Tiny Tapeout IHP 26b Datasheet

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

September 22, 2026

Table of Contents

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/ttihp26b-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]

Inverted inverters

by **Mikail Gedik**

0003 HDL Project

`github.com/mikailgedik/invertedinverter`

"*Long chain of inverters*"

How it works

These are just a bunch of inverters in a long chain. This can maybe generate some randomness

How to test

To simulate combinatorial loops, we replace each not/nor with a dummy gate. Each dummy gate gets a seed and has an internal counter. That counter is used to give it schmitt-trigger like behaviour, so to simulate a real gate (even if it's a bad sim). The simulation takes forever and isn't very representative though.

Since the sim takes forever for the rng-stuff, we only test it locally and remotely we only check whether the registers work correctly

Gatelevel simulation cannot work, since we have combinatorial loops! It is therefore disabled

External hardware

Not really

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A0	D0o	D0i
1	A1	D1o	D1i
2	A2	D2o	D2i
3	A3	D3o	D3i
4	A4	D4o	D4i
5	A5	D5o	D5i
6	SRC	D6o	D6i
7	WE	D7o	D7i

Tiny FPGA

by **Yuri Honegger**

0005 50 MHz HDL Project

github.com/recursivetree/tt-fpga

"A simple FPGA"

How it works

This is a tiny FPGA. It has 4-input LUTs and a routing fabric!

This project was a spontaneous submission. I did not have the time to properly validate, document or optimize the design.

How to test

If you are still undeterred:

1. Generate a bitstream using the bitstream tools
2. enable the configmode pin (ui_in[7])
3. shift the bitstream in
4. test the fpga using the io pins

It might be advisable to read the testbenches in https://github. com/recursivetree/fpga/ to see how it works.

External hardware

None required

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	input0	output0	unused
1	input1	output1	unused
2	input2	output2	unused
3	input3	output3	unused
4	input4	output4	unused
5	input5	output5	unused
6	unused	unused	unused
7	config_mode	unused	unused

3LFCC PS-PWM Modulator

by **aaaaaa**

`0007` `27 MHz` `HDL Project`

`github.com/JorgeMarinN/ttihp-verilog-template_pucv-test1`

> "*Modulador phase-shift PWM con tiempo muerto para un convertidor Flying Capacitor de 3 niveles, con los dos ciclos de trabajo ajustables por pines*"

How it works

Modulador **PS-PWM** (*Phase-Shift PWM*) para un convertidor Flying Capacitor de 3 niveles. Es el modulador del diseño FPGA de Nicolás Villegas para la Tang Nano 20K (`github.com/nic0villegasc/LushayLabs-TangNano20K`, carpeta `3LFCC`), llevado al chip, con los dos ciclos de trabajo expuestos a pines.

Funciona en **lazo abierto**: no mide ni corrige, genera el PWM que se le pide.

El chip compara cada ciclo de trabajo contra una portadora triangular de 7 bits. Las dos ramas usan dos portadoras **desfasadas 180 grados**, lo que hace que conmuten alternadas y reduce el rizado de la salida. La portadora cuenta de 0 a 127 y vuelve, con un período de 254 ciclos de reloj: **106 kHz** de frecuencia de conmutación a 27 MHz.

Cada rama tiene un PMOS y un NMOS que trabajan de forma complementaria. Para que nunca conduzcan a la vez, cada señal pasa por un generador de **tiempo muerto**: el flanco de apagado es inmediato y el de encendido va retrasado **5 ciclos de reloj (185 ns)**. Así siempre se apaga uno antes de que se prenda el otro.

Con el reset activo, las cuatro llaves quedan apagadas.

Parejas de cada rama

Rama	PMOS (activo a bajo)	NMOS (activo a alto)
1	`uo[0]`	`uo[3]`
2	`uo[1]`	`uo[2]`

How to test

1. Reloj de **27 MHz**.
2. Fija **D1** en `ui[6:0]` y **D2** en `uio[6:0]`. El valor es el ciclo de trabajo en binario, de 0 a 127:

Valor	Ciclo de trabajo
`0000000`	0 %
`1000000`	50 %
`1111111`	100 %

3. Suelta el reset. En `uo[0..3]` aparecen las cuatro señales de puerta.
4. Usa `uo[4]` como disparo del osciloscopio: da un pulso en cada extremo de la portadora.
5. Para comprobar el tiempo muerto, mirá a la vez `uo[3]` y `uo[0]`: entre que el NMOS2 se apaga y el PMOS1 se enciende tiene que haber al menos 185 ns.
6. Para operación simétrica, D1 = D2. Para desbalancear las ramas a mano, poné valores distintos.

Ojo con D = 0: con ciclo de trabajo cero los PMOS no llegan a encender y `uo[0]` y `uo[1]` se quedan en alto. Es correcto, no un fallo.

External hardware

- Etapa de potencia de un convertidor Flying Capacitor de 3 niveles, con drivers de puerta en `uo[0..3]`. **Respetar la polaridad:** los PMOS son activos a bajo y los NMOS activos a alto.
- Interruptores o el RP2040 de la placa de demostración para fijar D1 y D2.
- Osciloscopio para observar las salidas.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	D1 bit 0	PWM0 - puerta PMOS1 (activa a bajo)	D2 bit 0 (entrada)
1	D1 bit 1	PWM1 - puerta PMOS2 (activa a bajo)	D2 bit 1 (entrada)
2	D1 bit 2	PWM2 - puerta NMOS1 (activa a alto)	D2 bit 2 (entrada)
3	D1 bit 3	PWM3 - puerta NMOS2 (activa a alto)	D2 bit 3 (entrada)
4	D1 bit 4	Disparo de fin de rampa de la portadora	D2 bit 4 (entrada)
5	D1 bit 5	—	D2 bit 5 (entrada)
6	D1 bit 6	—	D2 bit 6 (entrada)
7	—	—	—

simple SPI flash streaming NPU

by **Léon Romanens**

0009 100 MHz HDL Project

`github.com/leonllrmc/ttihp26b_simple_NPU`

"A simple NPU streaming weights form a SPI flash"

How it works

TODO ## How to test

TODO

External hardware

A SPI flash, any that can be read with the sequence: [0x03, A[23:16], A[15:8], A[7:0], D0, D1]

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	initial_activation[0]	activation_out[0]	project_extflash_spi_sck
1	initial_activation[1]	activation_out[1]	project_extflash_spi_mosi
2	initial_activation[2]	activation_out[2]	project_extflash_spi_cs
3	initial_activation[3]	activation_out[3]	extflash_spi_miso
4	activation_index[0]	intermediary_activation_out[0]	—
5	activation_index[1]	intermediary_activation_out[1]	—
6	activation_index[2]	intermediary_activation_out[2]	—
7	save_activation	intermediary_activation_out[3]	—

6-bit multi function ALU (eldawly_V2)

by **ziad mokhtar**

0032 HDL Project

`github.com/ziadmokhtar1253-gif/ttihp-verilog-template`

"6-bit ALU featuring RCA, CLA, multipliers, logic, shifts, popcount, and comparison operations"

How it works

This project implements `tt_um_alu_bns`, a 16-bit multi-functional ALU core wrapped in a byte-serial load/read interface to fit within TinyTapeout's 8-bit pin budget.

ALU core (`alu_core`) — a purely combinational block that computes all eight operations in parallel and selects the result by `opcode`:

opcode	Operation	sub_op meaning
000	Ripple Carry Adder (RCA)	—
001	Carry Lookahead Adder (CLA)	—
010	Array Multiplier (16x16→32)	—
011	Wallace Tree Multiplier (16x16→32)	—
100	Logic unit	00=AND, 01=OR, 10=XOR
101	Shift / Popcount	00=shift left, 01=shift right, 10=popcount
110	Comparator (A vs B)	—
111	Popcount (forced, ignores sub_op)	—

All operands, sums, and comparator/logic results are 16-bit; the two multipliers produce a 32-bit product. `Result` is always packed into a 32-bit register (upper bits zero-extended for non-multiply ops), with `Cout` and `Valid` flags set alongside it.

Byte-serial wrapper (`tt_um_alu_bns`) — since `ui_in`/`uio_in` are only 8 bits wide, two 16-bit operands can't be loaded in one clock cycle. Instead, the wrapper exposes a small register-addressed load/read protocol driven by `clk`:

- `ui_in[7:0]` = data byte to load
- `uio_in[7:5]` = REG_SEL (which register/action this cycle targets)
- `uio_in[4]` = LD (load strobe, sampled on the clock edge)

- `uio_in[1:0]` = READ_SEL (which byte of the 32-bit result to output)

REG_SEL values: `000`/`001` load A's low/high byte, `010`/`011` load B's low/high byte, `100` loads the control word (`opcode`, `sub_op`, `Cin` packed into one byte), and `101` (START) latches A/B/Cin/opcode/sub_op into the combinational core and captures `Result`/`Cout`/`Valid` into output registers on that same edge.

`uo_out` continuously shows one byte of the latched 32-bit `Result`, selected by `READ_SEL`. `uio_out[3:2]` expose `Valid` and `Cout`; all other `uio_out` bits are unused. `uio_oe` marks only those two status bits as outputs — every other `uio` pin stays an input, since they're used to drive REG_SEL/LD/READ_SEL from outside.

How to test

Each operation takes 6 clock cycles: load A (2 bytes), load B (2 bytes), load control, then START.

1. **Load A low byte**: `uio_in = 8'b000_1_00xx` (REG_SEL=000, LD=1), `ui_in = A[7:0]`, pulse `clk`.
2. **Load A high byte**: `uio_in = 8'b001_1_00xx`, `ui_in = A[15:8]`, pulse `clk`.
3. **Load B low byte**: `uio_in = 8'b010_1_00xx`, `ui_in = B[7:0]`, pulse `clk`.
4. **Load B high byte**: `uio_in = 8'b011_1_00xx`, `ui_in = B[15:8]`, pulse `clk`.
5. **Load control word**: `uio_in = 8'b100_1_00xx`, `ui_in = {2'b0, Cin, sub_op[1:0], opcode[2:0]}`, pulse `clk`.
6. **Start**: `uio_in = 8'b101_1_00xx`, pulse `clk` — the core computes and the result is latched.

To read back the 32-bit result, set `uio_in[1:0] = READ_SEL` (00=bits 7:0, 01=bits 15:8, 10=bits 23:16, 11=bits 31:24, no LD/clock needed — it's combinational) and read `uo_out` for each byte. Check `uio_out[3]` (Valid) and `uio_out[2]` (Cout) for the status flags.

Example — 16-bit RCA, A=0x1234, B=0x0001, Cin=0: load A_LOW=0x34, A_HIGH=0x12, B_LOW=0x01, B_HIGH=0x00, control byte `{Cin=0, sub_op=00, opcode=000} = 8'h00`, then START. Expected `Result` = 0x00001235 (RCA opcode is zero-extended to 32 bits), `Cout=0`, `Valid=1`.

External hardware

None — this project has no external hardware dependencies.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A[0]	Result[0]	B[0]

#	Input	Output	Bidirectional
1	A[1]	Result[1]	B[1]
2	A[2]	Result[2]	B[2]
3	A[3]	Result[3]	B[3]
4	A[4]	Result[4]	B[4]
5	A[5]	Result[5]	B[5]
6	Cin	Result[6]	Opcode[1]
7	Opcode[0]	Result[7]	Opcode[2]

Preinception: Simple Compute Accelerator

by **Aakash KT**

0034 | 50 MHz | HDL Project

github.com/AakashKT/preinception-ihp26b-tapeout

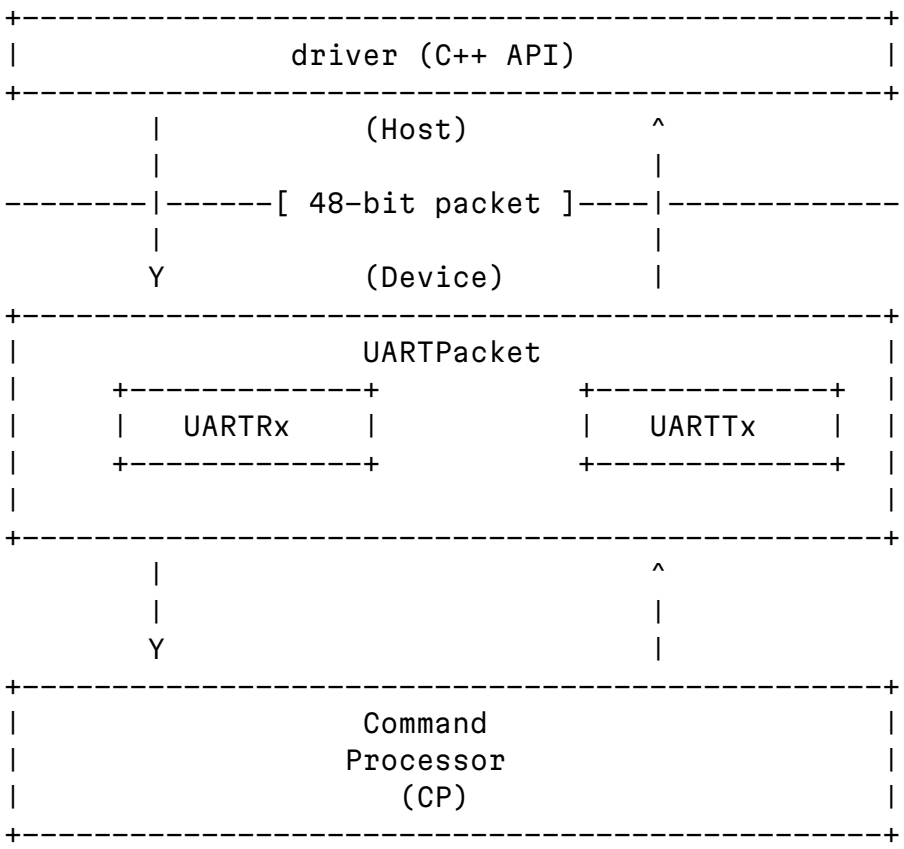
"Simple compute accelerator, connected via USB and driven with a C++ API and code on the host PC."

How it works

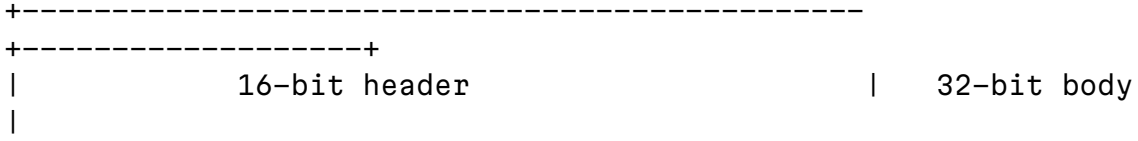
The C++ driver API, build instructions and hardware models for "Preinception" can be found in the PRESM repository.

Architecture

All commands are sent via the C++ driver.



Host-Device packet structure (48-bit packet)



```
++-----------------------------------------------
+------------------+
|| 4-bit     | 4-bit    | 4-bit    | 4-bit       ||   32-bits data
||
|| Unique ID | Type     | Command  | Sub-command ||
||
++-----------------------------------------------
+---------------+
```

How to test

Prerequisits

- A Linux PC with a spare USB port
- The demo board from Tiny Tapeout

Notes

- The code is officially tested on Ubuntu
- You can also use Ubuntu under WSL.
- With WSL however, the underlying port to which the demo board is connected will have to be exposed to WSL

Clone PRESM and navigate to the cloned directory.

Build the Tiny Tapeout config for Preinception:

```
python scripts/build.py --config hw_configs/preinception/
serial_tapeout_tt_ihp26b.json
```

Next, we need to connect the chip to the PC.

The chip can be connected in two ways:

- Connect the demo board to the PC with USB.
- With an exteral UART module attached to the demo board. The UART module is connected to the PC with USB.

As of now, not sure which one will work, so give both a go when chips become available.

You may need to run the following command to activate the FTDI drivers:

```
sudo modprobe ftdi_sio
```

Direct Demo Board Connection

This is straightforward. Testing is done by running verification apps:

```
python scripts/verify.py --config hw_configs/preinception/
serial_tapeout_tt_ihp26b.json
```

You should see an output like so:

```
++ Copying files from /home/aakashkt/presm/build/verification/ to
verify_runs/00491
```

```
++ Copying file /home/aakashkt/presm/build/driver/matrix/
libmatrix_serial.so to verify_runs/00491
++ Copying file hw_configs/preinception/
serial_fpga_tangnano20k.json to verify_runs/00491/hw_config.json
++ ======================
++ PRESM Execution Begin
++ ======================
++ Changing working directory to: /home/aakashkt/presm/
verify_runs/00491
++ Executing: ./verification sanity device
++ Environment:
{'LD_PRELOAD': 'libmatrix_serial.so'}
++ Changing working directory to: /home/aakashkt/presm
++ ======================
++ PRESM Execution End
++ ======================

++ Changing working directory to: /home/aakashkt/presm/
verify_runs/00491
++ Executing: ./verification sanity host
++ Changing working directory to: /home/aakashkt/presm

++ Verification of "sanity" succeeded.

++ ======================
++ PRESM Execution Begin
++ ======================
++ Changing working directory to: /home/aakashkt/presm/
verify_runs/00491
++ Executing: ./verification addition device
++ Environment:
{'LD_PRELOAD': 'libmatrix_serial.so'}
++ Changing working directory to: /home/aakashkt/presm
++ ======================
++ PRESM Execution End
++ ======================

++ Changing working directory to: /home/aakashkt/presm/
verify_runs/00491
++ Executing: ./verification addition host
++ Changing working directory to: /home/aakashkt/presm

++ Verification of "addition" succeeded.

.....
.....
```

Via External UART module

On the demo board:

- Connect output pin 6 `uo[6]` (`PIN_HI` in datasheet) to input pin 0 `ui[0]` (`SEL_RX_TX` in datasheet).
  - This changes the UART RX and TX pins from the default to two other free pins
- Next, connect the external UART module
  - Connect TX to input pin 7 `ui[7]` (`RX_secondary` in datasheet)
  - Connect RX to input pin 7 `uo[7]` (`TX_secondary` in datasheet)

Now we can run the verification apps like above.

Using the chip with your own code

Make sure you have built Preinception with the tapeout configuration.

```
python scripts/build.py --config hw_configs/preinception/
serial_tapeout_tt_ihp26b.json
```

The build creates a package of the driver in `packages/`, which looks like:

```
├── packages/
│   ├── matrix/
│   │   ├── include/
│   │   │   ├── matrix.h # Driver API header
│   │   ├── lib/
│   │   │   ├── libmatrix_serial.so # Driver library
```

Lets compile a simple C++ code that uses the driver to add two numbers on preinception.

```
#include "include/matrix.h"
#include <iostream>

int main()
{
    mInit(); // Initialize the driver

    // Allocate memory for device, and initialize it
    MIntDeviceMemory a(-64);
    MIntDeviceMemory b(-80);
    MIntDeviceMemory c(10);

    mAdd(a, b, c); // Call add on 'a' and 'b', store in 'c'
    mSync(); // Wait for device to finish

    std::cout << a.getValue() << " + " << b.getValue() << " = " <<
c.getValue() << std::endl;

    mFree(); // Free driver resources
}
```

Compilation needs to link with the driver library:

```
g++ main.cpp -L./lib/ -lmatrix_serial -o main -Wl,-rpath,./lib
```

Running the program:

```
$ ./main
-64 + -80 = -144
```

Failure cases

- The driver currently assumes that the device shows up in one of these: `/dev/ttyUSB0` –> `/dev/ttyUSB4`
  - Essentially, it searches for and does a handshake protocol for devices from index 0 to 4.
  - If the device shows up in another name or after index 4, the driver will quit with an error.

External hardware

Possibly requires an external UART-USB module.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SEL_RX_TX	—	—
1	—	—	—
2	—	—	—
3	RX_primary	—	—
4	—	TX_primary	—
5	—	—	—
6	—	PIN_HI	—
7	RX_secondary	TX_secondary	—

AION

by **Filippo Quadri**

0042 25 MHz HDL Project

`github.com/filippoquadri/ttihp_aion`

"32-bit and 16-bit posit arithmetic unit behind a byte-wide register interface"

How it works

AION (*AI-Optimized Netlist-to-Layout*) is a posit arithmetic unit on IHP SG13G2. It adds, multiplies, compares and computes bitwise AND/OR/XOR on **Posit<32,2>** or **Posit<16,2>** numbers; the precision is chosen per command. Operands and results go through a small byte-wide register interface.

AION is a **proof of concept**: it tests whether AI can improve a circuit's performance by creating standard cells specific to the design, instead of using only the cells the PDK provides. **Every custom standard-cell layout on this chip was drawn by an AI agent and verified with open-source tools**. Besides the PDK's standard cells, the netlist uses two kinds of custom cell:

- **Mined cells**: pairs of PDK cells that occur often in the synthesized design, merged into single new standard cells.
- **Custom logic cells**: transmission-gate inverting 2:1 and 4:1 multiplexers and an inverting majority gate, which synthesis picks wherever they are smaller.

For each cell, an LLM agent wrote the program that draws the layout. Open-source tools alone decided whether it was accepted: DRC with Magic and KLayout, LVS with Netgen, parasitic extraction with Magic, timing characterization with ngspice, and a routability check with OpenROAD. The tile was then hardened with the open-source LibreLane flow (Yosys, OpenROAD, Magic, KLayout, Netgen). As a reference, the same design was also hardened with PDK cells only.

Results against the PDK-only version

Both versions use the same RTL, die, constraints (40 ns clock) and flow settings; only the cells differ.

Metric	PDK cells only	With AION cells	Change
Standard-cell area	121,741 µm²	111,546 µm²	−8.4%
Core utilization	46.9%	42.9%	−3.9 pts

Total power (typical corner, estimated)	129.1 mW	99.3 mW	−23.1%
Routed wirelength	406,637 µm	359,885 µm	−11.5%
Setup worst slack (slow corner)	+1.87 ns	+6.90 ns	+5.03 ns
Hold worst slack (fast corner)	+0.150 ns	+0.138 ns	−0.011 ns

Both versions are clean in DRC and LVS and have no setup or hold violations in any of the three corners. The tile fixes the die size, so the saved area shows up as lower utilization rather than a smaller chip. The gains come from both kinds of custom cell together.

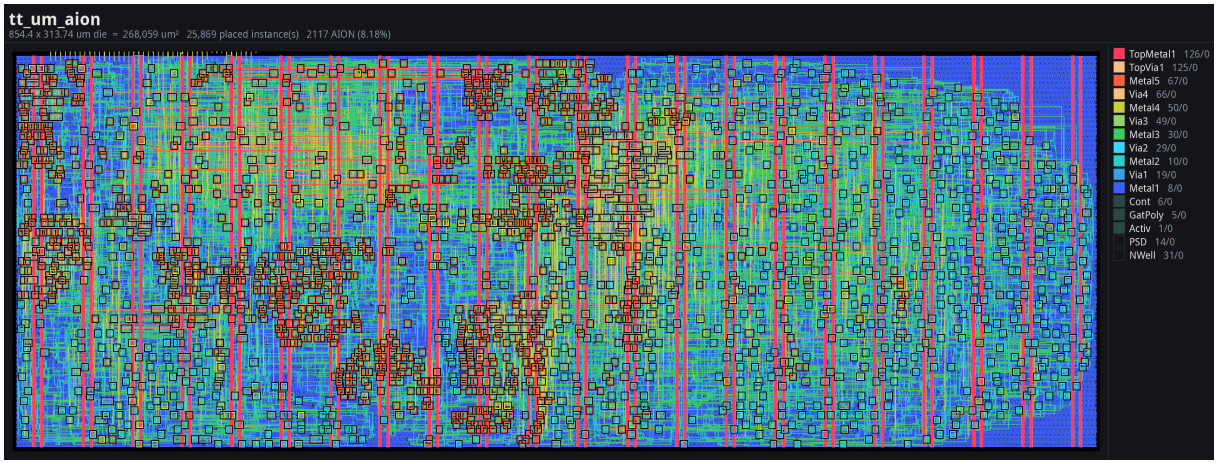


Figure 42.1: Hardened tile

The routed 4x2 tile, coloured by metal layer; every AION cell instance is ringed.

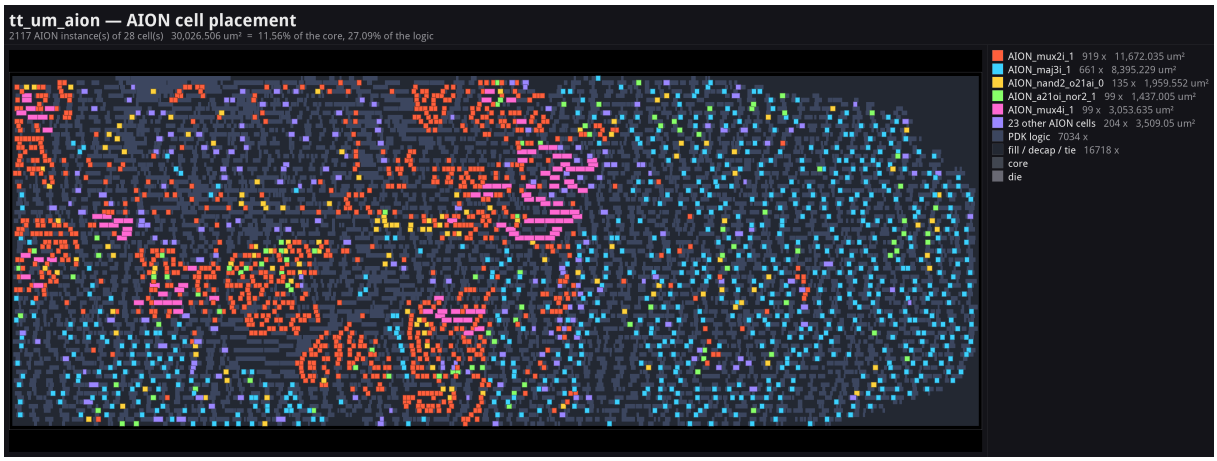


Figure 42.2: AION cell placement

Placement map: the most-used AION cells in their own colours, the rest grouped; PDK logic and fill in grey.

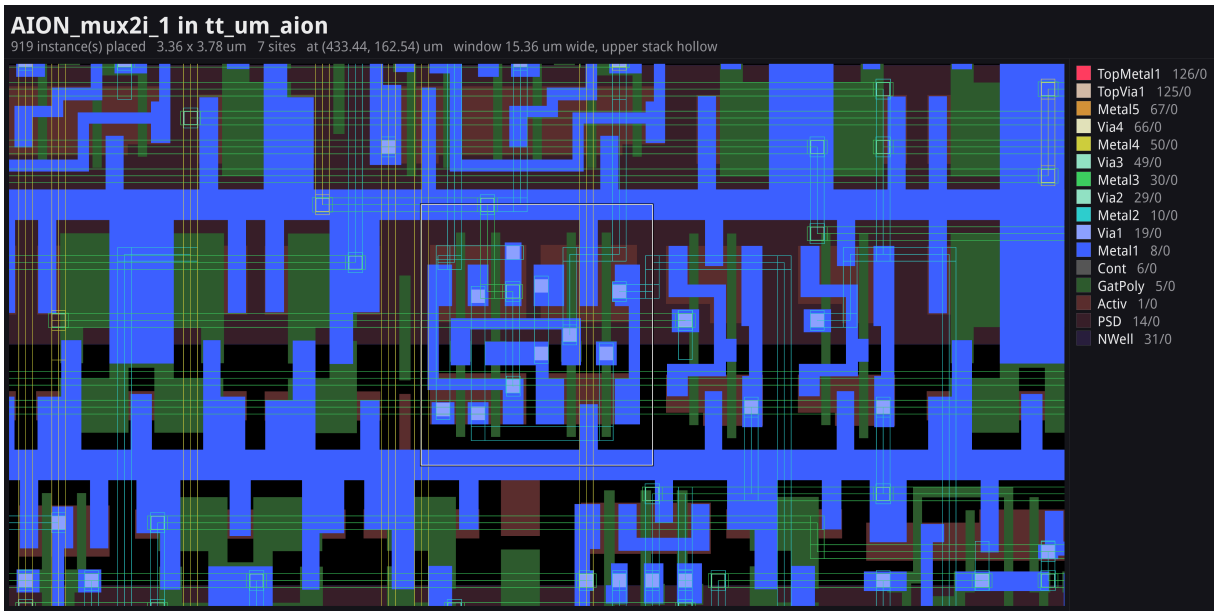


Figure 42.3: AION_mux2i_1 in place

An AI-drawn transmission-gate mux (`AION_mux2i_1`), abutted to its PDK neighbours.

Pins

Pin	Use
`ui_in[3:0]`	register address
`ui_in[7]`	`1` = write, `0` = read
`uio_in[7:0]`	write data (all bidirectional pins are inputs)
`uo_out[7:0]`	read data

Registers

Address	Register
`0x0`–`0x3`	operand A, low byte first
`0x4`–`0x7`	operand B, low byte first
`0x8`	control: `[3:0]` opcode, `[4]` precision (`0` = 32-bit, `1` = 16-bit), `[7]` start
`0x9`–`0xC`	result, low byte first
`0xD`	status: bit 0 = done

Opcodes: `0` add, `1` multiply, `2` equal, `3` less than, `4` AND, `5` OR, `6` XOR. Equal and less than return a flag in bit 0 of the result. For 16-bit commands, only the low two bytes of each operand and of the result are used.

How to test

1. Hold `rst_n` low for a few clock cycles, then release it. The design is timed for a 40 ns clock period (25 MHz); any slower clock works.
2. **Write** a register: set `ui_in = 0x80 | address` and `uio_in = data`, then apply one rising clock edge.
3. **Read** a register: set `ui_in = address` and read `uo_out` (no clock edge needed).
4. Write both operands, then write `control` with bit 7 set. Poll `status` until bit 0 is 1, then read the result. `done` clears when `control` is written and is set one clock edge later.

Example: 2.0 × 3.0 in Posit<32,2>.

Action	Value	Meaning
write `0x0`, `0x1`, `0x2`, `0x3`	`00`, `00`, `00`, `48`	A = `0x48000000` (2.0)
write `0x4`, `0x5`, `0x6`, `0x7`	`00`, `00`, `00`, `4C`	B = `0x4C000000` (3.0)
write `0x8`	`81`	multiply, 32-bit, start
read `0xD` until bit 0 = 1	`01`	done
read `0x9`, `0xA`, `0xB`, `0xC`	`00`, `00`, `00`, `54`	result = `0x54000000` (6.0)

The same product in Posit<16,2>: write `00`, `48` to `0x0`–`0x1` and `00`, `4C` to `0x4`–`0x5`, write `91` to `control`, and read `00`, `54` from `0x9`–`0xA`.

External hardware

None required. One 32-bit operation takes more than a dozen register writes and reads, so the practical way to drive it is a microcontroller, such as the one on the Tiny Tapeout demo board.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ADDR[0]	RDATA[0]	WDATA[0]
1	ADDR[1]	RDATA[1]	WDATA[1]
2	ADDR[2]	RDATA[2]	WDATA[2]
3	ADDR[3]	RDATA[3]	WDATA[3]
4	—	RDATA[4]	WDATA[4]
5	—	RDATA[5]	WDATA[5]
6	—	RDATA[6]	WDATA[6]
7	WRITE (1) / READ (0)	RDATA[7]	WDATA[7]

Scalable Banked L1 Memory Fabric for Edge AI

by **Vishnukumar Varatharaja Perumal**

0102 | 40 MHz | HDL Project

github.com/Vishnukumar4/tt-l1-fabric-ihp

"DFF-based silicon demonstrator of a scalable banked L1 weight memory fabric with DMA streaming and GEMM MAC, ported to IHP sg13g2"

How it works

This project is a silicon demonstrator of a scalable banked L1 weight memory fabric for edge AI accelerators, developed as part of an M.S. thesis at San Diego State University (SDSU IoT Laboratory).

The design contains four independent 4-word x 32-bit weight banks (DFF-based), a priority arbiter that grants DMA traffic priority over CPU access, a DMA streaming engine, and a weight-stationary GEMM MAC unit. The same arbiter, DMA, and GEMM RTL was verified at 33-bank scale with sky130 SRAM macros through a complete Cadence RTL-to-GDS flow (Xcelium, Genus, Innovus, Magic DRC = 0).

Writes assemble 32-bit words one byte at a time through the bidirectional bus; a write commits when byte 3 is written. Reads are combinational: select mode=read with a bank/word/byte address and the selected byte appears on the bidirectional bus.

How to test

All operations are driven through ui_in:

Bits	Field	Meaning
7:6	mode	00=write 01=read 10=DMA 11=GEMM
5:4	bank	bank select 0-3
3:2	byte	byte select within 32-bit word
1:0	word	word address within bank

Write a word: mode=00, drive each byte on uio (byte_sel 0,1,2,3). The write commits on byte 3. Read it back with mode=01: the selected byte appears on uio_out.

In DMA mode (10), byte_sel selects the DMA register: 0=source address, 1=transfer length, 2=start. The register is written with the value currently in

the 32-bit write accumulator (fill it first with a mode-00 write sequence). A transfer streams bank-0 words into the GEMM unit; dma_grant (uo_out[5]) asserts during the transfer and gemm_done_irq (uo_out[7]) fires when the MAC completes.

In GEMM mode (11), byte_sel=0 presents the accumulated MAC result: the low byte on uio_out and its low nibble on uo_out[3:0]. Example: weight 0x00030002 streamed 4 times gives 4x(2x3) = 0x18.

External hardware

None required. All testing is done through the Tiny Tapeout demo board I/O.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	mode[1] / bank_sel[1] / byte_sel[1] / word_sel[1]	gemm_result[0]	data[0]
1	mode[0] / bank_sel[0] / byte_sel[0] / word_sel[0]	gemm_result[1]	data[1]
2	bank_sel[1]	gemm_result[2]	data[2]
3	bank_sel[0]	gemm_result[3]	data[3]
4	byte_sel[1]	cpu_grant	data[4]
5	byte_sel[0]	dma_grant	data[5]
6	word_sel[1]	dma_done_irq	data[6]
7	word_sel[0]	gemm_done_irq	data[7]

IEEE Bandgap Reference

by **Paweł Pobłocki**

0104 Analog Project

github.com/magnetoField/ieee-sep-2026-analog-ihp

"IHP Analog Academy BG design example"

How it works

It is simple band gap reference. Design was adopted from https://github.com/IHP-GmbH/IHP-AnalogAcademy/blob/main/modules/module_1_bandgap_reference/part_3_layout/BGR_layout/final_bandgapreference/full_bandgap_layout_filled.gds. Main work was to insert it correctly into IHP PDK grid and take car of its integration into multi project wafer from Tiny Tapout.

How to test

Connect 5uA current source to Iout to bias opamp. Check if Vbg voltage is bandgap refernece (aorund 600mV). Check if bandgap reference voltage stays at 600mV over full temperature range −40 to 125 Celcius degries.

External hardware

Simple current mirror with potentiometer to set 5uA current should do the work.

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	0	iout
1	1	Vbg
2	2	Vout_amp

LDO

by **Youssef Emad**

0106 Analog Project

github.com/youssef0000oooo/ieee_LDO

"LDO circuit"

How it works

it has error amplifier and CS amplifier

How to test

see the output pin

External hardware

oscilliscope

Project Pinout

Digital Pins

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

Analog Pins

ua#	analog#	Description
0	3	Vout

iTALU: Interactive Testable Arithmetic Logic Unit

by **Your Name**

128 | 50 MHz | HDL Project

github.com/MaverickSemiUSA/tt_um_Pathan_Rehman_italu_dft

"8-bit ALU with DFT features including BIST and scan chain"

Project Overview

iTALU is an 8-bit Arithmetic Logic Unit (ALU) with comprehensive Design-for-Testability (DFT) features, designed for Tiny Tapeout using the IHP SG13G2 (130nm) technology. The project demonstrates industry-standard testing techniques — scan chain, LFSR/MISR-based BIST, and fault injection — with a simple serial user interface.

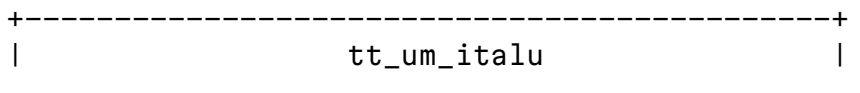
Key Features

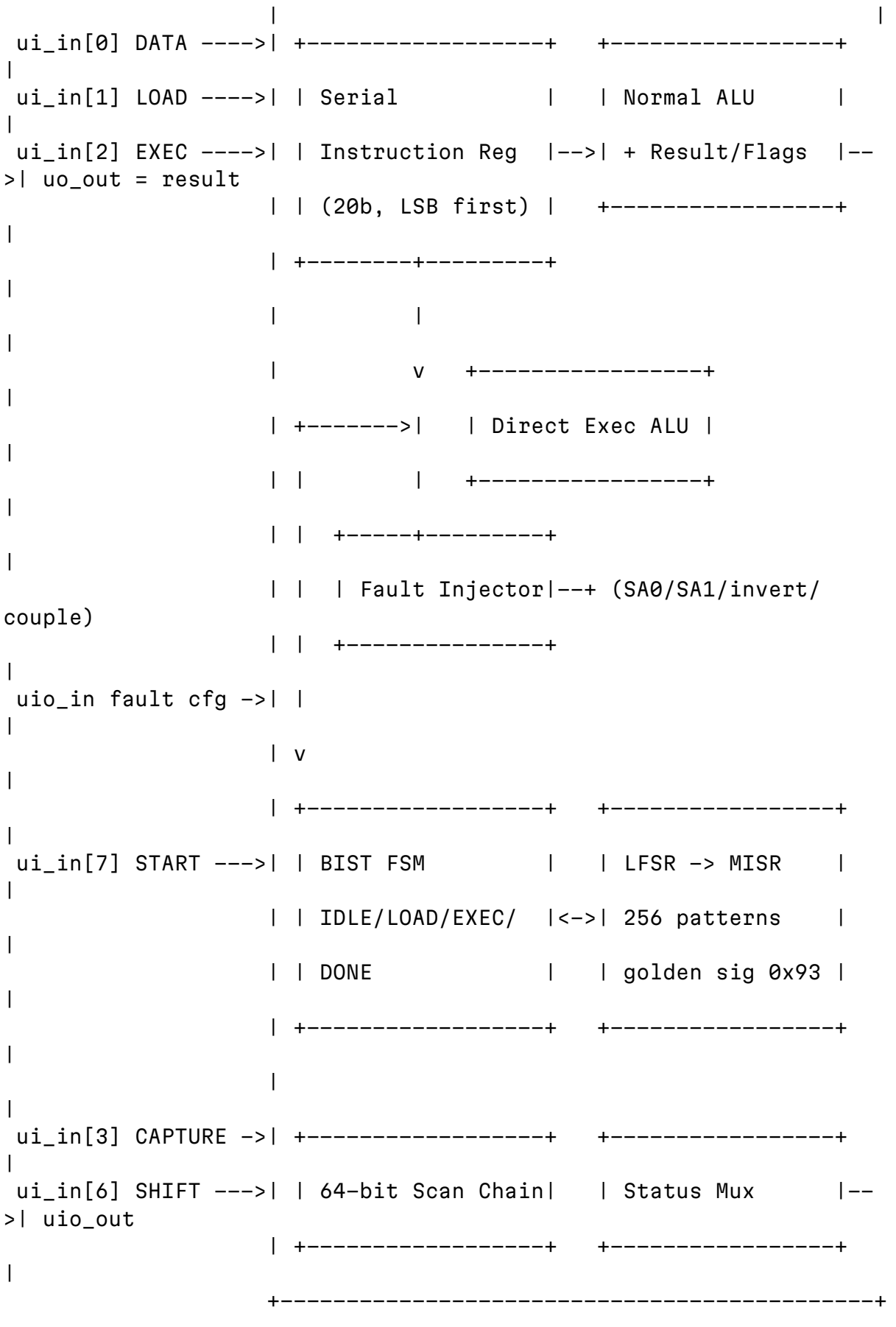
- 8-bit ALU with 16 operations (arithmetic, logic, shifts/rotates, compare, min/max, saturating arithmetic)
- 4 status flags (Zero, Carry, Negative, Overflow)
- 20-bit serial instruction interface (LSB first)
- 64-bit scan chain covering all internal state
- Built-In Self-Test (BIST) with 8-bit LFSR pattern generator and 8-bit MISR response compactor (256 patterns per run)
- User-controlled fault injection: stuck-at-0/1, inversion, coupling
- Status readback mux: flags/BIST status, MISR signature, fault counter, cycle counter

Specifications

Parameter	Value
Technology	IHP SG13G2 (130nm)
Tile Size	1x1
Clock Frequency	50 MHz (tested), no hard requirement
Top Module	`tt_um_italu`
I/O Pins	8 input, 8 output, 8 bidirectional

Architecture




ui_in[0] DATA
ui_in[1] LOAD
ui_in[2] EXEC
Serial Instruction Reg (20b, LSB first)
Normal ALU + Result/Flags
uo_out = result
Direct Exec ALU
Fault Injector
(SA0/SA1/invert/ couple)
uio_in fault cfg
BIST FSM IDLE/LOAD/EXEC/ DONE
LFSR -> MISR 256 patterns golden sig 0x93
ui_in[7] START
ui_in[3] CAPTURE
ui_in[6] SHIFT
64-bit Scan Chain
Status Mux
uio_out

ALU Operations

Code	Operation	Description
0x0	ADD	A + B with carry out
0x1	SUB	A - B (carry = no borrow)
0x2	AND	Bitwise AND
0x3	OR	Bitwise OR
0x4	XOR	Bitwise XOR
0x5	NOT	Bitwise NOT of A
0x6	SHL	Logical shift A left by 1
0x7	SHR	Logical shift A right by 1
0x8	SAR	Arithmetic shift A right by 1
0x9	ROL	Rotate A left by 1
0xA	ROR	Rotate A right by 1
0xB	SLT	Signed less-than: 1 if A < B else 0
0xC	MIN	Unsigned minimum of A and B
0xD	MAX	Unsigned maximum of A and B
0xE	SATADD	Saturating signed add (clamps at 0x7F / 0x80)
0xF	SATSUB	Saturating signed subtract (clamps at 0x7F / 0x80)

Status Flags

With `STATUS_SEL = 00`, `uio_out[3:0]` shows:

Flag	Bit Position	Description
Zero (Z)	bit 0	Result is 0x00
Carry (C)	bit 1	Carry out (ADD) or no borrow (SUB)
Negative (N)	bit 2	MSB of result is 1
Overflow (O)	bit 3	Signed arithmetic overflow

The same select also reports `[4]` = BIST done, `[5]` = BIST pass and `[6]` = scan out.

Design-for-Testability (DFT) Implementation

1. Serial Instruction Interface

A 20-bit shift register is filled LSB first while `LOAD_EN` is high:

```
[19:16] opcode
[15:8]  operand B
[7:0]   operand A
```

Pulsing `EXECUTE` latches the operands/op into the normal ALU registers and stores the (possibly fault-injected) result and flags.

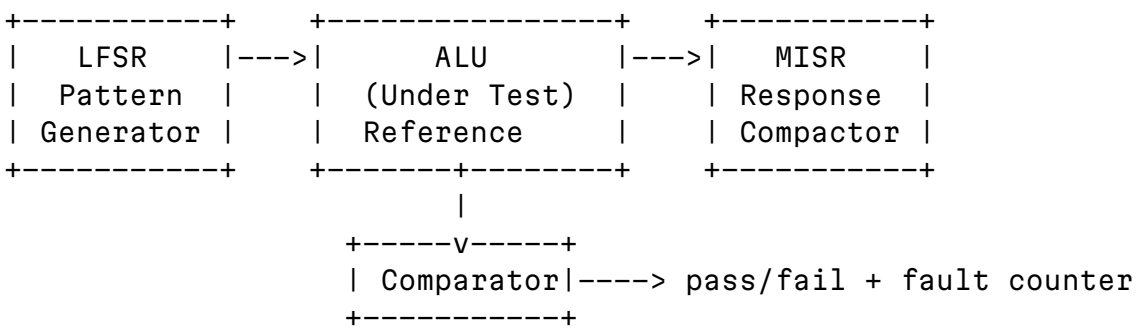
2. Scan Chain

A single 64-bit chain captures all internal state:

```
operand_a(8) -> operand_b(8) -> operation(4) -> alu_result(8) ->
flags Z/C/N/O(4) -> misr(8) -> bist_done(1) -> test_pass(1) ->
fault_counter(8) -> cycle_counter(8) -> padding(6)
```

- `SCAN_CAPTURE` (`ui_in[3]`) loads the chain from live state in one clock
- `SCAN_SHIFT` (`ui_in[6]`) shifts one bit per clock onto `SCAN_OUT` (`uio_out[6]`), LSB first

3. Built-In Self-Test (BIST)



- 8-bit LFSR generates operand A; operand B is a nibble-swap of A; the opcode comes from the low nibble of the LFSR value
- 256 patterns run automatically (IDLE → LOAD → EXEC → DONE FSM)
- Responses are compacted into an 8-bit MISR
- Fault-free golden MISR signature: **0x93**
- On completion `BIST_DONE` (`uio_out[4]`) asserts and **stays asserted** until the next `BIST_START`, so it can be polled reliably
- `TEST_PASS` (`uio_out[5]`) is high on a fault-free run, low when any mismatch or active fault injection was observed
- Detected faults increment an 8-bit fault counter (`STATUS_SEL = 10`)

4. Fault Injection

Configured via `uio_in` while the status output is driven:

Field	Pins	Values
Enable	`uio_in[0]`	1 = inject
Type	`uio_in[2:1]`	00 = stuck-at-0, 01 = stuck-at-1, 10 = inversion, 11 = coupling

Bit	`uio_in[5:3]`	Result bit 0–7 the fault applies to

The injected fault affects the normal ALU path, the serial-execution path and the BIST comparison simultaneously — so a faulted chip fails its own self-test, which is the core demonstration of the DFT flow.

5. Status Readback Multiplexer

`STATUS_SEL` (`uio_in[7:6]`)	`uio_out` content
00	Flags + BIST done/pass + scan out
01	MISR signature
10	Fault counter
11	Free-running cycle counter

Pin Configuration

Input Pins (ui_in)

Pin	Name	Description
ui_in[0]	DATA_IN	Serial data input for instruction loading
ui_in[1]	LOAD_EN	Shift one bit into the instruction register
ui_in[2]	EXECUTE	Execute the loaded instruction
ui_in[3]	SCAN_CAPTURE	Capture state into the scan chain
ui_in[4]	—	Unused
ui_in[5]	—	Unused
ui_in[6]	SCAN_SHIFT	Shift the scan chain one bit per clock
ui_in[7]	BIST_START	Start a BIST run

Output Pins (uo_out)

Pin	Name	Description
uo_out[7:0]	RESULT	8-bit ALU result

Bidirectional Pins (uio)

Inputs (fault configuration / status select):

Pin	Name	Description
uio[0]	FAULT_ENABLE	Fault injection enable
uio[2:1]	FAULT_TYPE	Fault type selection
uio[5:3]	FAULT_BIT	Faulted result bit

uio[7:6]	STATUS_SEL	Status output select

Outputs (status, selected by STATUS_SEL):

Pin	Select 00	Select 01	Select 10	Select 11
uio[7:0]	Flags/BIST/scan-out	MISR	Fault counter	Cycle counter

How to Test

RTL Simulation

Requires cocotb and Icarus Verilog:

```
cd test
make -B
```

This runs the full 11-test regression against `src/project.v`, producing `results.xml` and an FST waveform. See test/README.md for gate-level simulation and waveform viewing.

Regression Coverage

#	Test	What it verifies
1	`test_add`	Basic ADD: 0x0F + 0x03 = 0x12
2	`test_sub`	Basic SUB: 0x0A - 0x05 = 0x05
3	`test_all_alu_operations`	All 16 opcodes vs a Python reference model
4	`test_alu_flags`	Zero/Carry/Negative/Overflow flag generation
5	`test_normal_fault_injection`	Injected fault changes the ALU result
6	`test_bist_pass`	Fault-free BIST completes, passes, MISR = 0x93
7	`test_bist_fault_detection`	All 4 fault types detected by the BIST
8	`test_fault_counter`	Fault counter increments on detected faults
9	`test_scan_chain`	Capture + shift returns operands correctly
10	`test_cycle_counter`	Cycle counter advances with the clock
11	`test_complete_system`	End-to-end: ALU, clean BIST, fault detection

Status: **11/11 passing**, GDS hardened successfully with LibreLane.

Manual Testing on Hardware

Basic operation:

1. Reset the chip (`rst_n = 0`, then release).
2. Shift a 20-bit instruction in LSB first via `DATA_IN`/`LOAD_EN`.
3. Pulse `EXECUTE`; read the result on `uo_out` and the flags on `uio_out`.

Self-test:

1. Pulse `BIST_START`.
2. Poll `uio_out` until bit 4 (BIST done) is high.
3. Check bit 5 (pass/fail) and optionally read the MISR via `STATUS_SEL = 01`.

Fault demonstration:

1. Set `FAULT_ENABLE` with a type/bit combination on `uio_in`.
2. Re-run the BIST; `TEST_PASS` now goes low and the fault counter (`STATUS_SEL = 10`) increments.

Scan:

1. Pulse `SCAN_CAPTURE`, then clock `SCAN_SHIFT` 64 times reading `SCAN_OUT`.

External Hardware

No external hardware is strictly required — everything can be driven by a microcontroller or FPGA over the serial interface. For standalone bench use: push buttons/DIP switches for `ui_in`, LEDs or a logic analyzer on `uio_out`, and an LED bus or MCU reading `uo_out`.

License

This project is licensed under Apache-2.0.

Acknowledgments

- Tiny Tapeout for providing the platform
- IHP for the SG13G2 PDK
- Open source EDA community

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA_IN	RESULT_0	ZERO_FLAG
1	LOAD_EN	RESULT_1	CARRY_FLAG
2	EXECUTE	RESULT_2	NEG_FLAG

#	Input	Output	Bidirectional
3	—	RESULT_3	OVF_FLAG
4	—	RESULT_4	BIST_DONE
5	—	RESULT_5	TEST_PASS
6	SCAN_EN	RESULT_6	SCAN_OUT
7	BIST_START	RESULT_7	—

VGA Flower

by **Laura**, **Matt** & **Claude**

`0130` `25.2 MHz` `HDL Project`

`github.com/mattvenn/laura-asic-flower`

> "*Races the beam to draw a flower on a VGA display, no frame buffer*"

How it works

Draws a field of stick flowers scrolling slowly across a VGA display. It races the beam and holds no frame buffer at all: every pixel's colour is computed combinationally from the current scan position (`hpos`, `vpos`), which `hvsync_generator.v` (vendored unmodified from vga-playground.com) counts up at 640x480@60Hz.

The screen is three horizontal zones. The top quarter is sky, split into two shades of blue, with a yellow sun (a circle, same octagon distance approximation described below) that scrolls right and wraps back to the left edge when it goes off screen. The bottom quarter is grass, also split into two shades of green – the lighter one matches the flower stems. The middle half is white and holds the flowers.

Each of the 20 on-screen flower slots (640/32px) is a green stem of random length topped with a head of 4 lines through a point – 8 ray-ends, like a `*` – in a random colour from a small fixed palette. "Random" isn't a stored table or a stepped LFSR: `world_pix = scroll_int + hpos` gives each screen column an absolute coordinate in an infinite scrolling world, `world_pix >> 5` gives which 32px flower slot it's in, and a cheap xorshift-style hash (`mix16()`: constant shifts and XORs only, no multiplier) turns that slot index straight into a (length, colour) pair. A flower is a pure function of its own slot index, so it keeps the same appearance for as long as it's on screen, and new (but consistent) flowers appear automatically as new slots scroll in from the right – no state is stored for the visible flowers themselves, only the scroll position.

The flowers scroll left; the sun scrolls right at half that speed. Nothing in the design uses a general (variable x variable) multiplier: shapes are built from comparisons, adds/subtracts, and the same max+min/2 "octagon distance" circle approximation vga-playground's `rings` example uses.

How to test

Plug a Tiny VGA Pmod into the output header and a VGA monitor into the Pmod. Provide a clean 25.2MHz clock on `clk` and release `rst_n`. The field of

flowers and the sun should appear immediately; the flowers drift left and the sun drifts right (wrapping at the screen edge), both continuously.

External hardware

Tiny VGA Pmod driving a VGA monitor.

Project Pinout

Digital Pins

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

4-Bit Charge-Redistribution SAR ADC Controller

by **Josue Olivos**

0132 50 MHz HDL Project

`github.com/mariorp206/SAR_ADC_Controller`

> *"Three selectable digital controllers for an external 4-bit charge-redistribution SAR ADC: clean, manually enabled Trojan, and automatically triggered Trojan designs."*

How it works

This project contains three selectable 4-bit SAR ADC controllers:

- `00` – Clean controller
- `01` – Manual-Trojan controller
- `10` – Automatic-Trojan controller
- `11` – Clean/default

The controllers share a clock divider and comparator synchronizer. The selected controller drives an external capacitor DAC, sample switch, and comparator.

During each conversion, the SAR controller samples the input, tests each bit from MSB to LSB, reads the comparator, and stores the completed 4-bit result.

The live DAC-control code is available on `uo_out[3:0]`, while the stable final ADC result is available on `uio_out[3:0]`.

The manual Trojan is enabled with `ui_in[3]`. The automatic Trojan activates internally for 50 conversions after every 450 normal conversions. During an infected period, the physical DAC-control outputs are inverted.

Pin mapping

Inputs

- `ui_in[0]` – Comparator output
- `ui_in[2:1]` – Design selector
- `ui_in[3]` – Manual Trojan enable

Outputs

- `uo_out[3:0]` – Live DAC-control bits
- `uo_out[4]` – Sample-switch control
- `uo_out[7:5]` – FSM debug state

- `uio_out[3:0]` – Stable final ADC result

`uio_oe[3:0] = 1111`, configuring the lower four bidirectional pins as outputs.

How to test

Connect the external comparator to `ui_in[0]`, the DAC-control outputs to the capacitor switches, and `uo_out[4]` to the sample switch.

Select the desired controller using `ui_in[2:1]`, apply reset, and provide a known analog input and reference voltage.

Read the completed conversion from:

`uio_out[3:0]`

For the clean controller, the expected 4-bit result is approximately:

```
ADC code ≈ (VIN / VREF) × 15
```

To test the manual Trojan, select `01` and control it with `ui_in[3]`.

To test the automatic Trojan, select `10` and observe the outputs over multiple conversions. A logic analyzer or oscilloscope is recommended for observing the Trojan behavior.

External hardware

- Binary-weighted capacitor array
- CD4066 or similar analog switches
- External comparator such as LM393
- Sample switch
- Comparator pull-up resistor
- Reference-voltage source
- Analog input source
- Decoupling capacitors
- Breadboard or custom PCB
- Oscilloscope, logic analyzer, or microcontroller

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Comparator output	Selected live DAC bit 0 - LSB	Final ADC result bit 0 - LSB
1	Design select bit 0	Selected live DAC bit 1	Final ADC result bit 1
2	Design select bit 1	Selected live DAC bit 2	Final ADC result bit 2

#	Input	Output	Bidirectional
3	Manual Trojan enable	Selected live DAC bit 3 - MSB	Final ADC result bit 3 - MSB
4	—	Selected sample switch control	—
5	—	Selected FSM state bit 0	—
6	—	Selected FSM state bit 1	—
7	—	Selected FSM state bit 2	—

Laser Gyro Lock-in Readout Core

by **Marco Flückiger**

0134 HDL Project

github.com/marcomursik/Open-Loop-Lock-in-Demodulator

"Open-loop lock-in demodulator for the digital readout electronics of a single-axis laser gyroscope (fiber-optic interferometer)"

How it works

This design implements the digital readout electronics for a single-axis laser gyroscope (fiber-optic interferometer) using an open-loop lock-in demodulator architecture.

A configurable square-wave modulator (`mod_ref_gen`) generates the phase modulation reference signal, which drives an external phase modulator via `uo_out[0]`. The signal toggles every `period` clock cycles, so the full square-wave period is `2 × period`. `uo_out[1]` (`window_done`) emits a one-clock pulse every `period` cycles, i.e. once per half-cycle of the modulation reference.

The returning interferometer signal is digitized by an external 1-bit Delta-Sigma ADC and fed into `ui_in[0]`. The lock-in demodulator (`lockin_demod`) correlates the incoming bitstream with the modulation reference: it adds +1 per clock when the bit equals the reference and −1 otherwise. The result is accumulated over one modulation half-cycle (`period` clocks) and latched into `demod_out` at the end of each window.

Output scaling: the accumulator magnitude is bounded by `period` (max. 2^16−1). Since `demod_out` is a signed 16-bit value, periods up to 32767 clocks are transferred losslessly (no scaling shift); above that the output saturates at ±32767 instead of wrapping. Example: `period` = 1000 → full-scale reading ±1000 corresponds to a fully correlated/anticorrelated bitstream.

While `ui_in[1]` (Enable) is low, the accumulator and `demod_out` hold their values — useful for reading a single measurement over SPI without it changing mid-transfer.

An SPI slave interface (`spi_slave`, SPI mode 0, MSB first) provides register access:

- **Reg 0**: modulation period (read/write, default: 1000)
- **Reg 1**: demodulator output `demod_out` (read-only)

Protocol: 3-byte transfer. Command byte: bit 7 = read(1)/write(0), bit 0 = register address; then data-high and data-low bytes.

Examples:

- `0x00 0x00 0x14` → write Reg 0 = 20 (half-period = 20 clocks)
- `0x80 0x00 0x00` → read Reg 0 (MISO returns current period)
- `0x81 0x00 0x00` → read Reg 1 (MISO returns 16-bit `demod_out`, MSB first, during bytes 2 and 3)

How to test

1. Apply a clock to `clk` and assert `rst_n` (active-low reset).
2. Set `ui_in[1]` (Enable) high to activate the design.
3. Feed a 1-bit Delta-Sigma stream into `ui_in[0]`.
4. Observe the square-wave modulation reference on `uo_out[0]`.
5. The heartbeat LED on `uo_out[2]` toggles at approximately 1 Hz (at 10 MHz clock) when the design is running.
6. Use SPI (mode 0) to read the demodulated result:
   - Assert `CS_N` (active low)
   - Send 3 bytes: command `0x81` (read Reg 1), then two dummy bytes
   - Capture MISO during bytes 2 and 3 for the 16-bit `demod_out`
   - Optional: clear Enable after a `window_done` pulse to freeze the result before reading

External hardware

- External 1-bit Delta-Sigma ADC (photodiode frontend: TIA + comparator, or a ΔΣ modulator IC such as AD7401/AMC1306)
- External fiber-optic phase modulator driven by `uo_out[0]` — the low-cost option is a piezo (PZT) fiber stretcher, see hardware.md
- SPI master (microcontroller or FPGA) for register readout; example: `examples/host_readout.py`

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Delta-Sigma bitstream from photodiode ADC	Modulation reference (square wave)	SPI SCK
1	Enable (high = active)	Window done pulse (1x per modulation cycle)	SPI MOSI
2	—	Heartbeat LED (~1 Hz)	SPI MISO
3	—	—	SPI CS_N (active low)
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Timing-Prediction Test Vehicle

by **ECHO-HELLO-WORLD424**

`0136` `10 MHz` `HDL Project`

`github.com/ECHO-HELLO-WORLD424/tinyint-ttihp26b`

"*Half-cycle self-checking arithmetic timing-failure test vehicle with ring-oscillator delay canaries (pre/post-silicon timing-prediction study on IHP SG13G2)*"

Submitted normal operation: 10 MHz, 50% duty (100 ns period). The 10–50 MHz research sweep deliberately exceeds that timing-safe specification. The design is verified: local hardening, RTL, GL and precheck pass, and canonical CI is all green (GDS run `35034979531`); verification is tracked in the attempt log. Earlier physical results below remain historical evidence; fresh results are in `data/safe10/`.

How it works

A timing-prediction test vehicle for the IHP SG13G2 open PDK. It contains:

- **DUT**: a 16-bit ripple-carry adder split into four 4-bit segments. The carry between segments (and the final carry-out) passes through configurable inverter-pair delay banks (taps at 0/16/32/48 pairs), giving programmable critical-path lengths for a rising-launch/falling-capture timing experiment. Worst-case carry propagation is operand dependent, so the first-failure frequency depends on the applied pattern.
- **Oracle**: an independent combinational reference adder recomputes the expected result. It is compared at the next frame boundary, after operands have been held for 19 cycles; the deliberately delayed DUT is captured much earlier.
- **Canaries**: two ring-oscillator delay proxies - a generic inverter-line RO and a structure-matched RO (looping through delay-bank + full-adder segments like the DUT) - each with a 16-bit ripple edge counter over a configurable window ($2^{8..2}$14 cycles). The window is **one-shot per reset**: `win_done` is cleared only by `rst_n`, so the ring stops when the window closes and the counter holds its value until the next reset. Take one canary sample per reset rather than treating the count as continuous telemetry. The effective gate-open interval is `(window cycles - 3)` external clock periods - the three boot cycles before the configuration commits do not count - i.e. 25.3 us at 10 MHz and 5.06 us at 50 MHz for the 2^8 window.
- **Measurement**: one timed operation per 19-cycle frame, 16-bit error and op counters (both saturating), first-error DUT byte capture, serial byte readout with an auto-incrementing pointer, a freeze input, and

FORCE_ERR/FORCE_CAN DFT bits that make the error-accounting path itself testable before tapeout.

Config word (16-bit, driven during reset and held through the third rising edge after reset release, when it is committed): [1:0]/[3:2]/[5:4]/[7:6] = delay bank taps per segment, [9:8] = pattern (0=PRBS, 1=worst-case carry, 2=carry-free alternating, 3=static hold), [11:10] = canary select, [13:12] = window select, [14] = force canary mask, [15] = force DUT error.

The DUT launches operands on a rising edge and samples exactly once on the immediately following falling edge. **Clock HIGH time is the measurement aperture**: 10 ns at 50 MHz/50% duty, not the full 20 ns period. A failed first sample holds until comparison. Frequency sweeps must record actual high time and duty cycle.

FREEZE blocks new launches. An already launched operation completes its pending falling-edge capture even if FREEZE rises during the high phase. Keep clocking for two complete cycles after asserting FREEZE before reading. Resume never repeats a capture. Maintain the normal clock waveform through the pending falling edge.

`ui[7]` has **no on-chip synchronizer**, so the host must transition it as if it were synchronous: assert or release FREEZE during the clock **HIGH** phase. That leaves between half and one full clock period of settling ahead of the edge that samples it - 50-100 ns at 10 MHz, 10-20 ns at 50 MHz - comfortably beyond the 4 ns input-path budget `src/pnr.sdc` already assumes with `set_input_delay 4.0000` on `ui_in[7]`. A host that can only act in the LOW phase must still land the transition at least 20 ns (10 MHz) or 10 ns (50 MHz) before the next rising edge, and prove it on a scope. FREEZE must be generated in the chip's clock domain (host FPGA register, PIO, or equivalent hardware); an OS-scheduled software GPIO write does not bound its own jitter to that window and is not an acceptable source. A transition landing inside the setup/hold aperture of a rising edge can make `update_en` resolve differently per register, which corrupts the 19-cycle frame alignment and can count a fabricated error. The normative rule, its scope check, and the exclusion policy are in the post-silicon protocol.

How to test

1. Hold `rst_n` low and drive the config word on `ui[7:0]` (LSB) and `uio[7:0]` (MSB). Release `rst_n` during the clock LOW phase before the first counted rising edge. Keep the config word stable through the third rising edge after releasing `rst_n` (the on-chip boot counter commits it at that edge), then set `ui[7]` low - obeying the FREEZE transition rule above - and release your `uio` drivers **within the following clock period**. The chip takes over the `uio` bus on the fourth rising edge; holding your

drivers past it makes both sides drive the pads, causing bus contention. Configuration is already latched at that point; subsequent `uio` values do not update it.

2. Run the experiment at the target clock frequency/voltage/temperature for a known number of operations (19 cycles each). Start at 1.20 V/ ambient and a low clock frequency. Record the clock high time as well as frequency.
3. Set `ui[7]` high during the clock HIGH phase (the FREEZE transition rule above), allow two complete clocks for pending capture and ripple settling, then read the 16 status bytes: `uio[7:0]` is the data byte selected by `uo[3:0]` (auto-incrementing pointer). Byte map: 0-1 = DUT error count (saturating), 2-3/4-5 = generic/matched RO edge counts (16-bit, wrap mod 65536 – telemetry, not saturating), 6-7 = operation count (saturating; it counts *launches*, so completed comparisons are `max(ops_cnt - 1, 0)` and using the raw count as the denominator biases every error rate low; once saturated, the true count is `>= 65535` and no longer recoverable from this byte), 8 = segment-tap echo `{seg3, seg2, seg1, seg0}`; 9 = status flags `{1, mat_ro_dead, gen_ro_dead, err_seen, can_sel[1:0], win_sel[1:0]}` (`can_sel` is bits 3:2, `win_sel` is bits 1:0, and bit 7 is 1); 10 = low 8 bits of the first failed DUT result capture; 11-15 = 0. While frozen, `uo[7:4]` = `{frame_strobe, mat_ro_dead, gen_ro_dead, dut_err}`.
4. Repeat across frequency/voltage/temperature; compare `f(error)` contours against the RO telemetry and static timing analysis.

External hardware

A controller that can drive the clock from 1-50 MHz, hold/step the config pins, and read the status pins (e.g. the Tiny Tapeout demo board or an FPGA host). For the PVT sweep: a variable core-voltage supply (verified accessible per Tiny Tapeout IHP powering rules) and a temperature chamber or controlled hot/ cold plate. An external low-jitter clock source with characterized duty cycle is recommended near the timing boundary. Elevated temperature is optional and subject to fixture/assembly limits; 125 °C is a library corner, not a required test condition or a certified board rating.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	cfg_seg0[0] (reset) / unused (run)	ro_ptr[0]	cfg[8] (reset in) / status[0] (run out)
1	cfg_seg0[1] (reset) / unused (run)	ro_ptr[1]	cfg[9] (reset in) / status[1] (run out)

#	Input	Output	Bidirectional
2	cfg_seg1[0] (reset) / unused (run)	ro_ptr[2]	cfg[10] (reset in) / status[2] (run out)
3	cfg_seg1[1] (reset) / unused (run)	ro_ptr[3]	cfg[11] (reset in) / status[3] (run out)
4	cfg_seg2[0] (reset) / unused (run)	dut_err (live)	cfg[12] (reset in) / status[4] (run out)
5	cfg_seg2[1] (reset) / unused (run)	gen_ro_dead	cfg[13] (reset in) / status[5] (run out)
6	cfg_seg3[0] (reset) / unused (run)	mat_ro_dead	cfg[14] (reset in) / status[6] (run out)
7	cfg_seg3[1] (reset) / FREEZE (run, active high)	frame_strobe	cfg[15] (reset in) / status[7] (run out)

Four-Metric VGA Nearest-Prototype Visualizer

by **zanderivo**

138 | 25.175 MHz | HDL Project

`github.com/zanderivo/tt-submission-cohort3`

"640x480 VGA nearest-prototype visualizer with four selectable distance metrics, manual prototype moves, and online training."

How it works

This project is a 1x2 Tiny Tapeout VGA nearest-prototype visualizer. It uses a 25.175 MHz pixel clock to generate a 640×480 display. Four colored prototypes divide the left side of the screen into nearest-prototype regions, while the right side shows the active mode and training status.

The input metric selects how distance is measured:

`ui_in[1:0]`	Metric
`00`	Manhattan / L1
`01`	Chebyshev / L-infinity
`10`	Octagonal L2 approximation
`11`	Binary-coordinate Hamming distance

White crosshairs show the prototype positions. Equal distances choose the lower-numbered prototype. The selected metric and training request update at a frame boundary so the displayed frame stays coherent.

Controls

Pin	Function
`ui_in[1:0]`	Select the distance metric
`ui_in[2]`	Enable online training
`ui_in[4:3]`	Select prototype 0–3
`ui_in[5]`	Select axis: `0` X, `1` Y
`ui_in[6]`	Select direction: `0` decrement, `1` increment
`ui_in[7]`	Step strobe for a manual move

A rising edge on Step moves the selected coordinate by one logical unit at the next frame boundary. Online training uses an internal pseudo-random

sample to slowly move the nearest prototype. Keep selector inputs stable around a Step pulse.

How to test

1. Connect a TinyVGA-compatible VGA PMOD to `uo_out` and a monitor that supports 640×480.
2. Supply a stable 25.175 MHz clock and release reset.
3. Use `ui_in[1:0]` to select a metric and observe the region pattern change on the next frame.
4. Select a prototype, axis, and direction; then pulse `ui_in[7]` to move its crosshair.
5. Set `ui_in[2]` high to enable online training.

External hardware

- TinyVGA-compatible VGA PMOD or resistor-DAC interface
- VGA monitor capable of 640×480
- 25.175 MHz clock source
- Switches or buttons for `ui_in`; use a debounced source for Step

The VGA outputs are `R1,G1,B1,VSync_n,R0,G0,B0,HSync_n` on `uo_out[0:7]`. The bidirectional pins are unused.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Metric request bit 0 (00 L1, 01 Linf, 10 octagonal L2, 11 Hamming)	TinyVGA red MSB (R1)	Unused (input only)
1	Metric request bit 1	TinyVGA green MSB (G1)	Unused (input only)
2	Training enable request	TinyVGA blue MSB (B1)	Unused (input only)
3	Manual centroid select bit 0	TinyVGA active-low vertical sync	Unused (input only)
4	Manual centroid select bit 1	TinyVGA red LSB (R0)	Unused (input only)
5	Manual axis (0 X, 1 Y)	TinyVGA green LSB (G0)	Unused (input only)
6	Manual direction (0 decrement, 1 increment)	TinyVGA blue LSB (B0)	Unused (input only)

#	Input	Output	Bidirectional
7	Manual step transaction strobe	TinyVGA active-low horizontal sync	Unused (input only)

SENT to I2C bridge

by **Enzo Nappi**

0160 | 10 MHz | HDL Project

`github.com/enzonappi/tt_um_enzonappi_sent_i2c`

"Full-duplex SAE J2716 SENT decoder + encoder bridged to an I2C register-map slave"

This chip is a full-duplex bridge between a single-channel **SENT** (SAE J2716) automotive sensor link and **I2C**.

Receive path: a SENT decoder listens on `ui[0]`. It auto-calibrates the tick length from each frame's sync pulse (no fixed tick assumed), then decodes a status nibble, 1-6 data nibbles (configurable), and a CRC-4 nibble (poly $x^{4+x}3+1$, seed 5, per SAE J2716). An optional pause pulse after the CRC can be skipped so it is never mistaken for the next frame's sync.

Transmit path: an independent SENT encoder runs concurrently on `uo[4]`, generating a second, runtime-configurable SENT frame (own tick length, nibble count, optional padding to a fixed frame duration) completely decoupled from the receiver — true full duplex, not time-multiplexed.

Both sides are exposed through a single **I2C slave** (address `0x50`, no clock stretching) with an auto-incrementing register pointer: write one byte to set the pointer, then read (or write, for writable registers) as many bytes as needed, wrapping at the top of the map. STATUS is clear-on-read for its sticky error/new-data bits. The register map (20 registers, `0x00`-`0x13`) covers, in order: received STATUS/STATUS_NIBBLE/DATA0-2/CRC/FRAME_COUNT and RX CONFIG (`0x00`-`0x07`); the TX payload/CONFIG/TICK/FRAME_DUR registers (`0x08`-`0x0F`); and two 16-bit registers, SYNC_MIN and SYNC_MAX (`0x10`-`0x13`), that bound how long a pulse must be to be accepted as a sync pulse — retunable at runtime for a different sensor tick length, with no recompile needed.

A shadow snapshot publishes STATUS_NIBBLE/DATA/CRC only when no I2C transaction is in progress, so a multi-byte read is never torn between two different SENT frames.

How to test

Wire a SENT source to `ui[0]` (a real sensor, or the transmit output of a second instance of this same chip looped back for a quick self-test) and an I2C master (address `0x50`, open-drain `uio[1]`=SDA / `uio[0]`=SCL, external pull-ups required — no clock stretching).

