Tiny Tapeout GF 26a Datasheet

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

June 22, 2026

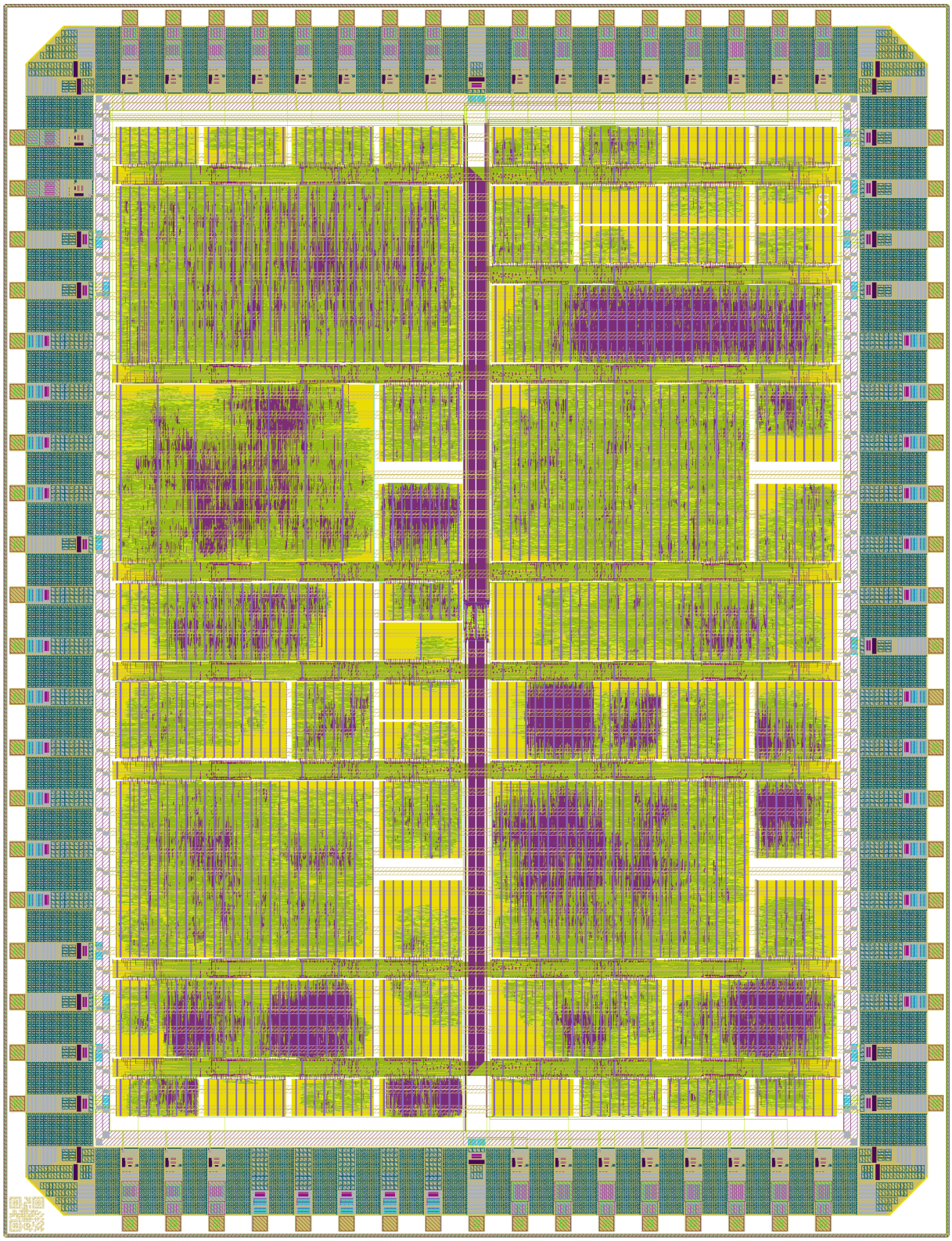
Table of Contents

Chip Renders

Online chip viewer

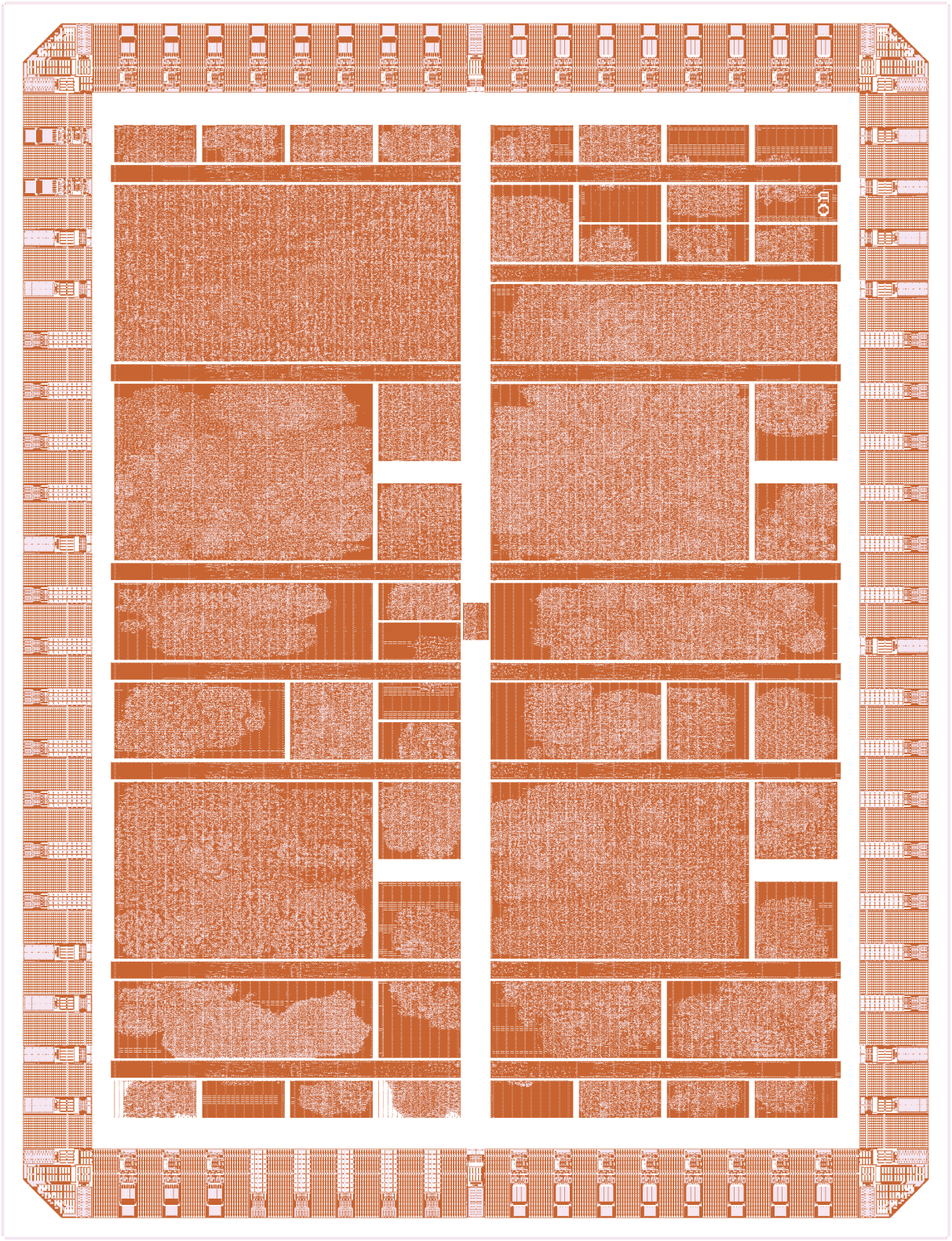
tinytapeout.com/
decap/ttgf26a

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]

Counter16Outputs

by **Alvaro Rubio**

0003 HDL Project

`github.com/AlvaroRub/tt-alvaros-first-wokwi`

"A 16-bit ring counter that sequentially activates its outputs at the clock rate."

How it works

This project implements a 16-bit "one-hot" ring counter. It consists of a circular shift register where a single high bit (1) advances sequentially across the 16 outputs.

The internal operation is as follows:

- **Initialization:** Upon activating the reset signal (the `rst_n` pin in Tiny Tapeout, which is internally inverted to an active-high `rst` in the counter's logic), the register is loaded with the initial value `16'b0000_0000_0000_0001`. This turns on only the first output.
- **Shifting:** On every rising edge of the clock signal (`clk`), the bits shift one position to the left. The most significant bit (MSB, bit 15) is fed back to the least significant position (LSB, bit 0), creating an infinite loop.
- **Output mapping:** To accommodate the 16 outputs within the Tiny Tapeout architecture, the first 8 bits (0 to 7) are assigned to the dedicated output pins (`uo_out`), and the next 8 bits (8 to 15) are assigned to the bidirectional pins (`uio_out`), which are permanently configured as outputs via `uio_oe`. ## How to test

To test the design on the physical evaluation board (or in the Wokwi/Verilator simulator):

1. **Clock Configuration:** Select a very low clock frequency (e.g., 1 Hz to 10 Hz) using the baseboard configuration or an external pulse generator. This will allow you to see the bit shifting visually.
2. **Reset:** Press the button associated with the reset signal (`rst_n` on the board is usually active-low). While the reset is held, only the first LED corresponding to `uo_out[0]` should be lit.
3. **Execution:** Release the reset button. You should observe the lit LED "traveling" sequentially from `uo_out[0]` to `uo_out[7]`, and then continuing its path through `uio_out[0]` to `uio_out[7]`. Once it reaches the last pin, the cycle starts again at `uo_out[0]`.

External hardware

No special external hardware is required if using the standard Tiny Tapeout demonstration board, as it includes a clock generator, pushbuttons for inputs, and LEDs to visualize the outputs.

If mounted on a custom breadboard, the following will be needed:

- A clock source (e.g., a 555 timer-based oscillator or a microcontroller).
- 16 LEDs connected to the `uo_out` and `uio_out` pins, along with their respective current-limiting resistors (e.g., 330 Ω to 1 kΩ).
- A pushbutton with a pull-up resistor for the reset signal.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	out_0	out_8
1	—	out_1	out_9
2	—	out_2	out_10
3	—	out_3	out_11
4	—	out_4	out_12
5	—	out_5	out_13
6	—	out_6	out_14
7	—	out_7	out_15

2x2 CNN Accelerator PE Grid with UART

by **Vidhyanandhan Sathishkumar**

0005 | 25 MHz | HDL Project

`github.com/Vidhyanandhan-01/tinytapeout-gf`

> "*UART-controlled 2x2 systolic processing-element grid for CNN inference. Send 3-byte packets over UART to drive activations/weights; receive 2-byte psum results.*"

How it works

This project implements a **2×2 systolic Processing Element (PE) grid** for CNN inference, controlled entirely over UART from a host PC.

Architecture

The core is `peg_1x` — a 2×2 array of multiply-accumulate PEs arranged in a systolic fashion. Each PE has a 2-bit signed weight register and a 13-bit accumulator. The array supports three dataflow patterns:

Direction	Encoding	Description
H-zigzag	`2'b10`	Activations enter via `h_in0`, snake left→up→left→down
V-zigzag	`2'b01`	Activations enter via `v_in0`, snake down→right→up→left
Circular	`2'b11`	Internal ring; boundary inputs ignored

The `uart_peg_top` wrapper sits on top and bridges a host PC to the PE grid over **8N1 UART at 115200 baud**.

UART Protocol

PC → FPGA: 3 bytes per DUT clock cycle

Byte	Bits	Signal
B0	[7]	`en`
B0	[6]	`move_en`
B0	[5]	`psum_shift_en`
B0	[4]	`psum_clr`
B0	[3]	`w_ld_en`
B0	[2:1]	`direction[1:0]`
B1	[7:2]	`h_in0[5:0]`

B1	[1:0]	`v_in0[5:4]`
B2	[7:4]	`v_in0[3:0]`
B2	[3:2]	`w_in[1][1:0]`
B2	[1:0]	`w_in[0][1:0]`

FPGA → PC: 2 bytes after each applied cycle

Byte	Bits	Signal
B0	[4:0]	`psum_out0[12:8]`
B1	[7:0]	`psum_out0[7:0]`

Each 3-byte packet applies the encoded inputs for exactly **one DUT clock cycle**. Transient control signals (`en`, `move_en`, `w_ld_en`, etc.) are auto-cleared after each cycle. Sticky signals (`direction`, `h_in0`, `v_in0`, `w_in`) retain their last value.

Pins

- `ui[0]` = UART RX (input from PC)
- `uo[0]` = UART TX (output to PC)

How to test

Connect a USB-to-UART adapter: adapter TX → `ui[0]`, adapter RX → `uo[0]`, GND → GND.

Install the Python host script dependencies:

```
pip install pyserial
```

Run the full verification suite against the chip:

```
python uart_peg_host.py /dev/ttyUSB0
```

This runs 9 test cases matching the simulation testbench (h-zigzag, v-zigzag, circular shift, negative weights, psum_clr, multi-stream accumulation).

Example: load weights and compute one dot product

```
from uart_peg_host import PegController

ctrl = PegController("/dev/ttyUSB0", baud=115200)
ctrl.clear_psums()
ctrl.load_weights(w0=1, w1=1)          # +1 both columns
ctrl.stream_h([5, 7], extra_drain=6)   # stream activations
result = ctrl.drain_psum()             # read accumulated psum
print(result)                          # → 12
ctrl.close()
```

External hardware

- USB-to-UART adapter (3.3V logic level, e.g. FTDI FT232 or CP2102)
- Host PC running `uart_peg_host.py` (Python 3, pyserial)

Project Pinout

Digital Pins

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

2x2 Systolic Array Matrix Multiplier

by **Irene-ux**

0007 | 50 MHz | HDL Project

github.com/Irene-ux/tt-systolic-array-2x2

"2x2 output-stationary systolic array for signed 4-bit matrix multiply on GF180"

How it works

This project implements a 2×2 output-stationary systolic array that computes signed 4-bit matrix multiplication, producing 8-bit results.

The design has 5 modules:

- **PE (Processing Element)** — bit-serial multiplier with accumulator. Each PE computes `acc += a_in × weight_in` every cycle when `en=1`.
- **array_module** — 2×2 grid of 4 PEs with input skew registers. `a_row1` and `weight_col1` are delayed by 1 cycle to align data correctly across the array.
- **FSM** — controls the computation sequence: IDLE → COMPUTE (2 cycles) → DRAIN (2 cycles) → READ (4 cycles) → IDLE. Generates `en`, `clear`, and `valid` signals automatically after `start` is pulsed.
- **output_serializer** — serializes the 4 accumulator results (`acc_00`, `acc_01`, `acc_10`, `acc_11`) onto `data_out` one per cycle when `valid=1`.
- **tt_um_systolic_mac_2x2** — TT wrapper with streaming counter. Loads two sets of input registers (cycle 1 and cycle 2 values) via `ui_in` and `uio_in`, then auto-streams them to the array after `start`.

Input loading protocol

Load all 4 register pairs in 4 cycles using `load=1` and `sel`:

sel	ui_in[7:4] (data_c1)	uio_in[7:4] (data_c2)
00	A[0][0]	A[0][1]
01	A[1][0]	A[1][1]
10	B[0][0]	B[1][0]
11	B[0][1]	B[1][1]

After loading, pulse `start=1` for 1 cycle. The wrapper streams cycle 1 values, then cycle 2 values, then zeros automatically. Results appear on `uo_out` one per cycle when `uio_out[0]` (valid) goes high.

Matrix multiply

Computes C = A × B where:

- A and B are 2×2 matrices of signed 4-bit integers (-8 to 7)
- C elements are signed 8-bit integers (-128 to 127)
- Output order: C[0][0], C[0][1], C[1][0], C[1][1]

How to test

1. Reset the design (`rst_n=0` for 5 cycles, then `rst_n=1`)
2. Load matrix A and B values using `load=1`, `sel`, `ui_in[7:4]`, `uio_in[7:4]`
3. Pulse `start=1` for 1 cycle
4. Wait for `uio_out[0]` (valid) to go high
5. Read 4 consecutive values from `uo_out`

Example — identity matrix test

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	start	data_out[0]	valid
1	load	data_out[1]	—
2	sel[0]	data_out[2]	—
3	sel[1]	data_out[3]	—
4	data_c1[0]	data_out[4]	data_c2[0]
5	data_c1[1]	data_out[5]	data_c2[1]
6	data_c1[2]	data_out[6]	data_c2[2]
7	data_c1[3]	data_out[7]	data_c2[3]

Hardware UTF Encoder/Decoder

by **Rebecca G. Bettencourt**

0032 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

2x2 MAC Systolic array with DFT

by **Vijay Sharma**

0034 50 MHz HDL Project

`github.com/Eulerianial/tt_gf_systolic_array`

"2x2 multiply and accumulate systolic array supporting signed 8 bit values with design for test infrastructure."

Systolic Array with DFT

Multiply and accumulate matrix multiplier ASIC with DFT(design-for-test) infrastructure

The design is orginially forked from https://github.com/Essenceia/Systolic_MAC_with_DFT

It is an ASIC design for a 2x2 systolic matrix multiplier supporting multiply and accumulate operations on int8 data alongside a design for test infrastructure to help debug both usage and diagnose design issues in silicon.

Pinout

This accelerator uses the following pinout:

ui (Inputs)	uo (Outputs)	uio (Bidirectional)
ui[0] = tck	uo[0] = result_o	uio[0] = data_i[7]
ui[1] = data_i[0]	uo[1] = result_o	uio[1] = data_valid_i
ui[2] = data_i[1]	uo[2] = result_o	uio[2] = data_mode_i
ui[3] = data_i[2]	uo[3] = result_o	uio[3] = data_rst_addr_i
ui[4] = data_i[3]	uo[4] = result_o	uio[4] = tdi
ui[5] = data_i[4]	uo[5] = result_o	uio[5] = tms
ui[6] = data_i[5]	uo[6] = result_o	uio[6] = tdo
ui[7] = data_i[6]	uo[7] = result_o	uio[7] = result_v_o

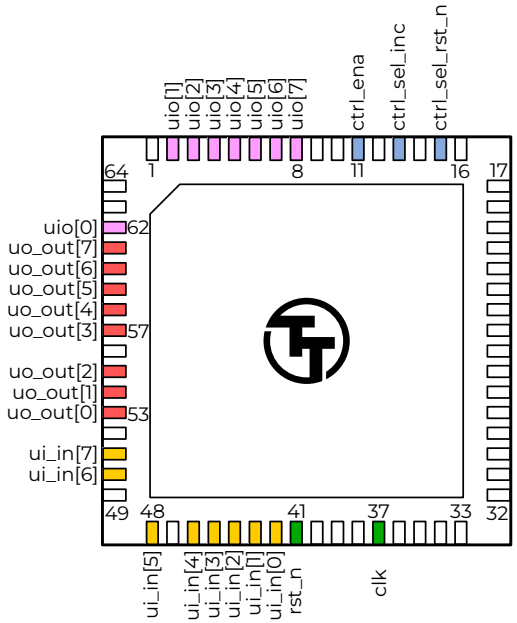


Figure 34.1: Chip pinout

MAC

This MAC accelerator operates at up to 50MHz and is capable of reaching up to 100 MMAC/s or 200 MIOPS/s.

Background

The goal of the MAC accelerator is to perform a matrix matrix multiplication between the input data matrix I and the weight matrix W.

```
\begin{gather}
I \times W = R \\
\begin{pmatrix}
i_{0,0} & i_{1,0} \\
 i_{0,1} & i_{1,1}
\end{pmatrix}

\times

\begin{pmatrix}
w_{0,0} & w_{1,0} \\
w_{0,1} & w_{1,1}
\end{pmatrix} =

\begin{pmatrix}
i_{0,0}w_{0,0}+i_{1,0}w_{0,1} & i_{0,0}w_{1,0}+i_{1,0}w_{1,1}\\
i_{0,1}w_{0,0}+i_{1,1}w_{0,1} & i_{0,1}w_{1,0}+i_{1,1}
w_{1,1}\end{pmatrix}

=
```

```
\begin{pmatrix}
r_{0,0} & r_{1,0} \\
r_{0,1} & r_{1,1}
\end{pmatrix}
\end{gather}
```

This MAC accelerator has 4 units and from this point on, we will refer to each MAC unit according to their unique (x, y) coordinates.

Each MAC unit calculates the MAC operation $c_{(t,x,y)}$, where :

- $w_{(x,y)}$ is the fixed weight configured for this unit; this value is fixed throughout a set of I and W input matrices.
- $i_{(t,y)}$ is a value from the y row of the I matrix that is circulated per timestep t through a row of the matrix.
- $c_{(t-1,x,y-1)}$ is the result at the previous timestep $t-1$ of the MAC unit above this MAC unit, circulated downwards per column.

```
c_{(t,x,y)} = i_{(t,y)} \times w_{(x,y)} + c_{(t-1,x,y-1)}
```

Given this accelerator was designed to operate on signed 8-bit integers, but that the successive application of the 8-bit multiplication and addition pushes the resulting value up to 17 bits, in order to prevent the size of the base datatype from increasing with each successive MAC operation, we need to clamp it down back within the 8-bit range.

As such, the MAC unit performs an additional clamping function $clamp_{i8}$ that remaps :

```
clamp_{i8}(c_{(t,x,y)}) = \begin{cases}
   127 &\text{if } c_{(t,x,y}) > 127\\
   c_{(t,x,y)} &\text{if } c_{(t,x,y)} \in [-128,127] \\
    -128 &\text{if } c_{(t,x,y}) < -128\\
\end{cases}
```

Our final full MAC operation is as follows :

```
c_{(t,x,y)} = clamp_{i8}(i_{(t,y)} \times w_{(x,y)} +
c_{(t-1,x,y-1)})
```

At each MAC timestep $t+1$:

- the result of a MAC unit $c_{(t,x,y)}$ is shifted downwards on the same column and becomes the input of the MAC unit $(x, y+1)$ below.
- $i_{(t,x)}$ is shifted rightwards and used as input to MAC unit $(x+1, y)$.

This data streaming allows such designs to make more efficient use of data, re-using it multiple times as the data circulates through the array, contributing to the final results without spending time on expensive data accesses, allowing us to dedicate more of our silicon area and cycles to compute.

Throughput

Assuming a pre-configured W weight matrix is being reused and the accelerator is receiving a gapless stream of multiple I input matrices, this MAC accelerator is capable of computing up to 100 MMAC/s or 200 MIOPS/s.

IO Bottleneck

Accelerator operations are stalled if a MAC operation has a data dependency on data that has yet to arrive. For example, calculating $r_{(0,0)}$ depends on both $i_{(0,0)}$ and $i_{(1,0)}$. In practice, each operation depends on two pieces of input data, yet our input interface being only 8 bits wide allows us to transfer only a single $i_{(x,y)}$ per cycle.

This limitation means our accelerator is actually operating at half maximum capacity due to this IO bottleneck. If the IO interface were either (a) at least 16 bits wide, or (b) 8 bits wide but operating at 100 MHz, resolving this bottleneck, our maximum throughput would be 200 MMAC/s or 400 MIOPS/s.

Usage

The typical sequence to offload matrix operations to the accelerator would go as follows:

1. Reset the accelerator (necessary on init)
2. Configure the weights W (can be re-used once configured)
3. Send the input data I
4. Read the result R

This design doesn't feature on-chip SRAM and has limited on-chip memory. Given weights have high spatial and temporal locality, this design allows each weight to be configured per MAC unit. This configuration can be reused across multiple matrices. The input matrix, on the other hand, is expected to be provided on each usage.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	tck	result_o	data_i[7]
1	data_i[0]	result_o	data_valid_i
2	data_i[1]	result_o	data_mode_i
3	data_i[2]	result_o	data_rst_addr_i
4	data_i[3]	result_o	tdi
5	data_i[4]	result_o	tms
6	data_i[5]	result_o	tdo

#	Input	Output	Bidirectional
7	data_i[6]	result_o	result_v_o

FSK Modem

by **Vaishnavi V**

0036 | 50 MHz | HDL Project

`github.com/vvedn/hydrocomms-tt`

"FSK modem for underwater acoustic communication (46-56 kHz, 100 bps)"

How it works

This is a digital binary FSK modem designed for underwater acoustic communication in the 46-56 kHz band.

Data bytes are framed as the following:

[preamble sync word payload CRC]

Then each bit selects between two frequencies — 48 kHz (mark/1) and 54 kHz (space/0).

Similar to the DDS algorithm, a numerically controlled oscillator (NCO) uses a 16-bit phase accumulator with a 32-entry sine lookup table to generate 8-bit samples at 200 kHz.

There is also a CRC-8 for error detection on the 8-bit payload.

On the receiving side, 8-bit ADC samples feed two parallel Goertzel filters at the chosen mark and space frequencies (48 and 54 kHz). A frame synchronizer detects the preamble/sync pattern, extracts the payload byte, and verifies the CRC.

SPI is the protocol used to interface with the design.

There is a loopback mode to test the chip on its own, which is standard.

How to test

All 8 output pins expose these signals for hardware bring-up:

- uo_out[1] RX_BYTE_VALID - Pulses once per decoded byte
- uo_out[2] CRC_OK - High = good frame, low = corrupt
- uo_out[3] PACKET_DETECTED - Goes high when sync word found
- uo_out[4] TX_ACTIVE - High while transmitting
- uo_out[5] SYMBOL_CLK - 100 Hz baud tick during TX
- uo_out[6] MARK_GT_SPACE - Real-time Goertzel comparison
- uo_out[7] SAMPLE_CLK - 200 kHz ADC/DAC sample clock

These can be monitored with a logic analyzer or oscilloscope during bench testing.

To write a register, set bit 7 of the address byte. To read, send the plain address followed by a dummy byte, the register value comes back on MISO during the dummy byte.

Note: the link is half-duplex — a board cannot receive while it is transmitting, since TX output and RX input share the same 8 pins. Note also that the receiver has no symbol-timing recovery: the Goertzel blocks free-run, so board-to-board reception requires the block phase to land close to the symbol boundaries.

Expect to need multiple frames (or a hardware retry loop) for the first sync in two-board tests.

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	SPI_SCLK	SPI_MISO	—
1	SPI_MOSI	RX_BYTE_VALID	—
2	SPI_CS_N	CRC_OK	—
3	—	PACKET_DETECTED	—
4	—	TX_ACTIVE	—
5	—	SYMBOL_CLK	—
6	—	MARK_GT_SPACE	—
7	—	SAMPLE_CLK	—

RRAM Characterization Platform (DC sweep + endurance + retention + histogram, GF180)

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

0038 25 MHz HDL Project

github.com/SanthoshSivasubramani/tt_um_santhosh_rsd_char_gf_pub

> *"GF180mcuD standalone characterization platform for one external resistive-switching device (RRAM / memristor / PCM). Drives an 8-bit DAC rail and reads back an 8-bit ADC rail, with four SPI-selectable modes: DC sweep (VSTART..VSTOP by VSTEP with on-chip 16-bin histogram and min/max tracking), pulse endurance (N cycles of SET/read-verify/RESET/read-verify with THR_LO / THR_HI checks, latches FAIL_IDX on first miss), retention (SET pulse then periodic reads at programmable interval), and free-running histogram capture. 25 MHz signoff clock."*

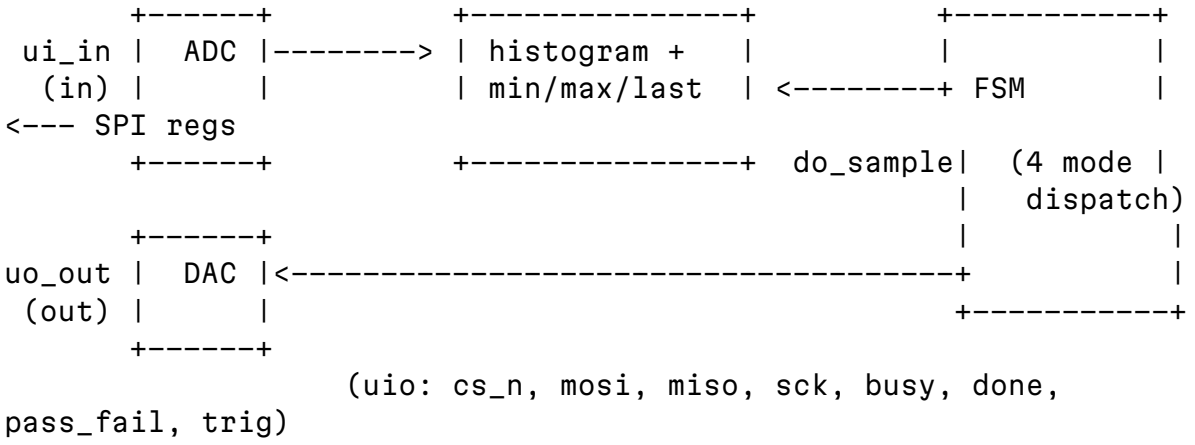
A **standalone characterization platform** for a single external resistive-switching device — RRAM / memristor / phase-change cell — implemented entirely in gf180mcuD digital. The host programs one set of parameters, asserts `start`, and the on-chip FSM runs the chosen experiment to completion, writing results into readable registers so the host never has to micro-manage pulses.

Four modes

mode	Name	What it does
00	DC sweep	DAC ramps VSTART → VSTOP by VSTEP with a PW-cycle settle at each step; every sample updates histogram, min, max, last.
01	Endurance	N cycles of {SET pulse → read-verify vs THR_LO, RESET pulse → read-verify vs THR_HI}. On the first miss, FAIL_IDX and PASS_FAIL latch and the FSM stops early.
10	Retention	Apply one SET pulse, then read every INTV clocks for N reads at VREAD, updating histogram / min / max / last each time.
11	Hist-only	Free-running ADC capture: sample ui_in every PW clocks for N samples without touching the DAC.

All four modes write into a shared 16-bin histogram (bins indexed by `adc[7:4]`, 8-bit saturating counters) and a min / max / last tracker.

Block diagram



Register map

Addr	Name	Description
`0x00`	`R_CTRL`	`{irq_en, –, clear_hist, mode[1:0], reset_sticky, start, global_en}`
`0x01`	`R_VSTART`	DC-sweep start DAC code (default `0x00`)
`0x02`	`R_VSTOP`	DC-sweep stop DAC code (default `0xFF`)
`0x03`	`R_VSTEP`	DC-sweep step (default `0x01`)
`0x04`	`R_VSET`	SET-pulse amplitude (default `0xC0`)
`0x05`	`R_VRESET`	RESET-pulse amplitude (default `0x40`)
`0x06`	`R_VREAD`	READ amplitude (default `0x80`)
`0x07`	`R_THR_LO`	verify threshold after SET (ADC must be ≥ `THR_LO`) (default `0xA0`)
`0x08`	`R_THR_HI`	verify threshold after RESET (ADC must be ≤ `THR_HI`) (default `0x60`)
`0x09`	`R_PW`	pulse / settle width in clocks (default `0x04`)
`0x0A`	`R_N_LO`	low byte of N (endurance cycles / retention reads / hist samples)
`0x0B`	`R_N_HI`	high byte of N (max `0xFFFF` = 65 535)
`0x0C`	`R_INTV_LO`	retention interval low byte
`0x0D`	`R_INTV_HI`	retention interval high byte
`0x10..0x1F`	`R_HIST0..15`	16-bin 8-bit saturating histogram
`0x20`	`R_STATUS`	`{busy, done_sticky, pass_fail, irq_sticky, –, mode[1:0], global_en}`

0x21/22	R_FAIL_LO/HI	cycle index where endurance first failed
0x23	R_LAST_ADC	most-recent sampled ADC value
0x24	R_MIN_ADC	minimum ADC observed since start
0x25	R_MAX_ADC	maximum ADC observed since start
0x26	R_DAC_DBG	current DAC drive value (read-only)

W1C on R_STATUS: write 1 << 6 to clear done_sticky; write 1 << 4 to clear irq_sticky. pass_fail is cleared by writing CTRL.reset_sticky=1.

Pinout

- ui_in[7:0] — ADC readback from the external cell
- uo_out[7:0] — DAC drive to the external cell
- uio[0] — spi_cs_n (in)
- uio[1] — spi_mosi (in)
- uio[2] — spi_miso (out)
- uio[3] — spi_sck (in)
- uio[4] — busy
- uio[5] — done_sticky
- uio[6] — pass_fail (endurance)
- uio[7] — trigger_out — high during SET/RESET pulse phases in DC / endurance / retention modes, useful as a scope trigger.

How to test

```
# Endurance: 100 cycles at default SET/RESET/READ/thresholds, PW=4
spi_write(R_PW, 4)
spi_write(R_N_LO, 100); spi_write(R_N_HI, 0)
spi_write(R_CTRL, 0x0B)           # global_en=1, start=1, mode=01
wait(status.done == 1)
if status.pass_fail:
    fail_cycle = R_FAIL_HI << 8 | R_FAIL_LO
else:
    fail_cycle = None
```

External hardware

An external RRAM/memristor test cell with a byte-granularity DAC drive and a byte-granularity ADC readback. The on-chip platform does not assume any particular voltage / current scaling — the 8-bit codes on uo_out map linearly to whatever analog rail the carrier board supplies, and ui_in is whatever the carrier board samples from the cell.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	adc_in[0] (external device readback, LSB)	dac_out[0] (DAC drive to external device, LSB)	spi_cs_n (in)
1	adc_in[1]	dac_out[1]	spi_mosi (in)
2	adc_in[2]	dac_out[2]	spi_miso (out)
3	adc_in[3]	dac_out[3]	spi_sck (in)
4	adc_in[4]	dac_out[4]	busy (out)
5	adc_in[5]	dac_out[5]	done_sticky (out)
6	adc_in[6]	dac_out[6]	pass_fail (out, endurance)
7	adc_in[7] (MSB)	dac_out[7] (MSB)	trigger_out (out, high during SET/RESET/ sample phases)

Programmable_Pipeline-RISC-V

by **Prem**

0039 | 25 MHz | HDL Project

`github.com/PrmBRana/ttgf_prem_programmable_RISC_v`

"Zero to ASIC demo project"

How it works

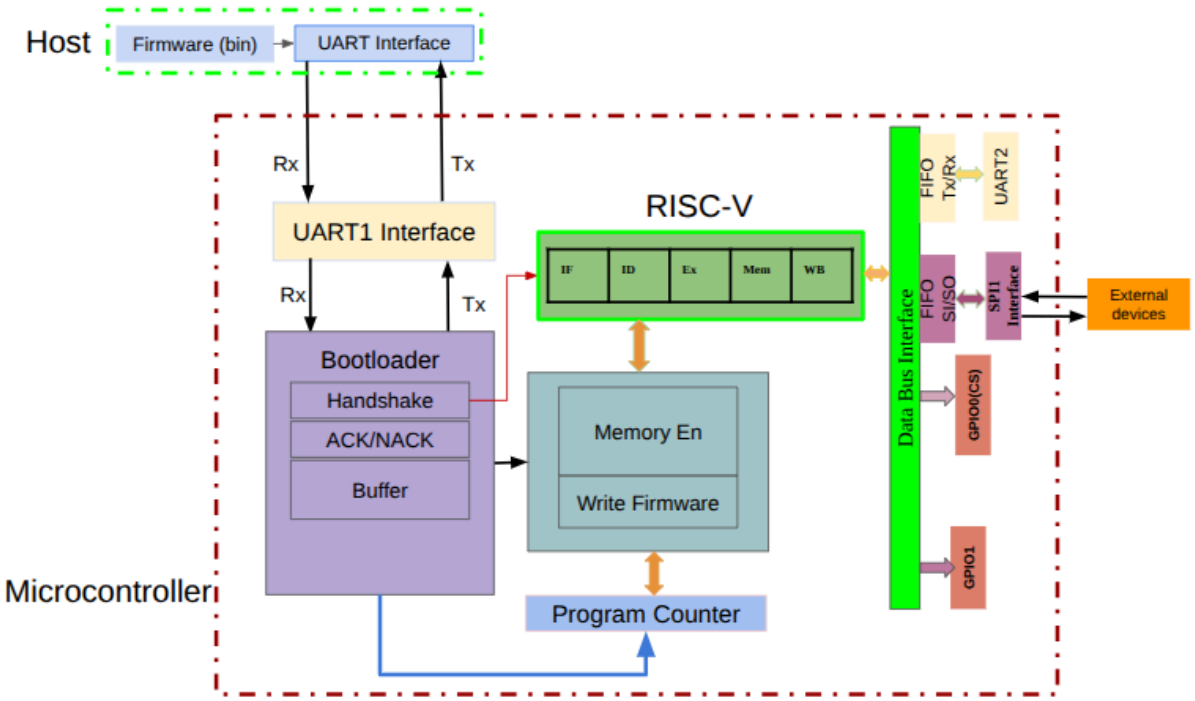


Figure 39.1: Block Diagram

This project implements a compact 32-bit RISC-V processor with a five-stage pipeline architecture consisting of Instruction Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), and Write-Back (WB) stages. The pipelined design allows multiple instructions to be processed concurrently, improving throughput while maintaining a small hardware footprint.

Peripheral Interfaces

The system includes the following communication and control interfaces:

1. **UART1:** Bootloader interface for program loading via serial protocol
2. **UART2:** General-purpose UART communication interface during execution
3. **SPI Master:** Peripheral communication interface (Mode 0, 4.17 MHz)
4. **GPIO1:** General-purpose output (LED control or similar)
5. **GPIO2:** Hardware Chip Select control for SPI slave peripherals

Operating Modes

Bootloader Mode (Reset): After reset, the processor enters bootloader mode via UART1. Instructions are received serially as bytes through the UART1 RX pin and stored sequentially into instruction memory. Once the bootloader detects the sentinel value (`0xBAADF00D`), it automatically transitions to execution mode.

Execution Mode: The processor fetches and executes instructions through the five-stage pipeline. Peripheral access is controlled via memory-mapped I/O registers.

Clock and Timing Specifications

- **System Clock:** 25 MHz
- **UART Baud Rate:** 115,200 baud with x16 oversampling (both UART1 and UART2)
- **SPI Clock Frequency:** 4.17 MHz (CLK_DIV = 3, Mode 0: CPOL = 0, CPHA = 0)

Peripheral Communication

Peripherals are accessed through memory-mapped I/O registers using a **polling-based architecture**. The CPU continuously reads status registers to determine when operations can proceed, rather than using interrupts. This simplifies the design while maintaining deterministic behavior.

