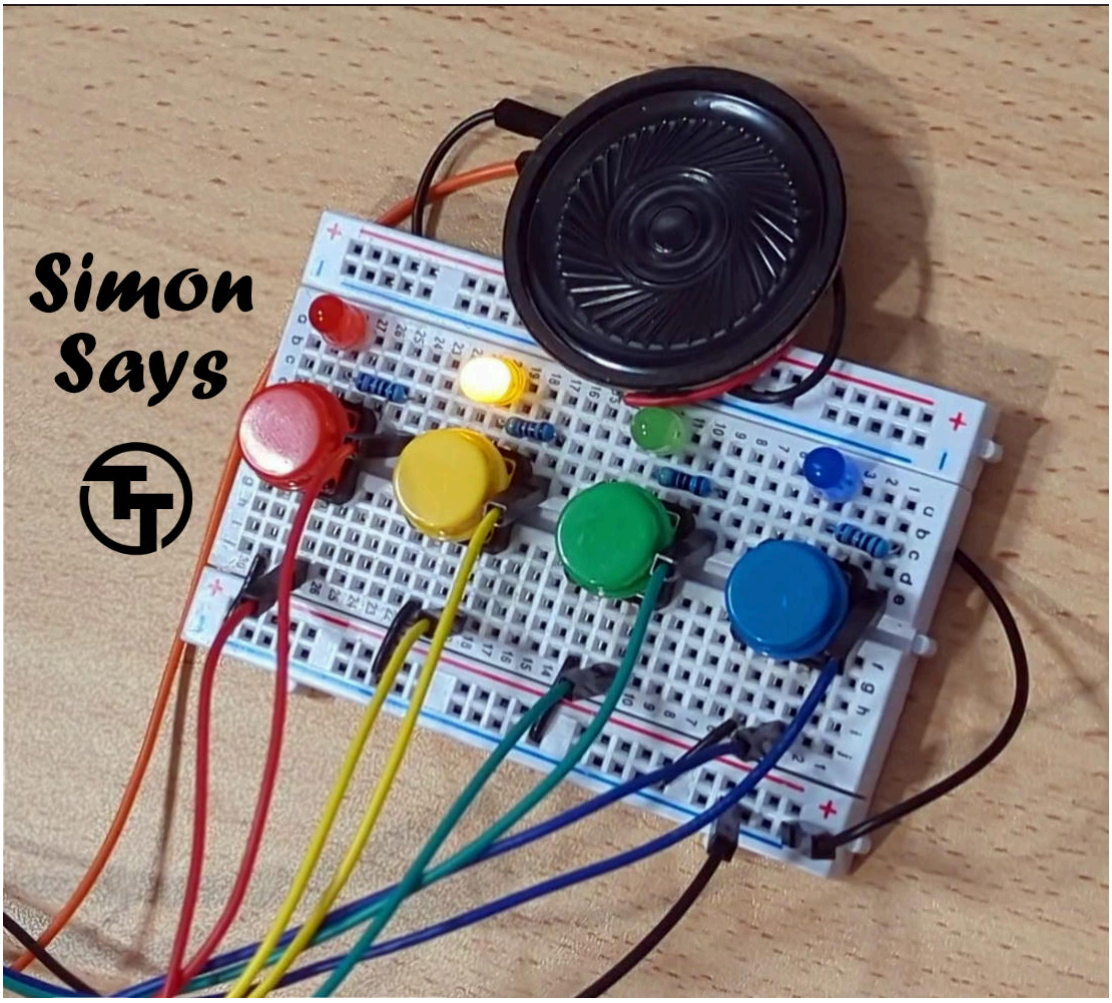


Figure 613.1: Simon Says Game

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	—

Nearest Neighbor Interpolation

by **Dan Mangum**