1. Write `0x07` (CONFIG) if you need to invert the input, enable the pause pulse, or change the expected data-nibble count from the 6-nibble default.
2. Point the register pointer at `0x00` and read: `STATUS`, `STATUS_NIBBLE`, `DATA0`, `DATA1`, `DATA2`, `CRC`, `FRAME_COUNT` — a 7-byte block read pulls one full decoded frame. `STATUS` bit0 (`new_data`) tells you a frame has arrived since the last time you read it.
3. To transmit, write the payload to `TX_STATUS_NIBBLE`/`TX_DATA0–2` (`0x08`-`0x0B`), then write `TX_CONFIG` (`0x0C`) with bit0 (`tx_enable`) set — frames start going out on `uo[4]` immediately, back-to-back.
4. If your sensor's tick length is outside the 1-6 us default window, write the new bounds (in raw clk cycles) to `SYNC_MIN_L/H`/`SYNC_MAX_L/H` (`0x10`-`0x13`) before expecting a lock.

The RTL testbench (`test/tb_sent_i2c.v`, Icarus Verilog) exercises the decoder, encoder, inverted polarity, the pause pulse, full-duplex RX-while-TX, and sync-mode frame padding end to end and is the fastest way to check a change hasn't broken anything.

External hardware

Any single-channel SENT sensor (or a second instance of this chip, transmit side, for a loopback bench test) on `ui[0]`, and an I2C master with two 4.7kΩ pull-ups to 3.3V on SCL/SDA. No other external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SENT signal in (receive)	new_data (STATUS bit0, live)	I2C SCL (input only, no clock stretching)
1	—	crc_error (STATUS bit1, live)	I2C SDA (open-drain)
2	—	sync_error (STATUS bit2, live)	—
3	—	frame_busy	—
4	—	SENT signal out (transmit)	—
5	—	tx_busy	—
6	—	—	—
7	—	—	—

FPU-130

by **Aarush** & **Yogith**

0161 | 50 MHz | HDL Project

github.com/140oo041/FPU

"a FPU for the 16bfloat format that takes commands over SPI"

How it works

FPU-130 is an SPI-controlled arithmetic coprocessor for the 16-bit `bfloat16` format. A `bfloat16` value contains one sign bit, eight exponent bits, and seven fraction bits. It is also the upper 16 bits of an IEEE-754 single-precision value; for example, `1.0`, `2.0`, and `3.0` are encoded as `0x3f80`, `0x4000`, and `0x4040` respectively.

The supported operations are:

Opcode	Operation	Description
`000`	ADD	`A + B`
`001`	SUB	`A - B`
`010`	MUL	`A * B`
`011`	DIV	`A / B`
`100`	NEG	`-A`
`101`	ABS	`abs(A)`
`110`	SLT	`1.0` when `A < B`, otherwise `0.0`
`111`	NOP	No arithmetic writeback

The SPI interface uses mode 0, active-low chip select, and MSB-first byte ordering. The command byte is arranged as follows:

Bits	Field
`[7:5]`	Opcode
`[4]`	Accumulator enable
`[3]`	Arity: `0` for unary, `1` for binary
`[2:0]`	Three-bit request tag

A binary frame contains six bytes:

`[command] [A high] [A low] [B high] [B low] [CRC-8]`

A unary frame contains four bytes:

```
[command] [A high] [A low] [CRC-8]
```

The final byte is CRC-8/AUTOSAR, calculated over every preceding byte in the frame with polynomial `0x2f`, initial value `0xff`, and final XOR value `0xff`.

The FPU response is 24 bits, MSB first:

```
[status] [result high] [result low]
```

The status byte contains the echoed tag in bits `[7:5]`, CRC/SPI error in bit 4, a valid marker in bit 3, underflow in bit 2, overflow in bit 1, and NaN in bit 0. SPI is full duplex, but an operation cannot be computed until its full request has arrived. Consequently, the bytes received while sending a request belong to the previous result. Clock the completed result out in a following transaction, or pipeline it with the next command.

How to test

Select the design, provide its system clock, pulse `rst_n` low, and then leave `rst_n` high. Connect an SPI controller as follows:

Signal	Tiny Tapeout pin	Direction relative to FPU-130
CS (active low)	`uio[0]`	Input
MOSI	`uio[1]`	Input
MISO	`uio[2]`	Output
SCLK	`uio[3]`	Input

Use SPI mode 0 and start with a low SPI clock frequency for bring-up. The SPI signals pass through synchronizers clocked by the project clock, so the project clock must remain active and substantially faster than SCLK.

As a known-good test, add `1.0` to `2.0` with tag zero:

```
Request:  08 3f 80 40 00 34
```

Here `0x08` selects binary ADD, `0x3f80` is `1.0`, `0x4000` is `2.0`, and `0x34` is the frame CRC. After sending the request, deassert CS and allow the FPU time to finish. For initial testing, waiting at least 100 project-clock cycles is conservative. Assert CS again and provide 24 SCLK cycles. The expected response is:

```
Response: 08 40 40
```

`0x08` is a clean status byte with its valid marker set, and `0x4040` is the `bfloat16` representation of `3.0`. The first transaction after reset does not return the result being submitted; it is produced only after that request has been received and processed.

External hardware

Testing requires a Tiny Tapeout carrier or compatible FPGA setup, an SPI controller like ESP32. Find the code for ESP32 on https://github.com/140oo041/FPU-micro-processor

Known Defects

Sometimes there may be a 2 bit offset using this interface. The best way to interface is to schedule operations together, and then send one big SPI transfer. After receveing the data, and CS goes down, reset the board.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	CS SPI
1	—	—	MOSI SPI
2	—	—	MISO SPI
3	—	—	SCK SPI
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

UART-SPI-I2C Bridge

by **Catalin Lazar**

0162 | 10 MHz | HDL Project

`github.com/catalinlazar/tt_um_catalinlazar_uart_spi_i2c_bridge`

> "*UART-controlled protocol bridge with an SPI master (4x CS, configurable CPOL/CPHA) and an I2C master (clock stretching), driven by a command interpreter with FIFO buffering and a memory-mapped register file. Fully testable standalone via UART commands, with a built-in loopback self-test mode.*"

A UART command interpreter drives an SPI master (4x chip-select, configurable CPOL/CPHA) and an I2C master (clock stretching, 7-bit addressing) over a shared 8-bit memory-mapped register file. UART RX/TX are FIFO-buffered so bursts of commands or data don't stall the host. See the top-level `README.md` for the full opcode table, register map, pinout, and the "Design history" section on how this ended up on a 1x2 tile.

How to test

Hold `ui_in[2]` (loopback_test_en) high and issue an `SPI_XFER` (opcode `0x03`) command over UART at 9600 baud (default): the chip should echo back exactly the bytes you sent, since MOSI is looped to MISO internally. Issuing `I2C_WRITE` in the same loopback mode should reliably reply with a NACK status byte (no real I2C slave is present), which exercises the START/ADDRESS/STOP sequencing. Connect a real SPI or I2C peripheral (with pull-ups on `uio[0]`/`uio[1]` for I2C) and drop `loopback_test_en` low for full end-to-end testing.

See `test/test.py` for an automated cocotb testbench covering register read/write, SPI loopback, I2C NACK detection, and STATUS/RESET.

External hardware

- Optional: any SPI peripheral, wired to `uo_out[1]` (SCLK), `uo_out[2]` (MOSI), `ui_in[1]` (MISO), and one of `uo_out[6:3]` (CS).
- Optional: any I2C peripheral on `uio[0]`/`uio[1]`, with external pull-up resistors (required — the design only drives these lines open-drain low).
- A USB-UART adapter on `ui_in[0]`/`uo_out[0]` for sending commands.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	uart_rx	uart_tx	i2c_scl
1	spi_miso	spi_sclk	i2c_sda
2	loopback_test_en	spi_mosi	—
3	—	spi_cs_n[0]	—
4	—	spi_cs_n[1]	—
5	—	spi_cs_n[2]	—
6	—	spi_cs_n[3]	—
7	—	heartbeat_led	—

ChipLab

by **Jörn Hoffmann**

0164 50 MHz HDL Project

`github.com/joernhoffmann/ChipLab`

"*Educational digital logic experiments*"

ChipLab is an educational digital logic chip for Tiny Tapeout IHP26b. Sections 1–9 are implemented, from basic logic to FSM-controlled datapaths and sound.

How it works

The design uses synthesizable SystemVerilog. `uio_in[5:0]` selects an experiment. `uio_in[7:6]` selects an operation where needed. All bidirectional pins are inputs (`uio_oe = 0`). `uio_out` is zero. For experiments 1 and 2, `ui_in[2:0]` provides C, B, A. Higher input bits are unused. Results appear on `uo_out[7:0]`. Section 2 uses the mappings below.

No.	Selection	Experiment
—	—	**1. Basic and Boolean Logic**
1	`0x01`	Basic gates
2	`0x02`	Boolean identities
—	—	—
—	—	**2. Data Selection and Coding**
3	`0x03`	Multiplexer
4	`0x04`	Demultiplexer
5	`0x05`	Binary decoder
6	`0x06`	Encoder
7	`0x07`	Priority encoder
8	`0x08`	Dual-priority encoder
9	`0x09`	BCD-to-7-segment decoder
10	`0x0A`	HEX-to-7-segment decoder
11	`0x0B`	Binary ↔ Gray code
12	`0x0C`	Parity generator and checker
13	`0x0D`	ROM
—	—	—
—	—	**3. Arithmetic and Data Operations**

14	0x0E	Half adder
15	0x0F	Full adder
16	0x10	4-bit adder
17	0x11	4-bit subtractor
18	0x12	Unsigned comparator
19	0x13	Signed comparator
20	0x14	Shifts
21	0x15	Rotation
22	0x16	ALU
—	—	—
—	—	**4. Storage Elements and Memory**
23	0x17	D latch
24	0x18	D flip-flop
25	0x19	T flip-flop
26	0x1A	JK flip-flop
27	0x1B	D latch and D flip-flop
28	0x1C	Synchronous and asynchronous reset
29	0x1D	Register with enable
30	0x1E	Register with load, hold, and clear
31	0x1F	4 × 4-bit read/write memory
32	0x20	Accumulator
33	0x21	4 × 4-bit FIFO
34	0x22	4 × 4-bit stack
—	—	—
—	—	**5. Shift Registers and Counters**
35	0x23	Shift register
36	0x24	Universal shift register
37	0x25	Binary counter
38	0x26	Up/down counter
39	0x27	Modulo-6 counter
40	0x28	BCD counter
41	0x29	Ring counter
42	0x2A	Johnson counter
43	0x2B	LFSR
—	—	—
—	—	**6. Input Synchronization and Timing**

44	`0x2C`	Edge detection
45	`0x2D`	Input synchronizer
46	`0x2E`	Push-button debouncer
47	`0x2F`	Clock enable divider
48	`0x30`	PWM
—	—	—
—	—	**7. Finite State Machines**
49	`0x31`	Moore release control
50	`0x32`	Mealy release control
51	`0x33`	Traffic light controller
52	`0x34`	Handshake controller
53	`0x35`	Parking lot occupancy counter
—	—	—
—	—	**8. FSM-Controlled Datapaths**
54	`0x36`	Sequential multiplier
55	`0x37`	Binarized neural network
—	—	—
—	—	**9. Sound**
56	`0x38`	Programmable sound generator
—	—	—
—	All other codes	All outputs zero

Experiments 1–22 are combinational. Experiments 23–56 store state and use `clk` or latch controls. `rst_n` resets the clocked storage elements. The two D latches have no reset. Allow signals to settle before sampling.

Output bit	Basic gates (`0x01`)	Boolean identities (`0x02`)
0	NOT A	NOT (A AND B)
1	NOT B	(NOT A) OR (NOT B)
2	A AND B	NOT (A OR B)
3	A OR B	(NOT A) AND (NOT B)
4	A NAND B	A AND (B OR C)
5	A NOR B	(A AND B) OR (A AND C)
6	A XOR B	A OR (A AND B)
7	A XNOR B	A

In the Boolean experiment, pairs (0, 1) and (2, 3) demonstrate De Morgan's laws, (4, 5) demonstrates distributivity, and (6, 7) demonstrates absorption. Each pair has the same settled output for every input combination. Synthesis

may merge equivalent logic. These experiments compare truth tables, not physical implementations or propagation delays.

Data selection and coding

Bit ranges below refer to `ui_in` and `uo_out`. Unlisted input bits are ignored. Unlisted output bits are zero.

Code	Input	Output
`0x03`	Data `[3:0]`, select `[5:4]`	Selected bit on `[0]`
`0x04`	Data `[0]`, select `[2:1]`	Data routed to one of `[3:0]`
`0x05`	Address `[2:0]`	One set bit on `[7:0]`
`0x06`	One-hot data `[7:0]`	Position `[2:0]`, valid `[3]`
`0x07`	Data `[7:0]`	Highest position `[2:0]`, valid `[3]`
`0x08`	Data `[7:0]`	Two positions and valid bits, see below
`0x09`	BCD digit `[3:0]`	Segments `[6:0]`
`0x0A`	HEX digit `[3:0]`	Segments `[6:0]`
`0x0B`	Value `[3:0]`, direction `[4]`	Converted value `[3:0]`
`0x0C`	Data `[6:0]`, received parity `[7]`	Generated parity `[0]`, error `[1]`
`0x0D`	Address `[2:0]`	ROM word `[7:0]`

- **Encoder:** exactly one set input bit is valid. Otherwise all outputs are zero.
- **Priority encoder:** bit 7 has highest priority. Zero input gives zero output.
- **7-segment:** active-high, bits 0–6 = a–g. Bit 7 is zero (decimal point off). BCD values 10–15 blank the display. HEX letters are A, b, C, d, E, F.
- **Gray:** direction 0 converts binary to Gray. Direction 1 converts Gray to binary.
- **Parity:** even parity. Error is 1 when data plus received parity has an odd number of set bits. It detects odd numbers of bit errors, not every possible error.
- **ROM:** addresses 0–7 store ASCII `ChipLab\0` (`43 68 69 70 4C 61 62 00` hex). Contents are fixed at synthesis. There is no write operation.

Dual-priority encoder (`0x08`)

All eight input bits are used. The circuit finds the highest set bit, clears it, and calls the same SystemVerilog function again to find the next match.

Output	Meaning
`uo_out[2:0]`	Highest position

`uo_out[3]`	First match valid
`uo_out[6:4]`	Second-highest position
`uo_out[7]`	Second match valid

Input `1011_0100` gives positions 7 and 5 (`uo_out = 0xDF`). Missing matches have position zero and a cleared valid bit. The function describes combinational logic. Its two calls do not take two clock cycles.

Arithmetic and data operations

For experiments 16–22, `A = ui_in[3:0]` and `B = ui_in[7:4]` unless noted.

Code	Result
`0x0E`	Sum `[0]`, carry `[1]`. Inputs are `ui_in[1:0]`
`0x0F`	Sum `[0]`, carry `[1]`. Carry-in is `ui_in[2]`
`0x10`	Sum `[3:0]`, carry `[4]`, signed overflow `[5]`
`0x11`	Difference `[3:0]`, no-borrow `[4]`, signed overflow `[5]`
`0x12`	Unsigned less `[0]`, equal `[1]`, greater `[2]`
`0x13`	Signed less `[0]`, equal `[1]`, greater `[2]`
`0x14`	Shifted value `[3:0]`
`0x15`	Rotated value `[3:0]`
`0x16`	Value `[3:0]`, carry `[4]`, overflow `[5]`, zero `[6]`, negative `[7]`

Shift and rotation use `ui_in[3:0]` as value and `ui_in[5:4]` as amount. For shifts, operation 0 is left, 1 is logical right, and 2 is arithmetic right. For rotation, operation bit 0 selects left or right. ALU operations are add, subtract, AND, and OR for operation values 0–3.

Storage elements and memory

Code	Input	Output
`0x17`	D `[0]`, gate `[1]`	Q `[0]`
`0x18`	D `[0]`	Q `[0]`
`0x19`	Toggle `[0]`	Q `[0]`
`0x1A`	J `[0]`, K `[1]`	Q `[0]`
`0x1B`	D `[0]`, latch gate `[1]`	Latch Q `[0]`, flip-flop Q `[1]`
`0x1C`	D `[0]`	Synchronous Q `[0]`, asynchronous Q `[1]`
`0x1D`	Data `[3:0]`, enable `[4]`	Register `[3:0]`

0x1E	Data [3:0], control [5:4]	Register [3:0]
0x1F	Data [3:0], address [5:4], write [6]	Read data [3:0]
0x20	Operand [3:0], enable [4], clear [5]	ALU result and flags

The D latches in 0x17 and 0x1B have an unspecified power-up value and ignore rst_n. To initialize one on the dev kit, select its experiment, set D with ui_in[0], then raise and lower the gate with ui_in[1] while keeping D stable. With the gate low or the experiment deselected, the latch holds its value. In 0x1B, reset still clears the flip-flop output.

The control values for 0x1E are hold, load, clear, and load. The accumulator feeds its low four bits back into the same ALU used by experiment 22. uio_in[7:6] selects add, subtract, AND, or OR.

FIFO and stack

Experiments 33 (0x21, FIFO) and 34 (0x22, stack) each store four 4-bit values. FIFO returns the oldest entry. Stack returns the newest entry.

ui_in[3:0] supplies write data. uio_in[7:6] selects the command:

Command	Action on a selected rising edge
00	Hold
01	Push: store the input value
10	Pop: remove the next entry
11	Hold (reserved)

Output bits	Meaning
[3:0]	Next entry. Zero when empty
[6:4]	Fill count (0–4). Bit [6] also indicates full
[7]	Empty

Read the next entry **before** the pop edge. After it, the following entry appears. The fill count is uo_out[6:4] (0–4). Push on full and pop on empty are ignored. A held command repeats each clock. Deselecting holds the buffer. Reset empties it even while deselected.

How to test

Shift registers and counters

Experiments 35–43 use four bits, shown on uo_out[3:0]. Bits 7–4 are zero. They update on rising clock edges only while selected. Switching exper-

iments preserves their state. Active-low `rst_n` resets all of them asynchronously: ring counter and LFSR start at `0001`, the others at `0000`.

Code	Inputs and behavior
`0x23`	Shift left. Serial input `ui_in[0]`, enable `ui_in[4]`
`0x24`	Universal shift register. Operation 0 hold, 1 left, 2 right, 3 parallel load
`0x25`	Binary counter: 0–15, then 0
`0x26`	Up/down counter. `ui_in[0]` = 0 up, 1 down. Wraps at 0 and 15
`0x27`	Modulo-6 counter: 0–5, then 0
`0x28`	BCD counter: 0–9, then 0
`0x29`	Ring counter: 1, 2, 4, 8, 1
`0x2A`	Johnson counter: 0, 1, 3, 7, 15, 14, 12, 8, 0
`0x2B`	LFSR: left shift with XOR feedback from bits 3 and 2. 15 nonzero states

The LFSR tap polynomial is $\mathbf{x^4 + x^3 + 1}$, numbering stages 1–4 from bit 0 to bit 3. With this left-shift convention, the forward sequence obeys `s[n+4] = s[n+1] XOR s[n]`, whose characteristic polynomial is the reciprocal $\mathbf{x^4 + x + 1}$. The all-zero state remains locked at zero. Reset therefore seeds `0001`.

Counters and LFSR use `ui_in[4]` as enable. The universal shift register uses `uio_in[7:6]` as operation, `ui_in[3:0]` for parallel load and `ui_in[0]` as serial input. Other input bits are ignored.

For example, select `0x27`, set `ui_in = 0x10`, reset, then apply clock pulses. The output counts 1, 2, 3, 4, 5, 0. Set `ui_in = 0` to hold.

Input synchronization and timing

Experiments 44–48 update on rising edges while selected. Reset is asynchronous and active low. State holds while deselected. Unlisted output bits are zero.

Code	Input	Output
`0x2C`	Synchronous signal `[0]`	Rising pulse `[0]`, falling pulse `[1]`, sampled level `[2]`
`0x2D`	Asynchronous signal `[0]`	Second synchronizer stage `[0]`
`0x2E`	Button `[0]`	Debounced level `[0]`, synchronized level `[1]`, stable count `[3:2]`
`0x2F`	Period minus one `[3:0]`, enable `[4]`	Tick `[0]`, counter `[4:1]`

0x30	Duty [4:0], enable [5]	PWM [0], phase [4:1]

The edge detector expects an input already synchronous to the clock. It emits one pulse per sampled transition. The synchronizer uses two stages. RTL tests check their latency but cannot model metastability.

The debouncer first synchronizes the button, then accepts a changed level after four consecutive samples. Use a slow external clock for a physical button: 400 Hz gives a 10 ms confirmation window plus synchronizer latency. At 50 MHz this is only a logic demonstration, not mechanical debouncing.

The divider emits one clock-enable tick every 1–16 enabled edges. Disabling it clears the tick and holds the count. Reducing the period below the current count causes a tick on the next enabled edge. No derived clock is generated.

PWM has a 16-clock period. Duty 0 is always low. 16–31 is always high. Disabling forces the output low and holds the phase. Duty changes apply immediately, so change duty at a period boundary for clean complete periods.

State machines

Experiments 49–53 use `ui_in[4]` as enable. They advance only while selected and enabled. Reset restores the initial state even while deselected. Inputs must be synchronous to `clk`.

Code	Input	Output
0x31	Start [0], stop [1], enable [4]	Moore release [0], state [1]
0x32	Start [0], stop [1], request [2], enable [4]	Mealy release [0], state [1]
0x33	Phase tick [0], enable [4]	Red [0], amber [1], green [2], state [4:3]
0x34	Request [0], complete [1], enable [4]	Ack [0], busy [1], state [3:2]

Both controllers have two states: **IDLE (0)** and **ACTIVE (1)**. Start enters ACTIVE and stop returns to IDLE on an enabled rising edge. Stop has priority if both inputs are high. Reset returns to IDLE.

Moore release is high whenever the state is ACTIVE. Mealy release is high only when the state is ACTIVE and request is high. Toggle request with the clock stopped to see the difference: the Mealy output follows immediately, while the Moore output stays high. Request is unused in the Moore experiment. Enable controls state updates only. It does not suppress either output.

The traffic light starts red (state 0). Each phase tick advances to red+amber (1), green (2), amber (3), then red. An external controller supplies the phase timing. Holding the tick high advances on every enabled edge.

The handshake starts idle (0). A request enters busy (1). Complete enters ack (2). Keep request high until ack, then lower it to return to idle. Holding request high in ack cannot start another transaction.

Parking lot occupancy counter

Select `0x35`. Sensor A is `ui_in[0]`, sensor B is `ui_in[1]`, and enable is `ui_in[4]`. Sensor value 1 means occupied. Supply synchronous sensor inputs.

- Enter: A/B = `00` → `10` → `11` → `01` → `00`.
- Exit: A/B = `00` → `01` → `11` → `10` → `00`.
- Repeated samples hold the state. Reversing follows the path back without counting.
- Skipped steps discard the crossing. Both sensors must be clear before restarting.

Output: occupancy `[3:0]`, entered `[4]`, exited `[5]`, crossing/recovery active `[6]`, invalid-sequence recovery `[7]`. Occupancy saturates at 0 and 15. The event outputs still report crossings at these limits and last one clock. Selection and enable pause the state and count. Event pulses still clear. Reset clears the count and returns to idle. A sensor sequence already in progress at reset cannot reliably identify a complete crossing.

FSM-Controlled Datapaths

Sequential multiplier

Select `0x36`. Inputs `ui_in[3:0]` and `ui_in[7:4]` are unsigned operands A and B. `uio_in[6]` is start and `uio_in[7]` selects the output view:

View	Output
0	8-bit product
1	Busy `[0]`, done `[1]`, state `[3:2]`, completed steps `[6:4]`

States are idle (0), run (1), and done (2). Assert start for an idle clock edge to capture the operands. Four more selected edges complete the calculation. The datapath reuses two 4-bit adders from group 3. During run, operand changes and start are ignored. The product view shows the partial sum. Done holds while start is high. A selected edge with start low returns to idle, retaining the product. Deselecting pauses the calculation. Reset aborts it.

For 7 × 15, apply `ui_in = 0xF7`, pulse start, and wait four more edges. The product is 105 (`0x69`).

Binarized neural network

Experiment 55 (`0x37`) implements one binary layer with eight shared input bits. `BNN_NEURON_COUNT` selects 1–8 neurons at synthesis time (default: 1). A simulation assertion rejects counts outside this range. Weights and thresh-

olds are programmed through registers. Training takes place outside the chip.

Each neuron compares its eight weight bits with the input using XNOR. A balanced popcount tree counts the matching bits (0–8). Its output is one when the count is at least the programmed threshold. Bits represent −1 and +1, so the corresponding signed dot product is `2 * count - 8`. The hardware only needs the match count. Threshold 0 always passes. Thresholds 9–15 always fail.

`uio_in[7:6]`	Action
`00`	Calculate one neuron per selected rising edge, if enabled and not finished
`01`	Latch the register address from `ui_in`
`10`	Write `ui_in` to the selected register
`11`	Read the selected register on `uo_out`

Address	Register	Meaning
`0x00`	INPUT	Eight shared input bits
`0x01`	CONTROL	Calculation enable in bit 0
`0x02`	OUTPUT	One result bit per neuron, read-only
`0x03`	STATUS	Result valid in bit 0, read-only
`0x04`	NEURON	Neuron selected for configuration and diagnostics
`0x05`	WEIGHTS	Eight weights of the selected neuron
`0x06`	THRESHOLD	Threshold of the selected neuron, bits 3:0
`0x07`	MATCHES	Live XNOR result of the selected neuron, read-only
`0x08`	COUNT	Live match count of the selected neuron, read-only

One shared datapath processes the neurons in ascending order. After exactly `BNN_NEURON_COUNT` processing edges, STATUS becomes 1 and the complete result holds. Other bus operations, disable, and deselection pause the calculation. Outside read mode, `uo_out` shows OUTPUT. Partial results are visible before STATUS becomes 1. Unused high result bits are zero.

Writing INPUT, WEIGHTS, or THRESHOLD clears OUTPUT and STATUS and restarts the sequence at neuron zero. Selecting a neuron does not restart calculation. Out-of-range neuron selections are ignored (the previous selection remains). Writes to read-only or unused registers are ignored. Unused ad-

dresses read zero. Reset clears the weight bank, thresholds, input, result, and enable. Diagnostics are combinational: immediately after reset, MATCHES is `0xFF` and COUNT is 8.

Example for neuron zero: program INPUT=`0xB2`, WEIGHTS=`0xB0`, THRESHOLD=7, then enable calculation and apply the required processing edges. MATCHES is `0xFD`, COUNT is 7, and OUTPUT bit 0 is one. Program the other thresholds above 8 to suppress their output bits.

Sound

Programmable sound generator

Experiment 56 (`0x38`) combines a register bank, square-wave generator, LFSR, falling envelope, and PWM output. It has one voice with tone, noise, tone XOR noise, or tone AND noise.

`uio_in[7:6]`	Action
`00`	Play. No register write
`01`	Latch the register address from `ui_in` on the rising edge
`10`	Write `ui_in` to the selected register on the rising edge
`11`	Read the selected register on `uo_out`

Address	Register
`0x00`	Tone period, low byte
`0x01`	Tone period, high byte
`0x02`	Volume `[3:0]`, 0–15
`0x03`	Enable `[0]`, source `[2:1]` (tone, noise, XOR, AND), envelope mode `[3]`
`0x04`	Envelope decay rate, 0–255
`0x05`	Writing bit `[0]` = 1 starts/restarts the envelope. Reads return zero
`0x06`	Tone prescaler exponent `[2:0]`, default 4 (divide by 16)
`0x07`	Envelope prescaler exponent `[2:0]`, default 2 (divide by 4096)

Registers `0x06` and `0x07` are enabled by `PSG_TONE_PRESCALE` and `PSG_ENVELOPE_PRESCALE` in the current build. If disabled, they read zero and ignore writes, and the corresponding divider stays fixed at its default. Unused addresses read zero and ignore writes. Unused register bits read zero. Reset clears the sound settings, restores the default prescalers, and silences the voice. A held write repeats on every edge, so return to `00` after a trigger write. Period bytes are independent. Write both while disabled for a clean note

change. Each period write restarts the tone divider and noise seed. Other writes do not stop playback.

Outside read mode the output is:

Bits	Meaning
`[0]`	PWM audio
`[1]`	Selected source (tone, noise, XOR, or AND), gated by enable
`[5:2]`	Current amplitude: volume or envelope level
`[6]`	Envelope has not finished
`[7]`	Sound enabled

With `P_tone = 2^TONE_PRESCALE`, the tone frequency is `f_clk / (2 * P_tone * (period + 1))`. At the default divide-by-16 setting and 50 MHz, period 3550 (`0x0DDE`) gives about 440 Hz. The PWM carrier is `f_clk / 15`. The PWM gate passes `level / 15` of the phase slots, so level 15 passes the source continuously. These ratios describe the gate, not the duty cycle of the complete audio waveform. Zero amplitude or disabled sound produces a low audio output. Route PWM through an external low-pass filter and amplifier. Read mode replaces the audio pins with register data. It interrupts the physical audio output even though the generator keeps running.

Noise advances at `f_clk / (P_tone * (period + 1))`. The 16-bit LFSR shifts left, XORs bits 15, 14, 12, and 3, and starts at 1. Tap polynomial: `x^16 + x^15 + x^13 + x^4 + 1`, with 65535 nonzero states. The taps follow AMD/Xilinx XAPP052, table 3.

A trigger loads the envelope from volume. With sound and envelope mode enabled, it drops by one every `P_env * (rate + 1)` clocks, stopping at zero. Here `P_env = 2^(10 + ENVELOPE_PRESCALE)`, which defaults to 4096. Disabling sound or envelope mode pauses decay. Changing volume does not reload an active envelope. Deselecting the PSG pauses all its state. Reset still works.

Example at 50 MHz: write period low `0xDE`, period high `0x0D`, volume `0x0F`, decay rate `0xFF`, control `0x09`, then trigger `0x01`. Return to operation `00`. This plays a roughly 440 Hz note whose amplitude falls to zero in about 0.315 s.

Basic checks

Select `0x01` and try all four combinations of A and B. Select `0x02` and try all eight combinations of A, B, and C. Each adjacent output pair must agree. For `0x08`, test zero, a single set bit, and multiple set bits. For example, A = 1, B = 0, C = 0 gives `uo_out = 0x5A` for `0x01` and `uo_out = 0xC3` for `0x02`.

Each experiment has its own cocotb test file. Combinational tests cover all 256 input bytes where useful. Shared tests check selection codes and experiment switching. Sequential tests provide their own clock and reset sequence.

See Local Simulation for setup and commands, and Experiment Plan for the full curriculum.

External hardware

Digital switches or a controller can supply the selection and input signals. Observe outputs with a logic analyzer or an LED interface with suitable drivers and current limiting. Follow the development board's I/O voltage requirements.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Data 0	Result 0	Experiment select 0
1	Data 1	Result 1	Experiment select 1
2	Data 2	Result 2	Experiment select 2
3	Data 3	Result 3	Experiment select 3
4	Data 4	Result 4	Experiment select 4
5	Data 5	Result 5	Experiment select 5
6	Data 6	Result 6	Operation select 0
7	Data 7	Result 7	Operation select 1

AstraPIO

by **Fabien**

`0166` `50 MHz` `HDL Project`

`github.com/fvannel/AstraPIO`

> "*General-purpose programmable IO prototype; not ready for tapeout*"

Branch warning: this development revision adds a configurable timed pulse-I/O engine beside one general-purpose PIO context. It is not physically qualified or submitted. The already submitted compact fallback is commit `946648ff` / shuttle PR 142, whose central checks passed. Its results do not qualify this extension. See `docs/dense-pio-v5.md` for the architecture, migration, tests and limitations.

Experimental general-purpose programmable digital IO coprocessor, **not yet qualified for fabrication**. This is ABI 0x0500; earlier single/dual-context and SRAM submissions describe different implementations.

How it works

One interpreter uses a programmable 16-word, 10-bit instruction store made from IHP standard-cell latches, with two-byte transmit and receive queues. One instruction slot occurs every four chip clocks. There is no SRAM macro. An autonomous pulse engine can concurrently capture a 1..24-bit prefix, replace it on a regenerated output stream and relay subsequent bits. Timing and routing are programmable; WS2812B V5 is one tested profile, not a hard-wired ASIC purpose.

The host uses SPI mode 0, MSB first: command byte (02 write / 03 read), register address byte, then a 16-bit payload. Program words occupy the low ten bits; upper bits must be zero. Stop the interpreter before changing its program. Probe identity 5049, ABI 0500 and context count 1 before using the driver. SPI high/low periods and CS setup/hold/gap each require at least six ASIC clocks. MISO is not tri-stated.

How to test

Reset, confirm ID/ABI/context count/capacity registers (00/01/02/0F), load and read back code, set entry PC and GPIO mask, then run with mask 1. Use assembler option `--abi 5`; old binaries are incompatible. The pin-level suite checks program integrity, 14 output indices, FIFO/SPI atomicity, UART timing, faults and reset, plus pulse-stream capture/replacement/relay while the interpreter runs independently.

See `docs/dense-pio-v5.md` for the register map, ISA migration and verification scope. Pulse timing must be configured for the actual attached device. No broad WS2812 compatibility or physical timing qualification is claimed at this development stage.

External hardware

A host MCU (intended LPC546xx), clock source and wiring for the selected programmable protocol. The exact LPC546xx board transport/DMA and board timing remain to be qualified. No external memory is required for the compact examples.

Pin use

ui0/1/2: host SCK/MOSI/CS_n; ui3..7: five dedicated PIO inputs.

uo0/1: MISO/IRQ; uo2..7: six dedicated PIO outputs.

uio0..7: eight bidirectional PIO pins. The single interpreter may own any of the 14 outputs; the pulse engine exclusively owns at most one. Masks cannot overlap. All pad output values are forced low and uio directions disabled on reset or deselection; dedicated outputs are not tri-state.

Qualification status

This branch is for development and independent CI. No SRAM waiver, DRC filtering or nonblocking signoff exception is permitted. Placement/routing, timing including latch/clock-gate paths, DRC, LVS and official precheck must all pass before a new revision can be considered.

The compact candidate at commit `946648f` completed the official flow in 1×2 tiles on 2026-09-18: Magic/KLayout DRC, LVS, XOR, antenna checks, nominal-RC timing at three cell corners, ten official prechecks and ten routed functional tests passed. The latest eleven-test suite also passes locally on that exact routed netlist. That compact candidate was submitted as PR 142; this timed extension has not been submitted. Board timing review remains separate. Detailed fallback evidence is in `docs/compact-validation.md`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	HOST_SCK	HOST_MISO	PIO_IO0
1	HOST_MOSI	HOST_IRQ	PIO_IO1
2	HOST_CS_N	PIO_OUT0	PIO_IO2
3	PIO_IN0	PIO_OUT1	PIO_IO3

#	Input	Output	Bidirectional
4	PIO_IN1	PIO_OUT2	PIO_IO4
5	PIO_IN2	PIO_OUT3	PIO_IO5
6	PIO_IN3	PIO_OUT4	PIO_IO6
7	PIO_IN4	PIO_OUT5	PIO_IO7

Kraken IO Subprocessor

by **Matthias Musch**

0168 HDL Project

github.com/matztron/tto_mini_kraken_IHP26b

"A PIO-like state machine that runs short pioasm scripts"

How it works

Kraken IO Subprocessor (mini) is a two-tile, RP2040-PIO-compatible programmable I/O block for Tiny Tapeout. It runs short programs compiled with pioasm and is meant for bit-banged protocols (UART, SPI, WS2812, and similar) without tying up a host CPU.

The design contains:

- **One PIO state machine** with an 8-word instruction memory (IMEM), 2-bit GPIO (`uo[1:0]`), TX/RX shift registers, and depth-2 TX/RX FIFOs.
- **A pin loader** for programming IMEM and SM configuration without a system bus. Set **`uio[7]=1`** to enter **config mode** set **`uio[7]=0`** for **run mode**.
- **A runtime clock divider** so the SM can run slower than the chip clock.

Instruction encoding and opcodes match the RP2040 PIO (`JMP`, `WAIT`, `IN`, `OUT`, `PUSH`/`PULL`, `MOV`, `IRQ`, `SET`). The SM fetches from IMEM, executes one instruction per enabled clock (after optional delay cycles), drives the two GPIO outputs, and can shift data to/from the host through the FIFO interface.

Pin loader (config mode, `uio[7]=1`)

Pulse **`uio[0]`** (rising edge) to apply the operation, with payload bytes on **`ui[7:0]`**:

Encoding	Operation	Details
`uio[6]=0`	**IMEM write**	`uio[4:2]` = address (8 words), `uio[1]` = half, `uio[0]` = strobe. Low byte on first strobe (`half=0`), high byte on second (`half=1`). Latch addr/half while `strobe=0`.
`uio[6]=1`	**Other ops**	`uio[5:3]` = op code, `uio[0]` = strobe

`uio[5:3]`	Operation	`ui[7:0]` meaning
1	EXEC wrap	`ui[3:0]` = wrap bottom, `ui[7:4]` = wrap top (low 3 bits used)

2	PIN config	Set/out/sideset counts, side-set enable, side-set pindir
3	CLKDIV low	Low byte of 16-bit divider
4	CLKDIV high	High byte of 16-bit divider
5	SHIFT	Input/output shift direction, autopush, autopull
6	THRESH	Push/pull bit thresholds
7	INPIN	Input base pin and jump pin

Run mode (`uio[7]=0`)

Pin	Role
`ui[1:0]`	GPIO inputs to the SM
`ui[7:0]`	TX FIFO data (e.g. UART TX bytes)
`uio[0]`	`tx_push` — rising edge pushes `ui[7:0]` when FIFO not full
`uio[1]`	`sm_enable` (with `ena`) — SM runs when high
`uio[2]`	`sm_restart` — rising edge clears SM and FIFOs
`uio[3]`	`rx_pop` — rising edge pops RX FIFO into a host-visible latch
`uio[4]`	RX hold acknowledge — clears latched RX byte on `uo_out`
`uo[1:0]`	PIO GPIO outputs
`uo[2]`	`tx_full`
`uo[3]`	`rx_empty`
`uo[7:4]`	IRQ flags (when no latched RX byte)
`uo[7:0]`	Latched RX byte (after `rx_pop`, until `uio[4]` ack)

How to test

RTL simulation

From the `test/` directory (requires pioasm on `PATH`, or macOS pico-sdk-tools):

```
make assemble   # hello_world.pio → generated/hello_world.hex
make -B         # cocotb: pin-load IMEM, enable SM, check uo[0]
square wave
```

The default test loads a two-instruction SET-pin square wave through config mode (`uio[7]=1`), then enables the SM in run mode and checks `uo[0]` toggles `1,1,0,0,...`.

On silicon (Tiny Tapeout IHP demo board)

1. **Select this design** on the chip multiplexer (reset `sel_rst_n`, pulse `sel_inc` to your project address).
2. **Hold reset** (`rst_n` low), then release.

3. **Program the SM** in config mode (`uio[7]=1`): write IMEM words, set wrap/clock divider/shift thresholds, then leave config mode (`uio[7]=0`).
4. **Run**: drive `uio[1]=1` (`sm_enable`). Pulse `uio[2]` once if you need a clean restart.
5. **UART TX example**: load a standard 8N1 UART TX pioasm program with `wrap` covering your program range, set `clkdiv` for your baud rate, then for each byte place it on `ui[7:0]` and pulse `uio[0]`. Watch `uo[2]` (`tx_full`) before pushing; serial data appears on `uo[0]` (or whichever OUT pin your program uses).
6. **UART RX example**: connect the incoming serial line to the GPIO input used by your program (`ui[0]` or `ui[1]`). When data arrives, pulse `uio[3]` to latch a received byte on `uo[7:0]`, read it from the host, then pulse `uio[4]` to release the latch. Check `uo[3]` (`rx_empty`) before popping.

Monitor `uo[7:4]` for IRQ flags if your program uses `irq` instructions.

Gate-level simulation (after hardening): copy the generated netlist to `test/gate_level_netlist.v` and run `make -B GATES=yes`.

External hardware

- **Tiny Tapeout IHP demo / breakout board** and a **3.3 V host microcontroller** (RP2040, Arduino, ESP32, etc.) to select the design, bit-bang the config loader, and drive run-mode control pins.
- **For UART**: a **USB–serial adapter** or **logic analyzer** on the PIO TX output (`uo[0]` or `uo[1]`, depending on your program). For RX testing, tie the adapter's TX to the PIO input pin, or loop TX back to RX on the breadboard.
- **Optional**: LEDs or a scope on `uo[1:0]` to visualize bit-banged waveforms (WS2812, SPI, etc.).

No PMOD or dedicated daughterboard is required; all control and data paths use the standard Tiny Tapeout `ui` / `uo` / `uio` pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	gpio_in (also LSBs of TX byte when pushing) / TX byte for UART programs	PIO GPIO out	tx_push (rising edge)
1	gpio_in (also LSBs of TX byte when pushing) / TX byte for UART programs	PIO GPIO out	sm_enable (level, with ena)

#	Input	Output	Bidirectional
2	TX byte for UART programs	tx_full	sm_restart (rising edge)
3	TX byte for UART programs	rx_empty	rx_pop (rising edge)
4	TX byte for UART programs	irq_flags[3:0] (or full uo[7:0]=rx byte when rx_hold_valid)	rx_hold ack (clear latched RX byte on uo)
5	TX byte for UART programs	irq_flags[3:0] (or full uo[7:0]=rx byte when rx_hold_valid)	config-mode select
6	TX byte for UART programs	irq_flags[3:0] (or full uo[7:0]=rx byte when rx_hold_valid)	config-mode select
7	TX byte for UART programs	irq_flags[3:0] (or full uo[7:0]=rx byte when rx_hold_valid)	config-mode select

snake game

by **Valeria Molano**, **Luisa Fernanda Avendaño** & **Juan David Guerrero**

0170 10 MHz HDL Project

github.com/divadnauj-GB/snake_game_uptc

> "*This is a simple snake game project selected as the best course project from the 'Digital Asic Design' 2025-2 course at Universidad Pedagogica y Tecnologica de Colombia*"

Authors: Valeria Molano and Luisa Fernanda Avendaño.

This is a simple snake game project selected as the best course project from the **Digital Asic Design** 2025-2 course at Universidad Pedagogica y Tecnologica de Colombia - Tunja. The course has been taught by professor Juan David Guerrero Balaguera.

Here you can find a Demo of an FPGA implementation of the Snake Game.

Architecture details

This project consist of an minimal ASIC design of the classical Snake videogame using four 8X8 Led Matrices and four input push buttons for controlling the game. The ASIC is all full implemented using verilog.

Technical Specification for Snake Game
Microchip Snake Game (snake_game_uptc)

- **ASIC Flow:** LibreLane / OpenLane
- **Top Module:** snake_top
- **Project:** Tiny Tapeout
- **License:** Apache-2.0

1. General System Description

Below you can see the description and hardware implementation (Verilog RTL) of the Snake game. The design is intended to control 4 coupled 8×8 LED matrices, forming a 16×16 pixel game area. The circuit integrates clean reading of mechanical buttons, the complete game logic through a state machine, snake and food position control, a BCD scoring system, and SPI communication to drive the external displays via the MAX7219 driver.

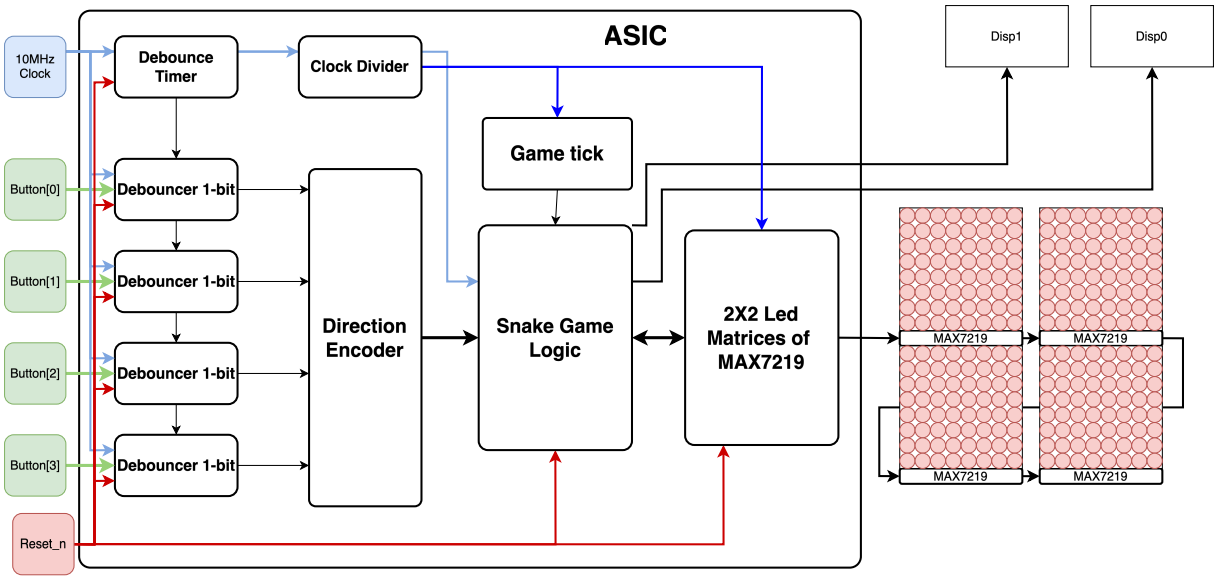


Figure 170.1: Figure 1: Hardware architecture and data flow in snake_top.

Figure 1: Hardware architecture and data flow in snake_top.

2. Internal Module Description

2.1 snake_top (Main Module)

- **Clock Management:** Takes the 10 MHz base clock (CLOCK_10) and generates a 1 MHz signal (slow_clk) exclusively for SPI transmission.
- **Game Speed:** The game_tick signal controls every step of the snake. It starts with a long wait cycle and, every time an apple is eaten, the set_timer signal reduces that wait time by 16,384 cycles. This progressively accelerates the game in 10ms approx.

2.2 Debounce (Debounce Filter)

Cleans the mechanical noise from the buttons. Each button (KEY [3:0]) enters a shift register. The signal only changes its internal logical state if it remains stable (high or low) for 8 consecutive clock cycles.

2.3 Game_core (Game Logic)

- **FSM (11 States):** Controls the entire flow: initializes the screen, updates the snake's coordinates, checks for collisions, and sends commands to draw the matrix row by row.
- **FIFO Memory:** A 32-position circular register that stores the X, Y coordinates of the snake's body. It uses pointers to add the new head and delete the last part of the tail with each movement.
- **LFSR Generator:** Creates pseudo-random numbers based on a linear-feedback shift register. It is used to calculate random coordinates for the appearance of food.
- **BCD Score:** A binary-coded decimal counter. Its outputs go directly to 8 pins (4 for tens, 4 for units), allowing simple LEDs to be connected to display the score in binary without needing additional decoders.

2.4 Spi_driver (Display Controller)

Receives 16-bit parallel commands from the game_core and serializes them. Generates the necessary clock, data, and chip select (CS) pulses to communicate in a standard way with the MAX7219 ICs of the LED matrices.

3. Pin Specification (Pinout)

Table 1: Input and output signals map of the Design.

Signal	Type	Width	Origin / Destination	Description
CLOCK_10	Input	1 bit	System	Base board clock (10 MHz).
SW [0]	Input	1 bit	Switch	Asynchronous reset (Active low 0).
KEY [0]	Input	1 bit	Button	Moves Left.
KEY [1]	Input	1 bit	Button	Moves Right.
KEY [2]	Input	1 bit	Button	Moves Down.
KEY [3]	Input	1 bit	Button	Moves Up.
MAX_DIN	Output	1 bit	SPI Driver	Serial data output to MAX7219.
MAX_CLK	Output	1 bit	SPI Driver	Clock for SPI transmission.
MAX_CS	Output	1 bit	SPI Driver	Chip Select (CS) signal for the matrix.
HEX0 [3:0]	Output	4 bits	Score LEDs	Counter units (for 4 LEDs).
HEX1 [3:0]	Output	4 bits	Score LEDs	Counter tens (for 4 LEDs).

Table 2: Input and output pins on the TinyTapeot chip. ||||| |-|-|-|-| |#|Input (ui)|Output (uo) |Bidirectional (uio)| |0|left | spi_din | disp1[0] | |1|right | spi_clk | disp1[0] | |2|down | spi_cs | disp1[0] | |3|up | - | disp1[0] | |4| - | disp0[0] | - | |5| - | disp0[0] | - | |6| - | disp0[0] | - | |7| - | disp0[0] | - |

How it works

See the General System Description and Internal Module Description sections above for the full technical breakdown of `snake_top`, `debounce`, `game_core`, and `spi_driver`.

How to test

1. Connect the 4-module MAX7219 LED panel and the 4 direction push-buttons (see the **Pin Specification** table above).
2. Release reset (`SW[0]`).
3. Use the direction buttons to guide the snake toward the food. The game ends on collision with a wall or with the snake's own body.

External hardware

- 4× individual 8×8 MAX7219 LED matrix modules (2×2 grid, `DIN`/`DOUT` daisy-chained, `CLK`/`CS` in parallel).
- 4× momentary push-buttons (active-low) for direction control.

Notes and Known Behaviors

Supplementary material in the repository:

1. **YouTube Demo Video:** Demonstration of physical operation on FPGA. Link: https://youtu.be/Ej_GrQa6S1c
2. **Confirmation Images:** Visual evidence of the states and rendering of the game.

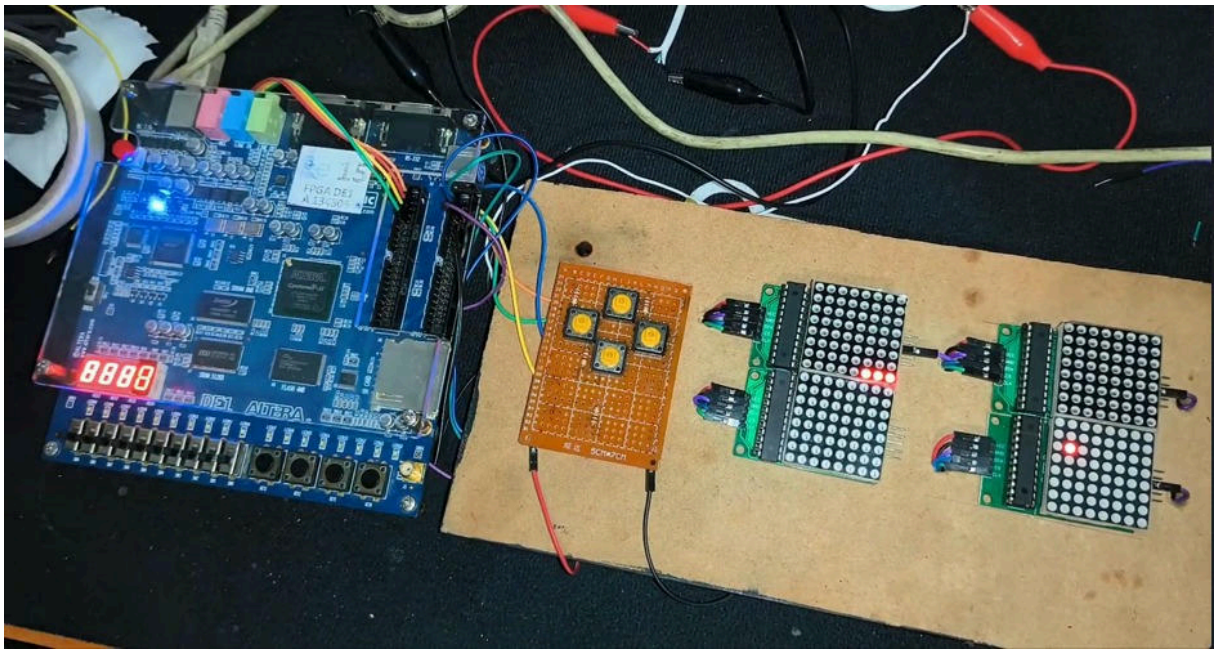


Figure 170.2: Figure 2: Implementation on FPGA

Figure 2: Implementation on FPGA

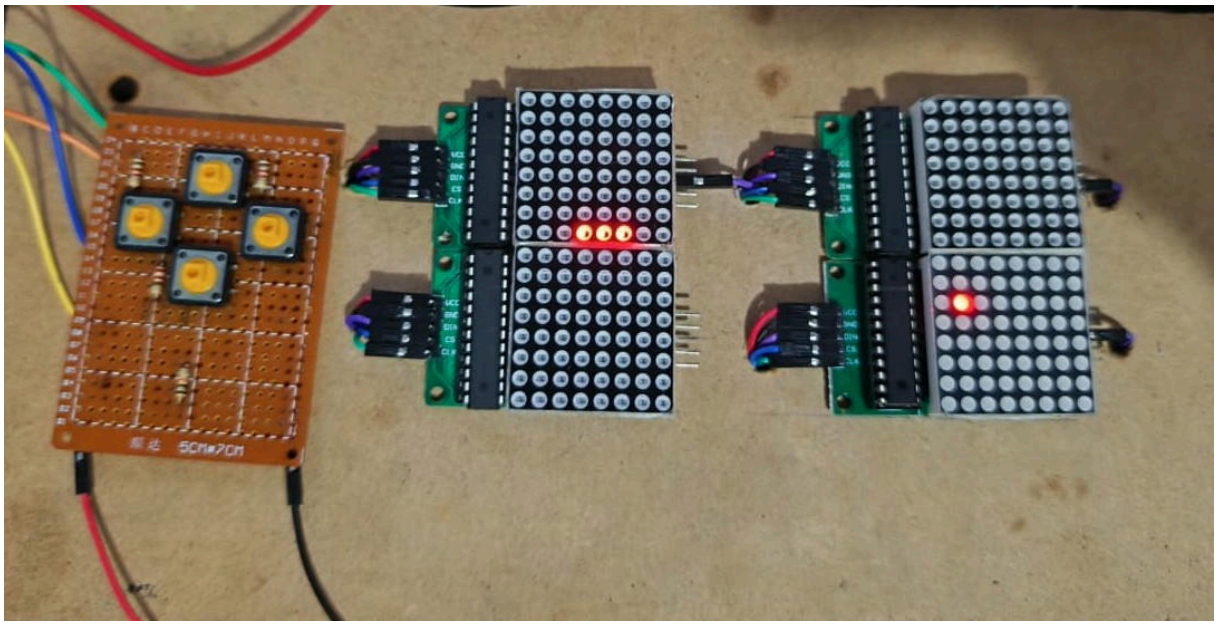


Figure 170.3: Figure 3: Working operation evidence

Figure 3: Working operation evidence

Table 2: Operational details to consider for hardware integration.

Code	Behavior	Recommended Solution
NOTE-01 — KEY Inversion	The direction pins on the main module are logically inverted (e.g., when receiving “Up”, “Down” is processed).	Physically cross the button connections on the motherboard during assembly.
NOTE-02 — Self-collision	If the direction opposite to the current movement is pressed (e.g., moving right and pressing left), the game applies the 180° turn before advancing, detecting a collision with its own body.	The player must make “U” turns in two steps (e.g., press “Up” and immediately “Left”).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	left	spi_din	disp1[0]

#	Input	Output	Bidirectional
1	right	spi_clk	disp1[1]
2	down	spi_cs	disp1[2]
3	up	—	disp1[3]
4	—	disp0[0]	—
5	—	disp0[1]	—
6	—	disp0[2]	—
7	—	disp0[3]	—

AdExp DPI Neuron

by **Saptarshi Ghosh**

0194 HDL Project

github.com/amisapta15/DPI-adexp-neuron-ttihp

"Adaptive Exponential Integrate-and-Fire Neuron "

🧠 AdExp DPI Neuron Network (tt_um_dpi_adexp)

A digital spiking-neuron network for the IHP SG13G2 shuttle (TTIHP-26b). The active source set emulates Adaptive Exponential (AdEx) integrate-and-fire dynamics with DPI-style synaptic coupling using **shifts, adds and subtracts** only: `project.v`, `adex_config.v`, `adex_block.v`, `adex_pair.v`, and `adex_network.v` contain no multiplier, divider, or lookup table.

The baseline configuration is a **population of four blocks forming two excitatory/inhibitory (E/I) pairs** with reciprocal inhibition. Each block is a self-contained population primitive: 1 membrane prime, 2 fast-positive units, 3 slow-negative adaptation units, and 3 small slow-period counters. The architecture is compositional: `adex_block` -> `adex_pair` -> `adex_network`.

How it works ⚙️

Arithmetic model (per block, per clock cycle)

All state variables are signed fixed-point. The membrane prime `v` is Q4.12 (1.0 = 4096), fast units are 10-bit signed, slow units are 12-bit signed. Every term is a shift-scaled constant or a state value; coupling is a binary selection of a constant, never a product.

Prime (membrane) update, no spike:

```
v' = v − (v >> KV) + fast_drive − slow_drive + Iext − inh + exc
```

Fast-positive units (drive the upstroke; accumulate while `v > VTRIG`):

```
f_i' = f_i − (f_i >> KF_i) + (v > VTRIG ? FINC_i : 0)
```

Slow-negative units use a 7-bit phase counter p_i and their configured period KS_i:

```
p_i' = (p_i = KS_i − 1) ? 0 : p_i + 1
```

On a period tick, the unit relaxes toward zero by one eighth of its magnitude, with a minimum one-count change to avoid a quantisation dead zone. Otherwise it holds its value. Every slow unit receives the shared runtime `WBUMP_Q` on the block's own spike:

```
w_i' = w_i + (p_i = KS_i - 1 ? relax(w_i) : 0) + (v > VTH ?
WBUMP_Q : 0)
```

On `v > VTH` the block emits a registered **spike** and performs a subtractive reset `v' = v - VSTEP` (classic AdEx spike-and-reset, without the exponential term: the fast units supply the upstroke instead). Drive contributions are

```
fast_drive = (f0 >> FSH0) + (f1 >> FSH1)
slow_drive = (w0 >> SSH0) + (w1 >> SSH1) + (w2 >> SSH2)
```

all state updates are saturated to their storage widths. This is the "shift-only AdEx emulation" described in `src/implementation_plan.md`; the synthesised RTL is definitive for fixed-point and shift semantics.

The three slow units and coprime periods

Each block carries **three** slow-negative units with distinct integer update periods. `KS_i` is the number of core cycles between relaxation updates, not a right-shift exponent. With the fixed one-eighth relaxation, the approximate exponential time constant is $8KS_i$ cycles at magnitudes above eight counts; the minimum one-count relaxation completes the return to zero at low magnitudes.

Pair	E slow periods in cycles (KS0, KS1, KS2)	I slow periods in cycles (KS0, KS1, KS2)
pair 0	(5, 7, 11)	(13, 17, 19)
pair 1	(23, 29, 31)	(37, 41, 43)
pair 2 (stretch, N_PAIRS=3)	(47, 53, 59)	(61, 67, 71)

All three slow units bump on the block's own spike (`W += WBUMP_Q` each). This produces spike-frequency adaptation: the slow units accumulate during a spike train and relax at their independently scheduled periods.

Network structure

- `adex_block` — the population primitive above (parameters in the next section).
- `adex_pair` — one E block and one I block with **reciprocal inhibition**: E spikes inhibit I and vice versa (`inh_in` port, magnitude = runtime `INH_AMT_Q`, default 512). E and I use different slow-period triples.
- `adex_network` — two pairs (baseline, `N_PAIRS=2`), each pair isolated from the other. Its optional three-pair configuration (`N_PAIRS=3`) adds an excitatory ring E0 -> E1 -> E2 -> E0. The submitted top wrapper fixes `N_PAIRS=2`; using the stretch configuration also requires widening its wrapper ports.

Configuration controls

`adex_config` supplies a reset-defaulted active register bank. SPI writes first update a shadow bank; a separate `COMMIT` frame transfers every field to the active bank on one core-clock edge. The following controls are runtime configurable.

Group	Runtime field	Default	Meaning
Per neuron	`VTH_Q`	4096	signed 14-bit spike threshold (max +8191; E0 test uses 5120)
Per neuron	`IEXT_Q`	1024	signed 12-bit input-current magnitude (default 1024, tests <=1024)
Global	`VTRIG_Q`	3072	signed 14-bit fast-unit trigger
Global	`VSTEP_Q`	4096	signed 14-bit subtractive reset step
Global	`FINC0`, `FINC1`	128, 192	unsigned 9-bit fast-unit increments
Global	`WBUMP_Q`	256	unsigned 10-bit bump for each slow unit (default 256, tests <=600)
Global	`INH_AMT_Q`	512	unsigned 12-bit reciprocal-inhibition magnitude (default 512, tests <=256)

The 14-bit V/VTRIG/VSTEP, 12-bit IEXT, and 12-bit INH_AMT field widths are the demonstrated operating ranges (see `src/adex_config.v` header); the 14-bit signed thresholds are required because the E0 phase-locked test raises `VTH_Q` to 5120 (13-bit signed caps at +4095), and 12-bit signed IEXT is required for +1024 (11-bit signed caps at +1023).

`VINIT_Q`, `KV`, `KF0/1`, `FSH0/1`, `KS0..2`, `SSH0..2`, `SLOW_DECAY_SHIFT`, and the optional stretch-ring excitation magnitude remain compile-time constants. Keeping shift counts static avoids variable shifters in the neuron datapath.

Pin map (baseline, N_PAIRS=2)

Pin	Direction	Function
`clk`	in	system clock
`rst_n`	in	active-low reset
`ena`	in	unused
`ui_in[0]`	in	PWM input current, E0
`ui_in[1]`	in	PWM input current, I0
`ui_in[2]`	in	PWM input current, E1

`ui_in[3]`	in	PWM input current, I1
`ui_in[7:4]`	in	unused
`uio_in[0]`	in	SPI `CS_N`
`uio_in[1]`	in	SPI mode-0 `SCLK`
`uio_in[2]`	in	SPI `MOSI`
`uio_in[7:3]`	in	unused
`uo_out[0]`	out	registered E0 spike indicator
`uo_out[1]`	out	spike I0
`uo_out[2]`	out	spike E1
`uo_out[3]`	out	spike I1
`uo_out[4]`	out	any-spike aggregate (E0
`uo_out[7:5]`	out	tied low
`uio_out`, `uio_oe`	out	tied low; this is a write-only SPI interface

`ui_in[7:4]`, `uio_in[7:3]`, and `ena` are ignored by the RTL. The four active drive pins must be driven to known binary values. They are sampled once per clock and act as binary current enables; an external source may provide PWM, while a constant high level supplies the configured `IEXT_Q` every cycle. An unresolved active input can propagate an unknown value through the state update.

How to test 🧪

1. **Reset**: hold `rst_n` low for at least 5 clock cycles, then release.
2. **Drive**: set some of `ui_in[3:0]` high (constant PWM = constant input current). E.g. drive all four high.
3. **Observe**: `uo_out[0..3]` is high after a clock edge when the corresponding block's pre-update `v` was greater than `VTH_Q`; it is not edge-detected. `uo_out[4]` is the OR of those four registered indicators.
4. **Configure (optional)**: hold `CS_N` low, send one or more 32-bit MSB-first SPI mode-0 write frames, then send a `COMMIT` frame. `SCLK` must be no faster than `clk/8`; keep `CS_N` stable for at least two `clk` cycles before and after each frame.

Frame	Bits [31:28]	Bits [27:24]	Bits [23:20]	Bits [19:4]	Bits [3:0]
Write	`0xA`	target 0=E0, 1=I0, 2=E1, 3=I1, F=global	field ID	16-bit value	`0`
Commit	`0xC`	`0`	`0`	`0`	`0`

Per-neuron fields are 0=`VTH_Q`, 1=`IEXT_Q`. Global fields are 0=`VTRIG_Q`, 1=`VSTEP_Q`, 2=`FINC0`, 3=`FINC1`, 4=`WBUMP_Q`, and 5=`INH_AMT_Q`. Unsigned fields use the least-significant 9, 11, or 15 bits of the value field as applicable.

With the default parameters, the intended pin-level checks are:

- All four blocks eventually spike when all four active drive pins are high.
- **Adaptation**: E0's late inter-spike intervals exceed its early inter-spike intervals.
- **Inhibition**: E0's firing rate decreases while I0 is driven and recovers after I0 stops.
- **Pair isolation**: driving pair 0 does not create a direct input to pair 1 in the baseline configuration.
- **No lock**: E0 and I0 do not sustain in-phase spiking under the default drive used by the testbench.

These are behavioural checks, not validated numerical characterisation. The RTL has been re-verified with the full 14-test suite (see Verification scope below).

The automated suite is `test/test.py` (cocotb, **14 tests**: reset state, directed E0 block arithmetic against a Python fixed-point reference, SPI shadow/commit behavior, silence, spiking + aggregate OR, adaptation ratio, tonic f-I response, fast spiking, inhibition suppression, pair isolation, no-lock, phase-locked alternation, bursting pattern, and burst length vs WBUMP). The arithmetic test uses nine directed vectors and exercises period-counter wrap; the SPI test checks that a write has no effect before commit and reaches E0 after commit. Run with `make -B` in `test/`. Three tests reach into internal hierarchy (`dut.net.pair0.e_block` or `dut.u_config`); at gate level the netlist can flatten that hierarchy, so their internal checks log a warning and return. `test_reset_state` still checks that the visible outputs are zero in reset.

Verification scope (final pre-tapeout run, 2026-08-20)

- The active synthesis source list contains the wrapper, configuration bank, block, pair, and network modules; the legacy LUT core is excluded.
- **Toolchain:** Icarus Verilog 13.0 (stable) with cocotb 2.0.1 on Python 3.11 (the `tt` mamba environment), driven by `make -B` in `test/`.
- Cocotb **14/14 PASS** at RTL (`TESTS=14 PASS=14 FAIL=0 SKIP=0`). Measured spike metrics from this run: basic spiking E0=627, I0=376, E1=328, I1=329 over 6000 cycles with the aggregate-OR check clean; adaptation ISI head8=6.5 -> tail100=9.2 (E0 fired 872 times); tonic f-I IEXT=512->241 spikes/ISI 8.28 vs IEXT=1024->424 spikes/ISI 4.71; fast spiking 997 spikes/ISI 2.00 with max high-run 1 (pulse-clean); inhibition E0 alone=437, with-I0=419, after=435 while I0 fired 252; pair isolation drove pair 0 giving [E0,I0,E1,I1]=[315,191,0,0]; no-lock coincidence fraction=0.42 (threshold 0.5);

phase-locked alternation E=399/I=400 at ISI (5.00, 5.00) with disjoint alternating one-cycle pulses; bursting 852 spikes/166 bursts, avg burst size 5.1, avg inter-burst gap 73.9 vs intra-burst ISI 4.1; WBUMP sweep avg burst size 5.1 (WBUMP=200) -> 2.0 (WBUMP=600).

- Three tests (`test_reset_state`, `test_arith_block`, `test_spi_shadow_commit`) access internal hierarchy; at gate level these log a warning and return without checking internals. The cocotb run emits deprecation warnings from the testbench's use of the older `units=`/`binstr`/`signed_integer` APIs under cocotb 2.0.1; they are cosmetic and do not affect any assertion.
- `verilator --lint-only -Wall` (Verilator 5.050) on the full five-source set (`project.v`, `adex_config.v`, `adex_block.v`, `adex_pair.v`, `adex_network.v`) is **warning-clean — zero warnings, exit 0**.

RTL area-reduction edits (behaviour-identical, verified by the 14-test suite)

- Factored the duplicated `(spike_now ? wbump_14 : 0)` term into one shared `wbump_term_14` wire across the three slow-unit accumulators.
- Narrowed the `sat16` helper input from 20-bit to 18-bit (the prime accumulator `v_sum` is 18-bit; the dropped bits were pure sign-extension).
- Removed the dead, unreferenced `wbump_q16` widen wire, and dropped the unreachable negative branch of `slow_relax` (the slow units start at 0 and only ever relax toward zero, so their state is provably non-negative). All of these preserve every state the reference model drives; the arithmetic lock `test_arith_block` still passes and the full five-source set now lints `-Wall` clean with zero warnings. Base area is roughly 78% of the 2x2 core by local yosys estimate (behaviour-identical to the pre-reduction baseline); final density is settled by the shuttle's OpenROAD place-and-route run, not by further RTL shaving.

Known limitations

- **SPI is write-only and clock-domain limited.** There is no MISO/readback path. The implementation synchronises SPI inputs into `clk`, so it is intended for slow configuration traffic only (`SCLK <= clk/8`), not a high-speed independent SPI clock.
- **Runtime scope is intentionally lean.** Shift counts, slow periods, reset value, and ring-excitation strength remain compile-time to avoid barrel shifters and a larger configuration bank. The `N_PAIRS=3` stretch branch reuses E0/I0 runtime controls for pair 2; the submitted wrapper is fixed at `N_PAIRS=2`.
- **Observability**: at gate level only the spike pins are visible; internal `v/f/w` states are not exposed (the old debug bus is gone).

- `src/adex_neuron_system_tt_lut32.v` is the deprecated Q8.7 LUT-based core from the earlier iteration. It contains LUT, multiplication, and division logic, is not in `info.yaml`'s source list, and is not synthesised.

External hardware

N/A. Self-contained digital core; no external components required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PWM_E0	spike_E0	SPI_CS_N
1	PWM_I0	spike_I0	SPI_SCLK
2	PWM_E1	spike_E1	SPI_MOSI
3	PWM_I1	spike_I1	—
4	—	any_spike	—
5	—	—	—
6	—	—	—
7	—	—	—

snn-voice-calculator

by **mauro**

`0202` `1 MHz` `HDL Project`

`github.com/mauro-ciccone/ttihp-snn-voice-calculator`

"snn-powered voice recognition single digit calculator"

How it works

This is a Spiking Neural Network for audio classification.

Explain how your project works

How to test

This is a Spiking Neural Network for audio classification.

Explain how to use your project

External hardware

This is a Spiking Neural Network for audio classification.

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

PDM microphone

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	unused	dummy_out_0	unused
1	unused	dummy_out_1	unused
2	unused	dummy_out_2	unused
3	unused	dummy_out_3	unused
4	unused	dummy_out_4	unused
5	unused	dummy_out_5	unused
6	unused	dummy_out_6	unused
7	unused	dummy_out_7	unused

Tiny FABulous FPGA

by **Leo Moser**

0203 HDL Project

github.com/mole99/tt-fabulous-ihp-26b

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

How it works

Tiny FABulous FPGA for IHP26b.

This design implements a tiny FPGA with 168 LUT4+FF. The FPGA fabric is 9x5 tiles in size, of which 7x3 are LUT4x8_ha tiles. The logic cells include a vertical carry-chain in upwards direction, allowing for fast additions up to 23-bits.

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

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

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

The exact available resources can be seen in this table:

Primitive	Available	Description
FABULOUS_LC	168	Logic cells with LUT4+FF and carry-chain.
IOBUF	26	Input/output buffers.
GBUF	4	Global buffers to supply clock, reset and enable to the flip-flops.
SYS_RESET	1	Can be used to reset the design after configuration.

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

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

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

How to test

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

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

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

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

External hardware

None

Project Pinout

Digital Pins

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

Configurable Low-Latency Match-Action Packet Processor

by **Giulio Girelli**

224 | 50 MHz | HDL Project

`github.com/GiulioGirelli/tiny-tapeout-packet-processor`

"Configurable low-latency packet-processing ASIC with programmable match-action classification"

Configurable Low-Latency Match-Action Packet Processor

Author: Giulio Girelli

How it works

A byte stream of packets flows into the processor, one byte per clock cycle. Each packet is a sequence of *chunks*: a metadata (type) byte followed by zero or more data bytes belonging to the most recent metadata. The processor filters the stream and collects, for each of 3 programmable type values, the first 2 data bytes (16 bits, MSB-first) into a 3×16-bit collection table. Partial, interleaved and repeated chunks are legal; overflow bytes are dropped.