The SPI master interface supports full-duplex communication with external peripherals such as sensors or external memory devices via dedicated signals (MOSI, MISO, SCLK, and CS).

Peripheral Address Map & Polling Schemes

1. UART2 Control Interface

Memory-mapped registers for UART2 (general-purpose communication):

Register	Address	Description
`UART_TX`	`0x1000_0000`	Write data byte to transmit
`UART_RX`	`0x1000_0004`	Read received data byte
`UART_TX_STATUS`	`0x1000_0008`	TX buffer status (Bit 0: 1 = ready/empty, 0 = busy)
`UART_RX_STATUS`	`0x1000_000C`	RX buffer status (Bit 0: 1 = data available, 0 = empty)

Polling Protocol:

- **TX:** Poll `UART_TX_STATUS` Bit 0 until high, then write to `UART_TX`

- **RX:** Poll `UART_RX_STATUS` Bit 0 until high, then read from `UART_RX`

2. SPI Master Control Interface

Memory-mapped registers for SPI master mode (Mode 0):

Register	Address	Description
`SPI_TX`	`0x4000_0000`	Write data byte to transmit
`SPI_RX`	`0x4000_0008`	Read received data byte
`SPI_TX_STATUS`	`0x4000_0004`	TX status (Bit 0: 1 = idle/ready, 0 = shifting)
`SPI_RX_STATUS`	`0x4000_000C`	RX status (Bit 0: 1 = data available, 0 = empty)

Polling Protocol:

- **TX:** Poll `SPI_TX_STATUS` Bit 0 until high, then write to `SPI_TX` to initiate transfer
- **RX:** Poll `SPI_RX_STATUS` Bit 0 until high, then read from `SPI_RX`
- **Timing:** SPI transfers are full-duplex; RX data becomes available after the TX completes

3. GPIO Configuration

Memory-mapped GPIO output registers:

Register	Address	Description
`GPIO1`	`0x3000_0000`	General-purpose output (LED or other peripheral control)
`GPIO2`	`0x3000_0004`	SPI Chip Select control (typically active-low)

How to test

The design can be tested using both simulation and hardware deployment.

Simulation Environment

Use Icarus Verilog and cocotb for functional verification:

1. **Compilation:** Standard testbench compiles the Verilog design with timing parameters
2. **Stimulus:** Apply clock and reset signals; execute functional test sequences
3. **Waveform Analysis:** Use GTKWave to inspect signal transitions, pipeline activity, and register updates

UART Testing (Bootloading)

1. **Program Loading:** Use UART1 to send a sequence of 32-bit instructions serialized as bytes
2. **Acknowledgment:** Verify the bootloader correctly stores instructions in instruction memory
3. **Sentinel Detection:** Once the sentinel value (`0xBAADF00D`) is received, the bootloader halts and execution begins
4. **Execution Verification:** Monitor pipeline progression, register state changes, and instruction completions

SPI Testing

1. **External Driver:** An external SPI master drives SCLK, MOSI, and CS signals
2. **Data Observation:** Monitor MISO output from the processor
3. **Timing Verification:** Confirm Mode 0 timing compliance (CPOL=0, CPHA=0)
4. **Data Integrity:** Compare transmitted and received byte streams

Functional Verification Checklist

- ☐ **Bootloader Integrity:** Confirm UART1 correctly receives binary instruction bytes, populates instruction memory, detects sentinel value `0xBAADF00D`, and transitions to execution mode cleanly
- ☐ **Pipeline Progression:** Verify concurrent execution across IF/ID/EX/MEM/WB stages; confirm dependencies, branches, and register write-backs resolve without hazards
- ☐ **Memory-Mapped I/O Polling:** Validate that the processor correctly handles status polling for UART2 and SPI without deadlocks
- ☐ **SPI-to-UART Relay:** Test the interlock routine under load conditions; ensure no data is dropped and transfers complete reliably

---

Sample Verification Program: SPI-to-UART Polling Loop

The following assembly routine demonstrates a practical I/O sequence:

- Assert SPI Chip Select (GPIO2)
- Poll and write dummy data (`0xAA`) to SPI
- Poll and read the SPI response byte
- Poll and relay the byte over UART2 to the host
- Repeat for 200 iterations, then release Chip Select

```
#
================================================================================
# RISC-V MMIO SPI Polling and UART2 Relay Loop
# Target: 32-Bit Pipelined Core (RV32I)
```

```
# SPI to UART test
#
================================================================================

    # --- Setup Base Address Pointers ---
    lui   x10, 0x40000        # x10 = SPI Base Address
(0x4000_0000)
    lui   x18, 0x30000        # x18 = GPIO Base Address
(0x3000_0000)
    lui   x19, 0x10000        # x19 = UART Base Address
(0x1000_0000)

    # --- Calculate Register Offsets ---
    addi  x11, x10, 8         # x11 = SPI_RX Data Offset
(0x4000_0008)
    addi  x12, x10, 4         # x12 = SPI_TX Status Offset
(0x4000_0004)
    addi  x13, x10, 12        # x13 = SPI_RX Status Offset
(0x4000_000C)
    addi  x21, x19, 8         # x21 = UART_TX Status Offset
(0x1000_0008)
    addi  x14, x18, 4         # x14 = GPIO2 CS Offset
(0x3000_0004)

    # --- Initialize Loop Constants ---
    addi  x3,  x0, 1          # x3  = GPIO2 HIGH (CS de-assert)
    addi  x16, x0, 0          # x16 = GPIO2 LOW (CS assert)
    addi  x5,  x0, 200        # x5  = Transfer counter (200
iterations)
    addi  x22, x0, 1          # x22 = Status bit mask (Bit 0)
    addi  x7,  x0, 0xAA       # x7  = Dummy data byte

    # --- Assert SPI Chip Select ---
    sw    x16, 0(x14)         # GPIO2 = 0 (CS asserted to
peripheral)

    loop:
        beq   x5, x0, release_cs   # If counter == 0, exit loop

    # --- Wait for SPI TX Ready ---
    wait_spi_tx:
        lw    x6, 0(x12)          # Load SPI_TX_STATUS
        and   x6, x6, x22         # Isolate Bit 0 (ready flag)
        beq   x6, x0, wait_spi_tx # If 0 (busy), loop back
        sw    x7, 0(x10)          # Write 0xAA to SPI_TX, initiate
transfer

    # --- Wait for SPI RX Data ---
    wait_spi_rx:
        lw    x8, 0(x13)          # Load SPI_RX_STATUS
```

```
        and   x8, x8, x22        # Isolate Bit 0 (data available
flag)
        beq   x8, x0, wait_spi_rx # If 0 (no data), loop back
        lw    x9, 0(x11)         # Read received byte from SPI_RX

    # --- Wait for UART TX Ready ---
    wait_uart_tx:
        lw    x6, 0(x21)         # Load UART_TX_STATUS
        and   x6, x6, x22        # Isolate Bit 0 (ready flag)
        beq   x6, x0, wait_uart_tx # If 0 (busy), loop back
        sw    x9, 0(x19)         # Write SPI byte to UART_TX

    # --- Next Iteration ---
        addi  x5, x5, -1         # Decrement counter
        jal   x0, loop           # Jump to next iteration

    release_cs:
        sw    x3, 0(x14)         # GPIO2 = 1 (CS de-asserted from
peripheral)
        ecall                    # Halt simulation
```

Compiled Machine Code (SPI to UART)

Corresponding 32-bit hexadecimal instruction sequence: (link:https://riscvasm.lucasteske.dev/#)

```
0x40000537,  # lui   x10, 0x40000
0x30000937,  # lui   x18, 0x30000
0x100009b7,  # lui   x19, 0x10000
0x00850593,  # addi  x11, x10, 8
0x00450613,  # addi  x12, x10, 4
0x00c50693,  # addi  x13, x10, 12
0x00898a93,  # addi  x21, x19, 8
0x00490713,  # addi  x14, x18, 4
0x00100193,  # addi  x3,  x0, 1
0x00000813,  # addi  x16, x0, 0
0x03200293,  # addi  x5,  x0, 200
0x00300b13,  # addi  x22, x0, 1
0x0aa00393,  # addi  x7,  x0, 0xAA
0x01072023,  # sw    x16, 0(x14)
0x02028663,  # beq   x5, x0, release_cs
0x00752023,  # sw    x22, 0(x12) [wait_spi_tx start - poll]
0x0006a403,  # lw    x6, 0(x12)
0xfe040ee3,  # beq   x6, x0, wait_spi_tx
0x0005a483,  # lw    x8, 0(x10)
0x000aa303,  # lw    x3, 0(x13)
0x01637333,  # and   x6, x6, x22
0xfe031ce3,  # beq   x6, x0, wait_spi_rx
0x0099a023,  # sw    x9, 0(x11)
0xfff28293,  # addi  x5, x5, -1
0xfd9ff06f,  # jal   x0, loop
```

```
0x00372023,  # sw    x3, 0(x14) [release_cs]
0x00000073,  # ecall
0xBAADF00D   # SENTINEL VALUE (bootloader halt marker)
#
# ============================================================================
# RISC-V MMIO SPI Polling and UART2 Relay Loop
# Target: 32-Bit Pipelined Core (RV32I)
#GPIO test
#
# ============================================================================

    lui   x13, 0x30000        /* GPIO1 base = 0x30000000 GPIO1*/
    addi  x14, x13, 4         /* GPIO2 = 0x30000004 GPIO2*/

    addi  x5, x0, 10          /* loop counter */

    addi  x6, x0, 1           /* HIGH */
    addi  x7, x0, 0           /* LOW */

loop:

    /* BOTH OFF */
    sw    x7, 0(x13)
    sw    x7, 0(x14)

    /* delay */
    addi  x3, x0, 1000
delay1:
    addi  x3, x3, -1
    bne   x3, x0, delay1

    /* BOTH ON */
    sw    x6, 0(x13)
    sw    x6, 0(x14)

    /* delay */
    addi  x3, x0, 1000
delay2:
    addi  x3, x3, -1
    bne   x3, x0, delay2

    addi  x5, x5, -1
    bne   x5, x0, loop

    /* OFF both */
    sw    x7, 0(x13)
    sw    x7, 0(x14)

    ecall
```

Compiled Machine Code (GPIO test)

```
0x300006B7
0x00468713
0x00A00293
0x00100313
0x00000393
0x0076A023
0x00772023
0x3E800193
0xFFF18193
0xFE019EE3
0x0066A023
0x00672023
0x3E800193
0xFFF18193
0xFE019EE3
0xFFF28293
0xFA029EE3
0x0076A023
0x00772023
0x00000073,  # ecall
0xBAADF00D   # SENTINEL VALUE (bootloader halt marker)
```

Bootloader Operation: When the bootloader receives the sentinel value `0xBAADF00D`, it:

1. Stops accepting instruction bytes
2. Clears the bootloader hardware state
3. Enables the execution pipeline
4. The processor begins fetching from address 0x0000_0000

External Hardware Requirements

Standard external signals required for operation:

1. **Clock Input:** 25 MHz reference clock
2. **Reset Signal:** Active-low asynchronous reset (`rst_n`)
3. **Enable Signal:** System enable (`ena`) from external controller
4. **UART1 Interface:**
   - RX (input from bootloader tool)
   - TX (output, optional acknowledgment)
5. **UART2 Interface:**
   - RX (input for general communication)
   - TX (output for general communication)
6. **SPI Interface:**
   - MOSI (Master Out, Slave In) — output from processor
   - MISO (Master In, Slave Out) — input to processor
   - SCLK (Serial Clock) — output from processor

    - CS (Chip Select) — output from GPIO2
7. **GPIO1:** General-purpose output (open-drain or push-pull driver)
8. **GPIO2:** Chip Select output for SPI slave (open-drain or push-pull driver)

Notes

- This project is an educational, hands-on RISC-V SoC design
- It is developed for learning CPU architecture, pipeline execution, and hardware-software interaction
- Intended for academic and experimental use in Nepal
- Focus is on clarity and understanding rather than commercial optimization
- The design prioritizes simplicity, determinism, and observability of CPU behavior
- No interrupt system is implemented; all peripherals use polling only
- Instruction format assumes little-endian byte ordering

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	BOOT_UART1_RX	BOOT_UART1_TX	SPI_CS_GPIO2_TOP
1	PER_UART2_RX	PER_UART2_TX	SPI_MOSI_TOP
2	—	—	SPI_MISO_TOP
3	—	—	SPI_SCLK_TOP
4	—	—	GPIO1_TOP
5	—	—	SPI_CS_GPIO3_TOP
6	—	—	—
7	—	—	—

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

Antonio's first Wokwi design

by **Antonio**

0065 Wokwi Project

github.com/aglcoin04/tt-antonios-first-wokwi

wokwi.com/projects/465731440267947009

"An elite project"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

sdft wokwi 1

by **Santiago Díaz Fernández de la Torre**

0066 Wokwi Project

github.com/santidft/tt-sdft-wokwi-1

wokwi.com/projects/465731403724006401

"some bs"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Guille's first Wokwi design.

by **Guillermo**

0067 Wokwi Project

github.com/guillermomillansolis-rgb/tt-guilles-first-wokwi

wokwi.com/projects/465732706576877569

"Proyecto elite."

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Line Follower Robot controller

by **Escuela Colombiana de Ingenieria Julio Gravito**

0068 512 kHz HDL Project

github.com/Arvaloez/ECI_SEGUIDOR_LINEA_IEEE_BOOTCAMP

"*Basic controller for a line follower of 8 sensors*"

How it works

The proposed project consists of a mini controller designed for an 8-sensor line-following robot car. The controller reads the digital signals from the sensor array and computes a tracking error using a weighted method. This calculated error represents the position of the line relative to the center of the robot. Based on this value, the system generates a correction that adjusts the PWM signals driving the two DC motors, one on the right side and the other on the left side of the robot. By modifying the duty cycle of each motor independently, the controller corrects the trajectory and keeps the robot aligned with the line. Additionally, the design includes a 4-pin configuration input that allows the user to modify certain tuning parameters, enabling the robot to turn more aggressively or more smoothly depending on the selected setting. The tables for this settings are show above:

uio_in[1:0]	Condition	Operation	Adjustment
00	sum_error > 0	error = sum_error	0
01	sum_error > 0	error = sum_error + 6	+6
10	sum_error > 0	error = sum_error + 11	+11
11	sum_error > 0	error = sum_error + 15	+15

uio_in[3:2]	Condition	Operation	Adjustment
00	sum_error < 0	error = sum_error	0
01	sum_error < 0	error = sum_error − 6	−6
10	sum_error < 0	error = sum_error − 11	−11
11	sum_error < 0	error = sum_error − 15	−15

How to test

In order to test this design, you will need a little car with a 8CH IR TRACK MODULE, and to DC motors. Also you can prove the functionality with the test_bench file in the test folder.

External hardware

1) 8-Channel IR Line Tracking Sensor Array An infrared reflective sensor array is required to detect the position of the line relative to the robot. The array consists of eight IR emitter–receiver pairs placed in a row under the robot chassis. Each sensor measures the reflected infrared light from the floor surface, allowing the system to distinguish between the line and the background.

2) Analog Comparators for Sensor Thresholding Since the IR sensors produce analog voltage outputs, external analog comparators are required to convert these signals into digital logic levels. Each comparator compares the sensor voltage against a reference threshold and outputs a logic '1' when the reflected signal exceeds the threshold, indicating detection of the line. A total of eight comparators are required, one per sensor channel.

3) Dual DC Motors The robot uses two DC motors, one on the left side and one on the right side. These motors provide differential drive, allowing the robot to steer by adjusting the speed of each wheel independently through PWM control signals generated by the chip.

4) H-Bridge Motor Driver An external dual H-bridge motor driver is required to interface the PWM outputs of the controller with the motors. The H-bridge allows proper power amplification and enables the motors to be driven safely with the required current. It also provides the necessary switching for motor control.

5) Configuration Switches (4 Pins) A set of four digital configuration inputs connected to switches or jumpers allows the user to modify the controller tuning parameters. These inputs adjust the magnitude of the correction applied to the computed tracking error, enabling the robot to turn more aggressively or more smoothly.

6) Power Supply A suitable DC power supply or battery pack is required to power the sensors, logic circuitry, and motors. The supply may include voltage regulation stages to provide stable logic levels for the controller and analog circuitry.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Sensor 1	PWM Output Left Motor	ADJUST 1
1	Sensor 2	PWM Output Right Motor	ADJUST 2
2	Sensor 3	LED ON	ADJUST 3
3	Sensor 4	—	ADJUST 4

#	Input	Output	Bidirectional
4	Sensor 5	—	—
5	Sensor 6	—	—
6	Sensor 7	—	—
7	Sensor 8	—	—

MIPS-Lite 8-bit Processor

by **Jesus (jalperdev)**

0069 50 MHz Wokwi Project

github.com/jalperdev/tt-chip-design-nucleo

wokwi.com/projects/465731481873367041

"8-bit MIPS-inspired single-cycle processor with 8 registers, ALU, branches and jumps"

How it works

This is an 8-bit MIPS-inspired single-cycle processor designed for Tiny Tapeout. It preserves the core architectural concepts of a full 32-bit MIPS processor, scaled down to fit in a single tile:

Architecture

- **8 general-purpose registers** (R0 hardwired to 0, R1-R7 writable)
- **8-bit ALU** with ADD, SUB, AND, OR, SLT operations
- **16-bit instruction format** supporting R-type, I-type, and J-type instructions
- **5-bit Program Counter** addressing 32 instruction ROM entries
- **Single-cycle execution** — one instruction per clock cycle

Instruction Set

Opcode	Name	Format	Operation
0000	R-type	R	ADD, SUB, AND, OR, SLT (based on funct field)
0001	ADDI	I	R[rt] ← R[rs] + sign_ext(imm)
0010	ANDI	I	R[rt] ← R[rs] & zero_ext(imm)
0011	ORI	I	R[rt] ← R[rs] | zero_ext(imm)
0100	BEQ	I	if R[rs]==R[rt]: PC ← PC+1+imm
0101	BNE	I	if R[rs]!=R[rt]: PC ← PC+1+imm
0110	J	J	PC ← addr
0111	LUI	I	R[rt] ← imm << 2
1111	HALT	-	Stop execution

Demo Program

The hardcoded ROM contains a Fibonacci + ALU test program that:

1. Computes Fibonacci numbers (1, 1, 2, 3, 5)

2. Tests AND, OR, SUB, SLT operations
3. Tests BEQ (not taken) and BNE (taken) branches
4. Tests Jump instruction
5. Halts

How to test

1. Apply reset (rst_n = 0) for at least one clock cycle, then release (rst_n = 1)
2. The CPU will execute the hardcoded program automatically
3. Use `ui_in[2:0]` to select which register (R0-R7) to observe on `uo_out[7:0]`
4. Monitor `uio_out[4:0]` for the current Program Counter value
5. Check `uio_out[5]` for the halt flag (goes high when HALT is reached)

Expected results after execution:

Register	Value	Description
R0	0x00	Hardwired zero
R1	0x03	Fibonacci intermediate
R2	0x05	Fibonacci result
R3	0x05	Fibonacci result
R4	0x15	ADDI result (21)
R5	0x0F	ORI result (15)
R6	0xFE	SUB result (-2 signed)
R7	0x01	SLT result (3 < 5 = true)

External hardware

No external hardware required. Connect LEDs to outputs for visual observation:

- 8 LEDs on `uo_out` to display the selected register value
- 5 LEDs on `uio_out[4:0]` to show PC
- 1 LED on `uio_out[5]` for halt indicator
- 3 switches on `ui_in[2:0]` for register selection

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	reg_sel[0]	reg_data[0]	pc[0]
1	reg_sel[1]	reg_data[1]	pc[1]
2	reg_sel[2]	reg_data[2]	pc[2]

#	Input	Output	Bidirectional
3	—	reg_data[3]	pc[3]
4	—	reg_data[4]	pc[4]
5	—	reg_data[5]	halt
6	—	reg_data[6]	alu_zero
7	—	reg_data[7]	—

xoroshiro64

by **Vivek Chiranjit**

0070 | 25 MHz | HDL Project

`github.com/ChiranjitPatel/ttgf_xoroshiro64`

"xoroshiro TRNG"

How it works

This design implements a lightweight **xoroshiro64+ random number generator** with **ring oscillator (RO) noise injection** to provide entropy.

The generator maintains two 32-bit internal states (`s0` and `s1`). On each enabled clock cycle:

1. A 32-bit random word is generated using a ripple-carry adder:

   ```
   random_word = s0 + s1
   ```

2. The xoroshiro64+ state update equations compute the next values of `s0` and `s1`.
3. The RO noise input is XORed into the least significant bit of `s0` to introduce true randomness.
4. The generated 32-bit word is loaded into a serializer.
5. The serializer shifts out one bit per clock on `serial_out`.

A pulse on `valid_out` indicates the start of a new 32-bit random word.

The internal state can also be seeded through an 8-bit seed-loading interface.

How to test

1. Apply a clock to `clk`.
2. Release reset by setting `rst_n = 1`.
3. Set `enable = 1`.
4. Provide a changing signal on `ro_noise` (for example, from a ring oscillator).
5. Monitor `serial_out`.
6. Observe `valid_out`; it pulses high when a new 32-bit random word begins shifting out.
7. Optionally load custom seeds using `seed_data`, `seed_wen`, and `seed_sel`.

External hardware

- 1-bit Ring Oscillator (RO) noise source connected to `ro_noise`.
- No other external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	seed data	serial_out_w	seed_sel
1	seed data	valid_out_w	seed_sel
2	seed data	osc_ro_fast	seed_sel
3	seed data	osc_ro_slow	seed_wen
4	seed data	—	enable
5	seed data	—	ro_noise
6	seed data	—	—
7	seed data	—	—

Juan`s first Worki design

by **Juan Andres**

0071 Wokwi Project

github.com/JuanAndres6386/tt-juans-first-worki

wokwi.com/projects/465736612213902337

"*Proyecto bombi de la elite, drip drip*"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Wafer.space Logo VGA Screensaver

by **Uri Shaked**

0096 | 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 96.1: 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	—

happyhop

by **deadcast2**

0098 | 25.175 MHz | HDL Project

`github.com/deadcast2/tt_um_happyhop`

"A bouncing smiley face rendered live to VGA via the standard TT VGA PMOD. No framebuffer; pixels computed on the fly."

How it works

A 16x16 smiley sprite bounces around a 640x480 VGA screen. No framebuffer — each pixel is computed live from the beam position and the ball's `(x, y)`. The smiley blinks every 2 seconds and its eyes shift toward the direction of motion.

How to test

Plug the TT VGA PMOD into the chip's output PMOD, connect a 640x480 @ 60 Hz monitor, power on. No input controls.

External hardware

- TT VGA PMOD (Matt Venn)
- VGA monitor at 640x480 @ 60 Hz

Project Pinout

Digital Pins

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

Dino-7: 7-Segment Runner Game

by **Dr Sopes**

`0100` `25 MHz` `HDL Project`

`github.com/DrSopes/Dino-7-GF`

> *"A minimalist single-button endless runner game inspired by the Chrome Dinosaur game, rendered on a single 7-segment display. Features an LFSR of 32 bits, Hardware Difficulty Selector, and a High Score memory."*

How it works

Dino-7 is a minimalist endless runner inspired by the Chrome Dinosaur game, rendered on a single 7-segment display. The game uses a finite state machine with idle, run, jump, hit, and score-display behavior, plus pseudo-random obstacle generation and a retained high score.

Game states

State	Description
`S_IDLE`	Shows the stored high score on the 7-segment display
`S_RUN`	Player is on the ground and obstacles advance
`S_JUMP`	Player is in the air for a short fixed jump time
`S_HIT`	Collision detected, all segments light up
`S_SCORE`	Final score is shown, alternating with high score

Display mapping

The 7-segment display is used as a tiny side-view game field with a horizontal obstacle flow:

Output	Segment	Meaning
`uo[0]`	`a`	Obstacle far (spawning)
`uo[1]`	`b`	Obstacle mid (approaching)
`uo[2]`	`c`	Player on ground
`uo[3]`	`d`	Obstacle has passed
`uo[4]`	`e`	Player in air (jumping)
`uo[5]`	`f`	Unused
`uo[6]`	`g`	Obstacle close (collision zone)
`uo[7]`	`dp`	Jump cooldown active / High score indicator

Obstacle movement

Obstacles move through four visible phases across the display:

1. `a` — a new obstacle appears far away.
2. `b` — the obstacle approaches the player.
3. `g` — the obstacle reaches the close collision zone.
4. `d` — the obstacle has passed.

A 32-bit LFSR is used to generate pseudo-random obstacle spawn timing from the seed bits on `ui[7:4]`.

Jump and score logic

The player is shown on `c` while running and on `e` while jumping. After a jump, the decimal point turns on briefly to indicate jump cooldown, so the jump button is temporarily blocked.

If the player avoids an obstacle successfully, the score increments up to 9. The game speed increases gradually depending on the selected difficulty and score progression.

Score and high score

In idle mode, the display shows the saved high score with the decimal point enabled. After a collision, the game briefly lights all segments, then shows the current score and alternates it with the stored high score.

How to test

Controls

Pin	Function
`ui[0]`	Jump button (active high)
`ui[1]`	Game reset (active high)
`ui[3:2]`	Difficulty selector
`ui[7:4]`	LFSR seed bits

Step-by-step

1. Apply power and release `rst_n`.
2. The display starts in idle mode and shows the high score with the decimal point on.
3. Press `ui[0]` to start the game.
4. Watch the obstacle move through `a` → `b` → `g` → `d`.
5. Press `ui[0]` to jump before the obstacle reaches the close collision zone.
6. During cooldown, the decimal point turns on.
7. On collision, all segments light up, then the score screen appears.
8. Press `ui[1]` to reset the game and return to idle.

External hardware

- One common-cathode 7-segment display connected to `uo_out[7:0]`
- One push button on `ui[0]` for jump
- One push button on `ui[1]` for game reset
- Optional switches on `ui[3:2]` and `ui[7:4]` for difficulty and seed selection

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Jump button (active high)	Segment a – obstacle far (spawning)	—
1	Reset game (active high)	Segment b – obstacle mid (approaching)	—
2	Difficulty Selector LSB (00=Norm, 11=Insane)	Segment c – player on ground	—
3	Difficulty Selector MSB	Segment d – obstacle has passed	—
4	LFSR seed bit 0	Segment e – player in air (jumping)	—
5	LFSR seed bit 1	Segment f – unused	—
6	LFSR seed bit 2	Segment g – obstacle close (collision zone)	—
7	LFSR seed bit 3	Decimal point – jump cooldown active / High Score indicator	—

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

Custom DVD Screensaver for VGA

by **Jazmin Nayeli / Kensy Buatista**

`0102` `25.175 MHz` `HDL Project`

`github.com/nayelikensy930-lang/tt_um_uacj_`

"Bouncing DVD logo on VGA with color change on wall hits, speed control, invert, and flip options"

A Tiny Tapeout digital design that displays a **128×40 pixel** bitmap logo bouncing around a 640×480 VGA screen, with real-time visual effects controlled via the input pins.

Overview

The design generates a standard VGA signal (640×480, 60 Hz) and animates a bitmap logo that bounces within the visible area, cycling through colors on each wall collision. Eight visual effects are individually controlled by the bits of `ui_in`.

Pinout

Inputs (`ui_in`)

Bit	Name	Description
0	`cfg_tile`	Tile mode: repeats the logo across the entire screen
1	`cfg_color`	Enables dynamic color palette
2	`cfg_invert`	Swaps foreground and background colors
3	`cfg_slow`	Halves the animation speed
4	`cfg_flip`	Flips the logo vertically
5	`cfg_checker`	Enables checkerboard background pattern
6	`cfg_scanline`	Simulates CRT scanlines
7	`cfg_glitch`	Enables the visual corruption (glitch) engine

Outputs (`uo_out`) — TinyVGA PMOD

Bits	Signal
[7]	HSYNC
[6]	B[0]

[5]	G[0]
[4]	R[0]
[3]	VSYNC
[2]	B[1]
[1]	G[1]
[0]	R[1]

Compatible with the standard Tiny Tapeout **TinyVGA PMOD**.

Bidirectionals (`uio_*`)
Not used. `uio_out` and `uio_oe` are held at `0`.

Timing Parameters

Parameter	Value
Clock (`clk`)	25 MHz (40 ns period)
Reset	Active-low (`rst_n`)
Resolution	640 × 480 pixels
Refresh rate	60 Hz
Logo size	128 × 40 pixels

Modules

`tt_um_uacj` **(top)**
Top-level module. Instantiates all submodules and implements:

- **Bounce logic**: the logo starts at (200, 200) and travels diagonally. On each wall hit, the corresponding direction is reversed.
- **Collision flash**: for 6 frames after a bounce, the logo is rendered in full white.
- **Speed divider** (`cfg_slow`): when active, the logo advances one pixel every two frames.
- **Dynamic color**: the color index increments on each bounce and is applied to the logo, background, and checkerboard pattern through the 8-color palette.

`hvsync_generator`
Generates horizontal and vertical sync signals according to the VGA standard:

Parameter	Value

H_DISPLAY	640
H_FRONT porch	16
H_SYNC	96
H_BACK porch	48
V_DISPLAY	480
V_BOTTOM border	10
V_SYNC	2
V_TOP border	33

Exposes `hpos` and `vpos` (10-bit each) and the `display_on` signal indicating when the beam is within the visible area.

`bitmap_rom`

A **640-byte** read-only memory (40 rows × 16 bytes) storing the monochrome logo bitmap at 128×40 pixels (1 bpp).

- Addressing: `addr = y[5:0] * 16 + x[6:3]`
- Pixel extraction: `pixel = mem[addr][x[2:0]]`
- Supports vertical flip via `cfg_flip`.

`palette`

An 8-entry LUT in 6-bit `RRGGBB` format (2 bits per channel):

Index	Color	Value
0	Cyan	`001011`
1	Pink	`110110`
2	Green	`101101`
3	Orange	`111000`
4	Purple	`110011`
5	Yellow	`011111`
6	Red	`110001`
7	White	`111111`

Instantiated three times: logo color, background color, and checkerboard color.

Glitch Engine

When `cfg_glitch = 1`, a visual corruption engine is activated based on an **8-bit LFSR** (taps at positions 7, 5, 4, 3). The engine runs a 3-state FSM:

State	Description
0	Idle — waiting for a trigger
1	Active corruption (runs for `glitch_timer` frames)
2	Recovery (1-frame transition back to idle)

Effects applied during corruption:

- **Horizontal/vertical XOR scrambling**: pixels are displaced using an LFSR-derived mask.
- **Horizontal tearing**: selected rows are offset laterally by a pseudo-random amount.
- **Chromatic aberration**: each RGB channel receives an independent XOR from the LFSR.
- **Channel swapping**: R↔G and G↔B are permuted based on LFSR bits.
- **Inversion flash**: at maximum intensity, colors are inverted on select frames.

The glitch engine also triggers automatically approximately every 150 frames while `cfg_glitch = 1`.

How to Use

1. Connect the **TinyVGA PMOD** to the `uo_out` pins.
2. Connect a VGA monitor to the PMOD.
3. Supply a **25 MHz** clock on `clk`.
4. Assert `rst_n = 0` briefly at startup, then release (`rst_n = 1`).
5. Control visual effects with the 8 bits of `ui_in` as described in the pinout table above.

Project Files

File	Description
`tt_um_uacj.v`	Top module: animation, effects, RGB output
`hvsync_generator.v`	VGA sync signal generator
`bitmap_rom.v`	Logo bitmap ROM (128×40, 1 bpp)
`palette.v`	8-color palette LUT
`config.json`	Tiny Tapeout build config (25 MHz clock)

License

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	cfg_tile	R1	—
1	cfg_color	G1	—
2	cfg_invert	B1	—
3	cfg_slow	VSync	—
4	cfg_flip	R0	—
5	cfg_checker	G0	—
6	cfg_scanline	B0	—
7	cfg_glitch	HSync	—

Minimal RV32E SoC with UART Loader

by **Kris / KLuterIRV**

`0103` `20 MHz` `HDL Project`

`github.com/KLuterIRV/INNOVA-IRV-ULP-RISCV`

> "*Minimal RV32E microcontroller with UART program loader, internal instruction memory, GPIO and configurable 7-segment output*"

This project implements a minimal RV32E-based microcontroller for Tiny Tapeout.

Features

- Simplified RV32E CPU core
- 16 x 32-bit register file
- Internal instruction memory, programmable through UART
- Internal data memory and memory-mapped peripherals
- UART RX program loader
- UART TX output
- Shared 8-bit GPIO / seven-segment output
- Configurable seven-segment mode:
  - GPIO mode
  - HEX mode
  - ASCII mode
  - RAW segment mode
- `ebreak`-based halt/debug signal

Pinout

Pin	Function
`ui_in[0]`	Boot mode enable
`ui_in[1]`	UART RX
`uo_out[7:0]`	GPIO output or seven-segment output
`uio_out[0]`	UART TX
`uio_out[1]`	Core halted debug
`uio_out[2]`	Loader done debug
`uio_out[3]`	Loader error debug

How it works

On reset, the design can operate in two modes depending on `ui_in[0]`.

When `ui_in[0] = 1`, the system enters boot mode. In boot mode, the RV32E core is held at `PC = 0` while the UART loader receives a program through `ui_in[1]`. The loader writes 32-bit instruction words into the internal instruction memory. When the loader finishes, `loader_done_debug` is asserted and the core starts executing from address `0x00000000`.

When `ui_in[0] = 0`, the core starts directly from its internal instruction memory. The instruction memory contains a default demo program that writes characters to the seven-segment output.

The CPU is a simplified RV32E core with 16 integer registers. It supports arithmetic, logic, shifts, branches, jumps, loads, stores, LUI/AUIPC, and an `ebreak`-based halt mechanism.

The output `uo_out[7:0]` is shared between normal GPIO mode and seven-segment display mode. The seven-segment peripheral supports HEX, ASCII and RAW segment patterns.

The UART loader protocol is:

```text 0x55 N_WORDS WORD0 byte0 WORD0 byte1 WORD0 byte2 WORD0 byte3 ...

Words are transmitted little-endian. For example:

0x100000B7 -> B7 00 00 10

## How to test

To test the default demo, keep ui_in[0] = 0, apply reset, and clock the design. The core should start executing the internal demo program and drive uo_out[7:0] with seven-segment ASCII patterns.

To test UART program loading:

Set ui_in[0] = 1 to enable boot mode. Send the UART sync byte 0x55 to ui_in[1]. Send the number of 32-bit words. Send each instruction word little-endian. Wait for uio_out[2], which indicates loader_done_debug. The core will then execute from PC = 0.

Useful pins:

Pin Function ui_in[0] Boot mode ui_in[1] UART RX uo_out[7:0] GPIO / seven-segment shared output uio_out[0] UART TX uio_out[1] Core halted debug uio_out[2] Loader done debug uio_out[3] Loader error debug External hardware

The design can be tested using a USB-UART adapter connected to ui_in[1] for RX and uio_out[0] for TX.
```

The seven-segment output is active-high by default. If the target display is active-low, the firmware can enable the active-low mode in the seven-segment control register.

Memory map Address Register 0x1000_0000 GPIO output 0x1000_0004 Seven-segment data 0x1000_0008 Seven-segment control 0x1000_0100 UART TX data 0x1000_0104 UART status 0x1000_010C UART RX data Seven-segment control

SEVENSEG_CTRL bits:

Bit Meaning 0 Enable seven-segment output 1 ASCII mode 2 RAW mode 3 Active-low output

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sram_we	sram_rdata[0]	sram_wdata[6]
1	sram_addr[0]	sram_rdata[1]	sram_wdata[7]
2	sram_wdata[0]	sram_rdata[2]	sram_addr[1]
3	sram_wdata[1]	sram_rdata[3]	sram_addr[2]
4	sram_wdata[2]	sram_rdata[4]	sram_addr[3]
5	sram_wdata[3]	sram_rdata[5]	sram_addr[4]
6	sram_wdata[4]	sram_rdata[6]	sram_addr[5]
7	sram_wdata[5]	sram_rdata[7]	—

Tiny Tapeout Template Copy

by **Pablo Ruiz**

0128 Wokwi Project

github.com/pabloruizlopez2001-cloud/tt-template-copy

wokwi.com/projects/465731521356457985

"7-Segment "

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

7-Segment Neural Predictor

by **Abdul Hannan A.**

0129 HDL Project

`github.com/Nontinium/ttgf-machine-learning`

"A small neural network that predicts digits 0-9 from 7-segment display patterns."

How it works

Uses a two-layer neural network with 4 hidden neurons to decode 7-segment patterns.

How to test

Feed 7-segment codes into ui_in and read the 4-bit prediction on uo_out.

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	segment A	bit 0 prediction	—
1	segment B	bit 1 prediction	—
2	segment C	bit 2 prediction	—
3	segment D	bit 3 prediction	—
4	segment E	—	—
5	segment F	—	—
6	segment G	—	—
7	—	—	—

El primer diseño

by **Maria Saldaña**

0130 Wokwi Project

github.com/02mariasv/tt-el-primer-diseo

wokwi.com/projects/465731520738845697

"Mi primer diseño con Wokwi"

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	switch 1	salida a display 1	—
1	switch 2	salida a display 2	—
2	switch 3	salida a display 3	—
3	switch 4	salida a display 4	—
4	switch 5	salida a display 5	—
5	switch 6	salida a display 6	—
6	switch 7	salida a display 7	—
7	switch 8	salida a display 8	—

Mi copia del Tiny Tapeout

by **Antonio**

0132 Wokwi Project

github.com/VallejoPretelA/tt-mi-copia-del

wokwi.com/projects/465731458628527105

"Test del Taller Tiny Tapeout"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

test1

by **Alberto**

0134 Wokwi Project

github.com/aperezfer/tt-test1

wokwi.com/projects/465736691688630273

"First contact with the application"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

serv_soc_wb

by **Juan David Guerrero Balaguera**

0135 25 MHz HDL Project

github.com/divadnauj-GB/tt_um_divadnauj-GB_serv_soc_wb

"Custom SOC using the SERV processor under RV32e"

How it works

RISC-V SoC based on the SERV

This is a RISC-V SoC based on the SERV and SERVILE cores from https://github.com/olofk/serv.git. The SERV core has been adapted to support the RV32E specifications. The register file was implemented via an DFF-based SRAM of 19x32 bit, 15 General Purpose Registers (GPR) and 4 CSR.