`0614` `25 MHz` `HDL Project`

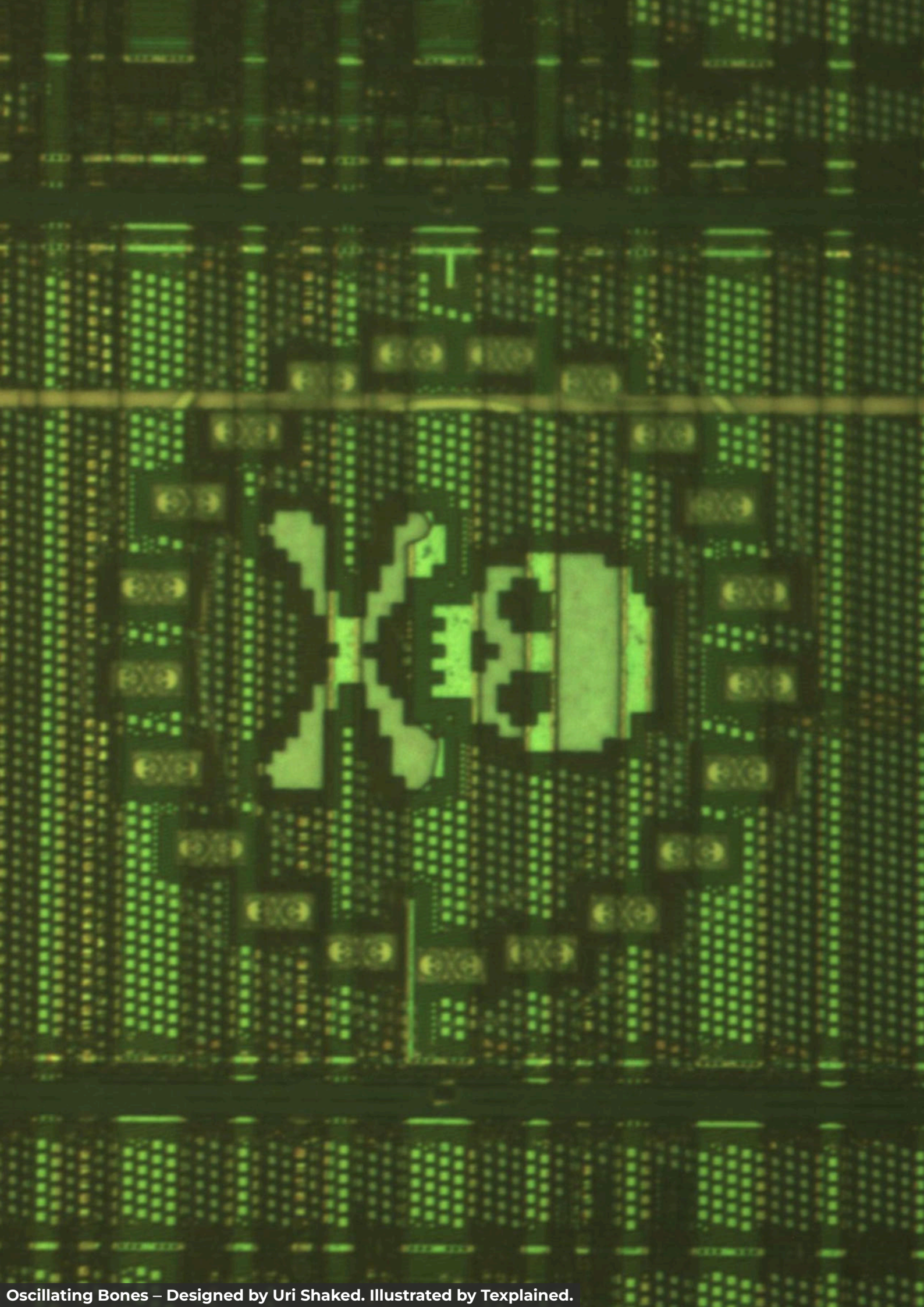
`github.com/hasheddan/nni`

"Nearest neighbor interpolation demosaicing for RGGB Bayer CFAs over UART"