When all 3 slots are full, 3 configurable rules are evaluated in parallel. Each rule has a flag byte (1 enable bit, 3 type-check enable bits, 4 action bits) and three 16-bit match values. A rule hits when it is enabled and every enabled type matches the collected value; if several rules hit, the highest rule number wins. The result (winning rule number, number of checked types, 4-bit action) is presented on the output pins while the packet keeps streaming, until one cycle after the end-of-packet flag; the table is then cleared for the next packet. All rules, type values and statistics are programmed and inspected through a configuration mode (single-cycle READ, two-cycle WRITE) over a 5-bit, 32-register address map. Statistics counters (16-bit total bytes, 16-bit total packets, 8-bit per-rule and no-rule hit counters) are read-only and wrap naturally. Synchronous soft resets (`rst_type`) clear the collection table/packet state (01), the rule table and type registers (10), or the statistics counters (11). Resets 10/11 preserve ongoing packet/result/error state while transient responses expire normally. Type reset keeps the current chunk's slot selection; the next metadata byte uses the new type values. The declared clock is 50 MHz.

`ena` is sampled at rising clock edges: when low, inputs are ignored, internal state holds and outputs become idle at that edge. There is no asynchronous

output clamp when the clock is stopped. The dedicated active-low hard reset remains asynchronous and works even with `ena=0`.

How to test

Configuration mode (`cfg_mode=1`): present the register address with `packet_status=01` (READ, value on `out` next cycle) or `10` (WRITE address), then the data with `packet_status=11` (WRITE completes). Packet mode (`cfg_mode=0`): stream metadata bytes (`packet_status=10`), data bytes (`packet_status=01`) and end-of-packet (`packet_status=11`); watch `out_state` — 001 packet active, 010 result available, 011/110 WRITE/READ completed, 100 end-of-packet too early, 101 WRITE error, 111 input error.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0]	out[0]	cfg_mode
1	in[1]	out[1]	packet_status[0]
2	in[2]	out[2]	packet_status[1]
3	in[3]	out[3]	rst_type[0]
4	in[4]	out[4]	rst_type[1]
5	in[5]	out[5]	out_state[0]
6	in[6]	out[6]	out_state[1]
7	in[7]	out[7]	out_state[2]

Universal Binary to Segment Decoder

by **Rebecca G. Bettencourt**

0225 HDL Project

`github.com/RebeccaRGB/ttihp-ubcd`

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

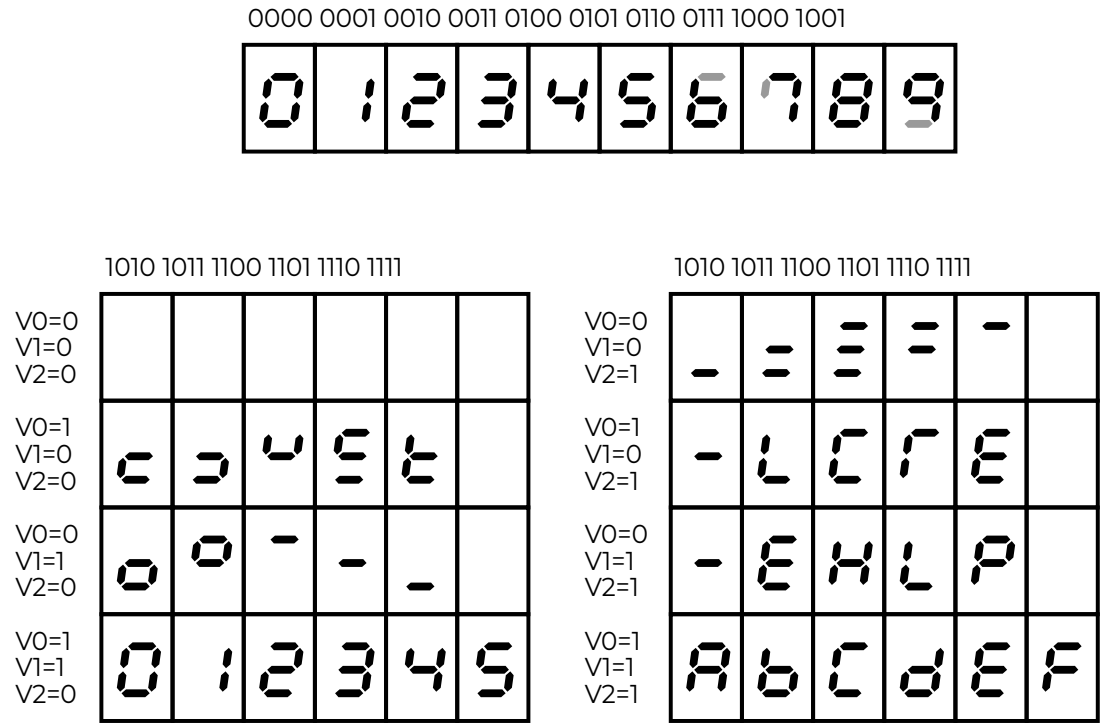
How it works

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:

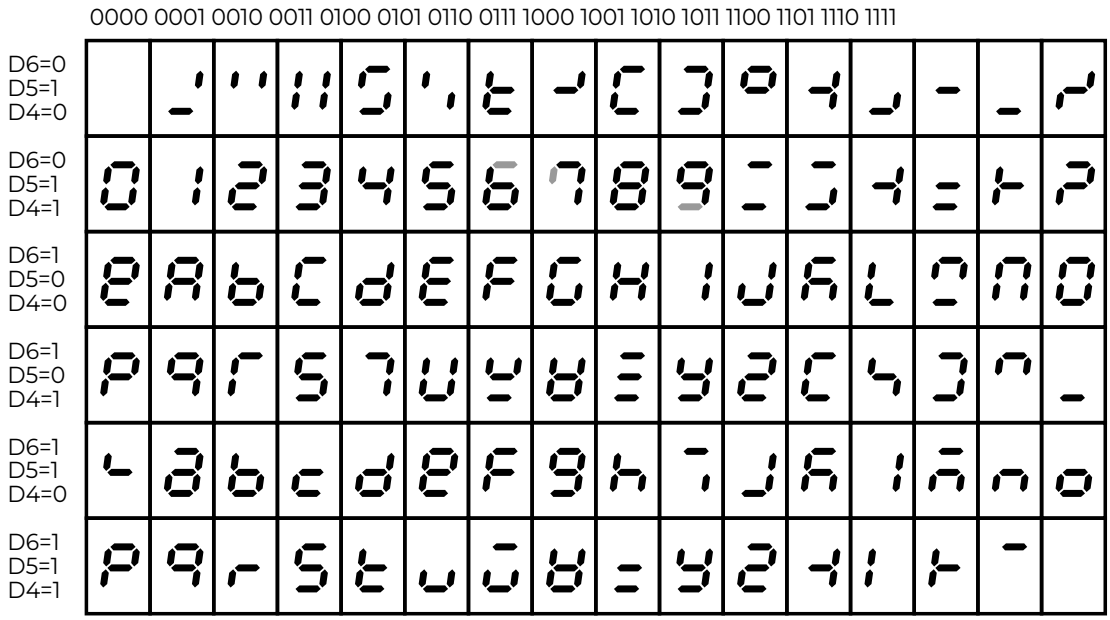
—	Dedicated Input	Dedicated Output	Bidirectional
0	A	Segment a	Input - X6
1	B	Segment b	Input - X7

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

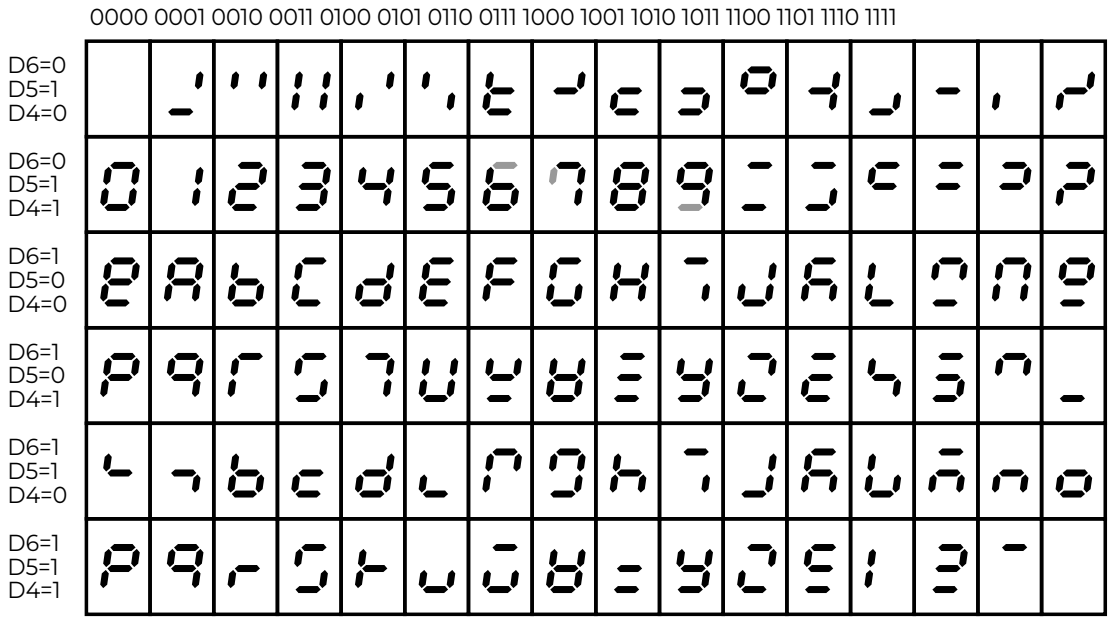
ASCII to seven segment decoder

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).

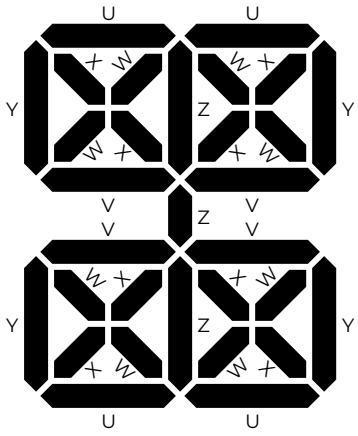
The pin assignments in this mode are:

—	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

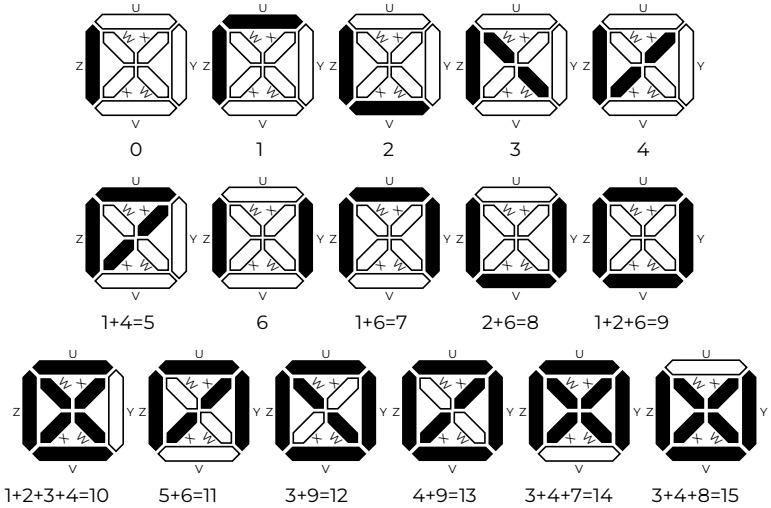
Dual BCD to Cistercian numeral decoder

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.

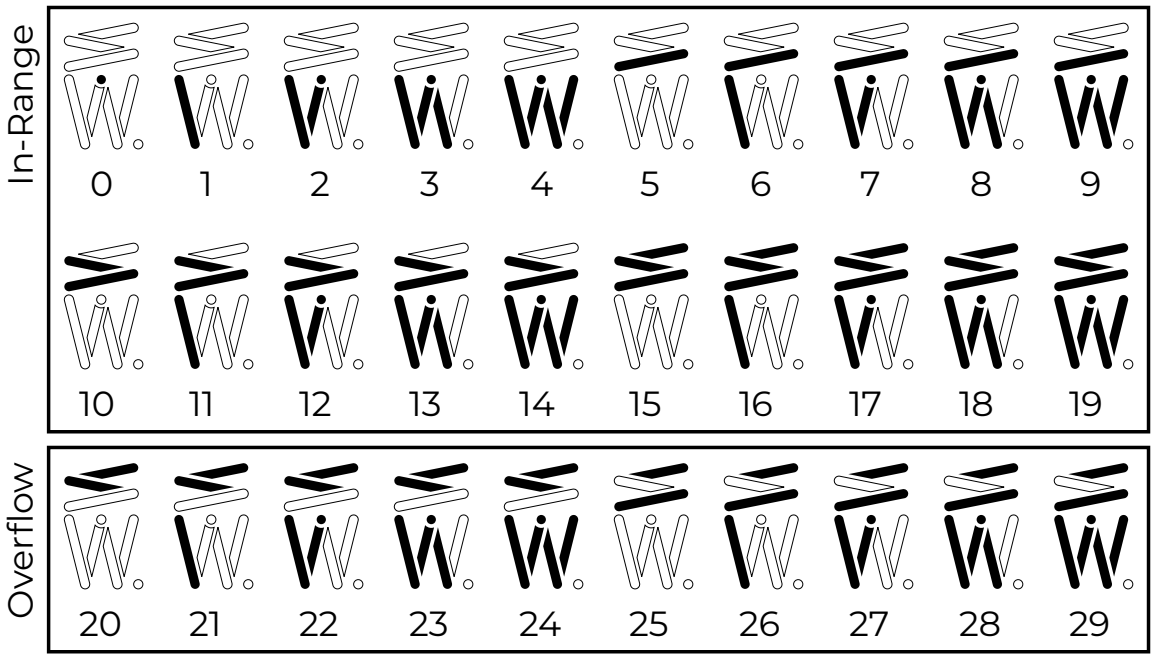
The pin assignments in this mode are:

—	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

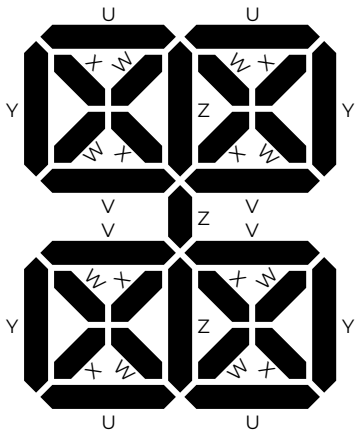
How to test

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)

OMEGA INFINITY KAORU 3D Metal Grid Processor

by **Kaoru Aguilera Katayama**

226 | 10 MHz | HDL Project

github.com/PolloXDDD/ttihp-CRYPTOGRAPHY-DESTROYER-OMEGA-INFINITY

"Physical BEOL 3D metal wire grid mesh coupled to a CMOS Circuit-SAT engine"

How it works

This processor interfaces an integrated 3D BEOL physical wire mesh (`omega_metal_grid_3d.gds`) fabricated in the SkyWater 130nm metallization stack (`met1` through `met5`) directly with standard-cell CMOS threshold inverters.

The metal mesh provides continuous analog potential relaxation across physical conductors. The state of these physical wires is quantized into digital logic levels that feed a runtime-programmable Boolean circuit DAG (Directed Acyclic Graph) to evaluate Circuit-SAT formulations directly in hardware.

How to test

1. Hold `rst_n` low for 4 clock cycles, then bring high.
2. Enable programming mode by asserting `uio_in[0] = 1`.
3. Stream gate operations by providing `prog_op` on `ui[2:0]`, `prog_a` on `ui[7:3]`, `prog_b` on `uio[7:4]`, and toggling `uio[1]` (`prog_we`).
4. Select the target evaluation node by driving `prog_a` with `uio[1] = 0`.
5. Switch to execution mode (`uio_in[0] = 0`) and assert `uio[3] = 1` (`grid_stable`).
6. Read `uo_out[0]` for SAT, `uo_out[1]` for UNSAT, and `uo_out[7:3]` for the evaluated physical tap states.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	prog_op[0]	SAT	prog_en (1=prog, 0=run)
1	prog_op[1]	UNSAT	prog_we (write strobe)
2	prog_op[2]	DONE	unused

#	Input	Output	Bidirectional
3	prog_a[0]	tap_out[0]	grid_stable (relaxation trigger)
4	prog_a[1]	tap_out[1]	prog_b[0]
5	prog_a[2]	tap_out[2]	prog_b[1]
6	prog_a[3]	tap_out[3]	prog_b[2]
7	prog_a[4]	tap_out[4]	prog_b[3]

INTERCAL ALU

by **Rebecca G. Bettencourt**

0227 HDL Project

github.com/RebeccaRGB/ttihp-intercal-alu

"An ALU for the five operators of the INTERCAL programming language."

How it works

As an educational project, it is inevitable that Tiny Tapeout would attract various pedagogical examples of common logic circuits, such as ALUs. While ALUs for common operations such as addition, subtraction, and binary bit-wise logic are surprisingly common, it is much rarer to encounter one that can calculate the five operations of the INTERCAL programming language. Due to either the cost-prohibitive nature of Warmenhovian logic gates or general lack of interest, such a feat has never been performed until now. With chip production finally within reach of the average person, all it takes is one person who has more dollars than sense to design the fabled INTERCAL ALU (Arrhythmic Logic Unit).

The pin assignments for this design are roughly as follows. The /OE (output enable) and /WE (write enable) signals are active low, so should be set HIGH by default.

#	Dedicated Input	Dedicated Output	Bidirectional I/O
0	A0 (address)	D0 (output only)	D0 (input and output only)
1	A1 (address)	D1 (output only)	D1 (input and output only)
2	S0 (selector)	D2 (output only)	D2 (input and output only)
3	S1 (selector)	D3 (output only)	D3 (input and output only)
4	S2 (selector)	D4 (output only)	D4 (input and output only)
5	S3 (selector)	D5 (output only)	D5 (input and output only)
6	/OE (output enable)	D6 (output only)	D6 (input and output only)
7	/WE (write enable)	D7 (output only)	D7 (input and output only)

This ALU has two 32-bit registers, B and A (in no particular order). (These may also be thought of as four 16-bit registers, AL, AH, BL, and BH.) To write a byte to a register, set A0 and A1 to the byte address, set S0 LOW for the A register or HIGH for the B register, set S1 through S3 LOW, set the bidirectional I/O pins to the byte value, set /WE LOW, then set /WE HIGH again. (Do not set S1 through S3 HIGH when writing, or else something unpredictable will happen, most likely nothing.)

To read a register or result, set A0 and A1 to the byte address, set S0 through S3 to the desired operation, set /OE LOW, read the byte value from the bidirectional I/O pins, then set /OE HIGH. Results can also be read from the dedicated outputs; the dedicated outputs are not affected by the /OE signal, as they do not need to care about your feelings.

The operations supported are listed below. An attempt was made to make it understandable.

						Address				
						A	3	2	1	0
Selector						A1	1	1	0	0
S	S3	S2	S1	S0	Operation	A0	1	0	1	0
0	0	0	0	0	A		AH		AL	
1	0	0	0	1	B		BH		BL	
2	0	0	1	0	AND16		& AH		& AL	
3	0	0	1	1	AND32		& A			
4	0	1	0	0	OR16		V AH		V AL	
5	0	1	0	1	OR32		V A			
6	0	1	1	0	XOR16		? AH		? AL	
7	0	1	1	1	XOR32		? A			
8	1	0	0	0	MINGLE16L		AL $ BL			
9	1	0	0	1	MINGLE16H		AH $ BH			
10	1	0	1	0	SELECT16		AH~ BH		AL~ BL	
11	1	0	1	1	SELECT32		A ~ B			

Operations 0 and 1 simply return the current value of the A or B register, respectively. This corresponds with the values of S0 through S3 used in write mode. This is not unintentional. This might also explain why S1 through S3 must be LOW in write mode.

Operations 2 through 7 correspond to INTERCAL's unary AND, unary OR, and unary XOR operators, represented by ampersand (&), book (V), and what (?), respectively. From the INTERCAL manual:

These operators perform their respective logical operations on all pairs of adjacent bits, the result from the first and last bits going into the first bit of the result. The effect is that of rotating the operand one place to the right and ANDing, ORing, or XORing with its initial value. Thus, #&77 (binary = 1001101) is binary 0000000000000100 = 4, #V77 is binary 1000000001101111 = 32879, and #?77 is binary 1000000001101011 = 32875. Operations 2, 4, and 6 work on the 16-bit halves of the A register independently, while operations 3, 5, and 7 work on the 32-bit whole of the A register.

Operations 8 and 9 correspond to INTERCAL's *interleave* (also called *mingle*) operator, represented by big money ($). From the INTERCAL manual:

The interleave operator takes two 16-bit values and produces a 32-bit result by alternating the bits of the operands. Thus, #65535$#0 has the 32-bit binary form 101010....10 or 2863311530 decimal, while #0$#65535 = 0101....01 binary = 1431655765 decimal, and #255$#255 is equivalent to #65535. Operation 8 returns the interleave of the lower halves of A and B, while operation 9 returns the interleave of the upper halves of A and B. (Should the chip fabrication process allow for it, operation 8½ will, of course, return the interleave of the middle halves of A and B.)

Operations 10 and 11 correspond to INTERCAL's *select* operator, represented by sqiggle ( ). From the INTERCAL manual:

The select operator takes from the first operand whichever bits correspond to 1's in the second operand, and packs these bits to the right in the result. Both operands are automatically padded on the left with zeros. [...] For example, #179 #201 (binary value 10110011 11001001) selects from the first argument the 8th, 7th, 4th, and 1st from last bits, namely, 1001, which = 9. But #201 #179 selects from binary 11001001 the 8th, 6th, 5th, 2nd, and 1st from last bits, giving 10001 = 17. #179 #179 has the value 31, while #201 #201 has the value 15. To help understand the select operator, the INTERCAL manual also provides a helpful circuitous diagram.

Use of operations 12 and above is not recommended, unless undefined behavior is required.

How to test

The following example calculations found in the INTERCAL manual should be particularly illuminating.

S	A	B	F
`MINGLE16L` (8)	0	256	65536
`MINGLE16L` (8)	65535	0	2863311530
`MINGLE16L` (8)	0	65535	1431655765

MINGLE16L (8)	255	255	65535
SELECT16 (10)	51	21	5 *
SELECT16 (10)	179	201	9
SELECT16 (10)	201	179	17
SELECT16 (10)	179	179	31
SELECT16 (10)	201	201	15
AND16 (2)	77	—	4
OR16 (4)	77	—	32879
XOR16 (6)	77	—	32875

These test cases are included in the (unfortunately Python and not INTERCAL) `test.py` file. As these are likely more INTERCAL operations than any sensible person will ever perform, they should be sufficient for testing purposes. However, for curiosity's sake, an extensive set of additional test cases have also been included.

* Not found in the INTERCAL manual.

External hardware

The ALU may be used without external hardware, although seeing the output values may present a challenge. Instead, it is recommended to use a microcontroller of some sort to drive the inputs and read the outputs, as microcontrollers are designed to do. The implementation of the rest of the INTERCAL language is left as an exercise for the reader.

Further reading

The INTERCAL Programming Language Revised Reference Manual by Donald R. Woods and James M. Lyon with revisions by Louis Howell and Eric S. Raymond (can recommend highly enough)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A0 (address)	D0	D0
1	A1 (address)	D1	D1
2	S0 (selector)	D2	D2
3	S1 (selector)	D3	D3
4	S2 (selector)	D4	D4
5	S3 (selector)	D5	D5

#	Input	Output	Bidirectional
6	/OE (output enable)	D6	D6
7	/WE (write enable)	D7	D7

MC14500B Extended 1-bit Microcontroller SoC

by **Alexander Co Abad**

228 | 50 MHz | HDL Project

`github.com/alexandercoabad/MC14500B_IHP`

"An advanced 1-bit MCU SoC containing an MC14500B Core, 256-word ROM, 16-bit RAM, and dual IO expansion ports"

How it works

This project transforms the classic 1-bit Motorola MC14500B Industrial Control Unit (ICU) architecture into a complete, independent System on Chip (SoC) micro-computer scaled across a 1x2 tile layout footprint. It is explicitly target-hardened for the **TTIHP26b (IHP 130 nm BiCMOS SG13G2)** silicon shuttle run.

The SoC operates completely autonomously, executing an internal, pre-loaded program code layout without needing external microcontrollers or clock-stretching hardware logic to drive its execution pipeline.

Core Architectural Features:

- **Sub-Core CPU:** Fully independent 1-bit MC14500B CPU running the original 16-opcode Boolean logic Instruction Set Architecture (ISA).
- **Program Counter (PC):** An internal 6-bit sequential stepping address counter register that loops through the 64-word program memory space.
- **On-Chip ROM Program Memory:** 64 Words x 8-bit instruction bus width. Instructions use a split-bus strategy where the upper nibble (`[7:4]`) represents the CPU opcode, and the lower nibble (`[3:0]`) maps the data address operand.
- **On-Chip Data RAM Scratchpad:** 8 independent, single-bit internal static register memory cells (`4'h0` to `4'h7`). Addresses `4'h8` to `4'hF` are **not** general RAM — they are memory-mapped peripherals and reserved space (see below).

Memory & I/O Mapping Matrix:

- **Data Registers `4'h0` to `4'h7`:** General-purpose single-bit read/write internal scratchpad data storage registers.
- **Rising Edge Detector (`4'h8`):** Hardware edge capture module that latches a rising-edge event on `ui_in[0]`, live and independent of CPU stepping. Read the flag at address `8`; write `1` to address `8` to clear it (once per instruction).

- **Programmable Clock Divider (`4'h9`):** Read/write control bit for CPU execution speed. Writing `1` parks the CPU's instruction-fetch pipeline so it advances roughly once every 4096 real clock cycles instead of every cycle; writing `0` returns to full speed.
- **Hard-Wired Zero (`4'hA` to `4'hB`):** Always read as `0`; writes have no effect.
- **Output Latch Array (`4'hC`):** A dedicated 8-bit shift register drives the parallel output bus (`uo_out[7:0]`). Each `STO`/`STOC` to address `4'hC` shifts one new bit in; it is **not** a mirror of the scratchpad RAM.
- **Parallel Input Taps (`4'hD` to `4'hF`):** Only the top three bits of the physical input bus — `ui_in[5]`, `ui_in[6]`, and `ui_in[7]` — are captured, once per CPU instruction step (one cycle of latency), and exposed read-only at addresses `4'hD`, `4'hE`, and `4'hF` respectively. `ui_in[4:0]` are not addressable by the core.
- **Real-Time Signal Monitors:** The bidirectional bus pins (`uio_out`) are configured as outputs for physical logic analyzer probing: `uio_out[5:0]` breaks out the Program Counter, and `uio_out[7]` breaks out the core write-enable strobe. `uio_out[6]` is unused (always `0`).

Writes to the scratchpad and peripheral registers fire exactly once, on the first real clock edge after an instruction becomes current — this is decoupled from the clock divider. So a `STO`/`STOC` that gets parked by the divider for many physical clock cycles still commits its write immediately and then holds steady, rather than repeating on every physical edge or waiting for the divider's next pulse.

How to test

The design can be evaluated via behavioral RTL simulations, gate-level netlists, or directly on the physical hardware breakout board once manufactured.

Behavioral & Gate-Level Simulation (cocotb)

The integrated test framework uses Python-driven `cocotb` test scripts to step through clock events and monitor responses.

1. Navigate your terminal environment into the test suite boundary: `cd test`
2. Run the automated testing routine: `make`

The test harness sets up a stable 50 MHz simulation clock, executes a hardware reset, streams static bit patterns into the dedicated input pins, and verifies that the resulting parallel output states and instruction stepping lines align with the embedded ROM execution sequence.

Manual Hardware Testing

1. **Power-Up Reset Sequence:** Drive the `rst_n` pin low, establish a stable running clock frequency source on the `clk` pin, and then return the `rst_n` pin high to begin the execution sequence.
2. **Signal Stimulus:** Apply static or dynamic digital logic high/low voltages to individual lines across the dedicated parallel input bus (`ui_in`).
3. **Trace Observation:** Monitor the `uo_out` parallel output bus pins using an oscilloscope or logic analyzer. Watch the output registers change state as the internal MCU loops through its ROM program, executes bitwise operations, and stores results back into the parallel latch array.

External hardware

This SoC is designed to be self-contained for standard verification loops, but can easily interface with basic external digital hardware components:

- **Logic Analyzer / Oscilloscope:** Connect to the bidirectional bus pins (`uio[7:0]`) to capture physical trace files of the internal execution loop, track program counter stepping, and verify timing margins.
- **Parallel Input Switches:** Connect a standard 8-pin DIP switch module or sensor array to the dedicated inputs (`ui[7:0]`) to feed parallel runtime data into the internal register mapping space.
- **LED Driver Array:** Connect low-power LED diagnostic indicators to the dedicated output port pins (`uo[7:0]`) to display register processing states in real time.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ext_in[0]	ext_out[0]	pc_out[0]
1	ext_in[1]	ext_out[1]	pc_out[1]
2	ext_in[2]	ext_out[2]	pc_out[2]
3	ext_in[3]	ext_out[3]	pc_out[3]
4	ext_in[4]	ext_out[4]	pc_out[4]
5	ext_in[5]	ext_out[5]	pc_out[5]
6	ext_in[6]	ext_out[6]	rr_out
7	ext_in[7]	ext_out[7]	write_pulse

IEEE Flappy Bird VGA Game

by **Escobar Mamani Alejandra Heidi**

`0229` `25.175 MHz` `HDL Project`

`github.com/alejandraheidiescobar-ops/FLAPPYBIRD`

"Flappy Bird hardware game with parallax scrolling for TinyVGA PMOD"

How it works

This project implements a hardware-driven Flappy Bird game that outputs real-time 640x480 @ 60 Hz VGA graphics using procedural combinational logic without any frame buffers or external RAM.

How to test

Test the design by running tt_um_flappy_bird at 25.175 MHz on VGA Playground, executing the Cocotb testbench locally with make, or deploying on Tiny Tapeout hardware using a TinyVGA PMOD and push button.

External hardware

Tiny VGA Pmod driving a VGA monitor.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Jump Button P1	R1	—
1	Jump Button P2	G1	—
2	—	B1	—
3	—	VSync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	—	HSync	—

stapel gerät

by **Nicolas Tscherter**

`0230` `50 MHz` `HDL Project`

`github.com/TscherterJunior/stapel-geraet`

“stack based cpu”

How it works

wip

How to test

wip

External hardware

Just the rasby chip

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	EXT_IN_0	ADDR_8	MEM_BUS_0
1	EXT_IN_1	ADDR_9	MEM_BUS_1
2	EXT_IN_2	ADDR_10	MEM_BUS_2
3	EXT_IN_3	ADDR_11	MEM_BUS_3
4	EXT_IN_4	ADDR_12	MEM_BUS_4
5	EXT_IN_5	ADDR_13	MEM_BUS_5
6	EXT_IN_6	ERROR	MEM_BUS_6
7	EXT_IN_7	MEM_WE	MEM_BUS_7

ALU CASS PUCV

by **CASS PUCV**

`0231` `10 MHz` `HDL Project`

`github.com/MaHt-275/ALU_tt`

> *"ALU de 8 bits con interfaz serial de 24 pines (FSM de 3 estados); soporta operaciones aritméticas, lógicas y de desplazamiento vía opcode de control."*

Este proyecto implementa una ALU de 8 bits. Debido a la restricción de 24 pines del formato Tiny Tapeout, la ALU utiliza una arquitectura de carga serial multiplexada en el tiempo, controlada por una máquina de estados finitos (FSM) de 3 estados, que permite cargar operandos y opcode de forma secuencial antes de ejecutar la operación.

How it works

La ALU decodifica un opcode de 8 bits según un esquema tipo ISA, donde los dos bits más significativos determinan el modo de operación:

- `00`: desplazamiento (barrel shifter) — izquierda, derecha lógico o derecha aritmético, con magnitud de 0 a 7 bits.
- `01`: aritmética — suma o resta en complemento a dos, con soporte de carry_in.
- `10`: lógica — AND, OR, XOR o NOT.
- `11`: bypass — pasa `in_a` directamente a la salida.

Internamente, `alu_top.v` instancia tres módulos combinacionales en paralelo (`arithmetic.v`, `Logico_8bits.v` y `barrel_shifter.v`); sus resultados y banderas (carry, overflow) llegan a un multiplexor que selecciona la salida final según `select[1:0]`. Las banderas `zero_flag` y `negative_flag` se calculan directamente sobre la salida `out`, independientemente del modo activo (en modo lógico, `negative_flag` no tiene significado de signo).

Nota de diseño relevante: en modo resta con `carry_in = 0`, el resultado es `A - B - 1` y no `A - B`, ya que `arithmetic.v` opera en complemento a uno para la resta.

How to test

Para probar este diseño, se deben asignar los valores de entrada a través de los pines correspondientes y observar los resultados en los pines de salida. (Agreguen un par de líneas sobre qué estímulos aplican en los testbench).

INSTRUCCIONES ALU el control es una palabra de 8 bits los primeros 2 bit mas significaticos estan reservados para definir que operacion donde: 00: Operaciones de Desplazamiento (Shift) 01: Operaciones Aritméticas 10: Operaciones Lógicas 11: Paso Directo (Bypass)

intrucciones por Type

Type 00 Desplazamiento: estructura de Señal control: 00DASSSX 00 indica type

D direccion de desplazamiento: 0 izquierda. 1 derecha.

A Indica si el desplazamiento a la derecha es o no aritmetico: 0 logico. 1 aritmetico.

SSS indica en binario la cantidad a desplazar 000 No desplaza 001 Desplaza 1 010 Desplaza 2 011 Desplaza 3 100 Desplaza 4 101 Desplaza 5 110 Desplaza 6 111 Desplaza 7

X valor sobrante

Type 01 Aritmerica estructura de Señal control: 01TCXXXX. 01 indica Type.

T indica si suma o resta. 0 Suma. 1 resta(sin Carry-in).

C Indica si Carry-in. 0 Falso. 1 Verdadero.

X valor sobrante.

Type 10 Logico

estructura de Señal control:10LLXXXX 10 indica Type

LL Indica operacion a realizar 00 A and B 01 A or B 10 A Xor B 11 Not A

X valor sobrante

Type 11 Bypass
estructura de Señal control: 11XXXXXX
Este pasa directo el valor del resgistro A hacia la salida

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	in[0] - bus serie (instruccion/A/B)	out[0] - resultado ALU	carry_flag
1	in[1] - bus serie (instruccion/A/B)	out[1] - resultado ALU	negative_flag

#	Input	Output	Bidirectional
2	in[2] - bus serie (instruccion/A/B)	out[2] - resultado ALU	overflow_flag
3	in[3] - bus serie (instruccion/A/B)	out[3] - resultado ALU	zeros_flag
4	in[4] - bus serie (instruccion/A/B)	out[4] - resultado ALU	—
5	in[5] - bus serie (instruccion/A/B)	out[5] - resultado ALU	—
6	in[6] - bus serie (instruccion/A/B)	out[6] - resultado ALU	—
7	in[7] - bus serie (instruccion/A/B)	out[7] - resultado ALU	—

Ring oscillator frequency meter

by **Maxwell Pauly**

0232 | 50 MHz | HDL Project

github.com/mgpauly1458/ttihp26b-project

> "*Reciprocal frequency counter that measures any of eight on-chip ring oscillators (incl. a trimmable one) and reports the count over the pins*"

How it works

A reciprocal frequency counter for eight on-chip ring oscillators: it counts reference-clock (`clk`) cycles inside exactly `TARGET_N` periods of the selected ring, divided by `2^TAP_SEL` (1..128).

```
f_ring = TARGET_N * 2^TAP_SEL * f_ref / RESULT
```

Only `f_ref` must be accurate, and it is a clock on a digital input. The divider, counter, window state machine and register file are shared, so a difference between two rings is a real difference. One 8-bit trim code is broadcast to all rings.

slot	ring
0..3	21-stage minimum-drive, identical netlists
4	21-stage high-drive
5	11-stage minimum-drive
6	tap-select trimmed, 5..19 stages by `code[2:0]`
7	analog hard macro: 11-stage current-starved ring, 8-bit binary-weighted current DAC; 2 MHz at code 0 to 164 MHz at code 255 typical, monotonic, about 3x over PVT

How to test

`ui_in[2:0]` write address, `ui_in[3]` write strobe, `uio_in[7:0]` write data, `ui_in[6:4]` selects the byte on `uo_out`. Drive `clk` from an accurate source; 50 MHz assumed below.

1. Reset. Set `ui_in[6:4]` = 7; `uo_out` must read `0xA5`.
2. Write (address and data on the pins, `ui_in[3]` high for at least 4 clocks): addr 1 `RING_SEL` = 0, addr 2 `TAP_SEL` = 3, addr 3/4 `TARGET_N` = 200 (0xC8, 0x00), addr 5/6 `TIMEOUT` = 16000 (0x80, 0x3E).
3. Write 1 to addr 7 (`START`).
4. Read `STATUS` (`ui_in[6:4]` = 0) until bit 0 `done`. Bit 2 `timeout_error` means the ring did not run.

5. Read `RESULT` bytes with `ui_in[6:4]` = 1..4. A 350 MHz ring gives about 229; `f = 200 * 8 * 50 MHz / RESULT`.
6. Repeat over `RING_SEL` 0..7 and `TRIM_CODE` (addr 0) 0..255.
7. Optional, statistics: repeat step 3..4 N times without changing any register, then set `ui_in[7]` = 1, put an index on `uio_in[4:0]`, and read `uo_out`: 0..1 `COUNT`, 2..3 `MIN`, 4..5 `MAX`, 6..9 `SUM` of `RESULT` (little-endian) since the last register write. `SUM / COUNT` is the average, `MAX − MIN` the peak-to-peak spread. Index 10 is a random byte, 11 reads `0x5A`, 12..25 spell a message.

Rules:

- The divided ring must stay below 250 MHz. `TAP_SEL` = 3 is safe for every ring at every corner; 2 suffices below 1 GHz.
- Slot 7 at code 0 is about 2 MHz: 200 periods through tap 3 take 800 us, longer than 16000 cycles at 50 MHz. Use tap 0 or 1 for low codes, or raise `TIMEOUT` (max 65535 cycles, 1.3 ms).
- Change the code only between measurements.

Register map, pin map and sweep script: `docs/` and `scripts/` in the repository.

External hardware

An accurate clock on `clk` (the demo board's for bring-up, a lab source for characterisation). A microcontroller or the demo board's RP2040 to run the sweep.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ADDR[0] - write address	RDATA[0] - byte selected by RSEL	WDATA[0] - write data (input)
1	ADDR[1] - write address	RDATA[1] - byte selected by RSEL	WDATA[1] - write data (input)
2	ADDR[2] - write address	RDATA[2] - byte selected by RSEL	WDATA[2] - write data (input)
3	WE - write strobe, rising edge writes WDATA to ADDR	RDATA[3] - byte selected by RSEL	WDATA[3] - write data (input)
4	RSEL[0] - read select	RDATA[4] - byte selected by RSEL	WDATA[4] - write data (input)
5	RSEL[1] - read select	RDATA[5] - byte selected by RSEL	WDATA[5] - write data (input)

#	Input	Output	Bidirectional
6	RSEL[2] - read select	RDATA[6] - byte selected by RSEL	WDATA[6] - write data (input)
7	STATS - 1 shows the statistics byte indexed by uio[4:0] on uo	RDATA[7] - byte selected by RSEL	WDATA[7] - write data (input)

S4xU4

by **Yann Guidon**

0233 200 MHz HDL Project

`github.com/ygdes/S4xU4`

"4×4 bit multiplier with Tiny Tapeout in Verilog on IHP SG13G2"

How it works

Two 4-bit operands are latched by DFFs on rising edges:

- Clk latches the Unsigned operand
- Sen latches the signed operand

The product is available on the next clock cycle, as a signed 8-bit byte at the Sum output.

The MSB and LSB are trivial to get:

- LSB is `S[0] & U[0]` (because only "odd times odd" can make "odd").
- MSB is S[3] unless U=0000.

The 6 other bits require the data to go through 3 stages:

Complement

S and U are latched, then complemented (`(x-1)^s`) with the sign bit of S (unless S=1000). The result is buffered to drive the following stage.

Replication

This is the typical "shift & and" circuit that performs the typical binary multiplication. Except that the 2 most significant bits can be merged because S=8 (`U<<3`) can not happen when S > 8 (`U<<2`), thus saving some gates. It's an A22OI instead of a AND.

Addition

The 3 partial results are compressed by a 3->2 layer then go through a classic radix-3 CLA adder.

Et voilà.

Extra bonus feature

Since the multiplier is so small, I put 3 of them on the tile and the 3 products are added by the same type of 3-input adder. This turns the tile into a MAC, except that the inputs must be fed sequentially, but "imagine" if it was part of a larger system!

Each set of S and U input is selected separately, using six “enable” signals. Inputs and outputs are latched so allow 2 cycles to see the result.

How to test

I have pushed the synthesiser to 200MHz to see if it could reach it, but the real performance is limited by the IO pins and their low number. There is little to test beyond functionality, it’s a proof of concept.

If you really want to use it, follow the algo in `test/test.py`: update the S and U inputs, raise the respective latch-enable signals and observe the result at the Sum output port after 2 clock cycles.

External hardware

Nothing fancy, a microcontroller will do the job.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Signed operand bit 0	Sum bit 0	Sen0
1	Signed operand bit 1	Sum bit 1	Sen1
2	Signed operand bit 2	Sum bit 2	Sen2
3	Signed operand bit 3	Sum bit 3	Uen0
4	Unsigned operand bit 0	Sum bit 4	Uen1
5	Unsigned operand bit 1	Sum bit 5	Uen2
6	Unsigned operand bit 2	Sum bit 6	Sum bit 8
7	Unsigned operand bit 3	Sum bit 7	Sum bit 9

IEEE Workshop Simple CPU

by **Hadjer Bouyahiaoui** & **Maria Debbaghi**

`0234` `5 MHz` `HDL Project`

`github.com/OpenSilicon-workshop/IEEE_Tiny_CPU`

> *"8-bit accumulator CPU using a real IHP 256x8 SRAM macro for data memory; multi-cycle to tolerate the macro's registered read latency"*

How it works

This project implements a small **8-bit accumulator CPU** designed as an educational Tiny Tapeout project.

The CPU contains:

- A **4-bit program counter (PC)**, allowing up to 16 program instructions.
- A **16 × 8-bit program ROM** implemented as synthesizable Verilog `case` logic.
- A **16 × 8-bit data RAM** used by load, store, arithmetic, and logic instructions.
- An **8-bit accumulator (ACC)** used as the main working register.
- An **ALU** supporting addition, subtraction, AND, OR, pass-through, and decrement.
- A **zero flag (Z)** used by the conditional `JZ` instruction.
- An 8-bit input port and an 8-bit output port.
- A `halted` status output for observing when the CPU executes `HLT`.

The CPU uses an 8-bit instruction format:

```text 7 4 3 0 +———+———+ | opcode | operand | +———+———+ 4 bits 4 bits
```

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	gpio_in[0]	gpio_out[0]	halted (output)
1	gpio_in[1]	gpio_out[1]	uart_tx (output)
2	gpio_in[2]	gpio_out[2]	—
3	gpio_in[3]	gpio_out[3]	—
4	gpio_in[4]	gpio_out[4]	—
5	gpio_in[5]	gpio_out[5]	—
6	gpio_in[6]	gpio_out[6]	—

#	Input	Output	Bidirectional
7	gpio_in[7]	gpio_out[7]	—

Space CA

by **Alexander Mordvintsev**

`235` `25.175 MHz` `HDL Project`

`github.com/znah/ttihp26b_space_ca`

"VGA demo: Giant Computer in Space"

How it works

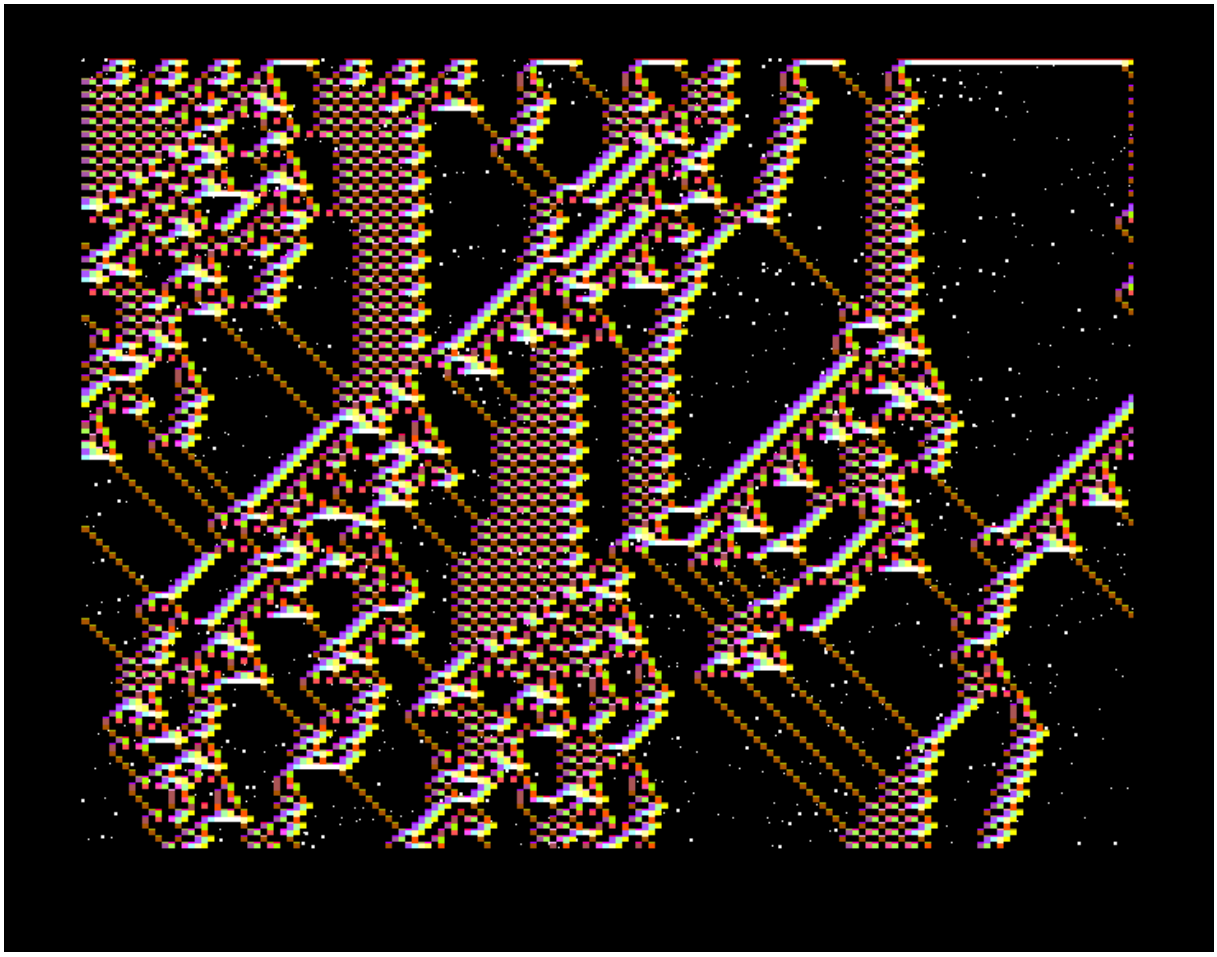


Figure 235.1: Space CA VGA Output

Space CA is a real-time hardware VGA demo ("Giant Computer in Space") that simulates and visualizes a 1D cellular automaton evolving over time against a procedural background starfield.

- **VGA Generator**: Generates standard 640×480 @ 60 Hz timing from a 25.175 MHz pixel clock.
- **Cellular Automaton**:
  - Grid width: 160 cells across the screen (`GRID_W = 160`), rendered as 4×4 pixel blocks (`CELL_SIZE = 4`).

    - Rule: 5-neighborhood toroidal elementary cellular automaton (`RULE = 32'h6C1E53A8`).
    - Evolution: Rows are computed line-by-line as the beam scans downwards, showing space horizontally and time vertically (120 generations per frame).
- **Coloring & Effects**:
    - Active cells are colored using their 5-cell neighborhood window state (`{1'b1, window}`) mapped to 6-bit color (R2G2B2).
    - Procedural background starfield synthesized using integer hashing hardware (`starfield` module).
- **Memory**:
    - Uses two latch banks (`cells` and `first_row_cells`) synthesized with IHP `sg13g2_dlhq_1` latches to maintain the 160-bit state between scanlines and frame boundaries without consuming standard flip-flop area.
- **Interactive Controls**:
    - `ui_in[0]` (`advance_mode`): Toggles between smooth 1-row vertical scrolling and jumping by a full frame.
    - `ui_in[1]` (`inject_gliders`): Injects gliders into cells 80..83 of row 0.

How to test

1. Connect a standard Tiny Tapeout digital VGA PMOD to the output pins (`uo_out`).
2. Provide a 25.175 MHz clock to `clk`.
3. Set `ena = 1` and release reset by setting `rst_n = 1`.
4. Connect to a VGA display (640×480 @ 60 Hz).
5. Use input switches to interact:
    - `ui_in[0]`: Toggle advance mode (0 = scroll 1 row/frame, 1 = full frame jump).
    - `ui_in[1]`: Toggle glider injection.

External hardware

- Tiny Tapeout VGA PMOD (2 bits per color channel R2G2B2 + HSync + VSync)
- VGA monitor supporting 640×480 @ 60 Hz

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	advance_mode	R1	—
1	inject_gliders	G1	—

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

Endless Runner

by **Bianca Manganaan**

`0257` `25.175 MHz` `HDL Project`

`github.com/Biancamgn/endless-runner`

"VGA endless driving game with a procedurally generated road, obstacles and scenery"

How it works

A top-down endless driving game rendered at 640x480 @ 60Hz over VGA.

Nothing about the world is stored anywhere. The road centre at screen row `y` is a pure function of `w = scroll - y`:

```
centre(w) = T(w) - T(3w)/8 - T(5w)/8 + slow_sweep(w)
```

where `T` is a triangle wave taken straight from counter bits. Subtracting the third and fifth harmonics rounds off the corners a plain triangle wave would give, and both are shift-adds (`3w = w + 2w`, `5w = w + 4w`), so the smoothing is close to free.

Obstacles and scenery come from a 10-bit xorshift hash of position. The road is divided into bands of 64 world units; the hash of the band index decides whether an obstacle is present and which of eight lanes it sits in. Trees, bushes and rocks are scattered on a 32x32 grid using a second hash seed. Because both are pure functions of position, the course is infinite, identical on every run, and costs no memory.

The score is kept as three BCD digits rather than a binary counter, which avoids needing a divide-by-ten to display it. The digits are drawn as seven rectangles each, using the same segment decoder a 7-segment display would.

Difficulty is derived, not stored: both the scroll speed and the road half-width are functions of `scroll`, so the road narrows and the game speeds up with distance at no extra cost in state.

Leaving the road or striking an obstacle flashes the screen red and restarts. Collision is checked once per frame by latching the road edges and the obstacle extent at the car's scanline.

How to test

Try it in the browser first at https://vga-playground.com - paste the contents of `src/project.v` into the editor.

- `ui[0]` steers left, `ui[1]` steers right, `ui[2]` restarts
- Stay between the red and white kerbs
- Avoid the orange hazards in the road
- The road narrows and the speed rises as the score climbs

The cocotb testbench in `test/` checks the VGA timing (800 clocks per scanline, 525 scanlines per frame), that the blanking intervals are black, and that a visible scanline actually contains road, grass and kerb pixels.

External hardware

- TinyVGA Pmod on the dedicated output pins
- Three momentary pushbuttons on `ui[2:0]`

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	STEER_LEFT	R1	—
1	STEER_RIGHT	G1	—
2	RESTART	B1	—
3	—	vsync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	—	hsync	—

KIBO Leak-Inspection Target Controller (VGA)

by **Your Name**

259 | 25.175 MHz | HDL Project

`github.com/ccaeljan-lang/KIBOLeak-InspectionTargetController`

"8-state leak-inspection approach/align/inspect FSM with a live 640x480 VGA status display and a latched safety abort."

How it works

An FSM controls an inspection robot (modelled on Int-Ball2 / Kibo-RPC) as it approaches a suspected leak-inspection target, and a VGA generator draws its live status on a 640x480 @ 60 Hz display (Tiny VGA PMOD on `uo`, 25.175 MHz clock).

States (also on `uio[2:0]`):

Code	State	Behaviour
000	IDLE	Waits for `target_detected`
001	SEARCH	Waits for `target_detected` to confirm a target
010	APPROACH	`move_cmd` = FORWARD until `target_reached`
011	ALIGN	Corrects position and orientation until both are aligned
100	HOLD	Holds for 3 FSM steps
101	INSPECT	`inspect_enable` = 1 until `inspection_done`
110	COMPLETE	`target_locked` = 1; `new_target` -> SEARCH, else IDLE
111	ABORT	`abort_cmd` = 1, all motion stopped, latched until reset

The FSM takes one step every 16 video frames (about 0.27 s) so it is easy to watch. `system_fault` or `obstacle_detected` is checked on every clock and sends the FSM to ABORT immediately from any state, ahead of every other input. Inputs are passed through a 2-flip-flop synchroniser.

Display layout

Each row is 64 px tall, cells are 64 px wide starting at x = 64. Lit cells are coloured, unlit cells are dark grey.

Row	Content
0	Banner in the current state's colour (blinks dark/red in ABORT)

1	8 state boxes, left to right: IDLE, SEARCH, APPROACH, ALIGN, HOLD, INSPECT, COMPLETE, ABORT
2	The 8 input flags, left to right: `ui[0]`..`ui[7]` (obstacle and fault light red)
3	`move_cmd`, left to right: STOP (white), FORWARD, BACKWARD, LEFT, RIGHT, UP, DOWN
4	`align_cmd`, left to right: none, rotate left, rotate right, aligned (green)
5	Lamps: inspect_enable (magenta), target_locked (green), abort_cmd (red)
6	Mission progress bar (cyan), green when COMPLETE, solid red in ABORT

State colours: IDLE grey, SEARCH light blue, APPROACH blue, ALIGN yellow, HOLD orange, INSPECT magenta, COMPLETE green, ABORT red.

`move_cmd` and `align_cmd` appear only on the display. The other command outputs are on `uio[5:3]`.

How to test

1. Connect a Tiny VGA PMOD to the `uo` connector and a monitor. Set the clock to 25.175 MHz.
2. Pulse `rst_n` low then high. The IDLE box lights.
3. Set `ui[0]` (target_detected): the FSM moves to APPROACH.
4. Set `ui[1]`: ALIGN. Set `ui[2]` and `ui[3]`: HOLD, then INSPECT.
5. Set `ui[4]`: COMPLETE, then IDLE (or SEARCH if `ui[7]` is set).
6. Set `ui[5]` or `ui[6]` at any time to see ABORT. Only reset clears it.

For simulation, set `uio[7]` high so the FSM steps every clock instead of every 16 frames.

External hardware

Tiny VGA PMOD (or any 2-bit-per-channel VGA DAC with the same pin order) and a VGA monitor. Switches on `ui` and, optionally, LEDs on `uio[5:0]`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	target_detected	R1	state[0]
`1`	target_reached	G1	state[1]

#	Input	Output	Bidirectional
2	position_aligned	B1	state[2]
3	orientation_aligned	vsync	inspect_enable
4	inspection_done	R0	abort_cmd
5	obstacle_detected	G0	target_locked
6	system_fault	B0	—
7	new_target	hsync	fast_step (1 = step every clock, for simulation)

TinyQV SoC (Dual Memory Backend)

by **LKhanh**

`0260` `64 MHz` `HDL Project`

`github.com/DivineMK/ttihp26b-tapeout`

"TinyQV RISC-V SoC supporting both QSPI PMOD and RP2040 spi-ram-emu"

A TinyQV RV32EC microcontroller with two memory backends selected by a reset strap: Quad-SPI Flash/PSRAM via a QSPI PMOD, or single-SPI RAM emulated on an RP2040 (`spi-ram-emu`, 23LC512 protocol). System clock 64 MHz. Occupies a 3x2 tile.

How it works

CPU

TinyQV is a 4-bit-serial RV32EC core (M-mode only) with Zcb/Zicond, a 32x16-bit multiplier (`mul16`), multi-word load/store (`lw2/lw4`, `sw2/sw4/sw4n`), and basic trap/interrupt support (trap vector 0x4, interrupt vector 0x8). `gp` is hardwired to `0x01000400`, `tp` to `0x80000000` for fast access to RAM data and peripheral registers. `TIME` ticks in microseconds at 64 MHz.

Dual memory backend

`ui_in[0]` sampled during reset selects the backend (`uo_out[1]` mirrors the sampled value):

`ui_in[0]`	Mode	Backend
0 (default)	Single-SPI	RP2040 `spi-ram-emu` (23LC512: `0x03` read, `0x02` write)
1	QSPI PMOD	W25Q128 Flash (continuous read) + 2x APS6404 PSRAM (QPI)

Single-SPI mode runs the bus at 8 MHz (within the emulator's 12.5–15.6 MHz ceiling) and maps the 25-bit CPU address into the emulator's 64 KB as `{addr[24], addr[14:0]}`: instruction fetch lands in the lower 32 KB (`0x0000–0x7FFF`), RAM data (e.g. via `gp`) in the upper 32 KB (`0x8000–0xFFFF`, so `gp` = `0x8400`). A 48-cycle CS-high cooldown between transactions gives the RP2040 PIO time to re-arm. QSPI mode uses the untouched upstream controller (32 MHz bus): Flash must already be in continuous-read mode (mode bits `0xA0`), PSRAM in QPI mode; code executes from Flash only.

Address map

Range	Target
`0x0000000-0x0FFFFFF`	Flash (QSPI) / code window (SPI-emu low 32 KB)
`0x1000000-0x17FFFFF`	RAM A (QSPI) / data window (SPI-emu `0x8000–0xFFFF`)
`0x1800000-0x1FFFFFF`	RAM B (QSPI)
`0x8000000`	GPIO output register
`0x8000010`	UART TX data (write byte to send; read returns busy flag)

Pinout

Pin	Function
`ui[0]`	Backend strap (0 = SPI-emu, 1 = QSPI)
`ui[1:2]`	Spare
`ui[3:6]`	External interrupts `irq[0:3]`
`ui[7]`	Timer interrupt
`uo[0]`	UART TX (115200 baud)
`uo[1]`	Sampled strap mode
`uo[2:7]`	GPIO output bits
`uio[0]`	CS0 (Flash CS / SPI CS)
`uio[1]`	SD0 / MOSI
`uio[2]`	SD1 / MISO
`uio[3]`	SCK
`uio[4]`	SD2 (QSPI only)
`uio[5]`	SD3 (QSPI only)
`uio[6]`	CS1 (RAM A, QSPI only)
`uio[7]`	CS2 (RAM B, QSPI only)

How to test

RTL simulation (Icarus + cocotb):

```
cd test && make clean && make        # 2 tests, gate: no <failure>
in results.xml
GATES=yes make                        # functional gate-level sim
on the hardened netlist
```

`test_spi_ram_emu_mode` boots the CPU against a cycle-accurate 23LC512 model (including the CS-recovery check), runs `addi/sw/lw/sw/jal`, and

asserts `0x42` reaches both SPI RAM (`0x8400`) and the GPIO pins. `test_qspi_pmod_mode` boots the QSPI backend against Flash/PSRAM models and asserts `0x55` reaches GPIO. Both run at the true 64 MHz tapeout clock. Hardening (LibreLane 3.0.5, IHP SG13G2): `./tt/tt_tool.py --create-user-config --ihp`, then `--harden --ihp`. Timing closes with +3.02 ns setup / +0.11 ns hold slack; LVS/DRC/antenna clean.

External hardware

- QSPI PMOD (mole99 design): W25Q128JVSIQ 16 MB Flash + 2x APS6404L-3SQR-SN 8 MB PSRAM, wired per the `uio` table above.
- RP2040 board running `spi-ram-emu` for single-SPI mode (4 wires: CS/MOSI/MISO/SCK to `uio[0:3]`), plus a USB-UART tap on `uo[0]` for console output.

Post-tapeout bring-up (pending tasks)

- ☐ **RP2040 emulator firmware**: fork `spi-ram-emu` to preload the RISC-V binary into `emu_ram[0..0x7FFF]` (code window) before calling `setup_simulated_sram()`; hold the TinyQV board in reset until the preload is done, then release. Demo programs must fit 32 KB code with data addressed via `gp` in the upper 32 KB.
- ☐ **RP2040 wiring/clock**: connect CS/MOSI/MISO/SCK to `uio[0:3]`, tie `ui_in[0]` low, confirm the SPI clock stays within the emulator's ceiling (RTL divides to 8 MHz at 64 MHz system clock; the 48-cycle CS cooldown is already enforced in RTL).
- ☐ **QSPI PMOD provisioning**: pre-program the Flash image AND its continuous-read mode bits (`0xA0`) before power-on (TinyQV cannot enter the mode itself); issue the PSRAM QPI-mode entry sequence at board power-up; tie `ui_in[0]` high.
- ☐ **Board console**: 115200 baud tap on `uo[0]`; GPIO observation on `uo_out[7:2]`; verify the `uo_out[1]` strap indicator matches the intended backend before debugging anything else.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	strap_mode (0=SPI emu, 1=QSPI PMOD)	uart_tx	CS0 / SPI_CS
1	spare	strap_mode_out	SD0 / MOSI
2	spare	gpio[2]	SD1 / MISO
3	irq[0]	gpio[3]	SCK

#	Input	Output	Bidirectional
4	irq[1]	gpio[4]	SD2
5	irq[2]	gpio[5]	SD3
6	irq[3]	gpio[6]	CS1 (RAM A)
7	timer_irq	gpio[7]	CS2 (RAM B)

SEQ8 Programmable Sequencer

by **JET**

`0261` `10 MHz` `HDL Project`

`github.com/Jepwetics/SEQ8B`

> "*8-instruction programmable sequencer driving 8 output pins*"

How it works

SEQ8 is a very small programmable sequencer: an 8-instruction state machine that drives 8 output pins.

You shift a program into it over a simple SPI-like port, raise the RUN pin, and it starts executing from address 0. Each instruction either writes a new pattern to the output pins, pauses for a programmable length of time, or jumps, optionally depending on one of two input pins.

The behaviour is not baked into the silicon. One chip can be a traffic light, a stepper-motor driver, an LED chaser, a test pattern generator, or a servo pulse source, depending only on the 8 words you load into it.

Blocks

- **Program memory:** 8 words x 10 bits, written by the loader, read by the program counter.
- **Loader:** a shift register clocked by SCK while CS_N is low. Every 10 bits it writes one instruction and auto-increments the write address.
- **Tick generator:** a 12-bit counter that makes one "tick" every 1, 16, 256 or 4096 clock cycles, chosen by pins `ui[7:6]`. WAIT counts ticks.
- **Core:** a 3-bit program counter, an 8-bit wait counter and the 8-bit output register.

Instruction set

Every instruction is 10 bits: a 2-bit opcode and an 8-bit operand.

Opcode	Name	Operand	Effect
`00`	OUT	8-bit value	drive `uo_out`
`01`	WAIT	count	pause for (count + 1) ticks
`10`	JMP	`cond[7:6]`, `addr[2:0]`	jump; see below
`11`	NOP	-	do nothing

Jump conditions (operand bits 7:6):

cond	Behaviour

00	always jump to addr
01	jump to addr if IN0 (ui[4]) is high, otherwise continue
10	jump to addr if IN1 (ui[5]) is high, otherwise continue
11	HALT: stop and hold the current outputs

The program counter is 3 bits, so it wraps from address 7 back to 0 by itself. A program that fills all 8 words loops with no JMP at all. A shorter program must end with a JMP or HALT, otherwise it runs on into memory that was never loaded.

Tick speed

ui[7:6]	Clock cycles per tick
00	1
01	16
10	256
11	4096

For example, with ui[7:6] = 11 and a 4096 Hz clock, one tick is one second, so WAIT 9 lasts 10 seconds.

Loading a program

1. Hold RUN low. The core stays in reset while you load.
2. Pull CS_N low. The write address resets to 0.
3. For each instruction, shift 10 bits on the rising edge of SCK, most significant bit first. Instructions are written back to back starting at address 0.
4. Raise CS_N.
5. Set ui[7:6] for the tick speed, then raise RUN. Execution starts at address 0.

The chip samples SCK, MOSI and CS_N with its own clock, so keep every pin steady for at least 4 clock cycles per step. In practice, load the program with a fast clock (for example 1 MHz), then switch to the slow clock you want for running.

Dropping RUN low at any time clears the outputs and rewinds to address 0, so you can restart a program without reloading it.

How to test

The easiest way is with the RP2350 on the demo board driving the load pins in MicroPython. Shift in a program, raise RUN, and watch the output pins.

A worked example, a two-direction traffic light, is in `tools/traffic.s` along with an assembler (`tools/seq16asm.py`) that turns the source into the words you shift in.

For a quick manual check without any software, load this two instruction program and confirm that all eight `uo_out` pins go high: OUT `0xFF` (`0x0FF`), HALT (`0x2C0`).

The cocotb test suite in `test/` covers reset behaviour, the loader, WAIT timing, the tick speed select, both conditional jumps, program counter wraparound and a full traffic-light sequence.

External hardware

None required. LEDs with series resistors on the output pins make the behaviour visible. The design can also drive a 7-segment display, motor driver inputs, or anything else that takes logic-level signals.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SCK - program load clock	OUT0	—
1	MOSI - program load data	OUT1	—
2	CS_N - program load select, active low	OUT2	—
3	RUN - high to execute, low to reset the core	OUT3	—
4	IN0 - branch input 0	OUT4	—
5	IN1 - branch input 1	OUT5	—
6	TICK0 - tick speed select, low bit	OUT6	—
7	TICK1 - tick speed select, high bit	OUT7	—

tt_um_vga_slot_machine

by **Lerist Lagman**

0263 | 25.175 MHz | HDL Project

github.com/Seltir/IEEE-Project-by-Lagman

"a slot machine"

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	—	R1	—
1	—	G1	—
2	—	B1	—
3	—	VSync	—
4	—	R0	—
5	—	G0	—
6	—	B0	—
7	—	HSync	—

CDM PYTHON GAME

by **VINCE LUCES**

`265` `25.175 MHz` `HDL Project`

`github.com/nagu04/CDM_PYTHON_GAME`

"A hardware-rendered VGA Snake Game in Verilog with growing mechanics and dual keyboard/PMOD controls."

How it works

This project implements a hardware-rendered Snake game targeting Tiny Tapeout using digital Verilog logic and a VGA display interface.

Architecture Overview

1. **VGA Signal Generator:** Converts the 25.175 MHz pixel clock into standard 640x480 @ 60Hz timing signals (`hsync`, `vsync`, `video_on`).
2. **Coordinate & Grid Mapping:** Maps the 640x480 resolution into a 20x15 cell grid (each cell is 32x32 pixels).
3. **Snake Game Logic:** Runs movement and game state logic on every frame tick (`v_cnt == 480`).
   - Supports directional controls with anti-180° turn prevention.
   - Handles body segment shifting across a register array.
   - Detects collision between the snake's head and food to trigger dynamic length growth (`snake_len`).
   - Pseudo-randomly relocates food upon consumption.
4. **PS/2 Keyboard Decoder:** Decodes serial clock (`ui_in[4]`) and data (`ui_in[5]`) signals from an external PS/2 keyboard to extract arrow key presses.
5. **Pixel Pipeline:** Determines color generation on-the-fly ("beam racing") without requiring an external frame buffer memory:
   - **Green:** Snake head & body segments
   - **Red:** Food block
   - **Blue:** Playfield background
   - **Black:** Blanking / Non-active display region

How to test

In Web Simulator (VGA Playground)

1. Select the **Gamepad** preset or click directly on the game preview canvas.
2. Use your keyboard's **Arrow Keys** (or `ui_in[3:0]` buttons) to direct the snake.
3. Drive the snake into red food blocks to verify growth logic.

On Physical ASIC / FPGA Hardware

1. Connect a Tiny Tapeout VGA PMOD to `uo_out[7:0]`.
2. Connect a PS/2 Keyboard adapter to `ui_in[4]` (Clock) and `ui_in[5]` (Data), or connect directional push buttons to `ui_in[3:0]`.
3. Provide a standard 25.175 MHz system clock to `clk` and release active-low reset (`rst_n = 1`).

External hardware requirements

- **VGA PMOD:** Standard Tiny Tapeout 6-bit R2R DAC VGA PMOD (mapped to `uo_out[7:0]`).
- **Display:** Standard VGA-compatible monitor accepting 640x480 @ 60Hz.
- **Input Device:** PS/2 Keyboard or directional push buttons wired to `ui_in[5:0]`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Button Up	R1	—
1	Button Down	G1	—
2	Button Left	B1	—
3	Button Right	VSync	—
4	PS2 Clock	R0	—
5	PS2 Data	G0	—
6	—	B0	—
7	—	HSync	—

Frozen ternary backbone + loadable head

by **Rahul Mascarenhas**

266 | 10 MHz | HDL Project

`github.com/RahulMascarenhas/folded-nn`

"8x8 image classifier. Frozen ternary feature layer compiled into logic; 6-class head reloaded over SPI."

How it works

A 12x12 binary image, classified in two stages.

Frozen backbone, 144 -> 56. 3,584 ternary weights compiled into the netlist at synthesis: `+1` is a wire, `-1` an inverter, `0` deletes the input and its branch of the adder tree. About half are zero. One pixel enters per cycle and eight accumulator lanes sweep the image seven times; the sign of each accumulator is a feature. These weights cannot be changed.

Loadable head, 56 -> 6. 336 ternary weights and six 8-bit biases in a 720-bit shift register, reloaded as 90 bytes. One accumulator walks them and emits six signed scores, then picks the winner on-die.

1,344 cycles per inference, 134 us at 10 MHz.

One inference gives you three things: the six raw scores, the winning class, and the 56 frozen features. Loading a different 90 bytes retargets the chip to six different characters; reading the features instead lets a host run its own classifier.

Trained on EMNIST byclass. A fresh head on six unseen characters loses about five points against a fully trained network.

How to test

Pin	Function
`ui[0]` / `ui[1]`	image bit / shift
`ui[2]`	start
`ui[3]` / `ui[4]`	head bit / shift
`ui[5]`	advance feature readout
`uo[7:0]`	signed score while busy; winning class in `uo[2:0]` once `uio[1]` is high

`uio[0]` / `uio[1]` / `uio[2]`	score valid / done / busy
`uio[5:3]`	which class the current score belongs to
`uio[6]` / `uio[7]`	feature bit / last feature

Reset. Shift 720 head bits in on `ui[3]`, MSB first. Shift 144 pixels in on `ui[0]`, MSB first. Pulse `ui[2]`. Read `uo` on each `uio[0]` pulse for the six scores, then read it once more after `uio[1]` goes high for the winner.

The features latch when the backbone finishes and hold until the next start, so `ui[5]` can be pulsed 56 times to read them on `uio[6]` at any point after that, including while the head is still running.

Input must be normalised the way EMNIST is or accuracy drops sharply: crop to the ink, scale so the longer side fills 20/28 of the frame, centre by centre of mass, then average-pool to 12x12 and threshold at 0.15. Do the scaling at full resolution, not on the 12x12 grid.

A head bitstream is only valid for the preprocessing it was fitted against.

External hardware

None. The chip reports its own answer, so a host is optional.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	img_bit	score[0] / class[0]	score_valid
1	img_shift	score[1] / class[1]	done
2	go	score[2] / class[2]	busy
3	head_bit	score[3]	class[0]
4	head_shift	score[4]	class[1]
5	feat_shift	score[5]	class[2]
6	unused	score[6]	feat_bit
7	unused	score[7]	feat_wrap

UART Hello World

by **Rom DuPlain**

`0267` `50 MHz` `HDL Project`

`github.com/4kbyte/ttihp26b-uart-hello`

"Continuously transmits a hello-world message over the demoboard USB UART"

How it Works

The design repeatedly transmits `Hello, TinyTapeout!\r\n` over a 115200-baud UART. A state machine loads each byte from a fixed message table and shifts a start bit, eight data bits least-significant bit first, and one stop bit. A clock-counted one-second idle interval separates complete messages without changing the standard UART baud rate.