The SoC is composed of a SERV core, a GPIO, an UART a 64-bit timer and a QSPI controller. All components connected via a wishbone bus interconect. The QSPI controller allows the SoC to access an external QSPI FLASH and RAM memories using the QSPI Pmod board.

The following diagram ilustrates the main system interconection.

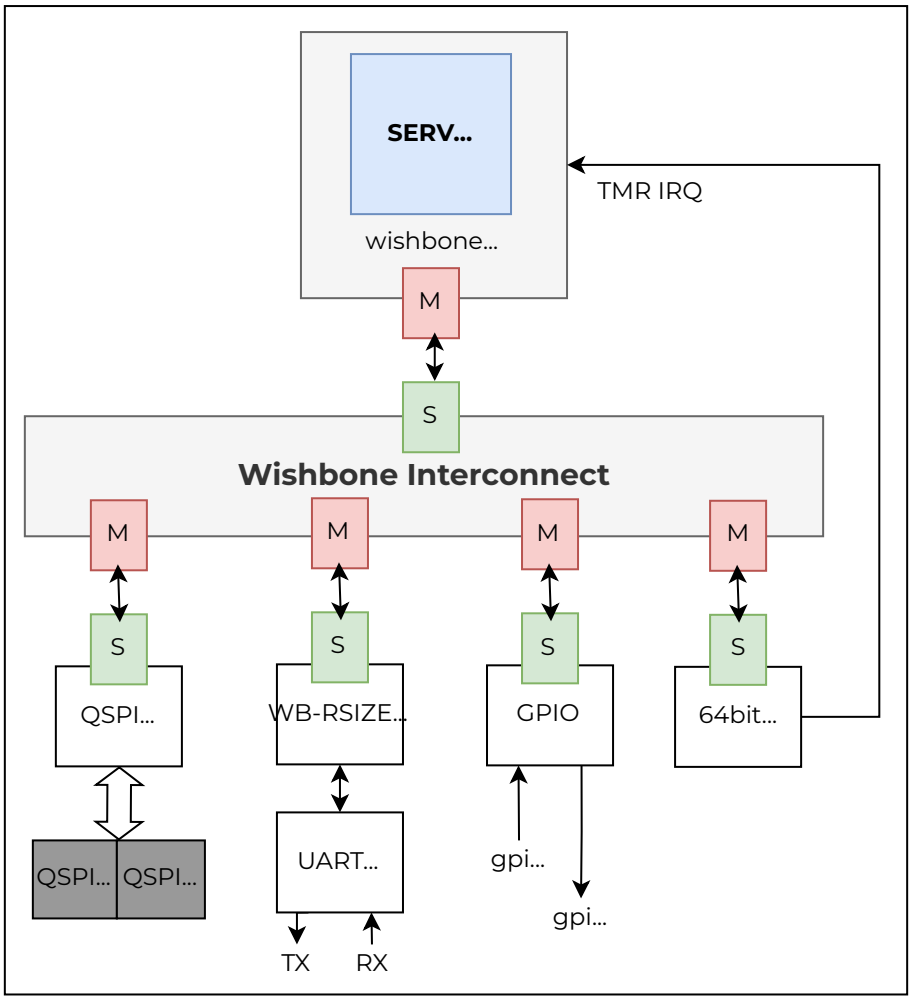


Figure 135.1: alt text

Memory Map

Peripheral	Address
QSPI-FLASH	0x00000000 - 0x00FFFFFF
QSPI-RAM1	0x01000000 - 0x017FFFFF
QSPI-RAM2	0x01800000 - 0x01FFFFFF
UART-16550	0x90000000 - 0x90000007
GPIO	0x91000000 - 0x9100001F
MTIMEL	0xFFFFFFF0 - 0xFFFFFFF3
MTIMEH	0xFFFFFFF4 - 0xFFFFFFF7
MTIMECMPL	0xFFFFFFF8 - 0xFFFFFFFB
MTIMECMPH	0xFFFFFFFC - 0xFFFFFFFF

NOTE: The whole system works maximum at 25Mhz on gf180mcu and 50MHz on sky130A.

It is worth mentioning that the SoC can execute code from both FLASH and SRAM memories, the execution from SRAM is posible thanks to an small bootloader loaded into the FLASH that receives the program from the UART and dumps it into the SRAM.

Pinout interface

The following is the pinout on the tiny tapeout chip:

The gpio_o can be connected to any type of output, LEDs, LCDs, etc and the gpio_i can be connected to any kind of input, buttons, sensors etc.

The uart must be connected to your laptop via an usb-serial converter (e.g., rs232rl or similar).

#	Input (ui)	Output (uo)	Bidirectional (uio)
0	gpio_i[0]	gpio_o[0]	cs_flash
1	gpio_i[1]	gpio_o[1]	sio0
2	gpio_i[2]	gpio_o[2]	sio1
3	gpio_i[3]	gpio_o[3]	sck
4	gpio_i[4]	gpio_o[4]	sio2
5	gpio_i[5]	gpio_o[5]	sio3
6	gpio_i[6]	gpio_o[6]	cs_ram0
7	uart0_rx	uart0_tx	cs_ram1

How to test

Install the RISC-V toolchain with RV32E support

1. Prerequisites on Ubuntu

```
#using bash
sudo apt-get install autoconf automake autotools-dev curl python3
python3-pip python3-tomli libmpc-dev libmpfr-dev libgmp-dev gawk
build-essential bison flex texinfo gperf libtool patchutils bc
zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev
libslirp-dev libncurses-dev
```

2. Prerequisites on MacOS

```
# using homebrew
brew install python3 gawk gnu-sed make gmp mpfr libmpc isl zlib
expat texinfo flock libslirp ncurses ninja bison m4 wget
```

3. Configure and compile the crosscompiler for RV32E:

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain.
git
cd riscv-gnu-toolchain
```

```
mkdir -p /opt/riscv32e # you can pickup any location in your
system, this examples is for ubuntu
./configure --prefix=/opt/riscv32e --with-arch=rv32e --with-
abi=ilp32e
make -j$(nproc)
# Make the instalation directory available and accesible ( you need
to run this every time you open a new terminal session)
export PATH=/opt/riscv32e/bin:$PATH
# This command will permanently add the toolchain directory to your
user environment; In this way the toolchain will be available
everytime you open a terminal session.
echo 'export PATH=/opt/riscv32e/bin:$PATH' >> ~/.bashrc
```

Run Examples

The following sugested schematic should be implemented to try out the proposed examples. The SoC requires external Flash and SRAM memories under the SPI or QSPI protocols; you can either use the QSPI Pmod board or individual memories W25Q128 and APS6404L as depicted in the schematic. The clock must be established to run at 25MHz, which is the maximum achievable SoC frequency. It is worth noting that any lower clk frequency can be used, in that case you need to modify the bootloader frequency accordingly.

The examples require to connect switches to the **gpio_i[*]** ports and LEDs to the **gpio_o[*]** ports. However, if you want to try your own programs, you can use these ports to connect any peripheral of your preference.

Finally, an USB to Serial converter is required to evaluate and interact with most of the examples. Furthermore, programming the SoC is also possible via the UART.

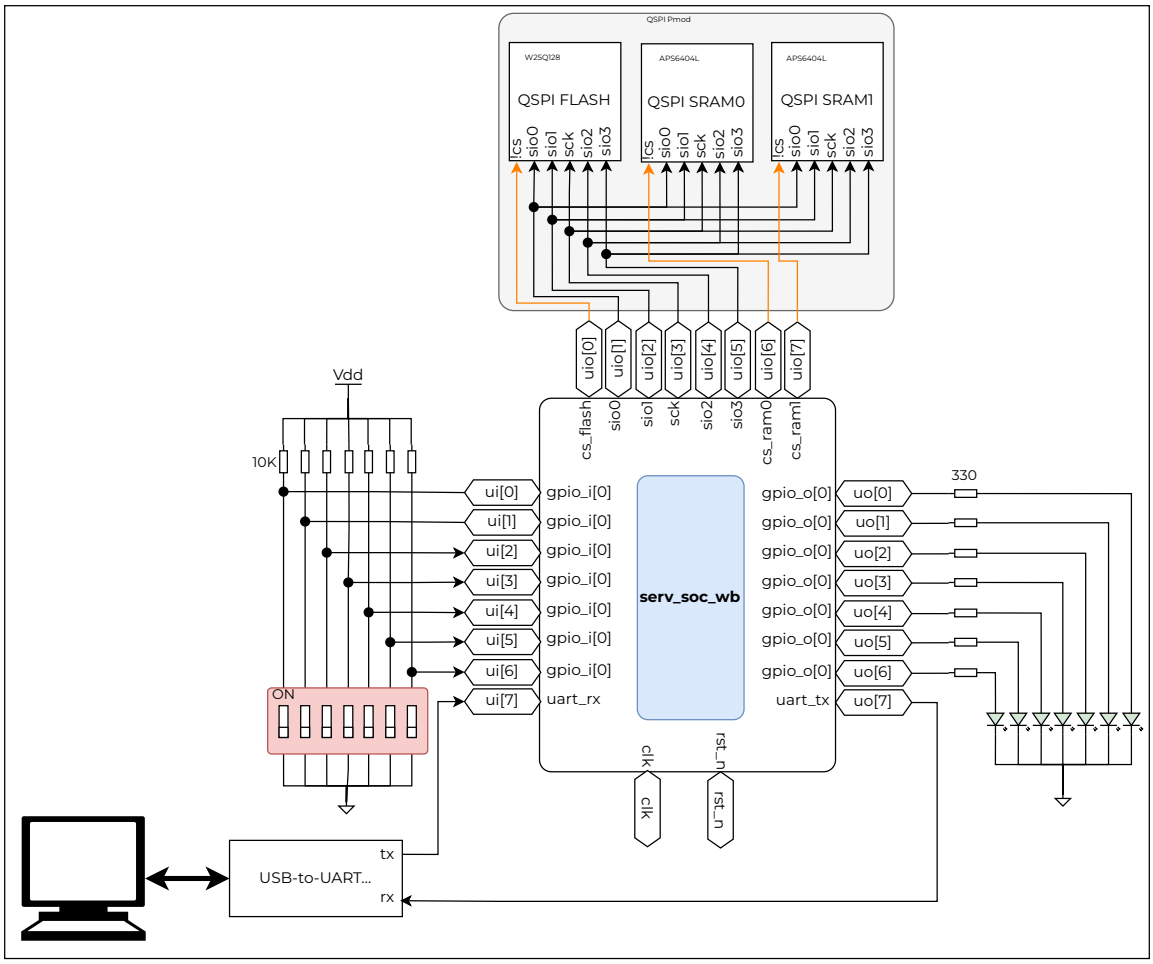


Figure 135.2: alt text

1. Flash the nmon bootloader:
   - Program the flash memory using the nmon_25MHz.bin bootloader, this step is required to be done only once. You can use any FLASH programer (e.g., serprog). > Eventually further guidelines might be provided regarding options for flashing the bootloader
2. Compile the program `bash cd ./sw/1-blink_led make clean build nmon`
3. Program the SERV-E-SoC with the compiled application `bash # check the actual USB port in your system expect nmon-loader.sh application.nmon /dev/ttyUSB0 115200` > Eventually another way of programming the SoC can be provided in the future.

The following is a set of basic examples that have been prepared to use the SERV-E SoC. Every example has details about how to compile and program on the SERV-E SoC.

0-ricv-nmon: This is a basic bootloader living in the flash memory, it can be used to dump a program into de SRAM comming from the uart port.

1-blink_led: Basic blink led.

2-gpio_echo: Simple GPIO echo, copies inputs and put it to the outputs.

3-uart_stub_1: Uses the nmon uart function to access the uart.

4-uart_stub_2: Uses the nmon uart function to access the uart.

5-uart_puts: Explicit uart driver implementation, just transmision from SoC to PC.

6-uart_getc: Explicit uart driver implementation RX/TX.

7-systmr-irq: Timer IRQ implementation example.

8-FreeRTOS-demo1: Simple FreeRTOS port for the SoC.

9-FreeRTOS-demo2: Simple FreeRTOS port for the SoC.

External hardware

This project requires an external SPI Flash/Ram memory. The project has been proven using QPMOD. You also need to connect an USB to Serial conversor to interact with the system.

FPGA proven design

This design has been proved on an FPGA device via the DE10-nano board. Currently working on porting the design into an open source friendly FPGA device.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	gpio_i[0]	gpio_o[0]	cs_flash
1	gpio_i[1]	gpio_o[1]	sio0
2	gpio_i[2]	gpio_o[2]	sio1
3	gpio_i[3]	gpio_o[3]	sck
4	gpio_i[4]	gpio_o[4]	sio2
5	gpio_i[5]	gpio_o[5]	sio3
6	gpio_i[6]	gpio_o[6]	cs_ram0
7	uart0_rx	uart0_tx	cs_ram1

Gen1 Digital Companion Tile

by **Juan Fernandez**

0192 | 20 MHz | HDL Project

`github.com/JuanCarlosFdezMlg/tt-juan-carloss-first`

"Symbolic pulse-update and write-verify companion tile for a future local memristive AI chip"

How it works

This project is a small digital companion tile for a future local memristive AI chip. It does not implement real memristors. Instead, it implements the digital control contract around a symbolic memristive update:

1. Load a symbolic target value with `LOAD_TARGET`.
2. Load a symbolic current value with `LOAD_CURRENT`.
3. Start a bounded write-verify loop with `START`.
4. Emit one-cycle `pulse_up` or `pulse_down` events while moving the symbolic current toward the target.
5. Finish with `verify_ok` when the target is reached, or `fault` when the attempt budget is exhausted.

The public output contract exposes status, pulse events, verification status, a fault bit, an attempt counter and an FSM state code. It does not expose the loaded target/current values directly on the public outputs in the tested scenarios.

How to test

Reset the design, then drive the command/data input bus:

- `01vvvvvv`: load target.
- `10vvvvvv`: load current.
- `11mmmm01`: start with max attempt count `m`.
- `11xxxx10`: clear public state.

Expected smoke tests:

- Target 5, current 2, max attempts 4: emits `pulse_up` events and finishes with `verify_ok`.
- Target 1, current 5, max attempts 4: emits `pulse_down` events and finishes with `verify_ok`.
- Target 8, current 0, max attempts 3: emits `fault` after timeout.

The cocotb tests in `test/test.py` cover these scenarios and the `CLEAR` command.

The Yosys-only contract checks in `formal/companion_contract.sv` prove the same terminal expectations for the packaged RTL and can be run with:

```
yosys formal/run_yosys_contract.ys
```

External hardware

No external hardware is required for the logic test. On a Tiny Tapeout demo board, switches or a microcontroller can drive the 8 input bits, and LEDs or a logic analyzer can observe the public outputs.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data0	privacy_ok	state0
1	data1	verify_ok	state1
2	data2	pulse_down	state2
3	data3	pulse_up	state3
4	data4	fault	attempt0
5	data5	done	attempt1
6	cmd0	busy	attempt2
7	cmd1	ready	attempt3

Tiny Tapeout Template Ayman

by **Ayman Harraz**

0194 1 kHz Wokwi Project

github.com/aymanhq96/tt-template-ayman

wokwi.com/projects/465732827753495553

"Tinytapeout"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

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

Tiny Pixel Processor

by **Julian Schlager**, **Thomas Lindinger**, **Patrick Pollak**, **Sebastian Gmeiner** & **Simon Vogelhuber**

0195 50.35 MHz HDL Project

github.com/vogelhubersimon/Tiny-Pixel-Processor

"A custom processor built for generating procedural graphics"

How it works

Tiny Pixel Processor is a custom processer designed to procedurally generate graphics.

Specifications:

- 640x480 VGA Output
- 64x48 pixels resolution
- 16-bit custom instruction set
- 8 registers (4 general purpose, 4 read-only)

Instruction set

The processor uses a custom 16-bit instruction format, which is designed to be simple and efficient for graphics generation. The instruction set includes operations for arithmetic, logic, and control flow, as well as special instructions (sin, ramp, saw, rand) for generating procedural patterns.

EBNF

The assembler language for this processor consists of several instruction types, each with their own syntax (look at the later code examples).

```
Shader = {Instruction "\n"}.
Instruction = Type0 | Type1 | Type2 | Type3 | Type4 | Type5 | Type6
| Type7.

Type0 = "NOP".
Type1 = "SET" RDestination Immediate [Condition].
Type2 = ( "SL" | "SR" ) RSourceDestination Immediate [Condition].
Type3 = "MOV" RDestination RSource [Condition].
Type4 = ( "ADD" | "SUB" | "AND" | "NAND" | "OR" | "NOR" | "XOR" |
"SIN" | "RAMP" | "SAW" ) RSourceDestination RSource [Condition].
Type5 = "COMP" RSource RSource [Condition].
Type6 = "OUT" RSource [Condition].
Type7 = ( "FH" | "TT" | "Credits" | "FlagP" ) RDestination
[Condition].

Condition = "EQ" | "LT" | "GT"
RSource = "R" ( "0" | "1" | "2" | "3" | ( "4" | "X" ) | ( "5" |
```

```
"Y" ) | ( "6" | "T" ) | ( "7" | "R" ) ).
RSourceDestination = RDestination.
RDestination = "R" ( "0" | "1" | "2" | "3" ).
Immediate = "#" 0 ... 63.
```

Instruction Type 0 (for NOP)

OP	Unused
5 Bit	11 Bit

- 5 Bit OP-Code
- 11 Bit Unused
- 16 Bit Total

Instruction Type 1 (for SET)

OP	RD	Immediate	Condition
5 Bit	3 Bit	6 Bit	2 Bit

- 5 Bit OP-Code
- 3 Bit Destination Register Selection
- 6 Bit Immediate
- 2 Bit Condition
- 16 Bit Total

Instruction Type 2 (for SL, SR)

OP	RSD	Immediate	Condition
5 Bit	3 Bit	6 Bit	2 Bit

- 5 Bit OP-Code
- 3 Bit Destination/Source Register Selection
- 6 Bit Immediate
- 2 Bit Condition

Instruction Type 3 (for MOV)

OP	RD	RS	Unused	Condition
5 Bit	3 Bit	3 Bit	3 Bit	2 Bit

- 5 Bit OP-Code
- 3 Bit Destination Register Selection
- 3 Bit Source Register Selection
- 2 Bit Condition

Instruction Type 4 (for Register-Register Operations)

OP	RSD	RS	Unused	Condition
5 Bit	3 Bit	3 Bit	3 Bit	2 Bit

- 5 Bit OP-Code
- 3 Bit Destination/Source Register Selection
- 3 Bit Source Register Selection
- 2 Bit Condition

Instruction Type 5 (for COMP)

OP	RS1	RS2	Unused	Condition
5 Bit	3 Bit	3 Bit	3 Bit	2 Bit

- 5 Bit OP-Code
- 3 Bit Source Register 1 Selection
- 3 Bit Source Register 2 Selection
- 2 Bit Condition

Instruction Type 6 (for OUT)

OP	RS	Unused	Condition
5 Bit	3 Bit	6 Bit	2 Bit

- 5 Bit OP-Code
- 3 Bit Source Register Selection
- 2 Bit Condition

Instruction Type 7 (for ROMs)

OP	RD	Unused	Condition
5 Bit	3 Bit	6 Bit	2 Bit

- 5 Bit OP-Code
- 3 Bit Destination Register Selection
- 2 Bit Condition

Instruction descriptions

OP	Usecase	Description
NOP	NOP	Does nothing
SET	SET RD Imm	RD = Imm
MOV	MOV RD RS	RD = RS
ADD	ADD RDS RS	RDS = RDS + RS
SUB	SUB RDS RS	RDS = RDS - RS

SL	SL RDS Imm	RDS = RDS << Imm
SR	SR RDS Imm	RDS = RDS >> Imm
AND	AND RDS RS	RDS = RDS & RS
NAND	NAND RDS RS	RDS = RDS & RS
OR	OR RDS RS	RDS = RDS | RS
NOR	NOR RDS RS	RDS = RDS | RS
XOR	XOR RDS RS	RDS = RDS ^ RS
SIN	SIN RDS RS	RDS = sin(RS)
RAMP	RAMP RDS RS	RDS = ramp(RS)
SAW	SAW RDS RS	RDS = saw(RS)
COMP	COMP RS1 RS2	sets condition register
FH	FH RD	RD = BITMAP[RX][RY]
TT	TT RD	RD = BITMAP[RX][RY]
Credits	Credits RD	RD = BITMAP[RX][RY]
FlagP	FlagP RD	RD = BITMAP[RX][RY]
OUT	OUT RS	Output RS to the VGA

ROM Instructions There are also a few special instructions that are used to load bitmap data from a ROM.

- **FH** (University of Applied Sciences Upper Austria Logo)
- **TT** (TinyTapeout Logo)
- **Credits** (Project Credits)
- **FlagP** (Flag pole)

These Instructions can be called like every other instruction (including conditionals), but they will load the corresponding bitmap according to the current pixel coordinates (RX, RY) into the destination register.

Registers

General Purpose Registers

Register 0-3 are general purpose registers that can be used for any purpose. They can be read and written by instructions.

Read Only Registers

Registers 4 (**RX**) and 5 (**RY**) contain the current pixel coordinates. Register 6 (**RT**) contains the current time (the count of frames divided by a programmable divisor). Register 7 (**RR**) contains a deterministic random value for every pixel (every frame is generated the same, so it is useful for generating a noise pattern).

The registers can only be read by the instruction, writing to them is not recommended, as it may cause unexpected behavior.

UART

To reconfigure the processor, you can send commands via UART. The following table shows the available commands:

Writing to the instruction memory

To reprogram the processor, you can send a sequence of 3 bytes via UART.

1. First byte: Command (see table below)
2. Second byte: First half of the instruction (bits 15-8)
3. Third byte: Second half of the instruction (bits 7-0)

Command hex	Command bin	Description
0x80	1000 0000	Addr 0
0x81	1000 0001	Addr 1
0x82	1000 0010	Addr 2
0x83	1000 0011	Addr 3
0x84	1000 0100	Addr 4
0x85	1000 0101	Addr 5
0x86	1000 0110	Addr 6
0x87	1000 0111	Addr 7
0x88	1000 1000	Addr 8
0x89	1000 1001	Addr 9
0x8A	1000 1010	Addr 10
0x8B	1000 1011	Addr 11
0x8C	1000 1100	Addr 12
0x8D	1000 1101	Addr 13
0x8E	1000 1110	Addr 14
0x8F	1000 1111	Addr 15
0x90	1001 0000	Addr 16
0x91	1001 0001	Addr 17
0x92	1001 0010	Addr 18
0x93	1001 0011	Addr 19

Configuring the Time Register Divisor

The time register contains count of frames divided by a programmable divisor. This divisor can be changed via UART and ranges from 0 to 63. Default is 5.

Command hex	Command bin	Time Counter
0x40	0100 0000	0
0x41	0100 0001	1
0x42	0100 0010	2
0x43	0100 0011	3
0x44	0100 0100	4
0x45	0100 0101	5
0x46	0100 0110	6
...	...	...
0x7D	0111 1101	61
0x7E	0111 1110	62
0x7F	0111 1111	63

How to test

The default configuration includes a simple program that generates a procedural pattern. You can plug a VGA monitor into the tiny-vga board and see the output from the processor.

If you want to program the ASIC, you can use the `flash_gui.py` script in the `flash` folder. This program allows you to parse your assembly code and upload it via COM port (USB to UART converter is needed) to the processor.

The following python packages are needed to run the script:

```
pip install customtkinter serial
```

Code Examples

Code examples can be found here.

External hardware

The tiny-vga board is used to display the output of the Tiny Pixel Processor on a VGA monitor.

Additionally, a USB to UART converter is needed to upload the program to the processor. The chip uses 9600 baud rate, 8 data bits, no parity, and 1 stop bit.

Credits

This project was created by Patrick Pollak, Julian Schlager, Thomas Lindinger, Simon Vogelhuber and Sebastian Gmeiner. The project was developed as part of a course at the University of Applied Sciences Upper Austria, Campus Hagenberg. The course and project were supervised by Prof. DI Dr. Markus Pfaff.

This project was inspired by TinyShader by mole99, originally created for the TT06 Shuttle. While TinyShader served as a conceptual starting point, we made deliberate design decisions throughout development - including our own custom processor architecture and instruction set.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Uart_RX	R1	—
1	—	G1	—
2	—	B1	—
3	—	VSYNC	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	—	HSYNC	—

Julian_Proyecto

by **Julián**

0196 10 kHz Wokwi Project

github.com/jpachecorub/tt-julianproyecto

wokwi.com/projects/465731394728267777

"It shows my first letter of my name"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Tiny Tapeout Template Copy

by **Alicia**

0198 Wokwi Project

github.com/AliciaRRs/tt-template-copy

wokwi.com/projects/465731458535202817

"Test"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Very Low Resource Digital Implementation of Bioimpedance Analysis

by **Fabien**

0199 | 1 MHz | HDL Project

github.com/fabien-soulier/tt-gf26A-bioimpedance

"Digital bioimpedance demodulator with 8-stage I/Q cascade and UART output"

How it works

This project is a digital bioimpedance demodulator made in GF180 technology. It uses eight I/Q stages. Each stage takes a 1-bit input signal from a sigma-delta ADC and a clock The clock is divided by two at every stage. This means each stage works at a lower frequency than the one before. Each stage gives two values of 8 bits: one for I and one for Q. All eight stages produce 128 bits in total. These bits are stored in a register. When the data is ready, a signal triggers transmission. The UART module then sends the 16 bytes on the TX pin (uio[0]). Results can also be read via the 16 to 1 MUX using the MUX_ADDR pins

How to test

Turn on the board with the Tiny Tapeout setup. Connect the 1-bit ADC signal to ui[0]. Reset the chip by pulling rst_n low, then high. Use ui[1:4] (MUX_ADDR[0:3]) to select which byte to read on uo[0:7] (MUX_OUT). Read the UART output on uo[0] with 9600 bauderate
The chip sends 16 bytes. For each stage, you receive two bytes: first Q, then I. Stage 0 is the fastest. Stage 7 is the slowest. You can also use a USB-UART adapter and a serial terminal to see the data (Putty for example). To read via MUX:set MUX_ADDR (ui[1:4]) to select the desired byte (0 to 15). Read the result on uo[0:7] (MUX_OUT).

External hardware

A 1-bit signal source (for example a sigma-delta ADC) connected to ui[0]. A USB-UART adapter connected to uio[0] (TX) for serial readout.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ADC_IN	MUX_OUT[0]	TX
1	MUX_ADDR[0]	MUX_OUT[1]	CLK_SORTIE
2	MUX_ADDR[1]	MUX_OUT[2]	DAC_OUT
3	MUX_ADDR[2]	MUX_OUT[3]	QOUT
4	MUX_ADDR[3]	MUX_OUT[4]	—
5	BASCULE_IN	MUX_OUT[5]	—
6	—	MUX_OUT[6]	—
7	—	MUX_OUT[7]	—

UTOSS RISC-V core

by **University of Toronto Open Source Students**

0295 HDL Project

github.com/UTOSS/risc-v-ttgf

"RV32I core implementation"

How it works

To be done

How to test

To be done

External hardware

To be done

Project Pinout

Digital Pins

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

Hardware Entropy Explorer: UART/SPI TRNG and PUF

by **gojimmypi**

352 | 25 MHz | HDL Project

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

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

How it works

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

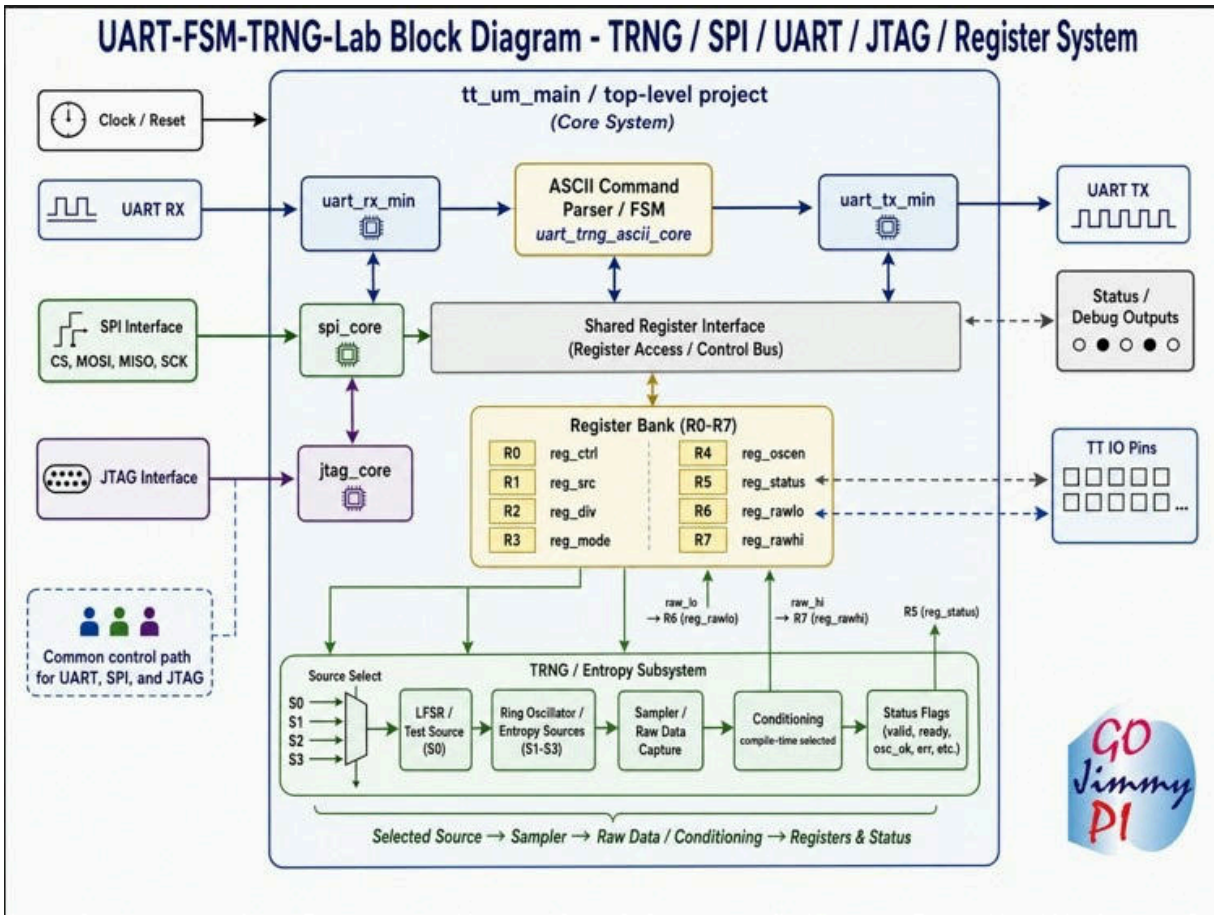


Figure 352.1: UART-FSM-TRNG-Lab-block-diagram.png

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

At a high level:

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

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

See presentations:

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

SPI vs JTAG Special Note

JTAG is experimental only.

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

For additional related information:

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

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

External Hardware

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

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

FPGA Tests

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

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

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

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

Commander App Tests

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

---

How to test

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

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

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

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

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

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

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

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

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

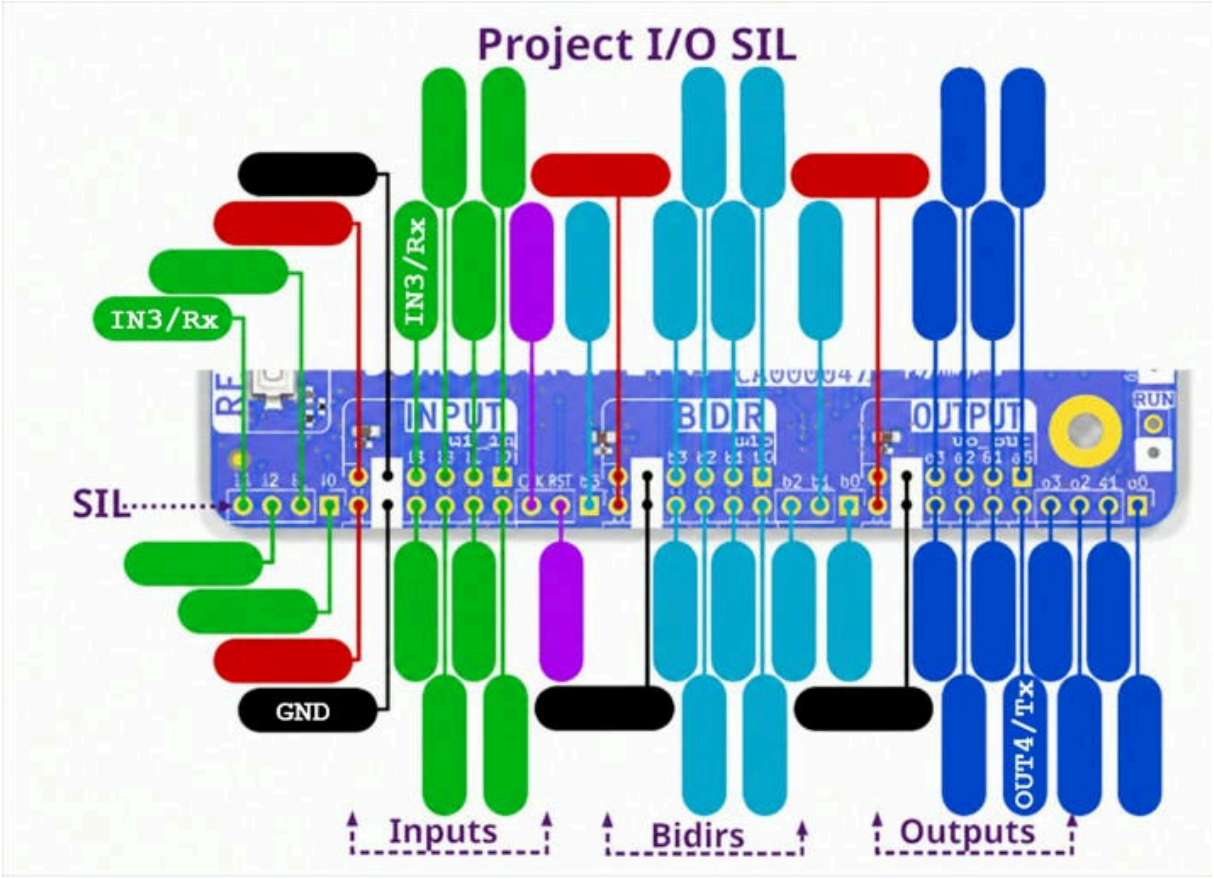


Figure 352.2: PMOD-connector-test1.png

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

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

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

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

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

Project config:

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

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

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

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

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

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

Terminal session at 25 MHz clock is

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

Or:

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

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

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

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

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

NIST Validation

NIST has a Resource for Random Bit Generation testing:

Overview

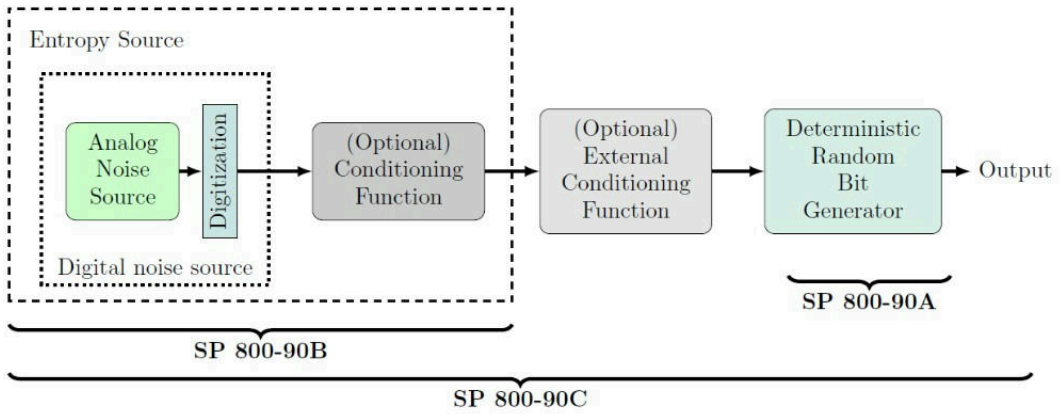
The National Institute of Standards and Technology (NIST) Random Bit Generation (RBG) project focuses on the development and validation of generating random numbers that are essential for cryptographic and security applications.

SP 800-90 Series

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

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

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



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

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

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

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

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

This script requires a build with `TRNG_BINARY_STREAM` enabled.

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

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

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

See also:

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

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

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

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

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

Quickstart Simulation

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

./my_test.sh

./jtag_test.sh
```

Quickstart Testing on TT Demoboard

If all the toolchains are installed:

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

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

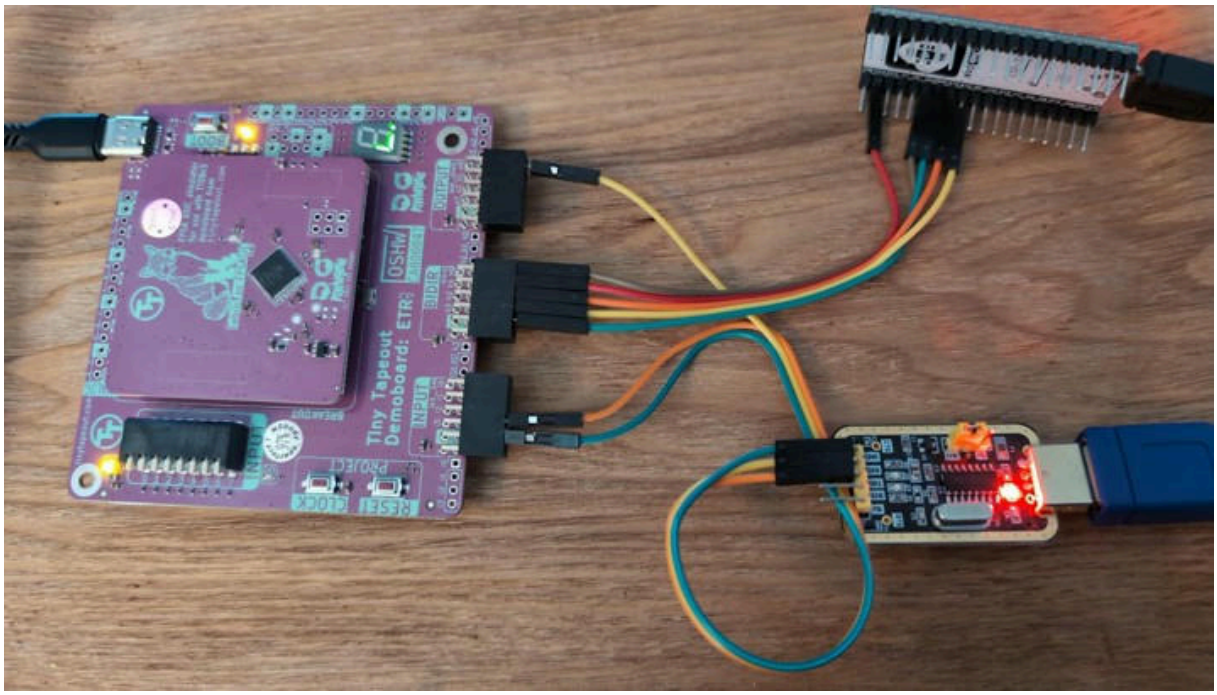


Figure 352.3: TT-Demoboard-SOFT_UART-SOFT-SPI-Wiring.jpg

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

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

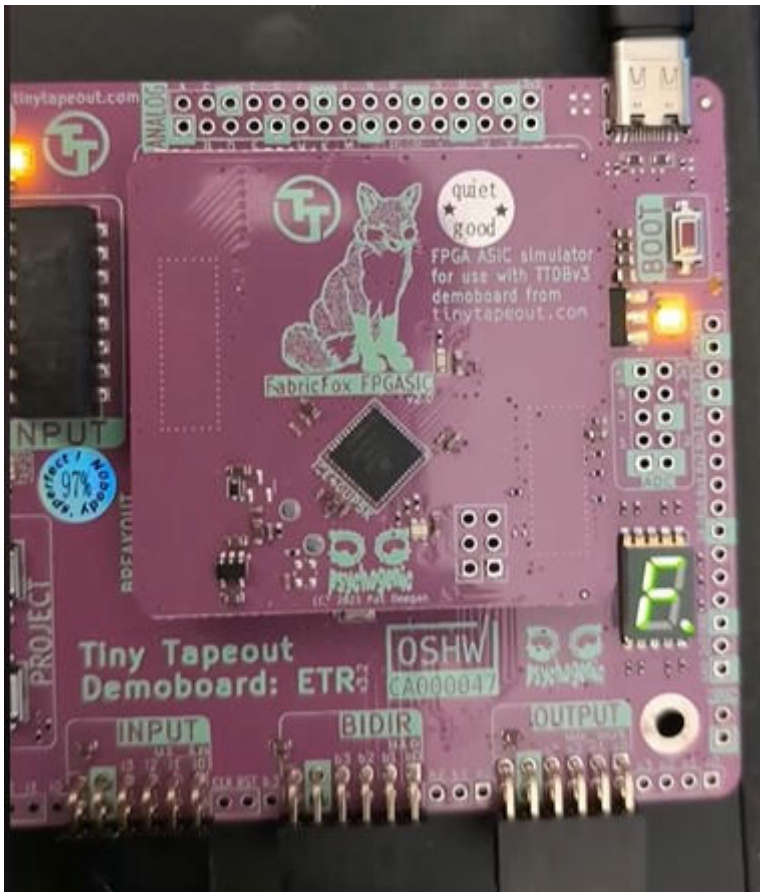


Figure 352.4: Demoboard_F_is_for_Fun_Success.jpg

From youtube.com/shorts/zFnfsl1DQHE

Quickstart Testing on ULX3S

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

ULX3S Connections

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

Soft External UART

⚠️ Do not connect to 5V TTY

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

Soft SPI

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

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

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

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

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

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

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

ULX3S ESP32 SPI Pins

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

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

Stand-alone ESP32 SPI Pins

⚠ Do not use these pins on the ULX3S ESP32.