Project Pinout

Digital Pins

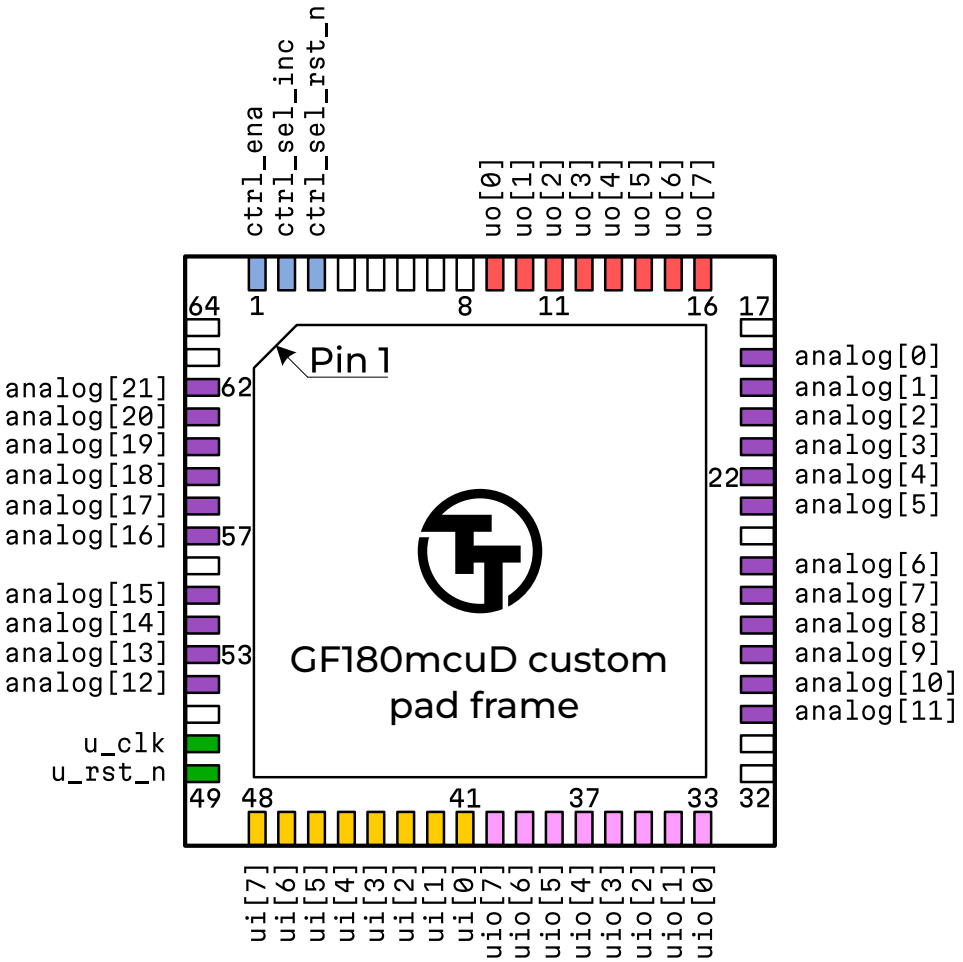
#	Input	Output	Bidirectional
0	uart_rx	uart_tx	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—



Oscillating Bones – Designed by Uri Shaked. Illustrated by Texplained.

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:

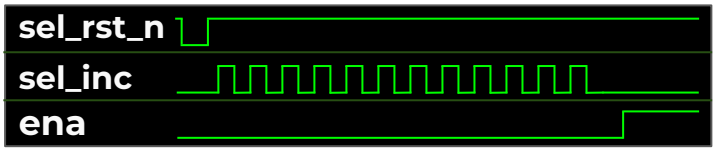


Figure 1: Mux signals for activating the design at address 12

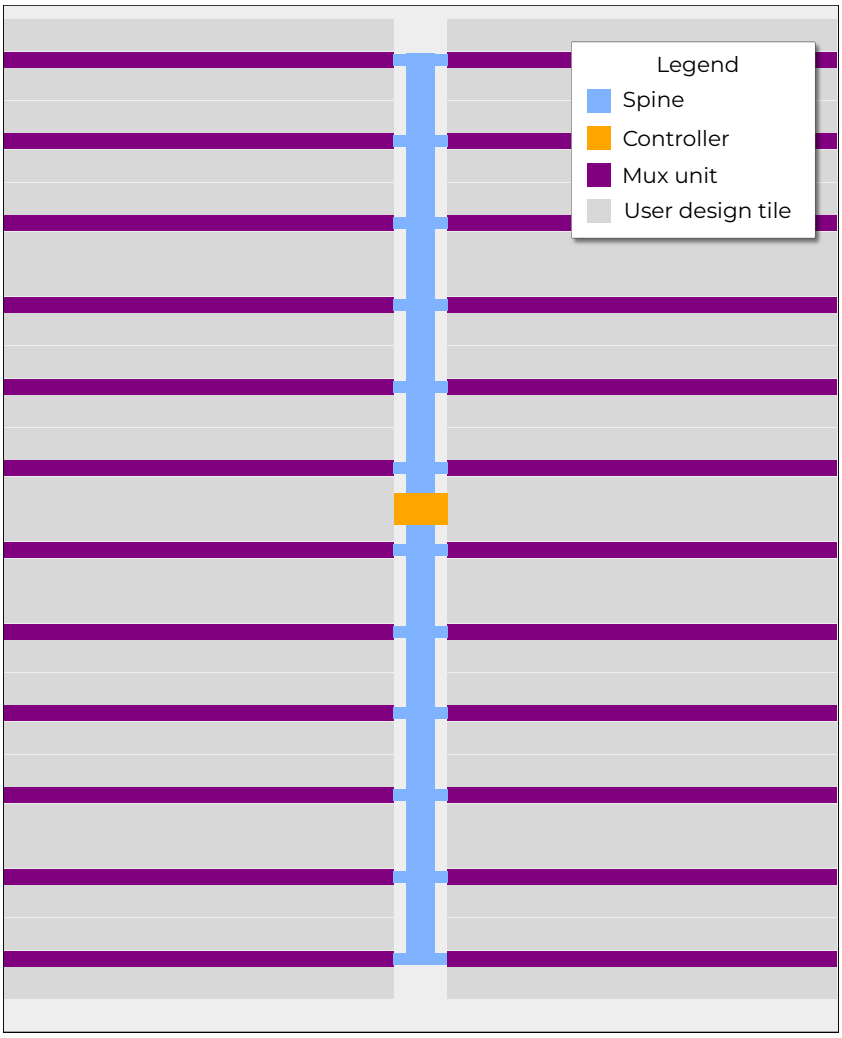


Figure 2: Mux Diagram

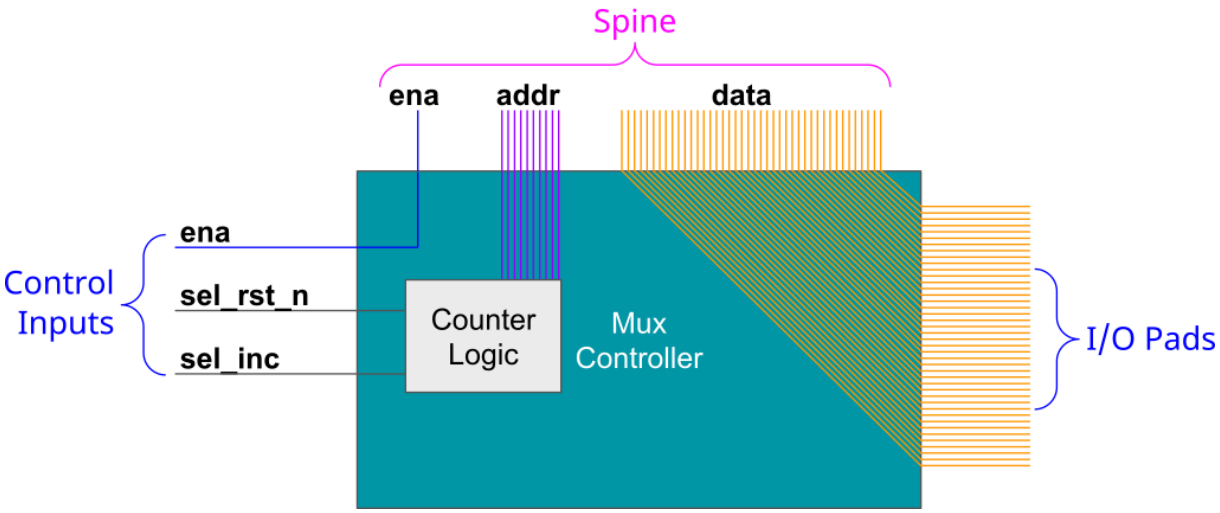


Figure 3: Mux Controller Diagram

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