The design expects the standard 50 MHz Tiny Tapeout project clock. Its rounded 434-clock divider produces approximately 115207 baud, an error of about +0.0064%. Reset is active low and sampled on the project clock. While reset is held through a rising clock edge, the UART output returns to its idle high level and the next transmission restarts from the first character.

How to Test

1. Select the project on the Tiny Tapeout demoboard.
2. Set the project clock to 50 MHz.
3. Configure the USB serial terminal for 115200 baud, 8 data bits, no parity, one stop bit, and no flow control.
4. Reset the project.
5. Confirm that `Hello, TinyTapeout!` appears repeatedly.

The UART TX signal is on `uo_out[4]`, the standard demoboard UART-to-USB mapping. `ui_in[3]` is reserved for a future UART RX extension and is ignored by this version. All other output pins are held low.

The submitted physical design meets its 50 MHz clock constraint across the analyzed process, voltage, and temperature corners, with no setup or hold violations. Explore the layout using the Tiny Tapeout 3D viewer or GDS Explorer.

External Hardware

No external hardware is required beyond a Tiny Tapeout demoboard and its USB connection.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	—	—
1	—	—	—
2	—	—	—
3	UART RX (reserved)	—	—
4	—	UART TX	—
5	—	—	—
6	—	—	—
7	—	—	—

Mixer

by **Niklaus Leuenberger**

352 HDL Project

`github.com/NikLeberg/tt_um_mixer`

"*Mixed-signal experimentation*"

How it works

Various inverting schmitt triggers with different thresholds are attached to an internal DAC. The voltage produced by the DAC can be controlled with the digital inputs `ui_in[7:0]`. Each schmitt trigger then presents its state on the corresponding `uo_out[7:0]` pin.

Additional free standing inverting schmitt triggers are available on the bidirectional pins. With these and external RC components simple oscillators can be built.

How it came to be

Thanks to swiss chips sponsoring I was able to participate in the previous IHP26a run with a 8-bit floating point multiplier. There I took everything the OSS tools offer to us to squeeze a near perfect design out of very imperfect and *lazy* HDL code. Great.

But now for this second IHP16b TT run I needed to go a level deeper: To the physical world. No nice zeros and ones but only the raw *analoginess* of CMOS technology.

The initial idea was to *again* build a multiplier. But it would multiply in the physical world! I.e. produce some voltages that by nature get multiplied together and then converted back to digital. Easy! No not so much. Just the amount of stuff necessary to make that last bit, the ADC work, is too much. Sure, others made some. But I'm chronically late and the design is due in a month.

So what else can we do? Analog tiles are not available for this run. But nothing prohibits us from using analog stuff inside a digital facade. So we have a digitally controlled voltage, well a DAC, which then feeds into multiple schmitt triggers each with different threshold. That way we can sort of *gauge* the voltage that was produced internally.

How to test

TBD

External hardware

TBD

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DAC[0]	trigger1v1	—
1	DAC[1]	trigger2v1	—
2	DAC[2]	trigger4v1	ST1v1_IN
3	DAC[3]	trigger8v1	ST1v1_OUT
4	DAC[4]	—	ST2v1_IN
5	DAC[5]	—	ST2v1_OUT
6	DAC[6]	—	ST4v1_IN
7	DAC[7]	—	ST4v1_OUT

TinyDMA: A Descriptor-Based Dual-PSRAM Memory Mover

by **bennett_lahn**

354 66 MHz HDL Project

`github.com/bennett-lahn/dma-tapeout`

"TinyDMA bulk-copies between dual QSPI PSRAM using in-memory descriptors"

How it works

TinyDMA is an asychronous memory mover targeting the Tiny Tapeout QSPI PMOD. It uses chains of memory-based descriptors to transfer memory between PSRAM chips (no flash support).

After the host pulses **START**, the ASIC copies bytes between two PSRAM chips using 11-byte in-memory transfer control descriptors (TCDs).

Each TCD names a source pointer, destination pointer, length, next-TCD pointer, and control flags. Flags indicate the write source/destination, device location for next TCD pointer and **QUIT** if that chain is the last in the sequence.

The first TCD is always at address 0 on PSRAM 0. Same-device and cross-device copies are supported. The ASIC does not read/write to the PMOD flash, but the MCU can still use it by raising **BUS_REQ** (`ui_in[2]`) and waiting for **BUS_GNT** (`uo_out[1]`).

For an in-depth description of the architecture, firmware, and verification, see the full TinyDMA documentation.

Host pins:

- `ui_in[0]` **START** - accepted only while idle and `BUS_REQ` is low
- `ui_in[2]` **BUS_REQ** - MCU wants the QSPI bus
- `uo_out[0]` **DONE** - high whenever the DMA is idle
- `uo_out[1]` **BUS_GNT** - MCU may drive `uio`
- `uio_out` - **QSPI PMOD**

Infinite TCD chains should be avoided. Terminate using `rst_n`. Target clock is 66 MHz; SCK is clk/2.

How to test

The associated GitHub repo has MicroPython firmware under `firmware/`, including an existing testbench.

One-time setup

1. Set `config.ini` on the board for `ASIC_RP_CONTROL` and `clock_frequency = 66e6`.
2. From a PC (WSL/Linux): `pip install mpremote`, then copy firmware and reset:

```
mpremote fs cp -r firmware :/firmware
mpremote reset
```

3. Connect the QSPI PMOD and enable this design on the mux (`tt_um_lahnb_sgdma`).

Interactive REPL

Open the USB serial REPL, then:

```
import firmware.session as session
session.init("tt_um_lahnb_sgdma")   # mux, 66 MHz clk, reset, idle
check
session.bring_up_psram()            # SPI reset + Enter Quad on
both RAMs
session.status()
```

Run one memory copy using REPL

Minimal smoke test (8 bytes from PSRAM0 `0x000100` to `0x000200`, head TCD at `0`, quit TCD at `0x000010`):

```
from firmware.tcd import Tcd, encode_tcd
from firmware.link import b64encode, b64decode

src, dst, quit_slot = 0x000100, 0x000200, 0x000010
session.write_span(0, src, b64encode(bytes(range(8))))
session.write_span(0, 0, b64encode(encode_tcd(Tcd(
    src_ptr=src, dest_ptr=dst, transfer_len=8,
    next_tcd=quit_slot, src_device=0, dest_device=0,
next_device=0))))
session.write_span(0, quit_slot,
b64encode(encode_tcd(Tcd(quit=True))))
session.start_and_wait()
line = session.read_span(0, dst, 8)
dest = b64decode(line.split(" ", 1)[1])
# expect dest == bytes(range(8))
```

- `device`: `0` = PSRAM A, `1` = PSRAM B.
- `session.write_span(device, addr, b64_data)` - Writes decoded bytes at `addr`. Each call takes **BUS_REQ**, writes in small chunks, then releases the bus again.
- `session.read_span(device, addr, length)` - Reads PSRAM. Returns an `OK <base64>` line (decode the part after `OK` with `firmware.link.b64decode`).

- `session.start_and_wait()` - pulses **START** and waits until **DONE** rises.

Automated tests from the host PC

All hardware tests are in `hil/` and use the included reference model. From the repo root (after using `source test/env.sh` if using the project venv):

```
pytest hil/tests/ -v --target=fpga        # builds/uploads
bitstream, then runs on Tiny Tapeout FPGA
pytest hil/tests/ -v --target=asic        # same tests on silicon
python -m hil --target=asic               # interactive menu for
selecting test cases
```

The MCU host builds TCDs, installs memory over QPI, pulses **START**, dumps memory contents, and compares against a reference model. Full API and bus rules: docs/human/architecture/firmware.md.

External hardware

- Tiny Tapeout QSPI PMOD with dual APS6404L (or compatible) PSRAM.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	START	DONE	FLASH_CS
1	—	BUS_GNT	SIO0
2	BUS_REQ	—	SIO1
3	—	—	SCK
4	—	—	SIO2
5	—	—	SIO3
6	—	—	RAM_A_CS
7	—	—	RAM_B_CS

Digital IQ Lock-in (IEEE)

by **Gabriela Sheilyn Ticona Jimenez**

`0356` `50 MHz` `HDL Project`

`github.com/gabSHtj-pixel/ttihp-lockin-ieee`

"Dual-phase digital demodulator with programmable block averaging — Equipo Triada CEC"

Overview

This project implements a **dual-phase digital lock-in amplifier / I/Q demodulator** for the Tiny Tapeout IHP shuttle.

The design receives signed 8-bit input samples, generates internal quadrature reference signals, performs I/Q synchronous demodulation, and applies programmable block averaging.

This revision has been specifically optimized to improve the probability of fitting inside a **single 1x1 IHP Tiny Tapeout tile**.

The main design goals are:

- signed 8-bit input samples;
- digital I/Q demodulation;
- programmable 16-bit phase step;
- 32-point reference waveform;
- selectable averaging windows of 16, 64, 256, or 1024 samples;
- coherent 16-bit I/Q result readout;
- low-area RTL architecture suitable for a `1x1` Tiny Tapeout allocation.

The project keeps:

```
tiles: "1x1"
```

in `info.yaml`.

A successful Tiny Tapeout **GDS workflow** is still required to confirm that placement, routing and timing physically fit inside the allocated 1x1 tile.

How it works

The system is divided into three main internal blocks:

1. `reference_generator`
2. `lockin_core`
3. `register_interface`

The top-level module is:

`tt_um_gstj_lockin`

implemented in:

`src/project.v`

---

1. Reference generator

The `reference_generator` contains a **16-bit phase accumulator**.

The phase is incremented only when the lock-in core actually accepts an input sample.

Conceptually:

```
phase = phase + phase_step
```

The programmable value `phase_step` therefore controls the frequency of the internally generated reference.

Only the five most significant bits of the phase accumulator are used to address the reference waveform:

```
phase_index = phase[15:11]
```

This produces:

```
2^5 = 32
```

possible phase positions.

The lower 11 bits are still preserved internally, so arbitrary 16-bit `phase_step` values retain fractional phase resolution.

The phase accumulator is reset to zero when the design is reset or when the active measurement configuration is changed.

---

2. I/Q demodulation

The lock-in amplifier performs two synchronous products for every accepted input sample.

Conceptually:

```
I = sample × cos(reference_phase)
Q = sample × −sin(reference_phase)
```

These products are accumulated over a programmable number of samples.

After the selected block has been completed, the accumulated values are divided by the block length.

The available averaging windows are:

`window_sel`	Number of samples	Division

00	16	/16
01	64	/64
10	256	/256
11	1024	/1024

The default value after reset is:

`window_sel = 2`

therefore the default averaging window is:

`256 samples`

---

3. Area-optimized architecture

The original RTL already requested a `1x1` tile, but declaring:

```
tiles: "1x1"
```

does not guarantee that the synthesized standard cells can physically fit inside that area.

For this reason, the datapath was redesigned specifically to reduce silicon area.

The most important changes are described below.

Serial multiplier

The previous architecture used a generic signed:

`8 × 8 bit combinational multiplier`

Although the multiplier was shared between the I and Q calculations, a combinational multiplier can still require a significant amount of standard-cell area.

The optimized architecture removes this multiplier.

Multiplication is now performed using a **serial shift-and-add datapath**.

The same arithmetic hardware is reused for I and Q.

Each reference coefficient has a maximum magnitude of 127, which requires seven magnitude bits.

Therefore one product requires approximately:

`7 serial multiplication cycles`

The datapath performs:

```
7 cycles for I
7 cycles for Q
```

instead of synthesizing a complete parallel 8x8 multiplier.

This deliberately exchanges throughput for lower silicon area.

---

4. Compact reference LUT

The reference waveform contains 32 phase positions.

The original implementation explicitly represented the complete sine waveform and evaluated sine/cosine references separately.

The optimized implementation exploits sine-wave symmetry.

Instead of storing all 32 signed values, only nine non-negative magnitudes are required:

Index	Magnitude
0	0
1	25
2	49
3	71
4	90
5	106
6	117
7	125
8	127

The other values are reconstructed using quarter-wave symmetry and sign inversion.

This produces the same 32-point numerical waveform while reducing duplicated lookup logic.

The same LUT hardware is reused sequentially for both quadrature references.

---

5. Reduced accumulator width

The original architecture used two signed 26-bit accumulators.

The optimized implementation uses:

```
25-bit signed accumulators
```

for both I and Q.

This reduction is mathematically safe.

The maximum magnitude of an input sample is:

`128`

because the signed 8-bit input range is:

`−128 ... +127`

The maximum reference magnitude is:

`127`

The largest averaging block contains:

`1024 samples`

Therefore the worst-case accumulated magnitude is:

`1024 × 128 × 127`

which gives:

`16,646,144`

A signed 25-bit value has the range:

`−16,777,216 ... +16,777,215`

Therefore 25 bits are sufficient to represent the complete worst-case accumulated value without saturation.

---

6. Fixed averaging shifts

The previous implementation calculated the average using a variable arithmetic right shift.

Conceptually:

```
sum >>> shift_count
```

where:

```
shift_count = 4, 6, 8 or 10
```

A variable shift can synthesize into a barrel-shifter or multiplexer network.

The optimized implementation instead uses fixed bit slices.

Conceptually:

```
/16   -> shift 4
/64   -> shift 6
/256  -> shift 8
/1024 -> shift 10
```

The required slice is selected according to `window_sel`.

The same averaging logic is reused sequentially for I and Q.

---

7. Result storage optimization

The previous architecture contained two copies of the I/Q result:

```
core result registers
+
register-interface snapshot registers
```

This duplicated approximately 32 bits of result storage.

The optimized architecture removes this duplication.

The completed results are written directly into the readback registers.

Internally each result is stored as a signed:

```
15-bit value
```

and is sign-extended when exposed through the 16-bit register interface.

The average of a signed 8-bit sample multiplied by a reference whose magnitude is at most 127 fits safely inside this range.

---

8. SNAPSHOT_STROBE behavior

Because the readback registers now directly contain the latest complete result, copying the result into another snapshot register is no longer necessary.

For pin compatibility the signal is still called:

```
SNAPSHOT_STROBE
```

but its function is now an **acknowledge signal**.

When asserted, it clears:

```
NEW_RESULT
```

The I/Q values themselves remain stored.

Therefore the recommended sequence is:

```
wait for NEW_RESULT = 1
read I
read Q
pulse SNAPSHOT_STROBE
```

After the pulse:

```
NEW_RESULT = 0
```

but the previous result remains available in registers 0-3 until a new result is generated or the measurement state is cleared.

Pin interface

Dedicated input bus

`ui_in[7:0]`

is used as:

`DATA_IN[7:0]`

For sample acquisition it represents a signed 8-bit value:

`-128 ... +127`

During register writes the same bus contains the register data.

Dedicated output bus

`uo_out[7:0]`

is:

`READ_DATA[7:0]`

The value depends on the register selected with `ADDR[2:0]`.

Bidirectional pins

Pin	Signal	Direction
`uio[0]`	`SAMPLE_STROBE`	Input
`uio[1]`	`WRITE_STROBE`	Input
`uio[2]`	`ADDR[0]`	Input
`uio[3]`	`ADDR[1]`	Input
`uio[4]`	`ADDR[2]`	Input
`uio[5]`	`SNAPSHOT_STROBE` / ACK	Input
`uio[6]`	`BUSY`	Output
`uio[7]`	`NEW_RESULT`	Output

The output-enable configuration is:

```
uio_oe = 8'b11000000;
```

Therefore only:

```
uio[6]
uio[7]
```

are driven by the ASIC.

Register map

The register address is selected using:

`uio_in[4:2]`

giving eight addresses.

Address	Read	Write
0	I result, bits 7:0	—
1	I result, bits 15:8	—
2	Q result, bits 7:0	—
3	Q result, bits 15:8	—
4	`phase_step[7:0]`	`phase_step[7:0]`
5	`phase_step[15:8]`	`phase_step[15:8]`
6	`window_sel`	`window_sel`
7	Status	Control

Phase-step configuration

The reference phase increment is a 16-bit value.

The low byte is located at:

`register 4`

and the high byte at:

`register 5`

Therefore:

```
phase_step = {register_5, register_4}
```

The default reset value is:

`0x1000`

Writing either phase-step byte clears the current accumulation and restarts the reference phase.

This prevents measurements obtained using different reference configurations from being mixed in the same averaging block.

Averaging-window configuration

Register:

`6`

contains:

`window_sel[1:0]`

The mapping is:

```
00 -> 16 samples
01 -> 64 samples
10 -> 256 samples
11 -> 1024 samples
```

Writing this register clears the current accumulation and restarts the reference phase.

Status register

Register:

`7`

returns:

`{5'b00000, OVERRUN, NEW_RESULT, BUSY}`

Therefore:

Bit	Signal
0	`BUSY`
1	`NEW_RESULT`
2	`OVERRUN`
7:3	Reserved / 0

BUSY

`BUSY` indicates that the arithmetic datapath is currently processing a sample.

A new sample should only be sent when:

`BUSY = 0`

The serial arithmetic engine spends approximately:

```
7 clocks -> I multiplication
7 clocks -> Q multiplication
```

for a normal sample.

The final sample of each averaging block additionally requires two cycles to output the completed I and Q averages.

Consequently the controller must always use the `BUSY` handshake rather than assuming that a new sample can be accepted every clock cycle.

At the configured:

`50 MHz`

clock, the architecture is still capable of processing input samples on the order of several million samples per second while using considerably less combinational arithmetic hardware than the parallel-multiplier architecture.

NEW_RESULT

`NEW_RESULT` is asserted after a complete I/Q result pair has been generated.

The core outputs the I result first and the Q result afterward.

`NEW_RESULT` is asserted only after Q has been stored.

Therefore software never treats a partially updated I/Q pair as a complete measurement.

When:

`NEW_RESULT = 1`

registers 0-3 contain one coherent result pair.

OVERRUN

`OVERRUN` becomes active if a new sample request is generated while the arithmetic core is already busy.

Conceptually:

`SAMPLE_STROBE while BUSY = 1`

causes:

`OVERRUN = 1`

The flag is sticky until the system is cleared.

The controller should therefore always check:

`BUSY = 0`

before generating another `SAMPLE_STROBE`.

Feeding a sample

To submit a sample:

1. Wait until:

`BUSY = 0`

2. Place the signed 8-bit sample on:

`ui_in[7:0]`

3. Pulse:

`SAMPLE_STROBE`

for one clock.

The interface performs rising-edge detection, so the strobe should return low before another sample request is generated.

When the sample is accepted, the phase accumulator advances exactly once.

Writing a register

To write configuration data:

1. Put the register address on:

`uio_in[4:2]`

2. Put the value on:

`ui_in[7:0]`

3. Pulse:

`WRITE_STROBE`

for one clock.

Writes to registers:

```
4
5
6
```

automatically clear the current measurement accumulation.

Reading the I/Q result

Wait until:

`NEW_RESULT = 1`

Then read:

```
register 0 -> I[7:0]
register 1 -> I[15:8]

register 2 -> Q[7:0]
register 3 -> Q[15:8]
```

The high result byte is sign-extended from the internal signed result representation.

After the four bytes have been read, pulse:

`SNAPSHOT_STROBE`

to acknowledge the result.

This clears:

`NEW_RESULT`

without deleting the stored I/Q data.

Clear / restart control

Writing register `7` with bit 0 equal to one:

```
data_in[0] = 1
```

clears the active measurement state.

This resets:

```
I accumulator
Q accumulator
sample counter
stored I result
stored Q result
NEW_RESULT
OVERRUN
reference phase
```

The configured:

```
phase_step
window_sel
```

are preserved.

Enable behavior

The Tiny Tapeout `ena` signal enables operation of the internal datapath and register events.

If:

```
ena = 0
```

the design does not accept new arithmetic operations.

Area optimization summary

The principal area reductions are:

Structure	Previous architecture	1x1-optimized architecture
Multiplier	Signed combinational 8x8 multiplier	Shared serial shift/add multiplier

I/Q multiplication	Shared multiplier used sequentially	Single serial datapath reused for I and Q
Reference waveform	Complete LUT evaluations	Folded quarter-wave LUT
Stored magnitudes	Full 32-point waveform	9 magnitudes
Accumulators	26-bit I + 26-bit Q	25-bit I + 25-bit Q
Averaging	Variable arithmetic shift	Fixed bit slices
Result registers	Core result + snapshot result	One readback result pair
Internal result width	16 bits	15 bits with sign extension
Placement density	60%	70%
Tile allocation	1x1	1x1

Physical implementation configuration

The project uses:

```
CLOCK_PERIOD = 20 ns
```

corresponding to:

```
50 MHz
```

The optimized placement target is:

```
PL_TARGET_DENSITY_PCT = 70
```

The tile allocation remains:

```
tiles: "1x1"
```

The density value was increased from the previous 60% target so that the placer can use the limited 1x1 floorplan more efficiently.

Increasing placement density does not itself reduce synthesized cell area, which is why the RTL datapath was also redesigned.

Main optimization strategy

The design follows an area-first strategy:

```
parallel arithmetic
        ↓
serialized arithmetic

duplicated logic
        ↓
```

```
shared logic

duplicated registers
        ↓
single result storage

generic variable operations
        ↓
fixed operations

full waveform LUT
        ↓
symmetry-reduced LUT
```

The architecture therefore intentionally prioritizes:

`small silicon area`

over:

`maximum sample throughput`

which is appropriate for the 1x1 Tiny Tapeout target.

Verification

The verification environment uses Cocotb.

The tests compare the RTL output against an independent integer model of the digital lock-in amplifier.

The verification includes:

- reset behavior;
- phase-step programming;
- all four averaging windows;
- multiple phase steps;
- phase accumulator wrapping;
- signed input samples;
- I/Q result generation;
- noisy signal recovery;
- amplitude/phase behavior;
- `BUSY` behavior;
- `NEW_RESULT`;
- overrun detection;
- enable gating;
- configuration changes;
- accumulation reset behavior.

The numerical 32-point reference waveform is preserved by the compact LUT architecture.

The serial multiplication architecture is intended to produce the same integer product as the previous parallel multiplication implementation.

Files used by the design

The relevant RTL files are:

```
src/project.v
src/reference_generator.v
src/lockin_core.v
src/register_interface.v
```

Physical implementation configuration:

```
src/config.json
```

Tiny Tapeout project configuration:

```
info.yaml
```

Verification:

```
test/test.py
test/tb.v
test/Makefile
```

Project documentation:

```
docs/info.md
```

External hardware

No external hardware is required for RTL simulation.

Simulation can provide signed 8-bit samples directly to the design.

For physical operation, an external controller must:

- supply signed 8-bit samples;
- generate `SAMPLE_STROBE`;
- wait for `BUSY`;
- configure the registers;
- detect `NEW_RESULT`;
- read the I/Q result registers.

If the input originates from an analog signal, external hardware is also required for:

```
ADC
signal conditioning
anti-alias filtering, if required
```

The ASIC design itself does **not** contain an ADC.

Final 1x1 verification

The project is structurally optimized for a Tiny Tapeout IHP:

`1x1`

tile.

However:

```
tiles: "1x1"
```

only defines the requested physical allocation.

It does not prove that synthesis, placement and routing will succeed.

The definitive verification is the Tiny Tapeout GitHub:

`GDS`

workflow.

The design should only be considered physically validated for 1x1 after the complete GDS flow passes successfully.

Important checks include:

```
synthesis
floorplanning
global placement
detailed placement
clock-tree synthesis
routing
timing
DRC / implementation checks
```

If global placement fails because of excessive utilization or congestion, the first step should be to inspect the synthesis and placement reports rather than immediately increasing the tile allocation.

The objective of this revision is to preserve:

```
tiles: "1x1"
```

and reduce the RTL area until the physical implementation successfully fits inside that allocation.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA_IN[0]	READ_DATA[0]	SAMPLE_STROBE
1	DATA_IN[1]	READ_DATA[1]	WRITE_STROBE
2	DATA_IN[2]	READ_DATA[2]	ADDR[0]

#	Input	Output	Bidirectional
3	DATA_IN[3]	READ_DATA[3]	ADDR[1]
4	DATA_IN[4]	READ_DATA[4]	ADDR[2]
5	DATA_IN[5]	READ_DATA[5]	SNAPSHOT_STROBE
6	DATA_IN[6]	READ_DATA[6]	BUSY
7	DATA_IN[7]	READ_DATA[7]	NEW_RESULT

Izhikevich event bridge (4 contexts)

by **Austin Biaselli**

`0357` `10 MHz` `HDL Project`

`github.com/Abiaselli/ttihp-IZH-logic-gate-2x3`

"Four-context time-multiplexed fixed-point Izhikevich neuron bank with programmable synapses and external digital spike capture"

How it works

This project implements four independently stored fixed-point Izhikevich neuron contexts in a `3x2` Tiny Tapeout IHP slot. A single signed serial multiplier and control engine are time-multiplexed across the four contexts to minimize duplicated arithmetic.

Each neuron stores membrane voltage V, recovery state U, synaptic current, bias/current, parameters a/b/c/d, two signed synaptic weights, and two source identifiers. Source IDs can select prior-step recurrent spikes or one of four external digital spike inputs. A STEP command advances all four neurons once and returns the spike bitmap.

The external event inputs are digital logic signals. Analog neurons should be connected through external comparators, Schmitt receivers, or appropriate level-conditioning circuitry. The intended small-system configuration can use two of the four event inputs and route one selected output back to an analog neuron through the FPGA/external pulse or current interface.

How to test

Run the RTL test with:

```
cd test
pip install -r requirements.txt
make -B
```

The test checks reset state, the Tiny Tapeout bidirectional-pin direction mask, register write/readback, the four-context boundary, a full network STEP, and the resulting membrane-voltage update.

After hardening, copy the IHP unpowered gate-level netlist to `test/gate_level_netlist.v` and run:

```
make -B GATES=yes
```

External hardware

Intended system:

- FPGA providing clock, packet transport, buffering, and larger network routing.
- External comparator/level-conditioned spike sources from analog neurons.
- Optional external DAC/current-pulse drivers controlled by the FPGA for feedback into analog neurons.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	command_data[0]	response_data[0]	command_valid
1	command_data[1]	response_data[1]	command_ready
2	command_data[2]	response_data[2]	response_ready
3	command_data[3]	response_data[3]	response_valid
4	command_data[4]	response_data[4]	spike_in0
5	command_data[5]	response_data[5]	spike_in1
6	command_data[6]	response_data[6]	spike_in2
7	command_data[7]	response_data[7]	spike_in3

PS/2 Keyboard Decoder for 68k

by **Ben Payne**

`0358` `25 MHz` `HDL Project`

`github.com/benpayne/ttihp-ps2-m68k`

"A PS2 keyboard decoder for retro 68k systems (IHP SG13G2)"

How it works

This decoder works by first debouncing the inputs to make sure that we get a clean sample of them that is synchronized to our clock. It then looks at the down transition of ps2_clk and reads the value of ps2_data. It shifts this into a 11 bit shift register. When ps2_clk remains high for more than 1/2 of the 10kHz ps2_clk cycle it knows that the end of the data has arrived. It then triggers a valid flag to tell the system that something has arrived. The valid flag, which is exposed on a pin, will trigger the fifo to read the byte of data and it will be stored for retrieval by the host. When valid is triggered it will also trigger the interrupt pin. The valid pin is a pulse for one system clock cycle, but the interrupt will remain set until it is cleared. We also include a data_rdy signal that tells the host that there is data to read. This is useful if your interrupt handler needs to read multiple bytes.

When the host wants to read a byte, it asserts the chip select (cs) signal. The uio bus is put into an output state immediately (combinationally) whenever cs is high, and is an input bus at all other times (but we never read it...). The read itself is edge-triggered and filtered: cs and clear_int are asynchronous host signals, so both pass through a 2-flop synchronizer, and cs must then be seen high on two consecutive clocks before one byte is popped from the FIFO. Holding cs high longer never pops a second byte.

Host timing (25 MHz clock, 40 ns period):

- cs must be high for at least 2 clocks (80 ns) to register a read.
- The byte appears on the bus 5 clock edges after cs rises (synchronizer + edge detect + FIFO read register), i.e. **allow at least 200 ns from cs rising before sampling the data**, and keep cs high until you have sampled it. For a 68000 this means generating DTACK externally with roughly 2 wait states at 8 MHz.
- clear_int must be high for at least 2 clocks (80 ns); the interrupt drops on the 3rd clock after it rises.
- interupt is **active-high** a 68k /IPL input needs an external inverter.
- valid is a single 40 ns pulse per byte - useful on a scope, not something a host should try to catch.

- These figures scale with the clock: the 128-cycle input debounce (5.1 µs) and the 100 µs end-of-frame timeout assume 25 MHz.

The design includes a fifo_full output signal that indicates when the FIFO buffer is full (4 bytes). When full, additional bytes from the keyboard will be silently dropped until space becomes available. Software should monitor this flag to detect potential data loss during rapid typing.

Interrupt semantics (important for driver writers): `interupt` is set by the *arrival* of a byte, not by FIFO occupancy. If two bytes arrive before the host services the first, reading one byte and then pulsing `clear_int` leaves the second byte queued with `interupt` low. An interrupt handler should therefore drain the FIFO while `data_rdy` is high (or re-check `data_rdy` after `clear_int`) rather than assume one interrupt equals one byte. Reading a byte never re-raises the interrupt; only a new arrival does.

Reading with the FIFO empty is harmless: the data bus simply holds the last byte read and the FIFO state is unchanged, so a polling host can read speculatively.

Inter-byte gap (known limitation): end-of-frame is detected by ps2_clk staying high for more than 100 µs. Two frames arriving closer together than that run together; the decoder keeps the last 11 bits, so the second byte is received and the first is lost. Real keyboards leave 1 ms or more between bytes, so this only matters for synthetic sources.

For debugging on the bench, uo[4] also carries a 115200 baud 8N1 UART transmission each time a valid byte is captured: a status byte followed by the decoded PS/2 byte. The status byte is `{0000, fifo_full, data_rdy, interupt, 1}` (bit 0 always 1 as a frame marker), sampled two clocks after `valid` so the flags reflect the byte just queued - e.g. `0x07` for a normal byte, `0x0F` when it landed in a full FIFO. Bytes dropped from a full FIFO are still echoed on the UART.

Bring-up and fault isolation

A returned chip can't be repaired, so the point of the on-chip instrumentation is to work out *which block* failed, well enough to decide what to change on the next shuttle. Three signals that are otherwise internal are brought out on the spare output pins for that purpose:

Pin	Signal	What it tells you
`uo[5]`	`ps2_clk_dbg`	The debounced PS/2 clock, i.e. the debouncer's output
`uo[6]`	`ps2_data_dbg`	The debounced PS/2 data

`uo[7]`	`cs_trigger_dbg`	The internal FIFO read strobe, one clock wide per read

They are direct taps on existing nets - no extra state, and nothing else in the design depends on them.

The reason they matter is that the UART on `uo[4]` reports `ps2_key_data`, which is the decoder's output and therefore the FIFO's *input*. On its own the UART can confirm the whole PS/2 front end works, but it can't see the FIFO's output, and it can't distinguish a dead input pad from a debouncer that is swallowing the signal. `uo[5]`/`uo[6]` close the first gap, `uo[7]` the second.

Bringing the chip up without a 68k, a keyboard, or a level shifter: nothing requires a real keyboard - the PS/2 input is a perfectly good injection port. Bit-bang known frames onto `ui[0]`/`ui[1]` from a microcontroller (the demo board's RP2040 will do), and read the UART back. Because the microcontroller drives the demo board directly, this also sidesteps the 5V level-shifting requirement for initial bring-up. Then drive `cs` and sample `uio` from the same microcontroller to exercise the host half.

Fault isolation table:

Observation	Conclusion
No UART traffic at all	Clock, reset, power, or the UART block
`uo[5]`/`uo[6]` never move while driving the PS/2 lines	Input pad or the debouncer
`uo[5]` toggles 11 times per frame but no `valid`	Shift register, frame validation, or the end-of-frame timeout
UART fires with the right scan code	Entire PS/2 half is good, up to and including the FIFO write
Status byte has `data_rdy=0`	Decoded correctly but the byte didn't land in the FIFO
`fifo_full` asserts on the 5th byte	FIFO counter logic is good
UART correct but a host read returns the wrong byte, `uo[7]` pulsing	`cs` path is fine; fault is in the FIFO read port or the `uio` pads
UART correct but `uo[7]` never pulses	`cs` synchronizer or the glitch filter

Port from TTGF0p2 (GF180) to TT IHP 26b (IHP SG13G2)

This project was originally built for the Tiny Tapeout GF0.2µm shuttle (GlobalFoundries GF180MCU, 180nm), which offered native 5V I/O tolerance and a 3.3V core. It has been ported here to the Tiny Tapeout IHP 26b shuttle, which targets the open source IHP SG13G2 (130nm SiGe BiCMOS) PDK.

What changed:

- **No VPWR/VGND ports** - the IHP template's top module interface (`ui_in, uo_out, uio_in, uio_out, uio_oe, ena, clk, rst_n`) doesn't require explicit power ports on every submodule, unlike the GF180 flow. All power routing is handled by the standard cell fabric.
- **I/O voltage** - IHP's standard digital pads are **not natively 5V-tolerant** like GF180's. PS/2 signaling is 5V, so this design now requires **external level shifting or a resistive voltage divider** on `ui_in[0]` (ps2_clk) and `ui_in[1]` (ps2_data) to bring the signal down to the IHP pad's supported input range before it reaches the chip. This is the same external hardware requirement the original Sky130-based TT08 version of this project needed - GF180's 5V tolerance was a temporary advantage that doesn't carry over to IHP.
- **Core logic unchanged** - the PS/2 protocol decoder, debouncer, and FIFO are the same design that passed all functional tests on the GF180 target; only the power/pad interface differs.
- **Host inputs synchronized** - cs and clear_int now pass through 2-flop synchronizers before anything samples them (the PS/2 inputs already did). On the GF180 version the raw cs pin fanned out to three flops in the glitch filter, which on a marginal edge could disagree and silently skip a read. Read latency is 200 ns instead of 120 ns as a result.
- **UART debug output fixed** - the GF180 version's UART state machine advanced past its "wait for transmit to finish" check one cycle early (the UART's busy flag has a cycle of latency), so only the status byte was ever transmitted and the data byte was dropped. The three UART tests had been skipped as "timing sensitive"; un-skipping them exposed the bug. Fixed here, and the UART module now uses an asynchronous reset like the rest of the design.

How to test

Level-shift (or resistor-divide) a standard PS/2 keyboard's 5V CLK/DATA lines down to a voltage compatible with the IHP SG13G2 I/O pads, then connect them to ui_in[0] (ps2_clk) and ui_in[1] (ps2_data). At this point you can hit keys and they will be queued in the FIFO. Then interface a retro computer to the CS, interrupt and data lines to read the FIFO. This will depend on the system you're using, but note you'll need external address decoding logic and for

chips like the 68000 you'll need to generate DTACK and other bus-timing signals elsewhere.

External hardware

Connect a standard PS/2 keyboard to the chip through a level shifter / voltage divider:

- PS/2 pin 1 (DATA) → level shifter → ui_in[1]
- PS/2 pin 5 (CLK) → level shifter → ui_in[0]
- PS/2 pin 3 (GND) → GND
- PS/2 pin 4 (VCC) → 5V supply (keyboard side only)

Note: Unlike the GF180 version of this project, IHP SG13G2 does not offer native 5V-tolerant I/O, so level shifting (or a simple resistive divider) is required between the keyboard and ui_in[0:1].

Interface to your microprocessor:

- Connect CS, interrupt, and data_out[7:0] signals
- Add external address decoding logic as needed
- For 68000: Generate DTACK externally based on CS timing

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	ps2_clk	valid	data_out[0]
1	ps2_data	interupt	data_out[1]
2	clear_int	data_rdy	data_out[2]
3	cs	fifo_full	data_out[3]
4	—	uart_tx	data_out[4]
5	—	ps2_clk_dbg	data_out[5]
6	—	ps2_data_dbg	data_out[6]
7	—	cs_trigger_dbg	data_out[7]

Neurona LIF con Aprendizaje STDP Dinamico (IEEE)

by **Casandra Saray Galiano Quillahuaman**

360 | 10 MHz | HDL Project

github.com/Cass-s-ui/mi_proyecto_neurona1

"Neurona artificial biologica con plasticidad configurable y monitoreo de peso en tiempo real. — Equipo Triada CEG"

How it works

The design implements a biologically-inspired Leaky Integrate-and-Fire (LIF) neuron with a digital STDP (Spike-Timing-Dependent Plasticity) learning core:

1. **`stdp_core`**: tracks the time elapsed since the last pre-synaptic (`spike_pre`) and post-synaptic (`spike_post`) spikes using two down-counters (`tau_pre`, `tau_post`), each reloaded to a configurable `window_size` on a spike. If the post-synaptic neuron fires while `tau_pre` is still counting down, the synaptic `weight` is potentiated (LTP). If a pre-synaptic spike arrives while `tau_post` is still counting down, the weight is depressed (LTD). Weight is a saturating 6-bit register (0-63), initialized to 32.
2. **`neuron_lif`**: integrates an 8-bit membrane potential `v_mem` by adding the (weight-scaled) `stimulus` each cycle and subtracting a configurable `leak`. When `v_mem` crosses the configurable `threshold` (scaled to 8 bits), the neuron emits one `spike` pulse and resets `v_mem` to 0.
3. **Top level (`project.v`)**: wires an external input pulse (`ui_in[0]`, `spike_pre`) into both the STDP core and, when active, into the LIF neuron's stimulus (scaled by the current synaptic weight). The LIF neuron's own output spike closes the loop back into the STDP core as `spike_post`, so the weight adapts based on the relative timing between the external input and the neuron's own firing.

How to test

Configure the neuron via `ui_in` and `uio_in`, then drive `spike_pre` pulses on `ui_in[0]` and observe `spike_post` and the live synaptic weight on `uo_out`:

- `ui_in[0]`: external input spike (`spike_pre`)
- `ui_in[4:1]`: leak configuration
- `ui_in[7:5]`: STDP time window
- `uio_in[3:0]`: firing threshold

- `uo_out[0]`: output spike (`spike_post`)
- `uo_out[6:1]`: current synaptic weight (0-63)

`test/test_project.py` drives repeated input pulses at a short interval to force potentiation (LTP), then lowers the threshold to force the neuron to fire ahead of the input and drive depression (LTD), logging the synaptic weight after every pulse.

External hardware

None - this project only exercises the dedicated and bidirectional I/O pins directly.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	spike_pre	spike_post	threshold[0]
1	leak[0]	weight[0]	threshold[1]
2	leak[1]	weight[1]	threshold[2]
3	leak[2]	weight[2]	threshold[3]
4	leak[3]	weight[3]	—
5	stdp_window[0]	weight[4]	—
6	stdp_window[1]	weight[5]	—
7	stdp_window[2]	unused	—

Multi Segment Monitor

by **Matt Venn** & **Claude**

`0361` `40 MHz` `HDL Project`

`github.com/mattvenn/multi-seg-monitor`

"Simulates a 64x37 array of 7 segment displays on a VGA screen"

How it works

Simulates a wall of 7 segment displays on a VGA screen: a **64 x 37 grid of digits — 2368 digits, 18944 segments**, each segment with its own 4 bit brightness, sent through a colour palette on the way out. Video is 800x600 @ 60 Hz from a 40 MHz clock.

There is no framebuffer. The chip races the beam and keeps only the digit row being drawn, in a 1 kB SRAM buffer holding four rows of 256 bytes. Each segment is the AND of an x zone and a y zone within its 12x16 cell.

`uio[7]` selects the data source:

- **Internal generator** (low): a slowly varying zone plate, no external data needed. The palette changes itself every 1024 frames (17 s), fading through black.
- **Stream port** (high): a byte on `ui_in` is latched on the rising edge of the strobe on `uio[6]`. Byte *n* of a frame is digit *n*/4, nibble pair *n*%4; 256 bytes per row, 9472 per frame, nibbles in the order a, b, c, d, e, f, g, DP. The write pointer resets on vsync, so a lost or extra byte costs one frame and then corrects.

A digit row is 16 scanlines and 256 bytes, so a host sending **one byte every 66 pixel clocks** tracks the raster exactly. The first byte must follow vsync within about 450 µs.

Reset strap

`ui_in[3:0]` is sampled while `rst_n` is low (the last value before it rises sticks):

Bit	Meaning
`ui_in[0]`	0 = Digilent PmodVGA, 1 = Tiny Tapeout VGA Pmod
`ui_in[3:1]`	palette: 0 grey, 1 blue, 2 green, 3 purple, 4 amber, 5 red, 6 cyan, 7 fire

DIP switches

Can be changed in generator mode only. All off is the default attract mode.

Bit	Off	On
`ui_in[3:1]`	starting palette	palette held in manual mode
`ui_in[4]`	palette cycles	hold the palette `ui_in[3:1]` names
`ui_in[5]`	ring speed wanders	steady ring speed
`ui_in[6]`	drift speed wanders	steady drift

Config packet

With `uio[7]` low, strobes on `uio[6]` carry config bytes. Each fall of `uio[7]` starts a packet. The first valid header also stops the DIP switches being read.

Byte	Contents
0	`{4'hA, load_preset, cycle_en, 0, pmod_type}`; without the `A` the packet is ignored
1–3	red curve: `{knee_x, knee_y}`, `m1`, `m2` (slopes in eighths, 5 bits)
4–6	green, the same
7–9	blue, the same

A palette is three per-channel curves, each two lines through a knee, clipped at 15. A header alone changes the Pmod or turns preset cycling on or off; `load_preset` restores the strapped preset. `tools/palette_builder` designs curves and exports the bytes.

How to test

Generator. Leave `uio[7]` low, apply a 40 MHz clock and release reset. Concentric rings should flow across the grid, with the palette changing every 17 s. Each DIP switch should change what you see within two frames.

Streaming. Set `uio[7]` high and push 9472 bytes per frame on `ui_in`, restarting on each vsync. `tools/video2seg.py` converts a video or image to that format and `firmware/seg_player.py` streams it from the demoboard's RP2350 over PIO and DMA.

Edit seg_player.py to set the Pmod type and palette choice.

External hardware

Digilent PmodVGA across both output headers: R and B nibbles on `uo_out`, G nibble plus hsync and vsync on `uio[5:0]`, 4 bits per channel. Strobe and mode select use `uio[7:6]`, which PmodVGA leaves unconnected.

Tiny Tapeout VGA Pmod: strap `ui_in[0]` high; video moves onto `uo_out` at 2 bits per channel and `uio[5:0]` become inputs.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	stream data 0 / reset strap: 0=Digilent PmodVGA, 1=Tiny VGA Pmod	Digilent: R0 / Tiny VGA: R1	Digilent: G0 (unused, Tiny VGA)
1	stream data 1 / strap+switch: palette bit 0	Digilent: R1 / Tiny VGA: G1	Digilent: G1 (unused, Tiny VGA)
2	stream data 2 / strap+switch: palette bit 1	Digilent: R2 / Tiny VGA: B1	Digilent: G2 (unused, Tiny VGA)
3	stream data 3 / strap+switch: palette bit 2	Digilent: R3 / Tiny VGA: vsync	Digilent: G3 (unused, Tiny VGA)
4	stream data 4 / switch: 1 = hold palette ui[3:1], 0 = change every 512 frames	Digilent: B0 / Tiny VGA: R0	Digilent: hsync (unused, Tiny VGA)
5	stream data 5 / switch: 1 = steady ring speed, 0 = let it wander	Digilent: B1 / Tiny VGA: G0	Digilent: vsync (unused, Tiny VGA)
6	stream data 6 / switch: 1 = steady source drift, 0 = let it wander	Digilent: B2 / Tiny VGA: B0	stream strobe; config strobe while uio[7] is low (same pin in both Pmod modes)
7	stream data 7	Digilent: B3 / Tiny VGA: hsync	1 = stream data, 0 = internal generator + config packets (same pin in both Pmod modes)

CDM Matrix

by **Gian**

`0362` `25.175 MHz` `HDL Project`

`github.com/giannn13/CDM_Matrix`

"Colegio de Muntinlupa MicroElectronics2026!"

How it works

Colegio De Muntinlupa | MicroElectronics2026! Digital Rain

How to test

Glyph mode of Colegio de Muntinlupa

External hardware

Display

Project Pinout

Digital Pins

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

EZ130 8T Mystery Circuit

by **Oscar Castaneda** & **Thomas Herzog**

0363 HDL Project

github.com/ofcastaneda/tt-ez130-8t-mystery

"A mystery circuit implemented with EZ130 8T cells"

How it works

This is a custom-GDS design using the EZ130 8T library from ETH Zurich. The specific design to be implemented remains yet to be a mystery; we will let you know once it is fixed!

How to test

Coming soon once the mystery is revealed!

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SerialIn0	Sum0	Sum8
1	SerialIn1	Sum1	Sum9
2	SerialIn2	Sum2	Sum10
3	SerialIn3	Sum3	HI
4	SerialEn	Sum4	LO
5	InvertIn	Sum5	HI
6	MaskIn	Sum6	SerialOut0
7	WrEnIn	Sum7	RegOut0

Hardware UTF Encoder/Decoder

by **Rebecca G. Bettencourt**

`384` `HDL Project`

`github.com/RebeccaRGB/ttihp-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

VGA Pride

by **Rebecca G. Bettencourt**

`385` `HDL Project`

`github.com/RebeccaRGB/ttihp-vga-pride`

"A VGA demo for showing pride flags"

How it works

Displays pride flags on the screen.

To add another flag, create a flag.v file and add it to `src/flag_index.v`, `test/Makefile`, and `info.yaml`, using the existing flags as examples.

How to test

Connect to a VGA monitor. Set the following inputs to change the displayed flag:

- `ui_in[7]` to display the first flag
- `ui_in[6]` to display the next flag
- `ui_in[5]` to display the previous flag
- `ui_in[4]` to display the flag whose index is on `uio_in`

Index	Flag
0	Rainbow flag, 6 stripes
1	Rainbow flag, 7 stripes
2	Rainbow flag, 8 stripes
3	Rainbow flag, 9 stripes
4	Philadelphia rainbow flag
5	Progress rainbow flag
6	Progress rainbow flag 2021 version
7	Trans pride flag
8	Abrosexual pride flag
9	Aceflux pride flag
10	Aegosexual pride flag
11	Agender pride flag
12	Androgyne pride flag
13	Androsexual pride flag
14	Aporagender pride flag

15	Aroace pride flag
16	Aroflux pride flag
17	Aromantic pride flag
18	Asexual pride flag
19	Aspec pride flag
20	Bigender pride flag (pink purple white purple blue)
21	Bigender pride flag (blue white purple white pink)
22	Bigender pride flag (pink yellow white purple blue)
23	Bisexual pride flag
24	Ceterosexual pride flag
25	Demiandrogyne pride flag (pink purple blue)
26	Demiandrogyne pride flag (green white green)
27	Demiboy pride flag
28	Demifluid pride flag
29	Demiflux pride flag
30	Demigender pride flag
31	Demigirl pride flag
32	Demiromantic pride flag
33	Demisexual pride flag
34	Disability rights flag (gold silver bronze tricolor)
35	Disability rainbow flag
36	Gender-neutral pride flag
37	Genderfluid pride flag
38	Genderflux pride flag
39	Genderqueer pride flag
40	Greygender pride flag
41	Greysexual pride flag
42	Gynosexual pride flag
43	Intersex pride flag (purple circle)
44	Intersex pride flag (blue/pink gradient)
45	Thislesbianlife lesbian pride flag (pink and red)
46	Sadlesbeandisaster lesbian pride flag, 7 stripes (orange and pink)
47	Sadlesbeandisaster lesbian pride flag, 5 stripes (orange and pink)
48	Lydiandragon lesbian pride flag (violet crocus dill rose)

49	Maya Kern lesbian pride flag (violet rose crocus dill)
50	RebeccaRGB femme lesbian pride flag (violet lavender pink rose)
51	Littleender pride flag
52	Maverique pride flag
53	Leonis Ignis MLM pride flag (brown and blue)
54	Vincian MLM pride flag, 7 stripes (green and blue)
55	Vincian MLM pride flag, 5 stripes (green and blue)
56	Vincian MLM pride flag (light blue and light green)
57	Multigender pride flag
58	Multisexual pride flag
59	Neptunic pride flag
60	Neutrois pride flag
61	Nonbinary pride flag
62	Objectum pride flag
63	Omnisexual pride flag
64	Pangender pride flag
65	Pansexual pride flag
66	Polyamory pride flag (blue, red, black with yellow pi)
67	Polyamory pride flag (blue, magenta, purple with yellow heart)
68	Polygender pride flag
69	Polysexual pride flag
70	Pomosexual pride flag
71	Proculsexual pride flag
72	IBM PS/2 pride flag
73	Queer pride flag
74	Trains pride flag (*Train Landscape*, Ellsworth Kelly, 1953)
75	Transfeminine pride flag
76	Transmasculine pride flag
77	Transneutral pride flag
78	Trigender pride flag
79	Unlabeled pride flag
80	Uranic pride flag
81	Voidpunk pride flag
82	Dorian Rutherford butch lesbian pride flag (blue and purple)

83	Butchspace butch lesbian pride flag (orange and yellow)
84	Sapiosexual pride flag (green brown blue)
85	Sapiosexual pride flag (white pink blue)

External hardware

TinyVGA PMOD

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	address mode	R1	A0
1	—	G1	A1
2	—	B1	A2
3	—	VSync	A3
4	set	R0	A4
5	prev	G0	A5
6	next	B0	A6
7	reset	HSync	A7

IEEE_UPB_TT_1

by **Rodolfo Gironda Trujillo**

`0386` `Wokwi Project`

`github.com/rodolf28/ieee_upb_tt_1`

`wokwi.com/projects/475369131246576641`

"A basic digital design (using only logic gates) used in a class for beginners."

How it works

Simple Boolean Logic

How to test

Only follow the Boolean Expression

External hardware

LED display

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	—

SENT Receiver with SPI Interface

by **Philipp Kraft**

`0387` `40 MHz` `HDL Project`

`github.com/philipp-kraft/ttihp-sent2spi`

"Configurable SENT receiver with SPI slave interface"

How it works

The chip receives a SENT signal on `ui[0]` and decodes it into a 32-bit frame, checked against its SAE J2716 CRC-4. The decoded frame and a config register are available over an SPI slave interface on the bidirectional pins: CS (`uio[0]`), MOSI (`uio[1]`), MISO (`uio[2]`) and SCK (`uio[3]`).

Every SPI transaction is an 8-bit command byte (bit 7 = write, bits 6:0 = register address), MSB first, followed by a 32-bit data word MSB first (shifted out on a read, or in on a write):

Address	Register	Access	Contents
`0x00`	`frame_data`	RO	the latest good decoded frame
`0x01`	`config`	RW	bit `[4]`: CRC check enabled (default on); bit `[3]`: pause pulse present after the CRC nibble (default off); bits `[2:0]`: data-nibble count, 1-6 (default 6)
`0x02`	`status`	RO	bit `[1]`: data_error; bit `[0]`: data_valid (same as `uo[1:0]`, see below)

Writes to `config` outside the 1-6 nibble-count range are ignored; every other bit is written as given. Since a write replaces the whole register, always send the full desired value, not just the bit you're changing. Writing `config` mid-frame doesn't disturb a frame already being received; the new settings take effect on the next one.

`uo[0]` (data_valid) is set once at least one frame has been decoded successfully since reset. `uo[1]` (data_error) is set when the most recent frame attempt failed - a bad CRC, an out-of-spec pulse length, or a watchdog timeout waiting on the sensor - and clears again on the next good frame. Both bits are also readable over SPI at `status` for a master with no free GPIO to watch them on directly.

How to test

Drive a SENT signal into `ui[0]`. Once a full frame has been received (`uo[0]` goes high), pull `uio[0]` (CS) low, clock out the command byte `0x00` (read `frame_data`) on `uio[1]` (MOSI), then clock 32 more times to shift the decoded frame out on `uio[2]` (MISO), MSB first.

External hardware

A SENT sensor connected to `ui[0]`, and an SPI master to read out the decoded data.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sent_in	data_valid	SPI CS
1	—	data_error	SPI MOSI
2	—	—	SPI MISO
3	—	—	SPI SCK
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

Smart Traffic Light Controller

by **Kaisy Maina**

0388 | 50 MHz | HDL Project

`github.com/Kaisymaina/smart-traffic-light`

"A smart traffic light controller with pedestrian and emergency controls"

How it works

The smart traffic light controller uses a finite state machine to control the car and pedestrian traffic lights. The car lights follow a sequence of green, yellow, red, and yellow states. A pedestrian button request is remembered by the controller and is serviced when the car light reaches the red state. During pedestrian crossing, the pedestrian light turns green, the car light remains red, and the buzzer provides a repeating audible signal. An emergency button toggles emergency mode, which turns all lights red. Pressing the emergency button again returns the controller to the normal sequence starting from red.

How to test

The design can be tested using the included Cocotb testbench. Fast test mode is enabled using `ui_in[2]` so that the traffic-light timing is shortened during simulation. The testbench checks the normal traffic-light sequence, pedestrian crossing, pedestrian buzzer operation, and emergency toggle behavior.

Run the tests from the `test` directory using:

```bash
make
```

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Pedestrian button	Car red light	—
1	Emergency toggle button	Car yellow light	—
2	Fast test mode	Car green light	—
3	—	Pedestrian red light	—
4	—	Pedestrian green light	—
5	—	Pedestrian buzzer	—
6	—	—	—

#	Input	Output	Bidirectional
7	—	—	—

IEEE Digital EEG Threshold Event Detector

by **Jacob Kebaso**

390 | 10 kHz | HDL Project

`github.com/Kebaso-J/tt_EEG-threshold-detector`

> *" A digital threshold-crossing event detector with hysteresis and a refractory period, intended as the event-detection stage downstream of an external ADC digitizing an EEG signal. No analog pins are used; all thresholding and hysteresis logic is implemented digitally."*

What is this project?

This project is a digital threshold-crossing event detector with hysteresis and a refractory period — effectively a "digital Schmitt trigger" designed to sit downstream of an external ADC that has already digitized a physiological signal such as an EEG channel.

The motivation comes from wearable seizure-detection hardware. A common building block in EEG-based seizure onset detectors is a threshold-crossing detector: raise a flag when a signal feature (raw amplitude, line length, band energy, etc.) exceeds a threshold for long enough to be a real event rather than noise, and only clear that flag once the signal has genuinely settled back down — not the instant it dips below the trigger point. That "settle back down" behaviour is hysteresis, and doing it with two separate thresholds (a high arming threshold and a lower re-arming threshold) is exactly how an analog Schmitt trigger comparator behaves, except implemented here entirely in synchronous digital logic.

No analog pins are used anywhere in this design. All thresholding, hysteresis, and timing logic is implemented as ordinary synthesizable Verilog operating on an 8-bit digital sample bus. This keeps the design low-risk for a first tapeout (fully simulatable and testable with cocotb before submission) while still directly modelling a real analog behaviour in the digital domain.

The project is intended as a standalone building block that could later feed into a larger seizure-detection or neural-event-detection pipeline — the output is a simple digital flag that a downstream microcontroller or state machine can act on (log the event, trigger an alert, wake up a higher-power processing stage, etc.).

How it works

Interface summary

Pin	Direction	Meaning
`ui_in[7:0]`	in	Digitized sample value (normal mode) or configuration value (config mode)
`uio_in[0]`	in	`cfg_mode` — 1 selects a configuration write cycle, 0 selects normal sample processing
`uio_in[2:1]`	in	`cfg_addr` — selects which configuration register is written during a config cycle
`uio_in[7:3]`	in	unused
`uio_out[7:0]`	out	unused, driven low
`uio_oe[7:0]`	out	all zero — every `uio` pin is an input
`uo_out[0]`	out	`event_detected` — high while the detector is in the `DETECTED` state
`uo_out[1]`	out	`armed` — high while the detector is in the `MONITOR` (idle/watching) state
`uo_out[2]`	out	`above_high` — raw comparator debug bit: is the current sample ≥ `th_high`?
`uo_out[3]`	out	`in_refractory` — high while the detector is in the `REFRACTORY` state
`uo_out[7:4]`	out	unused, driven low

Configuration registers

The detector has four 8-bit configuration registers, written one at a time over `ui_in` while `cfg_mode = 1`:

`cfg_addr`	Register	Meaning
`2'b00`	`th_high`	Arming threshold. A sample at or above this value counts toward triggering a detection.
`2'b01`	`th_low`	Re-arming (hysteresis) threshold. Must be set lower than `th_high`. The detection flag is only cleared once the sample drops below this value — not merely below `th_high`.
`2'b10`	`debounce_limit`	Number of consecutive samples that must be at or above `th_high` before a detection is actually declared. This suppresses single-sample noise spikes.

2'b11	refractory_len	Number of clock cycles the detector holds in a refractory (locked-out) state after a detection clears, before it can arm and trigger again.

On reset, sensible defaults are loaded automatically (`th_high = 200`, `th_low = 150`, `debounce_limit = 4`, `refractory_len = 32`) so the design is meaningfully functional even before any configuration writes occur — useful for quick bench testing.

Detection state machine

The core of the design is a three-state finite state machine, evaluated once per clock cycle whenever `cfg_mode = 0`:

1. **MONITOR** (idle / armed): the detector watches incoming samples. Each cycle a sample is at or above `th_high`, an internal counter increments; any sample below `th_high` resets that counter to zero. Once the counter reaches `debounce_limit`, the state machine transitions to `DETECTED`.
2. **DETECTED**: the `event_detected` output is asserted. The detector stays in this state — continuing to report a detected event — for as long as the sample remains at or above `th_low`. This is the hysteresis band: even if the signal dips briefly below `th_high` (but stays above `th_low`), the detection is not cleared prematurely.
3. **REFRACTORY**: once the sample genuinely falls below `th_low`, the detector clears `event_detected` and enters a lockout period lasting `refractory_len` clock cycles, during which no new detection can be declared even if the signal spikes again immediately. This mirrors the refractory period of a biological neuron and prevents rapid re-triggering on a single noisy event. After the countdown reaches zero, the detector returns to `MONITOR` and can arm again.

Why this design avoids analog pins

A "true" analog comparator front end was considered first, but Tiny Tapeout's standard digital IO pads already buffer every `ui_in`/`uio_in` signal through a fixed-threshold (and possibly hysteretic) digital buffer before user logic ever sees it — so a hand-designed transistor-level comparator inside a normal digital tile cannot see a true continuous analog input without a `ua[]` analog pin. Rather than fight that constraint, this project embraces it: it assumes an external ADC (or a companion microcontroller, which is realistic for an actual EEG wearable pipeline anyway) performs the analog-to-digital conversion, and this chip implements the threshold/hysteresis/refractory logic that would otherwise require an analog comparator — entirely in the digital domain, fully synthesizable, and fully verifiable in simulation before submission.

How to test

Simulation (cocotb)

The included cocotb testbench (`test.py`) exercises the full state machine and should be run before submission using the standard Tiny Tapeout test flow (from the project's `test/` directory):

```
make -B
```

The testbench does the following, in order:

1. Applies reset and confirms the detector powers up in the MONITOR (armed) state.
2. Writes all four configuration registers (`th_high=100`, `th_low=60`, `debounce_limit=3`, `refractory_len=5`) using the `cfg_mode`/`cfg_addr` protocol.
3. Drives a run of low-amplitude "baseline" samples and confirms `event_detected` stays low and `armed` stays high.
4. Drives a run of samples above `th_high` for more than `debounce_limit` cycles and confirms `event_detected` asserts.
5. Drives a sample that falls between `th_low` and `th_high` and confirms the detection is not cleared (verifying hysteresis).
6. Drives a sample below `th_low` and confirms the detector transitions to REFRACTORY, clearing `event_detected`.
7. Waits out the refractory period and confirms the detector re-arms (`armed` goes high again).
8. Repeats a high-amplitude run and confirms the detector can re-trigger after re-arming.

Bench testing on real silicon

Once the chip is back and mounted on the Tiny Tapeout demo board:

1. Hold `rst_n` low briefly, then release it — the detector will power up in MONITOR with the default thresholds already active, so you can test immediately without sending any configuration writes.
2. Drive `ui_in[7:0]` from a function generator through an external ADC (or from a microcontroller producing 8-bit digital samples) representing your signal of interest.
3. Watch `uo_out[1]` (armed) and `uo_out[0]` (event_detected) on a logic analyzer or LEDs. Slowly ramp the input above the default `th_high` (200) for a few cycles and confirm `event_detected` goes high; bring it back down below `th_low` (150) and confirm the detector clears and then re-arms after the refractory window.
4. To try custom thresholds, drive `uio_in[0] = 1` with `uio_in[2:1]` set to the desired register address and `ui_in` set to the desired 8-bit value, for

one clock cycle per register, then return `uio_in[0]` to 0 before resuming normal sample streaming.

External hardware

No external hardware is strictly required to exercise the digital logic itself (samples can be driven directly as 8-bit digital values from a microcontroller or test bench). For an end-to-end demonstration against a real analog signal, an external ADC (e.g. an SPI ADC such as the MCP3008, or a microcontroller's built-in ADC) is needed to digitize the analog signal into the 8-bit sample stream expected on `ui_in`.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sample[0]	event_detected	cfg_mode
1	sample[1]	armed	cfg_addr[0]
2	sample[2]	above_high (debug)	cfg_addr[1]
3	sample[3]	in_refractory (debug)	—
4	sample[4]	—	—
5	sample[5]	—	—
6	sample[6]	—	—
7	sample[7]	—	—

UART SPI RAM Loader

by **romd**

`0391` `50 MHz` `HDL Project`

`github.com/4kbyte/ttihp26b-uart-loader`

"CRC-protected UART loader for a 64 KiB SPI SRAM"

How it works

The design exposes every byte of a 23LC512-compatible 64 KiB SPI SRAM through a bounded, CRC-protected UART protocol. It runs from the 50 MHz project clock, uses 115200 baud 8N1 UART, and drives the SRAM in SPI mode 0 at 6.25 MHz.

After reset, the controller writes sequential mode (`WRMR 40`) and reads the mode register back. Wait for `uo_out[0]` to go high before sending memory commands. Reset reinitializes the controller but does not erase SRAM contents. If mode verification fails, `uo_out[7]` goes high and memory commands return a memory-fault response.

How to test

1. Select the project and provide a 50 MHz clock.
2. Connect the SRAM as shown below, then pulse active-low reset.
3. Wait for `uo_out[0]` (`memory ready`) to go high.
4. Open the demoboard USB serial port at 115200 baud, 8N1, with no flow control.
5. Clone the project repository, install PySerial, and run the host CLI from the repository root. The CLI is `tools/uart_loader.py`.

```
python -m pip install pyserial==3.5
python tools/uart_loader.py --port COM4 ping
python tools/uart_loader.py --port COM4 status
python tools/uart_loader.py --port COM4 write 0x100 0x12 0x34
python tools/uart_loader.py --port COM4 read 0x100 2
python tools/uart_loader.py --port COM4 load firmware.bin
python tools/uart_loader.py --port COM4 verify firmware.bin
```

Replace `COM4` with the demoboard serial port, such as `/dev/ttyACM0` on Linux. The default maximum transfer is 16 bytes; the host queries the actual limit with `CAPABILITIES` and chunks larger operations automatically.

The complete wire contract, error codes, CRC definition, and worked PING vector are in the repository protocol documentation. Requests start with `A5`

5A; responses start with 5A A5. Both use CRC-16/CCITT-FALSE over the version-through-payload bytes.

External hardware

Connect a 23LC512-compatible SRAM at a voltage compatible with the demoboard:

Project pin	Direction	SRAM signal
uio[0]	output	CS, active low
uio[1]	output	SI / MOSI
uio[2]	input	SO / MISO
uio[3]	output	SCK

Connect common power and ground, and hold the SRAM HOLD input high when the hold function is unused. UART RX is ui_in[3]; UART TX is uo_out[4]. These are the demoboard USB serial receive and transmit paths, respectively.

Status outputs

Output	Meaning
uo_out[0]	SRAM initialization succeeded
uo_out[1]	Memory transaction active
uo_out[2]	Sticky UART framing error
uo_out[3]	UART receive FIFO has capacity
uo_out[4]	UART TX
uo_out[5]	Sticky UART receive overrun
uo_out[6]	Sticky protocol validation error
uo_out[7]	SRAM initialization fault

The sticky error outputs clear on electrical reset.

Layout

- TinyTapeout 3D viewer
- GDS Explorer

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Unused	SPI memory ready	SPI SRAM chip select, active low

#	Input	Output	Bidirectional
1	Unused	Memory transaction active	SPI SRAM MOSI
2	Unused	UART framing error	SPI SRAM MISO
3	UART RX	UART RX has capacity	SPI SRAM clock
4	Unused	UART TX	Unused
5	Unused	UART RX overrun	Unused
6	Unused	Protocol error	Unused
7	Unused	SPI memory fault	Unused

8N1 UART Transceiver

by **NUAT Labs**

0392 HDL Project

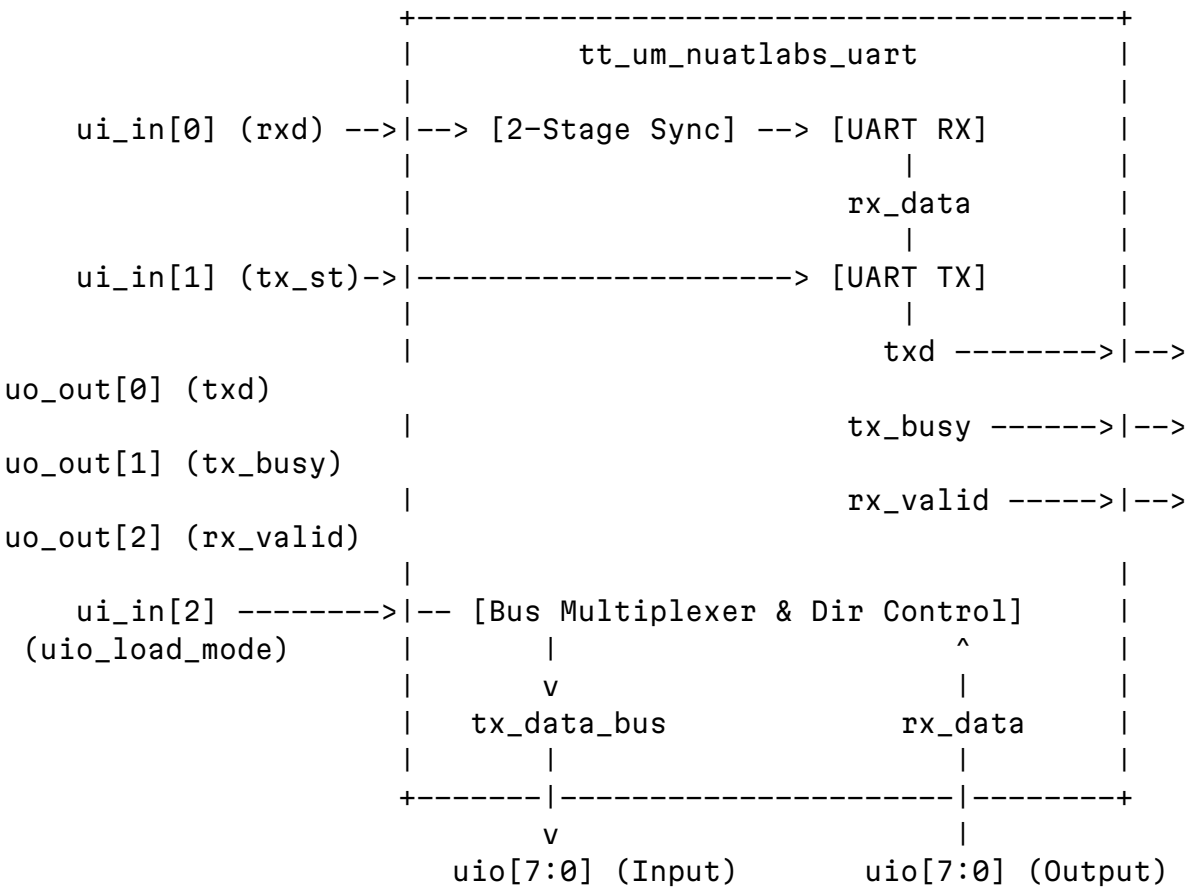
github.com/nuatlabs/UART_TT

"Full-duplex 8N1 UART transceiver (TX + RX) with runtime bidirectional data bus and mid-bit sampling"

How it works

The **tt_um_nuatlabs_uart** design is a full-duplex, 8N1 (8 data bits, no parity, 1 stop bit) Universal Asynchronous Receiver-Transmitter (UART) peripheral hardened on the 130nm IHP SG13G2 BiCMOS process via Tiny Tapeout. It fits into a single standard tile (1x1) and features a runtime time-multiplexed bidirectional data bus that allows full 8-bit parallel data transfer using Tiny Tapeout's 8 shared GPIO pins (uio).

Architecture Overview



1. **UART Transmitter (uart_tx)**:
   - Implemented as a clean 3-block Moore Finite State Machine (ST_IDLE, ST_START, ST_DATA, ST_STOP).

- Latches `tx_data` from the `uio_in` bus upon detecting a single-cycle high pulse on `ui_in[1]` (`tx_start`).
- Drives `uo_out[1]` (`tx_busy`) high throughout transmission.
- Frames the byte by outputting a low start bit (0), shifting 8 data bits out (LSB first), and terminating with a high stop bit (1) onto `uo_out[0]` (`txd`).
- Bit duration is governed by the parameter `BAUD_DIV = f_clk / baud_rate`.

2. **UART Receiver (`uart_rx`)**:
   - Asynchronous serial input `ui_in[0]` (`rxd`) passes through a 2-stage flip-flop synchronizer (`rxd_sync1`, `rxd_sync2`) to eliminate metastability risks.
   - Falling edge detection initiates the start bit sequence.
   - Samples incoming data at the exact midpoint of each bit interval (`BAUD_DIV / 2`), providing maximum noise immunity and tolerance against clock jitter.
   - Shifts incoming bits into an internal shift register (LSB first).
   - Upon successful verification of the stop bit, latches the parallel byte into `rx_data` and asserts `uo_out[2]` (`rx_valid`) for exactly one clock cycle.
3. **Time-Multiplexed Bidirectional Data Bus (`uio[7:0]`)**:
   - Because standard Tiny Tapeout tiles provide 8 dedicated inputs, 8 dedicated outputs, and 8 bidirectional IOs, dedicating 16 separate pins for parallel TX and RX bytes is impossible.
   - This design uses `ui_in[2]` (`uio_load_mode`) to dynamically govern the direction of the bidirectional `uio` bank:
     - **Load / Transmit Mode (`uio_load_mode = 1`)**: Sets `uio_oe = 8'h00` (input mode). The external host or microcontroller presents the 8-bit byte to transmit on `uio_in[7:0]`.
     - **Read / Display Mode (`uio_load_mode = 0`)**: Sets `uio_oe = 8'hFF` (output mode). The chip drives the last successfully received byte onto `uio_out[7:0]`.

Baud Rate Configuration

The hardware divider is parameterized via `BAUD_DIV = f_clk / baud_rate`:

- In this build, `BAUD_DIV` is defaulted to 4 for rapid, deterministic simulation in Cocotb.
- For physical silicon running at a typical devkit clock (e.g., 50 MHz internal clock, or 10 MHz), set the external clock frequency or divider according to the target baud (e.g. 9600, 115200, 1Mbaud).