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

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

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

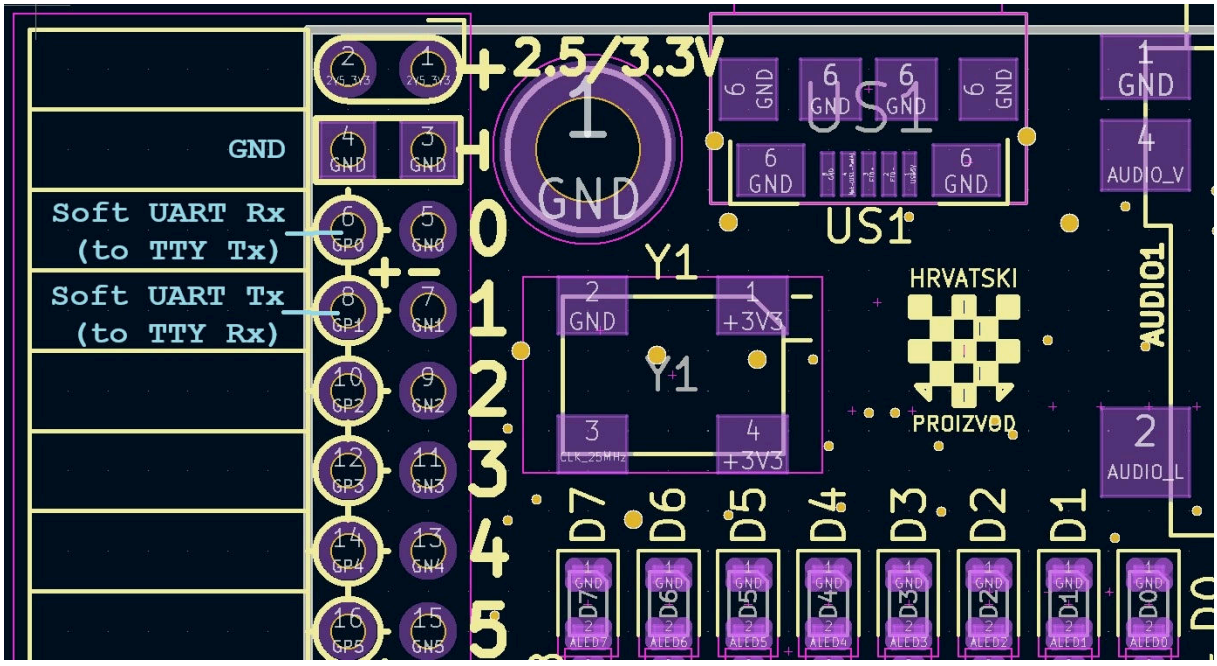


Figure 352.5: ULX3S-Pin-Connections.jpg

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

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

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

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

Quickstart on ULX3S ESP32

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

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

PORT=/dev/ttyS3

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

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

Register Overview

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

Key Concepts

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

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

Typical Flow

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

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

UART TRNG Command Interface

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

`R<n>=<value>`

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

Write Commands

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

Special:

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

Read Commands

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

Examples

Enable and configure:

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

Read back registers:

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

Version query:

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

Binary raw stream, when enabled:

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

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

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

Notes

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

UART

Connect with your favorite terminal program such as putty.

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

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

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

```
# UART pins for testing

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

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

Comprehensive Testing

There are TT simulation tests and local ULX3S FPGA tests.

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

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

Testing on the Tiny Tapeout FPGA Development Kit

See the overview video for the FPGA Development Kit.

Testing ULX3S / TT

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

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

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

Testing SPI with ESP32

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

Current testing scripts:

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

source ./export.sh

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TT Simulation tests

Commit changes. See results in actions.

In particular, note the output of the gds workflow:

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

Test on ULX3S FPGA

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

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

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

Example:

```
cd ulx3s

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

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

Local Loopback Test

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

Basic Loopback Test

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

```
    assign uart_tx_pin = uart_rx_sync;
```

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

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

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

Deep Loopback Test

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

Extensive loopback tests

Additional loopback tests:

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

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

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

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

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

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

```
cd "$TT_PROJECT_ROOT/test-hw

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

Local Automated Hardware Operation Tests

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

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

```
cd test-hw

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

UART FSM TRNG Lab Datasheet

Document revision: 1.0.5 RTL revision string: `Version 1.0.5 6/21/2026`
Project family: Tiny Tapeout UART/SPI configurable TRNG experiment
Primary top modules: `tt_um_gojimmypi_ttgf_UART_FSM_TRNG_Lab` (conditional based on build) License: Apache-2.0, as declared in the source files

1. Overview

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

The core supports multiple sample sources:

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

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

2. Key Features

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

3. Design Status and Intended Use

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

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

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

4. Source Files

The main RTL files are:

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

5. Top-Level Parameters

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

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

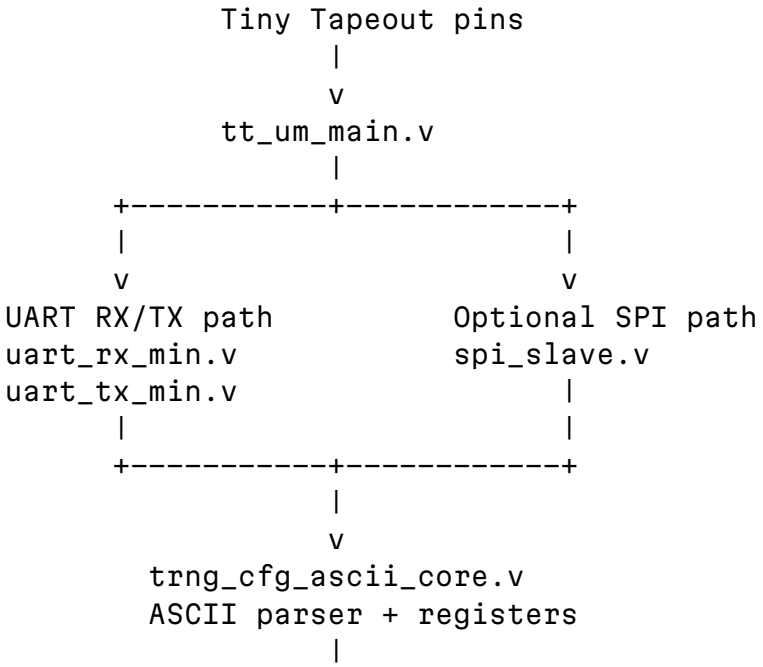
The ULX3S_USE_GN12_50MHZ configuration path can set PROJECT_CLOCK_HZ to 50 MHz for the optional ULX3S gn12 clock path. Normal builds use the 25 MHz project clock.

6. Build-Time Feature Defines

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

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

7. Functional Block Diagram



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

8. Clock and Reset

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

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

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

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

9. Tiny Tapeout Pin Map

Dedicated inputs: `ui_in[7:0]`

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

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

Dedicated outputs: `uo_out[7:0]`

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

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

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

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

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

Bidirectional IO when SPI is disabled

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

10. UART Interface

UART settings

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

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

UART command summary

Command	Arguments	Effect	Reply

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

Invalid syntax returns `?<CR>`.

UART command examples

```
V<CR>       -> Version 1.0.5 6/21/2026<CR>
R2<CR>      -> R2=10<CR>
E1<CR>      -> OK<CR>
D10<CR>     -> OK<CR>
S0<CR>      -> OK<CR>
O01<CR>     -> OK<CR>
R6<CR>      -> R6=HH<CR>
R7<CR>      -> R7=HH<CR>
```

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

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

11. SPI Interface

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

SPI electrical/protocol settings

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

SPI command byte

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

SPI read transaction

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

For example, reading `reg_rawlo` at address 6:

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

The useful read value is the second received byte.

SPI write transaction

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

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

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

```
TX: 02 10
```

12. Register Map

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

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

13. Control Register: `reg_ctrl`, Address 0

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

UART aliases:

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

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

14. Source Register: `reg_src`, Address 1

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

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

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

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

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

```
sample_ctr >= reg_div
```

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

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

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

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

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

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

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

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

The default build contains 7 .. 21 stages:

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

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

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

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

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

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

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

19. Raw Output Registers: reg_rawlo and reg_rawhi

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

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

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

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

20. Sampling Behavior

A sample event occurs when either condition is true:

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

Where:

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

On each sample event:

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

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

21. LFSR Details

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

```
0x1ACE
```

The next LFSR bit is computed as:

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

The LFSR then shifts as:

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

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

22. Ring Oscillator Path

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

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

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

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

23. Recommended Bring-Up Sequence

A conservative UART bring-up sequence is:

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

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

24. Known Deterministic Regression Sequence

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

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

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

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

25. UART and SPI Concurrency Notes

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

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

26. Limitations

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

27. Characterization Recommendations

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

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

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

28. Revision History

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

Example Outputs

- Example ESP32 Output
- Example Text Results

Project Pinout

Digital Pins

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

DOTTEE VGA demo (TTGF26a)

by **algofoogle (Anton Maurovic)**

`353` `25.175 MHz` `HDL Project`

`github.com/algofoogle/dottee-demo`

"DOTTEE: Pretty little 1-tile Tiny Tapeout ASIC demo (TTGF26a version)"

How it works

DOTTEE is a Verilog/ASIC demoscene project, originally submitted to the Tiny Tapeout ttsky26a Demoscene Competition in the 1-tile category. **This is an enhanced version**, resubmitted to the TTGF26a shuttle for fun.

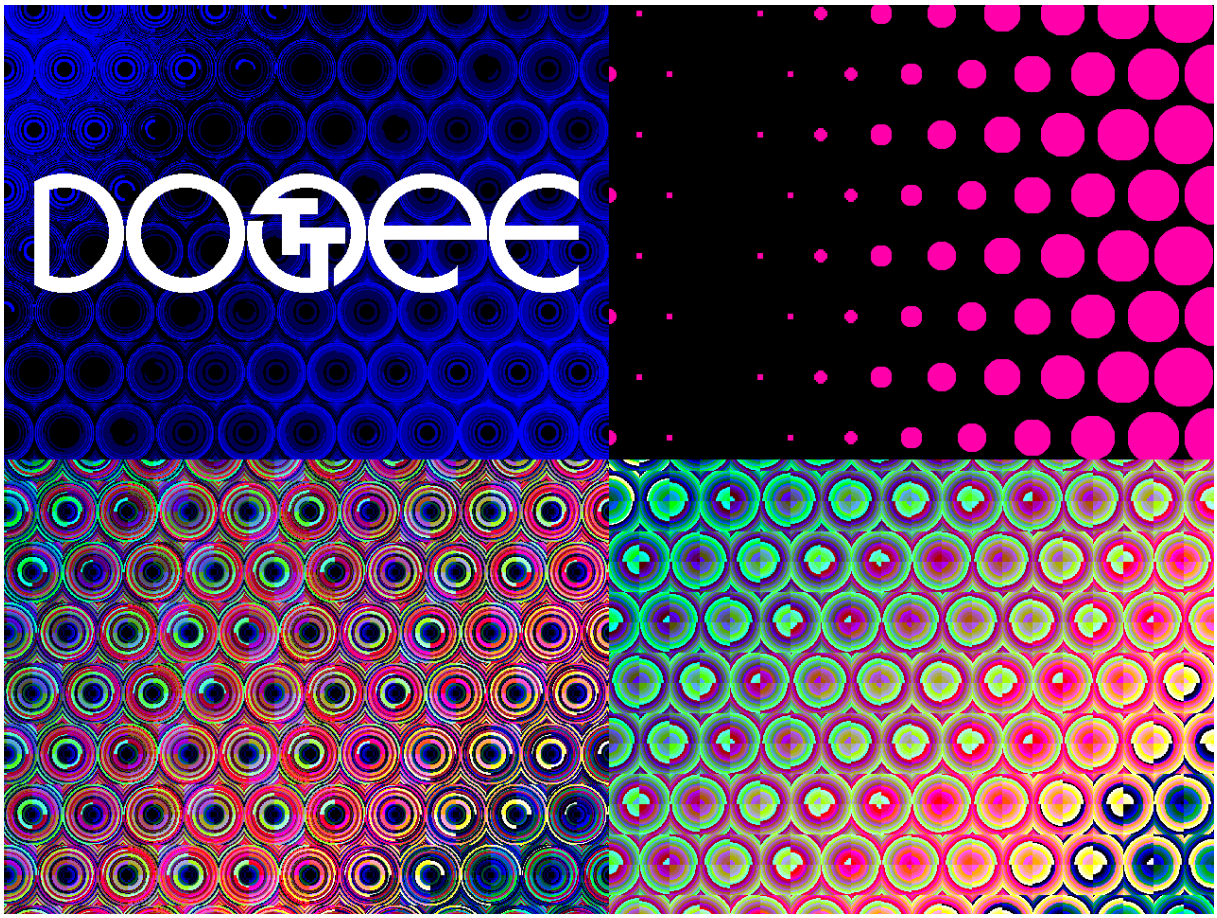


Figure 353.1: Screenshots of some scenes in the DOTTEE demo

This demo races the beam to create a VGA 640x480 60fps output, while generating some music using a 1-bit sigma-delta DAC. It loops after 4096 frames (about 68 seconds). The main theme of the demo is: **dots** (well, circles). It uses a rough approximation of squared Euclidean distance for per-pixel rendering of an array of multi-coloured circles of various radii. It also uses a Bresenham-like circle calculation to slowly compute the edges of two

larger circles (during HBLANK) which are used to construct the “DOTTEE” logo (where the ‘TT’ in that is a rendition of the Tiny Tapeout logo).

This demo also synthesises some music using a couple of voices (i.e. phase accumulators) that generate triangle waves (but also mangle them a bit to create other effects). New to this TTGF26a version too is a simple noise generator hack (based on vertical line index and some other bit-mangling) that adds simple percussive sounds. Audio output is by converting 6-bit audio samples (at the HSYNC frequency of about 31.5kHz) to 1-bit PDM (at `clk` frequency; i.e. about 25MHz) using a sigma-delta DAC.

How to test

Keep `ui_in` bits all set to `0`. Attach a Tiny VGA PMOD adapter to the output (`uo_out`) port and a Tiny Audio PMOD adapter to the bidirectional (`uio`) port, supply a 25MHz clock, and reset the design. Watch and listen!

`ui_in` also gets ORed with the upper 8 bits of the 12-bit frame counter (from which all stages and timing of the demo are derived), so it can be used for debugging by jumping ahead in multiples of 16 frames.

External hardware

- Tiny VGA PMOD for driving a VGA monitor.
- TT Audio PMOD for mono sound.

Attribution

By all means feel free to build upon this project with your own work, but please acknowledge this project and credit me (Anton Maurovic, https://github.com/algofoogle, https://foogle.com/anton) as the original author.

Project Pinout

Digital Pins

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

8-bit RISC CPU

by **Jan Kozina**

`354` `5 MHz` `HDL Project`

`github.com/hofi505/risc-v-tt26a`

"8-bit RISC CPU with ALU, registers, data memory and external 16-bit instructions"

Overview

This project is an 8-bit RISC CPU written in Verilog for Tiny Tapeout GF. The CPU is intentionally small and simple: it has no internal program ROM, so an external controller provides one 16-bit instruction at a time through the Tiny Tapeout input pins.

The design executes the instruction and returns either the instruction result or the current program counter on the 8-bit output bus.

Main building blocks:

- 8-bit program counter
- 8 x 8-bit register file
- 8-bit ALU
- immediate generator
- branch-control logic
- 16-byte data memory

The target clock configured for this project is 5 MHz.

Pin Interface

The instruction bus is 16 bits wide and is split across `ui_in` and `uio_in`. The bidirectional `uio` pins are used only as inputs.

Tiny Tapeout signal	Direction	CPU signal
`ui_in[7:0]`	input	`instruction[15:8]`
`uio_in[7:0]`	input	`instruction[7:0]`
`uo_out[7:0]`	output	CPU result / PC output
`uio_out[7:0]`	output	Always `0`
`uio_oe[7:0]`	output	Always `0`
`clk`	input	system clock
`rst_n`	input	active-low reset
`ena`	input	Tiny Tapeout enable, unused internally

To drive an instruction:

```
ui_in  = instruction[15:8]
uio_in = instruction[7:0]
```

For example, instruction `0x1843` is driven as:

```
ui_in  = 0x18
uio_in = 0x43
```

This means `ui_in[0]` carries `instruction[8]`, and `uio_in[0]` carries `instruction[0]`.

Execution Timing

The CPU alternates between two internal phases:

1. NOP / PC phase
2. execute phase

For normal use, keep each instruction stable for two clock cycles.

During the execute phase, `uo_out` shows the instruction result. On the next clock edge, register, memory, and PC updates are committed. During the following NOP / PC phase, `uo_out` shows the current program counter.

One instruction slot looks like this:

Moment	External action / observation
Before clock edge 1	Drive the 16-bit instruction and keep it stable
After clock edge 1	`uo_out` shows the instruction result
Clock edge 2	The register file, data memory, and PC update
After clock edge 2	`uo_out` shows the updated PC

Typical external-controller sequence:

1. Drive `rst_n = 0` for reset.
2. Release reset with `rst_n = 1`.
3. Put a 16-bit instruction on `{ui_in, uio_in}`.
4. Hold the instruction stable for two clock cycles.
5. Read the result from `uo_out`.
6. Change to the next instruction and repeat.

Architecture Details

Register File

There are eight 8-bit general-purpose registers, `r0` through `r7`.

After reset:

Register	Value
r0	0
r1	0
r2	0
r3	0
r4	0
r5	0
r6	0
r7	0

Data Memory

The data memory contains 16 bytes. Address calculations are 8-bit, but only the lower 4 address bits select the memory entry. In other words, data-memory addresses wrap modulo 16.

Example:

`0x08, 0x18, and 0xF8 all access memory index 8`

Program Counter

The program counter is 8 bits wide. Normal instructions increment the PC by 1. Taken branches update the PC using:

```
next_pc = pc + imm6
```

PC arithmetic wraps modulo 256.

Instruction Encoding

This is a compact custom ISA for this Tiny Tapeout design, not a standard RISC-V-compatible encoding.

The opcode is always stored in `instruction[15:12]`.

R-Type Format

R-type instructions use two source registers, one destination register, and a 3-bit ALU function field.

Bits	Field
[15:12]	opcode, always `0000`
[11:9]	destination register `rd`
[8:6]	source register `rs1`
[5:3]	source register `rs2`
[2:0]	ALU function `funct3`

Encoding formula:

```
instruction = (rd << 9) | (rs1 << 6) | (rs2 << 3) | funct3
```

Immediate Format

Most non-R-type instructions use a source/base register and a 6-bit immediate.

Bits	Field
`[15:12]`	opcode
`[11:9]`	destination register `rd`, or unused depending on instruction
`[8:6]`	source/base register `rs1`
`[5:0]`	immediate `imm6`

Encoding formula:

```
instruction = (opcode << 12) | (rd << 9) | (rs1 << 6) | (imm6 &
0x3F)
```

The `imm6` value is sign-extended to 8 bits before ALU use. For example, `imm6 = 0x3F` represents `–1`, which becomes `0xFF` in the 8-bit datapath.

The `LI` instruction is a special case:

```
instruction = (0x2 << 12) | (rd << 9) | imm8
```

Store And Branch Format

Store and branch instructions use a split 6-bit immediate so that `rs2` can be encoded independently.

Bits	Field
`[15:12]`	opcode
`[11:9]`	immediate `imm6[5:3]`
`[8:6]`	source/base register `rs1`
`[5:3]`	source register `rs2`
`[2:0]`	immediate `imm6[2:0]`

Encoding formula:

```
instruction = (opcode << 12)
            | (((imm6 >> 3) & 0x7) << 9)
            | (rs1 << 6)
            | (rs2 << 3)
            | (imm6 & 0x7)
```

For `SW`, `rs2` selects the register whose value is written to data memory. For branch instructions, `rs1` and `rs2` are compared, while `imm6` is used as the signed PC offset when the branch is taken.

Instruction Set

Opcode	Mnemonic	Operation
`0000`	R-type	ALU operation selected by `funct3`
`0001`	ADDI	`rd = rs1 + imm6`
`0010`	LI	`rd = imm8`
`0011`	LW	`rd = memory[rs1 + imm6]`
`0100`	SW	`memory[rs1 + imm6] = rs2`
`0101`	BEQ	branch when `rs1 == rs2`
`0110`	BNE	branch when `rs1 != rs2`
`0111`	BLT	branch when `rs1 < rs2`
`1000`	ANDI	`rd = rs1 & imm6`
`1001`	ORI	rd = rs1 | imm6
`1010`	XORI	`rd = rs1 ^ imm6`
`1011`	SLLI	`rd = rs1 << imm6`
`1100`	SRLI	`rd = rs1 >> imm6`
`1111`	NOP	no operation

NOP and unsupported opcodes leave the PC, registers, and data memory unchanged.

ALU Functions

R-type instructions use `funct3`.

`funct3`	Operation
`000`	ADD
`001`	SUB
`010`	AND
`011`	OR
`100`	XOR
`101`	Shift left
`110`	Shift right
`111`	Defaults to ADD

Shift operations use the lower three bits of the second ALU operand as the shift amount, giving a range from 0 to 7 positions.

The branch instructions use the SUB path internally to compare `rs1` and `rs2` and generate comparison flags. The less-than comparison is signed, so both operands are interpreted as 8-bit two's-complement values.

Worked Examples

Example 1: Basic Register And ALU Sequence

Step	Instruction	Encoding	Result on `uo_out`
1	`LI r1, 4`	`0x2204`	4
2	`ADDI r4, r1, 3`	`0x1843`	7
3	`ADDI r5, r4, 2`	`0x1B02`	9
4	`ADD r6, r4, r5`	`0x0D28`	16

Explanation:

```
r1 = 4
r4 = 4 + 3 = 7
r5 = 7 + 2 = 9
r6 = 7 + 9 = 16
```

Example 2: Load Immediate And Negative Immediate

Instruction	Encoding	Result
`LI r6, 42`	`0x2C2A`	`r6 = 42`
`ADDI r7, r6, -1`	`0x1FBF`	`r7 = 41`

The second instruction uses `imm6 = 0x3F`, which sign-extends to `0xFF` and acts as `-1` in the 8-bit datapath.

Example 3: Store And Load

Step	Instruction	Encoding	Effect
1	`LI r1, 4`	`0x2204`	`r1 = 4`
2	`ADDI r4, r4, 16`	`0x1910`	`r4 = 16`
3	`SW r1, [r4 + 8]`	`0x4308`	writes 4 to memory index 8
4	`LW r6, [r4 + 8]`	`0x3D08`	`r6 = 4`

The address is `16 + 8 = 24`. Because data memory uses only the lower 4 address bits, address 24 accesses memory index 8.

Example 4: Taken Branch

Step	Instruction	Encoding	Behavior
1	`LI r1, 4`	`0x2204`	`r1 = 4`
2	`BEQ r1, r1, +4`	`0x504C`	branch is taken

After `LI`, the current PC is 1, so the taken branch updates it to:

```
pc + imm6 = 1 + 4 = 5
```

Running The RTL Simulation

The repository includes a cocotb end-to-end testbench that drives instructions through the Tiny Tapeout top-level pins and checks reset behavior, ALU instructions, immediate instructions, memory access, branches, and PC updates.

From the repository root:

```
cd test
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
make
```

The expected result is:

```
TESTS=1 PASS=1 FAIL=0
```

Running Gate-Level Simulation

After local hardening or a successful GDS flow, copy the powered netlist into the test directory and run the test with `GATES=yes`.

For GF180 local hardening, the netlist is normally:

```
runs/wokwi/final/pnl/tt_um_8bit_risc_cpu.pnl.v
```

Example:

```
export PDK=gf180mcuD
export PDK_ROOT=/path/to/pdk/root

cp runs/wokwi/final/pnl/tt_um_8bit_risc_cpu.pnl.v test/
gate_level_netlist.v
cd test
make -B GATES=yes
```

The expected result is again:

```
TESTS=1 PASS=1 FAIL=0
```

Running Local Hardening

This project targets Tiny Tapeout GF, so Tiny Tapeout tooling should be run with the `--gf` flag.

If the `tt/` support-tools directory is not present, clone it first:

```
git clone https://github.com/TinyTapeout/tt-support-tools tt
```

Typical local setup:

```
python3 -m venv ~/ttsetup/venv
source ~/ttsetup/venv/bin/activate
pip install --upgrade pip
pip install -r tt/requirements.txt

export PDK_ROOT=~/ttsetup/pdk
export PDK=gf180mcuD
export LIBRELANE_TAG=3.0.3
pip install librelane==$LIBRELANE_TAG
```

Create the merged config and run hardening:

```
./tt/tt_tool.py --create-user-config --gf
./tt/tt_tool.py --harden --gf
./tt/tt_tool.py --print-warnings --gf
```

If the PDK is installed through Ciel but gate-level simulation cannot find `gf180mcuD`, enable the installed GF180 PDK:

```
ciel enable --pdk-root ~/ttsetup/pdk --pdk-family gf180mcu
<version>
```

Then rerun the hardening or gate-level simulation command.

If hardening succeeds, the final GDS, LEF, netlists, and metrics are placed under:

```
runs/wokwi/final/
```

Hardware Usage

On real Tiny Tapeout hardware, an external controller such as a microcontroller, FPGA, or test setup must provide the clock, reset, and instruction input.

The most important rule is to hold each instruction stable for two clock cycles. Then read `uo_out` during the execute phase for the result and during the next NOP / PC phase for the updated program counter.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	instruction[8]	CPU_Out[0]	instruction[0]
1	instruction[9]	CPU_Out[1]	instruction[1]
2	instruction[10]	CPU_Out[2]	instruction[2]
3	instruction[11]	CPU_Out[3]	instruction[3]
4	instruction[12]	CPU_Out[4]	instruction[4]
5	instruction[13]	CPU_Out[5]	instruction[5]
6	instruction[14]	CPU_Out[6]	instruction[6]
7	instruction[15]	CPU_Out[7]	instruction[7]

Simon Says memory game

by **Uri Shaked**

`0355` `50 kHz` `HDL Project`

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

"*Repeat the sequence of colors and sounds to win the game*"

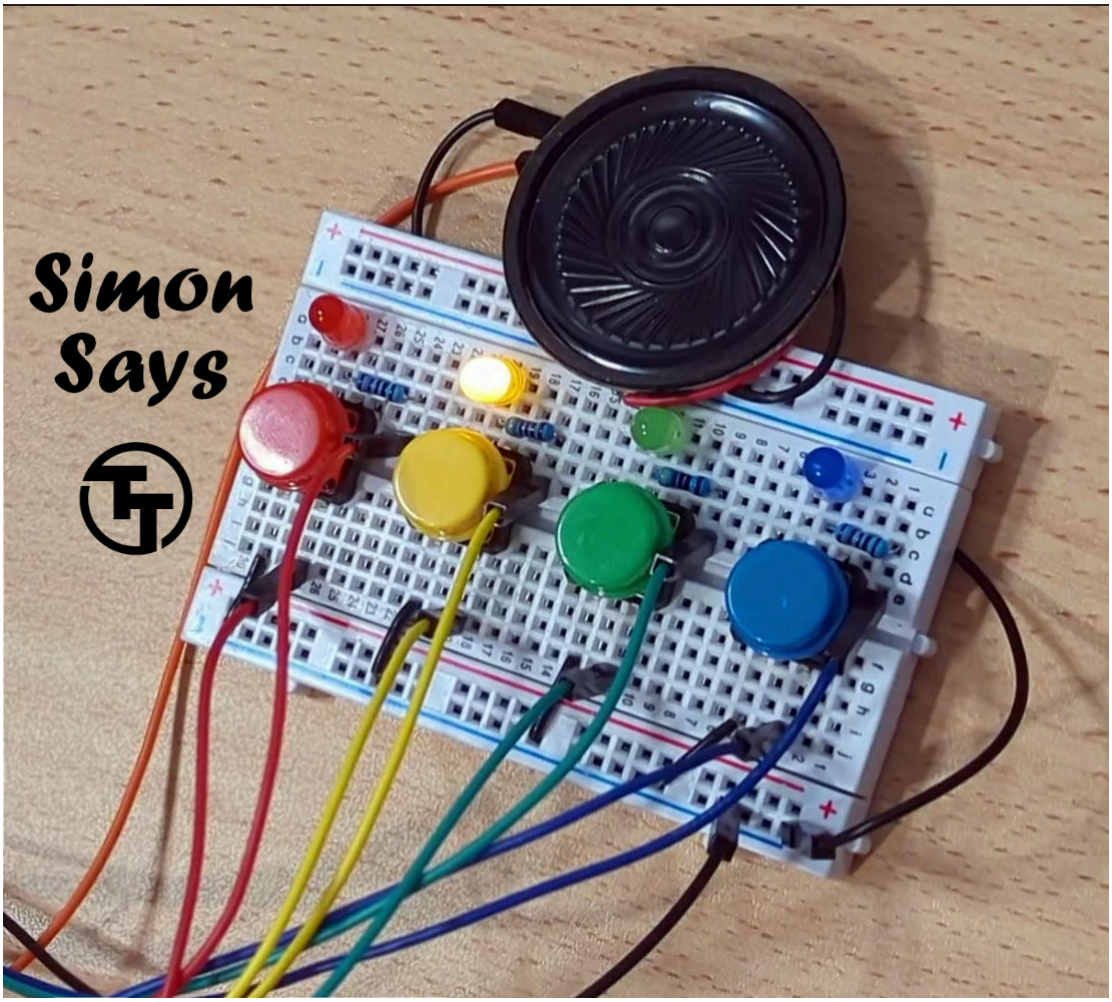


Figure 355.1: Simon Says Game

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	—

RHD2164-MCU-SPI Bridge

by **Manuel Monge**

`0356` `25 MHz` `HDL Project`

`github.com/manuel-monge/tt2606`

"Various SPIs"

How it works

This design implements a custom SPI module to interface with `RHD2164` neural amplifier chips and selects specific channels to transfer the data to a MCU via SPI. The design also implements a register bank accessible either by SPI or scanchain/shift register.

How to test

To be added.

External hardware

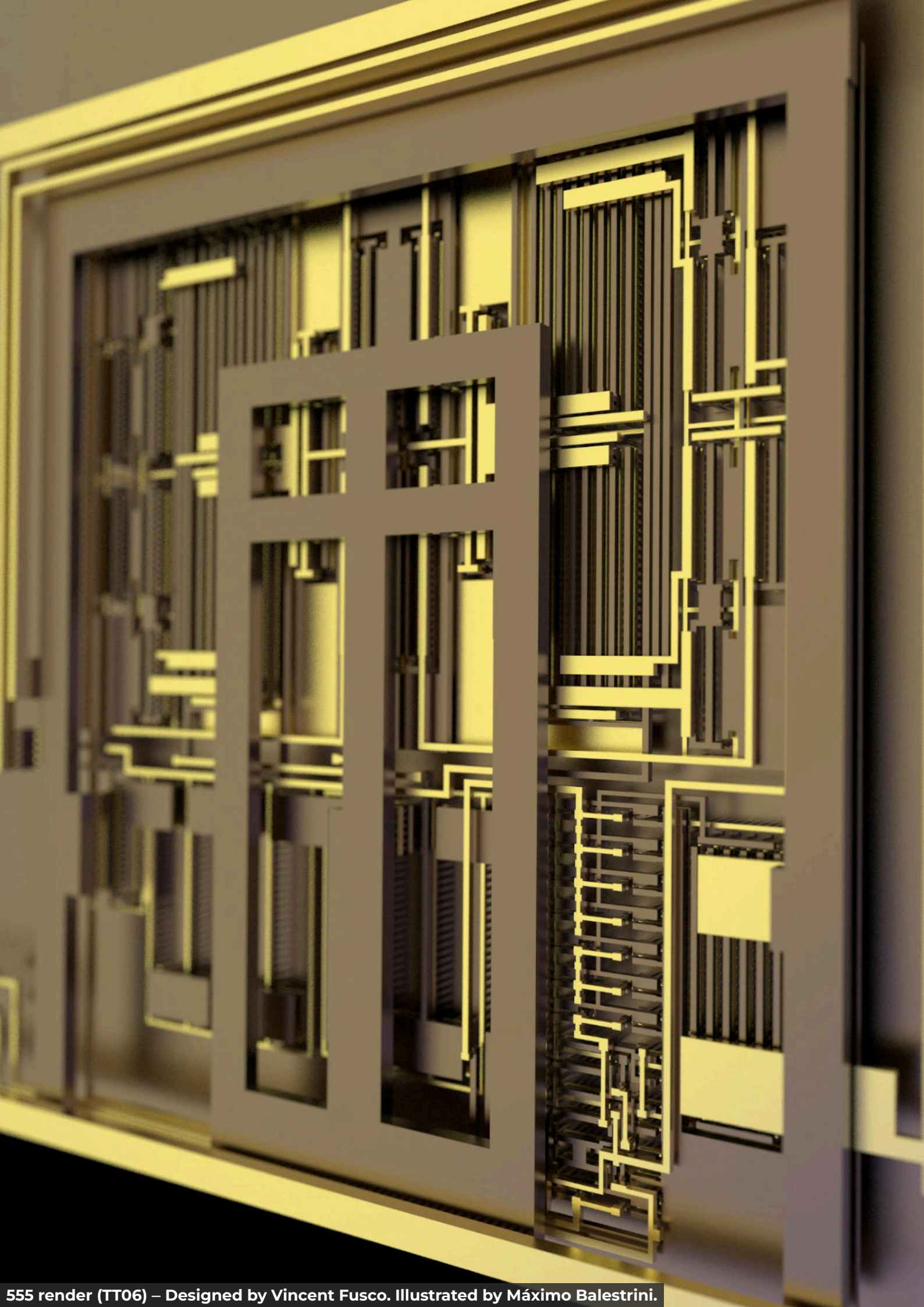
You can test this chip in two different ways.