How to test

1. Verification in Simulation (Cocotb)

Run the automated test suite from the repository root:

```
cd test
make clean
make SIM=icarus
```

The testbench validates 3 comprehensive scenarios:

- **`test_uart_tx_shape`**: Checks bit timing and serial frame shape: start bit (0), 8 data bits (LSB-first), stop bit (1), and active duration of `tx_busy`.
- **`test_uart_loopback_single_byte`**: Emulates an external jumper wire by dynamically copying `uo_out[0]` (`txd`) to `ui_in[0]` (`rxd`) cycle-by-cycle, verifying full transmission, reception, and reconstruction.
- **`test_uart_loopback_multiple_bytes`**: Stress tests back-to-back transmissions across boundary and alternating values (`0x00`, `0xFF`, `0x55`, `0xAA`, `0x81`, `0x7E`).

2. Physical Hardware Testing on Tiny Tapeout Devkit

A. Basic Bench Loopback Test (No External Tools Required)

1. Connect a standard breadboard jumper wire between pin `uo[0]` (`txd`) and pin `ui[0]` (`rxd`).
2. Set DIP switch `ui[2]` (`uio_load_mode`) to `1` (input mode).
3. Set the 8 DIP switches on `uio[7:0]` to the desired 8-bit test byte (e.g., `0xA5` / `10100101`).
4. Pulse DIP switch or push button `ui[1]` (`tx_start`) high then low.
5. Observe `uo[1]` (`tx_busy`) LED toggle high while transmitting.
6. Observe `uo[2]` (`rx_valid`) LED pulse when reception completes.
7. Flip switch `ui[2]` (`uio_load_mode`) to `0` (output mode).
8. The 8 LEDs connected to `uio[7:0]` will display the received byte, matching the byte sent in step 3.

B. Interfacing with a Host PC (USB-to-UART Adapter)

1. Connect a 3.3V USB-to-UART bridge (FTDI, CP2102, or RP2040):
   - PC TXD -> Tiny Tapeout `ui[0]` (`rxd`)
   - Tiny Tapeout `uo[0]` (`txd`) -> PC RXD
   - Common Ground (GND -> GND).
2. Open a terminal (PuTTY, minicom, screen, or PySerial) at the baud rate configured for your clock frequency.
3. Transmit characters from PC to chip or trigger transmissions from chip to PC.

External hardware

- **Minimal Loopback Demo**: A single jumper wire connecting `uo[0]` to `ui[0]`.
- **Host PC Interfacing**: 3.3V USB-to-UART bridge adapter (CP2102, FT232R, or Raspberry Pi Pico).

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	rxd (serial input)	txd (serial output)	tx_data[0] (in) / rx_data[0] (out)
1	tx_start (transmit start pulse)	tx_busy (transmitter busy flag)	tx_data[1] (in) / rx_data[1] (out)
2	uio_load_mode (1=load tx_data, 0=show rx_data)	rx_valid (receiver valid pulse)	tx_data[2] (in) / rx_data[2] (out)
3	—	—	tx_data[3] (in) / rx_data[3] (out)
4	—	—	tx_data[4] (in) / rx_data[4] (out)
5	—	—	tx_data[5] (in) / rx_data[5] (out)
6	—	—	tx_data[6] (in) / rx_data[6] (out)
7	—	—	tx_data[7] (in) / rx_data[7] (out)

Async FIFO with CDC + PWM Peripheral

by **NUAT Labs**

394 HDL Project

`github.com/nuatlabs/Fifo_TT`

"8-deep async FIFO with Gray-code CDC and live 7-seg occupancy display, plus a small reusable PWM peripheral"

How it works

This project packs two independent digital blocks into one Tiny Tapeout tile:

1. **Async FIFO with Gray-code CDC.**

An 8-entry, 4-bit-wide FIFO whose write side and read side can run on two completely independent, unrelated clocks. Correctness across the clock-domain crossing is guaranteed by the classic Gray-coded pointer + 2-flop synchronizer technique: both the write and read pointers are kept in both binary (for arithmetic) and Gray-code (for safe crossing, since only one bit ever changes between consecutive values) form, and each domain sees the *other* domain's Gray pointer only after it has passed through two back-to-back flip-flops. The current occupancy (0-8) is shown live on a 7-segment display.

2. **PWM peripheral.**

A free-running 8-bit counter compared against a 4-bit duty register produces a 16-step PWM waveform. The duty value is loaded from a 4-bit input bus on a load pulse. It's deliberately written as a small, self-contained, reusable block – the same RTL is a candidate to be dropped straight into a larger RISC-V SoC later as a memory-mapped timer/PWM peripheral.

How to test

No extra hardware needed (recommended first test):

1. Set `ui_in[4] = 1` (read domain uses the main devkit clock – a synchronous fallback mode, so you don't need a second clock source).
2. Put a 4-bit value on `uio_in[3:0]`, pulse `ui_in[0]` high for one clock to write it into the FIFO. Watch the 7-segment display (`uo_out[6:0]`) count up.
3. Pulse `ui_in[1]` high to read a word back out (available on `uio_out[7:4]`); watch the 7-segment count back down.
4. To exercise the PWM block: put a duty value (0-15) on `uio_in[3:0]`, pulse `ui_in[2]` to load it, then set `ui_in[3] = 1` to enable. `uo_out[7]` will show the PWM waveform (scope or LED).

True async CDC test (needs a second clock source):

1. Set `ui_in[4] = 0`.
2. Feed an independent clock into `ui_in[7]` (a second RP2040/Pico GPIO, a bench signal generator, or even a slow hand-toggled switch both work) – this becomes the FIFO's read-domain clock.
3. Write on the main clock as above, read using `ui_in[1]` pulses timed to the external clock on `ui_in[7]`. The occupancy display correctly lags by a couple of read-clock edges after each write – that lag *is* the CDC synchronizer working as intended, not a bug.

External hardware

None required for the basic demo (everything above runs on the stock Tiny Tapeout devkit's switches, LEDs and 7-segment display). For the true asynchronous CDC demonstration, an external clock source wired into `ui_in[7]` is optional but recommended.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	fifo_wr_en	SEG_A	fifo_wr_data[0] / pwm_duty_in[0]
1	fifo_rd_en	SEG_B	fifo_wr_data[1] / pwm_duty_in[1]
2	pwm_duty_load	SEG_C	fifo_wr_data[2] / pwm_duty_in[2]
3	pwm_enable	SEG_D	fifo_wr_data[3] / pwm_duty_in[3]
4	fifo_rd_clk_sel (1=use main clk, 0=use ui_in[7])	SEG_E	fifo_rd_data[0] (output)
5	—	SEG_F	fifo_rd_data[1] (output)
6	—	SEG_G	fifo_rd_data[2] (output)
7	ext_rd_clk (optional 2nd clock for true CDC demo)	pwm_out	fifo_rd_data[3] (output)

Hepiarisc with SPI flash

by **Léon Romanens**

`0395` `100 MHz` `HDL Project`

`github.com/leonllrmc/ttihp26b_llr_hepiarisc`

> *"A Hepiarisc processor fetching instructions over a SPI flash"*

How it works

This is an implementation of the **HEPIARISC** ISA used by HEPIA to teach basic CPU architecture. Here are a summary of the specs

You can find a customasm base file at customasm_sample.s

MMIO Periherals

TBD

How to test

You can compile a program with customasm, then flash the binary in big endian to the flash, and enjoy !

External hardware

A SPI flash, any that can be read with the sequence: [0x03, A[23:16], A[15:8], A[7:0], D0, D1]

TODO

- test RAM
- add GPIO peripheral
- add SPI/I2C
- add "bank switch instruction"
- programmable IRQ timer (EN (either none, timer or ext IRQ), div 8-256,)

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	extflash_spi_miso	project_extflash_spi_sck	—
1	irq_n	project_extflash_spi_mosi	—
2	—	project_extflash_spi_cs	—
3	—	—	—
4	GPI_0	GPO_0	GPIO_0

#	Input	Output	Bidirectional
5	GPI_1	GPO_1	GPIO_1
6	GPI_2	GPO_2	GPIO_2
7	GPI_3	GPO_3	GPIO_3

1D CA/GO/SO CFAR radar detector

by **Lars Nitschke**

`0416` `50 MHz` `HDL Project`

`github.com/larsnit/tt-cfar`

> "*A constant false alarm rate detector for pulse radar, processing one 8-bit magnitude sample per clock and emitting one detection decision per range cell, with cell-averaging, greatest-of and smallest-of variants selectable at runtime.*"

How it works

A constant false alarm rate (CFAR) detector for pulse radar. One unsigned 8-bit magnitude sample enters per clock, one per range cell. A 21-stage shift register forms a sliding window: 8 training cells each side, 2 guard cells each side, and the cell under test in the middle. Two incremental half-sums track the training groups, updated by adding the entering cell and subtracting the leaving one, so cost does not grow with window size. A mode multiplexer selects cell-averaging (both halves), greatest-of, smallest-of, or a single-sided half for use at profile boundaries; the single-sided modes double the chosen half so every mode presents a nominal 16-cell sum. The threshold is a hardwired shift-add multiple of that sum, selected by three pins, and the comparison is done at 19 bits with no multiplier and no divider. Latency is a fixed 12 clock cycles.

How to test

Hold `rst_n` low, then release it. Assert `sop` high for one cycle concurrent with the first sample of a range profile, with `mode` and `alpha_sel` valid on the same cycle — `sop` must be asserted at least once after reset before any output is meaningful. Feed one 8-bit magnitude sample per clock on `ui_in`; every clock is treated as a sample and there is no data-valid input. The `valid` output rises 22 cycles after `sop` and detections are meaningful from then on. The host must supply 10 cells of margin either side of the region of interest, so for 512 cells of interest clock in 532 samples. Note `valid` protects the leading edge only and cannot detect the end of a profile.

External hardware

None. Magnitude samples are provided digitally from off-chip; there is no on-chip RF or ADC.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sample[0]	det	mode[0]
1	sample[1]	valid	mode[1]
2	sample[2]	half_sel	mode[2]
3	sample[3]	v_dbg[0]	alpha_sel[0]
4	sample[4]	v_dbg[1]	alpha_sel[1]
5	sample[5]	v_dbg[2]	alpha_sel[2]
6	sample[6]	v_dbg[3]	sop
7	sample[7]	v_dbg[4]	—

Circuitli C061618G2

by **circuitli**

0417 HDL Project

`github.com/circuitli/tt_um_c061618g2`

"A clockless MMU with an integrated anti-glitch filter network for 8-bit processors with a 16-bit address bus."

How it works

This project is a Memory Management Unit (MMU). It handles memory mapping, peripheral address decoding, and operating system banking across the computer's memory map.

- **Address Decoding:** The circuit accepts high-order CPU address lines (A11 to A15) alongside primary system configuration flags to determine which physical hardware target should be active during a memory read/write cycle.
- **Banking Logic & PORTB:** It actively monitors control inputs. This includes decoding signals to dynamically switch the internal 16KB Operating System ROM, the BASIC ROM, and the Self-Test ROM in and out of the memory map, freeing up underlying DRAM when disabled.
- **Peripheral Mapping:** The internal combinatorial decoding grid maps out active-low chip select signals for custom sub-systems and the master RAM network.

This design acts as a purely combinatorial logic array.

How to test

This project is currently validated using an automated simulation environment (Cocotb or HDL testbench). You can run the testbench and inspect the generated waveform file (VCD) using the following verification steps:

1. **Initialize Simulation:** Run the test suite using `make test'.
2. **Verify Basic OS ROM Mapping:**
   - In the testbench stimulus, drive the address bus inputs `A[15:11]` to `5'b11111` (representing memory space `$F800–$FFFF`).
   - Assert the `REN` (ROM Enable) control input high.
   - Inspect the waveform viewer to verify that the OS Chip Select output (OS_L) and CI_L drop to 0.
3. **Verify BASIC Banking Logic:**
   - Drive the address inputs to `5'b10101` (representing memory space `$A000–$BFFF`).

- Toggle the `BE_L` simulation signal to 0.
- Assert that BASIC_L falls to 0. Then, drive BE_L to 1 and verify the output switches back to exposing the underlying system memory by resetting CI_L` to 1.

4. **I/O Device Matrix Test:**
   - Force the address lines to match the hex-equivalent binary for `$D300`.
   - Check the simulation logs or waveform to confirm that only the IO_L output line drops to logic 0.

External hardware

None for now.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A11	REN	S5_L
1	A12	REF_L	BASIC_L
2	A13	MPD_L	OS_L
3	A14	BE_L	CI_L
4	A15	—	IO_L
5	MAP_L	TRIGGER_OUT_L	S4_L
6	RD4	FLG_IN_L	FLG_L
7	RD5	—	—

LayerNorm

by **Jakubie**

`419` `1 MHz` `HDL Project`

`github.com/jakubiee/LayerNorm`

"8-element LayerNorm accelerator"

How it works

This project implements a simplified LayerNorm accelerator for eight signed 8-bit integer samples. It calculates an integer mean and variance, then scales each sample's deviation from the mean using a reciprocal square-root lookup table.

The reciprocal square-root lookup table supports integer variances from 1 to 15 and uses Q0.9 fixed-point coefficients. A variance of 0 or a variance greater than 15 produces zero outputs. Results are signed 8-bit integers truncated toward zero.

The design does not implement epsilon, learnable scale or bias, or overflow protection.

Interface

- ui_in[7:0]: signed input sample (two's complement).
- uio_in[0] — VALID: accepts one sample per rising clock edge during input collection.
- uio_in[1] — START: starts a new operation while idle.
- uo_out[7:0]: signed output sample.

All bidirectional pins are configured as inputs. There is no output-valid or busy signal.

How to test

1. Hold rst_n low for at least one rising clock edge, then release it.
2. Pulse START high for one clock cycle.
3. Starting on the next rising edge, supply eight samples with VALID high. Set VALID low afterward.
4. Taking the eighth sample's capture edge as cycle 0, read outputs after rising edges 35, 39, 43, 47, 51, 55, 59 and 63, allowing signals to settle.
5. The design returns to idle at cycle 64. A new START can be accepted at cycle 65.

Outputs appear in input order and remain available for one clock cycle each. The output bus is zero between results; zero is also a valid result.

Example input: [0, 1, 2, 3, 4, 5, 6, 7] Expected output: [−1, 0, 0, 0, 0, 0, 1, 1]

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA[0]	OUTPUT[0]	VALID
1	DATA[1]	OUTPUT[1]	START
2	DATA[2]	OUTPUT[2]	—
3	DATA[3]	OUTPUT[3]	—
4	DATA[4]	OUTPUT[4]	—
5	DATA[5]	OUTPUT[5]	—
6	DATA[6]	OUTPUT[6]	—
7	DATA[7]	OUTPUT[7]	—

QAMER CryptoUART: Encrypted UART with LED Status

by **QAMER**

421 | 50 MHz | HDL Project

`github.com/MaverickSemiUSA/tt_um_QAMER-CryptoUART`

"UART echo with LFSR-based byte encryption and LED activity status."

How it works

QAMER CryptoUART is an encrypted UART demonstration that receives serial data, echoes the received plaintext byte, and then transmits an encrypted version of the same byte.

The design operates at a default clock frequency of 50 MHz and a UART baud rate of 9600.

The 8-bit `ui_in` input is used as follows:

- `ui_in[0]` is the UART RX input.
- `ui_in[7:1]` provides a 7-bit key seed used to initialize the LFSR.

After reset, the LFSR is initialized from the 7-bit key seed. For each received UART character, the current 8-bit LFSR keystream value is XORed with the received byte:

```
ciphertext = plaintext XOR keystream
```

The design then:

1. Receives a UART byte on `ui_in[0]`.
2. Generates the corresponding encrypted byte using the LFSR keystream.
3. Echoes the original plaintext byte through `uo_out[0]`.
4. Transmits the encrypted ciphertext byte through `uo_out[0]`.
5. Updates the LFSR for the next received character.
6. Updates the four LED status outputs after each received character.

The four LED outputs build up as characters are received:

- Character 1: `0001`
- Character 2: `0011`
- Character 3: `0111`
- Character 4: `1111`

After four characters, the LED sequence starts again with the next group.

Additional status signals are provided on `uo_out`:

- `uo_out[5]`: RX valid pulse
- `uo_out[6]`: UART TX busy status
- `uo_out[7]`: Ciphertext valid pulse

The encrypted byte is also continuously available on `uio_out[7:0]`. The `uio_oe` signal is fixed to `8'hFF`, so all eight `uio` pins operate as outputs.

How to test

Connect a UART transmitter to `ui_in[0]` and use the default UART settings:

- Clock: 50 MHz
- Baud rate: 9600
- Data: 8 bits
- Start bit: 1
- Stop bit: 1

Provide the 7-bit encryption key through `ui_in[7:1]`. The key is sampled when the design initializes after reset.

For example, the supplied functional test uses:

```
KEY_SEED = 7'h55
```

and transmits the characters:

- `A` (`0x41`)
- `B` (`0x42`)

For each character, the design first sends the plaintext byte back through the UART TX output and then sends the corresponding encrypted byte.

The expected ciphertext is calculated using the LFSR keystream:

```
ciphertext = plaintext XOR keystream
```

The ciphertext can also be observed directly on `uio_out[7:0]`.

The four LED outputs indicate the number of characters received within the current group of four. The RX-valid and ciphertext-valid signals can be used as one-cycle indicators when the corresponding data becomes available.

The automated Cocotb testbench verifies reset behavior, UART plaintext echo, ciphertext generation, LFSR progression, LED status, the live ciphertext output, and the `uio_oe` configuration.

External hardware

No external hardware is required for simulation.

For physical demonstration, a standard USB-to-UART adapter can be connected to the UART RX/TX signals. The four LED status outputs can be

connected to LEDs through appropriate current-limiting circuitry if the target hardware does not already provide LEDs.

The encryption key is provided through the seven available `ui_in[7:1]` input pins.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	UART_RXD	UART_TXD	CIPHERTEXT[0]
1	KEY_SEED[0]	LED[0]	CIPHERTEXT[1]
2	KEY_SEED[1]	LED[1]	CIPHERTEXT[2]
3	KEY_SEED[2]	LED[2]	CIPHERTEXT[3]
4	KEY_SEED[3]	LED[3]	CIPHERTEXT[4]
5	KEY_SEED[4]	RX_VALID	CIPHERTEXT[5]
6	KEY_SEED[5]	TX_BUSY	CIPHERTEXT[6]
7	KEY_SEED[6]	CIPHER_VALID	CIPHERTEXT[7]

DSP MAC Engine

by **Maaz Ahmed Khan**

0423 | 50 MHz | HDL Project

github.com/MaverickSemiUSA/tt_um_Maaz-Ahmed-Khan-DSP-MAC-Engine

"8x8 signed multiply-accumulate engine with byte-serial I/O"

How it works

The DSP MAC Engine is an 8-bit signed multiply-accumulate datapath.

Two signed 8-bit operands are loaded through the `ui_in` input bus using control signals on `uio_in`. When `mac_en` is asserted, the two operands are multiplied to produce a signed 16-bit product. The product is sign-extended to 24 bits and added to the internal 24-bit accumulator.

The accumulator can be cleared using `clr_acc`. Its 24-bit value can be read one byte at a time through `uo_out` using the `rd_next` control signal.

The input and control signals are:

- `ui_in[7:0]`: 8-bit signed operand data bus
- `uio_in[0]`: `load_a` — loads the first operand
- `uio_in[1]`: `load_b` — loads the second operand
- `uio_in[2]`: `mac_en` — performs the multiply-accumulate operation
- `uio_in[3]`: `clr_acc` — clears the accumulator
- `uio_in[4]`: `rd_next` — advances the accumulator byte being read
- `uio_in[7:5]`: unused

The 24-bit accumulator is presented through `uo_out[7:0]` one byte at a time. This allows the complete accumulated result to be read using the 8-bit Tiny Tapeout output bus.

The design operates synchronously from the Tiny Tapeout `clk` input and uses an active-low reset through `rst_n`.

How to test

The design can be tested by applying a clock and reset, loading two signed 8-bit operands, and then enabling the MAC operation.

For example, to perform:

`3 × 5 = 15`

load `3` into operand A using `load_a`, load `5` into operand B using `load_b`, and assert `mac_en` for a clock cycle.

The resulting accumulator value can then be read through `uo_out[7:0]` using `rd_next`.

The supplied Cocotb testbench performs reset, loads A = 3 and B = 5, executes one multiply-accumulate operation, and verifies that the resulting value is 15.

The accumulator can also be cleared using `clr_acc` before starting a new calculation.

External hardware

No external hardware is required for simulation.

For a physical demonstration, the `ui_in` bus can be driven by an external digital controller or FPGA, while the `uio_in` control signals can be used to control operand loading, MAC execution, accumulator clearing, and result readout.

The 8-bit `uo_out` bus can be connected to a logic analyzer, FPGA, or other digital interface to observe the accumulator result.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	operand data bus bit 0	accumulator byte out 0	load_a (strobe)
1	operand data bus bit 1	accumulator byte out 1	load_b (strobe)
2	operand data bus bit 2	accumulator byte out 2	mac_en (strobe)
3	operand data bus bit 3	accumulator byte out 3	clr_acc (strobe)
4	operand data bus bit 4	accumulator byte out 4	rd_next (strobe)
5	operand data bus bit 5	accumulator byte out 5	—
6	operand data bus bit 6	accumulator byte out 6	—
7	operand data bus bit 7	accumulator byte out 7	—

4-Tap Signed MAC Unit

by **Mohammed Abdul Mujeeb**

0425 | 50 MHz | HDL Project

`github.com/MaverickSemiUSA/tt_um_mohammed_abdul_mujeeb_4tap_mac`

"Four-tap signed multiply-accumulate unit for DSP and dot-product operations."

How it works

The **4-Tap Signed MAC Unit** performs four signed 4-bit multiplication operations and combines their results into a multiply-accumulate result.

The operation is:

`Y = (W0 × X0) + (W1 × X1) + (W2 × X2) + (W3 × X3)`

Four signed 4-bit weights and four signed 4-bit samples are stored internally. Each corresponding weight and sample are multiplied to produce an 8-bit signed product. The four products are then added together.

The design also supports a running accumulation mode. When `ACCUMULATE` is asserted, the newly calculated MAC sum is added to the previously stored accumulator value. Otherwise, the current MAC operation replaces the accumulator value.

Input and control signals

Pin	Function
`ui_in[1:0]`	Selects one of the four MAC taps
`ui_in[2]`	`LOAD_WEIGHT` — loads a weight into the selected tap
`ui_in[3]`	`LOAD_SAMPLE` — loads a sample into the selected tap
`ui_in[4]`	`START_MAC` — starts the MAC calculation
`ui_in[5]`	`ACCUMULATE` — adds the new result to the existing accumulator
`ui_in[6]`	`CLEAR_ACC` — clears the accumulator and result
`ui_in[7]`	Unused
`uio_in[3:0]`	Signed 4-bit data input
`uio_in[7:4]`	Unused

The calculated result is available on `uo_out[7:0]`. The internal accumulator is 12 bits wide, while the lower 8 bits are provided at the output.

The design uses an active-low reset through `rst_n` and operates synchronously with the Tiny Tapeout `clk` input.

How to test

First apply reset to initialize the four weights, four samples, accumulator, and output.

Loading a weight

1. Select the desired tap using `ui_in[1:0]`.
2. Place the signed 4-bit weight on `uio_in[3:0]`.
3. Assert `ui_in[2]` (`LOAD_WEIGHT`) for one clock cycle.

Loading a sample

1. Select the desired tap using `ui_in[1:0]`.
2. Place the signed 4-bit sample on `uio_in[3:0]`.
3. Assert `ui_in[3]` (`LOAD_SAMPLE`) for one clock cycle.

Repeat these operations for all four taps.

After loading the weights and samples, assert `ui_in[4]` (`START_MAC`) to perform the calculation.

For example:

```
Weights = [1, 2, 3, 4]
Samples = [1, −1, 2, −2]
```

The expected MAC result is:

```
(1 × 1) + (2 × −1) + (3 × 2) + (4 × −2)
= 1 − 2 + 6 − 8
= −3
```

The signed result `−3` is represented in two's complement. Its lower 8 bits can be observed on `uo_out[7:0]`.

To perform another MAC operation while retaining the previous result, assert `ui_in[5]` (`ACCUMULATE`). To clear the stored result before a new calculation, assert `ui_in[6]` (`CLEAR_ACC`).

External hardware

No external hardware is required for simulation.

For a physical demonstration, the `ui_in` control signals and `uio_in[3:0]` data bus can be driven by an FPGA, microcontroller, or other digital controller.

The `uo_out[7:0]` bus can be connected to a logic analyzer or other digital interface to observe the MAC result.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	TAP_SEL[0]	MAC_RESULT[0]	DATA_IN[0]
1	TAP_SEL[1]	MAC_RESULT[1]	DATA_IN[1]
2	LOAD_WEIGHT	MAC_RESULT[2]	DATA_IN[2]
3	LOAD_SAMPLE	MAC_RESULT[3]	DATA_IN[3]
4	START_MAC	MAC_RESULT[4]	UNUSED
5	ACCUMULATE	MAC_RESULT[5]	UNUSED
6	CLEAR_ACC	MAC_RESULT[6]	UNUSED
7	UNUSED	MAC_RESULT[7]	UNUSED

PRISM with Risc-V (TinyQV) SoC

by **Ken Pettit with Michael Bell's TinyQV**

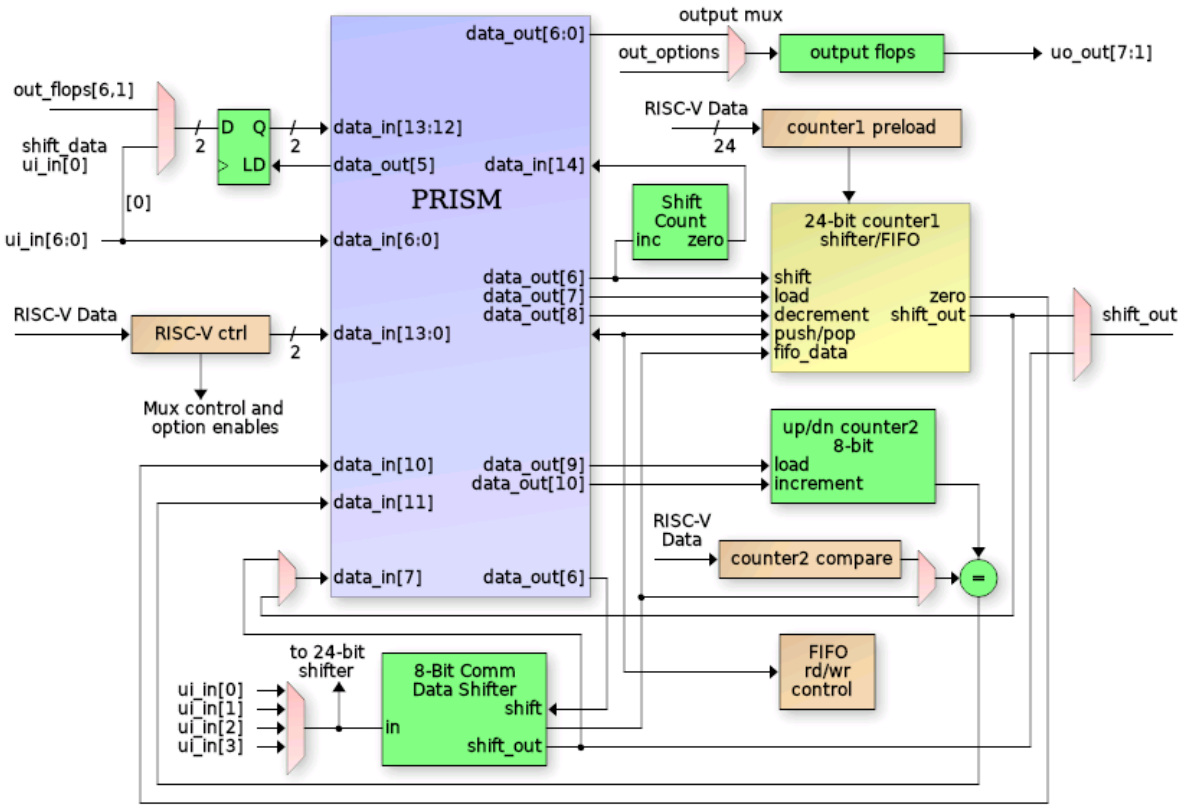
8426 | 64 MHz | HDL Project

`github.com/kdp1965/ihp26b-um-pettit-prism-lite`

"A PRISM controller suported by a TinyQV Risc-V SoC"

How it works

This tile is **PRISM**, a Programmable Reconfigurable Indexed State Machine: a protocol engine whose states, transitions, inputs and outputs are loaded at run time as a bitstream compiled from an ordinary Verilog state machine. A small RISC-V core, TinyQV, sits beside it as the host processor. PRISM does the bit- and byte-level timing of a protocol in hardware, one state transition per clock; the CPU loads the state machine, configures the datapath, moves data through the FIFOs and runs the protocol's upper layers. Protocols that have run on it in simulation include UART, SPI (slave, and single- or quad-lane master), I2C (master and slave), 1-Wire, WS2812, a quadrature encoder, a 24-bit GPIO expander and a USB low-speed device; the 8x4 build with SRAM FIFOs adds 10BASE-T Ethernet transmit and receive.

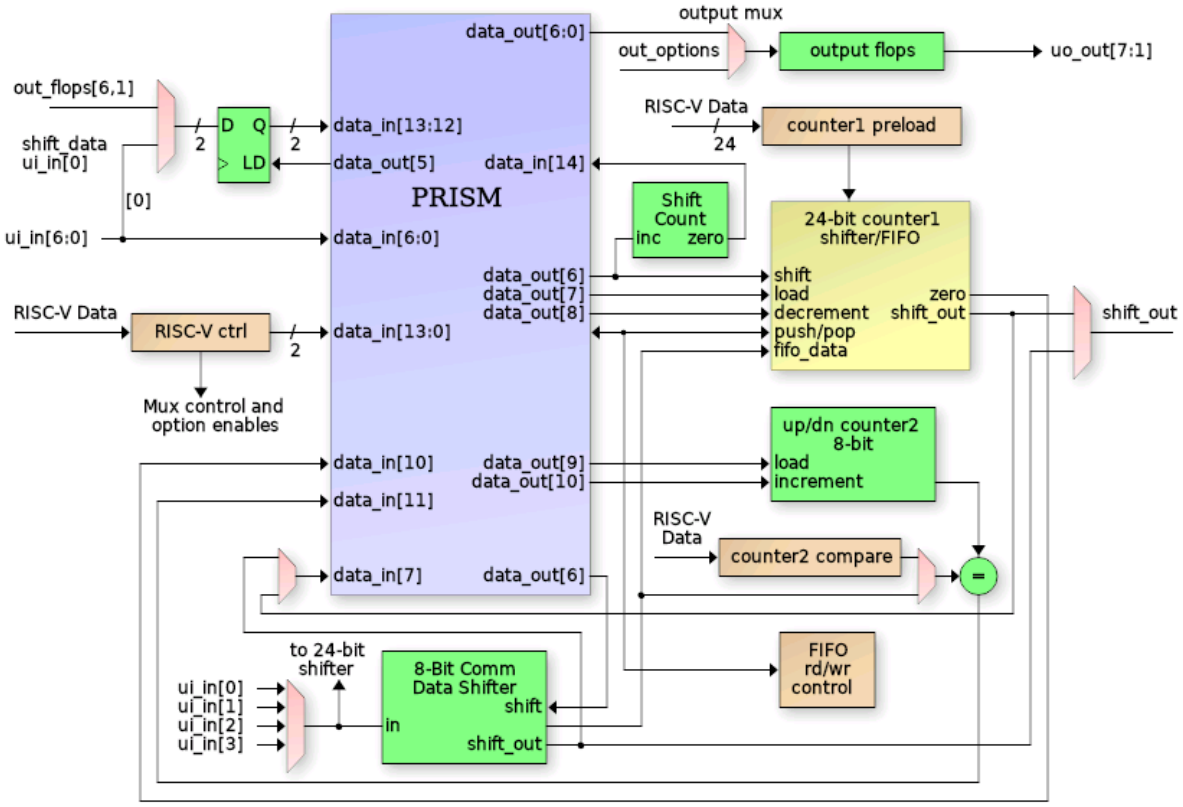


What is on the tile

- **The PRISM core**: 32 states, each described by a 128-bit State Execution Word (STEW). Six input muxes pick from 32 inputs; two 3-input LUTs form an `if / else if / else` decision per state; each branch drives its own set of 21 outputs; two more 2-input LUTs drive conditional outputs that follow the inputs within a state.
- **Two shards**: the machine can be fractured into two independent 16-state state machines, each with its own state index, inputs, outputs, datapath, FIFOs and host interrupt, talking to each other through semaphores. Unfractured, one 32-state machine owns everything.
- **A datapath per shard**: a 24/32-bit counter that doubles as a wide shift register, an 8-bit counter with compare, an 8-bit communication shifter (1 to 4 bits per shift), four constants, a 64-byte FIFO (32 in the smaller build), a CRC unit (8/16/32) that doubles as a 32-bit up/down counter with compare, a Manchester bit recoverer, an edge-clocked sampler, edge-capture flops and a second timer.
- **CFGMEM**: eight latch-array macros (16 rows x 32 bits each) built with a DFFRAM-style flow hold the state table; the CPU loads them through a shift chain per bank.
- **A debugger** with halt, single step and condition-qualified breakpoints per shard. This 5x4 tile is the SRAM-less build of the design: the 8x4 tile adds a 2 KB SRAM FIFO per shard and an execution tracer into them.
- **TinyQV**: RV32EC + Zcb + Zicond, code from a QSPI flash and data in QSPI PSRAM, with a UART, a debug UART and GPIO.

PRISM: the Programmable Reconfigurable Indexed State Machine

PRISM executes a state machine that is loaded at run time. The state machine is written as an ordinary Verilog Mealy FSM, a **chroma**, and compiled by a Yosys backend into a table of 32 **State Execution Words** (STEWs) of 128 bits plus the configuration words for the datapath. The host CPU loads the table, configures the datapath, enables the machine, and from then on PRISM makes one decision per clock from its 32 inputs and drives 21 outputs, with no software in the loop. This document describes the machine on the IHP sg13g2 8x4 tile. The design record with the reasoning behind each feature is prism_interface.md the register constants used by the tests are in `test/user_peripherals/prism/regs.py`.



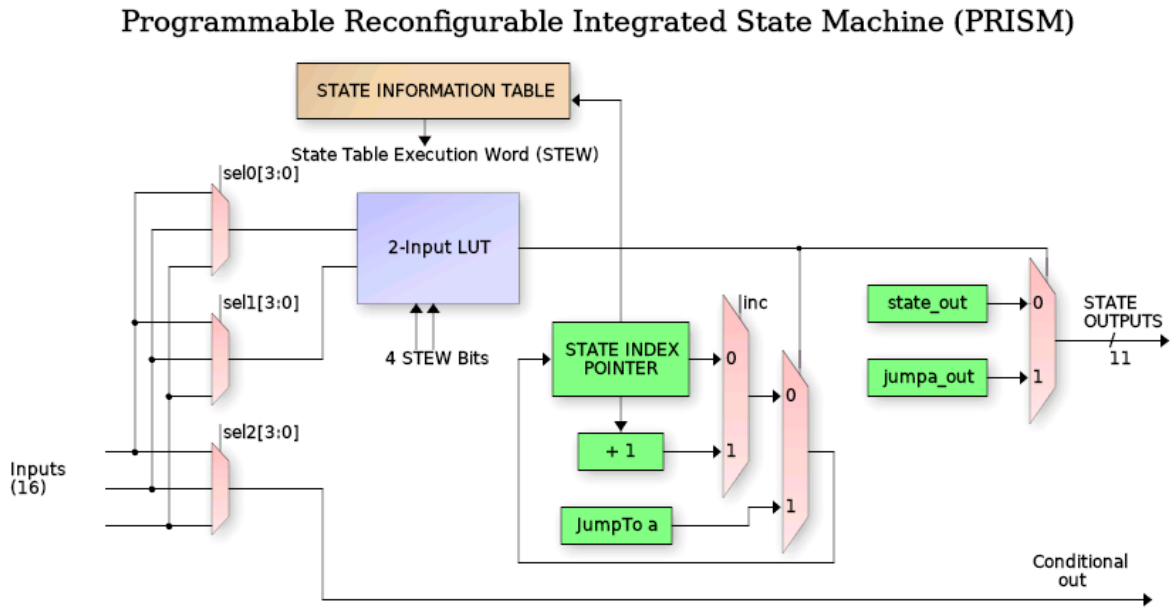
1. At a glance

States	32, in two banks of 16
State word	128 bits (STEW), from the CFGMEM latch macros
Inputs	32 per shard, 6 input muxes per state
Decision	two 3-input LUTs per state: `if` (tree 0), `else if` / `else` (tree 1)
Outputs	21 per shard, a separate set for the resting state and for each of the two branches
Conditional outputs	two per shard, each a 2-input LUT of the current inputs
Fracture	one 32-state machine, or two independent 16-state shards
Per shard	24/32-bit count1 (counter or wide shifter), 8-bit count2 with compare, 8-bit comm shifter, 4 constants, constant table, 64-byte FIFO, 2 KB SRAM FIFO, CRC / counter unit, Manchester recoverer, edge sampler, 4 edge-capture flops, timer 2, interrupt

Debug	halt, single step, two breakpoints per shard with entry / if / else-if / exit conditions, live and registered readback
Trace	1024 entries per SRAM (2048 with both), entry = state, decision inputs, outcome
Host	TinyQV, 512-byte register region at 0x8000200; CFGMEM loader at 0x8000100

Variants of the same RTL: the CMOS5L tile has 16-byte flop FIFOs and neither the counter mode nor the 32-bit FIFO access; the sg13g2 5x4 tile has no SRAMs, so no SRAM FIFOs and no tracer.

2. The core



The current **state index** (SI, 5 bits) addresses the State Information Table; the word that comes back is the STEW for that state. While the machine sits in a state the STEW's *default outputs* drive the 21 outputs. Six 5-bit fields of the STEW select six of the 32 inputs. Muxes 0-2 feed tree 0, a 3-input LUT programmed by 8 STEW bits; muxes 3-5 feed tree 1. Tree 0 is the chroma's `if`: when its LUT is true the machine jumps to the tree 0 target state and drives the *tree 0 outputs* for that one clock. Tree 1 is the `else if` (or a plain `else`, compiled as an always-true LUT): it is taken when it matches and tree 0 does not, with its own target and its own outputs. When neither fires the machine stays. Two more LUTs, 2 inputs each (cond 0 from muxes 1 and 4, cond 1 from muxes 3 and 5), drive the **conditional outputs**, whose value follows the inputs within the state; a chroma writes them as `cond_out[0] = 1'b0; if (x) cond_out[0] = 1'b1;`.

The **inc bit** starts or continues the auto-loop: in a state with `inc` set, "no match" moves to the next state instead of staying, and the first such state is remembered; a later state without `inc` and no match returns there. The compiler emits it for the idiom "a chain of tests with a trailing `else -> next state`", so a scan over several states costs no decision tree.

STEW layout (`chromas/tinyqv32.cfg`):

bits	field
0	inc
30:1	mux 0 .. mux 5 input selects, 5 bits each
35:31	tree 1 jump target
40:36	tree 0 jump target
61:41	default outputs
82:62	tree 1 outputs
103:83	tree 0 outputs
111:104	tree 0 LUT3
119:112	tree 1 LUT3
123:120	cond 0 LUT2
127:124	cond 1 LUT2

3. Shards and fracturing

The 32 states live in two banks of 16: bank A (CFGMEM "lo" macros, states 0-15) and bank B ("hi", states 16-31). **Unfractured** (FRAC_CFG = 0) shard 0 is the whole machine: its SI selects the bank with its top bit, shard 1's index register silently tracks shard 0's low bits so both banks are always addressed without a mux, and shard 1's datapath idles (its registers stay accessible, and shard 0 can use shard 1's FIFO as a second FIFO, section 8). **Fractured** (FRAC_CFG = 1) each bank is an independent 16-state machine with its own SI, its own 32 inputs, its own 21 outputs, its own datapath, its own interrupt and its own debugger. A chroma compiled with `tinyqv32_shard.cfg` (16 states) loads into either bank.

The two shards see each other in three ways:

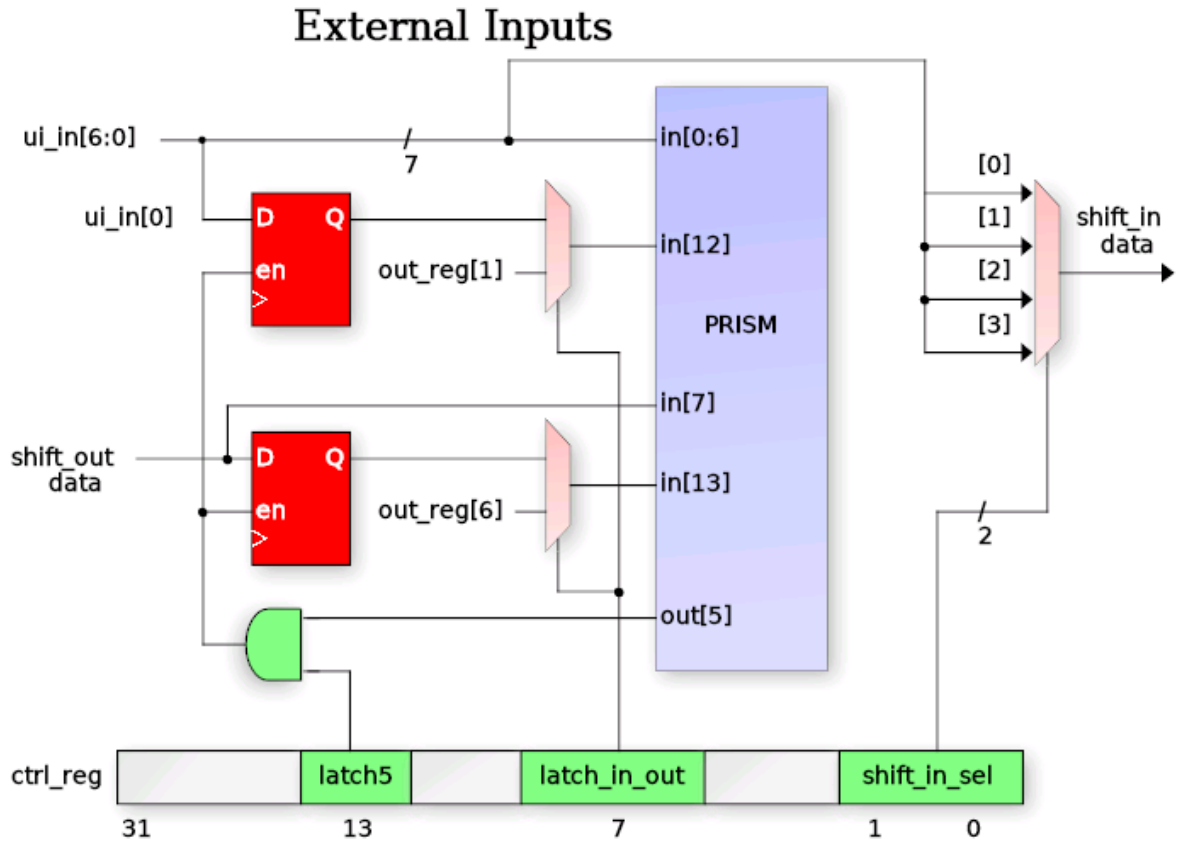
- **Semaphores**: OUT_SEMA_SET (output 19) in one shard sets a sticky flag the other shard reads as input 24 and clears with its OUT_SEMA_CLEAR (output 15). If set and clear meet in the same cycle the receiving shard's CFG0[24] decides who wins. The host reads both flags in INT_STATUS[3:2].
- **Input 25**: the other shard is halted (registered).
- **Pins**: each shard's PINMUX names a source for every uo_out[7:1] pin, or code 7 for "not mine". A pin claimed by shard 0 goes to shard 0, otherwise

to shard 1, otherwise to GPIO; a pin freezes while its owning shard is halted.

The output and conditional-output masks at 0x44-0x50 gate which of a shard's outputs reach its datapath while fractured; they reset to zero, so a fractured configuration must set them.

4. Inputs

Every shard has the same 32 inputs. Bits 0-15 keep the meaning of the original two-tile PRISM so old chromas read the same; 16 and up carry the newer features.



input	meaning
0-6	ui_in[6:0], after the shard's synchroniser select: CFG0[19:18] = 0 two flops, 1 one flop, 2 raw
7	shift_data: the serial-out bit of the selected shifter
8, 9	host_in[1:0], written by the host (a byte write to +0x15 toggles bit 0 and clears the interrupt)
10	count1_term: count1 reached 0 counting down, or the preload / roll-over counting up
11	count2_cmp: count2 >= COMPARE

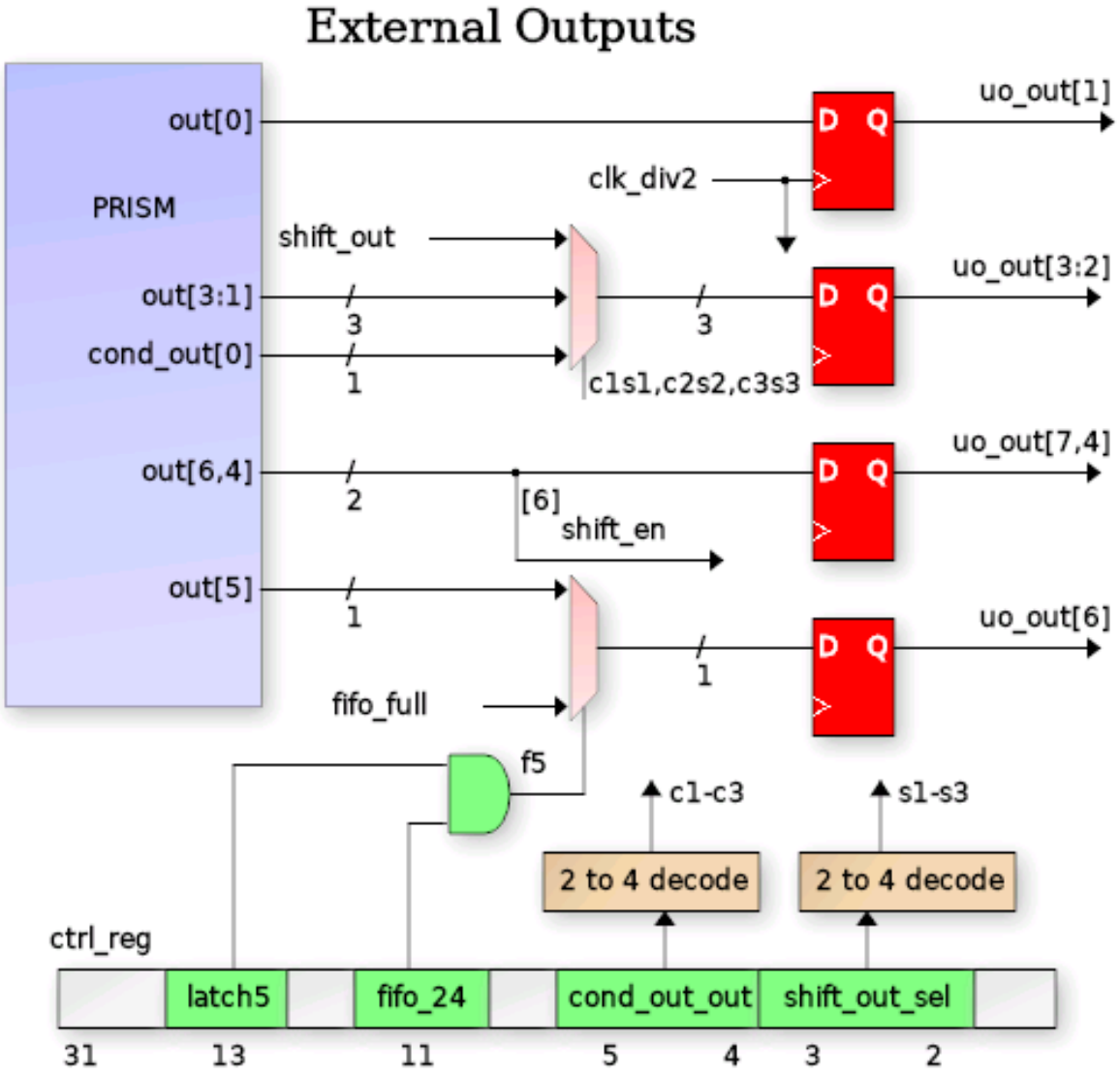
12, 13	latched_in: {shift_data, cond_out[0]} captured by OUT_LATCH, or the latched outputs with CFG0[7], or two FSM flags with CFG0[29]
14	shift_term: the shift count reached the configured length
15	count2 == comm
16-19	input slots, default the four edge-capture flops in_prev[3:0] (section 4.1)
20, 21	own FIFO flags: empty and full by default, CFG1[25:24] / [27:26] pick almost-empty / almost-full / the other side
22	crc_ok: the CRC equals CRC_EXPECTED; in counter mode, count >= CRC_EXPECTED
23	count1_wrap: count1 rolled over counting up
24	semaphore set by the other shard
25	the other shard is halted
26, 27	FIFO B's flags for shard 0 unfractured (CFG1[29:28] / [31:30]); 0 otherwise
28-31	input slots, default 0; input 28's default is the timer 2 tick (section 11)

Input slots (CFG2, four bits per slot for inputs 16-19 and 28-31) let a chroma bring other signals into a state's decision: code 0 the slot's default, 1-4 in_prev[0..3], 5-12 comm[0..7] (so the decision trees decode a received byte, a USB PID or a command, without a match register per value), 13 comm == K3, 14 flag2, 15 the Manchester bit-valid / sampler edge-pending flag.

4.1 Edge capture (in_prev)

Four flops per shard follow a source each names in CFG1[15:0] (four bits per flop: 0-6 a ui_in pin after the synchroniser, 8 or 9 a host_in bit). A flop captures its source in exactly the cycle a decision tree whose muxes read that source fires and the transition executes, so `if (pin ^ in_prev0) -> next` fires once per transition of the pin and re-arms itself. Two edges can be watched from one state through tree 0 and tree 1; a flop whose source the firing tree did not read keeps its value, so several states can share an edge condition. Nothing captures while halted, a single step captures once, and auto-loop transitions never capture. Program the sources before enabling: a flop only changes on a capture. OUT_LATCH (output 4) and CFG3[26] (the sampler) can capture all four at once.

5. Outputs



output	name	effect
0-3	pin_out[3:0]	pin values, routed to uo_out[7:1] by PINMUX
4	OUT_LATCH	capture the in_prev flops, latched_in or the FSM flags
5	OUT_FIFO_WR_RD	push comm into an RX FIFO / pop a TX FIFO into comm (direction CFG0[23])
6	OUT_COUNT1_INC_DEC	step count1, direction CFG0[14]
7	OUT_COUNT1_CLEAR_LOAD	clear count1, or load it from PRELOAD (CFG0[6])

8	OUT_SHIFT	shift the selected shifter (CFG0[10]: comm or count1)
9, 10, 11	OUT_COUNT2_INC / DEC / CLEAR	count2
12	OUT_CRC_CLEAR	preset the CRC (counter mode: preset the count)
13	OUT_CRC_UPDATE	feed the shifter's current bit into the CRC (counter mode: + 1)
14	OUT_HOST_INTERRUPT	raise the shard's interrupt, once per assertion
15	OUT_SEMA_CLEAR / OUT_FIFO_PUSH_POP	fractured: clear the semaphore; unfractured shard 0: output 5 acts on FIFO A (0) or FIFO B (1)
16	OUT_COMM_LOAD	load comm from PRELOAD[7:0], a constant K (CFG0[30]) or the constant table
17	OUT_LOAD_CRC	load the selected shifter from the CRC, one byte per load through comm (counter mode: - 1)
18, 20	OUT_K_SEL0 / 1	which constant OUT_COMM_LOAD takes, or the table's index step; in multi-bit shift mode the width - 1
19	OUT_SEMA_SET / OUT_FLAG2	fractured: set the other shard's semaphore; with CFG0[29] the value OUT_LATCH stores in flag2

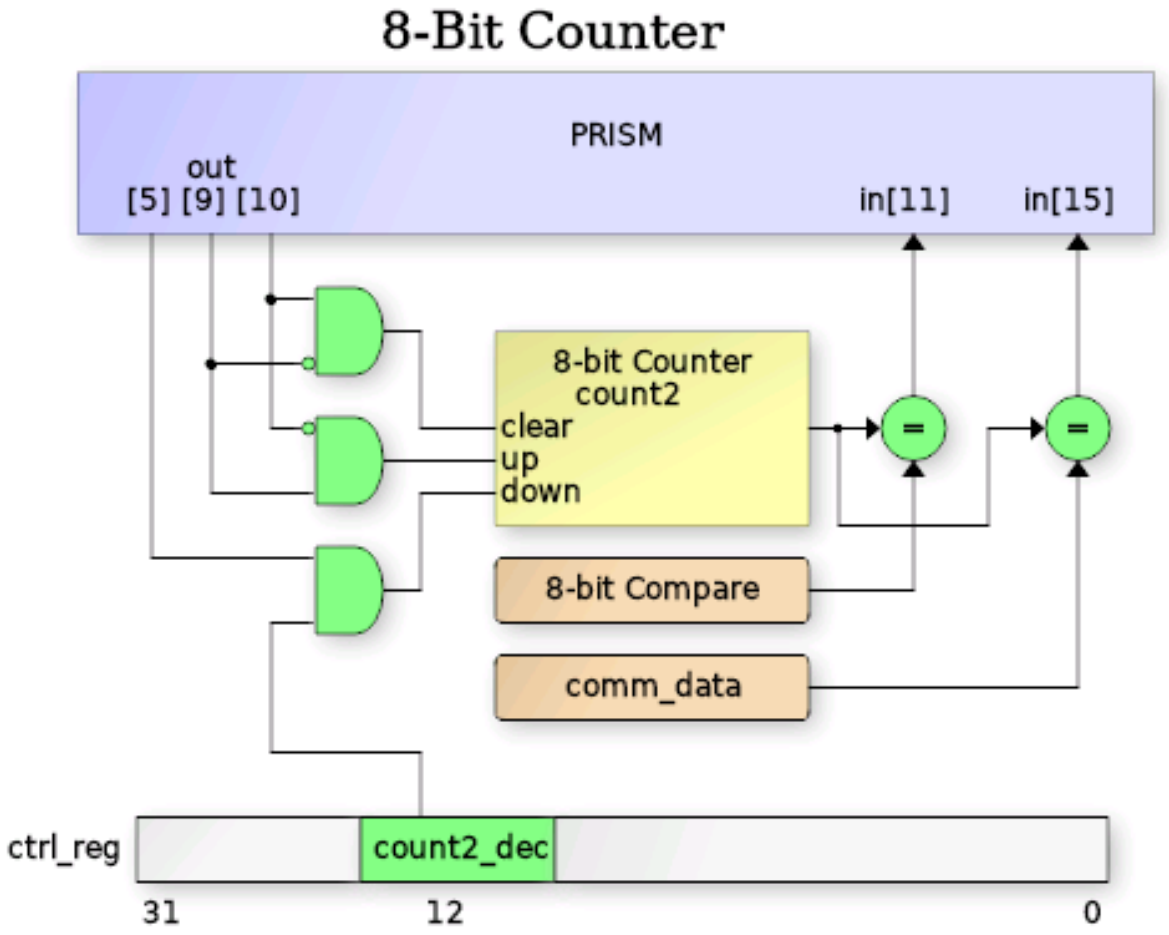
Each pin of uo_out[7:1] has a 3-bit PINMUX field: 0-3 the shard's pin_out[n], 4 and 5 its conditional outputs, 6 the shifter's serial bit (or a lane of the comm window in multi-bit mode), 7 not driven by this shard.

6. The datapath

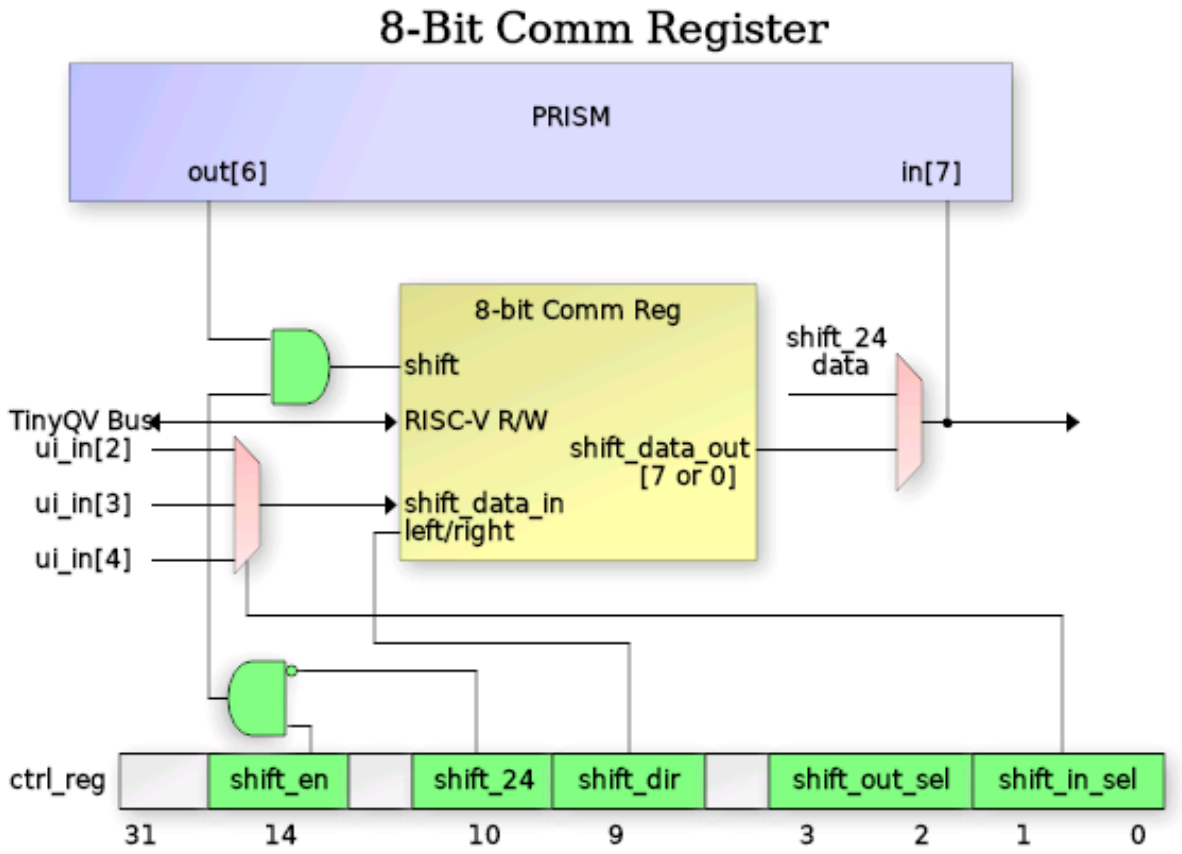
Each shard has its own copy (`prism_datapath.v`), driven by its own outputs and configured by its CFG0 (the chroma's `ctrl_reg`).

count1 is a 24-bit register, 32 bits with CFG0[11], that is either a counter or the wide shift register. As a counter it steps on output 6, down by default

or up with CFG0[14]; output 7 clears it or loads it from PRELOAD (CFG0[6]); count1_term (input 10) marks zero counting down, or the preload (CFG0[15]) or the natural roll-over (input 23, count1_wrap) counting up; the host reads the count at +0x0C and a write loads it. As the wide shifter (CFG0[10]) it shifts on output 8, MSB or LSB first (CFG0[9]), taking its input from the pin CFG0[1:0] names, from cond_out[0] (CFG0[28]: a bit the FSM decoded, NRZI or Manchester) or from the Manchester recoverer; a load can count as the first bit already shifted (CFG0[16]) so shift_term marks the last. OUT_LOAD_CRC can load all 32 bits from the CRC.



count2 is an 8-bit up / down / clear counter (outputs 9-11; the decrement is enabled by CFG0[12]) compared against the COMPARE byte (input 11, count2 >= COMPARE) and against comm (input 15, equality).



comm is the 8-bit communication shifter: it shifts on output 8 when CFG0[10] selects it, MSB or LSB first, from the same input choices as count1; it is what the FIFOs push and pop, what OUT_COMM_LOAD fills, and what OUT_LOAD_CRC loads one CRC byte at a time. A 3-bit shift count marks the byte boundary on shift_term, and a load or a FIFO pop can count as the first bit (CFG0[17]). With CFG0[2] (**multi-bit shift**) each OUT_SHIFT moves {OUT_K_SEL1, OUT_K_SEL0} + 1 bits at once from pins CFG0[1:0] .. + 3, and COMM_PINS (+0x50) names a 4-bit window of comm and a lane of it for every output pin, so a byte plays out as four bit pairs or two nibbles, PIO style; the shift count advances by the width so shift_term still marks the byte.

Constants: CONST (+0x38) holds K0..K3; with CFG0[30] OUT_COMM_LOAD loads K[{OUT_K_SEL1, OUT_K_SEL0}] (sync words, handshake codes), and K3 is also the byte comm is compared with for slot code 13. **Constant table**: when a shard runs on its SRAM FIFO the 64-byte latch FIFO's first 16 rows become an indexed table (CONST_TAB, +0x4C): OUT_COMM_LOAD loads the row at a 4-bit index, and the two K-select outputs say how the index moves on each load (clear, + 1, + add_to_idx, load idx_load), pre or post; a MAC address or a fixed reply costs no states.

Latches and flags: OUT_LATCH stores {shift_data, cond_out[0]} in latched_in (inputs 12-13); with CFG0[7] those inputs show two latched outputs instead; with CFG0[29] OUT_LATCH stores {cond_out[1], cond_out[0]} there and out-

put 19 in flag2 (slot code 14), three flags a chroma sets and tests. CFG3[28] lets flag2 swap the sampler's edge.

The host sees the datapath in COUNTS (+0x10: count2, COMPARE, comm, and the shift counts) and FLAGS (+0x18: count1_term, count1_wrap, count2_cmp, count2 == comm, shift_term, shift_data, latched_in, FIFO empty and full, crc_ok, sampler edge pending).

7. Timers

count1 in counter mode is the general timer: load it from PRELOAD, count down, watch input 10.

Timer 2 (PRELOAD2, +0x40) is a free-running 24-bit timer with no FSM output: it reloads itself and raises input 28 for one clock every period + 1 clocks (0 = off), pacing things while count1 is busy (Ethernet link pulses every 16 ms need no CPU). With PRELOAD2[24] the count restarts whenever the shard enters state PRELOAD2[29:25], which makes the tick a retriggerable timeout ("nothing for N clocks since the last entry into RX_BIT") with no STEW bits; with [30] as well it ticks once per timeout and waits for the next entry.

8. FIFOs

Each shard has a **64-byte FIFO** built from latch rows. CFG0[23] sets its direction: RX (0), the FSM pushes comm with output 5 and the host pops by reading +0x20; TX (1), the host pushes by writing +0x20 and output 5 pops the head into comm. A push on full and a pop on empty are ignored. FIFO_STATUS (+0x24) reads the count and the flags, and any write flushes. The almost-empty and almost-full levels are CFG1[19:16] / [23:20] in 4-byte units (almost-empty when count <= 4 x level, almost-full when count >= 64 - 4 x level); inputs 20 and 21 show two of the four flags, chosen by CFG1[27:24]. Contents survive PRISM disable, so a TX FIFO can be filled before the machine starts.

Two FIFOs for one machine: unfractured, shard 0 owns both FIFOs, A its own and B shard 1's (configured and served by the host through the shard 1 window). Output 15 says which one output 5 strobes, each per its own direction, so with A = RX and B = TX output 15 reads as "push / pop", and inputs 26-27 show B's flags. Single-FIFO chromas (output 15 = 0) behave as before.

SRAM FIFOs: with CFG0[31] a shard's FIFO is a 2 KB FIFO on its own IHP 512x32 SRAM macro (`prism_sram_fifo.v`), four bytes per word with a cached head word, the same push / pop / flags interface, the count field widened, and the levels in 64-byte units. An Ethernet receiver in one shard and a transmitter in the other each get a full-frame buffer; unfractured, shard 0 reaches shard 1's SRAM as FIFO B.

32-bit access (CFG3[11], FIFO32 at +0x54): the host moves words. TX: a word written there is pushed a byte at a time, low byte first, one every fourth clock, waiting on a full FIFO; FIFO_STATUS[5] is busy meanwhile and a word written while busy is dropped, so poll it when the FIFO may be full (straight-line code never sees it: four pushes take 13 clocks, TinyQV's instructions 16). RX: a pop machine takes four bytes as soon as the FIFO holds them, low byte first, into a word register; FIFO_STATUS[4] and the shard's interrupt say the word is complete, and a read of FIFO32 takes it. A byte read of +0x20 while the word register holds anything is served from its low byte, the rest shift down and the machine refills the top, so byte order is the same however word and byte reads are mixed; a short message's stragglers show in FIFO_STATUS (the count, or [7:6] after byte reads) and are read as bytes.

9. CRC unit and counter mode

A bit-serial CRC per shard (`prism_crc.v`): width from CFG0[21:20] (8, 16 or 32 bits; 0 = off), a programmable polynomial (CRC_POLY, +0x28), reflected (LSB first, CFG0[22]) or not, preset to zero or all ones (CFG0[25]) by output 12, fed one bit per output 13 with the bit the shifter is receiving or transmitting that clock (CFG0[27]), complemented on output (CFG0[26]). The host reads or presets the value at +0x2C; input 22 (crc_ok, also FLAGS[10]) says the value equals CRC_EXPECTED (+0x30), the residue after data plus checksum, so a receiver checks a frame with one input. To transmit a checksum, output 17 loads the selected shifter from the CRC: the wide shifter takes all 32 bits, comm takes one byte per load in wire order and the register advances, so CRC16 / CRC32 go out in 2 / 4 loads. CRC-8 0x07, CRC-16/CCITT 0x1021, CRC-16/USB 0xA001 (reflected), USB CRC5 0x14, CRC-32 0xEDB88320 (reflected, Ethernet) have all run.

Counter mode (CFG3[10], with the CRC off): the same 32-bit register is an up / down counter on the same strobes, output 12 presets it, 13 counts up, 17 counts down (both in one cycle: hold; no shifter load), and input 22 becomes the unsigned compare count >= CRC_EXPECTED. The host reads or presets the count at +0x2C and sets the compare at +0x30. One adder and one comparator per shard; no extra flops.

10. Receiving clocked and self-clocked lines

Manchester bit recoverer (`prism_mrx.v`, CFG3[9:0]): names a pin ([2:0]), the clocks per half bit ([7:4]) and an enable ([3]); an edge on the line is accepted when three quarters of a bit have passed since the last accepted edge, so the boundary transition between equal bits is skipped and every mid-bit edge re-times the decoder; the level after the accepted edge is the bit. With [8] the shifter takes that bit as its input, and "bit valid" is slot code 15, sticky until the FSM's shift consumes it, so a two-state receive loop never misses a bit. With

[9] the pin is sampled on both clock edges and [7:4] counts half clocks per half bit (6 at 64 MHz, 5 at 50 MHz), doubling the edge resolution; 10BASE-T receive keeps its margin down to a 50 MHz tile clock.

Edge-clocked sampler (CFG3[28:16]): names any PRISM input ([21:17]) and an edge ([23:22]: rising, falling, either), and on every such edge the datapath acts with no state transition: shift ([24]), count2 + 1 ([25]), capture the in_prev flops ([26]), count1 clear / load ([27]). A clocked slave protocol no longer spends states on its bit loop; the FSM acts at byte boundaries on shift_term or count2. With [28] flag2 swaps rising and falling, so one FSM flag turns a bidirectional protocol's sampling edge (an I2C target receives on SCL rising and drives after SCL falling). A sticky "edge pending" flag (slot code 15, FLAGS[11]) is cleared by the FSM's OUT_SHIFT or OUT_LATCH.

11. Interrupts

Each shard has one interrupt line to the host (TinyQV user interrupts 8 and 9, CTRL[31] and [29], INT_STATUS[1:0]). It is set by output 14 once per assertion (a state that holds it does not re-arm it), by a debugger halt, and in 32-bit RX mode while a complete word waits in FIFO32. It is cleared by a byte write with bit 7 to 0x003 (shard 0) or 0x007 (shard 1), by the host_in toggle write (+0x15), or by disabling PRISM; the word-ready source clears when the word is read.

12. Debugger

Each shard has a debug control word (0x04 shard 0, 0x08 shard 1):

bits	field	meaning
0	halt_req	halt (on the rising edge) and hold
1	step	rising edge: execute one transition
2, 3	bp_en0, bp_en1	breakpoint enables
8:4, 13:9	bp_si0, bp_si1	breakpoint state
15:14, 17:16	bp_cond0, bp_cond1	0 on entry, 1 when tree 0 matches in the state, 2 when tree 1 is taken, 3 when either is, that is on exit
18	new_si	write-only: load the state index from bits 23:19

A conditional breakpoint fires in the cycle the selected tree first matches while the machine sits in the state, before the transition's outputs happen, so the value that satisfied the condition is still there to read; a single step then performs the transition with its outputs and halts in the target. Keep halt_req set while stepping. STATUS (0x0C) shows, per shard, the current and next state index, the halt and which breakpoint hit; 0x10-0x1C read back the four

words of shard 0's current STEW; 0x34 the decision inputs and LUT results; 0x38 the outputs; 0x3C the input vector. While a shard is halted its datapath, pins, semaphores and interrupt edge are frozen with it.

13. Trace

Each shard has a tracer (TRACE_CFG +0x44, TRACE_CTRL +0x48). From its trigger on, every clock's `{executing, tree 1 taken, tree 0 matched, the six selected input bits, SI}` is written as a 16-bit entry, two per SRAM word, into an SRAM until the buffer is full: 1024 entries on one SRAM, 2048 with both as one buffer (TRACE_CFG[1]), and with TRACE_CFG[6] into the other shard's SRAM so a chroma streaming from its own SRAM FIFO can be traced while it does so. Triggers: at once; in a given state; that state taking either jump; or an edge (rising, falling, either) on any of the 32 inputs. Entry 0 is the trigger cycle. Arm with TRACE_CTRL = 1 (the traced SRAM's FIFO is flushed), wait for done (or stop with 2), then read the entries as FIFO bytes through the window that reads that SRAM: while an SRAM holds a finished trace the window sees it as its SRAM FIFO in RX mode, count = entries x 2, each read of +0x20 the next byte, low byte first. The 21 outputs of a traced clock are not stored: they follow from the entry and the loaded chroma (the STEW's three output fields), and the SDK rebuilds them. One shard traces at a time; shard 0 wins if both enable.

14. CFGMEM: the state table and how it is loaded

The table is eight latch-array macros, 16 rows x 32 bits each, generated by a DFFRAM-derived flow (`CFGMEM_IHP16` and its left-hand mirror `CFGMEM_IHP_LEFT16`; the views are in `macros/`), placed as two columns beside the standard cells. Four macros make bank A (states 0-15: row s of macros lo0..lo3 is state s's STEW, lo0 the low word) and four bank B (hi0..hi3, states 16-31). Every clock the machine reads a whole 128-bit row from each bank through the macros' read ports; latch arrays rather than flops because a row is read every clock and written only when a chroma is loaded, and a latch bit is far smaller than a flop with its write mux.

The CPU loads them through the CFGMEM loader peripheral at 0x8000100. Its control byte at 0x1F is `[3:0]` row address, `[4]` address select (the host addresses the row instead of the machine), `[5]` busy (read-only: the loader's shift takes 49 clocks, more than a back-to-back store), `[6]` bypass lo, `[7]` bypass hi. The four macros of a bank form a shift chain, host word -> macro 0 -> 1 -> 2 -> 3, and each macro's output is its addressed row, or its input while the bank's bypass bit is set. Loading is therefore: set the bank's bypass bit, then for each state from the highest down, write its four words, most significant first, to the macro ports (`0x00 + 4i` for lo i, `0x20 + 4i` for hi i, word 0 to macro 3 down to word 3 to macro 0), waiting on busy between writes; each write shifts

the chain one row, so state 0 ends in row 0. Clear the control byte afterwards to hand the row address back to the machine. Reads with the bypass clear and address select set return each macro's row at the addressed index, which is how a loaded table is verified. The compiler's `.py` / `.c` output is the word list in this order, so `prism_load_chroma()` is a loop.

15. Register map

All registers are word aligned at +4 offsets; a byte read returns the addressed byte lane in the low bits, so packed registers can be read and written a byte at a time. Offsets are from 0x8000200.

15.1 Common block

offset	register
0x00	CTRL: [30] enable, [31] shard 0 interrupt (RO), [29] shard 1 interrupt (RO); byte 0x03 write with bit 7 clears shard 0's interrupt, byte 0x07 shard 1's
0x04, 0x08	DBG_CTRL shard 0 / shard 1 (section 12); read back = {state index[23:19], control[17:0]}
0x0C	STATUS: shard 0 in [12:0], shard 1 in [25:13], each {break[1:0], halt, next SI[4:0], current SI[4:0]}
0x10-0x1C	the four words of shard 0's current STEW
0x20	ID
0x24	INT_STATUS: [1:0] shard interrupts, [3:2] semaphores as seen by shard 0 / shard 1
0x30	DEBUG_DOUT: the outputs a halted shard presents
0x34	DECISION: LUT inputs and results
0x38	OUT_DATA: the live output vector
0x3C	IN_DATA: shard 0's input vector
0x40	FRAC_CFG: [0] fractured
0x44, 0x48	shard 0 output mask, conditional-output mask (fractured)
0x4C, 0x50	shard 1 output mask, conditional-output mask

15.2 Shard window (shard 0 at 0x100, shard 1 at 0x180)

offset	register
+0x00	CFG0: datapath configuration, the chroma's `ctrl_reg` (section 15.3)
+0x04	PINMUX: uo_out[7:1] sources, 3 bits per pin, the chroma's `pinmux_reg`

+0x08	PRELOAD (32 bits)
+0x0C	COUNT1: read the count, write loads it
+0x10	COUNTS: byte 0 count2, byte 1 COMPARE, byte 2 comm, byte 3 {comm_count[2:0], shift_count[4:0]} (RO); bytes 0-2 writable
+0x14	HOST: host_in[1:0]; a byte write to +0x15 toggles host_in[0] and clears the interrupt
+0x18	FLAGS (RO): [0] count1_term, [1] count1_wrap, [2] count2_cmp, [3] count2 == comm, [4] shift_term, [5] shift_data, [7:6] latched_in, [8] FIFO empty, [9] FIFO full, [10] crc_ok / count >= compare, [11] sampler edge pending
+0x1C	CFG1: [15:0] in_prev sources, [19:16] / [23:20] FIFO almost-empty / almost-full levels, [25:24] / [27:26] input 20 / 21 flag selects, [29:28] / [31:30] inputs 26 / 27 (FIFO B)
+0x20	FIFO: byte write pushes (TX), byte read pops (RX)
+0x24	FIFO_STATUS: {count[21:8], word bytes[7:6], push busy[5], word full[4], almost_full[3], almost_empty[2], full[1], empty[0]}; any write flushes
+0x28	CRC_POLY
+0x2C	CRC: value; write presets (counter mode: the count)
+0x30	CRC_EXPECTED (counter mode: the compare value)
+0x34	CFG2: input slot selects, 4 bits each, inputs 16-19 in [15:0] and 28-31 in [31:16]
+0x38	CONST: K0 [7:0] .. K3 [31:24]
+0x3C	CFG3: Manchester recoverer [9:0], counter mode [10], 32-bit FIFO access [11], edge-clocked sampler [28:16]
+0x40	PRELOAD2: timer 2 [23:0] period, [24] restart on entry into state [29:25], [30] one-shot
+0x44	TRACE_CFG (write-only): [0] enable, [1] both SRAMs, [6] other shard's SRAM, [3:2] trigger (0 now, 1 in state [12:8], 2 that state jumping, 3 an edge on input [20:16], [5:4] 0 rising 1 falling 2 either)
+0x48	TRACE_CTRL: write [0] arm, [1] stop; read [0] armed, [1] running, [2] done, [3] big, [4] active
+0x4C	CONST_TAB: [0] enable, [1] add idx_load, [2] post, [7:4] idx_load, [10:8] add_to_idx, [19:16] index
+0x50	COMM_PINS: [2:0] window base, [2k+5:2k+4] lane for uo_out[k+1]
+0x54	FIFO32: 32-bit FIFO access (section 8)

15.3 CFG0, the chroma's control word

bits	name	meaning
1:0	shift_in_sel	the ui_in pin that feeds the shifter (multi-bit mode: pins sel .. sel + 3)
2	mshift_en	multi-bit comm shift, {out20, out18} + 1 bits per shift
6	clr_not_load	output 7 clears count1 instead of loading it
7	latch_in_out	inputs 13:12 show latched outputs instead of latched inputs
8	shift_en	shifting enabled
9	shift_dir	0 MSB first, 1 LSB first
10	shift_wide	output 8 shifts count1 (1) or comm (0)
11	count32	count1 is 32 bits (0: 24)
12	count2_dec_en	output 10 decrements count2
13	latch_en	OUT_LATCH enabled
14	count_up	count1 counts up
15	wrap_preload	counting up rolls over at PRELOAD instead of naturally
16	shift_load_one	a wide-shifter load counts as the first shift
17	comm_load_one	a comm load or pop counts as the first shift
19:18	in_sync_sel	pin synchroniser: 0 two flops, 1 one flop, 2 raw
21:20	crc_mode	0 off, 1 CRC-8, 2 CRC-16, 3 CRC-32
22	crc_reflect	LSB first into the CRC
23	fifo_dir	0 RX (FSM pushes, host pops), 1 TX (host pushes, FSM pops)
24	sema_set_wins	semaphore set beats clear in the same cycle
25	crc_init_ones	preset to all ones
26	crc_xor_out	complement on OUT_LOAD_CRC
27	crc_src	0 the shifter's input bit, 1 its output bit
28	shift_in_cond	the shifter's input is cond_out[0]
29	flag_latch	OUT_LATCH stores {cond_out[1], cond_out[0]} in latched_in and output 19 in flag2
30	comm_load_k	OUT_COMM_LOAD takes constant K[{out20, out18}]
31	fifo_sram	this shard's FIFO is its SRAM FIFO

16. Chromas

A chroma is a Verilog module with the fixed ports `clk, rst_n, fsm_enable, in_data[31:0], out_data[20:0], cond_out[1:0], ctrl_reg[31:0], pinmux_reg[20:0]`: a state register, a `case` on it, `if / else if / else` transitions on the inputs, and constant assignments to the outputs in each state. The compiler is a Yosys backend (yosys-prism, `synth_prism -cfg chromas/tinyqv32.cfg`): it maps the FSM onto the STEW fields, checks that each state's conditions fit the six muxes and two trees, emits the `inc` loop idiom, and writes the table as a Python list, a C array and a columnar listing, together with the `ctrl_reg` and `pinmux_reg` values the chroma declared. Rules it enforces: one wire per output bit, constants in the always block (an input-dependent conditional output is written as a default plus an `if`), and a trailing `else -> next state` behind at least one condition always compiles to the auto-loop.

```
module chroma_uart_tx ( input clk, rst_n, fsm_enable, input [31:0]
in_data,
                        output [20:0] out_data, output reg [1:0]
cond_out,
                        output reg [31:0] ctrl_reg, output reg
[20:0] pinmux_reg );
   ...
   STATE_IDLE:
      if (!fifo_empty)              // input 20
      begin
         fifo_pop = 1'b1;           // output 5: comm <= FIFO head
         count1_load = 1'b1;        // output 7: bit timer from
PRELOAD
         next_state = STATE_START;
      end
   ...
```

The library in `chromas/`, every one with a cocotb test against a model of the far end:

chroma	what it does
gpio24	24-bit GPIO expander: 74165 in, 74595 out
spislave	SPI slave with a CRC-8 trailer
spi_master	SPI master, single or quad lane
uart_tx	8N1 transmitter from the TX FIFO, CRC-8 on request
i2c_master	I2C controller through external open-drain buffers

i2c_slave	I2C target on the edge-clocked sampler
onewire	1-Wire master
ws2812	WS2812 LED transmitter
encoder	quadrature encoder decode
pio	4-channel logic analyser and 2-lane waveform generator (multi-bit shift)
usb_ls	USB low-speed device: NRZI, bit unstuffing, CRC-5 / CRC-16, DATA0 / DATA1
eth_tx	10BASE-T transmitter from the SRAM FIFO, CRC-32 FCS, link pulses on timer 2
eth_rx	10BASE-T receiver on the Manchester recoverer, FCS checked by crc_ok
fifo_loop, edge, const_tab, counter	unit tests of the two-FIFO mode, the edge capture, the constant table and the counter mode

The tests (`test/user_peripherals/prism/`) load each chroma through the CFGMEM loader exactly as software does, and run unchanged on the gate-level netlist. Host software uses `prism.h` from the tinyQV-sdk with `PRISM_CONFIG` janestreet: chroma loading, the datapath registers, FIFO and CRC access, the sampler, the recoverer, timer 2, the constant table, the debugger and the tracer, including the reconstruction of a trace's outputs.

The design record behind PRISM, decision by decision, is in prism_interface.md.

Pins

Pin	Use
ui_in[6:0]	PRISM inputs 0-6 (each shard chooses raw, one-flop or two-flop synchronised)
ui_in[7]	TinyQV UART RX (or ui_in[3], by register)
uo_out[7:1]	PRISM outputs: each pin is routed by the running chroma to one of its four pin outputs, a conditional output or the shifter bit, or left to GPIO
uo_out[0]	TinyQV UART TX
uo_out[6]	doubles as the debug UART when selected

uio[7:0]	QSPI flash and PSRAM (bidirectional PMOD)

TinyQV, the host processor

TinyQV is a small RISC-V CPU designed for Tiny Tapeout. It implements the RV32EC instruction set plus the Zcb and Zicond extensions, with a couple of caveats: addresses are 28 bits, program addresses 24 bits, and `gp` / `tp` are hardcoded to 0x1000400 / 0x8000000. Instructions are read over QSPI from flash and a QSPI PSRAM holds data; the two share the QSPI clock and data lines, so only one is accessed at a time. Code runs from flash only. In this design its job is to be PRISM's host: load a chroma, program the datapath, feed and drain the FIFOs, service the interrupts.

TinyQV Memory map (host view)

Address range	Device
0x0000000 - 0x0FFFFFF	Flash
0x1000000 - 0x17FFFFF	RAM A
0x1800000 - 0x1FFFFFF	RAM B
0x8000000 - 0x800003F	TinyQV debug and time
0x8000040 - 0x800007F	GPIO
0x8000080 - 0x80000FF	UART
0x8000100 - 0x800013F	PRISM CFGMEM programming (state table loader)
0x8000200 - 0x80003FF	PRISM: common block, shard 0 window at +0x100, shard 1 window at +0x180
0xFFFFF00 - 0xFFFFF07	TIME

PRISM's two shard interrupts are TinyQV user interrupts 8 and 9.

DEBUG

Register	Address	Description
SEL	0x800000C (R/W)	Bits 6-7 enable peripheral output on out6-7, otherwise out6-7 carry debug signals
DEBUG_UART_DATA	0x8000018 (W)	Transmits the byte on the debug UART
STATUS	0x800001C (R)	Bit 0: debug UART TX busy

See also debug docs.

TIME

Register	Address	Description
MTIME	0xFFFFF00 (RW)	Get/set the 1 MHz time count
MTIMECMP	0xFFFFF04 (RW)	Get/set the time to trigger the timer interrupt

A 32-bit timer in the spirit of the RISC-V timer: MTIME advances at 1/64 of the clock (nominally 1 MHz) and the interrupt asserts while MTIME is past MTIMECMP by less than 2^30 microseconds.

GPIO

Register	Address	Description
OUT	0x8000040 (RW)	out0-7 when the GPIO function is selected
IN	0x8000044 (R)	The current state of in0-7
FUNC_SEL	0x8000060 - 0x800007F (RW)	Function select for out0-7

Function Select	Peripheral
0	Disabled
1	GPIO
2	UART
8	PRISM (a pin PRISM does not claim falls back to GPIO)

UART

Register	Address	Description
TX_DATA	0x8000080 (W)	Transmits the byte
RX_DATA	0x8000080 (R)	Reads any received byte
TX_BUSY	0x8000084 (R)	Bit 0: TX busy; bit 1: a received byte is available
DIVIDER	0x8000088 (R/W)	13-bit clock divider for the baud rate
RX_SELECT	0x800008C (R/W)	UART RX pin: `ui_in[7]` when 0 (default), `ui_in[3]` when 1

How to test

Bring-up is TinyQV's: load a program image into flash, reset (rst_n high then low so the design sees a falling edge; program the flash and leave it in continuous read mode and the PSRAMs in QPI mode; drive the QSPI chip selects high with SD1:SD0 set to the read latency in cycles, 2 at 64 MHz; clock at least

8 times and stop with the clock high; release the QSPI lines; raise rst_n; then clock normally). The RP2040 on the Tiny Tapeout demo board does this from MicroPython. Build programs with the customised toolchain and the tinyQV-sdk, whose `prism.h` knows this tile's register map (`PRISM_CONFIG` janestreet).

A protocol is then a chroma, a Verilog Mealy state machine compiled by the yosys-prism backend into the 32 x 128-bit state table plus the datapath configuration words. The sequence, in software or from the test bench:

1. Write the chroma from `chromas/`, or take one of the seventeen there: `make` in that directory compiles them all (`make yosys` fetches and builds the compiler first).
2. Load the table through the CFGMEM loader (`prism_load_chroma()` in the SDK: one word write per row with the bank's bypass bit set), program the shard's CFG0 / PINMUX from the compiler's output, set the datapath registers the chroma needs (preload, compare, constants, FIFO direction and levels), and set the enable bit. The machine starts in state 0.
3. Talk to it through the shard window: host_in bits for handshakes, the FIFO for data, the interrupt for completion, the debugger to halt, step and break, the tracer to record what it did.

The test suite is the reference for all of this: `make -C test` runs the cocotb tests (`test/user_peripherals/prism/`), each a chroma driven by a model of the device on the other side of the wire: the GPIO expander, SPI slave and master, UART, I2C master and slave, 1-Wire, WS2812, the quadrature encoder, the PIO logic analyser and generator, a USB low-speed host talking to the device chroma; plus the fractured mode, the FIFOs and their 32-bit access, the second timer, the sampler, the counter mode and the debugger. The Ethernet, SRAM FIFO, constant table and tracer tests skip on this tile. The same suite runs on the gate-level netlist.

External hardware

The design expects the QSPI PMOD on the bidirectional PMOD (16 MB flash, two 8 MB RAMs); the UART is on the pins of the demo board's RP2040 hardware UART.

Beyond that, whatever the protocol needs: PRISM drives logic levels on uo_out and reads them on ui_in. A USB low-speed device wants the usual series resistors and a 1.5 kOhm pull-up on D-, with D+ / D- on two outputs and an output-enable pin driving an external tri-state buffer. 10BASE-T (on the 8x4 tile) wants magnetics with a differential driver on the transmit pair and a receiver on the receive pair. I2C wants external open-drain buffers, since the tile's outputs are push-pull. Buttons on the inputs are handy for the host-in handshakes while developing.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Interrupt 0	UART TX	Flash CS
1	Interrupt 1	—	SD0
2	—	—	SD1
3	—	—	SCK
4	—	—	SD2
5	—	—	SD3
6	—	Debug UART TX	RAM A CS
7	UART RX	Debug signal / PWM	RAM B CS / PWM

4-Input Signed Neuron / Perceptron

by **Mohammad Musharaf Qureshi**

`0427` `50 MHz` `HDL Project`

`github.com/MaverickSemiUSA/tt_um_MUSHARAF-NEURON-ACCELERATOR`

"Compact 4-input signed perceptron accelerator with ReLU activation and 8-bit output saturation."

How it works

The **4-Input Signed Neuron / Perceptron** is a compact hardware accelerator that calculates a weighted sum of four signed inputs and applies a ReLU activation function with 8-bit output saturation.

The neuron performs:

```
y = w0×in0 + w1×in1 + w2×in2 + w3×in3 + bias
```

The four weights, four inputs, and bias are stored internally as 6-bit two's-complement values. The signed 6-bit range is:

```
−32 to +31
```

A single shared 6×6 signed multiplier is reused for all four weight-input multiplications. The four products are calculated sequentially and accumulated in a 16-bit accumulator. After the four multiplications, the stored bias is added.

The result then passes through a ReLU activation and an 8-bit saturation stage:

```
if result < 0
    output = 0

else if result > 255
    output = 255

else
    output = result
```

Therefore, the final output is always between `0` and `255`.

Register Map

The registers are written through `ui_in[5:0]`. The register selected by `uio_in[3:0]` is updated when `uio_in[4]` (`WR_EN`) is asserted.

Address	Register
0	w0

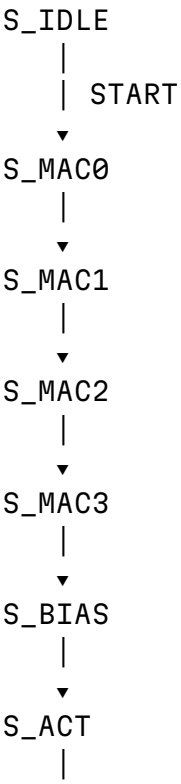
1	w1
2	w2
3	w3
4	bias
5	in0
6	in1
7	in2
8	in3

All stored values use 6-bit two's-complement representation.

Input and Control Interface

Signal	Function
ui_in[5:0]	6-bit register data
ui_in[7:6]	Unused
uio_in[3:0]	Register address
uio_in[4]	WR_EN — write selected register
uio_in[5]	START — start computation
uio_in[7:6]	Unused

When START is asserted, the accelerator enters its sequential computation:



▼
S_DONE

The shared multiplier performs:

```
S_MAC0 → w0 × in0
S_MAC1 → w1 × in1
S_MAC2 → w2 × in2
S_MAC3 → w3 × in3
```

The S_BIAS state adds the bias. The S_ACT state applies ReLU and saturation, and S_DONE indicates that the result is ready.

Output Interface

The final neuron result is available on uo_out[7:0].

Status signals are provided through uio_out:

Signal	Function
uio_out[0]	BUSY — computation is in progress
uio_out[1]	DONE — computation is complete
uio_out[7:2]	0

The corresponding output-enable signals are:

```
uio_oe[1:0] = 1
uio_oe[7:2] = 0
```

How to test

First apply an active-low reset using rst_n and keep ena high.

1. Write the weights, inputs, and bias

For each register:

1. Place the 6-bit two's-complement value on ui_in[5:0].
2. Place the register address on uio_in[3:0].
3. Assert uio_in[4] (WR_EN).
4. Apply a clock edge.
5. Deassert WR_EN.

For example, to write w0 = 3:

```
ui_in[5:0] = 000011
uio_in[3:0] = 0000
uio_in[4] = 1
```

Negative values are represented using 6-bit two's complement. For example:

```
-5 = 6'b111011
```

2. Start the neuron

After all weights, inputs, and bias values have been written, assert:

`uio_in[5] = 1`

This starts the calculation.

During computation:

`uio_out[0] = 1`

indicates that the accelerator is busy.

3. Read the result

When computation completes:

`uio_out[1] = 1`

indicates `DONE`.

The final ReLU and saturated result can then be read from:

`uo_out[7:0]`

After `START` is deasserted, the accelerator returns to the idle state.

Example 1 — Basic Positive MAC

```
Weights = [1, 2, 3, 4]
Inputs  = [1, 1, 1, 1]
Bias    = 0
```

Calculation:

```
1×1 + 2×1 + 3×1 + 4×1 + 0
= 10
```

Expected output:

`10`

Example 2 — ReLU

```
Weights = [−5, −5, −5, −5]
Inputs  = [1, 1, 1, 1]
Bias    = 0
```

Calculation:

```
−5×1 + −5×1 + −5×1 + −5×1
= −20
```

ReLU clamps the negative result to:

`0`

Expected output:

`0`

Example 3 — Output Saturation

```
Weights = [31, 31, 31, 31]
Inputs  = [4, 4, 4, 4]
Bias    = 31
```

Calculation:

```
31×4 + 31×4 + 31×4 + 31×4 + 31
= 527
```

Since the output is saturated to 8 bits:

```
527 → 255
```

Expected output:

```
255
```

Example 4 — Mixed Signed MAC

```
Weights = [3, −2, 4, −1]
Inputs  = [10, 5, 2, 3]
Bias    = −5
```

Calculation:

```
3×10 + (−2×5) + 4×2 + (−1×3) − 5
= 30 − 10 + 8 − 3 − 5
= 20
```

Expected output:

```
20
```

Example 5 — Minimum Signed Value

The minimum signed 6-bit value is:

```
−32
```

For:

```
Weights = [−32, 0, 0, 0]
Inputs  = [1, 0, 0, 0]
Bias    = 0
```

the result is:

```
−32
```

After ReLU:

```
0
```

Expected output:

```
0
```

External hardware

No external hardware is required for simulation.

For a physical demonstration, an FPGA, microcontroller, or other digital controller can be used to provide the register address, 6-bit data, write-enable, and start signals.

The `uo_out[7:0]` output can be connected to a logic analyzer or other digital interface to observe the neuron result.

The `uio_out[0]` and `uio_out[1]` signals provide `BUSY` and `DONE` status and can be monitored by the external controller.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA[0]	RESULT[0]	BUSY
1	DATA[1]	RESULT[1]	DONE
2	DATA[2]	RESULT[2]	—
3	DATA[3]	RESULT[3]	ADDR[0]
4	DATA[4]	RESULT[4]	ADDR[1]
5	DATA[5]	RESULT[5]	ADDR[2]
6	—	RESULT[6]	ADDR[3]
7	—	RESULT[7]	—

Oscillator Ising Machine

by **Ryan Hasenfratz / Hasi Industries**

449 | 50 MHz | HDL Project

github.com/Ryan-hasi/tt-ising-machine

"*Eight coupled ring oscillators that solve MAX-CUT by phase relaxation*"

How it works

Eight ring oscillators solve combinatorial optimisation problems by falling into a low-energy state, rather than by searching through candidate solutions. Each oscillator is one Ising spin; its phase relative to a reference is the spin value. Two coupled oscillators pull each other either into phase or into anti-phase, and which one they prefer is the sign of the coupling. The coupling matrix *is* the problem, and the phases the oscillators settle into *are* the answer.

This is the same principle behind quantum annealers, implemented classically at room temperature in ordinary standard cells.

Coupling

Coupling acts on the **round-trip delay** of a ring, not on its signal value. A phase detector compares the ring's own quadrature tap against the phase of the currently selected neighbour, and the result switches two extra inverters into or out of the ring. The ring stays closed and oscillating throughout; it just runs a few percent faster or slower, which pulls its phase.

Three details matter and are easy to get wrong:

- The detector compares the ring's **quadrature** tap (90°) against the neighbour's phase, not phase against phase. An XOR phase detector settles where its mean output is one half, which is 90° of separation. Taking the quadrature tap moves that equilibrium to 0°, so the coupling actually pulls into phase rather than into quadrature.
- Phase binarisation (SHIL) needs the same trick one octave up. The chip compares against a doubled-frequency signal taken from taps offset by `STAGES/8`. Without that offset the loop settles at 45° of the fundamental — precisely the phase SHIL exists to forbid.
- The trim branch is switched by a flip-flop clocked from the ring's own quadrature tap, so switching always happens midway between edges. Switching during an edge makes the two branches differ at that instant, the mux emits an extra edge, and that edge circulates as a second front — the ring jumps to a higher-order mode.

Reference ring

A ninth ring holds half its trim branch permanently closed, placing it in the **middle** of the spins' tuning range. A spin can only ever get slower than nominal, so a reference sitting at the fast end could never be caught up to.

Its doubled-frequency signal comes from a single XOR across its phase and quadrature taps. That lands exactly on 2f and tracks the spins across process, voltage and temperature — unlike a separate half-length ring, whose NAND and mux delays do not halve along with the inverter chain.

Modes

Mode is set on `ui[3:2]`:

Mode	Name	Behaviour
0	CHAR	All rings free-running. The ring selected by `uio[5:3]` appears on `uio[0]` divided by 256, the 2f reference on `uio[1]`. Use this to measure oscillator frequency, process spread, and temperature and voltage dependence.
1	LOCK	Coupling permanently enabled, no annealing ramp. Shows directly whether the rings lock to each other.
2	ANNEAL	Full operation. Set `ui[7]` to run; the coupling probability ramps up at the rate given by `ui[6:4]`.
3	HOLD	Coupling frozen, for stable read-out after a run.

Modes 0 and 1 are deliberate insurance: they return useful measurements even if the annealing in mode 2 fails to converge.

How to test

Measure the oscillators (mode 0). Hold reset, release it, set mode 0, select a ring on `uio[5:3]`, and count edges on `uio[0]`. Multiply by 256. Expect roughly 400–450 MHz at nominal supply, so about 1.7 MHz on the pin. Sweep the supply and repeat to get the voltage coefficient.

Check locking (mode 1). Load a coupling matrix, set mode 1, and watch `uo[7:0]`. If the rings lock, the pattern is stable; if they do not, it changes constantly.

Solve a problem (mode 2).

1. Assert reset, then release.
2. Shift in 64 bits of coupling matrix on `ui[0]` with `ui[1]` high — last row first, and within each row column 0 first. The matrix must be symmetric: bit `(r, c)` and bit `(c, r)` both set for an edge.

3. Set mode 2, rate on `ui[6:4]`, and raise `ui[7]`.
4. Wait for the ramp: roughly `2^(16-rate)` clock cycles.
5. Set mode 3 and read `uo[7:0]`. Bit `i` set means spin `i` has the same phase as spin 0.

Improve the hit rate by repeating. A single run lands on the optimum about 30 % of the time; the system sometimes freezes in a local minimum, as any annealer does. Five runs raise that to about 70 %, and a restart costs roughly a microsecond — just drop `ui[7]` and raise it again. Compute the energy of each result in software and keep the lowest.

External hardware

None required. A microcontroller driving the pins is enough; an oscilloscope or frequency counter on `uio[0]` is useful for mode 0.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	J matrix serial data in	spin 0 (always 1, reference)	OUT selected ring oscillator, divided by 256
`1`	J matrix shift enable	spin 1	OUT 2f reference, divided by 256
`2`	mode bit 0	spin 2	OUT running
`3`	mode bit 1	spin 3	IN ring select bit 0
`4`	anneal rate bit 0	spin 4	IN ring select bit 1
`5`	anneal rate bit 1	spin 5	IN ring select bit 2
`6`	anneal rate bit 2	spin 6	unused
`7`	start	spin 7	unused

Tbilisi CORDIC Engine

by **Levan Chagelishvili**

`0451` `50 MHz` `HDL Project`

`github.com/ChagelishviliLevan/Tbilisi_CORDIC_Engine`

"A fixed-point CORDIC engine that calculates sine and cosine using iterative shift-and-add operations."

How it works

The **Tbilisi CORDIC Engine** calculates sine and cosine using the iterative CORDIC rotation algorithm. It uses only arithmetic shifts, additions, and subtractions, avoiding hardware multipliers.

When start is asserted while *calc* is low, the engine accepts the selected angle and initializes the working vector with **x = 1/K and y = 0**, where **K is the CORDIC gain**. It then performs 14 micro-rotation iterations. During every iteration, the direction of rotation is selected from the sign of the remaining angle.

calc remains high while the calculation is running. After the final iteration, *calc* returns low and finish is asserted for one clock cycle. The calculated sine and cosine values are output on *sin_out* and *cos_out*.

The angle is represented by a *16-bit fixed-point* value in which **16'h0000 represents 0 degrees and 16'h4000 represents 90 degrees**. The **top-level** design contains 16 predefined angles from 0 to 90 degrees in steps of 6 degrees. Each accepted start request selects the current angle and advances the angle index for the next calculation.

The sine and cosine outputs are *signed 16-bit fixed-point values with 15 fractional bits*.

How to test

TODO: Explain how to use your project

External hardware

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

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	switch [0]	VGA red [1]	—
1	switch [1]	VGA green [1]	—
2	switch [2]	VGA blue [1]	—
3	switch [3]	VGA vsync	—
4	switch [4]	VGA red [0]	output sticky_failure
5	switch [5]	VGA green [0]	TM1638 inout dio
6	switch [6]	VGA blue [0]	TM1638 output clk
7	UART rx	VGA hsync	TM1638 output stb

miniMAC_IHP26b

by **Yann Guidon**

0453 | 100 MHz | HDL Project

`github.com/ygdes/miniMAC_IHP26b`

"16-bit Scrambler/Framer/Error detector for TTIHP26 run"

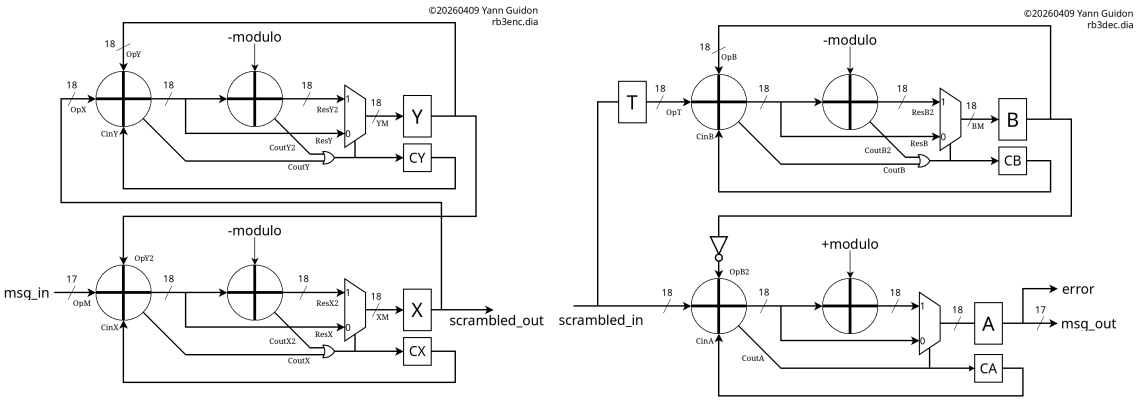
Note: This 3rd revision should work and be portable to IHP CMOS5L and G2. The gPEAC18 has been redesigned from scratch and uses only one cycle.

What is this miniMAC

IMPORTANT: This custom circuit and protocol is not at all compliant or even compatible, even remotely linked to any 802.3 standard. It's all explained and detailed on Hackaday at https://hackaday.io/project/198914

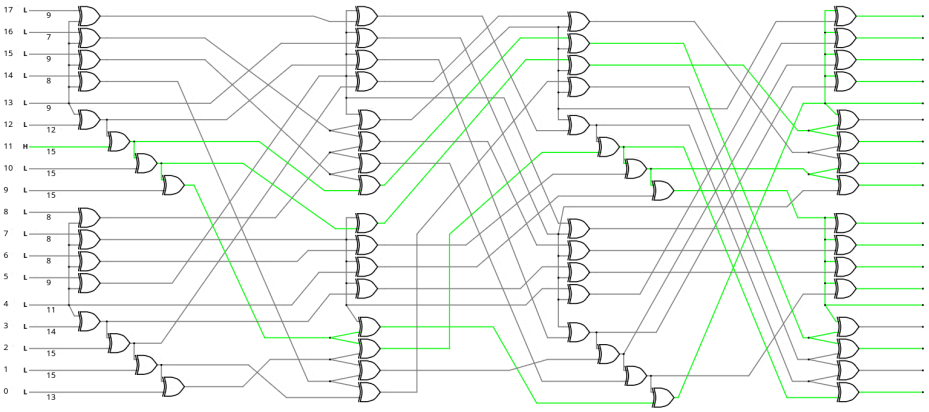
The miniMAC is a (currently partial) Media Access Controller for a simplified data link over twisted pairs. This part of the circuit provides error detection and scrambling of 16-bit data words, which are combined with a 17th bit for data/control framing (C/D). The 18-bit result is suitable for sending to a (custom) PHY (see https://hackaday.io/project/203186) for serialisation and line coding. This unit chains two sophisticated circuits:

- The term "gPEAC" means "generalised Pisano with End-Around Carry" (see https://hackaday.io/project/178998), a class of PRNG/scrambler/checksum that uses different mathematics than Galois-based LFSR. The gPEAC18 unit is a non-power-of-two additive-based scrambler-checksum, with modulus=258114, orbit=66623095108 and its state is impossible to flush or freeze. It combines the 17 bits and creates an extra check bit. They both work as super-parity bits.

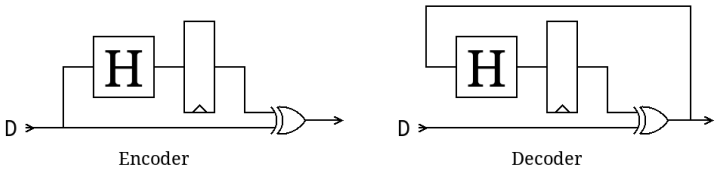


- "Hammer" is a contraction of the "Hamming distance maximiser". The Hammer18 unit is a XOR-based (bijective) scrambler that boosts the

Hamming distance on the scrambled 18-bit word. This version contains 3 layers of taylored permutations between 64 XOR2 gates, with very strong avalanche:



Conveniently, the same sea-of-XOR is identical, both for encoding and decoding, and the decoding side is “recursive” such that it amplifies any transmission error at the receiving end. The sorted avalanche for a single bitflip is : 7 8 8 8 8 9 9 9 9 1 12 13 14 15 15 15 15 15 (total=200). The 64 XOR2 gates have a propagation delay of 10 gates, yet the effective latency in the system is just one XOR in the critical datapath:



Encoder Decoder

These very different types of circuits are complementary, together they provide very strong scrambling, eliminate problems inherent with classical LFSRs, and detect errors very early. With an equivalent of 56 bits of state and uncrashable mathematics, the system remains fast, compact and tailored for safety and early retransmission to save bandwidth/latency and reduce buffer sizes (and cost). More circuits are required to implement the higher-level protocol, buffering and retransmission logic.

How it works

This version of gPEAC18 operates in one cycle, implementing two adders to first compute the sums, then to adjust the modulus. The Hammer18 circuit requires one depth of XOR, but at different places:

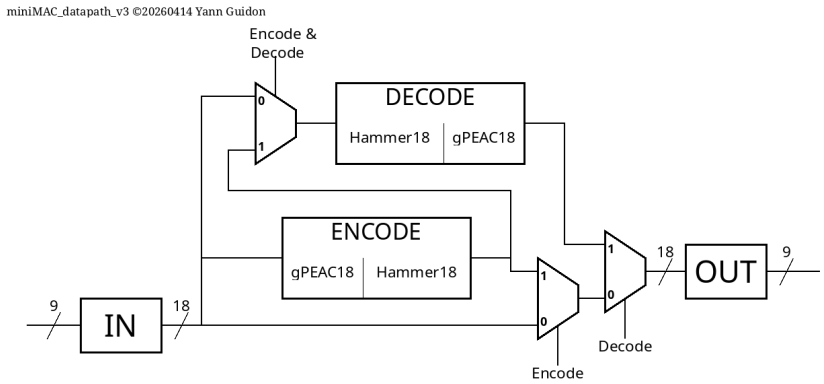
- For encoding, the input data goes through gPEAC18 then through Hammer18.
- For decoding, the scrambled data goes through Hammer18 before descrambling by gPEAC18.

Due to pin constraints, the 18-bit data words are transmitted in two cycles with 9-bit half-words. Counting input and output (2 cycles each), the overall latency is 5 cycles, following a sequence that is internally started when data is initially input with Den=1. Even at the low default 50MHz clock speed, that's still a bandwidth of 25M×18=450Mbps: fast enough to oversaturate a Cat5 twisted pair. P&R have been turned up to 200MHz, faster than the pins, so there is some margin.

This tile contains four main pipelined units, sequenced by a shift register:

- the input unit assembles a 18-bit word from two consecutive 9-bit halfwords (in SDR mode, although double data rate would have been possible)
- the encode unit scrambles 17 bits and generates a 18-bit word
- the decode unit descrambles the 18-bit word, restores the 17-bit word and eventually generates an error flag.
- the output unit splits the 18-bit words back into two consecutive 9-bit halfwords

The encode and decode units can be tested separately or together in the "loopback" mode.



How to test

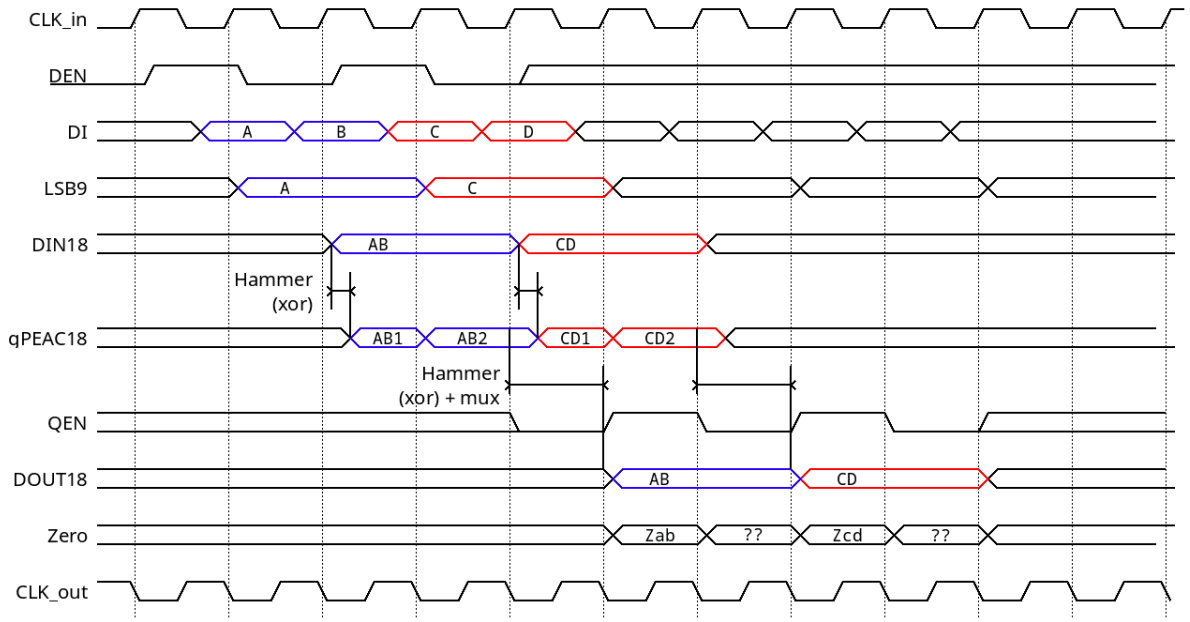
First let's examine the pinout. The inputs:

- CLK is the main clock, driving the whole circuit.
- Reset synchronously restores the registers' initial values.
- Enc and Dec select the pipeline routing mode:
  - Enc=0, Dec=0 : pass-through mode => no encodign or decoding, the delayed output must be identical to the input,
  - Enc=0, Dec=1 : decode mode => the scrambled input is restored to cleartext after 5 cycles.
  - Enc=1, Dec=0 : encode mode => the cleartext input is scrambled and output after 5 cycles,

    - Enc=1, Dec=1 : loopback mode => encodes then decodes, the delayed output (6 cycles) must be identical to the input.
- DI0 to DI8 are 9-bit data half-words that are sampled at the rising edge of CLK.
- DEN is Data ENable input, signalling the presence of the first 9-bit half-word of the pair on DI0:8. The second half MUST follow immediately, during the next cycle of CLK.

The outputs:

- CLK_out provides an appropriate clock, adjusted for phase and delay due to onchip routing, for easy chaining: output signals are updated on the falling edge of CLK_out.
- DO0 to DO8 are the data half-word.
- QEN is the “output enable” generated by the internal sequencer, that signals that a first half-word is available.
- Z is a flag set to 1 when the decoded output has DO[15:0] cleared (think of a 16-bit NOR), useful to implement the higher-level protocol.



Notes :

- Internal delay from input to output:
    - pass-through: 1 cycle
    - encode, decode: 2 cycles
    - loopback : 3 cycles
- Data half-words are clock-sourced, to allow seamless chaining of multiple chips.
- Decoding errors (and Zero) are signalled but not managed/acted upon, a FSM and appropriate circuits must reset the registers.

- The input/plaintext word contains the C/D bit and an unused bit. C/D should be on Dx8 of the first halfword for best detection.
- The output/scrambled word contains the C/D bit and an error bit (saves a pin)
- Asserting DEN during more than one cycle is an error condition.
- The Zero output is always active (encoding as well as decoding) but gives a valid result only when QEN is asserted. It does only check the data bits: [7:0] and [17:9], conveniently mapped to the output byte pins.
- Do not change the Enc and Dec while data are in the pipeline, do it during the Reset state.

External hardware

Custom boards or adapters will be made. I will try to get a pair of chips to connect together, such that I can verify a whole transmission chain.

Next developments

This started as a VHDL to Verilog+IHP PDK port but it will likely grow to a more-featured project.

- I'll try to get 2 boards to test both coder and decoder in a chain, to simulate noisy communications.
- In parallel I try to design a decent PHY, a much more difficult endeavour.
- Logically, the MAC must be completed with a FSM, a buffer and a host interface, likely in the next tapeout so I can play with memory blocks.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DI0	DO0	D08
1	DI1	DO1	QEN
2	DI2	DO2	CLK_out
3	DI3	DO3	Zero
4	DI4	DO4	Enc
5	DI5	DO5	Dec
6	DI6	DO6	DEN
7	DI7	DO7	DI8

SG13G2 Mystery Circuit

by **Oscar Castaneda** & **Thomas Herzog**

0455 HDL Project

github.com/ofcastaneda/tt-sg13g2-mystery

"A reference SG13G2 implementation for the EZ130 mystery circuit"

How it works

This is a custom-GDS design using the SG13G2 library. It implements a 32-bit multiplier, whose 64-bit output is divided into continuous 8-bit chunks. These eight 8-bit chunks are added together into an 11-bit word that is offered at the output.

How to test

Coming soon!

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SerialIn0	Sum0	Sum8
1	SerialIn1	Sum1	Sum9
2	SerialIn2	Sum2	Sum10
3	SerialIn3	Sum3	HI
4	SerialEn	Sum4	LO
5	InvertIn	Sum5	HI
6	MaskIn	Sum6	SerialOut0
7	WrEnIn	Sum7	RegOut0

EZ130 7T Mystery Circuit

by **Oscar Castaneda** & **Thomas Herzog**

0457 HDL Project

github.com/ofcastaneda/tt-ez130-7t-mystery

"A mystery circuit implemented with preliminary EZ130 7T cells"

How it works

This is a custom-GDS design using a preliminary version of the EZ130 7T library from ETH Zurich. It implements a 32-bit multiplier, whose 64-bit output is divided into continuous 8-bit chunks. These eight 8-bit chunks are added together into an 11-bit word that is offered at the output.

How to test

Coming soon!

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SerialIn0	Sum0	Sum8
1	SerialIn1	Sum1	Sum9
2	SerialIn2	Sum2	Sum10
3	SerialIn3	Sum3	HI
4	SerialEn	Sum4	LO
5	InvertIn	Sum5	HI
6	MaskIn	Sum6	SerialOut0
7	WrEnIn	Sum7	RegOut0

Mini 8-bit Accumulator CPU

by **Rodgers Odiwuor**

`0459` `10 MHz` `HDL Project`

`github.com/94442024-cpu/-mini-cpu-94442024`

> *"A tiny Von Neumann 8-bit accumulator CPU with a 16-instruction ISA (load/store, add/sub, bitwise logic, jump/jump-if-zero/jump-if-carry, output, halt)."*

How it works

This is a tiny 8-bit accumulator-based CPU with a Von Neumann architecture: 16 words of 8-bit memory shared between code and data. Each instruction is 8 bits: a 4-bit opcode and a 4-bit operand (a memory address or an immediate value). The CPU runs a simple two-state fetch/execute loop.

The built-in demo program computes (3 + 4), XORs the result with 9, inverts the final value, and outputs each intermediate result via the OUT instruction, then halts.

Supported instructions: NOP, LDA, ADD, SUB, STA, LDI, JMP, JZ, JC, OUT, ADI, AND, OR, XOR, INV, HLT.

How to test

Reset the chip (pulse rst_n low then high). Watch uo_out: it should show 7, then 14, then 0xF1 (241), in that order. The halted flag on uio_out[7] should go high once the program finishes. uio_out[3:0] exposes the live program counter and uio_out[4] the FSM state for debugging.

External hardware

None required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	—	acc bit 0 (OUT register)	debug: pc bit 0
1	—	acc bit 1 (OUT register)	debug: pc bit 1
2	—	acc bit 2 (OUT register)	debug: pc bit 2
3	—	acc bit 3 (OUT register)	debug: pc bit 3
4	—	acc bit 4 (OUT register)	debug: FSM state

#	Input	Output	Bidirectional
5	—	acc bit 5 (OUT register)	debug: zero flag
6	—	acc bit 6 (OUT register)	debug: carry flag
7	—	acc bit 7 (OUT register)	debug: halted flag

Tiny Dual-Channel DRAM-PIM Controller + PU

by **Corey Lammie**

481 | 10 MHz | HDL Project

github.com/coreylammie/Tiny-Dual-Channel-DRAM-PIM-Controller-for-TTIHP26b

"Standard-cell DRAM-like controller with a shared near-bank PU"

This project is a compact standard-cell DRAM-PIM controller plus near-bank processing unit for TinyTapeout IHP.

The controller exposes a 32-bit SPI command interface and implements two logical DRAM-like channels. Each channel has two banks, two 8-bit rows per bank, forced-refresh state, sticky error handling, and an 8-bit accumulator. One lane-serial near-bank PU is shared between the channels.

Implemented operations are `ACT`, `PRE`, `WR`, `RD`, `REF`, `STATUS`, `ABORT`, `NOP`, experimental `ATTEND`, `DOT`, `MAC`, and accumulator byte reads. `DOT` and `MAC` use unsigned INT1 bit semantics or signed INT4 lane semantics and are computed through the shared lane-serial accumulator datapath. `ATTEND` consumes the accumulator score from a preceding `DOT`/`MAC` to perform a packed INT4 attention-value update. `CONFIG`, INT2/INT8 compute, and the generic `VADD` slot are reserved in this area-focused branch.

Project Pinout

Digital Pins

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

IEEE DOORSH

by **Koło Naukowe BAZA**, **Politechnika Warszawska — ZT**, **PN**, **MK**, **JT** & **KK**

`0483` `1 MHz` `HDL Project`

`github.com/magnetoField/ieee-sep-2026-1st-ihp`

"PIN-authorized SIMON64/128 challenge-response authenticator with a 4x4 keypad and passive buzzer"

How it works

IEEE DOORSH is a PIN-authorized SIMON64/128 challenge-response authenticator developed by **Koło Naukowe BAZA, Politechnika Warszawska** (ZT, PN, MK, JT, KK) for Tiny Tapeout IHP26b.

The ASIC scans a 4×4 matrix keypad. Entering `1234` authorizes exactly one 64-bit challenge. The host sends the challenge over the SHIFT64 serial link, the ASIC encrypts it with the fixed demonstration key, and the host reads the 64-bit response in a second frame. A wrong PIN does not open the interface or produce a response. Three wrong PINs block further attempts until cold reset.

`SCLK` is sampled by the 1 MHz project clock; it is not an independent clock. Keep each SCLK high and low phase at least 16 project-clock cycles and change `CS_n` only while SCLK is low.

How to test

1. Apply reset and enable the project.
2. Press and release `1`, `2`, `3`, `4`.
3. Wait for `CHALLENGE_READY` on `uo_out[0]`.
4. Send `F15654A8D25FFA1C`, MSB first, on `SDI` using one 64-bit frame.
5. Wait 10,000 project-clock cycles (10 ms at 1 MHz).
6. Read a second 64-bit frame from `SDO`.

The expected response is `BA2A5234DEADBEEF` for the fixed key `BA2A19181312111 00B0A090803020100`. The distinctive hexadecimal pattern makes a successful transaction easy to recognize without any text decoding.

External hardware

- passive 4×4 matrix keypad;
- pull-ups on the four keypad column inputs;
- host capable of the SHIFT64 timing described above;
- passive buzzer with a suitable transistor or logic driver.

`uo_out[3]` generates a nominal 2 kHz tone for the passive buzzer for 50 ms after an accepted key event. Do not power a high-current transducer directly from the Tiny Tapeout output pad.

Pinout

Pins	Function
`ui_in[3:0]`	Keypad columns C0–C3, active low
`ui_in[4]`	SHIFT64 SDI
`ui_in[5]`	SHIFT64 SCLK
`ui_in[6]`	SHIFT64 CS_n
`ui_in[7]`	Unused
`uo_out[0]`	CHALLENGE_READY
`uo_out[1]`	Unused, tied low
`uo_out[2]`	SHIFT64 SDO
`uo_out[3]`	2 kHz passive-buzzer drive
`uo_out[7:4]`	Unused, tied low
`uio[3:0]`	Open-drain keypad row enables R0–R3
`uio[7:4]`	Unused, high impedance

The key and PIN are public demonstration constants. IEEE DOORSH is an educational ASIC demonstrator, not a production security device.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	keypad column 0, active low	CHALLENGE_READY after correct PIN	keypad row 0 open-drain enable
1	keypad column 1, active low	unused, tied low	keypad row 1 open-drain enable
2	keypad column 2, active low	SHIFT64 SDO	keypad row 2 open-drain enable
3	keypad column 3, active low	2 kHz passive buzzer drive	keypad row 3 open-drain enable
4	SHIFT64 SDI	unused, tied low	unused
5	SHIFT64 SCLK	unused, tied low	unused
6	SHIFT64 CS_n	unused, tied low	unused
7	unused	unused, tied low	unused

ULSR demo

by **Yann Guidon**

0485 1 Hz HDL Project

`github.com/ygdes/ULSR_demo`

"Ultracompact 2-pin scan chain"

How it works

As the name "Universal Latch-based Shift Register" implies, it's a shift register based on individual latches. The first advantage of this asynchronous method is the need of only two input signals.

- Instead of running on a single high-fanout clock pulse, there are two independent clock signals SD and SC with relaxed timing and fewer buffers.
- Instead of using one DFF per bit, it is split into its Master and Slave latches

All the operations, like shifting one bit, require non-overlapping, alternating pulses on SD and SC. This takes care of any transient condition and glitches. Capture and Update require slightly more complicated sequences but still use only SD and SC. The only exception is RESET, which is performed with an overlapping sequence.

How fast does it run ?

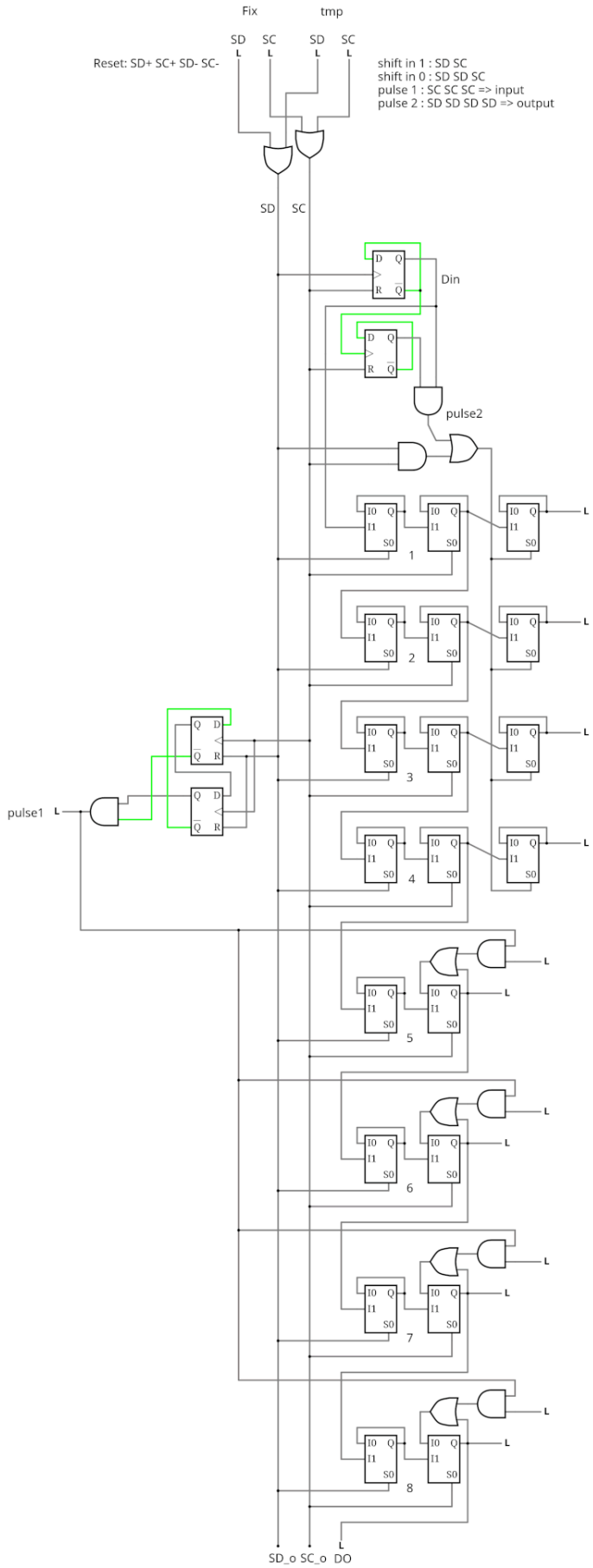
I have no idea, it's all asynchronous magic. Short chains should reach 10 or 20MHz easily but since it's meant to be driven from some Arduino-style MCU through USB or a serial port, raw speed is not critical. The focus is on minimising any impact on the target circuit (surface, routing, power,...).

Architecture

Below is an abridged diagram showing only 4 in and 4 out pins. Source : DTAP2-ULSR on Hackaday.io where transparent latches are replaced by back-fed MUX2 for simulation sake. There are 3 main stages :

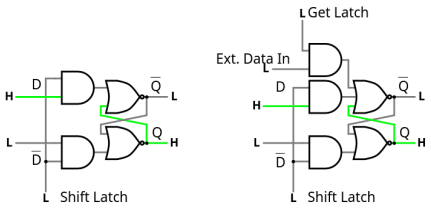
- data input generator (recreates the missing data in pin's value)
- output block (somewhat merged with the previous block)
- input block (with its own counter)

The actual implementation has 12 bits of depth, with the 4 middle ones being both inputs and outputs.


Fix
tmp
SD SC SD SC
Reset: SD+ SC+ SD- SC-
shift in 1 : SD SC
shift in 0 : SD SD SC
pulse 1 : SC SC SC => input
pulse 2 : SD SD SD SD => output
SD SC
Din
pulse2
pulse1
1
2
3
4
5
6
7
8
SD_o SC_o DO

Structure

The input, output and inout stages are made from 2 or 3 latches, made from standard A21OI and A221OI cells. They are smaller than a DFF.

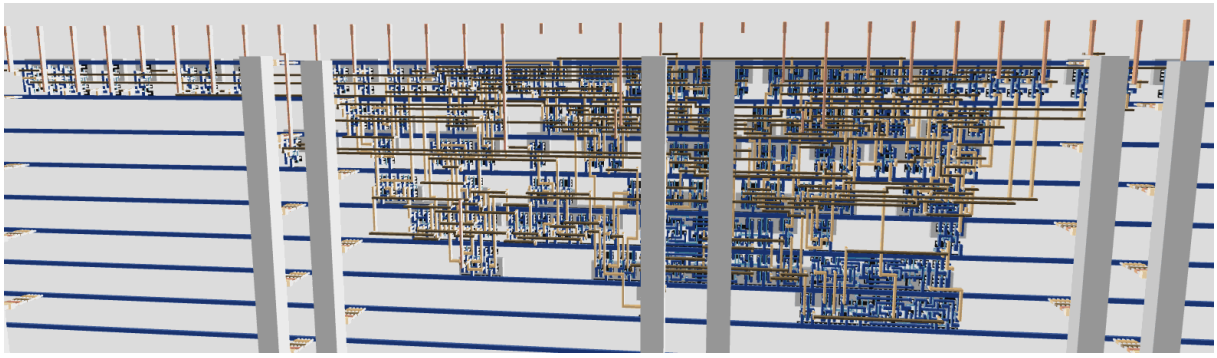


Cell usage

For 8 inputs and 8 outputs (including 4 combined in and out)

- `a22oi a221oi a21oi : 66` (including one for the full adder, the rest for the bulk of the scan chain)
- `buf : 35` (I never asked for them)
- `inv : 4` (including one for the full adder)
- `dfrbp : 4` (for the decoders/counters)
- `xor2 : 2` (Full adder)
- `and2 : 1` capture decoder.

Result:



Conclusion : `112 total cells (excluding fill and tap cells)` And it's still bloated by the toolchain with

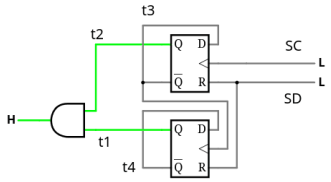
- many unwanted buffers (added at the interface),
- some "constant" cells (tiehi, tielow) for the io dir port and unused outputs,
- some gates for the extra full adder...

But can you do something this compact with the JTAG standard? And since each bit/stage does not require absolute synchronism or a tight timing, the clock network is relaxed and each register uses less space than a standard DFF.

Asynchronous design

Asynchronous circuits are frowned upon and notoriously prone to weird errors. After all, synchronous designs are much easier to analyse and get right. However the latter relies on the asumption of a magically simultaneous clock pulse that arrives everywhere at the same time. The ULSR does not assume this, in particular to make implementation more convenient: no clock tree synthesis, saving power and area. The ULSR runs slower the JTAG and requires more timing margin, but that's only a fair compromise.

One consequence is that a clock signal can drive only one of the synchronous gates like DFF, which must be chained, like below. This lets the Place&Route tools optimise for density without consideration for timing. Since only one DFF gets a clock signal, the setup & hold only depends on the host toggling the SD and SC pins slow enough, though there is ample margin at 1Mbps.



Protocol

You can find these operations in the `test/test.py` script.

- RESET is : set SD high, set SC high, set SD low, set SC low.
- inject a '0' bit in the chain : pulse SD, pulse SD, pulse SC.
- inject a '0' bit in the chain : pulse SD, pulse SC.
- Capture : RESET then pulse SC 4 times
- Update : pulse SD 4 times

A more elaborate protocol can be designed on top of this, for example: addressing specific registers by counting the number of bits injected. Let your imagination go wild!

How to test

Use some Arduino for example, and play with the SD/SC signals. About 1us between each bit toggle is a good ballpark. An Arduino sketch will be provided someday, transcribing the code in `test/test.py`

5 leftover pins are connected to a Full Adder that you can test it with the scan chain through external wires. Have fun injecting errors to see if the scan chain can detect them!

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	I0	O0	SD
1	I1	O1	SC
2	I2	O2	DO
3	I3	O3	FA1
4	I4	O4	FA2
5	I5	O5	FA3
6	I6	O6	FAS
7	I7	O7	FAC

ieee_tt_tinyopt4

by **Berthyn Rodrigo Tinini Chuquimia**

`0487` `10 MHz` `HDL Project`

`github.com/tiberthyn/ieee_tt-tinyopt4`

"4-tap LMS adaptive update engine for FIR system identification"

How it works

`tt_um_tinyopt4` is a compact sequential parameter update engine designed specifically for adaptive FIR identification. The core executes an 11-state, time-multiplexed schedule with a 10-cycle processing path around a shared arithmetic unit:

1. **State `S_IDLE`:** Polls control bits; captures incoming data sample $x[n]$ into the 4-element shift register when `start = 1` and `data_sel = 0`.
2. **States `S_MAC0` to `S_MAC3`:** Feeds pairs $(w_i[n], x[n-i])$ sequentially into the single signed 8×8 multiplier. Products are added into the 18-bit accumulator.
3. **State `S_ERR`:** Clamps the normalized accumulator to the 8-bit range $[-128, +127]$, subtracts this estimate from reference $d[n]$, and registers the bounded difference $e[n]$.
4. **States `S_UPD0` to `S_UPD3`:** Re-routes the multiplier to compute $e[n] \cdot x[n-i]$. The 16-bit intermediate product passes through an arithmetic barrel shifter configured by s, saturates, and updates register $w_i[n+1]$.
5. **State `S_DONE`:** Pulses `done` high for one cycle, deasserts `busy`, and returns the FSM to `S_IDLE`.

How to test

Verification relies on the Cocotb test framework running under Icarus Verilog (`iverilog`):

- **Target Verification Vector:** Automated simulation feeds pseudo-random $Q1.7$ sequences through a static target plant $W^* = [64, -32, 16, -8]$.
- **Pass/Fail Assertion:** After 1500 consecutive update cycles, the test checks that all four hardware registers converge within $| w_i - W_i^* | \leq 10$ LSB to account for truncated intermediate precision.
- **Gate-Level Check:** The post-synthesis netlist generated by LibreLane is re-simulated using the same Cocotb harness to verify zero functional regression across cell delays.

How to use

Bus and Control Mapping

Interface	Signal	Role
`ui_in[7:0]`	Data / Address	Input values ($x[n]$, $d[n]$, or shift exponent s). When idle, bits `[2:0]` act as register read address.
`uo_out[7:0]`	Data Out	Monitored byte selected by `ui_in[2:0]`.
`uio_in[0]`	`start`	Clock-synchronous execution strobe.
`uio_in[1]`	`data_sel`	`0` = Shift sample $x[n]$; `1` = Latch reference $d[n]$ and trigger 10-cycle update.
`uio_in[2]`	`cfg_sel`	`1` = Write learning rate exponent s from `ui_in[2:0]`.
`uio_out[4]`	`busy`	Held high by FSM during the 10 processing cycles.
`uio_out[5]`	`done`	Single-cycle pulse marking coefficient update completion.
`uio_out[6]`	`overflow`	Latched high if prediction or error calculation exceeds $Q1.7$ limits.
`uio_out[7]`	`w_sat`	Latched high if any tap update hits ± 127 saturation bounds.

Operational Procedure

1. **Reset Sequence:** Assert `rst_n = 0` for 3 clock cycles, then release (`rst_n = 1`).
2. **Set Step Exponent (Optional):** Drive `ui_in[2:0] = s` (e.g., 3 for $\mu = 2^{-3}$), pulse `uio_in = 3'b101` for one cycle, then return `uio_in` to `0`.
3. **Feed History Buffer:** For each of the initial samples, place $x[n]$ on `ui_in[7:0]`, strobe `uio_in = 3'b001` for one cycle, and wait one cycle.
4. **Trigger Filter & Adaptation:** Place $d[n]$ on `ui_in[7:0]`, strobe `uio_in = 3'b011` for one cycle, then clear `uio_in = 0`.
5. **Monitor Flags:** Wait 10 clock cycles until `uio_out[4]` (`busy`) returns low and `uio_out[5]` (`done`) pulses high.
6. **Readback Register:** With `uio_in = 0`, apply the target address to `ui_in[2:0]`:

- `3'b000`: Predicted response $\hat{y}[n]$
- `3'b001`: Error residual $e[n]$
- `3'b010` to `3'b101`: Filter taps w_0 to w_3

External hardware

None required. The module operates self-contained on any standard Tiny Tapeout carrier board:

- **Manual Inspection:** On-board DIP switches drive control/data lines; on-board LEDs display status flags and output bytes.
- **Automated Benchmarking:** Any standard 3.3V microcontroller (RP2040, ESP32) or USB-to-GPIO interface can drive the 8-bit parallel bus up to 10 MHz.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Data input bit 0 (x / d / cfg_mu)	Multiplexed output bit 0 (y_hat / e / w0-w3)	In: start
1	Data input bit 1	Multiplexed output bit 1	In: data_sel (0=x, 1=d)
2	Data input bit 2	Multiplexed output bit 2	In: cfg_sel (1=config mu)
3	Data input bit 3	Multiplexed output bit 3	In: unused / tied low
4	Data input bit 4	Multiplexed output bit 4	Out: busy
5	Data input bit 5	Multiplexed output bit 5	Out: done
6	Data input bit 6	Multiplexed output bit 6	Out: overflow flag
7	Data input bit 7	Multiplexed output bit 7	Out: weight saturation flag

IEEE VGA Animated Beach

by **Jose Miguel Caicedo Ortiz**

`489` `25.175 MHz` `HDL Project`

`github.com/JMCOMaster/ieee_vga_beach_animation`

> *"This project implements a dynamic 640x480 VGA beach scene animation for the Tiny Tapeout platform using pure combinational logic. The hardware generates a rhythmic moving tide, a sun with twinkling rays, a flying airplane, a palm tree, and a beach ball with a projected shadow. It utilizes a 6-bit RGB palette and internal frame counters to render and animate complex geometric shapes without requiring external RAM."*

How it works

The design generates a dynamic 640x480 VGA beach scene using pure combinational logic, entirely bypassing the need for external RAM or internal frame buffers. To fit within a single Tiny Tapeout tile, traditional circular equations ($x^2 + y^2 = r^2$) requiring resource-heavy hardware multipliers were replaced with Manhattan distance calculations ($| dx | + | dy | < r$). This area optimization utilizes simple adders and bitwise inversions to generate octagonal and diamond shapes, such as the sun, beach ball, and coconuts, reducing silicon area consumption by over 90%.A 10-bit frame counter driven by the vsync signal orchestrates all synchronous animation.

It acts as a shared timebase for the airplane's horizontal translation across the sky, the sun's twinkling rays using XOR bitwise logic, and the rolling sea foam. Additionally, a dynamic tide system continuously oscillates the shoreline's vertical position. A dedicated up/down counter increments every four frames, creating a smooth, sweeping tide that interacts with the sand and leaves a darker wet sand color band at its maximum height.

Finally, a priority multiplexer acts as the rendering layer engine. It evaluates the mathematical boundaries for the current pixel and outputs a 6-bit RGB signal, where foreground elements like the towel and beach ball are evaluated first in the conditional chain to naturally overlap background elements like the sea and sky.

How to test

For web simulation, paste the Verilog source code into an online simulator like VGA Playground (https://vga-playground.com/), ensuring the clock is set to the industry standard 25.175 MHz for 640x480 resolution at 60Hz, and define the top module as tt_um_vga_example. To test locally, run the included Cocotb testbench by executing make inside the test directory.

This stimulates the clock and reset inputs to verify that the hardware sync signals operate at the correct timings, allowing you to inspect the resulting output file using GTKWave. Once manufactured on the Tiny Tapeout ASIC, hardware deployment requires connecting a TinyVGA PMOD to the output pins, supplying a 25.175 MHz clock, pulling the reset pin high, and connecting a standard VGA monitor to view the real-time animation.

External hardware

Tiny VGA Pmod driving a VGA monitor.

Project Pinout

Digital Pins

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

IEEE Acoustic Drone Detector

by **Jędrzej Hasiura** & **Grzegorz Pawelski**

`0491` `50 MHz` `HDL Project`

`github.com/magnetoField/ieee-sep-2026-2nd-ihp`

"Self-contained acoustic drone detector: PDM microphone in, detection signal out."

How it works

This project is a self-contained acoustic drone detector. A one-bit PDM microphone stream enters on `ui[0]`; an active-high detection signal appears on `uo[1]`, `uo[2]`, and `uo[3]`. The complete model is fixed in the logic, with no processor, firmware, RAM, or external weight storage.

The multiplier-free integer pipeline is:

1. Divide the 50 MHz input clock by 32 and output the resulting 1.5625 MHz PDM microphone clock on `uo[0]`.
2. Feed microphone samples through a nine-stage dyadic one-pole cascade. One shared subtract/shift/add datapath services the rotating state ring.
3. Difference adjacent stages to obtain five octave bands, then encode their magnitudes as four-bit logarithmic features.
4. Keep each band's maximum over a 41.9 ms frame. Sixteen frames form a 671 ms classifier window.
5. Accumulate four 16×5 ternary templates in signed six-bit saturating accumulators. Two staggered windows reduce boundary sensitivity.
6. Requantize the hidden activations, apply the ternary output layer, compare against the trimmed threshold, and hold a detection for one 41.9 ms frame. The hold is deliberately short: a drone is a steady source that is still there on the next window, so the output tracks it instead of latching. An earlier build held for 629 ms, which lagged the aircraft and ran two passes together into one.

The shipped `drone_dads_lvl12_s4` weights were trained on DADS with every clip re-levelled across a −12…0 dB span, so the template is not tied to one recording level. Scored with the bit-exact chip model they reach 99.15 % AUC on the validation split and 98.98 % on the held-out test split; the seed was chosen on validation only. The RTL subtracts the header's own feature centre (`FEAT_OFF = WW_CENTRE = 8`) and asserts the equality at elaboration, and the classifier dot product is computed exactly before the six-bit saturation – both are fixes to the earlier `drone_4` build, which ran a different model from the one it was measured on. Hardening of this build completed in one IHP sg13g2 1×1

tile at 93.78% final core utilization with zero DRC, LVS, antenna, max-fanout, setup and hold violations at all three corners.

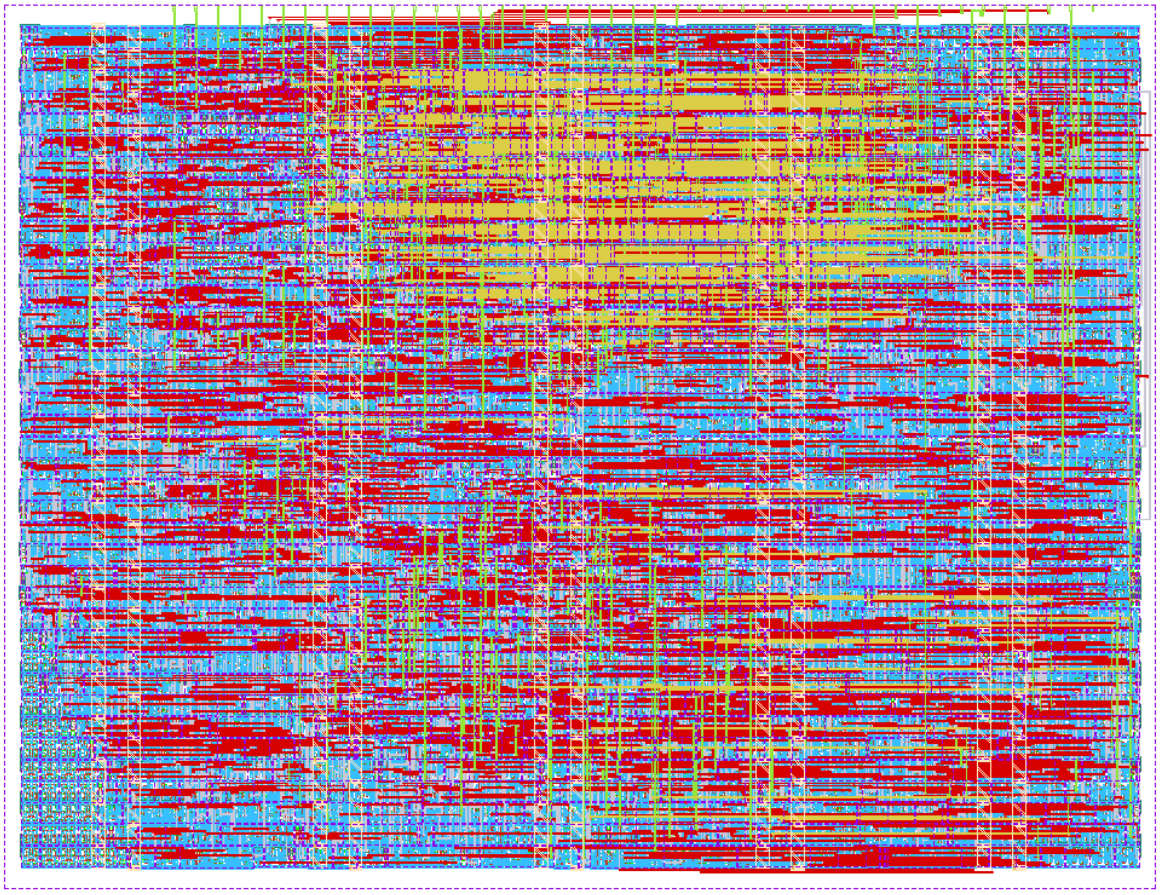


Figure 491.1: Reference drone_4 layout

How to test

1. Connect a compatible PDM microphone's data pin to `ui[0]` and clock pin to `uo[0]`, together with the appropriate board power and ground.
2. Apply the specified 50 MHz system clock and release reset.
3. Start with the neutral threshold trim `ui[7:1] = 7'd64`, which gives the shipped threshold of 5, the validation max-accuracy point. Detection appears on `uo[3:1]`.
4. For fewer false triggers, use trim 65 (threshold 9). For higher recall, use trim 63 (threshold 1).
5. Check `uo[7:4]` for a changing band-level value to confirm that microphone data is reaching the front end.