1. Connect the chip to an FPGA and use the RHD2164 emulator and MCU-SPI emulator available here. The register bank is tested using any MCU.
2. Connect the chip to a `RHD2164` neural amplifier chip in CMOS mode or in LVDS mode using `LVDS-to-CMOS` adapters. The selected data is connected to any MCU which is also connected to the register bank.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SC_SDI	RHD_MOSI	RHD_MISO0
1	SC_SCLK	RHD_SCK	RHD_MISO1
2	SC_SEN	RHD_CSb	MCU_MOSI
3	REG_MOSI	MCU_MISO	MCU_SCK
4	REG_SCK	SC_SDO	MCU_CSb
5	REG_CSb	REG_MISO	SELM0
6	SR_CLK	SR_DO	—
7	SR_EN	—	—



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

First Tinytapeout

by **Raúl**

`0357` `Wokwi Project`

`github.com/raullopez54/tt-first-tinytapeout`

`wokwi.com/projects/465483277165299713`

"First deployment with TinyTapeOut"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Stochastic neuron + STDP controller (merged, GF180)

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

358 | 25 MHz | HDL Project

github.com/SanthoshSivasubramani/tt_um_santhosh_stoch_stdp_pair_gf_pub

> *"GF180mcuD merged-pair cell: an 8-bit saturating LIF/stochastic neuron with an 8-entry sigmoid LUT, 8-bit Galois LFSR, programmable refractory, per-clock leak, host-triggered stimulus on ext_input rising edge, homeostatic theta adaptation over a 256-clock window (theta_live floats between theta_min and theta_max depending on spike count vs reg_target), and a 16-clock Bayesian confidence window emitting a 2-bit conf output. The second block is an STDP controller with 8-bit pre/post spike traces, per-clock exponential-shift decay (reg_tau_plus / reg_tau_minus), an anti-Hebbian polarity bit and optional reward-gated learning; on every LTP event (post spike with pre trace != 0) or LTD event (pre spike with post trace != 0) a 1-cycle weight_update pulse is emitted along with a learn_dir bit and a magnitude latch read over SPI. All registers are programmable via an 8-bit-data SPI mode-0 slave shared with the other TTGF26a tiles. 25 MHz signoff."*

TTGF26a Scenario B tile 1 — merged stochastic/LIF neuron + STDP learning controller on GF180mcuD.

How it works

The tile combines an 8-bit saturating leaky-integrate-and-fire (LIF) neuron with a stochastic mode (fire when `lfsr < sig_lut[v_mem[7:5]]`), homeostatic theta adaptation over a 256-clock window, and a 16-clock Bayesian confidence counter. A second block is an STDP controller with 8-bit pre/post traces, per-clock exponential-shift decay, an anti-Hebbian polarity bit, and optional reward-gated learning. On every LTP or LTD event the controller emits a 1-cycle `weight_update` pulse together with `learn_dir` and latches the magnitude in `R_LAST_MAG` for host readback.

All registers are programmable via a shared 8-bit SPI mode-0 slave (same module used by the other TTGF26a tiles).

How to test

1. Program the neuron (`R_STIM`, `R_THETA`, `R_LEAK`, `R_SIG_LUT[0..7]`, `R_POLY`, `R_TARGET`, `R_THETA_MIN`, `R_THETA_MAX`) and the STDP con-

troller (`R_TAU_PLUS`, `R_TAU_MINUS`, `R_LTP_LUT[0..7]`, `R_LTD_LUT[0..3]`, `R_STDP_CTRL`) via SPI.
2. Drive `ui_in[0]` to pick LIF or stochastic mode.
3. Assert `ui_in[4]` (or set CTRL[3]) to enable homeostatic adaptation.
4. Assert `ui_in[5]` (or set CTRL[4]) to enable Bayesian confidence counting.
5. Drive `ui_in[2]` high and write `R_STDP_CTRL[0]=1` to enable learning.
6. Pulse `ui_in[1]` (`ext_input`) once per presynaptic spike; observe `uo_out[0]` = post-synaptic spike and `uo_out[1]` = 1-cycle weight-update pulse.
7. Read `R_VMEM`, `R_TRACE_PRE`, `R_TRACE_POST`, `R_LAST_MAG`, `R_BAYES_CONF`, `R_THETA_LIVE`, `R_STATUS` via SPI to observe state.

External hardware

Only a microcontroller (or FPGA) driving the SPI slave is required. Any spike-train source can be wired to `ui_in[1]`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	mode_sel (0=LIF, 1=stochastic)	spike_out (post-synaptic spike)	spi_cs_n (input)
1	ext_input (pre-synaptic spike / drive)	weight_update (1-cycle STDP pulse)	spi_mosi (input)
2	learn_en	theta_hi (theta_live at ceiling)	spi_miso (output)
3	reward (R-STDP gate)	theta_lo (theta_live at floor)	spi_sck (input)
4	homeostatic_en	bayes_conf[0]	reserved (input, tied off)
5	bayes_en	bayes_conf[1]	reserved (input, tied off)
6	reserved	learn_dir (1=LTP, 0=LTD)	reserved (input, tied off)
7	reserved	busy (refractory OR weight_update in flight)	reserved (input, tied off)

Tiny Tapeout testtest 111233

by **Suresh Devanathan**

0359 Wokwi Project

github.com/idatarm/tt-testtest-111233

wokwi.com/projects/467219410242853889

"Test project for segment display"

How it works

this is just test

How to test

this is not aimed at specific criteria

External hardware

none

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in0	out0	—
1	in1	out1	—
2	in2	out2	—
3	in3	out3	—
4	in4	out4	—
5	in5	out5	—
6	in6	out6	—
7	in7	out7	—

teenytpu

by **joannec34**

384 | 50 MHz | HDL Project

`github.com/joannec34/teenytpu`

"2x2 INT8 systolic-array TPU with SPI memory interface"

How it works

TeenyTPU is a 2x2 INT8 systolic array TPU designed. At its core, it features a 2x2 grid of Processing Elements (PEs) that can perform matrix multiplication operations on INT8 data, resulting in 16-bit partial sum sequences.

SPI Pin Mapping

- **`ui[0]`**: SPI SCLK
- **`ui[1]`**: SPI CS_N
- **`ui[2]`**: SPI MOSI
- **`uo[0]`**: SPI MISO
- **`uo[1]`**: BUSY flag
- **`uo[2]`**: DONE flag

Architecture Overview

- **SPI Slave Bridge** — deserialises SPI transactions from an external host into register writes and command pulses. Uses 2-FF CDC synchronisers for SCLK, CS_N, and MOSI. Supports five opcodes:
  - `0x01 WRITE_WEIGHT` — load an 8-bit weight into a selected column/row.
  - `0x02 LOAD_INPUT` — load an 8-bit activation into a selected row.
  - `0x03 CMD_START` — trigger the compute FSM.
  - `0x04 READ_RESULT` — read back two 16-bit partial sums from a selected column.
  - `0x05 READ_STATUS` — read the `{done, busy}` status byte.
- **Control FSM** — sequences weight loading, shadow→active switch, activation feeding, draining, and result readback through six states (`IDLE → LOAD_W → SWITCH → FEED → DRAIN → DONE`).
- **2×2 Systolic Array** — four PE instances connected in a grid. Activations flow left -> right, partial sums flow top -> bottom, and weights propagate top→bottom during loading.

How to test

To test the TeenyTPU, you must implement an SPI master to drive the designated `ui` pins.

1. **Hardware Reset**: Provide a clock signal (`clk`) and assert `rst_n` low briefly to reset the FSM and the systolic array.
2. **Load Weights**: Assert `CS_N` low, send the `0x01` opcode, followed by the column index (e.g., `0x00`), and then the two 8-bit weights for that column. Deassert `CS_N`. Repeat this for column 1.
3. **Load Activations**: Assert `CS_N` low, send the `0x02` opcode, followed by the row index (e.g., `0x00`), and the 8-bit activation value. Deassert `CS_N`. Repeat this for row 1.
4. **Trigger Computation**: Send the `0x03` opcode to initiate computation. The `busy` pin (`uo[1]`) will assert high.
5. **Poll Status**: Wait for the `done` pin (`uo[2]`) to assert high, which indicates that the systolic array has finished computation. Alternatively, you can use the `0x05` opcode to read the `{6'b0, done, busy}` status byte repeatedly.
6. **Read Results**: Send the `0x04` opcode followed by the column index (`0x00` or `0x01`). The TPU will respond by shifting out 16 bits of result data (`MISO`) representing the partial sums of that column.

External hardware

An SPI master connected to `ui[0]` (SCLK), `ui[1]` (CS_N), `ui[2]` (MOSI), and `uo[0]` (MISO).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	spi_sclk	spi_miso	—
1	spi_cs_n	busy	—
2	spi_mosi	done	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Tiny Tapeout Workshop JuanF

by **Juan**

385 | 100 Hz | Wokwi Project

github.com/jfm92/tt-workshop-juanf

wokwi.com/projects/465736492859711489

"*TinyTapeOutWorkshop*"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Pong in Verilog

by **Jorobraun**, **TnTim1** & **ColdC5**

`0386` `25.175 MHz` `HDL Project`

`github.com/ColdC5/TinyTapeoutGFPong`

"A simple Pong game with VGA-style output"

How it works

WIP

How to test

WIP

External hardware

WIP

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Player1 Down	R1	—
1	Player1 Up	G1	—
2	Player2 Down	B1	—
3	Player2 Up	HSync	—
4	Auto Mode	R0	—
5	—	G0	—
6	—	B0	—
7	—	VSync	—

Tiny Tapeout Template_1234

by **Najlae**

0387 | 19.999 kHz | Wokwi Project

github.com/neshla/tt-template1234

wokwi.com/projects/465732744934845441

“hello world project ”

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Tiny TPU Systolic Array

by **Yu Xia□Bohong Gao**, **Junjie Yan**, **Yufeng Wang**, **Jinyao Sun** & **Arthur Stanson**

388 | 100 MHz | HDL Project

github.com/bgao43-wq/tt-tiny-tpu-gf180

"A small systolic-array-based matrix multiplication accelerator"

This project implements a small TPU-style matrix multiplication accelerator using a systolic array architecture. The design is intended for TinyTapeout and uses a simple SPI-style serial interface to configure the design, load data, start computation, and read results.

How it works

The design is built around a small systolic array made of processing elements. Each processing element performs multiply-accumulate operations and passes data through the array. The controller receives commands through the SPI interface, manages configuration and execution, and sends results back through the serial output.

The TinyTapeout wrapper maps the external signals as follows:

- `ui_in[0]`: SCLK
- `ui_in[1]`: MOSI
- `ui_in[2]`: CS_N
- `uo_out[0]`: MISO

The system clock and reset use the standard TinyTapeout `clk` and `rst_n` pins.

How to test

The project can be tested using the TinyTapeout cocotb testbench flow. The testbench drives the TinyTapeout wrapper interface, applies reset, sends SPI command sequences through `ui_in`, and observes the serial output through `uo_out[0]`.

To run the test locally:

```
cd test
make -B
```

The GitHub Actions workflow also runs the RTL test automatically after pushing changes to the repository.

External hardware

No external hardware is required. The design only uses the TinyTapeout digital input, output, clock, and reset pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SCLK	MISO	—
1	MOSI	—	—
2	CS_N	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Arturo's first Wokwi design

by **Artcraft**

`#389` `Wokwi Project`

`github.com/Artcraftx107/tt-arturos-first-wokwi`

`wokwi.com/projects/465731371445677057`

"My first ever Wokwi design "

How it works

Its just a very simple 7-segment display, after putting in the code 201 in binary as the inputs, the display segments will light up and make an "A", my initial.

How to test

Just run it, adjust the inputs if needed.

External hardware

Nothing (I think xd)

Project Pinout

Digital Pins

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

GPS_Accelerator

by **jpigdon**

0390 | 2.2 MHz | HDL Project

`github.com/jpigdon/ttgf_gnss_accelerator`

"GNSS Accelerator to demonstrate acquisition and tracking on Tiny Tapeout "

How it works

Implements a basic GNSS receiver including functionality to synchronise in time and frequency with different GNSS constellations.

Works in conjunction with a GPS front end capable of providing single bit (and potentially multi bit I/Q streams along with an ADC sampling clock.

Design is targeting MAX2769 and MAX2771 from Analog Devices, though testing may not occur with hw due to time limitations.

A table of features and maturity is below:

Requirement ID	State	Description
1.0	☑️	Support search for acquisition
1.1	☑️	Generate Gold Codes for GPS L1
1.2	☑️	Generate NCO for frequency correction
1.3	☑️	Generate timing signal for frame
2.0	☑️	Provide SPI slave interface for command and control
2.1	☑️	support clock domain crossing between SPI clock and sample clock
2.2	☑️	Provide readback for acquisition registers
2.3	☑️	provide readback for tracking registers
3.0	☑️	Enable tracking channel for time, freq, sv settings
3.1	☑️	provide update mechanism for tracking channels
3.2	☑️	provide early/mid/late tracking for tracking channels to improve acquition performance
3.3	❌	support selectable correlation duration

How to test

Will need a custom load on the RP2050 on the TT development kit to manage software interactions.

Needs a source of GNSS data, either recorded or simulated, look in /data for example data

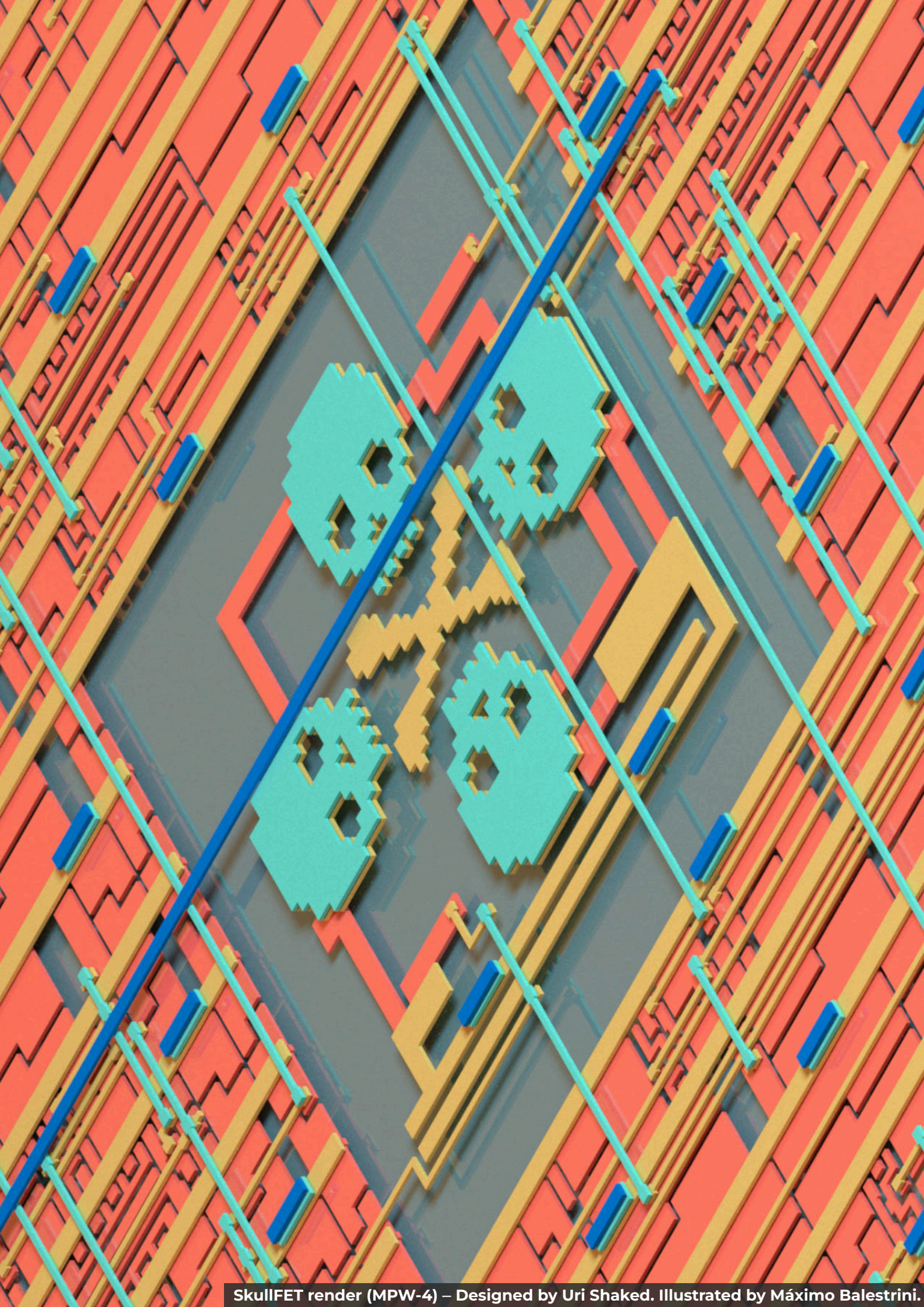
External hardware

Custom MAX2769 or MAX2771 board required, SPI interface to RP2050 for configuration and control.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	MAX2771 I0 - Quantised ADC I0 input stream	Interrupt to Software (Frame Timing or acq. exceedance)	SPI Registers - CS
1	MAX2771 I1 - Quantised ADC I1 input stream	Unused	SPI Registers - MOSI
2	MAX2771 Q0 - Quantised ADC Q0 input stream	Unused	SPI Registers - MISO
3	MAX2771 Q1 - Quantised ADC Q1 input stream	Unused	SPI Registers - SCK
4	MAX2771 CLKOUT Pin - configured for ADC sample clock	Unused	Nice to have MAX2771 Passthrough - CS
5	MAX2771 Lock Detect	Unused	Nice to have MAX2771 Passthrough - SDATA
6	Unused	Unused	Nice to have MAX2771 Passthrough - SCLK
7	Unused	Unused	Nice to have MAX2771 SHUTDOWN



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

Pacos first design

by **Paco**

0391 Wokwi Project

github.com/flozgom/tt-pacos-first-design

wokwi.com/projects/465549494272929793

"Just AND gate"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

APA102 to WS2812 Translator

by **Squidgeefish**

448 | 25 MHz | HDL Project

`github.com/squidgeefish/TT09`

"Convert a 7-LED APA102 stream to a WS2812-compatible one"

How it works

This is a converter from the SPI-style APA102 LED protocol to the single-line WS2812 protocol.

It's hard-coded to seven LEDs because I needed to set a limit, and this is clearly the simplest possible way to replace the Arduino Micro performing the same task on my 5n4ck3y-7r clone.

It clocks the SPI data on input bit 0 (clock) and bit 1 (data) and waits until it sees a string of 32 low bits to signal a valid start condition. At this point, it starts saving data into an internal shift register, handing that register's contents over to the WS2812 output data feed once all seven LEDs' values have been received. It continues clocking along until recognizing a stop condition (unconditionally 32 bits after the last LED value), at which point it goes back to waiting for a valid start condition. In order to address area concerns, I wound up cutting this down a bit - the internal mirror register was removed entirely, and the SPI reader now also handles discarding the first 8 bits of each 32-bit pixel value. Further tweaks that traded wiring complexity for combinatorics did not make it any better, unfortunately.

I wrote the SPI-parsing and bit-shuffling code from scratch, but the WS2812 output module is lifted from this TT05 submission. I did modify it to read the data stream MSB-first rather than LSB-first since that made my life a lot easier in bit-twiddler land.

Note that the first byte of each APA102 packet encodes an intensity, which I am ignoring since WS2812s do not support such a feature.

How to test

The way I will be testing this is by attaching `ui_in[0]` to SCK and `ui_in[1]` to SDO on a DEFCON badge that used APA102 LEDs. Attach `uo_out[0]` to drive a string of at least seven WS2812s. I suspect that level shifters will be needed since TinyTapeout ICs run at around 1.8V?

Alternatively, you could probably stream something over in MicroPython.

If you're hand-crafting your packets, a few notes:

- A packet stream must start with a 32-bit start packet (0x00000000)
- APA102s reserve the first byte for intensity: `0b11100000 | <5-bit intensity>`. We're ignoring this completely.
- APA102 color order for the remaining three bytes is Blue, Green, Red.

There is also a random feature added in to fill space - there should be a continuous UART output of "Arglius Barglius" on `uo_out[1]` at approximately 115200 baud; this can be read out with a serial bridge or sufficiently advanced logic analyzer.

External hardware

Some sort of SPI driver is necessary, as is a string of at least seven WS2812 LEDs (or I suppose a logic analyzer can verify it if you're allergic to blinkies).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	APA102_CK	WS2812_OUT	—
1	APA102_SD	UART_OUT	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Rafa’s first Wokwi design

by **Rafael Morales**

0449 Wokwi Project

github.com/RafaMG10/tt-rafas-first-wokwi

wokwi.com/projects/465731430225727489

“tipitaka chip”

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Simple MAC Engine w/ Postproc

by **Luca Goddijn**

450 | 25 MHz | HDL Project

github.com/Arty3/ttgf-mac-engine

"Quantized INT8 MAC engine with bias, activation, and quantizer post-proc, SPI-controlled"

How it works

`tt_um_arty3_mac_engine` is a standalone signed-INT8 multiply–accumulate (MAC) engine with an on-chip post-processing pipeline, controlled over a 4-wire SPI slave. It is intended as a co-processor for quantized neural- network inference on a small MCU: the host streams operands and bias values over SPI, issues compute commands, and reads back an INT8 result.

Block diagram

```
flowchart LR
    SPI["SPI slave<br/>(uio[3:0])"]
    RF["regfile"]
    FSM["cmd_fsm"]
    PE["conv_pe<br/>(MAC)"]
    PP["postproc<br/>bias &rarr; act &rarr; quant"]

    SPI <-->|"rf_addr / rf_wdata<br/>rf_rdata / we / re"| RF
    RF  <-->|"cmd_valid, cmd_code,<br/>op_a, op_b, status"| FSM
    FSM -->|"act_in, wt_in,<br/>acc_in, valid_in"| PE
    PE  -->|"acc_out"| FSM
    PE  -->|"acc_out (live + shadow)"| RF
    FSM -->|"in_data, in_valid,<br/>out_ready"| PP
    PP  -->|"out_data, out_valid<br/>(&rarr; RESULT)"| RF
    RF  -->|"bias, quant_shift,<br/>act_mode"| PP
```

- **`conv_pe`** - a single signed INT8 x INT8 → INT32 MAC with a registered accumulator. One MAC per `CMD_MAC`, no internal storage besides the accumulator.
- **`postproc`** - three-stage pipeline: `bias_add` → `activation` (NONE / ReLU / Leaky ReLU, slope = 1/8) → `quantize` (arithmetic right shift + signed saturation to INT8). Latency: 3 cycles.
- **`cmd_fsm`** - 7-state sequencer (IDLE, MAC, CLR, PP_FEED, PP_WAIT, SOFTRST, DOT4) that owns the handshake to `conv_pe` and `postproc`.
- **`regfile`** - 7-bit-addressed byte-wide register file. Holds operands, bias, quantizer/activation configuration, status flags, and the read ports for the accumulator and the result.

- **`spi_slave`** - mode 0 (CPOL=0, CPHA=0), MSB-first. Every transaction is exactly 16 SCLK cycles framed by CS_N.

Register map (summary)

All registers are 8 bits. The top bit of the SPI header is R/W#; the low 7 bits are the address.

Addr	Name	R/W	Reset	Description
0x00	`STATUS`	RO	0x01	`{0000, ACC_OVF_STK, RESULT_VALID, BUSY, IDLE}`
0x01	`CMD`	WO	-	Writing launches a command (see below)
0x02	`OP_A`	RW	0x00	INT8 multiplicand A
0x03	`OP_B`	RW	0x00	INT8 multiplicand B
0x04	`BIAS`	RW	0x00	INT8 bias added in `postproc`
0x05	`QUANT_SHIFT`	RW	0x00	Arithmetic right shift (0–31) applied in `postproc`
0x06	`ACT_MODE`	RW	0x00	`00`=None, `01`=ReLU, `10`=LeakyReLU(α=1/8)
0x08–0x0B	`ACC_B0..B3`	RO	0x00	Accumulator, little-endian. Reading `ACC_B0` snapshots the upper bytes into a coherent shadow.
0x0C	`RESULT`	RO	0x00	INT8 output of the last `CMD_POSTPROC`. Reading clears `RESULT_VALID`.
0x10	`FEATURE_ID`	RO	0xA1	Engine family + revision
0x12	`OP_A1`	RW	0x00	Lane-1 multiplicand A (`CMD_DOT4`)
0x13	`OP_B1`	RW	0x00	Lane-1 multiplicand B (`CMD_DOT4`)
0x14	`OP_A2`	RW	0x00	Lane-2 multiplicand A (`CMD_DOT4`)
0x15	`OP_B2`	RW	0x00	Lane-2 multiplicand B (`CMD_DOT4`)
0x16	`OP_A3`	RW	0x00	Lane-3 multiplicand A (`CMD_DOT4`)
0x17	`OP_B3`	RW	0x00	Lane-3 multiplicand B (`CMD_DOT4`)

Commands (write to `CMD`):

Code	Mnemonic	Action
0x00	`CMD_NOP`	No-op
0x01	`CMD_MAC`	`acc ← acc + OP_A * OP_B`
0x02	`CMD_CLR_ACC`	`acc ← 0` (config registers untouched)
0x03	`CMD_POSTPROC`	Drive `acc` through bias → activation → quantize; result lands in `RESULT`
0x04	`CMD_DOT4`	4-cycle burst MAC: `acc ← acc + Σ OP_A{i} * OP_B{i}` for i = 0..3 (lane 0 = `OP_A`/`OP_B`, lanes 1..3 = `OP_A1..3`/`OP_B1..3`)
0xFF	`CMD_RESET`	Soft reset: clears `acc`, `RESULT`, sticky flags; preserves `OP_A`/`OP_B`/`BIAS`/`QUANT_SHIFT`/`ACT_MODE`

Pin map

```
ui_in[7:0]   : reserved (tied off internally)
uo_out[0]    : STATUS.IDLE
uo_out[1]    : STATUS.BUSY
uo_out[2]    : STATUS.RESULT_VALID
uo_out[3]    : STATUS.ACC_OVF_STK
uo_out[6:4]  : cmd_fsm state[2:0]
uo_out[7]    : heartbeat (clk / 2^20, ~23.8 Hz at 25 MHz)
uio[0]       : SPI_CS_N (in,  active low)
uio[1]       : SPI_SCLK (in,  mode 0)
uio[2]       : SPI_MOSI (in)
uio[3]       : SPI_MISO (out, driven; held low while CS_N high)
uio[7:4]     : reserved (in)
```

`uio_oe = 8'b0000_1000`. MISO is **not** tri-stated - if you share this chip's SPI bus with other slaves you must add an external buffer.

Electrical / timing summary

- **System clock:** nominal 25 MHz (40 ns), max 50 MHz.
- **SPI clock:** must be ≤ sysclk/4 (≤ 6.25 MHz at 25 MHz sysclk).
- **SPI mode:** 0 (CPOL=0, CPHA=0), MSB-first, 8-bit framed, 2 bytes per transaction.
- **Reset:** active-low asynchronous `rst_n`, synchronized internally with a 2-flop async-assert / sync-deassert synchronizer.

For full specifications - SPI bit-level timing, the regfile's read-side effects, the same-cycle MAC/ACC race definition, the test matrix, and post-silicon bring-up procedure - see `docs/SPEC.md`.

How to test

Hardware

You need a SPI master capable of mode-0, MSB-first transactions at ≤ sysclk/4. An MCU dev board (e.g. Raspberry Pi Pico, STM32 Nucleo, Arduino with a hardware SPI peripheral) is sufficient. A logic analyzer on `uo_out[6:0]` is useful for watching the FSM state and status flags without polling over SPI.

Wire-up:

Chip pin	Host pin
`uio[0]`	SPI CS (slave-select), driven by host
`uio[1]`	SPI SCLK
`uio[2]`	SPI MOSI
`uio[3]`	SPI MISO (input on the host)
`clk`	host-provided system clock (≤ 50 MHz; 25 MHz nominal)
`rst_n`	host-controlled reset, active low

Smoke test

1. Hold `rst_n` low for at least 3 system clock cycles, then release it.
2. Read `STATUS` (addr `0x00`). Expected value: `0x01` (`IDLE = 1`).
3. Read `FEATURE_ID` (addr `0x10`). Expected value: `0xA1`.

If those two reads pass, the SPI slave, regfile, and reset synchronizer are all working.

Functional walkthrough: dot product `[3, −2] · [4, 5]` with ReLU

Each row below is one full 16-SCLK SPI transaction.

```
MOSI:  0x81 0x02     ; CMD_CLR_ACC                (acc = 0)
MOSI:  0x82 0x03     ; OP_A = 3
MOSI:  0x83 0x04     ; OP_B = 4
MOSI:  0x81 0x01     ; CMD_MAC                    (acc = 12)
MOSI:  0x82 0xFE     ; OP_A = −2
MOSI:  0x83 0x05     ; OP_B = 5
MOSI:  0x81 0x01     ; CMD_MAC                    (acc = 12 + (−10)
= 2)
MOSI:  0x84 0x00     ; BIAS        = 0
MOSI:  0x85 0x00     ; QUANT_SHIFT = 0
MOSI:  0x86 0x01     ; ACT_MODE    = ReLU
MOSI:  0x81 0x03     ; CMD_POSTPROC
MOSI:  0x00 0x00     ; poll STATUS until RESULT_VALID (bit 2) = 1
MOSI:  0x0C 0x00     ; read RESULT  -> MISO returns 0x02
```

Between commands you can poll `STATUS` (`bit 0 = IDLE`) to check that the FSM has returned to idle. Most commands complete in a handful of system clocks; the SPI transaction itself is the dominant latency.

Reading the full INT32 accumulator

Always read the four bytes in order `ACC_B0, ACC_B1, ACC_B2, ACC_B3`. Reading `ACC_B0` snapshots the upper three bytes into a coherent shadow; out-of-order reads return stale shadow bytes.

Status flags

- `IDLE` (bit 0): FSM is in IDLE and `postproc` is empty.
- `BUSY` (bit 1): a command is executing.
- `RESULT_VALID` (bit 2): `CMD_POSTPROC` has finished. Cleared by reading `RESULT`.
- `ACC_OVF_STK` (bit 3): a MAC overflowed the INT32 accumulator. Sticky until `CMD_RESET` or hard reset.

Simulation

The repository ships a cocotb-based test suite (Icarus Verilog) under `test/`. It bit-bangs the SPI interface, exercises every command, and includes saturation/overflow coverage. From `test/`:

```
make
```

External hardware

None required. The chip is a self-contained SPI slave; any host MCU with hardware SPI mode 0 is sufficient. A logic analyzer (Saleae, sigrok- compatible, etc.) on `uo_out[6:0]` is useful for bring-up but not mandatory. If sharing the SPI bus with other slaves, add an external buffer on `uio[3]` (MISO) because the pad is permanently driven.

Additional links

Project repository: ttgf-mac-engine Spec documentation: SPEC.md

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	unused	STATUS.IDLE	SPI_CS_N (in)
1	unused	STATUS.BUSY	SPI_SCLK (in)
2	unused	STATUS.RESULT_VALID	SPI_MOSI (in)
3	unused	STATUS.ACC_OVF_STK	SPI_MISO (out)
4	unused	FSM state[0]	reserved (in)
5	unused	FSM state[1]	reserved (in)
6	unused	FSM state[2]	reserved (in)
7	unused	heartbeat (clk/2^20)	reserved (in)

7 segment Display Fli-Flop Try-out

by **Carlos David**

0451 Wokwi Project

github.com/CarlosDavidTorresRico/tt-7-segment-display

wokwi.com/projects/465731458365332481

"An amazing flip-flop example with a seven segment display"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Simple Signal Generator

by **Matt Venn**

0452 | 50 MHz | HDL Project

`github.com/mattvenn/simple-signal-generator`

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

How it works

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

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

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

Phase control

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

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

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

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

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

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

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

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

Frequency formula

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

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

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

Setting `on_count = 0` silences both channels.

The SPI peripheral is reused from calonso88/tt07_alu_74181.

How to test

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

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

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

Register map

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

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

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

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

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

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

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

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

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

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

Encoder hardware

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

Testing

Simulation (cocotb)

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

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

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

Hardware-in-the-loop (HIL)

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

External hardware

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

Project Pinout

Digital Pins

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

DiseñoCursoTiny

by **José Ramñirez Díaz**

0453 10 kHz Wokwi Project

github.com/joserdElectronics/tt-diseocursotiny

wokwi.com/projects/465732744245929985

"biestables"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Tensor Processing Unit For GF

by **Mocus Zhang**

0454 30 MHz HDL Project

`github.com/mocusez/ttgf-tpu`

"multiplies fp8 matrices"

How it works

The project is an AI chip inspired by Google's TPU. It multiply 8-bit floating-point valued matrices. It does so by tiling in 2x2 to fit on the chip's tiny area, so expect performance degradation compared to regular chips. However, the chip's I/O bandwidth will be fully utilized and saturated.

High Level Block Diagram

Architecturally more simple than the previous project. The data moves through the blue, red, yellow, and green arrows.

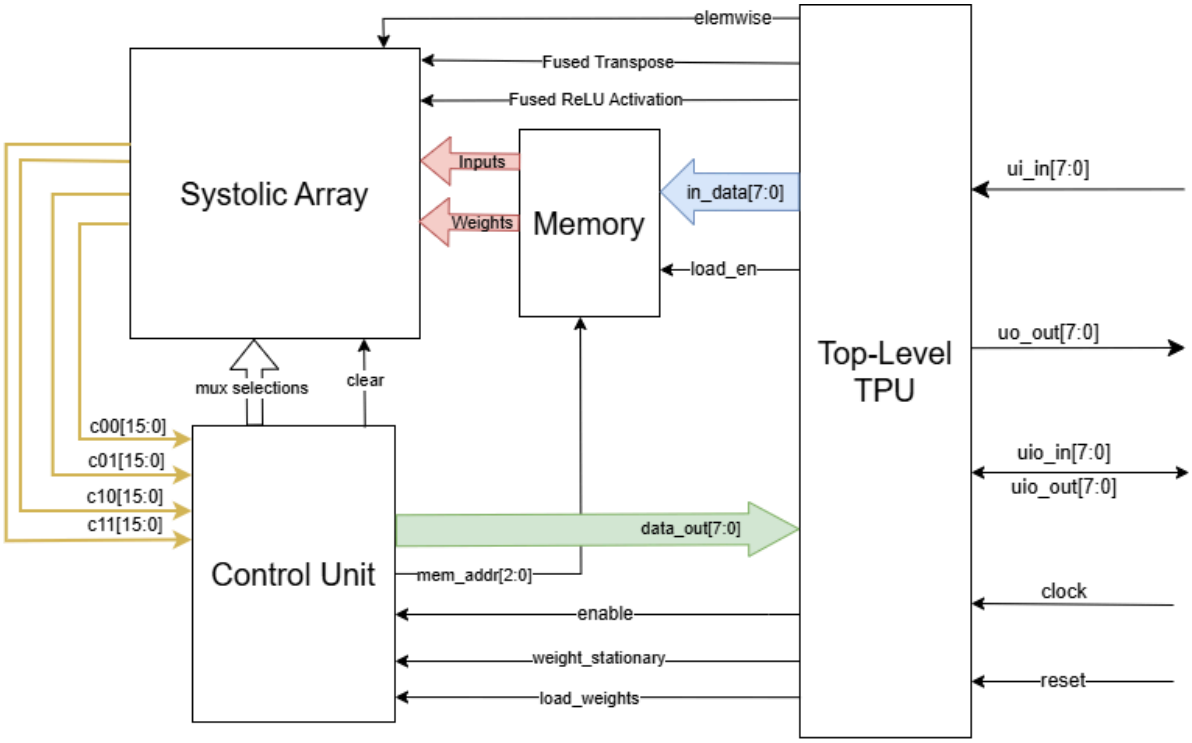


Figure 454.1: alt

How to test

Use cocotb and pyuvm to lean towards IEEE-1800.2.

External hardware

Connect the PCB board with the Tiny Tapeout chips (a.k.a. a Raspberry Pi) to a personal computer via USB.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	IN0	OUT0	LOAD (input)
1	IN1	OUT1	TRANSPOSE (input)
2	IN2	OUT2	ACTIVATION (input)
3	IN3	OUT3	INSTRUCTION (input)
4	IN4	OUT4	ENABLE (input)
5	IN5	OUT5	STATION_WEIGHTS (input, reserved)
6	IN6	OUT6	LOAD_WEIGHTS (input, reserved)
7	IN7	OUT7	Unused

Matt’s first Wokwi design

by **Pedro Da Silva**

0455 Wokwi Project

github.com/pedro-adasilva/tt-matts-first-wokwi

wokwi.com/projects/465731490568160257

“It shows 1 to 4”

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

spiPWMio

by **Sönke Appel**

480 | 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

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

Basic Circuit

by **Izan Amador**

0481 2 Hz Wokwi Project

github.com/izanamador/tt-basic-circuit

wokwi.com/projects/465732847401723905

"It's a basic circuit with 2 NOTs and 1 OR"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

SAP 8 Bit Computer

by **Navneet Prasad**, **Mohammed Riyaj J**, **Akash P** & **Vikash R**

6482 | 10 Hz | HDL Project

github.com/navneetprasad1311/ttgf180-8bitcomputer

> "*The processor implements a 16-instruction, 8-bit ISA inspired by SAP-style architectures, with support for memory access, immediate operations, arithmetic/logic operations, control flow, and basic input/output*"

SAP 8 Bit Computer

This project implements a custom 8-bit computer inspired by the SAP (Simple-As-Possible) architecture, designed and realized entirely at the Register Transfer Level (RTL) using Verilog HDL. The processor is constructed using a collection of independent, well-defined hardware modules that together form a complete and functional CPU.

The design follows a classical computer organization model consisting of a centralized internal data bus, a multi-cycle control unit, and explicit datapath elements such as registers, an arithmetic logic unit (ALU), program counter, memory interface, and input/output registers. Instruction execution is driven by a micro-sequenced control strategy, ensuring deterministic and cycle-accurate operation.

A modified instruction set architecture (ISA) is implemented, extending beyond traditional SAP-1 and SAP-2 designs to include immediate arithmetic operations, logical instructions, conditional branching, and basic input/output support. The system is fully synthesizable and suitable for both functional simulation and FPGA-based implementation.