Each trim increment changes the decision threshold by four score units. On the held-out test split the operating points were 96.6% recall / 4.4% negative clips firing at threshold 5 (trim 64), 74.8% / 1.1% at threshold 9 (trim 65) and 99.7% / 13.9% at threshold 1 (trim 63). Synthetic room tone never fires at

threshold 5. These clip-level figures do not predict alarms per hour in a real deployment.

External hardware

- One PDM MEMS microphone compatible with the TinyTapeout board's I/O voltage.
- Two signal connections: microphone data to `ui[0]` and microphone clock from `uo[0]`.
- DIP switches or another seven-bit source on `ui[7:1]` for threshold trim.
- An LED or logic input on any of `uo[1]`, `uo[2]`, or `uo[3]`.

All `uio` pins are debug outputs: `uio[3:0]` is the frame index, `uio[4]` is detection, `uio[6:5]` is the internal FSM state, and `uio[7]` is the microphone tick.

Limitations

The detector was evaluated on DADS clips, not long-duration recordings at a deployed site. Detection range is unknown. Rotorcraft, engines, lawn equipment, and other sustained low-frequency sounds are plausible confusers. Treat the output as a sensor indication that requires deployment-specific validation, not as an authenticated identification of an aircraft.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PDM microphone data input	PDM microphone clock (1.5625 MHz)	Frame index bit 0
1	Threshold trim bit 0 (LSB)	Drone detected	Frame index bit 1
2	Threshold trim bit 1	Drone detected (mirror)	Frame index bit 2
3	Threshold trim bit 2	Drone detected (mirror)	Frame index bit 3
4	Threshold trim bit 3	Band level bit 0	Drone detected (debug)
5	Threshold trim bit 4	Band level bit 1	FSM state bit 0
6	Threshold trim bit 5	Band level bit 2	FSM state bit 1
7	Threshold trim bit 6 (MSB)	Band level bit 3	Microphone tick

Low-Power LFSR-Based Test Pattern Generator

by **Umamaheswari Ramalingam**

513 | 10 MHz | HDL Project

`github.com/umaece1982/lfsr-test-pattern-generator`

"An 8-bit LFSR-based pseudo-random test pattern generator for VLSI testing."

How it works

This project implements an 8-bit Linear Feedback Shift Register (LFSR) for pseudo-random test-pattern generation for VLSI testing.

The LFSR uses the polynomial:

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

The 8-bit seed is provided through the dedicated input pins.

The control signals are:

- `uio_in[0]` – Load seed
- `uio_in[1]` – Enable LFSR

When reset is asserted, the LFSR is initialized to a non-zero value. When LOAD_SEED is enabled, the input seed is loaded into the LFSR. When ENABLE is asserted, the LFSR advances by one state on every rising clock edge.

The generated pseudo-random test pattern is available at `uo_out[7:0]`.

The all-zero state is prevented because an LFSR would remain locked in the zero state.

How to test

1. Apply reset.
2. Provide a non-zero 8-bit seed through `ui_in[7:0]`.
3. Assert `uio_in[0]` to load the seed.
4. Deassert `uio_in[0]`.
5. Assert `uio_in[1]` to enable LFSR operation.
6. Apply clock pulses.
7. Observe the pseudo-random sequence at `uo_out[7:0]`.

The project includes an automated Cocotb testbench that verifies reset, seed loading and multiple consecutive LFSR states.

External hardware

No external hardware is required for RTL verification.

For physical silicon testing, the generated output can be observed using the Tiny Tapeout demonstration PCB and suitable digital test equipment.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Seed[0]	LFSR[0]	LOAD_SEED
1	Seed[1]	LFSR[1]	ENABLE
2	Seed[2]	LFSR[2]	—
3	Seed[3]	LFSR[3]	—
4	Seed[4]	LFSR[4]	—
5	Seed[5]	LFSR[5]	—
6	Seed[6]	LFSR[6]	—
7	Seed[7]	LFSR[7]	—

Mini 8-bit Processor

by **Jorge Gutierrez**

0515 100 Hz HDL Project

github.com/directsgg/mini-proceo-8bit

"An 8-bit processor with a custom-designed instruction set and hardware-level experimentation"

How it works

This project implements a small 8-bit accumulator-based processor. It includes an 8-bit accumulator, program counter, instruction register, address and data registers, a 16-byte memory, input/output register and a five-step fetch/execute sequence. The instruction set provides memory, arithmetic, logic, control-flow, input/output, and accumulator-control operations.

The processor is controlled through `rio_in` and bidirectional `uio_in` pins. Results are available on `rio_out`, while `uio_out[7]` indicates whether the processor is running.

How to test

Press reset first. To load each memory word, write its address on `rio_in`, pulse `write_ar` on `uio_in[1]`, pulse `clear_rio` on `uio_in[0]`, write the instruction or data value on `rio_in`, and pulse `write_mem` on `uio_in[3]`. The current value of `RIO` can be observed on `rio_out` during this process. Repeat this sequence for the following minimal program:

```
0x00: 0x1A    LOAD 0x0A
0x01: 0x3B    ADD  0x0B
0x02: 0xB1    OUT
0x03: 0x01    HALT
0x0A: 0x05
0x0B: 0x03
```

For example, to store `0x3B` at address `0x01`, write `0x01` to `rio_in`, pulse `write_ar`, clear `RIO`, write `0x3B` to `rio_in`, and pulse `write_mem`. After loading all words, press reset again and pulse `start_cpu` on `uio_in[5]` to run the program. The expected result is `0x05 + 0x03 = 0x08`, visible on `rio_out`; `run_cpu` on `uio_out[7]` indicates execution and becomes inactive after `HALT`.

External hardware

List external hardware used in your project (e.g. PMOD, LED display, etc), if any 9 LEDs and 15 push buttons

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	rio_in[0]	rio_out[0]	clear_rio
1	rio_in[1]	rio_out[1]	write_ar
2	rio_in[2]	rio_out[2]	read_ar
3	rio_in[3]	rio_out[3]	write_mem
4	rio_in[4]	rio_out[4]	read_mem
5	rio_in[5]	rio_out[5]	start_cpu
6	rio_in[6]	rio_out[6]	stop_cpu
7	rio_in[7]	rio_out[7]	run_cpu

uart

by **bella**

517 | 49.152 MHz | HDL Project

`github.com/blonghi/tt_um_uart`

"basic uart w rx and tx"

How it works

This project implements a simple UART (Universal Asynchronous Receiver + Transmitter). It's capable of independently transmitting and receiving 8-bit serial data (8-N-1).

Field	Value
Data bits	8
Parity	None
Stop bits	1

The design is split into four modules:

- **`tt_um_blonghi_uart`** - the top-level wrapepr that maps pins to signals and instantiates the 3 following modules.
- **`baud_rate_gen`** - generates two enable ticks from the system clock. 'tx_counter' wraps around at 5119, producing 'tx_enb' (one pulse per baud period). 'tx_counter' resets every 'tx_sync' pulse (the start of a new frame), keeping TX bit timing aligned. 'rx_counter' wraps around at 319 (1/16th of 'tx_counter's range) This produces 'rx_enb' at 16x the rate to allow the receiver to oversample and locate the center of each incoming bit. 'rx_counter' resets every 'rx_sync' pulse (start-bit detection), keeping RX sampling aligned.
- **`transmitter`** - FSM that on write request serializes an 8-bit byte onto the tx line as: start bit, 8 data bits (LSB first), then a stop bit.

```
stateDiagram-v2
direction LR
    [*] --> IDLE
    IDLE --> START: wr_enb
    START --> DATA: tx_enb
    DATA --> STOP: 8 bits transmitted
    STOP --> IDLE: tx_enb
```

- **`receiver`** - FSM that watches the rx line for a falling edge (start bit) then samples 8 data bits at the center od each bit period (16x oversampling to

find bit-center), then checks for the stop bit and pulses rx_valid for one cycle with the received byte on rx_data.

```
stateDiagram-v2
direction LR
    [*] --> IDLE
    IDLE --> START: falling edge
    START --> DATA: rx_enb
    DATA --> STOP: 8 bits received
    STOP --> IDLE: valid stop bit
```

Limitations

- No parity bit.
- If there is a bad frame, it is simply never latched with no downstream indication the error occurred.
- `rx_data` is split between two buses: `uio_out` and `uo_out`

Pin mappings

TX (transmit) | Signal | Pin | Direction | |—|—|—| | `tx_data` | `ui_in[7:0]` | input | | `wr_enb` | `uio_in[0]` | input | | `tx` | `uo_out[0]` | output |

RX (receive) | Signal | Pin | Direction | |—|—|—| | `rx` | `uio_in[1]` | input | | `rx_valid` | `uo_out[1]` | output | | `rx_data[1:0]` | `uo_out[3:2]` | output | | `rx_data[7:2]` | `uio_out[7:2]` | output |

Note: keep in mind that `rx_data` is split across two separate output buses

Unused / fixed | Signal | Pin | Value | |—|—|—| | — | `uio_in[7:2]` | unused | | — | `uo_out[7:4]` | tied to 0 | | — | `uio_out[1:0]` | tied to 0 | | `uio_oe` | — | fixed `8'b1111_1100` |

Note: `uio_oe` is fixed since I needed to accommodate 6 continuous output pins for the upper bits of rx_data and 2 input pins for wr_enb and rx.

Baud rate assumptions

My Main Assumptions

- System clock: **49.152 MHz**
- Target baud rate: **9600**
- TX: exactly **5120 clock cycles per bit**
- RX: **16x oversampling**
- RX: exactly **320 clock cycles per tick**
- RX samples near **tick 8**

The Math:

The clock runs at $49,152,000$ Hz (not a round 50 MHz, see note below), which means $49,152,000$ clock cycles per second.

9600 baud means 9600 bits per second.

So the number of clock cycles in one bit is $\frac{49{,}152{,}000}{9600} = 5120$.

For RX, I use **16x oversampling**, meaning there are 16 RX ticks for every bit: $\frac{5120}{16} = 320$.

So the RX counter uses exactly 320 clock cycles per RX tick.

The receiver then counts these 16 RX ticks and samples the actual RX signal around tick 8.

Note: 50,000,000 / 9600 isn't a whole number (5208.33), which causes a small timing error. 49,152,000 was chosen specifically because it divides evenly at both levels: once by 9600 for the bit period, and again by 16 for the RX tick.

This all came out of FPGA testing, where a rounding mismatch (27MHz clock, 9600 baud) caused decode errors until a cleanly dividing rate was used instead.

How to test

RTL simulation

```
cd test
pip install -r requirements.txt
make -B
```

GLS

Requires one-time PDK setup which can be a headache ;-;.

Once the PDK is set up, this is the whole sequence. Run it again every time you change a `.v` file, since hardening makes a new netlist and the copy sitting in `test/` goes stale.

```
export PDK_ROOT=/path/to/IHP-Open-PDK
export PDK=ihp-sg13cmos5l
export LIBRELANE_TAG=3.0.0rc1

# from the repo root
./tt/tt_tool.py --harden --ihp

cd test
make -B                  # RTL simulation

TOP_MODULE=$(cd .. && ./tt/tt_tool.py --print-top-module --ihp)
cp ../runs/wokwi/final/nl/$TOP_MODULE.nl.v gate_level_netlist.v

make -B GATES=yes        # gate-level simulation
```

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	tx_data[0]	tx	wr_enb
1	tx_data[1]	rx_valid	rx
2	tx_data[2]	rx_data[0]	rx_data[2]
3	tx_data[3]	rx_data[1]	rx_data[3]
4	tx_data[4]	—	rx_data[4]
5	tx_data[5]	—	rx_data[5]
6	tx_data[6]	—	rx_data[6]
7	tx_data[7]	—	rx_data[7]

Sine Wave Synthesizer

by **Alberto Beccari**

519 | 12.288 MHz | HDL Project

`github.com/abeccari/tt_um_sine_synthesizer`

"This tile synthesizes sine waves in the audio frequency range by means of a CORDIC engine and pulse-density modulation."

How it works

`tt_um_abeccari_swsynth` is a **CORDIC sine-wave synthesizer**: it plays an equal-tempered musical note as an audio tone. You choose the pitch on the input pins, and the tone comes out three ways — a parallel digital sample bus and two 1-bit pulse-density-modulated (PDM) streams.

Figure 519.1: GDS layout assembled with IHP PDK.

Signal chain:

1. **Note select** — `ui[3:0]` picks a semitone (see the table below) and `ui[7:4]` an octave above A0 = 27.5 Hz. The output frequency is `f = 27.5`

· `2^(note/12)` · `2^octave` Hz, with the octave clamped to ≤ 8 so the tone stays below the Nyquist rate.

2. **NCO** — a frequency map turns the note into a phase increment that drives a 20-bit phase accumulator, stepped at the 48 kHz sample rate (`clk / 256`).
3. **CORDIC** — a rotation-mode CORDIC converts the phase into signed sine and cosine samples (8-bit accurate).
4. **Outputs** —
   - `uo[6:0]` — **7-bit parallel sine**, offset-binary (`0x40` = zero crossing), refreshed once per 48 kHz sample. `uio[0]` (**SAMPLE_EN**) pulses high for one clock at each new sample.
   - `uo[7]` — **PDM_I**, a 1-bit sigma-delta stream of the sine at the full clock rate; low-pass filtered, it reconstructs the analog sine.
   - `uio[7]` — **PDM_Q**, the same for the cosine (90° out of phase with PDM_I) — an I/Q pair.
   - `uio[6]` — **SQR**, a 1-bit square wave at the tone frequency (the sign of the sine); a clean digital output that needs no filter.
   - `uio[5]` — **SAW**, a 1-bit sigma-delta sawtooth at the tone frequency (the raw phase ramp); low-pass filter to reconstruct.
   - `uio[4]` — **NOISE**, a 1-bit pseudo-random bitstream from a maximal-length 20-bit LFSR, advanced one step per 48 kHz sample. Unlike the PDM outputs this is a baseband signal, not a density-coded one — its 0–24 kHz band already is audio-band white noise, so no demodulation is needed. Being a zero-order-hold at 48 kHz it carries spectral images above f_s/2; a gentle low-pass at ≈ 20–24 kHz rejects those ultrasonic images (anti-imaging / reconstruction) for clean band-limited noise and to spare the amplifier ultrasonic switching energy. The sequence is deterministic and repeats every 2^{20}–1 samples (≈ 21.8 s).

Semitone codes on `ui[3:0]` (codes 12–15 have no distinct note and fold back to A):

`ui[3:0]`	0	1	2	3	4	5	6	7	8	9	10	11	12–15
Note	A	A#	B	C	C#	D	D#	E	F	F#	G	G#	A

The clock is 12.288 MHz = 256 × 48 kHz.

How to test

1. **Clock** — drive `clk` at the 12.288 MHz design point; the pitch and 48 kHz sample rate scale with it (`f_s = clk / 256`), and being fully synchronous it also runs correctly at lower rates. Clocks up to 60 MHz are within the process corners (worst-case Fmax ≈ 66 MHz at the slow corner, from

signoff STA); the demo board's on-board clock tops out at 50 MHz, which the design meets with margin, so drive `clk` externally to go higher.

2. **Reset** — hold `rst_n` low for a few clock cycles, then release it high.
3. **Pick a note** — set `ui[7:4]` = octave and `ui[3:0]` = semitone. For example `ui_in = 0x40` → octave 4, note 0 (A) → **440 Hz** (concert A); `ui_in = 0x80` → note A, octave 8 → 7040 Hz.
4. **Monitor** —
   - Put a logic analyzer on `uo[6:0]` (the 7-bit sine) and use `uio[0]` (SAMPLE_EN, 48 kHz) as the sample clock / scope trigger. The value traces a sine centered on `0x40`.
   - Scope `uo[7]` (PDM_I) and `uio[7]` (PDM_Q): fast 1-bit streams whose pulse density follows the sine and cosine.
5. **Change the note** on `ui_in` at any time; the tone follows on the next samples.

External hardware

On the **TinyTapeout demo board** the on-board RP2040 selects the design, supplies the clock, drives `ui_in`, and reads the outputs — so you can set the note from the on-board DIP switches or a MicroPython script and capture the 48 kHz parallel-sine samples in software, with no extra parts. All I/O is also broken out on the Pmod and SIL headers for the analog add-ons below.

- **Hear it:** low-pass filter `uo[7]` (PDM_I) with a simple RC and drive a powered speaker or amplifier. The **Audio Pmod** does exactly this (RC reconstruction filter + amplifier + jack) and takes its input on `uo[7]`, so it plugs straight into the output Pmod.
- **Quadrature (I/Q):** filter `uio[7]` (PDM_Q) the same way for a second channel 90° out of phase — useful for demodulation experiments.
- **Parallel DAC:** feed `uo[6:0]` (offset-binary, midscale `0x40`) into a 7-bit R-2R resistor-ladder DAC on the output Pmod or a SIL header, latched by SAMPLE_EN (`uio[0]`), for an analog staircase sine; add a gentle RC afterward to smooth the sampling images.

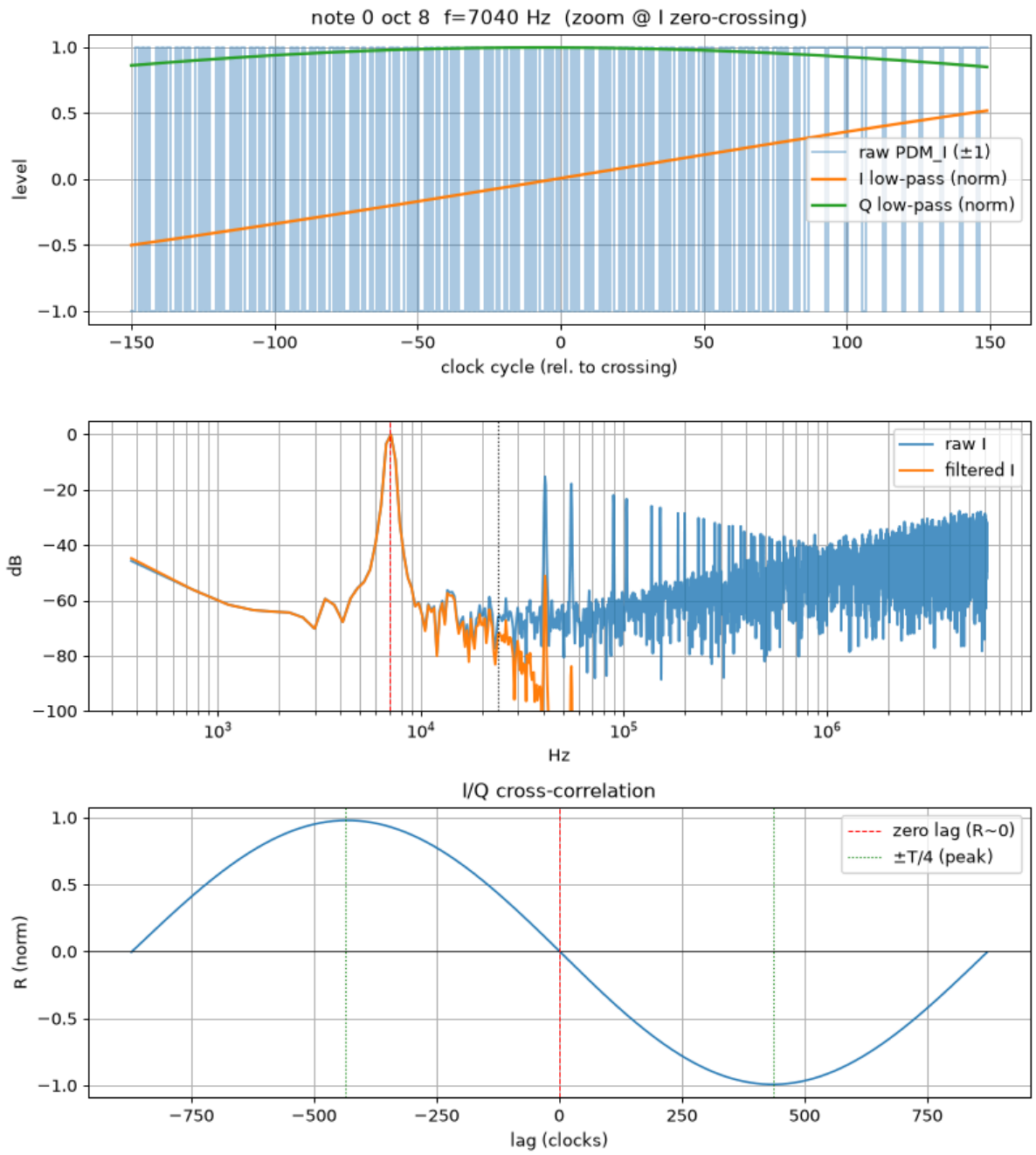


Figure 519.2: PDM I and Q streams low-pass filtered back to sine and cosine (top), their spectra before/after filtering (middle), and the I/Q cross-correlation confirming 90° quadrature (bottom), note A, octave 8 = 7040 Hz

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	NOTE0 (semitone LSB)	SINE0 (sine LSB, offset-binary)	SAMPLE_EN (48 kHz sample strobe)
1	NOTE1	SINE1	unused

#	Input	Output	Bidirectional
2	NOTE2	SINE2	unused
3	NOTE3 (semitone MSB; 0=A..11=G#)	SINE3	unused
4	OCT0 (octave LSB)	SINE4	NOISE (1-bit white noise, maximal-length LFSR PRBS)
5	OCT1	SINE5	SAW (1-bit sigma-delta sawtooth at the tone frequency)
6	OCT2	SINE6 (sine MSB)	SQR (1-bit square wave at the tone frequency)
7	OCT3 (octave MSB; above A0=27.5Hz)	PDM_I (1-bit sigma-delta audio out)	PDM_Q (1-bit sigma-delta audio out - 90 degrees out of phase from PDM_I)

tt_um_adxl362_test

by **Barry Kane**

0521 | 50 MHz | HDL Project

`github.com/red-coffee-spirit/tt_adxl362_test`

"A SPI to UART bridge for an ADXL362"

How it works

Reads an ADXL362 accelerometer over SPI and streams the samples as text on a UART.

On reset it reads DEVID_AD and checks it returns 0xAD, raising uo[1] "dev_ok" if so. POWER_CTL is then written to enter measurement mode. Then @10Hz it reads XDATA_L..ZDATA_H and prints a line as follows over UART @ 115200 baud:

X=00f5 Y=ffee Z=03f1

Values are the 12-bit two's complement readings sign-extended to 16 bits, in hex. It uses the default +/- 2 g range.

SPI is 1MHz mode 0 with the ADXL362's three-byte command header.

ui_in is unused. uo[7:2] are driven low.

How to test

Plug a Pmod ACL2 into the bidirectional PMOD connector, set the clock to 50 MHz, and press reset.

uo[0] follows the recommended UART pinout, so the demo board's RP2040 can bridge it to USB. Open that serial port at 115200 8N1.

Lines then appear ten times a second. Lying flat, one axis reads roughly +/- 1000 (1 g) and the other two sit near zero. Tilting the board should result in a change in the output.

uo[1] "dev_ok" can be connected to an LED or read through tt-commander-app

External hardware

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

Digilent Pmod ACL2 (ADXL362). The UART is on uo[0], the demo board's RP2040 provides the console over USB.

SPI pins follow both the TT recommended SPI pinout and Pmod Interface Type 2, so the ACL2 plugs straight into the bidirectional connector:

tile pin Pmod pin signal uio[0] 1 CS_N uio[1] 2 MOSI uio[2] 3 MISO uio[3] 4 SCK

Project Pinout

Digital Pins

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

4 Channel - 32 Tap Programmable Delay with Delay Locked Loop Calibration

by **han**

523 HDL Project

`github.com/HX2003/ttihp-delay`

> *"Delays a signal by between 0 and up to 31 taps, with each tap providing 0.625 ns of delay, resulting in a maximum total delay of 19.375 ns when using a 50 MHz external reference clock."*

Overview

This project is a 4 Channel - 32 Tap Programmable Delay with Delay Locked Loop Calibration. It delays a digital signal by between 0 and up to 31 taps, with each tap providing 0.625 ns of delay, resulting in a maximum total delay of 19.375 ns when using a 50 MHz external reference clock. Intended applications include data alignment/ adding additional delay to meet setup/hold constraints, which may be especially valuable given the high latency of Tiny Tapeout's mux infrastructure.

How to use

Provide a 50 MHz reference clock to the `ref_clk` pin (not `clk`), as well as any signal you want to delay on the channel input pins. You should observe a delayed version on the corresponding channel output pin. The total delay is a sum of delay caused by the delay line itself (0 to 19.375ns), the delay line multiplexer (0.6ns), and Tiny Tapeout's mux infrastructure (10+ ns).

Top Level Pin	Specific Name	Direction	Width	Description
clk	-	Input	1 bit	Non-free running clock for registers
rst_n	-	Input	1 bit	Active-low reset for registers
ui[3:0]	delay_in[3:0]	Input	4 bits	Signal to be delayed (4 independent channels)
ui[4]	ref_clk	Input	1 bit	50 MHz reference clock for DLL
ui[5]	dll_rst_n	Input	1 bit	Active-low reset for DLL

ui[7:6]	-	Input	2 bits	Unused
uio[2:0]	reg_addr[2:0]	Input	3 bits	Register address value to write
uio[7:3]	reg_val[4:0]	Input	5 bits	Register data value to write
uo[3:0]	delay_out[3:0]	Output	4 bits	Delayed signal (4 independent channels)
uo[4]	ref_clk_out	Output	1 bit	Delayed reference clock
uo[7:5]	-	Output	3 bit	Unused

The initialization procedure for the DLL and the register control can be done independently or simultaneously.

Initialization procedure for DLL:

1. Ensure `dll_rst_n` is low for at least 1ms to be safe.
2. Raise `dll_rst_n` high for at least 5ms to be safe.
3. Keep `dll_rst_n` high. The DLL should be locked at this point.

Initialization procedure for registers:

1. Ensure `rst_n` and `clk` are low for some time.
2. Raise `clk` high.
3. Make `clk` low.
4. Raise and keep `rst_n` high.

Once initialized, to write a delay value to a register, set the 3 bit register address and the 5 bit delay tap amount to the correct value. Wait for a little, then raise `clk` high for some time, then make `clk` low.

Register Address	Register Name
0	Channel 0 Tap Value
1	Channel 1 Tap Value
2	Channel 2 Tap Value
3	Channel 3 Tap Value
4	Reference Clock Tap Value

Note: Reference Clock Tap Value does not affect the DLL or the delay of the other channels.

Overview of architecture

This is mixed-signal design, with the `hx_delay_bank` analog macro block being layout by hand, and the digital register control logic written in verilog. Place-and-route for the digital block, as well as integration of the hand-crafted analog macro block is done automatically.

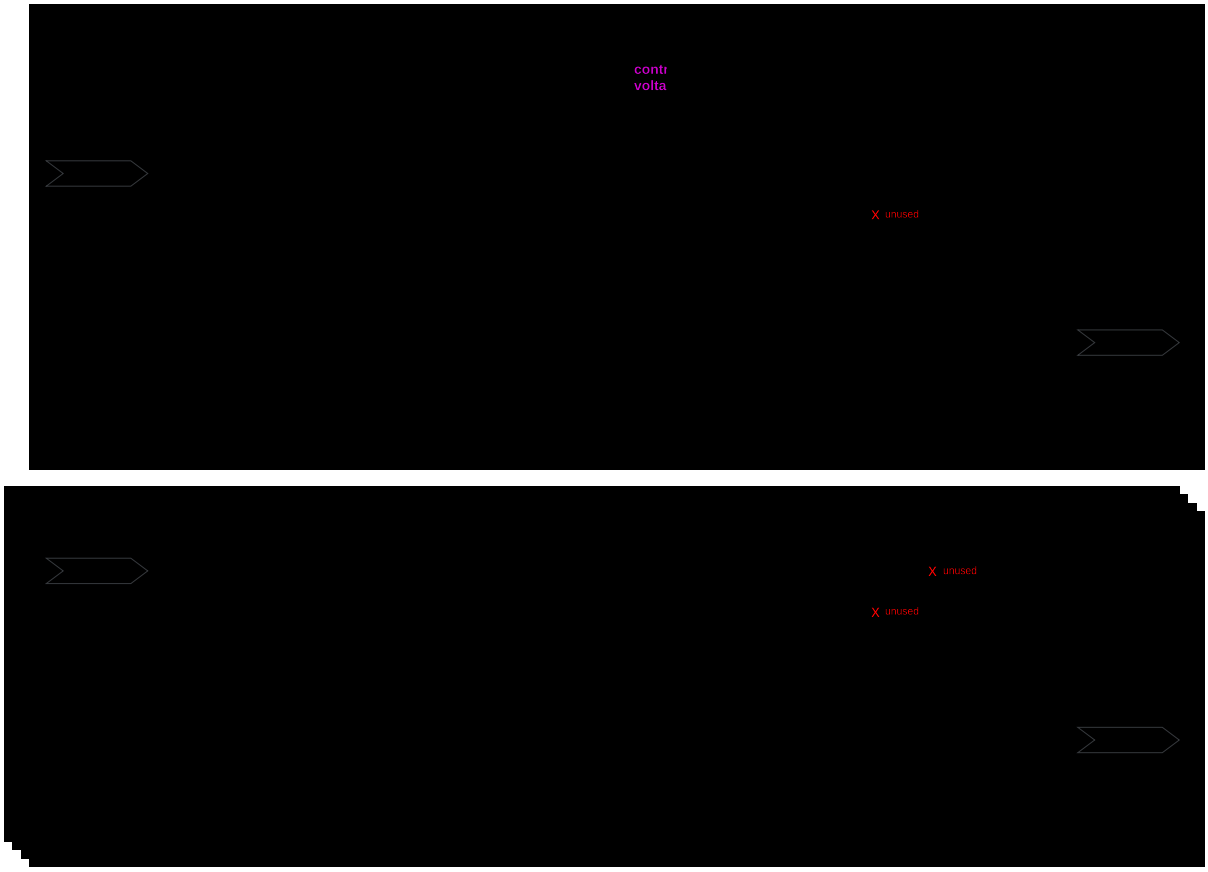


Figure 523.1: Overall Architecture

The `hx_delay_bank` analog macro block consists of five identical delay lines, one of which is used within a Delay-Locked Loop (DLL). The DLL determines the control voltage required, so as to set the total delay of that delay line to exactly one clock period. This same control voltage is used to set the delay of the remaining four delay lines to one clock period as well (assuming the delay lines are well matched). The desired delay is selected by tapping the appropriate point along the delay line using a multiplexer.

Here is the hierarchy for the analog macro block:

```
hx_delay_bank
  hx_delay_line_with_mux (5x total) ✓
    hx_delay_line
      hx_delay_cell (32x total)
      hx_delay_cell_left_endcap
      hx_delay_cell_right_endcap
    hx_mux
```

```
      hx_mux_predecoder
      hx_mux_4to1_leaf (8x total)
      hx_mux_4to1_intermediate (2x total)
      hx_mux_2to1_root ✓
  hx_delay_line_with_mux_dll
    hx_dll_controller
      hx_charge_pump_current_generator
      hx_charge_pump_switch
      hx_loop_capacitor
      hx_phase_detector
    hx_delay_line_with_mux (reused)
```

For more details, it is worthwhile to look into the Xschem schematics provided.

Delay Line Design

Each delay line consists of 32 delay cells. Each delay cell is built around a current-starved inverter, plus 2 additional inverters. The delay can be tuned by controlling the bias voltages (to be explained later). The additional inverters isolate the delay line from external loading effects, such as those from the mux. By making each delay cell inverting, any mismatches between rise and fall propagation delay is effectively canceled from one cell to the next, preventing them from accumulating over the entire delay line, as described in [1].

The delayed signal can then be tapped along the delay line. However, because each delay cell is inverting, to get the correct non-inverted signal, there would be a one-inverter delay difference between odd and even delay cells. This difference can be avoided by tapping every 2 cells, but the small delay difference did not justify the added area/ loss of tap resolution.

The minimum delay of the entire delay line (when `DELAY_VBIASN` is set to VDD), measuring from start to the end, is approximately 10ns when considering parasitics, and at a typical process corner. It was observed that parasitics and process corners play a large impact here. So this sets an upper bound on the max frequency of the reference clock that can be supplied to the DLL, which was why 50MHz was chosen.

Mux Design

To make the delay digitally programmable, a 32 to 1 mux constructed out of a tree of 4:1 and 2:1 transmission gate muxes. 2 to 4 predecoders [2] were necessary to convert select signals into one hot encoded signal for the 4:1 muxes.

Delay Locked Loop (DLL) Design

The DLL consists of a phase detector, charge pump (and current bias generator), loop filter (just a capacitor in parallel at the output of the charge pump), and voltage controlled delay line.

The DLL tries to make the phase error between the `REF_CLK` and its delayed version be 0. However, when using the conventional D flip flop phase detector, the DLL may lock to a multiple of TREF_CLK like 2TREF_CLK , 3TREF_CLK etc in a phenomenon called harmonic locking. Stuck/false locking is the special case when the DLL locks to the minimum delay possible which may occur when Tinitial is less than TREF_CLK.

- The problem of harmonic locking can be avoided by initializing the loop filter capacitor at VDD, and thus achieving minimum delay initially.
- The problem of stuck/false locking can be worked around by designing the phase detector to ignore the first rising edge of the reference clock.

Both of these solutions require the system to start from a known state, and hence `DLL_RESET` is required. Note however, that if the DLL somehow locks to an unwanted state, it cannot recover on its own, more sophisticated designs may be able to handle this. A wide range of design ideas can be found in the literature, see [3, 4, 5, 6, 7, 8, 9].

The current design does not implement a lock detection feature, that is, a feature to detect if the DLL has locked correctly to one clock period of the reference clock. Once again, a wide range of design ideas can be found in the literature, for example [10].

DLL: Phase detector design

The phase detector is a variation of the conventional D flip flop design. To work around the stuck/false locking problem mentioned earlier, an extra flip flop is needed, thus bringing the total number of flip flops to 3. Standard cell library D flip flops (with reset) were used as they are well tested.

DLL: Charge Pump 10uA Current Generator Design

The design is a simple ‘Beta Multiplier’ circuit with a resistor which generates a 10uA current. It attains 10% variation in current from 1.0 to 1.2V for supply voltage VDD (although note the supply voltage is fixed at 1.2V, so no big deal). A somewhat large variation across temperature is expected for this type of design (it is Proportional to Absolute Temperature (PTAT)), I got a 40% variation in current from 25 degree C to 125 degree C, which is bad, but the design should still be workable as DLL are inherently stable, just that the lock time/ bandwidth may vary.

Register Control Logic Design

The register control logic is rudimentary but functional; on the rising edge of system `clk`, the corresponding register is written with the values present on the data pins at that point in time. In the future, the delay line may be incorporated into a larger design which may include more robust interfaces like a standard SPI bus.

Additional notes

I did not add any antenna diodes into the macro itself, but some wires are actually a little long, so it may not be a bad idea to add them.

A real-world application of a delay line can be found in the SPI peripheral of the S32G Vehicle Network Processor [11].

Documentation

Diagrams were drawn using `draw.io`, with some icons from the Draw-io-ECE project.

References

1. Heck, G., Heck, L. S., Singhvi, A., Moreira, M. T., Beerel, P. A., & Calazans, N. L. (2015). Analysis and Optimization of Programmable Delay Elements for 2-Phase Bundled-Data Circuits. 2015 28th International Conference on VLSI Design, 321–326. https://doi.org/10.1109/vlsid.2015.60
2. Implementation and verification of decoder/de-multiplexer and encoder using logic gates. (n.d.). Virtual Labs. https://de-iitr.vlabs.ac.in/exp/decoder-demultiplexer-encoder/theory.html
3. Bae, Woorham. (2016). Delay-Locked Loops: False Locking Issues. https://doi.org/10.13140/RG.2.1.2798.1208
4. Gs, Javed & Khatri, Vishal & Raja, Immanuel & Lenka, Manas & Banerjee, Gaurab. (2014). Differential multi-phase DLL for reconfigurable radio frequency synthesizer. IEEE CONECCT 2014 - 2014 IEEE International Conference on Electronics, Computing and Communication Technologies. 1-5. 10.1109/CONECCT.2014.6740334. https://doi.org/10.1109/CONECCT.2014.6740334
5. Du, Q., Zhuang, J., & Kwasniewski, T. (2006). A Low-Phase noise, Anti-Harmonic programmable DLL frequency multiplier with period error compensation for spur reduction. IEEE Transactions on Circuits and Systems II Analog and Digital Signal Processing, 53(11), 1205–1209. https://doi.org/10.1109/TCSII.2006.883103
6. Chang, H., Lin, J., Yang, C., & Liu, S. (2002). A wide-range delay-locked loop with a fixed latency of one clock cycle. IEEE Journal of Solid-State Circuits, 37(8), 1021–1027. https://doi.org/10.1109/jssc.2002.800922

7. Song, E., Lee, S., Lee, J., Park, J., & Chae, S. (2004). A reset-free anti-harmonic delay-locked loop using a cycle period detector. IEEE Journal of Solid-State Circuits, 39(11), 2055–2061. https://doi.org/10.1109/jssc.2004.835840
8. Design of Delay Locked Loop with Frequency Division Technique. (2016). Proceedings of 2016 the 6th International Workshop on Computer Science and Engineering. https://doi.org/10.18178/wcse.2016.06.095
9. Cho, S., & Cho, Y. (2022). A 1.2 V 0.4 mW 20 200 MHz DLL Based on Phase Detector Measuring the Delay of VCDL. Electronics, 11(15), 2434. https://doi.org/10.3390/electronics11152434
10. Sherafath, M. a. M., & Aziz, Z. a. A. (n.d.). A Novel High Accuracy Digital Lock Detector with False Lock Detection. Scientific & Academic Publishing. http://article.sapub.org/10.5923.j.ac.20130303.06.html
11. SPI peripheral S32G QuadSPI Deep Dive. (n.d.). NXP Semiconductors. https://www.nxp.com/docs/en/application-note/AN13563.pdf

Additional References

Design of a PLL: https://github.com/LegumeEmittingDiode/tt08-tiny-pll/blob/main/docs/info.md

Excellent diagram of false and harmonic locking: https://www.semanticscholar.org/paper/A-2.2-mW-20%E2%80%93135-MHz-False-Lock-Free-DLL-for-Display-Moon-Kong/659ca674d31b180402f3ab807b3fcb037b88fa1e/figure/0

Diagram of harmonic locking: https://www.researchgate.net/figure/Explanation-of-harmonic-lock-and-proposed-shot-pulse-reset-operation_fig10_258383043

Diagram of DLL and delay line: https://www.researchgate.net/figure/Diagram-and-locations-of-IDELAY-and-IDELAYCTRL-modules_fig2_325562388

Good and clear educational series of videos about Delay-Locked Loop by SSCD IIT Kanpur:

- Lecture 3: Delay-locked loop, tunable delay line https://www.youtube.com/watch?v=GZT11SQSxH8
- Lecture 7: Locking in a DLL https://www.youtube.com/watch?v=7_8qSAc9hiE
- Lecture 8: Locking nonidealities in a DLL https://www.youtube.com/watch?v=vZTfCbJRciI
- Lecture 10: Charge pumps - II https://www.youtube.com/watch?v=xUf4xchqS0o

https://blog.eetop.cn/forum.php?mod=viewthread&tid=149458&page=1&mobile=no

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	delay_in[0]	delay_out[0]	reg_addr[0]
1	delay_in[1]	delay_out[1]	reg_addr[1]
2	delay_in[2]	delay_out[2]	reg_addr[2]
3	delay_in[3]	delay_out[3]	reg_val[0]
4	ref_clk	ref_clk_out	reg_val[1]
5	dll_rst_n	—	reg_val[2]
6	—	—	reg_val[3]
7	—	—	reg_val[4]

Proof

by **Kush Shah**

544 | 1 MHz | HDL Project

`github.com/kush1434/proof`

"Fixed-point dot-product engine that estimates the glycemic response of a meal"

How it works

`Proof` is a fixed-point dot-product engine that estimates how much a meal will raise blood sugar. It runs two modes over **one** datapath, selected by `MODE`.

- **Mode A — glycemic load.** A single weighted sum over recipe ingredients, `GL = sum(grams_i * w_i)`, where the host precomputes `w_i = available_carb_per_gram_i * GI_i / 100` and available carbohydrate is total carbohydrate minus dietary fibre. Needs no per-user data, so it works the moment the chip powers on. The chip also reports the standard per-serving category: low ≤ 10, medium 11–19, high ≥ 20.
- **Mode B — personalised response.** A two-layer quantised MLP, `h = ReLU((W1·x + b1) >> s1)` then `y = (W2·h + b2) >> s2`, over meal macronutrients and user context. Shift amounts are powers of two, so requantisation costs a shift rather than a second multiplier.

Every neuron of both layers is the same dot product. Only two things differ: where the activations come from, and what happens to the result. That is why this is one machine with a mode bit rather than two designs.

Weights are streamed in, not stored on chip. That began as an area decision — the parameters do not fit alongside the datapath in one tile — but it is also what makes the design personalisable: same silicon, different patient.

`x` is not buffered on chip either; the host re-streams it for each hidden neuron, which costs 48 bytes instead of 6 and saves 48 flip-flops. At meal timescales bandwidth is free and flip-flops are not. The hidden layer `h` *is* buffered, as a rotating shift register, because only the chip can produce it.

Biases need no bias-specific logic anywhere. In layer 1 they are ordinary extra terms. In layer 2 the chip supplies the constants 127 and 1 after the eight hidden values, so a bias is `v8·127 + v9·1` — exact to the unit for any 15-bit value.

The chip checks its own safety precondition

The monotonicity property below holds if the streamed weights satisfy `W1[j][carb] · W2[j] ≥ 0` for every hidden unit. That is a property of the weights, not of the design — and since the host streams a different weight set per

patient, trusting it is not good enough. Refitting an unconstrained network per person violates the condition in **44 of 44** cases measured on CGMacros.

So the chip checks. Both operands already pass through the pins: the first weight byte of hidden neuron `j` is `W1[j][carb]`, and layer-2 weight byte `k` is `W2[k]`. Their signs ride a register that rotates in lockstep with the hidden-activation register, and a disagreement raises `UNTRUSTED`. A unit is free to oppose carbohydrate *twice* — negative on both sides is a non-negative product — and a zero weight can never trigger it, which is why a non-zero bit is carried alongside each sign.

Everything saturates

The accumulator clamps instead of wrapping and raises the same sticky `UNTRUSTED` flag, and so do the output fields. One pin covers both: a numeric overflow and a void guarantee mean the same thing to a caller — do not act on this output — and the host holds the weights, so it can always tell which. This is not only about producing a sensible number: **a saturating sum is monotone and a wrapping one is not.**

That matters because the design's headline property is

> holding every other input fixed, increasing carbohydrate must never decrease the predicted response.

Saturating sums, arithmetic shifts, ReLU and clamps are each monotone, so the composition is too, provided every hidden unit satisfies `W1[j][carb] · W2[j] ≥ 0`. During development the property held internally but failed at the pins, because the output fields truncated — a one-count rise in carbohydrate could make the reported response fall from 31,293 to −31,209. Truncation wraps. The fields now saturate, and the property survives all the way to what the host reads.

This is not a medical device. It is an educational and research artifact, and every output is an estimate.

How to test

The host drives a byte stream on `DATA_IN`, qualified by `VALID`, with each byte tagged by `IS_WEIGHT`. Do not present a byte while `BUSY` is high; bytes offered then are ignored, not queued.

A neuron is a shift byte, then its terms, with `LAST` on the final one:

```
Mode A / Mode B layer 1   [s|wt] [w0|wt] [a0] [w1|wt] [a1] ... [aN|
last]
Mode B layer 2            [s|wt] [v0|wt] [v1|wt] ...      [v9|
wt,last]
```

The shift byte carries the requantisation shift, so the host sends `s1` for hidden neurons and `s2` for output neurons. In layer 2 there are no activation bytes at all — each weight multiplies the next value the chip supplies: `h[0..7]`, then 127, then 1.

`LAST` asserted on the *shift* byte — otherwise a don't-care — means "this is a new inference", which resets the neuron counter and clears the sticky overflow flag. Mode A ignores it, since every Mode A stream is its own inference.

`UNTRUSTED` marks the result unsafe to act on. `DONE` marks a neuron complete and its result readable. `RD_SEL` selects which byte appears on `RESULT`:

—	`RD_SEL` = 0	`RD_SEL` = 1
Mode A	`GL[7:0]`	`{category[1:0], GL[13:8]}`
Mode B	`value[7:0]`	`value[15:8]`

Eight output pins and a one-bit select give 16 bits, so rather than spend a whole byte on a 2-bit category it is packed above a 14-bit figure. "High" starts at 20, so a 14-bit field is orders of magnitude more than any real meal needs.

`rst_n` always recovers the chip, including from a truncated, stuck or mistagged stream.

The number of *inputs* is not fixed in silicon — a neuron ends when the host says `LAST` — and neither is the number of outputs, since the host simply stops streaming. Only the hidden-layer width is structural, being the depth of the `h` shift register.

There is no host MCU. The protocol is exercised by the cocotb testbench in `test/`, against both the RTL and the post-layout gate-level netlist, and every functional test is compared bit-exactly against an integer reference model. The gate-level run is done twice: once with no timing, and once with the post-route SDF back-annotated at all three corners, so the netlist is exercised with real cell and interconnect delays. Setup and hold are still checked by static timing analysis alone — the simulator implements no timing checks.

The monotonicity argument above is partly machine-checked rather than only reasoned: `formal/` proves the saturating accumulator monotone by k-induction (unbounded), and proves the value the host reads monotone through the whole core for a single-term Mode A inference. Reverting the fix for the truncation defect makes that second proof fail, so the historical bug is reproduced by a solver. Composition across a whole Mode B network is still the hand argument.

`VERIFICATION.md` records what is verified and what is not; `BUGS.md` records every defect found, including the ones found in the testbench itself.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA_IN[0]	RESULT[0]	VALID (in)
1	DATA_IN[1]	RESULT[1]	IS_WEIGHT (in)
2	DATA_IN[2]	RESULT[2]	LAST (in)
3	DATA_IN[3]	RESULT[3]	MODE (in)
4	DATA_IN[4]	RESULT[4]	RD_SEL (in)
5	DATA_IN[5]	RESULT[5]	DONE (out)
6	DATA_IN[6]	RESULT[6]	UNTRUSTED (out)
7	DATA_IN[7]	RESULT[7]	BUSY (out)

DNA_Accel

by **Garv R Jain**, **Suhani Das** & **Dr. Vishal Gupta**

0545 | 50 MHz | HDL Project

`github.com/Dragon1864/tt_DNA_Accel`

"SPI-driven hardware accelerator for transition/transversion-aware DNA sequence comparison"

How it works

DNA_Accel is a hardware-accelerated DNA sequence comparator, controlled entirely over SPI. The host loads an 8-base pattern and an 8-base DNA window (2 bits per base, packed into 16-bit registers), and the chip scores their similarity in hardware: each of the 8 base positions is compared in parallel and classified as an exact match, a transition (the biologically "softer" mismatch, A<->G or C<->T), or a transversion (any other mismatch). Exact matches score 2 points, transitions score 1, transversions score 0, giving a total similarity of 0-16.

Because a single similarity number is ambiguous (e.g. 14 could mean 7 exact + 1 transversion, or 6 exact + 2 transitions), the design also computes unambiguous hardware counts of exact/transition/transversion bases (each 0-8, always summing to 8), plus a per-base mismatch mask and a packed vector of raw 2-bit scores, so the host can tell exactly which bases differ and how.

Internally the design is layered: `spi_lite` is a generic byte-level SPI slave with no protocol awareness; `bioaccel_spi_decoder` implements the BioAccel command/register protocol (write/read/stream, register addressing); `dna_accelerator_top` is pure comparison logic (registers -> comparator -> similarity/mutation scoring) with no SPI awareness at all. The whole design runs on a single clock: the host's own SPI clock (SCK), wired to the chip's dedicated `clk` pin – there is no separate on-chip system clock.

How to test

Drive the SPI clock (SCK) on the dedicated `clk` pin, and control `cs_n` and `mosi` on `ui[0]` and `ui[1]`; read `miso` back on `uo[0]`. All communication uses SPI mode 0 (idle low, data driven on the falling edge, sampled on the rising edge), MSB first.

Write (3 bytes): `0x01` (CMD_WRITE), `address`, `data`

Read (4 bytes): `0x02` (CMD_READ), `address`, `0x00` (spacer), then the response byte is clocked out on `miso` during this 4th byte.

Register map:

Addr	Register	Access
0x00	Pattern low byte	Write
0x01	Pattern high byte (latches load)	Write
0x02	DNA low byte	Write
0x03	DNA high byte (latches load)	Write
0x04	Similarity score (0-16)	Read
0x05	Match flag (1 = perfect score)	Read
0x06	Position, low byte	Read
0x07	Position, high byte	Read
0x08	Mismatch vector, low byte	Read
0x09	Mismatch vector, high byte	Read
0x0A	Mismatch mask (1 bit/base)	Read
0x0B	Exact-match count (0-8)	Read
0x0C	Transition count (0-8)	Read
0x0D	Transversion count (0-8)	Read

To sanity-check the design: write the pattern and DNA registers with identical 16-bit values (e.g. both `0x0000`), then read address `0x04` – similarity should read back `16` and address `0x05` should read `1` (perfect match). Change a few bits in the DNA registers and re-read; similarity should drop, the mismatch mask (`0x0A`) should show which bases changed, and the exact/transition/transversion counts (`0x0B`-`0x0D`) should always sum to 8.

External hardware

None – the design only needs a host capable of driving an SPI-like clock/data/chip-select interface (e.g. a Raspberry Pi, ESP32, Arduino, or PC-based SPI adapter). No PMODs, displays, or other peripherals are required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	cs_n (active-low SPI chip select)	miso (SPI data out)	—
1	mosi (SPI data in)	—	—
2	—	—	—
3	—	—	—
4	—	—	—

#	Input	Output	Bidirectional
5	—	—	—
6	—	—	—
7	—	—	—

nanoPIO

by **catalinlazar**

546 | 10 MHz | HDL Project

`github.com/catalinlazar/tt_um_catalinlazar_nanopio`

> *"A tiny bit-banged programmable I/O engine (RP2040 PIO-inspired) with a reprogrammable 16-word instruction store, used to emulate UART/SPI/I2C timing."*

Instruction word (13 bits, 16-word program store)

```
[12:10] opcode
[9:7]   delay    (0–7 extra stall cycles applied after the
instruction's effect)
[6:0]   operand  (meaning depends on opcode)
```

opcode (3b)	mnemonic	operand layout	effect
000	JMP	`[6:5]=cond` `[3:0]=addr`	`PC <= addr` if cond true, else `PC+1`
001	WAIT	`[6]=pol [2:0]=pin idx`	stall (no PC advance) until `ui_in[idx] == pol`
010	IN	unused	`X <= ui_in`
011	OUT	`[6]=dest (0=uo_out,1=uio_out)`	`dest <= X`
100	SET	`[6:0]=imm7`	`X <= {1'b0, imm7}`
101/110/111	reserved	—	NOP (`PC+1`)

JMP conditions

cond (2b)	name	taken when
00	ALWAYS	always
01	X==0 (XZ)	`X == 0`
10	X!=0 (XNZ)	`X != 0`
11	DEC	`X != 0` (checked before decrement); `X` is then decremented regardless of branch direction

`DEC` is the classic PIO loop primitive: `SET X, N` then a body ending in `JMP DEC, body_start` counts down N times without a separate compare instruction.

Delay field

Every instruction (except WAIT while its condition is unmet) can attach 0-7 extra stall cycles after it executes. This lets protocol timing — a UART bit period, an SPI/I2C clock half-period — be encoded directly into the program instead of burning instruction words on busy-loops. This mirrors how RP2040 PIO folds `[delay]` into every instruction rather than giving it its own opcode, which matters a lot when the whole program store is only 16 words.

Registers

- **PC** — 4 bits, indexes the 16-word instruction memory
- **X** — 8-bit scratch/loop-counter register. No separate ISR/OSR shift registers like real PIO has: `IN` writes straight into `X`, `OUT` drives straight out of `X`. This is the single biggest area saving relative to RP2040 PIO, and it's a reasonable trade since TT's `ui_in`/`uo_out` are already 8 bits wide — there's no 32-bit-wide autopush/autopull to emulate.

There is no second (`Y`) counter in v0.1; nested loops (e.g. an outer bit-count loop and an inner sample-delay loop) have to share `X` or be restructured using the `delay` field instead of an inner loop.

Pinout

Pin	Role while running	Role while `LOAD_EN` (ui_in[7]) is high
`ui_in[4:0]`	WAIT/IN pin sources	ignored
`ui_in[5]`	—	`LOAD_DATA`
`ui_in[6]`	—	`LOAD_CLK`
`ui_in[7]`	—	`LOAD_EN`
`uo_out[7:0]`	`OUT PINS` target	—
`uio[7:0]`	`OUT UIO` target (output-only in v1)	—

Reprogramming (bit-bang loader)

The instruction memory is built from per-word latches gated by a shared write-address decoder (not flip-flops, and not a synthesized ROM), so it can be rewritten after fabrication. Latches roughly halve the per-bit storage cost vs. flip-flops on this PDK, which is what lets the 16-word store fit in a 1x1 tile. To load a program:

1. Hold `ui_in[7]` (`LOAD_EN`) high. This halts the core (PC forced to 0, `X` cleared) so it can't execute half-written instructions.

2. For each instruction word, shift a 17-bit frame MSB-first into `ui_in[5]` (`LOAD_DATA`), toggling `ui_in[6]` (`LOAD_CLK`) once per bit (rising-edge sampled, internally double-flopped for metastability):
   - bits `[16:13]` — 4-bit address
   - bits `[12:0]` — 13-bit instruction
3. Frames can be sent in any order and re-sent to patch a single word; `LOAD_EN` can stay high across many frames.
4. Drop `LOAD_EN` low. The core resumes execution from address 0.

`tools/nanopio_asm.py --frames` prints the exact bit pattern for each frame of an assembled program, in the order described above.

What's simplified vs. real RP2040 PIO (and why)

- **8-bit X instead of 32-bit ISR/OSR** — TT's pin buses are 8 bits wide, so a wider shift register would mostly be padding.
- **One scratch register, no Y** — halves the register file; loop nesting falls back to the per-instruction delay field where possible.
- **No autopush/autopull, no FIFO to the "system" side** — there's no CPU on the other end of this chip; `IN`/`OUT` talk directly to pins.
- **`uio` is output-only in v1** — avoids the extra pin-direction control logic and instruction bits a bidirectional `uio` would need; revisit if area allows.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PIO in[0] / WAIT PIN 0 source	PIO out[0]	PIO out[8] (uio, output only)
1	PIO in[1] / WAIT PIN 1 source	PIO out[1]	PIO out[9] (uio, output only)
2	PIO in[2] / WAIT PIN 2 source	PIO out[2]	PIO out[10] (uio, output only)
3	PIO in[3] / WAIT PIN 3 source	PIO out[3]	PIO out[11] (uio, output only)
4	PIO in[4] / WAIT PIN 4 source	PIO out[4]	PIO out[12] (uio, output only)
5	LOAD_DATA (reprogramming serial data in)	PIO out[5]	PIO out[13] (uio, output only)
6	LOAD_CLK (reprogramming bit clock)	PIO out[6]	PIO out[14] (uio, output only)
7	LOAD_EN (1 = reprogramming mode, core halted)	PIO out[7]	PIO out[15] (uio, output only)

Colegio de Muntinlupa DVD-like Display

by **CDM**

`0547` `25.175 MHz` `HDL Project`

`github.com/alexandercoabad/CDM_display`

"*Colegio de Muntinlupa DVD-like Display*"

How it works

Colegio de Muntinlupa DVD-like Display

How to test

Colegio de Muntinlupa DVD-like Display

External hardware

VGA PMOD

Project Pinout

Digital Pins

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

Circuitli C061618G2TR

by **circuitli**

548 HDL Project

`github.com/circuitli/tt_um_c061618g2tr`

> *"A triple redundant clockless MMU module with an integrated anti-glitch filter network for 8-bit processors with a 16-bit address bus."*

How it works

This project is a Memory Management Unit (MMU). It handles memory mapping, peripheral address decoding, and operating system banking across the computer's memory map.

- **Address Decoding:** The circuit accepts high-order CPU address lines (A11 to A15) alongside primary system configuration flags to determine which physical hardware target should be active during a memory read/write cycle.
- **Banking Logic & PORTB:** It actively monitors control inputs. This includes decoding signals to dynamically switch the internal 16KB Operating System ROM, the BASIC ROM, and the Self-Test ROM in and out of the memory map, freeing up underlying DRAM when disabled.
- **Peripheral Mapping:** The internal combinatorial decoding grid maps out active-low chip select signals for custom sub-systems and the master RAM network.

This design acts as a purely combinatorial logic array.

How to test

This project is currently validated using an automated simulation environment (Cocotb or HDL testbench). You can run the testbench and inspect the generated waveform file (VCD) using the following verification steps:

1. **Initialize Simulation:** Run the test suite using `make test'.
2. **Verify Basic OS ROM Mapping:**
   - In the testbench stimulus, drive the address bus inputs `A[15:11]` to `5'b11111` (representing memory space `$F800–$FFFF`).
   - Assert the `REN` (ROM Enable) control input high.
   - Inspect the waveform viewer to verify that the OS Chip Select output (OS_L) and CI_L drop to 0.
3. **Verify BASIC Banking Logic:**
   - Drive the address inputs to `5'b10101` (representing memory space `$A000–$BFFF`).

- Toggle the `BE_L` simulation signal to 0.
- Assert that BASIC_L falls to 0. Then, drive BE_L to 1 and verify the output switches back to exposing the underlying system memory by resetting CI_L` to 1.

4. **I/O Device Matrix Test:**
   - Force the address lines to match the hex-equivalent binary for `$D300`.
   - Check the simulation logs or waveform to confirm that only the IO_L output line drops to logic 0.

External hardware

None for now.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	A11	REN	S5_L
1	A12	REF_L	BASIC_L
2	A13	MPD_L	OS_L
3	A14	BE_L	CI_L
4	A15	—	IO_L
5	MAP_L	TRIGGER_OUT_L	S4_L
6	RD4	FLG_IN_L	FLG_L
7	RD5	—	—

Morse Tree LED Decoder

by **Matthias Wegmann**

`549` `10 MHz` `HDL Project`

`github.com/mattizen/morse-tree`

"Reads a Morse key and lights the matching node of the Morse code tree on an 8x8 LED matrix"

How it works

The design listens to a Morse key on `ui[0]` and walks the Morse code binary tree: every **dot** goes to the left child, every **dash** to the right child. The current tree node is shown on an 8x8 LED matrix, one LED at a time. If you glue the LEDs onto a poster of the Morse tree (with the letter written next to each LED), the light wanders down the tree while you key a character and stops on the decoded letter.

Tree nodes are numbered like a binary heap: the root is 1 and each symbol appends one bit (`node = node * 2 + symbol`, dot = 0, dash = 1). The 6-bit node number selects the LED: `node[5:3]` is the row (`uo_out`, one-hot, active high) and `node[2:0]` is the column (`uio_out`, one-hot, active low). Node 1 (root) is the "ready" LED, node 0 is an error LED (more than five symbols).

Timing is measured in units of the dot length. The dot length is `2^SPEED` base ticks, one base tick being 2^15 clock cycles (3.3 ms at 10 MHz):

SPEED (`ui[3:1]`)	0	1	2	3	4	5	6	7
dot length @ 10 MHz	3.3 ms	6.5 ms	13 ms	26 ms	52 ms	105 ms	210 ms	420 ms

- key pressed **shorter than 2 units** = dot, **2 units or longer** = dash
- key released for **2 units** = character complete (the LED stays on the letter)
- key released for **8 units** = back to the root (unless `HOLD` = 1, then the letter stays until the next key press)

Hand keying works well with SPEED 5 or 6. SPEED 0 and 1 are only meant for machine keying, because the 3 ms debounce filter would swallow such short dots. Other clock frequencies scale all times accordingly (e.g. 20 MHz halves every value).

The key input is synchronised and debounced (stable for about 3 ms). `KEY_INV` (`ui[7]`) inverts the key, so a button to GND with a pull-up resistor can be used directly.

`RAW` (`ui[5]`) switches `uo_out` from the row one-hot code to the plain node number: `uo[5:0]` = node, `uo[6]` = character complete, `uo[7]` = debounced key. This is handy for testing from the demo board or for driving an external decoder instead of the matrix.

LED positions

Node	Char	Code	Row (uo)	Col (uio)
0	(error)	more than 5 symbols	0	0
1	(root)	idle / keying	0	1
2	E	.	0	2
3	T	–	0	3
4	I	..	0	4
5	A	.–	0	5
6	N	–.	0	6
7	M	––	0	7
8	S	...	1	0
9	U	..–	1	1
10	R	.–.	1	2
11	W	.––	1	3
12	D	–..	1	4
13	K	–.–	1	5
14	G	––.	1	6
15	O	–––	1	7
16	H	....	2	0
17	V	...–	2	1
18	F	..–.	2	2
19	Ü	..––	2	3
20	L	.–..	2	4
21	Ä	.–.–	2	5
22	P	.––.	2	6
23	J	.–––	2	7
24	B	–...	3	0
25	X	–..–	3	1

26	C	`-.-.`	3	2
27	Y	`-.--`	3	3
28	Z	`--..`	3	4
29	Q	`--.-`	3	5
30	Ö	`---.`	3	6
31	CH	`----`	3	7
32	5	`.....`	4	0
33	4	`....-`	4	1
35	3	`...--`	4	3
39	2	`..---`	4	7
47	1	`.----`	5	7
48	6	`-....`	6	0
56	7	`--...`	7	0
60	8	`---..`	7	4
62	9	`----.`	7	6
63	0	`-----`	7	7

All other nodes 32..63 are the remaining five-symbol codes (punctuation, prosigns, accented letters); their node number is simply `1` followed by the five symbol bits.

How to test

1. Apply a 10 MHz clock and release reset. With `ui_in = 0` the root LED (row 0, column 1) is lit: `uo_out = 0x01`, `uio_out = 0xFD`.
2. Set the speed, e.g. `ui[3:1] = 5` (dot = 105 ms), and connect a push button to `ui[0]` (or set `ui[7] = 1` for an active-low button).
3. Key `.–` (short, long): after the dot the E LED lights (row 0, column 2), after the dash the A LED (row 0, column 5). Pause for two units and the A stays lit; after eight units the root LED comes back.
4. For an automated check set `ui[5] = 1` (RAW mode) and read the node number on `uo[5:0]`; `uo[6]` tells you that a character is complete.

The cocotb test in `test/test.py` keys several characters (S, O, A, H, 5, 1, …), checks the node after every symbol, the character-complete flag, the word gap, the HOLD mode, the error node, the matrix outputs and the inverted key input.

External hardware

- A push button or Morse key on `ui[0]`.

- Up to 64 LEDs wired as an 8x8 matrix: LED anodes to the row lines `uo[0..7]`, cathodes to the column lines `uio[0..7]`, one series resistor (about 470 Ω) per column line. Because only one LED is on at any time, no multiplexing or driver ICs are needed; the chip pins can only supply a few mA, so use efficient (low-current) LEDs or add transistor drivers for brighter ones.
- Instead of a matrix you can use RAW mode and feed the node number into external decoders.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	KEY (Morse key, 1 = pressed)	ROW0 / NODE0	COL0 (active low)
1	SPEED[0]	ROW1 / NODE1	COL1 (active low)
2	SPEED[1]	ROW2 / NODE2	COL2 (active low)
3	SPEED[2]	ROW3 / NODE3	COL3 (active low)
4	HOLD (keep last character until next key press)	ROW4 / NODE4	COL4 (active low)
5	RAW (uo_out = {KEY, DONE, NODE[5:0]})	ROW5 / NODE5	COL5 (active low)
6	—	ROW6 / DONE	COL6 (active low)
7	KEY_INV (1 = key is active low)	ROW7 / KEY	COL7 (active low)

Tiny_Divider

by **OzelHD**

0550 10 Hz Wokwi Project

github.com/OzelHD/Tiny_Divider

wokwi.com/projects/475490677474407425

"*Divide two numbers*"

How it works

The Input is via the 8 switches, where [0 to 5] are binary inputs for both nominator and denominator and [6] is a hold, [7] is the toggle to toggle nominator/ Denominator.

How to test

(Will be explained later)

External hardware

N/A

More Info will follow. - OzelHD

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Binary 1	Display	—
1	Binary 2	Display	—
2	Binary 3	Display	—
3	Binary 4	Display	—
4	Binary 5	Display	—
5	Binary 6	Display	—
6	Keep	Display	—
7	Nom/Denom Toggle	Display	—

tt_um_vga_hypno_spiral

by **Joshua Azriel Cuarteron**

0551 | 25.175 MHz | HDL Project

`github.com/Jousharo15/Project-Sample-PH-Bootcamp-2026-`

"*Black & White Hypnotizing Background*"

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	Reverse flow direction	R1	—
1	Tighter rings	G1	—
2	Twist select bit 0	B1	—
3	Twist select bit 1 (00=4 arms, 01=1 arm, 10=8 arms, 11=circles)	VSync	—
4	Invert black/white	R0	—
5	Soft grayscale shading	G0	—
6	—	B0	—
7	—	HSync	—

TinyAnalogExperiments

by **Hackin7**

`0552` `50 MHz` `HDL Project`

`github.com/Hackin7/ttihp26b-hackin7-analog-experiments`

"Dual ring oscillators (100/500 MHz) and 64-bit counter"

How it works

Hierarchical Tiny Tapeout top with analog ring leaves and a digital counter:

- 100 MHz transistor ring (`ring_oscillator` GDS macro).
- 500 MHz transistor ring (`ring_oscillator_500mhz` GDS macro).
- 64-bit digital counter (`digital_counter` RTL). `ui_in[0]` enables counting, `ui_in[4:2]` selects which counter byte drives `uo_out`.

Clock select (change only while `rst_n` is low):

`ui_in[1]`	`ui_in[5]`	Counter clock
0	x	Tiny Tapeout `clk`
1	0	100 MHz ring
1	1	500 MHz ring

Ring-clock modes are not timing qualified.

How to add more analog or digital leaves: see `docs/hierarchy.md`.

How to test

Hold reset, set `ui_in[1]` / `ui_in[5]` for the clock source, release reset, set `ui_in[0]` to enable the counter, and read `uo_out` while sweeping `ui_in[4:2]`.

External hardware

None beyond the Tiny Tapeout board clock and GPIO.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	counter_enable	counter_byte[0]	—
1	clock_select_ring	counter_byte[1]	—
2	counter_byte_select[0]	counter_byte[2]	—
3	counter_byte_select[1]	counter_byte[3]	—

#	Input	Output	Bidirectional
4	counter_byte_select[2]	counter_byte[4]	—
5	clock_select_500mhz	counter_byte[5]	—
6	—	counter_byte[6]	—
7	—	counter_byte[7]	—

3x3 Clock Rate Streaming Input Convolution Engine

by **Yahya Numan Incirkus**

`553` `500 kHz` `HDL Project`

`github.com/Cicikush1/CNV3x3`

"3x3 linear convolution through a singular MAC unit, 8-bit pixel input 6-bit weights"

How it works

Standard MAC operation for linear convolution ## How to test Feed 9 Pixels sequentially, 8 bits each with tuser pin ran high when 1st of each 9 pixel array is fed. wghupdates switches the pixel feed line into weight feed and rndgset sets the shifting amount to accomodate different filter structures. ## External hardware none

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA0 (pixel / weight / config)	RESULT0	S_TVALID (in)
1	DATA1	RESULT1	S_TUSER window start (in)
2	DATA2	RESULT2	S_TLAST window end (in)
3	DATA3	RESULT3	WGHUPD weight load (in)
4	DATA4	RESULT4	RNDGSET config load (in)
5	DATA5	RESULT5	M_TREADY (in)
6	DATA6	RESULT6	M_TVALID (out)
7	DATA7	RESULT7	S_TREADY (out)

Hamming(7,4) encoder/decoder (IEEE)

by **Esaú Quispe Checca**

`0554` `100 kHz` `HDL Project`

`github.com/204326-collab/triada-ceg-hamming74`

> *"Encodes a 4-bit word into a 7-bit Hamming(7,4) codeword, injects a single-bit error, then detects and corrects it — Equipo Triada CEG"*

How it works

The design implements a Hamming(7,4) encoder, a simulated single-bit channel error, and a decoder/corrector, all in one combinational block registered on `clk`:

1. **Encoder**: `ui_in[3:0]` is treated as a 4-bit data word (`d1..d4`). Three parity bits (`p1`, `p2`, `p3`) are computed and interleaved with the data to form the 7-bit codeword using the standard 1-indexed layout `1=p1 2=p2 3=d1 4=p3 5=d2 6=d3 7=d4`.
2. **Error injection**: `ui_in[6:4]` selects a codeword bit position (1-7) to flip, simulating a transmission error. `0` means no error is injected.
3. **Decoder**: the three parity checks are recomputed on the (possibly corrupted) codeword to produce a 3-bit syndrome. A non-zero syndrome points directly at the corrupted bit position, which is flipped back to recover the original codeword.
4. **Outputs**: `uo_out[3:0]` is the corrected 4-bit data (equal to the original input for any single injected error), `uo_out[4]` is an error flag, and `uio_out[6:0]` exposes the raw transmitted/received codeword bit-by-bit (`uio_out[i]` = codeword position `i+1`) for inspection on a logic analyzer or in the testbench waveform.

Everything is registered on `clk` with a synchronous active-low reset (`rst_n`), so the outputs reflect the inputs one clock cycle after they are applied.

How to test

Drive `ui_in[3:0]` with the 4-bit word to encode and `ui_in[6:4]` with the bit position (1-7) to corrupt (or `0` for a clean channel), wait one clock cycle, then check:

- `uo_out[3:0]` equals the original `ui_in[3:0]` (Hamming(7,4) always corrects a single-bit error).
- `uo_out[4]` is `1` whenever an error was injected (`ui_in[6:4] != 0`) and `0` otherwise.

- `uio_out[6:0]` shows the corrupted codeword actually “on the wire” before correction.

`test/test.py` sweeps all 16 data words against all 8 error positions (no error + each of the 7 possible single-bit flips) and asserts correct recovery in every case.

External hardware

None - this project only exercises the dedicated and bidirectional I/O pins directly.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	data_in[0]	data_out[0]	codeword[1] (p1)
1	data_in[1]	data_out[1]	codeword[2] (p2)
2	data_in[2]	data_out[2]	codeword[3] (d1)
3	data_in[3]	data_out[3]	codeword[4] (p3)
4	err_pos[0]	error_flag	codeword[5] (d2)
5	err_pos[1]	—	codeword[6] (d3)
6	err_pos[2]	—	codeword[7] (d4)
7	—	—	—

Fixed-ROM XOR Stream Engine

by **Owen Deng**

0555 50 MHz HDL Project

github.com/qd39l/ttihp26b-xor-stream

"*Ready/valid byte-stream XOR engine backed by a fixed 160-byte ROM*"

How it works

The Fixed-ROM XOR Stream Engine stores 160 fixed bytes in combinational standard-cell ROM. After START, a host supplies one same-position input byte at a time. The chip XORs that byte with the addressed ROM byte and returns the result through a ready/valid output register.