Features

- Custom 8-Bit CPU Architecture inspired by SAP design principles
- Fully Modular RTL Design with clear separation of control and datapath
- Centralized Internal Data Bus for all register and ALU transactions
- Custom Instruction Set Architecture (ISA) with memory, immediate, control, and I/O instructions
- Multi-Cycle Instruction Execution using T-state based micro-operations
- Arithmetic and Logic Unit (ALU) supporting ADD, SUB, XOR, AND operations
- Flag Register Support with Zero and Carry flag updates

- Conditional and Unconditional Branching (JMP, JC, JZ instructions)
- Input / Output Instruction Support

Instruction Set Architecture

Opcode (bin)	Mnemonic	T-Cycles	Type	Description
0000	NOP	3	Control	Includes Delay in Program Execution
0001	LDA	5	Memory	Load accumulator from memory
0010	ADD	6	Memory	Add memory value to accumulator
0011	SUB	6	Memory	Subtract memory value from accumulator
0100	STA	5	Memory	Store accumulator to memory
0101	LDI	4	Immediate	Load immediate value into accumulator
0110	JMP	4	Control	Unconditional jump
0111	JC	4 (if CF=1 else 3)	Control	Jump if carry flag is set
1000	JZ	4 (if ZF=1 else 3)	Control	Jump if zero flag is set
1001	ADI	5	Immediate	Add immediate value to accumulator
1010	SUI	5	Immediate	Subtract immediate value from accumulator

1011	XRA	6	Memory	Bitwise XOR between accumulator and memory value
1100	ANA	6	Memory	Bitwise AND between accumulator and memory value
1101	INP	7	I/O	Load external input into accumulator
1110	OUT	4	I/O	Output accumulator to output register
1111	HLT	4	Control	Halt CPU execution

Note: Every instruction runs all 6 T-cycles except **INP** (7 T-cycles) . The table mentions the operation T-cycle

Verification

The complete CPU design was developed and verified at the RTL level using Verilog HDL. Functional verification was performed through simulation and the design was subsequently implemented and tested on an FPGA platform to validate instruction execution, control sequencing, arithmetic and logic operations, branching behavior, memory access, and input/output functionality prior to ASIC submission.

GitHub Repository: 8-Bit Computer on FPGA

How it works

The processor implements a 16-instruction, 8-bit ISA inspired by SAP-style architectures, with support for memory access, immediate operations, arithmetic/logic operations, control flow, and basic input/output.

Operating modes

Program mode (ui_in[0] = 1): The 8 uio pins are driven by the design and display the CPU output byte. Use this mode while running a program.

Load mode (ui_in[0] = 0): The uio pins become inputs. Use them to enter bytes into RAM or provide input data requested by the CPU.

Pin usage

Pin(s)	Function
ui_in[7:4]	RAM address for manual loading.
ui_in[3]	load_ram - write the byte on uio_in into the selected RAM address.
ui_in[2]	inp_loaded – acknowledge that external input data is ready.
ui_in[1]	start – start or continue CPU execution.
ui_in[0]	prog_mode – selects load mode or program mode.
uio_in[7:0]	Data byte supplied by the user (RAM data or CPU input).
uio_out[7:0]	Output/display byte from the CPU when in program mode.
uo_out[7]	CPU input request flag (inp_req).
uo_out[6]	CPU halted flag.
uo_out[5:2]	Current program counter value.

Registers and Flags

The CPU contains the following registers:

- **Program Counter (PC)** – Holds the address of the next instruction.
- **Instruction Register (IR)** – Stores the currently executing instruction.
- **Accumulator (A)** – Primary working register used for arithmetic and logic operations.
- **B Register** – Temporary operand register for ALU operations.
- **Carry Flag** – Set when an arithmetic operation generates a carry/borrow.
- **Zero Flag** – Set when an ALU operation produces a result of zero.

Input Handling

The `INP` instruction uses a simple hardware handshake:

1. The CPU asserts `inp_req`.
2. External logic places a byte on the input bus.
3. The user asserts `inp_loaded`.
4. The CPU loads the value into the accumulator.
5. The value is then written to the RAM address specified by the instruction.

Execution pauses while waiting for input, enabling interactive programs.

Execution Status Outputs

Several status signals are exposed for debugging and monitoring:

Signal	Description

`uo_out[5:2]`	Current Program Counter (`PC[3:0]`)
`uo_out[6]`	CPU HALT status
`uo_out[7]`	Input request (`inp_req`)
`uio_out[7:0]`	CPU output value (Program Mode)

How to Test

Important Notes

Before programming or running the CPU:

- Apply an **active-low reset** by driving `rst_n = 0`.
- Release reset by setting `rst_n = 1`.
- Always reset the design before loading a new program.
- Use a **10 Hz clock** so execution can be observed easily.
- Ensure the CPU is not running while loading RAM (`prog_mode = 0`).

1. Program the RAM

Load Mode is selected with:

```
prog_mode = 0
```

For each instruction:

1. Select the target RAM address using `ui_in[7:4]`.
2. Place the instruction byte on `uio_in[7:0]`.
3. Pulse `load_ram` (`ui_in[3]`) high for one clock cycle.
4. Repeat until the entire program has been loaded.

Example Program: Add Two User Inputs

This program:

1. Reads the first input value and stores it in RAM address `E`.
2. Reads the second input value and stores it in RAM address `F`.
3. Loads the first value.
4. Adds the second value.
5. Outputs the result.
6. Halts.

Address	Instruction	Binary
0	INP E	`1101_1110`
1	INP F	`1101_1111`
2	LDA E	`0001_1110`
3	ADD F	`0010_1111`
4	OUT	`1110_0000`

5	HLT	1111_0000

Load the program into RAM addresses `0` through `5`.

Running the Program

1. Load the program into RAM.
2. Set `prog_mode = 1`.
3. Pulse `start`.

The CPU will immediately request the first input by asserting `inp_req` (`uo_out[7]`).

Example Test: 5 + 3

First Input

When `inp_req` goes high:

- Place `0x05` on `uio_in[7:0]`.
- Pulse `inp_loaded`.

The CPU stores `5` in RAM address `E`.

Second Input

When `inp_req` goes high again:

- Place `0x03` on `uio_in[7:0]`.
- Pulse `inp_loaded`.

The CPU stores `3` in RAM address `F`.

Expected Result

The CPU executes:

```
A ← RAM[E] = 5
A ← A + RAM[F] = 5 + 3 = 8
OUT A
```

Expected outputs:

Signal	Value
`uio_out[7:0]`	`00001000` (`0x08`)
`uo_out[6]`	1 (HALT asserted)

The output value 8 should remain visible on uio_out[7:0] after the program halts.

External Hardware

No external hardware is required.

For easier testing:

- DIP switches or push buttons for entering program data and control signals
- LEDs connected to `uo_out` and `uio_out`
- A logic analyzer for debugging and observing execution
- An 8-bit binary-to-7-segment decoder/display for viewing CPU outputs as decimal values

Contact

- Navneet Prasad, III year, ECE, Bannari Amman Institute of Technology
- Mohammed Riyaj J, III year, ECE, Bannari Amman Institute of Technology
- Akash P, III year, ECE, Bannari Amman Institute of Technology
- Vikash R, III year, ECE, Bannari Amman Institute of Technology

Acknowledgements

This project was inspired by the educational work of Albert Malvino and Ben Eater. The overall architecture and learning approach draw from the SAP (Simple-As-Possible) computer concepts presented in *Digital Computer Electronics* by Albert Malvino, as well as the practical computer-building demonstrations and educational content created by Ben Eater. Their work has been an invaluable resource in understanding computer architecture and digital system design.

We would also like to acknowledge the Centre for SoC & FPGA Design Laboratory at Bannari Amman Institute of Technology, Erode, Tamil Nadu, India, for providing a supportive environment for learning and experimentation in digital design and VLSI development. Special thanks to Dr. Elango S, Associate Professor, ECE, Bannari Amman Institute of Technology for his guidance, encouragement, and support throughout the development of this project.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PGRM	—	D0
1	START	—	D1
2	INP_LD	PC_0	D2
3	RAM_LD	PC_1	D3
4	ADDR_0	PC_2	D4
5	ADDR_1	PC_3	D5
6	ADDR_2	HLT	D6
7	ADDR_3	INP_SIG	D7

El primer diseño de Matt para Wokwi

by **Marcos**

483 Wokwi Project

github.com/marcosgafer/tt-el-primer-diseo

wokwi.com/projects/465731452481768449

"Mi primer proyecto"

How it works

Explain how your project works explicado

How to test

Explain how to use your project explicado

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any explciado

Project Pinout

Digital Pins

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

BIST-8: Built-In Self-Test for 8-bit CLA Adder

by **Mario Garcia Jimenez**

484 | 10 MHz | HDL Project

github.com/mgarciajimenezAirzone/tt-mmggjj

"BIST architecture for an 8-bit Carry-Lookahead Adder. Includes 12-bit LFSR pattern generator, 16-bit MISR response compactor, fault injection and 4-state FSM controller. Golden signature 0xCF77."

How it works

BIST-8 implements a complete Built-In Self-Test architecture for an 8-bit Carry-Lookahead Adder (CLA).

The design has two modes of operation:

Normal mode (bist_en=0): The CLA adder operates as a standard 8-bit adder. Operand A is provided via ui_in[7:2] and uio_in[7:6], operand B via uio_in[5:0]. The result appears on uo_out[7:3].

BIST mode (bist_en=1): The circuit self-tests in 4095 clock cycles automatically:

1. A 12-bit LFSR (polynomial $x^{12+x}11+x^{10+x}4+1$, seed 0xACE) generates 4095 unique test vector pairs.
2. Each pair is applied to the CLA adder (Circuit Under Test).
3. Each response is compacted into a 16-bit MISR (polynomial x^16+x15+x^2+1).
4. After 4095 cycles, the MISR signature is compared against the golden reference 0xCF77.
5. If they match, bist_pass=1. Otherwise bist_fail=1.

A fault injection input (fault_inject) forces bit 3 of the adder output to stuck-at-0, allowing demonstration of fault detection. The faulty signature is 0x927F, which differs from the golden value, proving the BIST detects the fault.

The FSM has 4 states: IDLE -> BIST_RUN -> COMPARE -> DONE.

How to test

Normal mode:

1. Set ui_in[0]=0 (bist_en=0)
2. Apply operand A on ui_in[7:2] and uio_in[7:6]
3. Apply operand B on uio_in[5:0]

4. Read result on uo_out[7:3]

BIST mode (no fault):

1. Assert reset (rst_n=0 then rst_n=1)
2. Set ui_in[0]=1, ui_in[1]=0
3. Wait 4095 clock cycles for uo_out[2]=1 (bist_done)
4. Expected: uo_out[0]=1 (bist_pass), uo_out[1]=0 (bist_fail)

BIST mode with fault injection:

1. Assert reset
2. Set ui_in[0]=1, ui_in[1]=1
3. Wait for uo_out[2]=1 (bist_done)
4. Expected: uo_out[1]=1 (bist_fail) - fault detected

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	bist_en	bist_pass	cycle_count[0]
1	fault_inject	bist_fail	cycle_count[1]
2	a_in[0]	bist_done	cycle_count[2]
3	a_in[1]	sum_out[3]	cycle_count[3]
4	a_in[2]	sum_out[4]	cycle_count[4]
5	a_in[3]	sum_out[5]	cycle_count[5]
6	a_in[4]	sum_out[6]	a_in[6]
7	a_in[5]	sum_out[7]	a_in[7]

Tiny Tapeout Template flip flop

by **Nerea**

0485 10 kHz Wokwi Project

github.com/nsuarezparrilla3/tt-template-flip-flop

wokwi.com/projects/465731502018614273

"*Circuito basico con puertas logicas*"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

BioPulse Tile

by **Roberto Medina**

`486` `1 kHz` `HDL Project`

`github.com/RobMBSos/tt_um_roberto_tiny_radar_tile`

> *"A tiny vital-sign radar event detector: tracks an 8-bit signal baseline, detects breathing cycles, and flags normal/fast/slow/irregular breathing and apnea."*

How it works

BioPulse Tile turns a single 8-bit radar/biosignal sample stream into vital-sign events and metrics, using only digital logic — no CPU, no software. The datapath processes one sample per clock.

Processing chain:

1. **Input select** — the 8-bit `ui_in` sample, or, in demo mode (`uio[0]=1`), an internal generator producing a synthetic breathing triangle plus a small heartbeat ripple.
2. **Baseline tracker** — a slow exponential moving average (alpha = 1/256) estimates the DC level; a faster EMA (alpha = 1/2) high-passes the signal to isolate the heartbeat band.
3. **Breathing detector** — a hysteresis comparator against `baseline ± thr` marks one breath per low→high crossing. `uio[1]` selects sensitivity.
4. **Classification FSM** — labels the breathing period normal / fast / slow / irregular, or raises apnea after 255 clocks with no breath.
5. **Heartbeat detector** — a second hysteresis path on the high-passed signal detects the faster heartbeat component.
6. **Rate estimators** — two sequential long-division units compute breaths-per-minute and heart-rate as `BPM = 1000 / period`.
7. **Signal-quality** — peak-to-peak amplitude over each frame gives a quality flag.
8. **UART transmitter** — every frame it streams a 5-byte packet (8N1, LSB first) out of `uio[5]`: `0xAA, breaths-per-min, heart-rate, peak-to-peak, flags`.

Reading the outputs

`uo_out` is dual-purpose, selected by `uio[2]` (BPM readout select):

- `uio[2]=0` → `uo_out` shows the status flags:

Bit	Meaning
0	breathing detected

1	apnea warning
2	fast breathing
3	slow breathing
4	irregular signal / motion
5	heartbeat detected
6	signal quality good
7	valid / status

- `uio[2]=1` → `uo_out` shows the 8-bit breaths-per-minute value.

`uio[5]` is the UART transmit line; `uio[7:6]` show the top two bits of breaths-per-minute as a coarse indicator.

Controls (`uio[4:0]`, inputs)
`uio[0]` demo mode, `uio[1]` sensitivity, `uio[2]` BPM readout select, `uio[4:3]` demo pattern (`00`=normal, `01`=fast, `10`=slow, `11`=apnea).

A note on timing
Periods and thresholds are in clocks. Run the chip at a low clock for realistic vital-sign timescales, or feed external samples at your sample rate.

How to test

Run the cocotb testbench:

```
cd test
make
```

It checks reset, the four demo patterns (normal / fast / slow / apnea), the heartbeat detector, the breaths-per-minute readout, UART activity, and external-input mode.

On hardware: set `uio[0]=1`, pick a pattern on `uio[4:3]`, and watch `uo_out` on LEDs (flags). Set `uio[2]=1` to read breaths-per-minute on the same LEDs, and capture `uio[5]` with a 3.3 V UART receiver to read all the streamed metrics.

External hardware

None required for the demo. For real measurements, connect the digitised output of a continuous-wave radar or biosignal analog front-end to `ui_in[7:0]`, LEDs to `uo_out`, and optionally a UART receiver to `uio[5]`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sample[0] - 8-bit radar/biosignal sample (external mode)	breathing detected / BPM[0]	demo mode enable (in)
1	sample[1]	apnea warning / BPM[1]	sensitivity select (in)
2	sample[2]	fast breathing / BPM[2]	BPM readout select (in)
3	sample[3]	slow breathing / BPM[3]	demo pattern select [0] (in)
4	sample[4]	irregular signal / BPM[4]	demo pattern select [1] (in)
5	sample[5]	heartbeat detected / BPM[5]	UART TX - streams metrics (out)
6	sample[6]	signal quality good / BPM[6]	coarse breaths-per-min [6] (out)
7	sample[7]	valid / status / BPM[7]	coarse breaths-per-min [7] (out)

Tiny Tapeout RJAP

by **Rodrigo Álvarez**

0487 1 Hz Wokwi Project

github.com/rodalvar23/tt-rjap

wokwi.com/projects/465732616714924033

"*Wokwi Tiny Tapeout Beginner Project*"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Tiny Tapeout JMCG

by **Jose Manuel Cano Garcia**

512 | 10 MHz | Wokwi Project

github.com/jmcanogarcia/tt-jmcg

wokwi.com/projects/465731455677830145

"*LSFR*"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

rgb_mixer

by **divadnaujGB**

513 | 50 MHz | HDL Project

`github.com/divadnauj-GB/tt_um_divadnaujGB_rgb_muxer_pb`

"This is my own version of the Z2A course of the rgb_mmixer"

How it works

This is a custom implementation of an rgb mixer. The designs uses four inputs, two push buttons (i.e., `inc` and `dec`) and a 2 position dip switch (i.e., `led[1:0]`), and three outputs named `PWM0`, `PWM1`, and `PWM2`. The `inc` and `dec` buttons are used to increase or decrease the Pulse Widht of a selected output specified by the `led[1:0]` switch.

The following indicates the output selection based on the `led[1:0]` input.

led[1:0]	Output
00	PWM0
01	PWM1
10	PWM2
11	-

How to test

Connect the push buttons and the dip-switches in a pull-up configurations to the following ports.

tt_inputs	connection
ui[0]	inc
ui[1]	dec
ui[2]	led[0]
ui[3]	led[1]
—	—

Connect an RGB LED to the following outputs.

tt_inputs	connection
uo[0]	PWM0
uo[1]	PWM1
uo[2]	PWM2

—	—

External hardware

It is only required two push buttons, one dip-switch and an RGB LED and some resistors.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	inc	PWM0	—
1	dec	PWM1	—
2	led[0]	PWM2	—
3	led[1]	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Paula's first Wokwi design

by **Paula García Jiménez**

0514 1 Hz Wokwi Project

github.com/paulagarcianm-cmd/tt-paulas-first-wokwi

wokwi.com/projects/465731858252480513

“Demo”

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

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

microlane demo project

by **htfab**

`0515` `5 Hz` `HDL Project`

`github.com/htfab/ttgf-microlane-demo`

"Scrolls a message on the 7-segment display. Hardened using microlane."

How it works

Scrolls the text "hardened using python" over the 7-segment display. This is a demo project for the microlane flow.

How to test

Select the project and provide a slow clock (say, 5 Hz).

External hardware

None

Project Pinout

Digital Pins

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

Pedro Template

by **Pedro Barbero**

516 Wokwi Project

github.com/PitBarber/tt-pedro-template

wokwi.com/projects/465732880722332673

"Plantilla curso tinytipeout"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Ball Display

by **Daniel Penas**

`517` `25 MHz` `HDL Project`

`github.com/danielpenas42/ttgf-verilog-danielpenas42`

"shows in vga display and bounces and you can send commands to move it through spi"

How it works

This project draws a small ball on a 640x480 VGA-style display. The top-level clock drives the VGA timing generator, SPI receiver, command decoder, and physics engine.

Commands are sent over SPI using one byte per command. The upper two bits select the command and the lower six bits hold the data:

- `00`: set X position
- `01`: set Y position
- `10`: set X velocity
- `11`: set Y velocity

Position values are shifted left by three before being stored. Velocity values are interpreted as signed 6-bit numbers and clamped to the range `-10` to `10`. The ball position updates every six video frames and rebounds by inverting the X or Y velocity when it reaches the screen edges.

How to test

Run the cocotb testbench from the `test` directory:

```
cd test
make clean
make
```

The tests send SPI commands to set position and velocity, then check movement, velocity clamping, reset behavior, and edge rebound behavior. The gate-level test uses a smaller top-level pin smoke test because internal RTL signal names are not preserved after synthesis.

External hardware

The design is intended for a TinyVGA-style PMOD connection on the output pins and SPI command input on the bidirectional pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	VGA red[0]	SPI chip select
1	—	VGA green[1]	SPI MOSI
2	—	VGA blue[1]	—
3	—	VGA vsync	SPI clock
4	—	VGA red[1]	—
5	—	VGA green[0]	—
6	—	VGA blue[0]	—
7	—	VGA hsync	—

ocxpkeWokwiDesign

by **Jose Ramírez**

518 Wokwi Project

github.com/ocxpke/tt-ocxpkewokwidesign

wokwi.com/projects/465731575275296769

"The first (of many) chip designs"

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. Made by ocxpke

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	pin0	sevSeg0	—
1	pin1	sevSeg1	—
2	pin2	sevSeg2	—
3	pin3	sevSeg3	—
4	pin4	sevSeg4	—
5	pin5	sevSeg5	—
6	pin6	sevSeg6	—
7	pin7	sevSeg7	—

Number Memory Game

by **Daniel Gabai** & **Shehroze Kiani**

0519 | 50 MHz | HDL Project

github.com/DanielGabai/ttgf26a-memory-game

"Number based memory game"

How it works

Number memory game

How to test

Test via pcb board input switches and LED output

External hardware

N/A

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	Submit Answer	—	—
7	Start Game	—	—

BF

by **Joseph Wan** & **Joseph Mensah**

0545 HDL Project

`github.com/Josephwann/jj-tinytapeout`

"Runs programs in the BF language"

Block diagram

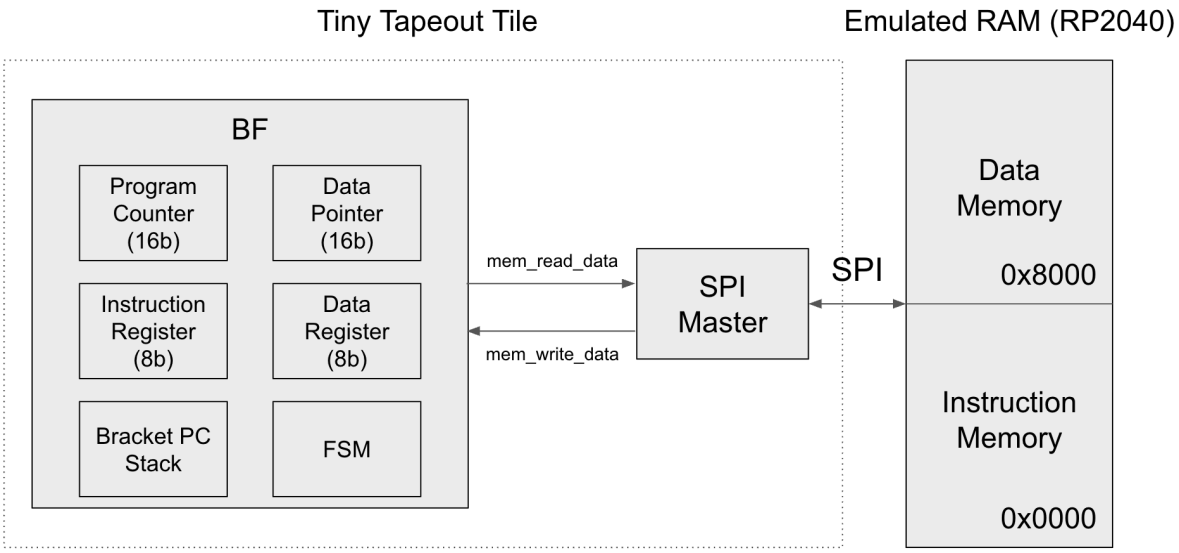


Figure 545.1: BF Block Diagram

How it works

This is a multicycle CPU that runs programs written in the BF (BF) language. BF has eight instructions operating on a tape of byte cells and a data pointer: `>` `<` move the pointer, `+` `-` change the cell, `.` `,` do output/input, and `[` `]` form loops.

The design has no on-chip RAM. **Both the program and the data tape live off-chip in the RP2040's emulated SPI RAM** (Michael Bell's `spi-ram-emu`, which behaves like a 23LC512: SPI mode 0, `0x03` read / `0x02` write, 16-bit address). The chip reaches memory through an on-chip **SPI master**, so every instruction performs at least one SPI transaction (arithmetic ops like `+` and `-` are a read-modify-write = two transactions).

Memory map (16-bit address): `0x0000`–`0x7FFF` is instruction memory, `0x8000`–`0xFFFF` is the data tape.

Blocks:

- **FSM (`bf`)** — fetch/decode/execute control, plus a hardware bracket stack for `[ ]` loops.
- **`spi_ram`** — adapter that turns each memory request into one SPI transaction.
- **`spi_master`** — generates SCLK/CS, shifts the 32-bit command/address/data frame.

How to test

1. Connect the SPI pins (`uio[0..3]`) to an RP2040 running `spi-ram-emu`, which provides the 64 KB memory.
2. Load a BF program into instruction memory starting at address `0x0000`.
3. Hold `rst_n` low to reset, release it, then pulse **`start`** (`uio[4]`) high for at least one cycle.
4. The core runs the program, fetching instructions and operating on data entirely over SPI.

The repo's cocotb tests verify the design at multiple levels: the SPI master (`Makefile.spi`), the isolated BF FSM (`Makefile.bf`), and the full chip with a mock SPI RAM (`Makefile.top`).

External hardware

An RP2040 (e.g. the Tiny Tapeout demo board) running `spi-ram-emu` to provide 64 KB of SPI RAM connected to `uio[0..3]`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0]	out[0]	SPI CS
1	in[1]	out[1]	SPI MOSI
2	in[2]	out[2]	SPI MISO
3	in[3]	out[3]	SPI SCK
4	in[4]	out[4]	start
5	in[5]	out[5]	out_valid
6	in[6]	out[6]	in_valid
7	in[7]	out[7]	in_ack

Memristive Crossbar Peripheral Controller (GF180)

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

`@547` `50 MHz` `HDL Project`

`github.com/SanthoshSivasubramani/tt_um_santhosh_xbar_ctrl_gf_pub`

> *"GF180mcuD port of the 8x8 memristive crossbar controller: READ/SET/RESET/FORMING/SWEEP modes, pulse trains, 16-bit pulse counter, compliance auto-abort with sticky history, V/2 half-select DAC, last-ADC-in-sweep snapshot. Direct cross-foundry sibling to the TTSKY26a xbar_ctrl for comparative characterisation."*

How it works

Device-agnostic peripheral controller for 8-row x 8-column memristive / spintronic / phase-change / ferroelectric crossbar arrays. Drives four operation modes — READ, SET, RESET, FORMING — with a configurable 1-to-511-cycle pulse width, repeatable pulse trains with programmable inter-pulse gap, a swept-DAC I-V characterization mode, a compliance-current auto-abort, a **16-bit** hardware pulse counter, a read-only V/2 half-select DAC value for sneak-path mitigation, a **sticky compliance-history register**, and a **last-ADC-in-sweep** snapshot. All parameters are programmed through a Mode-0 SPI slave.

The FSM is verbatim from the TTSKY26a sibling (IDLE -> SETUP -> PULSE -> SENSE -> GAP -> REPORT with an additional SWEEP outer loop). The design instantiates no foundry-specific standard cells, so the GF180mcuD port required no cell-library substitution; instead the extra gate budget that the GF 1x1 tile provides versus SKY (1.6x) is spent on features that were deferred on SKY:

Feature	SKY port	GF port
Pulse counter width	8-bit (wraps at 256)	**16-bit** (0x0E lo / 0x11 hi)
Compliance history	Only current trip in status	**Sticky 8-bit** (0x10), clear via `reg_ctrl[4]`
Last-sweep-ADC snapshot	not captured	**Register 0x12**
Register count (NUM_REGS)	16	**20**

Register map

Addr	Name	Access	Description
0x00	`reg_ctrl`	RW	`[0]=start`, `[1]=abort`, `[2]=auto_sweep`, `[3]=compliance_en`, `[4]=clear_hist` (self-clearing)
0x01	`reg_mode`	RW	`[1:0]` in {READ=00, SET=01, RESET=10, FORM=11}
0x02	`reg_row`	RW	3-bit row address
0x03	`reg_col`	RW	3-bit column address
0x04	`reg_pulse_l`	RW	Pulse width `[7:0]`
0x05	`reg_pulse_h`	RW	Pulse width `[8]` (9-bit total, default 10)
0x06	`reg_dac`	RW	8-bit DAC code (default 0x80)
0x07	`reg_status`	R	`[7]=compliance_hit`, `[3]=sense_in`, `[2]=op_error`, `[1]=op_done`, `[0]=busy`
0x08	`reg_adc_data`	R	Last ADC reading, `{4'b0, ui_in[7:4]}`
0x09	`reg_sweep_start`	RW	Sweep start DAC (default 0x00)
0x0A	`reg_sweep_end`	RW	Sweep end DAC (default 0xFF)
0x0B	`reg_sweep_step`	RW	Sweep increment (default 0x10)
0x0C	`reg_repeat`	RW	`[3:0]=repeat_count` (0 -> single), `[7:4]=gap x 16 cycles`
0x0D	`reg_compliance`	RW	Compliance ADC threshold (default 0xFF = off)
0x0E	`pulse_count[7:0]`	R	Pulse counter low byte
0x0F	`V/2`	R	`{1'b0, active_dac[7:1]}` half-select bias
0x10	`reg_compliance_hist`	R	**GF-new** sticky compliance history (cleared by `reg_ctrl[4]`)
0x11	`pulse_count[15:8]`	R	**GF-new** pulse counter high byte
0x12	`reg_sweep_last_adc`	R	**GF-new** ADC sample from last sweep step

SPI: Mode 0 (CPOL=0, CPHA=0), MSB first. CS=`uio[0]`, MOSI=`uio[1]`, MISO=`uio[2]` (tri-stated when CS high), SCK=`uio[3]`. First byte is `{rw, addr[6:0]}`; second byte is write data, or 8 clocks of MISO for reads.

Pinout

- `ui[0]` sense_in, `ui[1]` adc_ready, `ui[2]` ext_irq, `ui[3]` unused
- `ui[7:4]` 4-bit ADC data (captured into `reg_adc_data[3:0]`)
- `uo[0]` pulse_out, `uo[1]` row_enable, `uo[2]` col_enable, `uo[3]` op_done
- `uo[7:4]` = `active_dac[3:0]` (low nibble of DAC code)
- `uio[0..3]` SPI CS/MOSI/MISO/SCK
- `uio[4..6]` = `row_addr[2:0]`
- `uio[7]` = `col_addr[0]`

Full 3-bit column address is SPI-readable at `reg_col[2:0]`.

How to test

1. Apply reset. Configure over SPI. For a single SET pulse on row 2 / col 5 with DAC 0xA0: write `reg_mode=0x01`, `reg_row=0x02`, `reg_col=0x05`, `reg_pulse_l=0x32` (50 cycles), `reg_dac=0xA0`, then pulse `reg_ctrl[0]=1`.
2. Observe `pulse_out` high for 50 clock cycles; `row_enable`/`col_enable` stay high; `op_done` asserts on completion. Read `reg_adc_data` (0x08).
3. **READ:** `reg_mode=0x00`, start — skips PULSE, waits for `adc_ready`.
4. **Pulse train:** `reg_repeat=0x34` -> 4 pulses with 3x16=48-cycle gap. Counter (0x11:0x0E) reports actual pulses delivered.
5. **Compliance limit:** `reg_compliance=0x40`, `reg_ctrl[3]=1`. If ADC crosses the threshold during SENSE, the FSM aborts; compliance_hit latches in status AND in `reg_compliance_hist` (0x10, GF-new).
6. **Clear history:** write `reg_ctrl=0x10` to clear `reg_compliance_hist`; the bit self-clears next cycle.
7. **Sweep mode:** set `reg_ctrl[2]=1` + `reg_ctrl[0]=1`. After completion, the ADC sample at the last DAC step is retained in `reg_sweep_last_adc` (0x12, GF-new), separate from `reg_adc_data` (0x08).

A cocotb suite of 36 tests in `test/test.py` covers every mode, SPI round-trip, pulse-width boundaries (1 and 511 cycles), pulse trains, compliance abort, sweep (incl. step=0 edge case), V/2 readback, `uio_oe` direction, and all three GF-new features. All tests pass locally on icarus.

External hardware

- **SPI master** (MCU/FPGA) on `uio[0:3]`.
- **External DAC**: connect `uo_out[7:4]` to a 4-bit DAC for basic use, or read the full 8-bit `reg_dac` over SPI for higher resolution. Register 0x0F gives the V/2 half-select bias line.
- **External ADC**: 4-bit nibble on `ui_in[7:4]`, ready pulse on `ui_in[1]`.
- **Crossbar array**: row select on `uio_out[6:4]` (3 bits), column-low on `uio_out[7]`; decode full 3-bit column via SPI.

- Optional: external interrupt on `ui_in[2]`, single-bit sense-comparator on `ui_in[0]`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sense_in	pulse_out	spi_cs_n
1	adc_ready	row_enable	spi_mosi
2	ext_irq	col_enable	spi_miso
3	—	op_done	spi_sck
4	adc_data[0]	dac_code[0]	row_addr[0]
5	adc_data[1]	dac_code[1]	row_addr[1]
6	adc_data[2]	dac_code[2]	row_addr[2]
7	adc_data[3]	dac_code[3]	col_addr[0]

Ring Oscillator PVT Sensor & TRNG (GF180)

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

549 | 50 MHz | HDL Project

github.com/SanthoshSivasubramani/tt_um_santhosh_ring_osc_gf_pub

> *"GF180mcuD port of 5 SPI-configurable ring oscillators (7/11/15/21/31 gf180 stages) with 16-bit frequency counter, auto gate timer, configurable prescaler, XOR jitter TRNG, per-RO health history, differential RO beat-frequency mode, and frequency-bounds health monitor for cross-foundry PVT characterization"*

How it works

This design is the **GF180mcuD port** of our SKY130 ring-oscillator PVT sensor (`tt_um_santhosh_ring_osc`). It implements **five gatable ring oscillators** (7, 11, 15, 21, and 31 `gf180mcu_fd_sc_mcu7t5v0__inv_1` inverter-chain stages, each with a `gf180mcu_fd_sc_mcu7t5v0__nand2_1` enable gate) together with a 16-bit frequency counter, a three-stage CDC synchronizer, an auto gate timer, a configurable prescaler, an XOR-jitter true random number generator (TRNG), a frequency-bounds health monitor, and a differential (beat-frequency) measurement mode. It is intended for **cross-foundry PVT characterization**: pairing this tile with the SKY130 sibling on the same chip family lets us compare ring-oscillator frequency, TRNG statistics, and aging drift between the two nodes from identical RTL.

All oscillator stages are tagged with `(* keep, dont_touch *)` so that the synthesis and placement flows preserve them as physical inverter/NAND chains rather than optimizing them into constant nets.

The block supports two control modes, selected by `ui_in[6]` (`spi_mode`):

- **Parallel mode (`spi_mode=0`)** — `ro_sel[2:0]` (`ui_in[2:0]`) chooses one of the five ROs, `cnt_enable` (`ui_in[3]`) starts/stops counting, a rising edge on `cnt_clear_latch` (`ui_in[4]`) latches the 16-bit count and clears the accumulator, and `byte_sel` (`ui_in[5]`) multiplexes `uo_out` between the low (0) and high (1) bytes of the latched count. `uio_out[4]` is the overflow flag, `uio_out[5]` exposes the raw RO output, `uio_out[6]` the synchronized RO output, and `uio_out[7]` is `meas_done` from the auto gate timer.
- **SPI mode (`spi_mode=1`)** — a 16-bit SPI slave (Mode 0, MSB first; CS on `uio_in[0]`, MOSI on `uio_in[1]`, MISO on `uio_out[2]`, SCK on `uio_in[3]`) exposes a register file:

Addr	Name	Access	Description
0x00	`reg_ctrl`	R/W	[0]=`auto_gate_start` (self-clear); [1]=`clear_meas_done`; **[2]=`clear_health_history` (self-clear, new on GF)**
0x01	`reg_ro_sel`	R/W	[2:0] RO select
0x02	`reg_ro_en`	R/W	[4:0] per-RO enable — reset default `0x1F`
0x03	`reg_gate_l`	R/W	gate-time low byte
0x04	`reg_gate_h`	R/W	gate-time high byte
0x05	`reg_prescale`	R/W	[1:0] prescaler (÷8/÷16/÷32/÷64; reset default ÷16)
0x06	`reg_status`	R	[0]=`gate_active`, [1]=`overflow`, [2]=`meas_done`
0x07	`reg_count_l`	R	latched count [7:0]
0x08	`reg_count_h`	R	latched count [15:8]
0x09	`reg_trng_ctrl`	R/W	[0]=`trng_en`, [1]=`health_en`, [2]=`diff_mode`
0x0A	`reg_diff_sel`	R/W	[2:0] RO_B selector for beat/XOR modes
0x0B	`reg_health_lo`	R/W	expected frequency lower bound
0x0C	`reg_health_hi`	R/W	expected frequency upper bound
0x0D	`reg_trng_data`	R	8-bit XOR-jitter byte
0x0E	`reg_health_status`	R	[0]=below, [1]=above, [2]=stuck, [3]=ok, [4]=trng_valid
0x0F	**`reg_hh_stuck`**	**R**	**per-RO stuck history (sticky; new on GF)**
0x10	**`reg_hh_below`**	**R**	**per-RO below-lo history (sticky; new on GF)**

The raw RO is synchronized into the system clock domain through a 3-stage flip-flop synchronizer (with an optional ÷8/÷16/÷32/÷64 prescaler ahead of it) before being rising-edge detected and counted. When `diff_mode` is set, two user-selected ROs are XORed prior to counting, producing the beat frequency |f_A − f_B|, which is very sensitive to local PVT mismatch. The TRNG collects XOR jitter bits into an 8-bit register readable at `reg_trng_data`; the health monitor flags the selected RO against `reg_health_lo`/`reg_health_hi` bounds into `reg_health_status`.

New on GF: per-RO health history

GF180 1×1 tile area (55 712 μm²) holds more gates than SKY130 1×1 (17 954 μm²) after process-scaling, so on this port we re-enable a feature that was deferred on the SKY submission: **two sticky 5-bit history registers** (`reg_hh_stuck`, `reg_hh_below`) that latch a bit per RO whenever that RO is the currently-selected one and is flagged stuck or below-lo by the health monitor. The registers survive until explicitly cleared by writing `reg_ctrl[2]=1` (self-clearing pulse). This makes aging and fault detection possible without continuously polling from the host.

How to test

1. Apply reset (`rst_n` low for ≥10 clock cycles, then release).
2. **Simple parallel measurement:** set `spi_mode=0`, write `ro_sel[2:0]` to select RO (0=7, 1=11, 2=15, 3=21, 4=31), pulse `cnt_enable` high for the desired gate time, pulse `cnt_clear_latch` high for ≥1 cycle to latch the count, then toggle `byte_sel` and read `uo_out` to retrieve the low and high bytes of the 16-bit count. Check `uio_out[4]` for overflow.
3. **SPI-timed measurement:** set `spi_mode=1`, write `reg_gate_l/h` with the desired gate cycle count, set `reg_ctrl[0]=1` to start the auto gate timer, poll `reg_status` (or watch `uio_out[7]` = `meas_done`) for completion, and read `reg_count_l` / `reg_count_h`. Write `reg_ctrl[1]=1` to clear `meas_done` before the next run.
4. **TRNG:** in SPI mode, set `reg_trng_ctrl[0]=1` and enable at least two ROs; read `reg_trng_data` (0x0D) to consume 8 jitter-XOR bits.
5. **Health monitor:** set `reg_trng_ctrl[1]=1`, write `reg_health_lo`/`reg_health_hi` to the expected frequency window, and read `reg_health_status` (0x0E) after each gate. For long-term audits, sweep all 5 ROs and read `reg_hh_stuck` (0x0F) / `reg_hh_below` (0x10); write `reg_ctrl[2]=1` to clear.
6. **Differential mode:** set `reg_trng_ctrl[2]=1`, set `reg_ro_sel` to RO_A and `reg_diff_sel` to RO_B, then gate as usual — the count represents |f_A − f_B|.

A cocotb test suite (23 tests in `test/test.py`) exercises parallel counting, SPI register roundtrip, auto gate timing, prescaler selection, TRNG/health/differential modes, `uio_oe` direction control, and the new GF-only health-history registers and self-clearing `reg_ctrl[2]`. All tests pass on the RTL. Gate-level simulation is intentionally skipped in CI because the real gf180mcu inverter-chain ring oscillators produce ≈10^9 events/s of simulated time and are not tractable in icarus.

External hardware

No external hardware is required for core operation. For SPI control, an SPI master (microcontroller or FPGA) is connected to `uio[0]` (CS), `uio[1]` (MOSI), `uio[2]` (MISO), `uio[3]` (SCK) at any clock rate up to half the system clock. Optional: an oscilloscope on `uio_out[5]` (raw RO) and `uio_out[6]` (synced RO) for direct frequency observation, and any 8-bit digital sink (logic analyzer or microcontroller GPIO) on `uo_out[7:0]` for reading the latched count byte in parallel mode.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ro_sel[0]	freq_count[0]	spi_cs_n
1	ro_sel[1]	freq_count[1]	spi_mosi
2	ro_sel[2]	freq_count[2]	spi_miso
3	cnt_enable	freq_count[3]	spi_sck
4	cnt_clear_latch	freq_count[4]	overflow
5	byte_sel	freq_count[5]	ro_raw_out
6	spi_mode	freq_count[6]	ro_synced
7	—	freq_count[7]	meas_done

Tiny Quantum Circuit Simulator

by **Oliver Hüttenhofer**

0551 | 20 MHz | HDL Project

`github.com/ohuettenhofer/tiny-qsim-ttgf26a`

"Single-qubit quantum circuit simulator"

How it works

This design simulates a universal one-qubit quantum computer.

It can execute 8 operations:

- Reset the state to |0>
- Apply a X/Y/Z/H/S/T gate to the qubit (see https://en.wikipedia.org/wiki/Quantum_logic_gate for gate definitions)
- Measure and returned the result, upon which the state will collapse to the measured value

Amplitudes are stored in 8 bit fixed point numbers, where the value v is indicated by the binary value round(127 * v). A LFSR is used to generate pseudorandom numbers for measurement.

How to test

ui_in[2:0] indicate which operation should be executed. ui_in[3] contains the start signal, inverting its value will make the chip run the operation.

If the operation was the measure operation, the result can be read from uo[0]. After the operation was executed, the chip inverts the value of the done signal on uo[1] to match that of ready signal and waits for another command.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	opc[0]	msr_res	—
1	opc[1]	done	—
2	opc[2]	—	—
3	start	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Tiny Number Simon

by **Ernesto M**

`0576` `2 Hz` `Wokwi Project`

`github.com/ernestomgz/ernest-tiny-tapeout`

`wokwi.com/projects/465737601403996161`

"Simon-style digital memory game with eight number inputs and a 7-segment display"

Tiny Number Simon is a digital memory game. It displays a growing sequence of numbers from 1 to 8 on a 7-segment display. The player repeats the sequence with eight active-high number inputs.

How it works

The first round displays one number. After the player enters it correctly, the next round repeats the same number and adds one new number. Every successful round increases the score and the sequence length by one.

During computer playback, the display shows numbers 1 through 8 with the decimal point off. During player input, the digit is blank and the decimal point is on.

Each input must be released before it can be detected again. The game also waits for the final correct input to be released before starting the next round.

A wrong input ends the game. The score is the number of fully completed rounds. The display advances once per rising clock edge:

```
H -> blank -> tens digit -> blank -> units digit -> blank
```

For example, a score of 7 is displayed as `H`, blank, `0`, blank, `7`, blank. The score then returns to zero and a new game begins.

The maximum represented score is 63. No best score is saved between games.

Reset is active low. Keep the clock running while reset is held. Different reset hold times normally select different pseudo-random sequences. Reset must include at least one rising clock edge.

How to test

Use a slow clock or a manually stepped clock.

1. Drive all eight number inputs low.
2. Hold active-low reset low for several rising clock edges.

3. Release reset.
4. Confirm that the display shows one number from 1 to 8 and the decimal point is off.
5. Wait for input mode. The digit must be blank and only the decimal point must be on.
6. Drive the matching number input high for at least one rising edge, then return it low.
7. Confirm that the next round repeats the first number and adds one new number.
8. Continue entering the complete sequence.
9. Hold the final correct input high. The next round must not begin until the input returns low.
10. Enter a wrong number during a later round.
11. Verify `H`, blank, decimal tens, blank, decimal units, blank.
12. Verify that only fully completed rounds are counted.
13. Verify that a new one-number game starts after the final blank.
14. Repeat reset with a different hold duration and confirm that the sequence normally changes.

If several number inputs are high, the lowest-numbered input has priority. Normal play should use one input at a time.

Pinout

Pin	Function
`ui[0]`	Number 1
`ui[1]`	Number 2
`ui[2]`	Number 3
`ui[3]`	Number 4
`ui[4]`	Number 5
`ui[5]`	Number 6
`ui[6]`	Number 7
`ui[7]`	Number 8
`uo[0]`	7-segment `a`
`uo[1]`	7-segment `b`
`uo[2]`	7-segment `c`
`uo[3]`	7-segment `d`
`uo[4]`	7-segment `e`
`uo[5]`	7-segment `f`
`uo[6]`	7-segment `g`

`uo[7]`	Decimal point

All outputs are active high. Bidirectional `uio` pins are unused.

External hardware

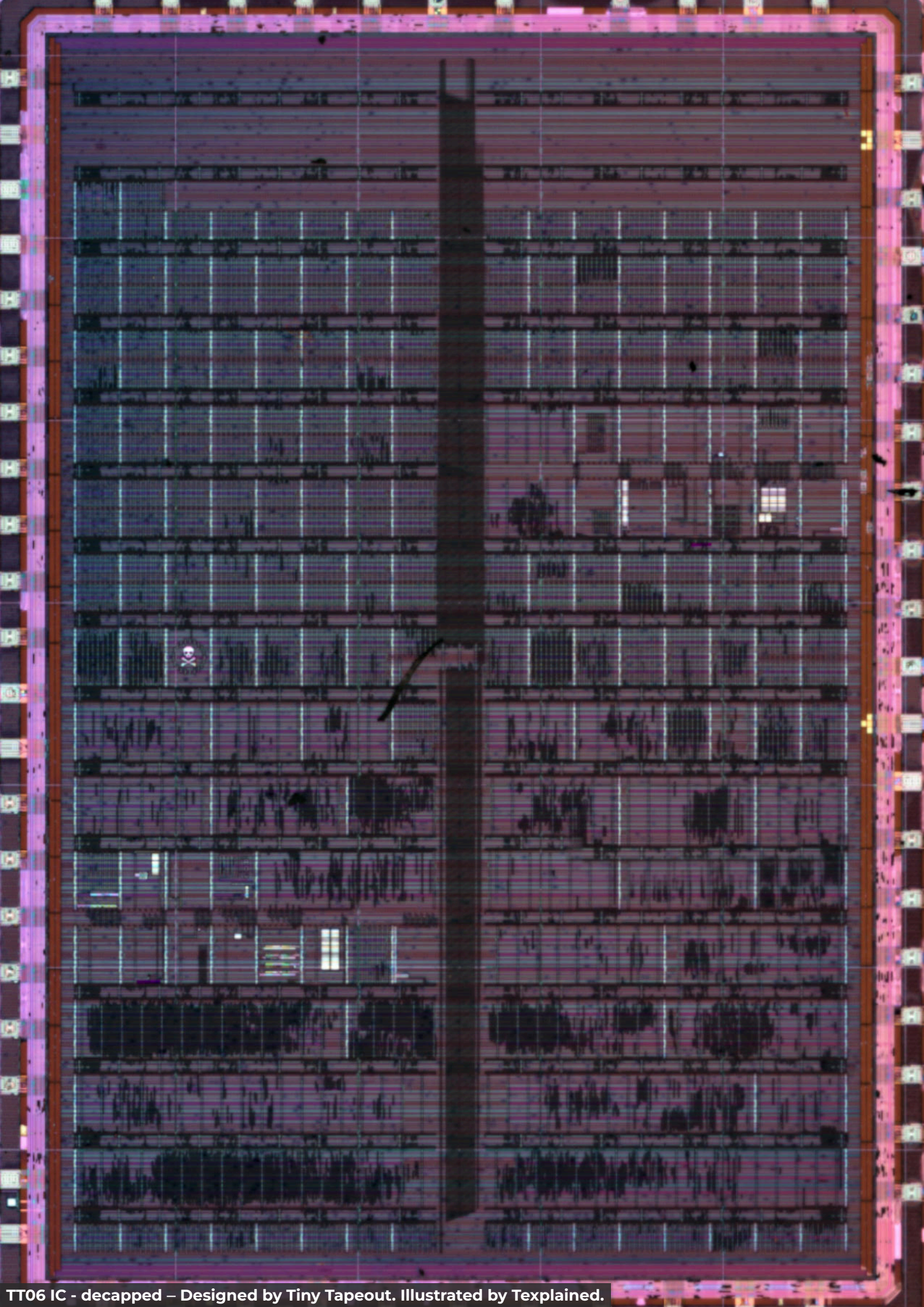
- Eight active-high buttons or switches
- One common-cathode 7-segment display
- One suitable current-limiting resistor per segment
- A slow or manually stepped clock source
- An active-low reset button or signal

No microcontroller or external memory is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	KEY_1	SEG_A	—
`1`	KEY_2	SEG_B	—
`2`	KEY_3	SEG_C	—
`3`	KEY_4	SEG_D	—
`4`	KEY_5	SEG_E	—
`5`	KEY_6	SEG_F	—
`6`	KEY_7	SEG_G	—
`7`	KEY_8	SEG_DP	—



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

Custom DVD Screensaver for VGA

by **Alan Sigala**

`0577` `25.175 MHz` `HDL Project`

`github.com/AlanPandaSK/tt_um_uacj_bouncing_DVD_screensaver`

> *"128x40 bouncing DVD logo with CRT scanlines, checkerboard background, color cycling on wall hits, and random VHS-style corruption effects"*

DVD Screensaver with UACJ IIT Logo

How it works

This project implements a **DVD-style bouncing logo** on a VGA monitor. The logo is a 128×40 bitmap that displays **UACJ IIT**. The design is written in Verilog and targets the Tiny Tapeout ASIC shuttle.

The system is composed of six main blocks: a VGA sync generator, bouncing logic with position registers, a bitmap ROM containing the logo pattern, a three-instance color palette, an RGB output multiplexer with scanline and glitch effects, and a VHS-style glitch/corruption engine. The diagram below shows the overall architecture.

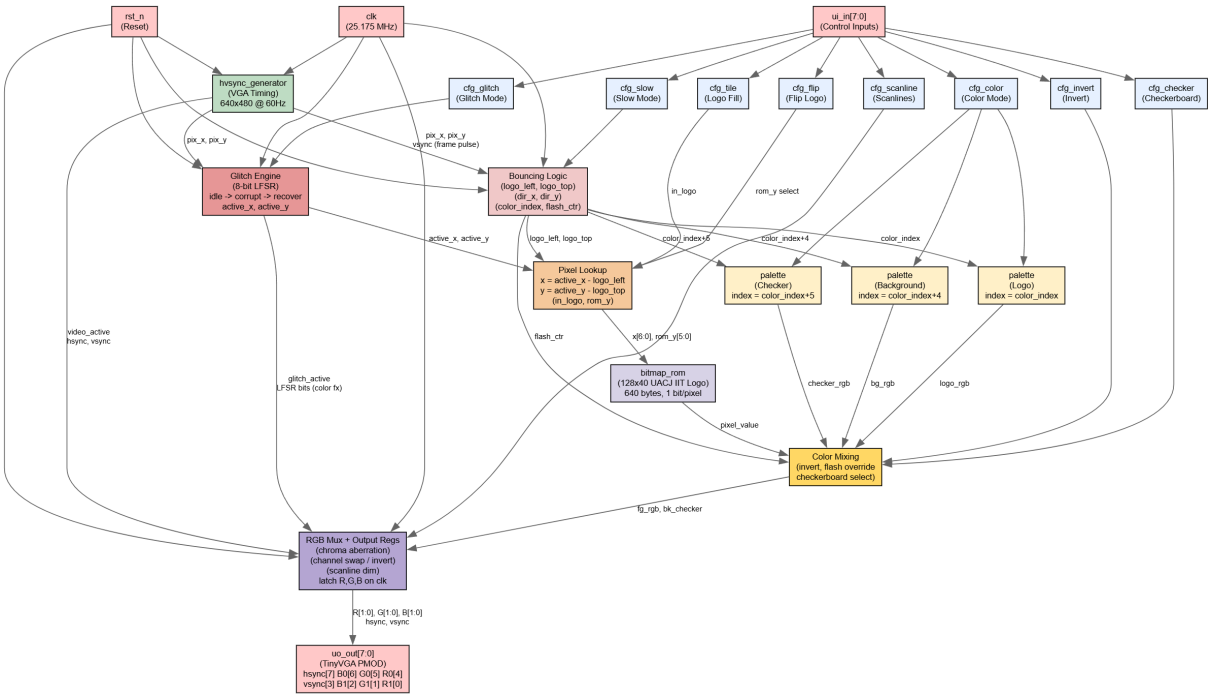


Figure 1. *Block diagram of the DVD screensaver system.*

Block description

- **hvsync_generator** – Generates the HSync and VSync timing signals required for a 640×480 @ 60 Hz VGA display. It also produces the current pixel coordinates (`pix_x`, `pix_y`) and an active video flag (`video_active`).
- **Bouncing Logic & Position Registers** – Maintains the current top-left corner position of the logo (`logo_left`, `logo_top`) and the direction of movement (`dir_x`, `dir_y`). On each frame (detected during vertical blanking), the position updates by one pixel. When the logo reaches a screen edge it bounces, increments the color index, and triggers a white flash. An optional `cfg_slow` mode halves the movement speed by updating only on every other frame.
- **bitmap_rom** – A 640-byte ROM (40 rows x 16 bytes/row) that stores the 128x40 UACJ IIT logo. Each bit represents one pixel: `1` for logo foreground, `0` for background. The ROM is addressed by the current pixel position relative to the logo's top-left corner.
- **palette** – Three instances of an 8-entry color palette (6-bit RGB, 2 bits per channel). The logo color cycles through the palette on each bounce. The background color is offset by +4 indices, and a checkerboard accent color is offset by +5 indices. A `cfg_color` input selects between color (palette) and monochrome (white on black).
- **Checkerboard Background** – When `cfg_checker` is enabled, the background tiles alternate between two colors using a 16x16 XOR pattern (`pix_x[4] ^ pix_y[4]`), adding visual depth behind the bouncing logo.
- **CRT Scanline Effect** – When `cfg_scanline` is enabled, every odd-numbered row has its RGB channels halved, simulating the dark lines of a CRT display.
- **Glitch / Corruption Engine** – A VHS-style corruption effect driven by an 8-bit LFSR (taps at bits 7, 5, 4, 3). The engine has three states: idle, corrupt, and recover. While active it applies:
  - **Horizontal XOR scrambling** – pixel X coordinates are XOR'd with an LFSR-derived mask scaled by glitch intensity.
  - **Vertical XOR scrambling** – pixel Y coordinates are similarly distorted.
  - **Horizontal tearing** – selected rows are offset by a pseudo-random amount, simulating VHS tape tracking errors.
  - **Chromatic aberration** – each RGB channel is XOR'd with independent LFSR bits.
  - **Channel swapping** – R↔G and G↔B swaps are triggered pseudo-randomly.
  - **Inversion flash** – at maximum intensity, colors are fully inverted on random pixels.

The glitch can be triggered manually by pressing `cfg_glitch` (rising-edge detected) or fires automatically at pseudo-random intervals while `cfg_glitch` is held high.

- **RGB Mux & Output Registers** – Combines the pixel value (from ROM), the selected colors (from palette), the flash signal, the scanline dimming, and the glitch distortion to produce the final 2-bit per channel RGB output. The result is latched in registers before being sent to the output pins.

Configuration inputs

The design accepts eight configuration inputs via the `ui_in[7:0]` pins:

Pin	Name	Description
`ui_in[0]`	`cfg_tile`	Debug mode: fill the entire screen with the logo pattern
`ui_in[1]`	`cfg_color`	0 = monochrome, 1 = color palette
`ui_in[2]`	`cfg_invert`	Swaps the logo and background colors
`ui_in[3]`	`cfg_slow`	Halves the movement speed (update every other frame)
`ui_in[4]`	`cfg_flip`	Vertically flips the bitmap when reading from the ROM
`ui_in[5]`	`cfg_checker`	Enables the 16×16 checkerboard background pattern
`ui_in[6]`	`cfg_scanline`	Enables the CRT scanline effect (every odd row darkened)
`ui_in[7]`	`cfg_glitch`	Activates VHS glitch mode; rising edge triggers a glitch burst; holding high enables auto-glitch at random intervals

[!NOTE] The VGA output follows the **TinyVGA PMOD** pin mapping: `uo_out = {hsync, B[0], G[0], R[0], vsync, B[1], G[1], R[1]}`. Connect directly to a TinyVGA PMOD or a compatible VGA DAC.

How to test

Simulation testing (CocoTB)

The project includes a CocoTB testbench that verifies the basic functionality. Due to the complexity of VGA simulation, the default test is configured to pass trivially. For full verification, you can implement the following test procedure:

1. **Navigate to the `test` folder** and ensure `test.py` and `Makefile` are present.

2. **Run the simulation** using:

   ```
   make
   ```

3. **Expected behaviour** (if you implement full testing):
   - The testbench should generate VGA timing signals and verify that the logo bounces correctly within the 640×480 screen boundaries.
   - On each bounce, the simulation should check that:
     - The direction toggles (horizontal or vertical)
     - The color index increments by 1
     - The flash counter (`flash_ctr`) becomes non-zero
   - The bitmap ROM output should match the expected pattern for the UACJ IIT logo.
   - With `cfg_glitch` asserted, the LFSR should advance each frame and `glitch_state` should transition from idle → corrupt → recover → idle.

Hardware testing (FPGA or final chip)

Required equipment

- VGA monitor (supports 640×480 @ 60 Hz)
- TinyVGA PMOD
- Switches for configuration inputs (optional)
- 25.175 MHz clock source (provided by the Tiny Tapeout harness)

Test procedure

1. **Basic functionality test**

Connect the TinyVGA PMOD to your board and to the monitor. Apply power and reset. You should observe:

- The **UACJ IIT logo** bouncing diagonally across the screen
- A brief **white flash** at the moment of impact
- Background colour remains constant (offset from logo colour)

2. **Configuration input tests**

Apply logic levels to the `ui_in[7:0]` pins and observe the behaviour:

Input combination	Expected behaviour
`cfg_tile = 1`	The entire screen fills with the logo pattern (debug mode)
`cfg_color = 1`	Logo changes colors everytime it hits a wall
`cfg_invert = 1`	Logo and background colours swap
`cfg_slow = 1`	Logo moves at half speed (updates every other frame)
`cfg_flip = 1`	Logo appears upside down (vertical mirror)

`cfg_checker = 1`	Background displays a 16×16 checkerboard pattern with an accent color
`cfg_scanline = 1`	Horizontal dark lines appear across the image (CRT scanline simulation)
`cfg_glitch = 1`	VHS-style corruption appears; brief pulse triggers a single burst; holding high triggers random bursts automatically

3. **Boundary testing**

Monitor the logo position as it approaches screen edges:

- It must bounce every time it hits a wall

4. **Flash verification**

The white flash should be visible for approximately 6 frames (about 100 ms at 60 Hz). You can verify this by:

- Using an oscilloscope on the RGB output pins
- Recording the VGA output with a capture card and stepping through frames

5. **Glitch effect verification**

With `cfg_glitch` asserted:

- A rising edge should trigger a single glitch burst lasting 2–17 frames (pseudo-random duration)
- During the burst, observe horizontal tearing, color channel noise, and possible inversion flashes
- After the burst, the image should recover cleanly with no residual corruption
- While `cfg_glitch` is held high, automatic glitch bursts should fire at irregular intervals (approximately every 150 frames when idle)

Expected results

After successful testing, the system should:

- Bounce reliably off all four screen edges
- Cycle through 8 distinct colours (only on wall hits, not continuously)
- Respond correctly to all eight configuration inputs
- Maintain stable VGA sync (no flickering or rolling image)
- Produce convincing CRT and VHS visual effects when the respective modes are enabled

External hardware

Required for operation

Component	Purpose	Specifications
VGA monitor	Display the bouncing logo	640×480 @ 60 Hz (supports standard VGA timings)
VGA cable	Connect the board to monitor	Male DB-15 to male DB-15
TinyVGA PMOD	Convert digital outputs to analog VGA signals	Uses 6 digital lines (2 bits per colour) + HSync + VSync
Clock source	Drive the VGA timing	25.175 MHz (provided by Tiny Tapeout harness)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	cfg_tile	R1	—
1	cfg_color	G1	—
2	cfg_invert	B1	—
3	cfg_slow	VSync	—
4	cfg_flip	R0	—
5	cfg_checker	G0	—
6	cfg_scanline	B0	—
7	cfg_glitch	HSync	—

Balanced Ternary Multiplier

by **Serge Rabyking**

`0578` `50 MHz` `HDL Project`

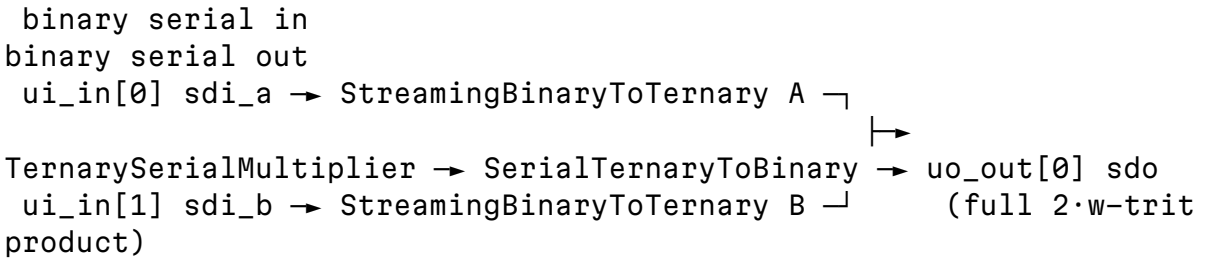
`github.com/lanserge/tt-ternary-multiplier-tiny`

"Serial balanced-ternary multiplier: two 8-bit signed in, 16-bit signed out"

How it works

`tt_um_ttmul` is a **balanced-ternary integer multiplier**. Internally it does not multiply in binary at all — it converts the binary operands to *balanced ternary* (digits −1, 0, +1), multiplies them with a serial ternary shift-and-add core, then converts the product back to binary. The binary interface makes it a drop-in multiplier for a conventional host, while the arithmetic is done the way the historic Soviet **Setun-1958** computer did it.

The datapath reuses building blocks from **Serge Rabyking's Amaranth HDL implementation of the Setun-1958 computer** (the Setun-HDL project):



- **Binary → ternary** uses Horner's method, one bit per clock — the same on-the-fly converter that fed the Tang Nano SPI-flash boot loader.
- **Ternary multiply** is a schoolbook shift-and-add over balanced trits (single full-adder + sign gate), producing the full `2·width`-trit product.
- **Ternary → binary** uses Horner's method, one trit per clock.

A trit is encoded in 2 bits internally (p,n): 0=`00`, +1=`01`, −1=`10`.

Operands are **8-bit two's complement** (−128..+127), converted to **6 trits** each:

Quantity	Value
Operand load bits	8 (two's complement)
Product trits	12
Result bits	16 (signed; 8×8 products span −16256..+16384)

How to test

The host steps `clk` and drives the control pins one bit at a time. All control inputs are active-high and sampled on the rising edge of `clk`.

1. **Reset:** hold `rst_n` low for a few clocks, then release.
2. **Load both operands together:** for 8 clocks, present each bit of A on `ui_in[0]` (sdi_a) and the matching bit of B on `ui_in[1]` (sdi_b), MSB first, two's complement, with `ui_in[2]` (wr) high.
3. **Start:** pulse `ui_in[3]` (go) high for one clock. `uo_out[1]` (busy) stays high while it computes.
4. **Wait** until `uo_out[2]` (ready) goes high.
5. **Read the result:** for 16 clocks, read `uo_out[0]` (sdo) MSB first (two's complement), pulsing `ui_in[4]` (rd) high to advance to the next bit. `uo_out[3]` (done) pulses high on the final bit.

The cocotb test in `test/test.py` exercises this protocol with fixed and random operands, including the corners `127 × 127 = 16129` and `−128 × −128 = 16384`.

External hardware

None required. Drive the pins directly from the RP2040 on the Tiny Tapeout demo board (or any microcontroller / logic). No external components needed.

Pinout

Pin	Direction	Function
ui_in[0]	in	`sdi_a` — operand A serial data in (MSB first, two's complement)
ui_in[1]	in	`sdi_b` — operand B serial data in (MSB first, two's complement)
ui_in[2]	in	`wr` — shift-in strobe (shifts one bit of A and B per asserted clk)
ui_in[3]	in	`go` — start multiply (pulse after both operands loaded)
ui_in[4]	in	`rd` — shift-out strobe (advance to next result bit)
uo_out[0]	out	`sdo` — serial data out (result bit, MSB first, two's complement)
uo_out[1]	out	`busy` — convert/multiply/convert in progress
uo_out[2]	out	`ready` — result available, drive `rd` to read it out
uo_out[3]	out	`done` — high in the cycle the last result bit is shifted out

The bidirectional `uio` pins are unused (driven as inputs, `uio_oe = 0`).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sdi_a - operand A serial data in (MSB first, two's complement)	sdo - serial data out (result bit, MSB first, two's complement)	—
1	sdi_b - operand B serial data in (MSB first, two's complement)	busy - convert/multiply/ convert in progress	—
2	wr - shift-in strobe (shifts one bit of A and B per asserted clk)	ready - result available, drive rd to read it out	—
3	go - start multiply (pulse after both operands loaded)	done - high in the cycle the last result bit is shifted out	—
4	rd - shift-out strobe (advance to next result bit)	—	—
5	—	—	—
6	—	—	—
7	—	—	—

VGA_screensaver_UACJ

by **Angelica Morales Gallegos**

`#579` `25.175 MHz` `HDL Project`

`github.com/angelicakoehn/tt_um_logoUACJ_MOGA`

"Screensaver VGA adaptado a TinyTapeout con logo UACJ y A.M.G"

How it works

This project is a VGA screensaver that displays a bouncing logo containing the text **UACJ** (top row) and **A.M.G** (bottom row) on a 640×480 @ 60 Hz display. It is based on the original *DVD Screensaver with Tiny Tapeout Logo (Tiny VGA)* by Uri Shaked, adapted and personalized by Angélica Morales Gallegos.

The design is composed of three main modules:

- **`vga_sync_generator`** – Generates the `hsync`, `vsync`, `display_on`, and pixel coordinate (`x`, `y`) signals following standard 640×480 VGA timing.
- **`bitmap_rom`** – Defines which pixels are active based on the current coordinates. The letters U, A, C, J (top) and A, M, G with dots (bottom) are drawn using rectangular coordinate ranges. Coordinates are clamped to the range 0–127 to prevent wrapping artifacts.
- **Top module (`tt_um_...`)** – Integrates both modules into the TinyTapeout standard interface. It handles the bouncing movement of the logo, color changes on each bounce, and drives the 6-bit RGB output (2 bits per channel).

When the logo reaches a screen edge, it reverses direction and cycles through a color palette, replicating the classic DVD screensaver behavior.

How to test

Connect the design to a VGA-compatible display using the TinyTapeout demo board or a compatible VGA PMOD. Apply a 25.175 MHz pixel clock (standard for 640×480 @ 60 Hz).

Inputs (via `ui_in`):

Bit	Function
0	`cfg_solid_color` – When high, selects a solid palette color; when low, enables gradient color mode

Outputs (via `uo_out`):

Bits	Function
7	`vsync`
6	`hsync`
5:4	Red channel (2 bits)
3:2	Green channel (2 bits)
1:0	Blue channel (2 bits)

Reset: Assert `rst_n` low to reset the design, then release high to begin operation.

Once running, the display will show the **UACJ / A.M.G** logo bouncing around the screen and changing color on each edge collision.

The design was verified on the TinyTapeout Playground, confirming correct movement, color changes, proper text rendering, and absence of visual artifacts.

External hardware

- **VGA display** – Any monitor or screen supporting 640×480 @ 60 Hz VGA input.
- **VGA PMOD or breakout board** – Required to connect the TinyTapeout demo board's output signals to a standard VGA connector (HD-15). A resistor DAC may be needed to convert the 2-bit-per-channel digital output to analog VGA levels.

Project Pinout

Digital Pins

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

Tiny Tapeout Workshop Malaga 2jun2026

by **Martin Gonzalez Garcia**

`580` `10 MHz` `Wokwi Project`

`github.com/mgonzalezg/tt-workshop-malaga-2jun2026`

`wokwi.com/projects/465731466664816641`

"4-bit lfsr with reset"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

SPI-Controlled 8-Channel LED Driver

by **Grace Eysenbach**

`0581` `50 MHz` `HDL Project`

`github.com/gey16/tt_um_grace_spi_led_driver`

"SPI peripheral with addressable 8-bit registers for controlling 8x LEDs"

How it works

This chip is an SPI-controlled 8-channel LED driver. An SPI master (such as the RP2040/RP2350 on the Tiny Tapeout dev board) communicates with the chip over a 4-wire SPI interface (CS, SCLK, MOSI, MISO). Each SPI transaction is 16 bits: the first byte selects a register address and direction (read or write), and the second byte carries the data.

Internally, the chip contains a 16-register file (9 read/write, 7 read-only). Eight BRIGHT registers (0x0–0x7) set the PWM brightness of 8 LED outputs on `uo_out[0..7]`.

A shared 8-bit counter (`pwm_counter`) cycles 0→255, ticking every `2^PRESCALER` system clock cycles. Each LED output is: `uo_out[n] = ENABLE && (pwm_counter < BRIGHT_n)`, with two special cases: BRIGHT=0xFF → always on, BRIGHT=0x00 → always off. Intermediate values set duty cycle: e.g. 0x80 = 50%, 0x40 = 25%.

With the default PRESCALER=8 and a 50 MHz system clock, PWM refresh rate ≈ 760 Hz (flicker-free).

SPI input signals (CS, SCLK, MOSI) are passed through 2-flop synchronizers before entering the SPI FSM, safely crossing from the asynchronous SPI clock domain into the chip's 50 MHz system clock domain.

Limitations

- SPI mode: CPOL=0, CPHA=0 only (sample on rising SCLK edge)
- SCLK must be ≤ system clock / 20 (≤ 2.5 MHz at a 50 MHz system clock)
- Single register access per CS assertion — no burst transfers
- Data is MSB first

Connection

Pin	Direction	Signal	Description
`uio[0]`	Input	`spi_cs_n`	SPI chip select (active low)
`uio[1]`	Input	`spi_mosi`	SPI master-out slave-in

uio[2]	Output	spi_miso	SPI master-in slave-out
uio[3]	Input	spi_clk	SPI clock
uo[0..7]	Output	LED0..LED7	LED drive outputs

ui_in and uio[4..7] are unused.

Protocol

SPI settings

Parameter	Value
Mode	CPOL=0, CPHA=0
Bit order	MSB first
Transaction length	16 bits
CS polarity	Active low
Max SCLK	2.5 MHz (at 50 MHz system clock)

CS must remain low for the entire 16-bit transaction.

Transaction format

```
Bit:   [15]    [14:12]      [11:8]    [7:0]
MOSI:  R/W     reserved     addr      data (write) | don't-care
(read)
MISO:  0       0            0         0 (write)    | data (read)
```

- R/W: 1 = write, 0 = read
- reserved: ignored on decode
- addr: 4-bit register address (0x0–0xF)
- data: value to write, or register contents returned on read

Register map

Address	Name	Access	Reset	Description
0x0	BRIGHT_0	RW	0x00	PWM brightness for LED 0. 0x00=off, 0xFF=always on, 0x01–0xFE=duty cycle (value/256)
0x1	BRIGHT_1	RW	0x00	PWM brightness for LED 1
0x2	BRIGHT_2	RW	0x00	PWM brightness for LED 2
0x3	BRIGHT_3	RW	0x00	PWM brightness for LED 3
0x4	BRIGHT_4	RW	0x00	PWM brightness for LED 4
0x5	BRIGHT_5	RW	0x00	PWM brightness for LED 5
0x6	BRIGHT_6	RW	0x00	PWM brightness for LED 6

0x7	BRIGHT_7	RW	0x00	PWM brightness for LED 7
0x8	CTRL	RW	0x80	Bits[7:4]: PRESCALER (counter ticks every 2^PRESCALER cycles, default=8). Bits[3:1]: reserved. Bit[0]: ENABLE (1=LEDs active, 0=all off)
0x9	ID	RO	0xA5	Fixed magic byte — read to verify SPI is working
0xA	VERSION	RO	0x01	Design version
0xB	STATUS	RO	0x00	Bit[0]: mirrors CTRL ENABLE. Bit[1]: 1 after a write, 0 after a read. Bits[7:2]: reserved, read 0
0xC	COUNTER	RO	—	Current value of the free-running PWM counter (0–255). Useful for debug
0xD–0xF	RESERVED	RO	0x00	Reserved, always returns 0x00

Writes to addresses 0x9–0xF are silently dropped.

External hardware

Pmod 8LD (e.g., Digilent 410-076 or compatible clone) — 8 discrete LEDs on a standard 12-pin Pmod connector. Plug into the output Pmod header on the TT dev board. `uo_out[0..7]` maps directly to LD0..LD7. No additional components required; the Pmod draws 1 mA per LED from the signal pins.

How to test

Recommended bring-up sequence (each step must pass before the next is meaningful):

1. **SPI alive** — Read ID register (0x9). Expect `0xA5`. If this returns 0x00 or 0xFF, check wiring and CS polarity.
2. **Address decode** — Read VERSION (0xA). Expect `0x01`.
3. **RW loopback** — Write `0xA5` to BRIGHT_0 (0x0), read back. Expect `0xA5`. Repeat with `0x5A`.
4. **RO protection** — Write `0xFF` to ID (0x9), read back. Expect `0xA5` (write was dropped).
5. **ENABLE control** — Write `0x81` to CTRL (0x8) (PRESCALER=8, ENABLE=1). Read STATUS (0xB), expect bit[0]=1.
6. **LED output** — With ENABLE=1, write `0xFF` to BRIGHT_0 (0x0). `uo_out[0]` should go high (always on). Write `0x80` — 50% brightness. Write `0x00` — LED off.

7. **All channels** — Repeat step 6 for BRIGHT_1–BRIGHT_7 to verify all 8 outputs.
8. **COUNTER live** — Read COUNTER (0xC) twice with a short delay between. Values should differ (counter is free-running).

Example MicroPython (RP2040, SoftSPI):

```
import time
from machine import SoftSPI, Pin
import random

# Set project Clock Speed + Reset Project
tt.clock_project_PWM(10_000_000)
tt.reset_project(True)
import time
time.sleep_ms(10)
tt.reset_project(False)
time.sleep_ms(10)

spi = SoftSPI(baudrate=100_000, polarity=0, phase=0,
              sck=Pin(28), mosi=Pin(26), miso=Pin(27))
cs = Pin(25, Pin.OUT, value=1)

def spi_write(addr, data):
    cs(0)
    spi.write(bytes([(1 << 7) | addr, data]))
    cs(1)

def spi_read(addr):
    cs(0)
    buf = bytearray(2)
    spi.write_readinto(bytes([addr, 0x00]), buf)
    cs(1)
    return buf[1]

# Verify SPI is alive (read ID + Version registers)
assert spi_read(0x9) == 0xA5    # ID = 0xA5
assert spi_read(0xA) == 0x01    # Version = 0x01

# STATUS register checks
spi_write(0x8, 0x81)                    # write to CTRL
assert spi_read(0xB) & 0x02 != 0       # LAST_OP_WAS_WRITE=1
spi_read(0xB)                           # read STATUS
assert spi_read(0xB) & 0x02 == 0       # LAST_OP_WAS_WRITE=0
assert spi_read(0xB) & 0x01 != 0       # ENABLE=1 mirrors CTRL

# COUNTER register — verify it's live
c1 = spi_read(0xC)
time.sleep_ms(50)
c2 = spi_read(0xC)
```

```
assert c1 != c2, "counter not incrementing!"
print(f"counter: {c1} → {c2}")


# Enable LEDs (PRESCALER=8, ENABLE=1) and test channel 0
spi_write(0x8, 0x81)   # CTRL: PRESCALER=8, ENABLE=1
spi_write(0x0, 0xFF)   # BRIGHT_0: always on
spi_write(0x0, 0x80)   # BRIGHT_0: 50% brightness
spi_write(0x0, 0x00)   # BRIGHT_0: off

# Light all 8x LEDs one at a time (full brightness)
for i in range(8):
    spi_write(i, 0xFF)
    time.sleep_ms(500)

# Test PWM Control
spi_write(0x8, 0x81)   # CTRL: PRESCALER=8, ENABLE=1
for brightness in [0x20, 0x80, 0xC0, 0xFF, 0x00]:
    for i in range(8):
        spi_write(i, brightness)
    time.sleep_ms(500)

# Party Mode
spi_write(0x8, 0x81)   # CTRL: PRESCALER=8, ENABLE=1
for _ in range(30):
    for i in range(8):
        brightness = random.choice([0x00, 0x20, 0x40, 0x80, 0xC0,
0xFF])
        spi_write(i, brightness)
    time.sleep_ms(150)
```

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	LED0	spi_cs_n
1	—	LED1	spi_mosi
2	—	LED2	spi_miso
3	—	LED3	spi_clk
4	—	LED4	—
5	—	LED5	—
6	—	LED6	—
7	—	LED7	—

Simple Sprinkler

by **Noah Perez**

582 Wokwi Project

github.com/noahzperez29/Noah_Simple_Sprinkler

wokwi.com/projects/451184391728659457

"Simple sprinkler that functions based on 4 conditions."