This keeps the ASIC deliberately small: it stores only the fixed ROM and one output byte at a time. Input acquisition, formatting, and display are left to external hardware. The design targets one TTIHP26b tile.

This is a one-time-pad design, not authenticated encryption. The production key must be uniformly random, exactly the ciphertext length, kept private, and never reused. A wrong key still produces output; the chip cannot detect it.

The ROM contents are public. Any application that relies on one-time-pad confidentiality must keep its external key uniformly random, private, exactly the ROM length, and never reuse it.

The physical layout also includes a passive TopMetal1 artwork macro. It is electrically disconnected and does not affect the digital interface.

How to test

1. Reset the design with `rst_n=0`.
2. Pulse `uio_in[1]` (START).
3. When `uio_out[4]` (KEY_READY) is high, place the next key byte on `ui_in[7:0]` and pulse `uio_in[0]` (KEY_VALID).
4. Keep `uio_in[2]` (OUT_READY) high and capture `uo_out[7:0]` whenever `uio_out[6]` (OUT_VALID) is high.
5. After the last byte transfers, `uio_out[7]` (DONE) stays high.

`uio_in[3]` (ABORT) synchronously stops a session and clears progress.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	KEY_DATA[0]	OUT_DATA[0]	KEY_VALID
1	KEY_DATA[1]	OUT_DATA[1]	START
2	KEY_DATA[2]	OUT_DATA[2]	OUT_READY
3	KEY_DATA[3]	OUT_DATA[3]	ABORT
4	KEY_DATA[4]	OUT_DATA[4]	KEY_READY
5	KEY_DATA[5]	OUT_DATA[5]	BUSY
6	KEY_DATA[6]	OUT_DATA[6]	OUT_VALID
7	KEY_DATA[7]	OUT_DATA[7]	DONE

launch deployment timer

by **Eric Pearson**

0577 48 MHz HDL Project

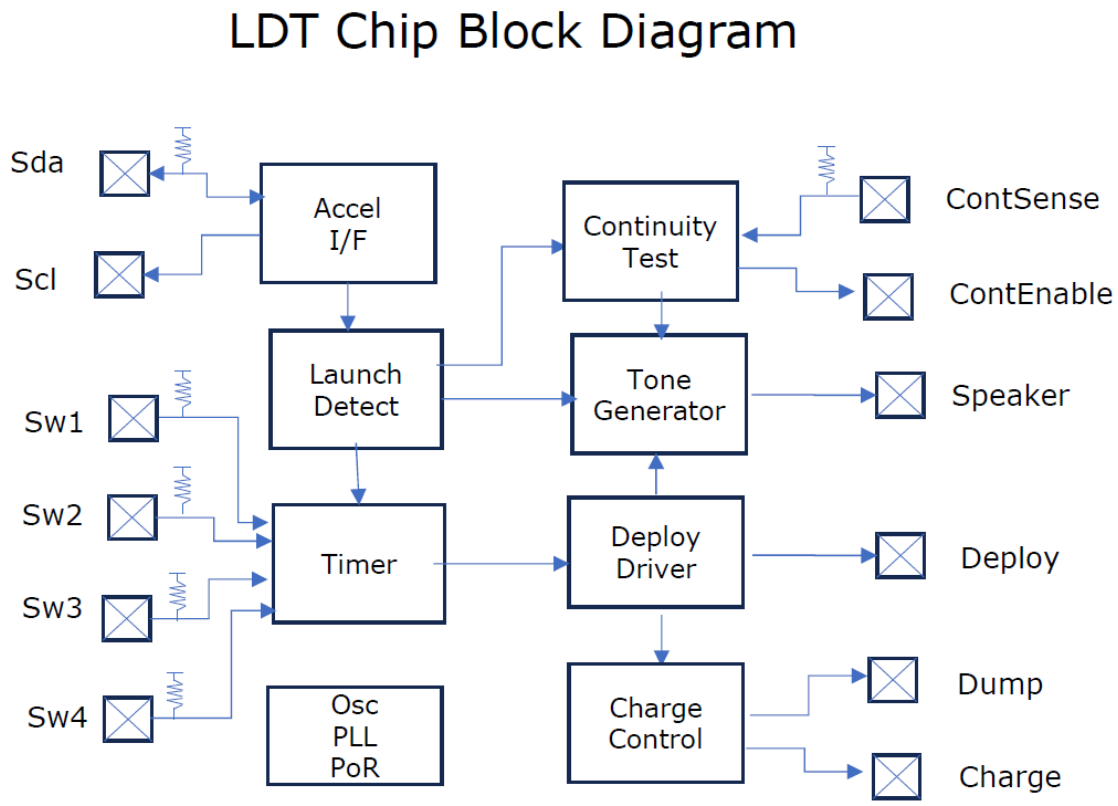
github.com/ericpearson1313/tt_deployment_timer

"Accelerometer triggered timer with deployment sequencing"

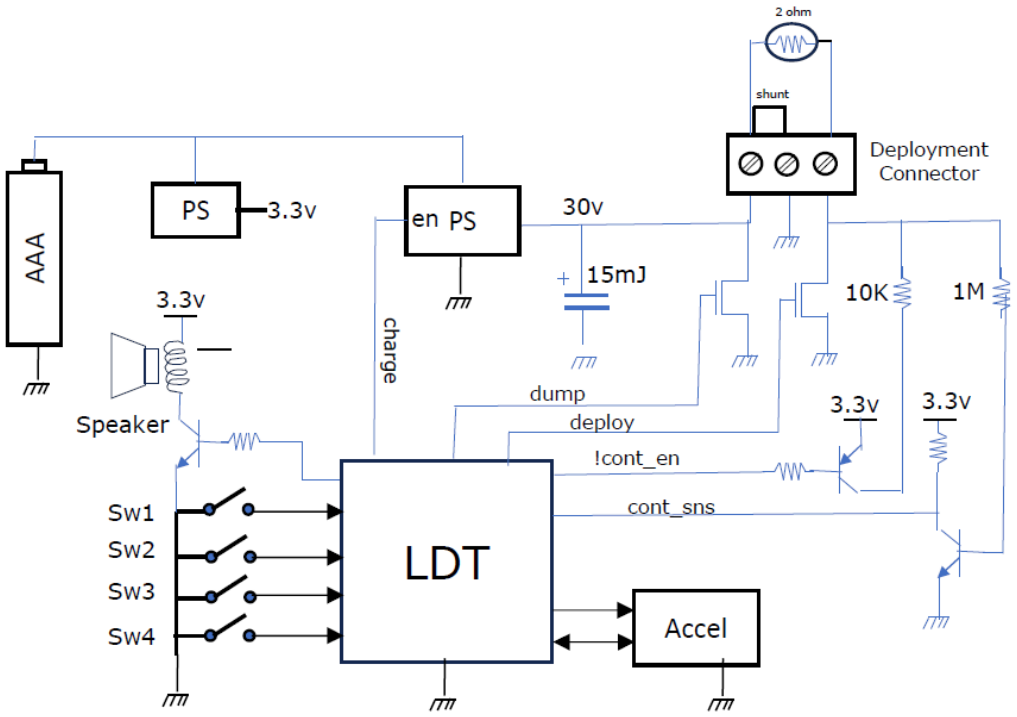
How it works

This TinyTapeout design implements a complete launch-detect deployment timer (LDT) for model rocket recovery. It polls an external I²C accelerometer, then waits to for a launch accelleration (sustained >2 g), it executes a fixed sequence: a programmable delay, a pre-charge window, a one-tick deployment pulse, and follows with a post-deployment warble tone on a piezo output. Prior to launch continuity checking is done at 1Hz with the piezo giving single beep when ok, and double beep if continuity test failed. Once the launch sequence begins all further inputs are ignored, making the device a single-shot, deterministic controller that fits comfortably within a 1×1 TinyTapeout tile.

This chip starts with a datasheet where the chip I/O is nailed down and expanded.



The I/O is expanded internally into a chip block diagram.



And the I/O is expanded out into the system diagram so the chip context is clear.

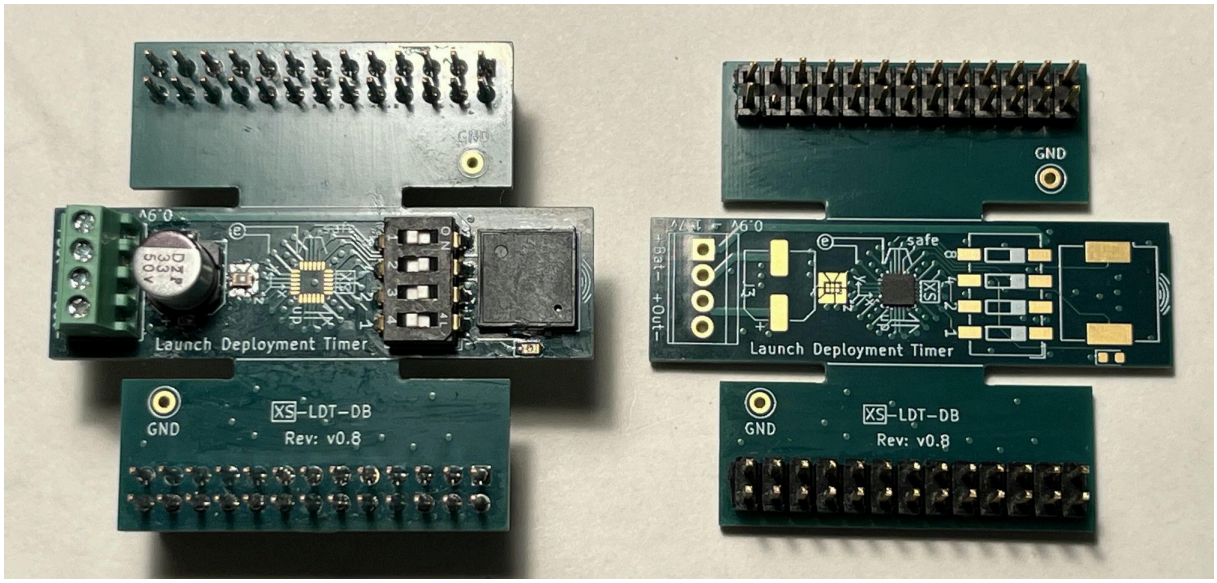


Photo of dev board. Left board populated except LDT chip, right with LDT chip (SLG47910V with TT logic). The emulation header breaks out all the chip I/O and allows two boards for testing.

How to test

A development board was designed, built, and tested. Along with a Max10 I/o Monitoring fpga, the TT pmods can be connected directly to emulation headers on the development board. Also before endangering the hardware a Max10 chip tester (using the sys model used in verificaiton) can be used to test and observe safe reset and operation.

Verification Summary

This design was validated through a full end-to-end pipeline: accelerated RTL simulation (Icarus/Verilator) and gate-level reset/X-prop checks; MAX10 FPGA prototyping with real accelerometer and FRAM hardware-in-loop; a flight test with recovered logs that exposed and resolved a periodic write bug; Forge silicon bring-up using the MAX10 full-chip emulator and an independent monitor for signal observation; iterative board revisions driven by real hardware behavior; and final OTP, PCB-level, and system-level tests confirming correct sequencing, safety behavior, and deployment timing. The commit history captures each stage of this progression, demonstrating repeated, real-world validation before tapeout

External hardware

Primary hardware: MXC400 accelerometer by I2C bus, a piezo, Dip_sw4. Two connections to a continuity test circuit and 4 deployment connections.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	sw1	cont_enable	sda_accel
1	sw2	speaker	scl_accel
2	sw4	speaker_n	—
3	sw8	deploy	—
4	cont_sense	dump	—
5	—	charge	—
6	—	status_led	—
7	—	—	—

Simon Says memory game

by **Uri Shaked**

0579 | 50 kHz | HDL Project

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

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

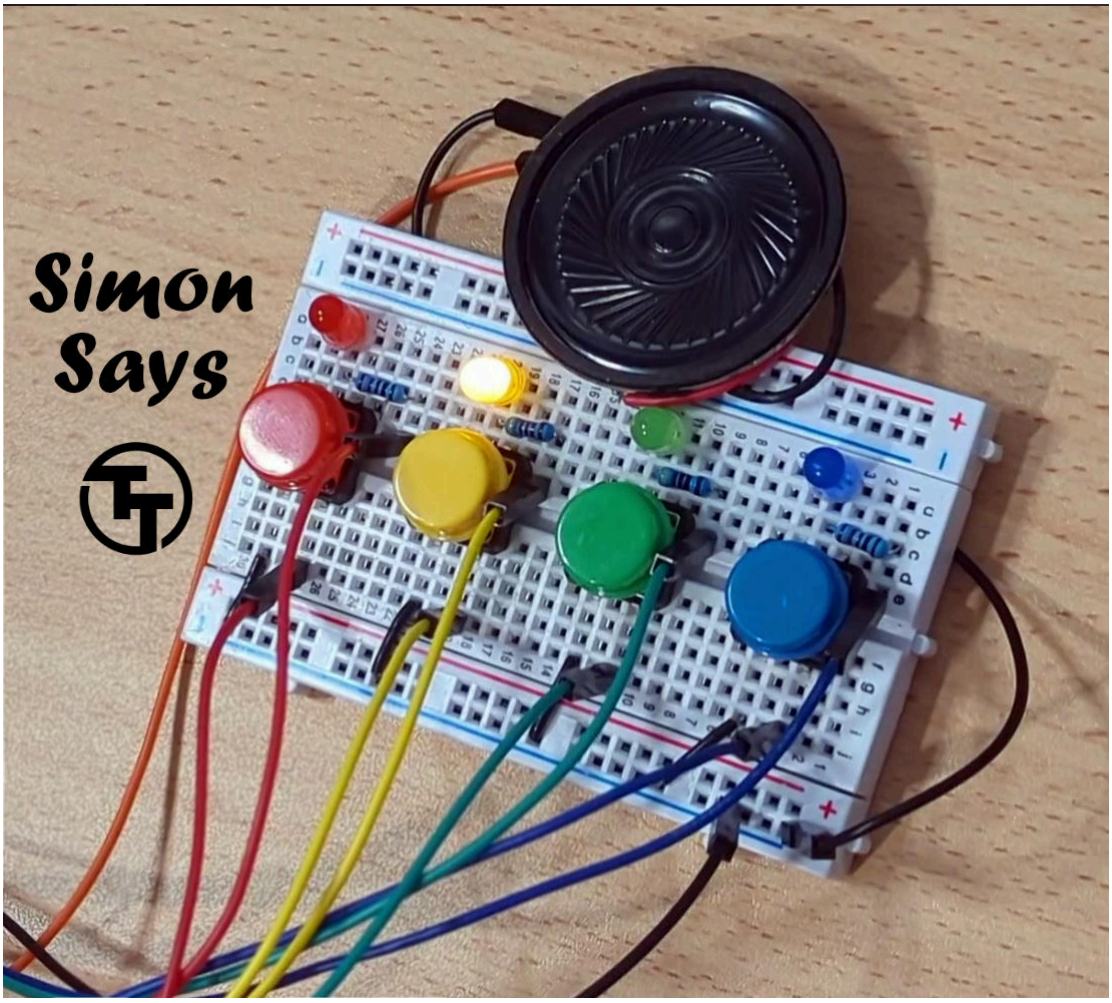


Figure 579.1: Simon Says Game

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, with a frequency of 56 kHz.

The internal clock is generated by a 13-stage ring oscillator, oscillating at 457 MHz, as measured on TTIHP25a silicon. It is then divided by 8192 to land near the 50 kHz of the external-clock case.

The ring oscillator only runs while `clk_sel` is high, so it costs nothing when the external clock is used.

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	—

Kinematic Wave Engine

by **Jeremie Willemin**

`0581` `10 MHz` `HDL Project`

`github.com/Nimelli/ttihp-kinematic-wave-engine`

"8-channel servo control for kinematic sculpture"

How it works

The Kinematic Wave Engine is a dedicated 8-channel servo controller. It autonomously drives an 8-rod kinetic sculpture, generating a continuous travelling sinusoidal wave that (is supposed) to move a ping-pong ball back and forth along its track.

It is an interactive art piece demonstrating parallel hardware timing, real-time procedural animation, and custom silicon design.

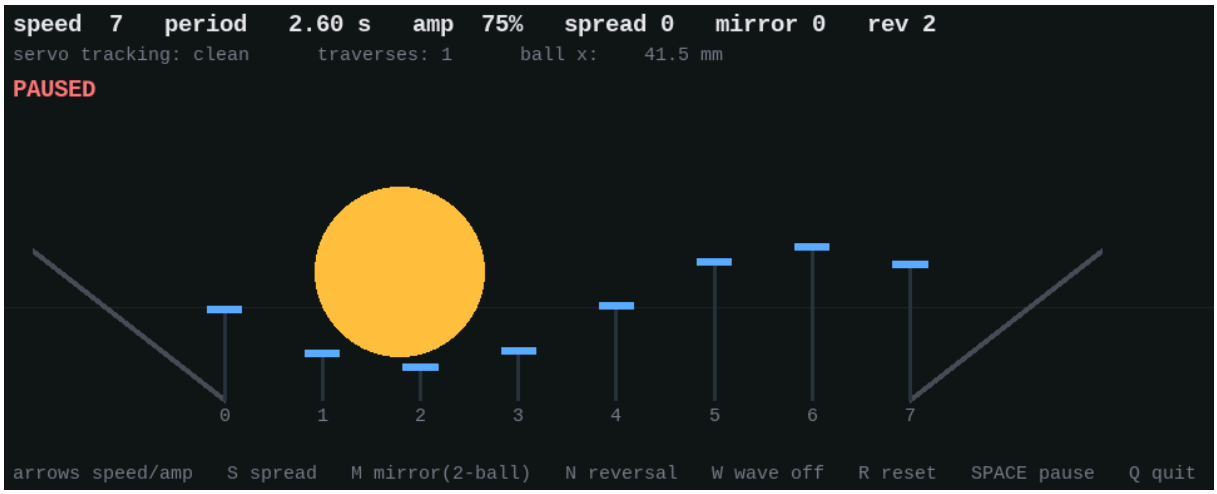


Figure 581.1: alt text

This project was essentially a learning experience for me. It may not be the most useful project, it may not be optimized (or even working at the end), but it's mine and has been a great learning vehicle :)

How to test

Simulation

You can find a Python-based 2D physics simulation in the sim/ folder to better understand about the final goal. The simulation was used to fine-tune default and hardcoded parameters.

FPGA

To increase the confidence even more, the design was tested on an FPGA (CMOD A7 devkit - Artix 7). All looked good, fingers crossed.

For Real

The final kinematic sculpture does not exist yet. I will have the full manufacturing time to develop it !

It basically will be a simple mechanical assembly of 8 rods controlled by 8 servo motors. (see the simulation view) The servos will need an external power source. Each servo motor signal will connect to the 8x outputs. The IC is doing the precise timing control of each servo signal and generates the traveling wave.

Input signals are used to provide some minimal control:

- ui_in[4:1] - speed selection: 16 wave periods, 20.1 s .. 0.26 s
- ui_in[6:5] - amplitude selection: 25 / 50 / 75 / 100 %
- ui_in[7] - spread selection: 0 = full wavelength, 1 = half
- uio_in[4] - mirror: 1 = two-ball mirrored mode
- uio_in[6:5] - reverse selection: reverse after 2 / 3 / 4 / 6 cycles

External hardware

8x servo (like SG90) + kinematic rod sculpture

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	MODE_SW	SERVO_PWM[0]	SPI_CS
1	SPEED[0]	SERVO_PWM[1]	SPI_SCK
2	SPEED[1]	SERVO_PWM[2]	SPI_MOSI
3	SPEED[2]	SERVO_PWM[3]	SPI_MISO
4	SPEED[3]	SERVO_PWM[4]	MIRROR
5	AMP[0]	SERVO_PWM[5]	REVERSE[0]
6	AMP[1]	SERVO_PWM[6]	REVERSE[1]
7	SPREAD	SERVO_PWM[7]	—

project

by **Roaa_Ahmed**

`0583` `100 MHz` `HDL Project`

`github.com/roaa773/UART_TX_TinyTapeout`

“UART_TX”

Block_Diagram

How it works

The UART TX module converts parallel data into a serial UART frame and transmits it through the TX output.

Each frame consists of:

- 1 Start bit
- 8 Data bits
- Optional Parity bit
- 1 Stop bit

The module provides `BUSY` status to indicate that a transmission is in progress.

How to test

Inputs

- `DATA_VALID` – Starts a new transmission.
- `PAR_EN` – Enables/disables the parity bit.
- `PAR_TYP` – Selects the parity type.
- `P_DATA[7:0]` – 8-bit data to be transmitted.

Outputs

- `TX_OUT` – Serial UART output.
- `BUSY` – Indicates that the transmitter is busy.

The design can be tested using the provided testbench and simulated using QuestaSim/Icarus Verilog. ## External hardware

No hardware

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
`0`	0	0	0

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

CRC-8 Serial LFSR

by **Mena Emil**

0585 | 10 MHz | HDL Project

github.com/mena-emil/tt-crc8-lfsr

"8-bit serial LFSR-based CRC generator/checker, seed 8'hD8, LSB-first data shift-in"

How it works

This is an 8-bit **Linear Feedback Shift Register (LFSR)** used as a serial **CRC-8** generator/checker. The register (`R7..R0`) is seeded with `8'hD8` on reset, with two internal XOR feedback taps (between `R7`/`R6` and `R3`/`R2`). The feedback signal is `Feedback = DATA_in ^ R0`.

While `ui_in[1]` (ACTIVE) is high, one bit of `ui_in[0]` (DATA) is shifted in per clock, LSB first, XORed into the feedback path. After 8 clocks (one full byte), the register holds the CRC remainder.

Once ACTIVE drops low, the register shifts itself out serially for 8 clocks — `R0` first — while `uo_out[1]` (Valid) is held high on `uo_out[0]` (CRC). Valid drops low again once all 8 bits have been shifted out.

How to test

The automated Cocotb test is in `test/test.py`. From the `test` directory run:

```
make
```

The test runs the 10 test cases from DATA_h.txt and Expec_Out_h.txt. For each byte, it resets the design, shifts the data byte in LSB-first while holding ACTIVE high, waits for Valid, and checks the 8 output CRC bits against the expected value.

External hardware

None. This project uses only the TinyTapeout dedicated I/O pins (`ui_in[0]`, `ui_in[1]`, `uo_out[0]`, `uo_out[1]`) — no PMOD, display, or other external hardware is required, either in simulation or on the fabricated chip.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	DATA - serial input data bit (LSB first)	CRC - serial CRC output bit	—
1	ACTIVE - high while DATA is being shifted in	Valid - high while CRC bits are being shifted out	—
2	—	—	—
3	—	—	—
4	—	—	—
5	—	—	—
6	—	—	—
7	—	—	—

TinyNPU4

by **Essam Mohammed Elkholy**

0587 | 50 MHz | HDL Project

github.com/Essam-Elkholy/TinyNPU4

"Host-controlled INT4 neural-network accelerator with two parallel MAC lanes"

How it works

TinyNPU4 (also referred to as CatDog TinyNPU) is a host-controlled 64-4-1 quantized neural-network accelerator. The host converts a photo to 8x8 grayscale signed INT4 pixels. Two signed INT4 MAC lanes share each streamed pixel and consume two independent weights, so two hidden neurons are computed in parallel. Each lane has a signed 16-bit saturating accumulator. Hidden results use ReLU; the final result uses Linear activation. `ui_in[0]` selects a lane during bias, finish and debug commands.

After the output neuron, `uo_out[6] = 0` means Cat and `uo_out[6] = 1` means Dog. `uo_out[3:0]` is the saturated signed INT4 score.

How to test

The automated Cocotb test is in `test/test.py`. From the `test` directory run:

```
make
```

The test checks reset, pin directions, a complete two-lane 64-4-1 inference schedule, both MAC lanes, ReLU, Linear saturation, final class sign, independent signed 16-bit saturation boundaries, and full accumulator debug readback.

A separate ModelSim demo (`run_trained_demo.do`, driving `tb_catdog_trained_model.sv`) replays the trained and quantized model against 10 real cat/dog photos directly through the RTL. All 10 were classified correctly (5/5 cats, 5/5 dogs).

Train and quantize the real Cat/Dog model separately with:

```
python -m pip install -r ml/requirements.txt
python ml/train_and_quantize.py
```

Training creates `ml/artifacts/model_int4.json`, containing the weights, biases and shifts streamed to the chip.

External hardware

No external hardware is needed for RTL simulation. A fabricated-chip test needs a Tiny Tapeout demo board and a host computer or microcontroller to drive the input and command pins. No PMOD or display is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	activation[0]_or_lane_select	result[0]	command[0]
1	activation[1]	result[1]	command[1]
2	activation[2]	result[2]	command[2]
3	activation[3]	result[3]	command_valid
4	weight_lane0[0]	done	weight_lane1_or_param[0]
5	weight_lane0[1]	overflow	weight_lane1_or_param[1]
6	weight_lane0[2]	accumulator_negative	weight_lane1_or_param[2]
7	weight_lane0[3]	result_valid	weight_lane1_or_param[3]

Highspeed voltage discriminator

by **Schwallsunk**

0608 HDL Project

github.com/schwallsunk/tt_signal_discriminator

"Used as a digital front end for event counting"

High-Speed Voltage Window Discriminator

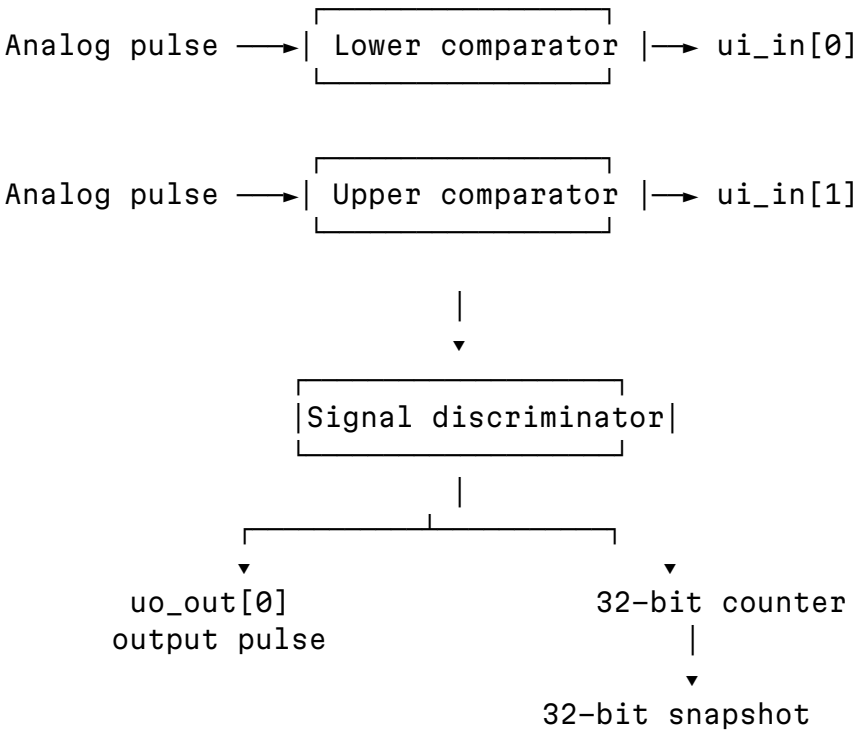
A digital signal discriminator for detecting voltage pulses that fall within a programmable voltage window.

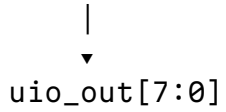
The design accepts two digital signals from external comparators representing a **lower** and **upper** voltage threshold. It determines whether a pulse crosses the lower threshold without subsequently crossing the upper threshold, generates a short digital output pulse, and counts the detected events with a 32-bit counter.

The design is intended for high-speed physical signal processing and experimentation with standard-cell propagation delays.

How it works

The analog signal is **not directly connected to the TinyTapeout tile**. Instead, two external comparators convert the analog input into digital threshold signals:



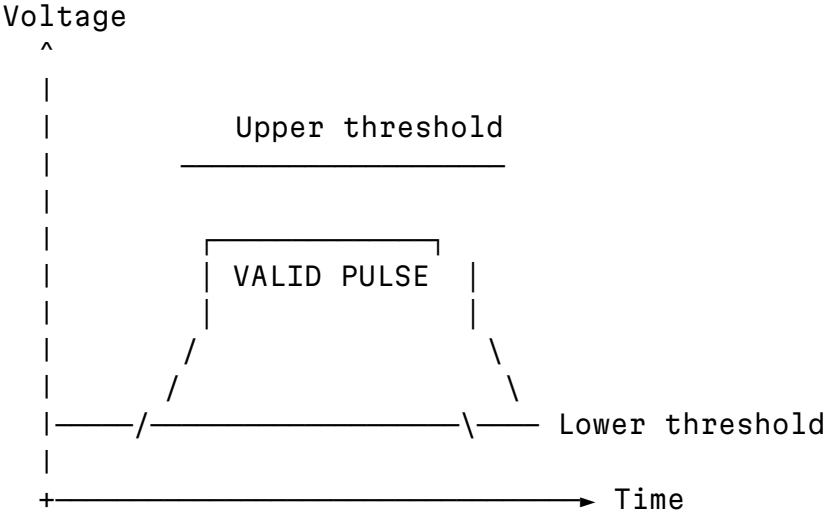


The discriminator uses the relative timing of the two comparator outputs to determine whether the input pulse lies inside the voltage window.

A pulse that crosses the lower threshold but does **not** cross the upper threshold before the lower-threshold condition ends produces a discriminator output event.

A pulse that crosses both thresholds is rejected.

Conceptual voltage window



The actual threshold voltages are established externally by the comparator circuitry.

Discriminator output

The discriminator output is available on:

`uo_out[0]`

The other dedicated outputs are driven low.

The discriminator output is a short digital pulse whose width can be selected using the physical delay chain.

Four delay settings are available:

ui_in[3:2]	Delay chain
00	1 stage
01	3 stages
10	9 stages
11	27 stages

The delay is implemented using `sg13g2_dlygate4sd2_1` standard cells.

This is intentional: the delay is produced by physical standard-cell propagation rather than by RTL simulation delays.

The exact pulse width depends on the fabricated process, supply voltage, temperature, loading, and routing. The values should therefore be considered approximate rather than guaranteed timing specifications.

Event counter

Every accepted discriminator event increments a 32-bit counter.

The counter is implemented as a hybrid architecture:

- The lower 4 bits form a ripple-style prescaler.
- The upper 28 bits are synchronously incremented from the prescaler output.

This reduces the switching frequency of the upper counter stages while providing a full 32-bit event count.

The counter can therefore record:

`0 ... 4,294,967,295`

events before rolling over.

Counter readout

Because the TinyTapeout user IO interface is only 8 bits wide, the 32-bit counter is read in four bytes.

Before reading the counter, its current value is copied into a 32-bit shadow register by applying a rising edge to:

`ui_in[4]`

The shadow register then holds a stable snapshot while the counter continues counting.

This allows the counter to be read without requiring the external system to capture all 32 bits simultaneously.

Byte selection

The byte presented on `uio_out[7:0]` is selected using `ui_in[6:5]`:

`ui_in[6:5]`	Output
`00`	Counter `[7:0]`
`01`	Counter `[15:8]`
`10`	Counter `[23:16]`

11	Counter [31:24]

For example, to read the complete counter:

1. Generate a rising edge on `ui_in[4]`.
2. Set `ui_in[6:5] = 2'b00` and read `uio_out`.
3. Set `ui_in[6:5] = 2'b01` and read `uio_out`.
4. Set `ui_in[6:5] = 2'b10` and read `uio_out`.
5. Set `ui_in[6:5] = 2'b11` and read `uio_out`.

All four bytes belong to the same captured counter value.

Pinout

Dedicated inputs

Pin	Function
`ui_in[0]`	Lower-threshold comparator output
`ui_in[1]`	Upper-threshold comparator output
`ui_in[2]`	Delay selection bit 0
`ui_in[3]`	Delay selection bit 1
`ui_in[4]`	Counter snapshot/latch
`ui_in[5]`	Counter byte select bit 0
`ui_in[6]`	Counter byte select bit 1
`ui_in[7]`	Counter enable
`rst_n`	Active-low global reset

Dedicated outputs

Pin	Function
`uo_out[0]`	Discriminator output
`uo_out[7:1]`	Unused, driven low

User IO

Pin	Function
`uio_out[7:0]`	Selected byte of the counter snapshot
`uio_oe[7:0]`	All user IO pins configured as outputs
`uio_in[7:0]`	Unused

The TinyTapeout `clk` input is not used by the discriminator or counter. The circuit is event-driven by the comparator signals and internal standard-cell logic.

How to test

1. Reset the design

Hold:

`rst_n = 0`

This resets the discriminator state, event counter, and counter snapshot register.

Then set:

`rst_n = 1`

to enable normal operation.

2. Connect the comparators

Connect the output of the lower-threshold comparator to:

`ui_in[0]`

and the output of the upper-threshold comparator to:

`ui_in[1]`

The comparator polarity must match the expected threshold-crossing convention of the discriminator.

3. Generate test pulses

Apply pulses to the external comparator inputs.

A pulse that crosses the lower threshold but remains below the upper threshold should generate an output event on:

`uo_out[0]`

A pulse that also crosses the upper threshold should be rejected.

4. Check the counter

After generating a known number of valid pulses, create a rising edge on:

`ui_in[4]`

This captures the counter.

Then select each byte using:

`ui_in[6:5]`

and read the result on:

`uio_out[7:0]`

For example:

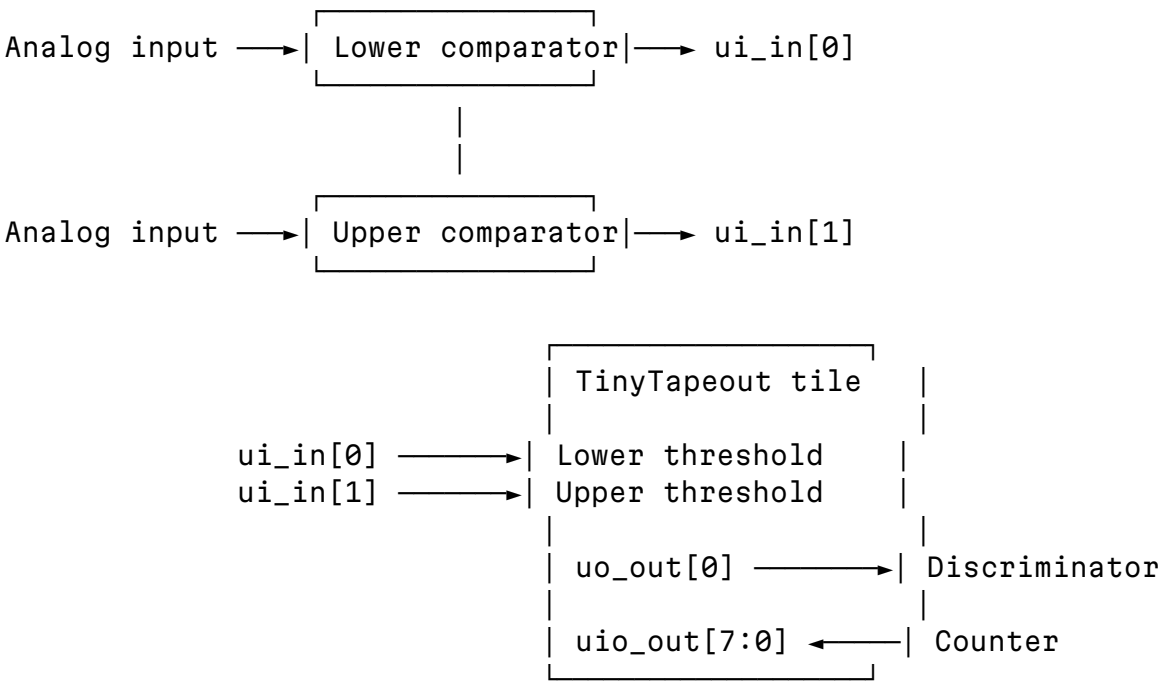
```
ui_in[6:5] = 00  → bits  7:0
ui_in[6:5] = 01  → bits 15:8
```

```
ui_in[6:5] = 10  → bits 23:16
ui_in[6:5] = 11  → bits 31:24
```

External hardware

The TinyTapeout tile expects **digital comparator outputs**, not an analog voltage directly.

For a complete voltage-discrimination setup, external hardware is therefore required:



For high-speed operation, the external comparators and their signal paths should have sufficiently fast propagation time and clean logic transitions.

The delay-chain output is particularly useful for laboratory measurements because it allows the generated discriminator pulse to be stretched sufficiently for observation with conventional high-bandwidth oscilloscopes.

Implementation

The design uses SkyWater/S130 `sg13g2` standard-cell primitives for the timing-critical portions of the circuit, including:

- Delay cells
- Flip-flops
- Buffers
- Logic gates
- Multiplexers

The delay paths are deliberately implemented using physical standard cells rather than behavioral Verilog delay statements.

The project therefore relies on the physical characteristics of the fabricated standard-cell implementation. Measured timing will depend on process, voltage, temperature, routing, fanout, and the external measurement setup.

Project goals

The main goal of this project is to explore high-speed digital pulse discrimination using physically implemented standard-cell delay chains.

Potential applications include:

- Pulse-height discrimination
- Particle detector readout
- Event counting
- Time-domain experiments
- Fast laboratory instrumentation
- Detector signal processing

The design is intentionally simple at the interface: external comparators perform the analog-to-digital threshold conversion, while the TinyTapeout tile performs the high-speed digital discrimination, pulse generation, and event counting.

Discriminators of the fast kind. The faster the better.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	low discriminator threshold	event out	output counter bit 0
1	high discriminator threshold	—	output counter bit 1
2	addr MUX delay 0	—	output counter bit 2
3	addr MUX delay 1	—	output counter bit 3
4	latch to buff	—	output counter bit 4
5	MUX IO to countr 0	—	output counter bit 5
6	MUX IO to countr 1	—	output counter bit 6
7	counter enable	—	output counter bit 7

TinySoC

by **Maadesh**

0609 | 50 MHz | HDL Project

`github.com/Maadeshwar/Maadeshwar_IEEE_EDS_VITC`

"An 8-bit Harvard Architecture microcontroller with UART, GPIO, and PWM"

Overview

TinySoC is a highly optimized, 8-bit Harvard Architecture microcontroller designed to fit securely inside a 1x1 Tiny Tapeout tile (IHP SG13G2 130nm). It executes instructions out of an external ROM or memory emulator connected to the input pins, while handling internal data through its onboard 8-byte RAM and memory-mapped peripheral registers.

Despite its microscopic footprint, the core features a fully functional 3-stage pipeline (Fetch, Fetch Operand, Execute) ensuring stable timing at 50 MHz, and a rich 17-instruction ISA supporting conditional branching, arithmetic flags, and hardware subroutines (CALL/RET).

Architecture Block Diagram

- **CPU Core:** 8-bit accumulator-based state machine.
- **Instruction Memory:** External via `ui[7:0]`.
- **Data Memory:** 8 Bytes Internal RAM.
- **Peripherals:** Memory-mapped UART, PWM, Timer, GPIO.

Pinout Mapping

- `ui[7:0]` - **Instruction Bus** (Input from external ROM)
- `uo[7:0]` - **Program Counter** (Output to ROM address lines)
- `uio[4:0]` - **GPIO Array** (Bidirectional)
- `uio[5]` - **UART TX** (Transmit output)
- `uio[6]` - **UART RX** (Receive input)
- `uio[7]` - **PWM OUT** (Pulse Width Modulation output)

Memory Map

Memory space is divided between Internal RAM and Peripherals.

Address	Description	Type
`0x00` - `0x07`	Internal Scratchpad RAM (8 Bytes)	R/W
`0x20`	GPIO Data (Read gets Input, Write sets Output)	R/W

0x21	GPIO Direction Register (1 = Output, 0 = Input)	R/W
0x22	Hardware Timer (Read: Current value, Write: Reset to 0)	R/W
0x23	UART RX Data (Read gets data, automatically clears ready flag)	R
0x24	UART TX Data (Writing initiates transmission)	W
0x25	UART Status (Bit 0: RX Ready, Bit 1: TX Busy)	R
0x26	PWM Duty Cycle	R/W
0x28	UART Baud Divider Low Byte (BAUD_DIV_L)	R/W
0x29	UART Baud Divider High Byte (BAUD_DIV_H)	R/W

Instruction Set Architecture (ISA)

All instructions are fixed 16-bit words (Opcode + Operand).

Opcode	Mnemonic	Description	Flags Affected
0x01	LDI val	Load Immediate val into Accumulator (ACC)	Z
0x02	LDR addr	Load ACC from Memory Address addr	Z
0x03	STR addr	Store ACC to Memory Address addr	None
0x04	ADD addr	Add memory at addr to ACC	Z, C
0x05	SUB addr	Subtract memory at addr from ACC	Z, C
0x06	JMP addr	Jump unconditionally to addr	None
0x07	JZ addr	Jump to addr if Zero Flag (Z) is 1	None
0x08	AND addr	Bitwise AND memory at addr with ACC	Z
0x09	OR addr	Bitwise OR memory at addr with ACC	Z
0x0A	XOR addr	Bitwise XOR memory at addr with ACC	Z
0x0B	SHL addr	Logical Shift Left ACC by memory at addr	Z
0x0C	SHR addr	Logical Shift Right ACC by memory at addr	Z

0x0D	CALL addr	Push return address and Jump to addr	None
0x0E	RET	Return from subroutine	None
0x0F	JNZ addr	Jump to addr if Zero Flag (Z) is 0	None
0x10	JC addr	Jump to addr if Carry Flag (C) is 1	None
0xFF	NOP	No Operation (Any unmapped opcode is a NOP)	None

Peripheral Usage Guides

UART (Dynamic Baud Rate)

The UART features a dynamic 16-bit baud rate divider mapped to 0x28 and 0x29. The required divider value is Clock_Frequency / Target_Baud_Rate. It defaults to 434 on reset, which provides 115200 baud at 50 MHz.

- **Transmit:** Poll 0x25 until Bit 1 is 0 (not busy). Write data to 0x24.
- **Receive:** Poll 0x25 until Bit 0 is 1 (data ready). Read 0x23 to get the byte (this automatically clears the ready flag).

PWM

Write any 8-bit value to 0x26 to set the duty cycle of the PWM OUT pin. 0x00 is fully off, 0xFF is fully on (100% duty cycle). The PWM timer runs independently of the CPU.

Tooling

A custom Python assembler is provided in the software/ directory of the repository. It converts custom mnemonics and resolves jump labels into C-compatible hex arrays for immediate deployment onto microcontroller ROM emulators (like the RP2040 on the TT Demo Board).

See software/demo.asm for an example application.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	Instruction Bus IN 0	Program Counter Address OUT 0	GPIO 0
1	Instruction Bus IN 1	Program Counter Address OUT 1	GPIO 1
2	Instruction Bus IN 2	Program Counter Address OUT 2	GPIO 2
3	Instruction Bus IN 3	Program Counter Address OUT 3	GPIO 3
4	Instruction Bus IN 4	Program Counter Address OUT 4	GPIO 4
5	Instruction Bus IN 5	Program Counter Address OUT 5	UART TX

#	Input	Output	Bidirectional
6	Instruction Bus IN 6	Program Counter Address OUT 6	UART RX
7	Instruction Bus IN 7	Program Counter Address OUT 7	PWM OUT

8-bit educational SAP-style CPU

by **University of La Frontera**

`0610` `HDL Project`

`github.com/tiarinix/8bit-CPU`

"A small 8-bit CPU (5 general-purpose registers, ALU, conditional jumps) with a shared bus and byte-programmable memory"

How it works

An 8-bit SAP-style (Simple-As-Possible) CPU: one shared 8-bit bus, 5 general-purpose registers (A–E), an 8-operation ALU, conditional jumps, and 14 bytes of byte-addressable program/data memory (Von Neumann — program and data share the same address space).

Every instruction executes in up to 4 cycles:

1. **FETCH1** — reads the opcode from memory at the current program counter, and advances the program counter.
2. **FETCH2** — only for 2-byte instructions (loads, stores, jumps): reads the operand byte.
3. **EXECUTE** — decodes the opcode and drives the ALU or control signals it needs.
4. **STORE** — writes the result to memory or to a register, if the opcode produces one.

The instruction set covers memory load/store, all 8 ALU operations against the accumulator (register A), 7 conditional jump variants (zero/carry/negative/overflow and their inverses), input/output through the chip's pins, and halt. See the full opcode table in the repository README.

How to test

1. Hold `rst_n` low for at least 2 clock cycles, then release it.
2. **Flash a program:** set `uio_in[0]` high (`program_mode`), then for each byte of the program, set `ui_in` to that byte's value and pulse `uio_in[1]` (`program_wr`) for one clock cycle.
3. Drop `program_mode` low and pulse `rst_n` again so the program counter returns to `0` before execution starts.
4. Watch `uo_out` for results written by an `OUT` instruction, or drive `ui_in` with the value an upcoming `IN` instruction should capture.

A minimal example program (`LD A`, `LD B`, `ADD B`, `OUT A`, `HLT`) loads two values from memory, adds them, and outputs the sum. A working, fully-worked

example driving these exact steps through cocotb is in `test/test.py` in this repository, including load/store, conditional jumps, halt, and external I/O.

External hardware

None required. `ui[0:7]` and `uo[0:7]` are general-purpose 8-bit input/output buses driven by the `IN`/`OUT` instructions — connect switches, LEDs, or any other digital I/O you like.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	external_input[0]	external_output[0]	program_mode
1	external_input[1]	external_output[1]	program_wr
2	external_input[2]	external_output[2]	—
3	external_input[3]	external_output[3]	—
4	external_input[4]	external_output[4]	—
5	external_input[5]	external_output[5]	—
6	external_input[6]	external_output[6]	—
7	external_input[7]	external_output[7]	—

Barrel Shifter

by **Mahmoud Magdy**

`0611` `HDL Project`

`github.com/mahmoudmq/ieee-barrel-shifter`

"*Barrel Shifter handles 1 bit, 2 bit, and 4 bit shifts*"

How it works

A barrel shifter shifting input but 1 bit, 2 bit, or 4 bit based on the control signal. ## How to test Input Bits and a control signal and you should notice the output shifts by a certain bits based on the control signal.

External hardware

You may use LEDs for visualization.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	b0	b7	—
1	b1	b8	—
2	b2	b9	—
3	b3	b10	—
4	b4	b11	—
5	b5	b12	—
6	b5	b13	—
7	b6	b14	—

BOOTCAMP

by **ELI**

`0612` `25.175 MHz` `HDL Project`

`github.com/eleazarmejos04-code/HATSUNE-MIKU-BOOTCAMP`

"HATSUNE MIKU"

How it works
This project implements a hardware-based VGA pattern generator that renders an animated Hatsune Miku visual motif at standard 640×480 @ 60 Hz resolution. The design operates purely in combinational and sequential digital logic without an external framebuffer, microprocessor, or video RAM.

Scanline & Timing Engine: Generates active-low horizontal sync (hsync) and vertical sync (vsync) pulses alongside horizontal (hpos, 0–799) and vertical (vpos, 0–524) pixel counters driven by a 25.175 MHz pixel clock.

Character & Silhouette Renderer: Uses geometric bounding-box and coordinate decoders to draw Hatsune Miku's iconic aesthetic—featuring her signature twin-tails in teal/cyan (#39C5BB), dark gray/black silhouette framing, and magenta/pink hair-tie and accessory accents.

Animation & Rhythm Control: Utilizes a vertical sync frame strobe to cycle dance movements, swaying twin-tails, and bounce offsets synchronously with the display refresh rate.

Blanking Guard: Forces all color signals to black during horizontal and vertical blanking/porch intervals to maintain standard VESA voltage compliance.

Output Mapping: Formats the video stream into standard 6-bit color (2-bit Red, 2-bit Green, 2-bit Blue) along with sync pulses configured for the standard TinyVGA PMOD pinout.

How to test

1. Browser Simulation (VGA Playground) You can test the design directly in your browser without hardware:

Navigate to https://vga-playground.com/?repo=https://github.com//

VGA Playground compiles the Verilog code using Verilator in WebAssembly and displays the real-time 60 Hz monitor output.

2. Automated Simulation (cocotb) Run the test suite locally to verify pixel timing, reset behavior, and sync pulse polarity:

Bash cd test make To run gate-level netlist simulation against the synthesized standard cells:

Bash make GATES=yes

3. Silicon / Carrier Board Setup Plug a standard TinyVGA PMOD into the dedicated output header (uo_out).

Connect a VGA cable from the PMOD to a 640×480 @ 60 Hz capable monitor.

Supply a 25.175 MHz clock to the clk pin (25.0 MHz is also supported by most modern monitors).

Assert active-low reset (rst_n = 0), then release (rst_n = 1) to start the frame counters.

Select this design's address on the carrier board DIP switches to display Hatsune Miku on screen.

External hardware TinyVGA PMOD: 6-bit R2G2B2 resistor-ladder DAC providing analog RGB, HSYNC, and VSYNC outputs.

VGA Monitor: Standard display supporting 640×480 @ 60 Hz (31.468 kHz horizontal frequency).

VGA Cable: Standard 15-pin D-sub male-to-male cable.

Clock Source: 25.175 MHz oscillator or carrier board clock generator.

Project Pinout

Digital Pins

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

Approximate DSP: Time-Multiplexed MAC Coprocessor

by **praneel**

`0613` `50 MHz` `HDL Project`

`github.com/Praneel-1G/Praneel_IEEE_EDS_VITC`

"An 8-bit SPI-programmable approximate DSP block featuring a 4-tap time-multiplexed approximate MAC for Smart Dust Edge AI."

The Approximate MAC Coprocessor is a physical-design-focused Tiny Tape-out submission that explores area and power reduction for "Smart Dust" Edge AI applications. It implements a 4-tap Finite Impulse Response (FIR) DSP engine controlled via a lightweight SPI bus.

To maximize silicon area efficiency on the SkyWater 130nm node, this design time-multiplexes a single **Approximate Multiply-Accumulate (MAC) unit**. By structurally removing the 4x4 LSB multiplication array, the design intentionally trades a mathematically acceptable amount of precision for a massive reduction in dynamic power and gate count.

The Architecture

1. **The SPI Bus:** Communication is handled via a standard 8-bit SPI Mode 0 interface (`SCLK`, `MOSI`, `MISO`, `CS_N`), making it compatible with almost any microcontroller.
2. **Coefficient Memory:** An internal 4-byte register file stores the filter taps.
3. **Circular Delay Line:** A 3-stage data delay line holds incoming streaming sensor samples.
4. **The Approximate Datapath:** A state machine cycles through the 4 taps (current sample + 3 delayed samples), multiplies them against the stored coefficients using the approximate math core, and accumulates the 16-bit result. The top 8 bits are then shifted out over MISO.

Host Framing Protocol

The SPI interface uses a simple command-byte framing structure. Pull `CS_N` low to initiate a transaction.

1. Load Coefficients (`0x01`)

Send the `0x01` command followed by four 8-bit coefficients. The FSM will store them in taps C0 through C3 and return to the IDLE state.

- **MOSI:** `[0x01] -> [C0] -> [C1] -> [C2] -> [C3]`

2. Stream Data (`0x02`)

Send the `0x02` command. Keep `CS_N` low and stream continuous 8-bit data samples. For every byte sent, the hardware computes the 4-tap accumulation and shifts out the upper 8 bits of the *previous* computation cycle.

- **MOSI:** `[0x02] -> [Data0] -> [Data1] -> [Data2] ...`
- **MISO:** `[ junk ] -> [ junk  ] -> [Out 0] -> [Out 1] ...`

The Approximation Mathematics

A standard 8x8 multiplication generates a 16-bit product. To save gates, this design splits the 8-bit inputs into 4-bit nibbles (High and Low) and omits the Low-Low multiplication entirely:

`Exact: (A_hi * B_hi)<<8 + (A_hi * B_lo)<<4 + (A_lo * B_hi)<<4 + (A_lo * B_lo)` `Approximate: (A_hi * B_hi)<<8 + (A_hi * B_lo)<<4 + (A_lo * B_hi)<<4`

This results in a slight underestimation of the true product, which acts as acceptable noise in signal filtering applications (like IIR/FIR sensor smoothing) while drastically shrinking the physical footprint of the multiplier logic.

Testing & Verification

See the repository's `tb.py` file for a full Cocotb test suite. The testbench automatically issues SPI transactions to load coefficients, streams data, and asserts that the RTL output matches a software-calculated mathematical approximation model.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	SCLK (SPI Clock)	MISO (SPI Data Out)	unused
1	MOSI (SPI Data In)	DEBUG_STATE_LOAD	unused
2	CS_N (SPI Chip Select)	DEBUG_STATE_STREAM	unused
3	unused	unused	unused
4	unused	unused	unused
5	unused	unused	unused
6	unused	unused	unused
7	unused	unused	unused

Tic Tac Toe

by **Hanz Baga** & **John Edward Macaraeg**

0614 | 25.175 MHz | HDL Project

github.com/bagaHU/tt_um_vga_tictactoe

> *"An interactive 3×3 Tic-Tac-Toe game on VGA, showcasing real-time gameplay, cursor control, turn-based play, and automatic win detection."*

How it works

The project implements a VGA Tic-Tac-Toe game using Verilog. It displays a 3×3 game board on a VGA screen, allowing two players to take turns placing X and O marks. The game detects winning combinations and displays the winner or a draw.

Link to VGA Playground: https://vga-playground.com/?repo=https://github.com/bagaHU/tt_um_vga_tictactoe

How to test

Connect the FPGA to a VGA display and keyboard, then program the FPGA with the project. Use the keyboard to control the cursor and play the game.

Controls [1] UP [2] DOWN [3] LEFT [4] RIGHT [5] ATTACK / PLACE MARK [7] NEW GAME

External hardware

- FPGA development board
- VGA display
- USB keyboard

Project Pinout

Digital Pins

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

#	Input	Output	Bidirectional
6	—	B0	—
7	—	HSync	—

Market-split oracle

by **joesagents**

`0615` `50 MHz` `HDL Project`

`github.com/joesagents/ttihp26b-market-split-oracle`

> "*Checks candidate assignments for a fixed market-split instance*"

How it works

Accumulates row sums as the four input bytes arrive, then compares them with the targets in `test/instance.json`.

How to test

Write `0x69, 0xc1, 0x3d, 0x34` with SEL = 0, 1, 2, 3, WE high for one clock each. DONE and EXACT_OK go high on the fourth write. An out-of-order write voids the load; restart with SEL = 0.

External hardware

None.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	X0	SYN0	SEL0
1	X1	SYN1	SEL1
2	X2	SYN2	WE
3	X3	SYN3	—
4	X4	PAR_OK	—
5	X5	DONE	—
6	X6	EXACT_OK	—
7	X7	—	—

DVD player

by **Gian Mustar**

`0616` `25.175 MHz` `HDL Project`

`github.com/giannn13/CDM-Bootcamp-DVD-player-2026`

“dvd”

How it works

DVD PLAYER

How to test

Screen saver

External hardware

Display

Project Pinout

Digital Pins

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

Simple comparator + 2 DACs analog layout in a 1x1 tile

by **algofoogle (Anton Maurovic)**

0617 HDL Project

`github.com/algofoogle/ttihp26b-analog-junk`

"IHP sg13g2 custom layout analog experiments in a regular TT 1x1 digital tile"

How it works

A custom GDS layout implements some simple analog circuits with an interface to the Tiny Tapeout digital pins of a 1x1 tile. In particular `ui_in` goes to an 8-bit RDAC, as does `uio_in`, and together they form the positive and negative inputs to a comparator whose output is on `uo_out[7]`.

How to test

Try comparing values on `ui_in` with those on `uio_in` – if `ui_in` is greater, `uo_out[7]` should go high. Note that the comparator is generally considered to be reliable for voltages in the range of 0.3V to 0.9V internally – DAC codes 64..192.

External hardware

You'll need the Tiny Tapeout demo board to assert values on the bidirectional pins, but other than that no external hardware is required.

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	pdac[0]	—	ndac[0]
1	pdac[1]	—	ndac[1]
2	pdac[2]	—	ndac[2]
3	pdac[3]	—	ndac[3]
4	pdac[4]	—	ndac[4]
5	pdac[5]	—	ndac[5]
6	pdac[6]	—	ndac[6]
7	pdac[7]	comparator_out	ndac[7]

KATSEYE

by **CLEA**

`0618` `25.175 MHz` `HDL Project`

`github.com/cleajane/katseye-bootcamp`

"Kawaii symbol rain that reveals KATSEYE RULES in the center"

How it works

A VGA demo with a "kawaii symbol rain" effect. Columns of 8x8 pixel glyphs (hearts, stars, sparkles, flowers, cats, music notes, smileys and moons) fall down the screen at two different speeds, leaving fading trails behind a bright white head.

After a short delay, a reveal counter starts uncovering the message "KATSEYE RULES" in the center of the screen, one letter at a time. Each new letter flashes white, then settles into a shimmering pastel color. A framed plate with a color-cycling border appears behind the text.

There are four pastel palettes: sakura pink, mint soda, lavender dream, and peach cream. The design uses 2 bits per color channel (6-bit color) and outputs standard 640x480 VGA at 25.175 MHz.

How to test

Connect a TinyVGA PMOD to the output pins and plug it into a VGA monitor. There is no other setup needed.

- `ui[1:0]` selects the color palette (00 = sakura pink, 01 = mint soda, 10 = lavender dream, 11 = peach cream).
- `ui[7:6]` selects the VGA mode. Leave both low for the default.

External hardware

- TinyVGA PMOD
- VGA monitor

Project Pinout

Digital Pins

#	Input	Output	Bidirectional
0	PAL0 (palette select bit 0)	R1	—
1	PAL1 (palette select bit 1)	G1	—
2	—	B1	—

#	Input	Output	Bidirectional
3	—	VSYNC	—
4	—	R0	—
5	—	G0	—
6	MODE0 (VGA mode bit 0)	B0	—
7	MODE1 (VGA mode bit 1)	HSYNC	—

4x4NPU: Dual-Lane INT4 Neural Accelerator

by **Maadeshwar**

0619 | 50 MHz | HDL Project

`github.com/Maadeshwar/Maadeshwar_4x4NPU_IEEE_EDS_VITC`

> *"Host-controlled IHP SG13G2 INT4 neural accelerator with two parallel MAC lanes, saturation, ReLU, and accumulator readback."*

4x4NPU is a host-controlled two-lane signed INT4 neural datapath targeting a single IHP SG13G2 Tiny Tapeout tile. It uses a three-process controller FSM:

1. `IDLE` captures one command and its operands.
2. `EXEC` performs the command and updates the datapath.
3. `DONE` returns the controller to `IDLE` and provides a clean command-cycle boundary.

Datapath

`ui_in[3:0]` is a signed INT4 activation. `ui_in[7:4]` is lane-0's signed INT4 weight. `uio_in[7:4]` is lane-1's signed INT4 weight during `MAC`. Each accepted MAC command updates both independent signed 16-bit accumulators. Products are exact signed INT4 products and accumulators saturate at −32768 and +32767. Overflow is sticky until `CLEAR` or reset.

Commands

`uio_in[2:0]` selects the command and `uio_in[3]` qualifies it:

Command	Name	Function
000	NOP	No operation
001	CLEAR	Clear biases, accumulators, result, and overflow
010	BIAS_LOW	Load the low bias nibble for the lane selected by `ui_in[0]`
011	BIAS_HIGH	Combine the high nibble with the stored low nibble as signed INT8 bias
100	MAC	Accumulate both lane products
101	FINISH_RELU	ReLU, arithmetic shift, and signed INT4 saturation

110	FINISH_LINEAR	Arithmetic shift and signed INT4 saturation
111	READ_ACC	Read one accumulator nibble selected by `uio_in[5:4]`

The parameter field is `uio_in[7:4]`. For `FINISH_*`, it is the arithmetic right-shift amount. For `READ_ACC`, its low two bits select accumulator nibble 0 through 3. For bias commands, it carries the bias nibble.

Outputs

`uo_out[3:0]` is the quantized result or accumulator nibble. `uo_out[4]` is a one-cycle `DONE` indication for finish/read commands. `uo_out[5]` is sticky overflow, `uo_out[6]` is the selected accumulator sign, and `uo_out[7]` is `RESULT_VALID`.

All bidirectional pads are input-only (`uio_oe = 0`) to avoid contention with the host board.

Applications

The block can execute quantized neural layers, two-neuron perceptrons, small convolution/dot-product kernels, sensor-feature classifiers, and compact DSP filters. Training and quantization remain off-chip; the fabricated chip performs the deterministic INT4 inference arithmetic.

Verification boundary

The project includes Cocotb RTL tests and Tiny Tapeout IHP130 GDS/ precheck/ gate-level workflows. Final silicon claims require all CI jobs, STA, DRC, LVS, and post-fabrication measurements to pass.

Project Pinout

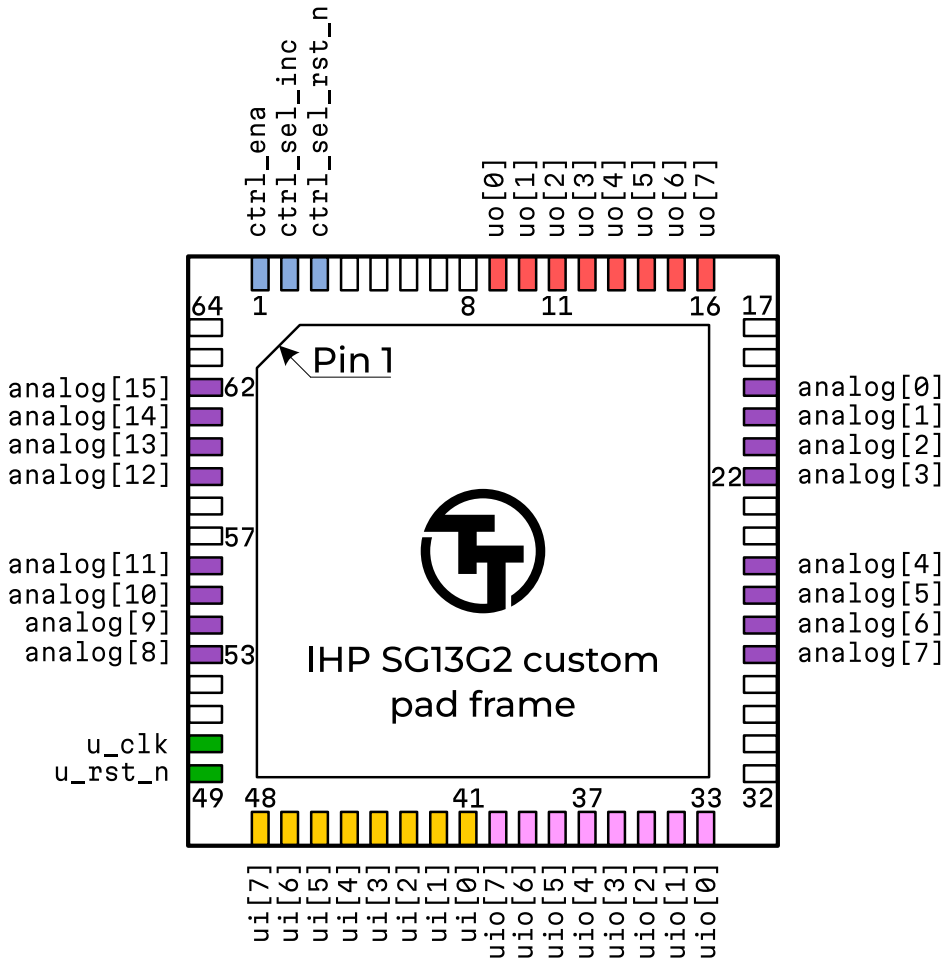
Digital Pins

#	Input	Output	Bidirectional
0	activation[0] / lane select	INT4 result bit 0	command bit 0
1	activation[1]	INT4 result bit 1	command bit 1
2	activation[2]	INT4 result bit 2	command bit 2
3	activation[3]	INT4 result bit 3	command valid
4	weight lane 0 bit 0	DONE	weight lane 1 / parameter bit 0
5	weight lane 0 bit 1	OVERFLOW	weight lane 1 / parameter bit 1

#	Input	Output	Bidirectional
6	weight lane 0 bit 2	accumulator negative	weight lane 1 / parameter bit 2
7	weight lane 0 bit 3	RESULT_VALID	weight lane 1 / parameter bit 3

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:

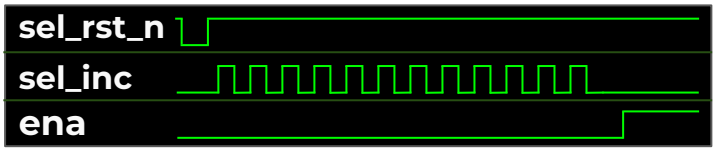


Figure 1: Mux signals for activating the design at address 12

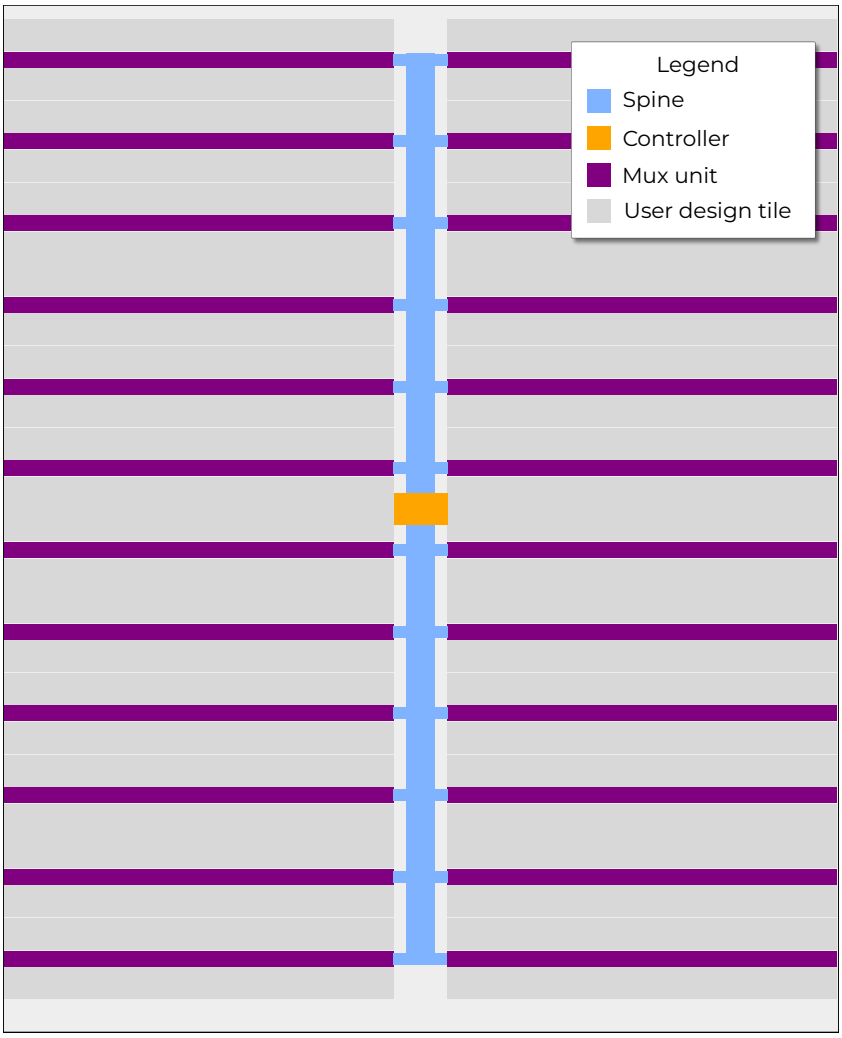


Figure 2: Mux Diagram

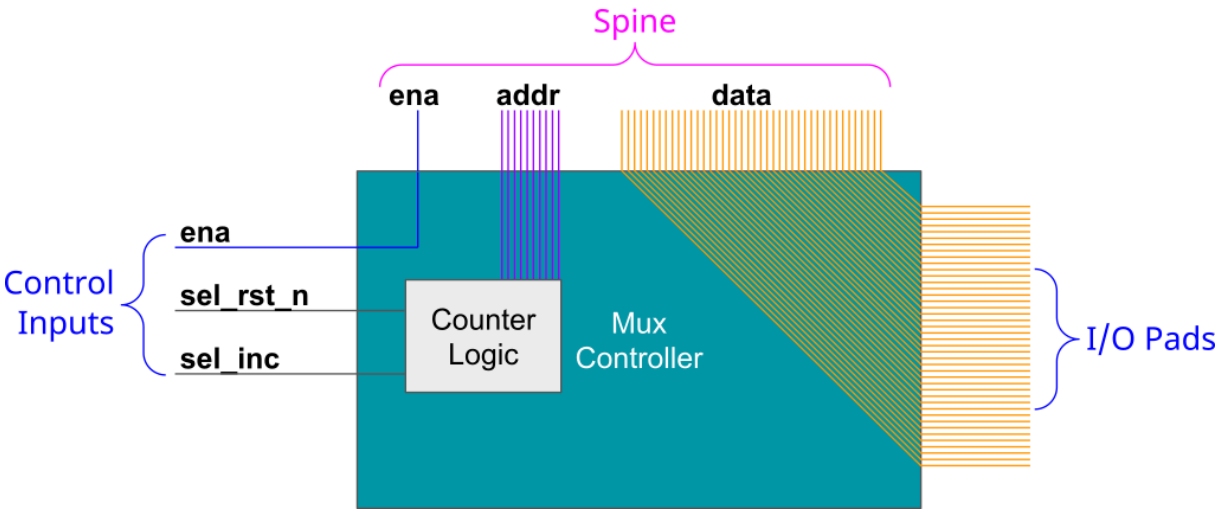


Figure 3: Mux Controller Diagram

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	9	Output	uo[0]
9	10	Output	uo[1]
10	11	Output	uo[2]
11	12	Output	uo[3]
12	13	Output	uo[4]
13	14	Output	uo[5]
14	15	Output	uo[6]
15	16	Output	uo[7]
16	17	Power	VDD IO
17	18	Ground	GND IO
18	19	Analog	analog[0]
19	20	Analog	analog[1]
20	21	Analog	analog[2]
21	22	Analog	analog[3]
22	23	Power	VDD Analog
23	24	Ground	GND Analog
24	25	Analog	analog[4]
25	26	Analog	analog[5]
26	27	Analog	analog[6]
27	28	Analog	analog[7]
28	29	Ground	GND Core
29	30	Power	VDD Core
30	31	Ground	GND IO
31	32	Power	VDD IO
32	33	Bidirectional	uio[0]
33	34	Bidirectional	uio[1]
34	35	Bidirectional	uio[2]

Die Pad	QFN64 pin	Function	Signal
35	36	Bidirectional	uio[3]
36	37	Bidirectional	uio[4]
37	38	Bidirectional	uio[5]
38	39	Bidirectional	uio[6]
39	40	Bidirectional	uio[7]
40	41	Input	ui[0]
41	42	Input	ui[1]
42	43	Input	ui[2]
43	44	Input	ui[3]
44	45	Input	ui[4]
45	46	Input	ui[5]
46	47	Input	ui[6]
47	48	Input	ui[7]
48	49	Input	u_rst_n †
49	50	Input	u_clk †
50	51	Ground	GND IO
51	52	Power	VDD IO
52	53	Analog	analog[8]
53	54	Analog	analog[9]
54	55	Analog	analog[10]
55	56	Analog	analog[11]
56	57	Ground	GND Analog
57	58	Power	VDD Analog
58	59	Analog	analog[12]
59	60	Analog	analog[13]
60	61	Analog	analog[14]
61	62	Analog	analog[15]
62	63	Ground	GND Core
63	64	Power	VDD Core
SUB	EPAD	Ground	—

† 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.

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