How it works

The Simple Sprinkler depends on 2 conditions to properly function: the sprinkler system must be enabled and it must have a sufficient supply of water. Of those 2 conditions, they are dependent on a further 4 conditions.

- A (Rain)
  - A = 1 (Currently raining)
  - A = 0 (Not raining)
- B (Water Supply)
  - B = 1 (Sufficient water available)
  - B = 0 (Water supply insufficient)
- C (Soil Moisture)
  - C = 1 (Soil is already moist)
  - C = 0 (Soil is dry)
- D (Time)
  - D = 1 (Current time is within watering period)
  - D = 0 ((Current time is not within watering period)

How to test

Adjust inputs A through D (IN1 - IN4) to see which combinations toggle the LEDs. For the sprinkler to fully function both LEDs must be lit. The truth table is included in README.md.

External hardware

None

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Rain (A)	Sprinkler System (S)	—
1	Water Supply (B)	Water Supply (W)	—

#	Input	Output	Bidirectional
2	Soil Moisture (C)	—	—
3	Time (D)	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Universal Binary to Segment Decoder

by **Rebecca G. Bettencourt**

583 HDL Project

`github.com/RebeccaRGB/ttgf-ubcd`

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

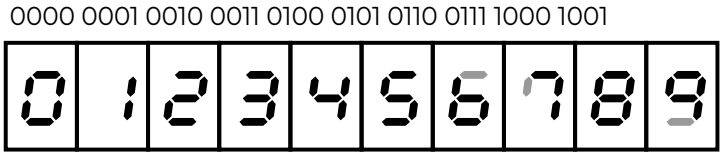
How it works

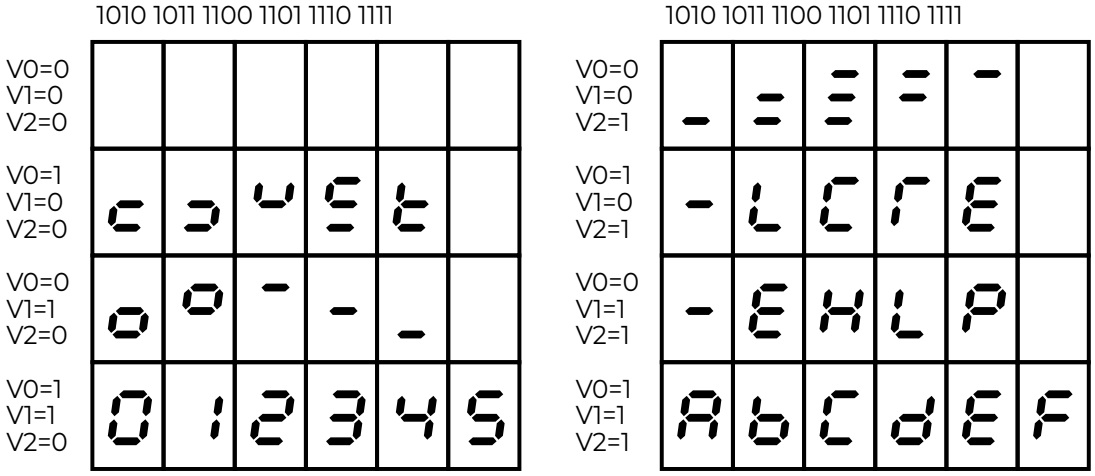
This project is composed of four modules:

- 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:

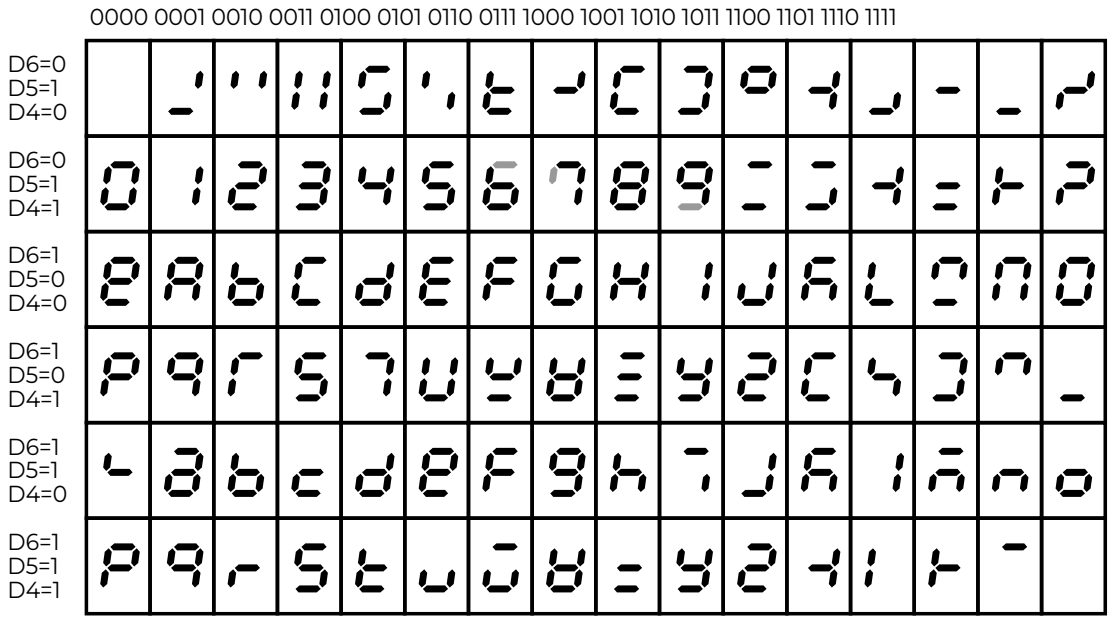
—	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Input - X6
1	B	Segment b	Input - X7

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

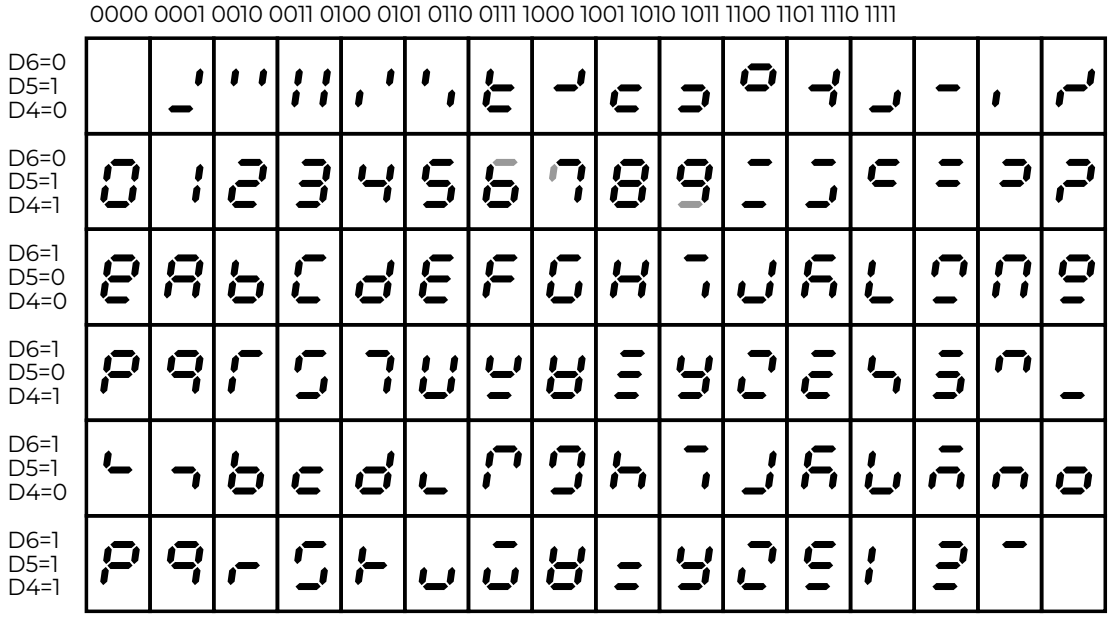
ASCII to seven segment decoder

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).

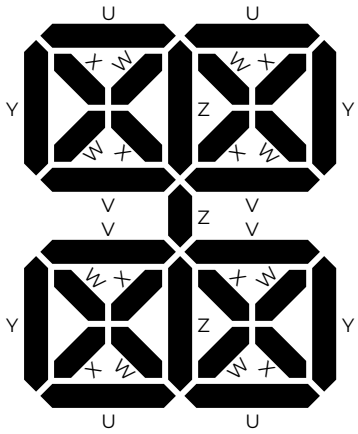
The pin assignments in this mode are:

—	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

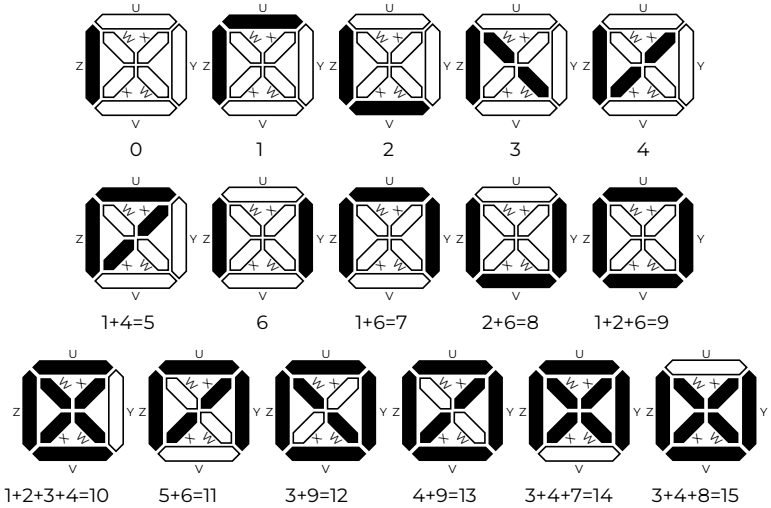
Dual BCD to Cistercian numeral decoder

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.

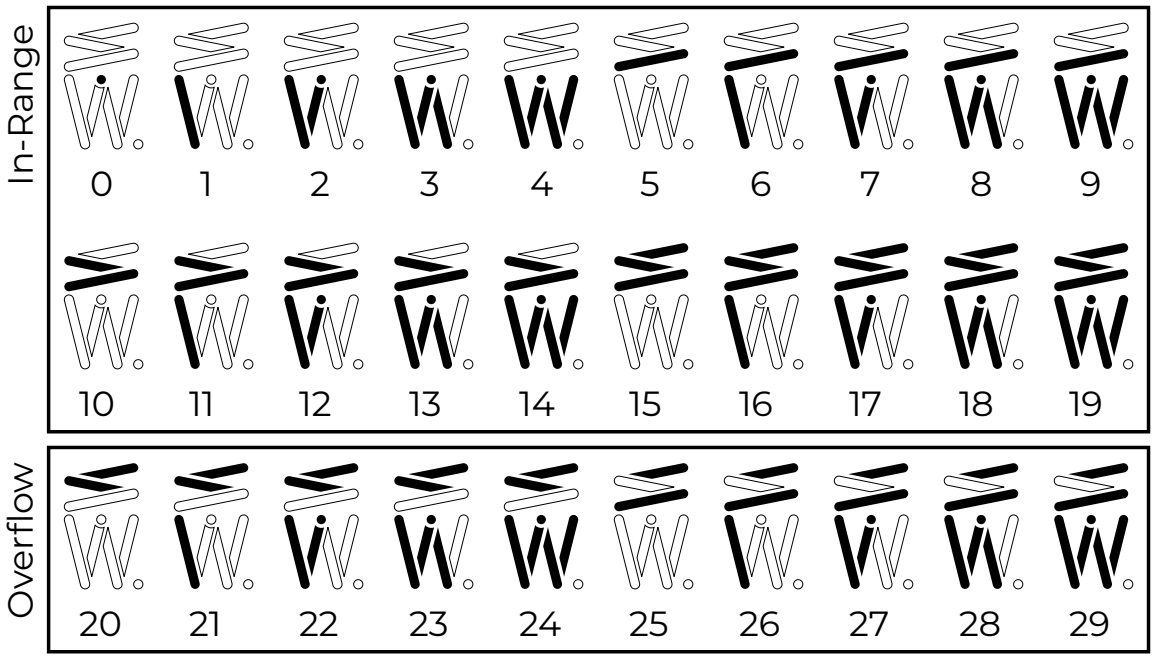
The pin assignments in this mode are:

—	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

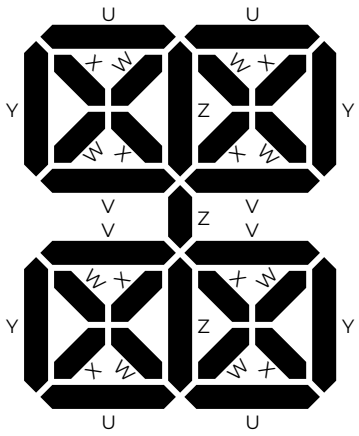
How to test

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)

Rhyloo's first Wokwi design

by **Jorge Benavides Macías**

0608 50 MHz Wokwi Project

github.com/rhyloo/tiny-tapeout-test

wokwi.com/projects/465731439156454401

"Just a simple logic"

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	Input_0	Output_0	—
1	Input_1	Output_1	—
2	Input_2	Output_2	—
3	Input_3	Output_3	—
4	Input_4	Output_4	—
5	Input_5	Output_5	—
6	Input_6	Output_6	—
7	Input_7	Output_7	—

VCO driven by DAC

by **algofoogle (Anton Maurovic)**

`0609` `HDL Project`

`github.com/algofoogle/ttgf26a-vco`

"*Mixed-signal project with DAC setting a VCO's control voltage*"

How it works

A VCO (specifically a current-starved ring oscillator consisting of a 5-inverter ring) is supplied a control voltage by a 9-bit DAC (giving an estimated 6.45mV resolution over the range 0..3.3V). This means it can be digitally controlled to hit different frequencies estimated to be in the range of 3MHz to 400MHz.

This in turn goes into a simple digital block hardened with LibreLane (using 9T gf180mcuD standard cells) to drive a 5-bit counter which is then used to provide different clock divider output stages.

How to test

No external clock is used with this project. Just do the following:

1. Set `ui_in` to 0 and `uio_in[0]` to 0.
2. Apply power.
3. Expect no toggling on `uio_out[7:2]`.
4. Set `ui_in` to 77 (0x4D, or `0100_1101`).
5. Expect toggling on `uio_out[7]` of about 1.9MHz. Multiply this by 32 to get the actual internal VCO frequency (maybe 61MHz?)
6. Observe frequency doubling as you go down from `uio_out[6]` to `uio_out[3]`
7. `uio_out[2]` is the direct VCO output, so will be the fastest.
8. Try varying `{ui_in[7:0],uio_in[0]}` which together form the 9-bit input to the VCO's control DAC:
   - Note that because the LSB is in `uio_in[0]`, setting `ui_in` to 77 (as above) is actually shifted by one, so it represents a DAC code of `77<<1`, i.e. 154, or `(154/512)*3.3 = 0.993V`.
   - Setting the 9-bit input code to anything 85 (0.55V) will probably stop the VCO from oscillating.
   - Setting it to anything above 388 (2.5V) might make the counter unstable.

External hardware

- Oscilloscope just to monitor the `uio_out[7:2]` pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	vco_in[1]	—	vco_in[0]
1	vco_in[2]	—	—
2	vco_in[3]	—	vco_out
3	vco_in[4]	—	counter[0]
4	vco_in[5]	—	counter[1]
5	vco_in[6]	—	counter[2]
6	vco_in[7]	—	counter[3]
7	vco_in[8]	—	counter[4]

Tiny Tapeout Marcos Fernandez

by **Marcos**

0610 Wokwi Project

github.com/marcosfdez99/tt-marcos-fernandez

wokwi.com/projects/465732536551273473

"Test"

How it works

Explain how your project works

How to test

Explain how to use your project

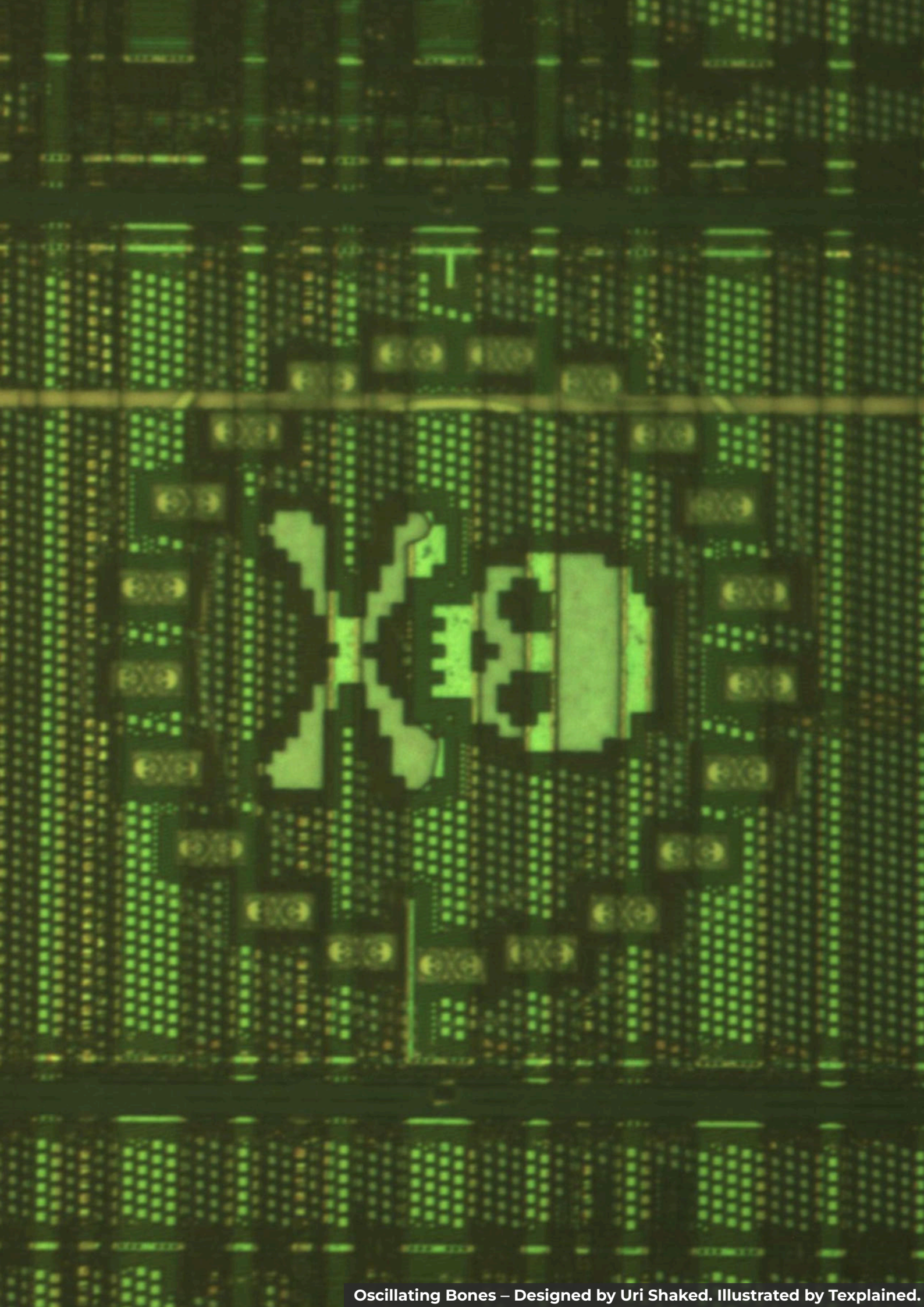
External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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



Oscillating Bones – Designed by Uri Shaked. Illustrated by Texplained.

UNIZG-FER VGA project

by **Daniel Hofman**

0611 HDL Project

github.com/hofi505/fer-vga-tt26a

"*FER logo and music*"

How it works

The project displays a FER logo and plays music

How to test

Connect VGA display to the VGA PMOD on the output pins. Use inputs 2-5 (left/right/up/down) to control the direction of the bouncing FER logo. Audio output is on uio[0].

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	cfg_tile - tile mode	R1 - VGA red bit 1	audio out
1	—	G1 - VGA green bit 1	—
2	btn_left	B1 - VGA blue bit 1	—
3	btn_right	vsync	—
4	btn_up	R0 - VGA red bit 0	—
5	btn_down	G0 - VGA green bit 0	—
6	—	B0 - VGA blue bit 0	—
7	—	hsync	—

Tiny Tapeout Template

by **Sankori**

0612 | 100 Hz | Wokwi Project

github.com/sankori-ai/tt-template

wokwi.com/projects/465737290543084545

"My first project"

How it works

Explain how your project works

How to test

Explain how to use your project

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any

Project Pinout

Digital Pins

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

Dyadic PWM

by **Maksudjon Usmonov**

`0613` `50 MHz` `HDL Project`

`github.com/maqsudbek/ttgf-dyadic-pwm-iot`

"*Digital PWM generator with selectable width, dyadic and dithering modulation*"

How it works

This is a **digital PWM (DPWM) generator** with selectable resolution, dyadic modulation and dithering. A 12-bit control word sets the duty cycle; the design produces complementary high-/low-side outputs with dead-time, plus a sync clock.

Selectable width. The PWM resolution is configurable to **5, 6, 7, 8 or 9 bits**. All widths share the same **513-cycle switching period** (97.5 kHz at a 50 MHz clock); the chosen B-bit duty is scaled onto that period by `scaled = duty·2^(9−B) + max(1, 2^(8−B))`.

Modes. Beyond plain **Normal** PWM, the design adds:

- **Dyadic** — the lower `m` bits of the control word are distributed as a +1 sequence across a 2^m-period window (the bit selected each period is `lsb[highest_set_bit(counter)]`), raising the *effective* resolution by up to 7 bits without extra base width.
- **Dithering v1/v2/v3** — a sigma-delta-style +1 decision (`lsb ≥ counter`); v2 samples the LSB once per 2^m window, v3 also samples the base duty once per window.
- A **constant dyadic word** can replace the control LSBs as the modulation source.

Configuration interface. Static settings live in a small register file written over the pins. `uio_in[7]` selects the mode:

`uio_in[7]`	Meaning	`ui_in[7:0]`	`uio_in[6:4]`	`uio_in[3:0]`
`0` (run)	drive control word	`ctrl[11:4]`	—	`ctrl[3:0]`
1 (config)	write a register	write data	reg address	—

Config registers (written while `uio_in[7]=1`):

Addr (`uio_in[6:4]`)	Data (`ui_in`)

0	[2:0]=dyadic_len (0–7), [5:3]=mode (0=Normal,1=Dyadic,2–4=Dither v1–v3), [6]=const_dyadic_flag
1	[2:0]=pwm_bits_sel (0→5-bit … 4→9-bit)
2	[6:0]=dyadic_word (constant modulation word)

After reset the design defaults to **8-bit Normal** PWM (dyadic_len=0), so it behaves as a plain 8-bit PWM until configured. Outputs: `uo_out[0]`=high-side, `uo_out[1]`=low-side (complementary, 6-cycle / 120 ns dead-time), `uo_out[2]`=sync clock, `uo_out[7:3]`=duty MSBs (debug, normalised to the 9-bit domain). All `uio` pins are inputs.

How to test

1. Hold `rst_n` low for several clocks, then release. With no configuration the design is an 8-bit PWM: drive `ui_in` with the duty (e.g. `0x80` ≈ 50 %), keep `uio_in = 0x00`, and observe `uo_out[0]`/`uo_out[1]` (complementary) and the 97.5 kHz sync clock on `uo_out[2]`.
2. **Configure** features by pulsing the config interface: set `uio_in[7]=1`, put the register address on `uio_in[6:4]` and the data on `ui_in`, clock once, then drop `uio_in[7]` back to 0.
   - *Select 5-bit width:* write addr 1 = `0x00`.
   - *Dyadic, 4-bit:* write addr 0 = `0x0C` (mode=1, dyadic_len=4); then run with `uio_in[3:0]` as the LSB word — the average duty shifts by `lsb/16` over 16 periods.
   - *Dithering v1:* write addr 0 = `0x14` (mode=2, dyadic_len=4).
   - *Constant dyadic word:* write addr 0 = `0x4C` (const flag + dyadic mode, len 4) and addr 2 = the 7-bit word; the modulation then ignores the control LSBs.
3. The cocotb testbench (`test/test.py`) exercises reset, all five widths, max-duty/dead-time, the sync clock, dyadic mode, the three dithering modes and the constant-word path.

External hardware

None required for bench testing. For power-electronics use, wire `uo_out[0]` (high-side) and `uo_out[1]` (low-side) to a half-bridge gate driver — the built-in dead-time prevents shoot-through in, e.g., a synchronous buck converter or class-D stage.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ctrl[4] / cfg_data[0]	pwm_high	ctrl[0] / -
1	ctrl[5] / cfg_data[1]	pwm_low	ctrl[1] / -
2	ctrl[6] / cfg_data[2]	sync_clk	ctrl[2] / -
3	ctrl[7] / cfg_data[3]	duty[4]	ctrl[3] / -
4	ctrl[8] / cfg_data[4]	duty[5]	cfg_addr[0]
5	ctrl[9] / cfg_data[5]	duty[6]	cfg_addr[1]
6	ctrl[10] / cfg_data[6]	duty[7]	cfg_addr[2]
7	ctrl[11] / cfg_data[7]	duty[8]	cfg_we (1=config write)

ram 1 bit Copy

by **siningua_alagartos**

0614 | 10 kHz | Wokwi Project

github.com/siningua/tt-ram-1-bit

wokwi.com/projects/465630130495825921

"ram de 1 bit con dos posiciones"

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	set/reset	dato	—
1	posicion	—	—
2	r/w	—	—
3	dato	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Video mode tester

by **htfab**

`0615` `64 MHz` `HDL Project`

`github.com/htfab/ttgf-vga-tester`

"*Experiment with different VGA timing parameters*"

Author: htfab

Peripheral index: 28

What it does

There is quite some variability between screens (and VGA/HDMI adapters) in the set of VGA timing configurations they support.

Due to constraints and optimization pressures, VGA designs on Tiny Tapeout typically use a single resolution that cannot be changed without a respin. It would therefore be useful to gather some crowdsourced information on what VGA modes are well supported among the community.

This peripheral facilitates gathering that information.

It allows setting the horizontal and vertical timing parameters (visible pixels, front porch, sync pulse, back porch) and displays a simple test pattern on the screen. There is a thin white border along the screen edges to quickly check whether anything was cut off.

Each phase (visible pixels, front porch, sync pulse, back porch) is described by its length in pixels (a 13-bit integer) and 3 single-bit flags. Internally all 4 phases are identical and the flags are the mechanism to differentiate their behaviour:

- bit 15: keep hsync/vsync high during this phase
- bit 14: allow data on the r/g/b pins during this phase
- bit 13: advance to the next line/frame at the end of this phase

For instance, a video mode with positive hsync/vsync polarity could use flags 010 for the visible pixels, 000 for the front porch, 100 for the sync pulse and 001 for the back porch.

Register map

Address	Name	Access	Description
0x00	DATA	R/W	Horizontal visible pixels, high byte (incl. flags)
0x01	DATA	R/W	Horizontal visible pixels, low byte

0x02	DATA	R/W	Horizontal front porch, high byte (incl. flags)
0x03	DATA	R/W	Horizontal front porch, low byte
0x04	DATA	R/W	Horizontal sync pulse, high byte (incl. flags)
0x05	DATA	R/W	Horizontal sync pulse, low byte
0x06	DATA	R/W	Horizontal back porch, high byte (incl. flags)
0x07	DATA	R/W	Horizontal back porch, low byte
0x08	DATA	R/W	Vertical visible pixels, high byte (incl. flags)
0x09	DATA	R/W	Vertical visible pixels, low byte
0x0a	DATA	R/W	Vertical front porch, high byte (incl. flags)
0x0b	DATA	R/W	Vertical front porch, low byte
0x0c	DATA	R/W	Vertical sync pulse, high byte (incl. flags)
0x0d	DATA	R/W	Vertical sync pulse, low byte
0x0e	DATA	R/W	Vertical back porch, high byte (incl. flags)
0x0f	DATA	R/W	Vertical back porch, low byte

How to test

To use the universally supported 640x480 @ 60 Hz video mode, we would like to set

- Horizontal visible pixels: 640 (high byte 2, low byte 128)
  - 0x00: 66 (“visible” flag adds 64)
  - 0x01: 128
- Horizontal front porch: 16 (high byte 0, low byte 16)
  - 0x02: 0
  - 0x03: 16
- Horizontal sync pulse: 96 (high byte 0, low byte 96)
  - 0x04: 128 (“sync” flag adds 128)
  - 0x05: 96
- Horizontal back porch: 48 (high byte 0, low byte 48)
  - 0x06: 32 (“advance” flag adds 32)
  - 0x07: 48
- Vertical visible pixels: 480 (high byte 1, low byte 224)
  - 0x08: 65 (“visible” flag adds 64)
  - 0x09: 224
- Vertical front porch: 10 (high byte 0, low byte 10)
  - 0x0a: 0
  - 0x0b: 10
- Vertical sync pulse: 2 (high byte 0, low byte 2)
  - 0x0c: 128 (“sync” flag adds 128)

    - 0x0d: 2
- Vertical back porch: 33 (high byte 0, low byte 33)
    - 0x0e: 32 (“advance” flag adds 32)
    - 0x0f: 33

After setting the pixel clock to 25 MHz and writing these registers the test pattern should appear on the screen connected to the Tiny VGA PMOD.

External hardware

Tiny VGA PMOD

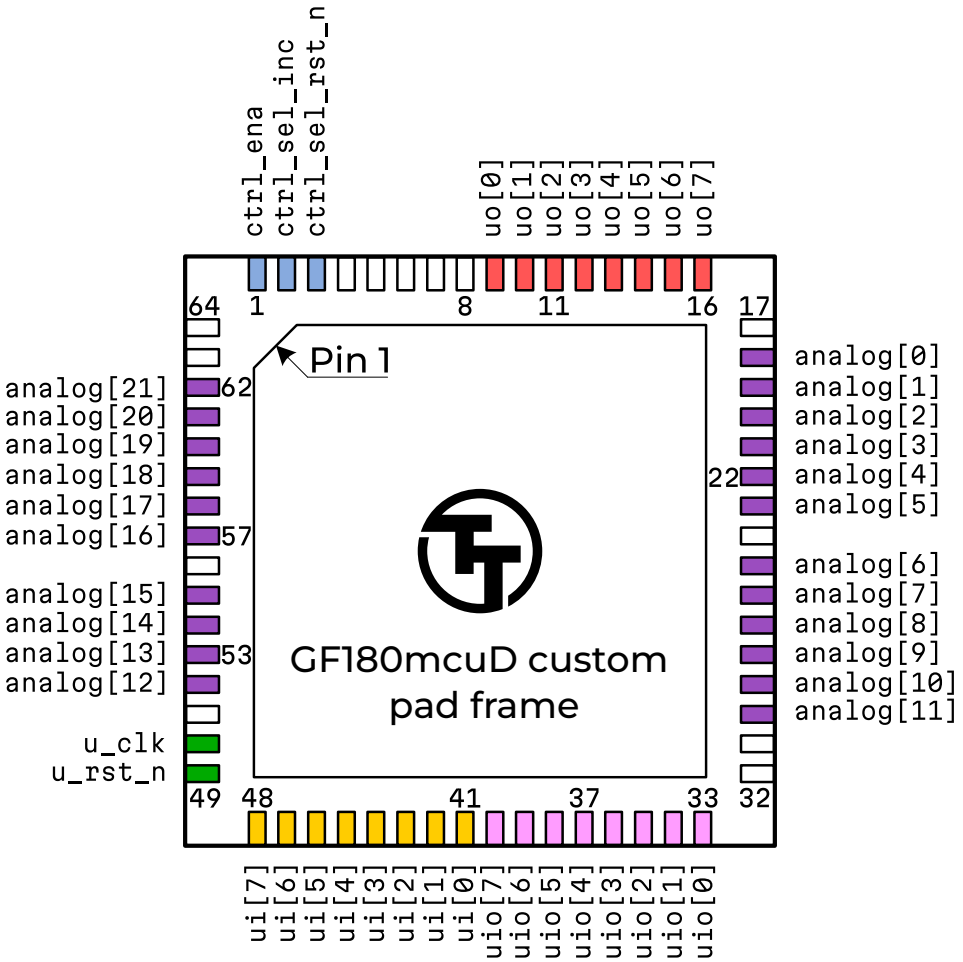
Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	TinyVGA red 1	—
1	—	TinyVGA green 1	—
2	—	TinyVGA blue 1	—
3	—	TinyVGA vsync	spi_miso
4	—	TinyVGA red 0	spi_cs_n
5	—	TinyVGA green 0	spi_clk
6	—	TinyVGA blue 0	spi_mosi
7	—	TinyVGA hsync	—

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:

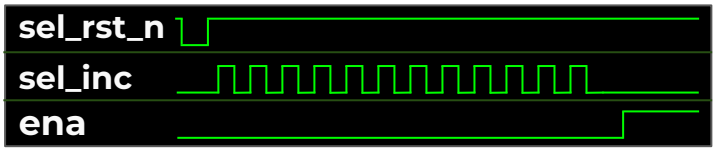


Figure 1: Mux signals for activating the design at address 12

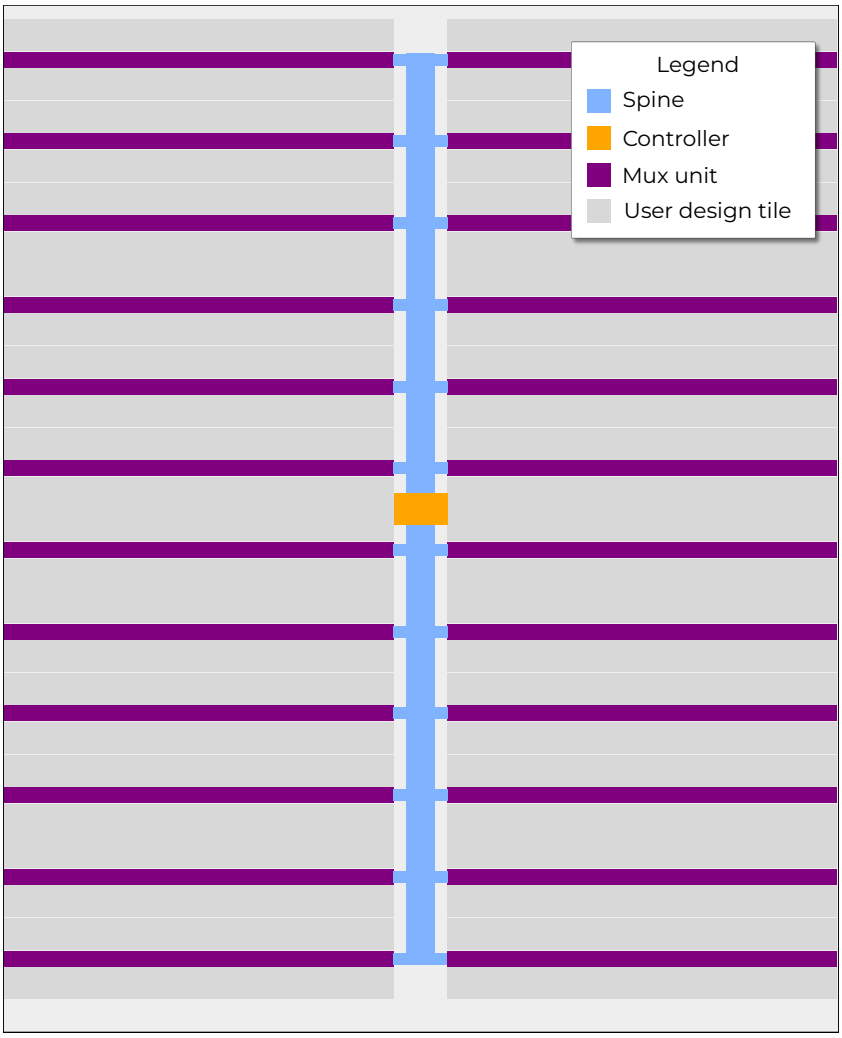


Figure 2: Mux Diagram

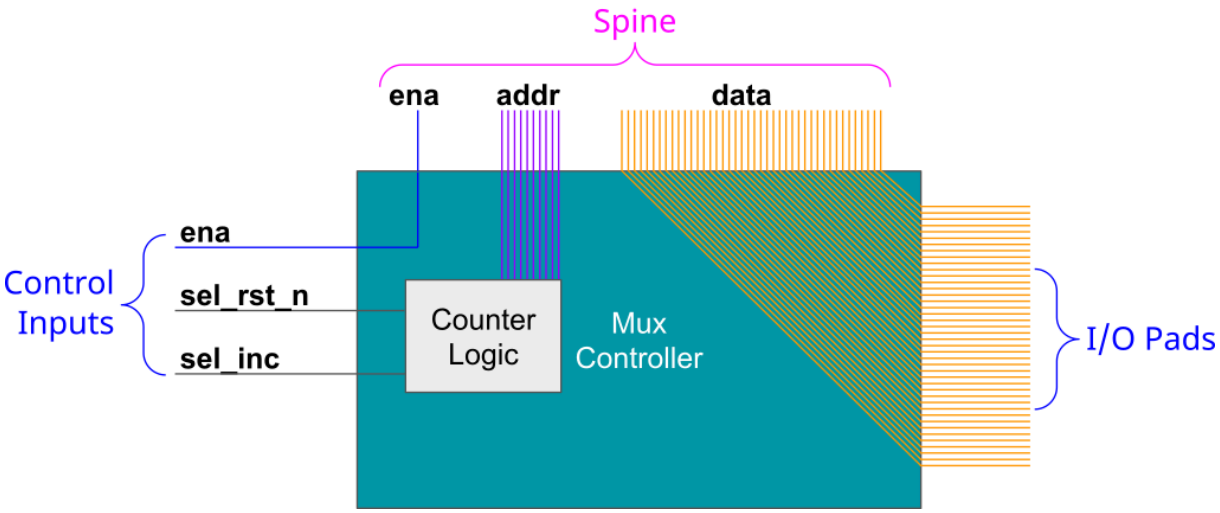


Figure 3: Mux Controller Diagram

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